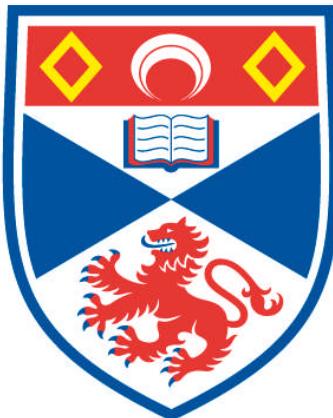


**ON THE CONSTRUCTION OF DECENTRALISED
SERVICE-ORIENTED ORCHESTRATION SYSTEMS**

Ward Jaradat

**A Thesis Submitted for the Degree of PhD
at the
University of St Andrews**



2016

**Full metadata for this item is available in
Research@StAndrews:FullText
at:**

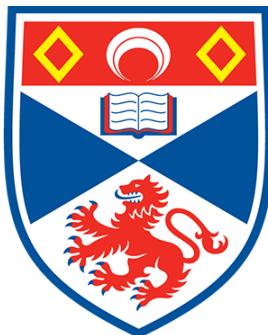
<http://research-repository.st-andrews.ac.uk/>

**Please use this identifier to cite or link to this item:
<http://hdl.handle.net/10023/8036>**

This item is protected by original copyright

On the Construction of Decentralised Service-oriented Orchestration Systems

Ward Jaradat



University of
St Andrews

This thesis is submitted in partial fulfilment for the degree of

Doctor of Philosophy

May 2015

Abstract

Modern science relies on workflow technology to capture, process, and analyse data obtained from scientific instruments. Scientific workflows are precise descriptions of experiments in which multiple computational tasks are coordinated based on the dataflows between them. Orchestrating scientific workflows presents a significant research challenge: they are typically executed in a manner such that all data pass through a centralised compute server known as the engine, which causes unnecessary network traffic that leads to a performance bottleneck. These workflows are commonly composed of services that perform computation over geographically distributed resources, and involve the management of dataflows between them. Centralised orchestration is clearly not a scalable approach for coordinating services dispersed across distant geographical locations. This thesis presents a scalable decentralised service-oriented orchestration system that relies on a high-level data coordination language for the specification and execution of workflows. This system's architecture consists of distributed engines, each of which is responsible for executing part of the overall workflow. It exploits parallelism in the workflow by decomposing it into smaller sub workflows, and determines the most appropriate engines to execute them using computation placement analysis. This permits the workflow logic to be distributed closer to the services providing the data for execution, which reduces the overall data transfer in the workflow and improves its execution time. This thesis provides an evaluation of the presented system which concludes that decentralised orchestration provides scalability benefits over centralised orchestration, and improves the overall performance of executing a service-oriented workflow.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Adam Barker, for his constant patience and continuous guidance throughout my doctoral studies. I am also especially indebted to Prof. Alan Dearle, who taught me a great deal about being a true computer scientist. This thesis would not have seen the light without their invaluable advice and excellent knowledge.

During my time at the School of Computer Science, I have had the pleasure to work alongside remarkable academics in a challenging research environment. Thanks to all the people who inspired, supported, and encouraged me in St Andrews, the beautiful town I have considered my home for the past few years. These years have been profoundly rewarding and have changed me, hopefully, to a better person.

Of course, I am deeply grateful to my family. I would not be the person I am without the love of my parents, my greatest teachers. They deserve more respect and praise than I can express in a few short words. Special thanks to Shaden, my brother and source of inspiration, for his guidance and solid support through difficult times.

Declarations

I, Ward Jaradat, hereby certify that this thesis, which is approximately 61,500 words in length, has been written by me, and that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree. I was admitted as a research student and as a candidate for the degree of Doctor of Philosophy in May 2011; the higher study for which this is a record was carried out in the University of St Andrews between 2011 and 2015.

Date: _____ Signature of candidate: _____

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date: _____ Signature of supervisor: _____

Permission for Electronic Publication

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis: *Access to printed and electronic publication of this thesis through the University of St Andrews.*

Date: _____ Signature of candidate: _____

Date: _____ Signature of supervisor: _____

Publications

Some of the ideas presented in this thesis have been published in academic journal, conference and workshop articles. These articles are listed as follows:

- Ward Jaradat, Alan Dearle, and Adam Barker. Towards an autonomous decentralised orchestration system. *Concurrency and Computation: Practice and Experience* (2015).
- Ward Jaradat, Alan Dearle and Adam Barker. Workflow Partitioning and Deployment on the Cloud using Orchestra. *In Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing* (UCC 2014), pages 251-260, IEEE Computer Society, 2014.
- Ward Jaradat, Alan Dearle and Adam Barker. A Dataflow Language for Decentralised Orchestration of Web Service Workflows. *In Proceedings of the IEEE Ninth World Congress in Services* (SERVICES 2013), pages 13-20, IEEE Computer Society, June 2013.
- Ward Jaradat, Alan Dearle and Adam Barker. An Architecture for Decentralised Orchestration of Web Service Workflows. *In Proceedings of IEEE 20th International Conference on Web Services* (ICWS 2013), pages 603-604, IEEE Computer Society, June 2013.

To my beloved parents.

Table of Contents

Abstract	iii
Acknowledgements	v
Declarations	vii
Permission for Electronic Publication	ix
Publications	xi
Table of Contents	xv
List of Figures	xxi
List of Tables	xxv
1 Introduction	1
1.1 Research Statement	10
1.2 Research Challenges	11
1.2.1 Design Challenges	11
1.2.2 Decentralised Service Orchestration Challenges	12
1.2.3 Global Computation Challenges	12
1.3 Research Hypothesis	14
1.4 Research Contributions	15
1.4.1 Primary Contributions	15
1.4.2 Secondary Contributions	16
1.4.3 Tertiary Contributions	17
1.5 Preview of Research Solution	17
1.6 Glossary of Terms	20
1.7 Thesis Organisation	22

1.8 Conclusion	23
2 Literature Review	25
2.1 Introduction	25
2.2 Service Computing	26
2.2.1 Web Services	28
2.2.2 Web Service Model and Related Standards	33
2.2.3 Central Limitations of the Web Service Model	37
2.2.4 Web Service Specifications	38
2.3 Workflows	39
2.3.1 Scientific Workflows	40
2.3.2 Design Patterns of Scientific Workflows	41
2.3.3 Lifecycle of Scientific Workflows	44
2.3.4 Workflow Management Systems	48
2.4 Service Orchestration	50
2.4.1 Centralised Service Orchestration System Architectures	50
2.4.2 Decentralised Service Orchestration System Architecture	53
2.5 Service Orchestration Cost Models	54
2.5.1 Mathematical Notation	54
2.5.2 Preliminaries	55
2.5.3 Cost Model for Centralised Service Orchestration	56
2.5.4 Cost Model for Decentralised Service Orchestration	60
2.6 Review of Workflow Technologies	64
2.6.1 Workflow Management Systems	64
2.6.2 Workflow Languages	73
2.6.3 Dataflow Optimisation System Architectures	82
2.6.4 Workflow Scheduling	84
2.6.5 Related Surveys	86
2.7 Requirements Analysis	88
2.7.1 Global Scientific Collaboration	88
2.7.2 Usability Support	89
2.7.3 Service Interoperability Support	90
2.7.4 Parallelism Support	90
2.7.5 Scalability Support	91
2.7.6 Deployment and Optimisation Support	92
2.7.7 Reusability Support	93
2.7.8 Fault-tolerance Support	93

2.7.9	Dynamic Reconfiguration Support	94
2.8	General Discussion	95
2.9	Conclusion	97
3	The Web Service Orchestra Language	99
3.1	Introduction	99
3.1.1	Language Overview	100
3.1.2	General Syntax	104
3.2	Design of the <i>Orchestra</i> Language	108
3.2.1	Guiding Principles	108
3.2.2	Design Methodology	109
3.2.3	Language Grammatical Rules	110
3.2.4	Dataflow Computation Model	113
3.3	Service Composition	116
3.3.1	Pipeline Pattern Specification	116
3.3.2	Data Aggregation Pattern Specification	117
3.3.3	Data Distribution Pattern Specification	119
3.4	Distributed Computation	121
3.5	Type System and Data Structures	122
3.5.1	Base Types and Scalars	123
3.5.2	Data Structures	123
3.6	Conclusion	127
4	System Architecture Design	129
4.1	Introduction	129
4.1.1	Architecture Overview	130
4.1.2	Interactions and Configuration of Components	132
4.2	Engine Design	134
4.2.1	Description of the Engine’s Internal Modules	134
4.2.2	Interactions of the Engine’s Internal Modules	137
4.3	Decentralised Orchestration Stages	138
4.3.1	Compilation	139
4.3.2	Partitioning	139
4.3.3	Network Resource Monitoring	142
4.3.4	Placement Analysis	144
4.3.5	Deployment and Execution	147
4.3.6	Execution Monitoring	148

4.4	Decentralised Service Orchestration Scenario	149
4.4.1	Workflow Specification Example	150
4.4.2	Workflow Orchestration Example	152
4.5	Conclusion	163
5	Reference Implementation	165
5.1	Introduction	165
5.2	Engine Interface	166
5.3	Engine States	169
5.4	Construction of Engine Modules	174
5.4.1	Compiler Module	174
5.4.2	Partitioner Module	189
5.4.3	Network Resource Monitoring Module	195
5.4.4	Placement Analysis Module	199
5.4.5	Deployment Module	201
5.4.6	Execution Module	203
5.4.7	Datastore Module	203
5.5	Conclusion	204
6	Evaluation	205
6.1	Introduction	205
6.2	Examination of Hypothesis	206
6.2.1	Scope	206
6.2.2	Rationale	207
6.2.3	Scalability Dimensions	208
6.3	Experimental Apparatus	211
6.3.1	Experimental Environment	212
6.3.2	Hardware Specifications	215
6.3.3	Software Specifications	216
6.3.4	Experimental Environment Setup	217
6.4	Experimental Methodology	220
6.5	Experiments	222
6.5.1	Experimental Configuration	223
6.5.2	Effect of Engine Locality	224
6.5.3	Pipeline Pattern Evaluation	227
6.5.4	Data Aggregation Pattern Experiments	229
6.5.5	Data Distribution Pattern Experiments	232

6.5.6	End-to-end Workflow Experiment	234
6.5.7	Measurement of Compilation and Partitioning Times	236
6.6	General Discussion	237
6.6.1	Summary of Experimental Results	238
6.6.2	Environmental and Network Factors	240
6.6.3	Evaluation Limitations	244
6.7	Conclusion	248
7	Conclusion and Future Research	251
7.1	Introduction	251
7.2	Thesis Motivation	252
7.3	Thesis Hypothesis	253
7.4	Thesis Summary	254
7.5	Review of Contributions	254
7.6	Future Research Directions	257
7.6.1	Further Evaluation	257
7.6.2	Dynamic Optimisation	258
7.6.3	Real-time Distributed Monitoring of Network Resources	259
7.6.4	Constraint-based Partitioning and Deployment	260
7.6.5	Failure Handling and Recovery	261
7.7	Concluding Remarks	261
References		263

List of Figures

1.1	Example of a small <i>Epigenomics</i> workflow.	4
1.2	Example of a small <i>Montage</i> workflow.	5
1.3	Example of a small inspiral analysis workflow.	6
1.4	Example of a small <i>CyberShake</i> workflow.	7
1.5	High-level preview of the research solution.	19
2.1	A web service consuming, analysing, and producing messages.	28
2.2	The publish-find-bind web service architecture model.	33
2.3	UML diagram of the service description document based on WSDL.	35
2.4	Sequence diagrams of Message Exchange Patterns (MEPs).	36
2.5	The process pattern.	42
2.6	The pipeline pattern.	43
2.7	The data aggregation pattern.	43
2.8	The data distribution pattern.	44
2.9	Scientific workflow lifecycle adapted from Deelman et al. [121].	46
2.10	Generic workflow management system.	49
2.11	Sequence diagrams of service invocations.	56
2.12	Sequence diagram of the pipeline pattern based on centralised orchestration.	57
2.13	Sequence diagram of the data aggregation pattern based on centralised orchestration.	58
2.14	Sequence diagram of the data distribution pattern based on centralised orchestration.	59
2.15	Sequence diagram of the pipeline pattern based on decentralised orchestration.	61
2.16	Sequence diagram of the data aggregation pattern based on decentralised orchestration.	62
2.17	Sequence diagram of the data distribution pattern based on decentralised orchestration.	63

3.1	Dataflow graph of a simple service invocation.	104
3.3	UML diagram of the service description document used in Listing 3.1.	106
3.5	Dataflow graph of services composed in a serial manner.	116
3.7	Dataflow graph of services composed using the aggregation pattern.	118
3.9	Dataflow graph of services composed using the distribution pattern.	120
4.1	Decentralised orchestration architecture.	131
4.2	The execution engine's internal modules and their interactions.	135
4.3	Workflow partitioning algorithm.	141
4.4	Placement analysis algorithm.	145
4.5	Placement analysis example.	146
4.6	A Directed Acyclic Graph (DAG) workflow.	149
4.8	Orchestration of the workflow specification provided in Listing 4.7.	153
4.9	Sub workflows after decomposing the workflow shown in Figure 4.6.	156
4.10	Composite workflow that corresponds to partition (P1) in Figure 4.8.	158
4.11	Composite workflow that corresponds to partition (P2) in Figure 4.8.	158
4.12	Composite workflow that corresponds to partition (P3) in Figure 4.8.	159
5.2	Finite-state machine diagram of the engine states and their transitions.	170
5.3	Class diagram showing the main classes and packages used to implement the compiler.	176
5.4	Dataflow graph structure based on the workflow example in Figure 4.6.	180
5.5	Class diagrams showing the classes of the symbol table package.	181
5.6	Class diagram showing classes of the symbols package.	183
5.7	Class diagram showing the classes of simple and record data type representations package.	184
5.8	Class diagram showing the classes of service-based type representations package.	186
5.9	Class diagram showing the classes of error registry package.	189
5.10	Dataflow graphs based on the sub workflows in Figure 4.9.	192
5.11	Dataflow graph that represents the workflow in Figure 4.10.	
	193	
5.12	Dataflow graph that represents the workflow shown in Figure 4.11.	
	194	
5.13	Dataflow graph that represents the workflow shown in Figure 4.12.	
	194	

5.14	Class diagram showing the workflow partitioning and placement analysis classes.	202
6.1	Geographic network regions of <i>Amazon Elastic Compute Cloud</i> (EC2).	213
6.2	Round-trip times for invoking a service in N.Virginia.	225
6.3	Network latency and bandwidth readings.	226
6.4	Orchestration of pipeline pattern workflows.	228
6.5	Orchestration of data aggregation pattern workflows.	231
6.6	Orchestration of data distribution pattern workflows.	234
6.7	Structure of an end-to-end workflow application.	235
6.8	Experimental results for orchestrating an end-to-end workflow that combines common dataflow patterns.	236

List of Tables

2.1	Elements of a service description document.	35
4.1	Placement of sub workflows shown in Figure 4.9 onto candidate engines. .	157
4.2	Composition of sub workflows shown in Figure 4.9.	157
5.1	Mapping of XML schema built-in data types to <i>Orchestra</i>	188
6.1	Summary of virtual machine types used in the experiments.	216
6.2	Summary of network metrics between the engines and the service in N. Virginia.	226
6.3	Configuration of services used in pipeline pattern workflows.	227
6.4	Summary of average speedup for orchestrating pipeline pattern workflows. .	229
6.5	Configuration of services used in data aggregation pattern workflows. . . .	230
6.6	Summary of average speedup for orchestrating data aggregation workflows.	232
6.7	Configuration of services used in data distribution pattern workflows. . . .	233
6.8	Summary of average speedup for orchestrating data distribution workflows.	233
6.9	Summary of the average compilation and partitioning times for some workflows.	237

Chapter 1

Introduction

Since ancient times scientists have attempted to reshape the conceptual landscape through which they comprehend the world. Early empirical scientists such as Newton, Galileo, Hooke and others were not only considered polymaths but also craftsmen of instruments with which scientific discoveries are made. Such instruments were not only used in conducting research but also helped in establishing new scientific frontiers by exposing unknown phenomena or revealing bizarre observations that contradict with existing theories. It is therefore safe to argue that the examination of such instruments, their engineering and construction is valuable to the progression of science. In the 21st century, modern science relies on software instruments or tools to complement theory and experiment through capturing, curating and analysing data that comes in different forms covering scientific observations and experimental results. For example, computer simulations have become essential for scientists to explore domains that are inaccessible to theory and experiment such as the evolution of the universe, and predicting climate change according to Bell et al. [1]. Scientific data may be captured and processed along with computer-generated information that is most likely to reside in a curated state for continuous experimentation, and analysis over a period of time. However, some scientific areas are facing a rapid expansion of data obtained from instruments (e.g. satellites, telescopes, sensor networks, and supercomputers), and generated from experimental simulations [2]. Simulations in the field of astronomy may generate petabytes of data each year [1]. Gray [3] argues that as scientific experiments yield

ever more data, a *fourth paradigm* is emerging, consisting of techniques and technologies needed to perform data-centric science. Scientific experiments must be managed easily using scalable data processing methods that may be beyond the skill set of many scientists (e.g. astronomers, bioinformaticians, and chemists).

Modern scientific experiments are typically specified as a set of computational tasks in a precise order known as a *workflow*. Each task represents the execution of a process that accepts an input data set and produces a result that may be consumed by subsequent tasks. The principal objective of using workflows is to liberate scientists from the laborious routine of processing and analysing data so they can focus on scientific research and discovery. There are a number of benefits of using workflows. Firstly, a workflow provides a systematic method of conducting analysis across diverse datasets and applications. Secondly, a workflow provides the ability to gather datasets captured from scientific instruments from low-cost sensors to supercomputers, and employs resources to perform computational tasks over gathered data. Thirdly, a workflow captures the steps of an experiment and its associated results. This permits scientists to use pre-cooked workflows to conduct experiments repeatedly and reproduce results as needed using the latest data sources when required. Finally, workflows permit the integration of independent computation and data resources from different providers. They are becoming essential for enabling science and facilitating the collaboration between scientists over shared data and computation resources on a global scale, and extracting computational knowledge.

Goble and De Roure argue that applying computational workflow technology to data-centric scientific research is crucial [4]. This is because workflow technology provides systematic and automated means of conducting analysis over data and across different applications, and permits the scientific process (e.g. experiment) to be captured and executed repeatedly to reproduce results. This helps scientists to review the experimental results, and refine the experiment by introducing modifications to the workflow as necessary. The same workflow technology used in executing experiments and capturing their results have resulted in an unprecedented explosion of data that is commonly known as the *data deluge*. This term was first coined by Hey and Trefethen [5]. Scientific research is being affected by

the data deluge. Callaghan et al. [6] discuss how the growth of data increases the complexity of managing the execution of scientific workflow applications. There are many examples of large-scale scientific projects that rely on workflows. For example, the *Panoramic Survey Telescope and Rapid Response System* (Pan-STARRS) [7] project focuses on detecting potentially hazardous objects in the solar system using telescope technology that captures terabytes of images per year which are processed using the *Trident* [8] workflow management system. The *Low Frequency Array for Radio Astronomy* (LOFAR) [9] is another project example that permits scientists to detect coronal mass ejections from the Sun, and produce maps of the solar winds. Such information is crucial to predict geomagnetic storms and their effect on Earth. This project relies on interferometric array of radio telescopes based on thousands of small antennas distributed across Europe. It involves processing data captured in real-time from different stations connected through a wide-area network using the IBM Blue Gene/P supercomputer [10]. The *Collaborative Adaptive Sensing of the Atmosphere* (CASA), and the *Linked Environments for Atmospheric Discovery* (LEAD) are complementary projects that aim to study weather phenomena such as storms and tornados [11]. The CASA project provides a distributed, collaborative, sensor network of low-power, and high-resolution radars, whereas the LEAD project provides a service-oriented workflow management system that is designed to support on-demand analysis of data obtained from the sensor network. The projects described above employ large-scale workflows that involve capturing, processing, analysing and generating data. Gordon et al. [1] argue that there is no generic data-centric solution that is readily available to cope with workflows of this kind of complexity which may involve this amount of experimental data.

Scientific workflows are typically modelled as *Directed Acyclic Graph* (DAG) in which the vertices are tasks with directed edges between them as data dependencies that indicate the data movement in and out of tasks. Bharathi et al. [12] provides a set of scientific workflows that are taken from the *Pegasus* [13] project. This thesis describes the structure of these workflows which are composed of common dataflow patterns that include the pipeline, data aggregation and distribution patterns. The pipeline pattern is used to compose workflow tasks sequentially by directing the output of a particular task as an input to a subsequent task.

The data aggregation pattern is used for collecting outputs from different tasks, and passing them all as inputs to a single task that acts as a data sink. The data distribution pattern is used for distributing the output of a single task that acts as a data source to multiple subsequent tasks as an input. This thesis provides a detailed discussion of these patterns in Section 2.3.2. It is essential to note that the workflow examples provided below are only used to describe the general structure of scientific workflows, and they are not used as use case studies for evaluation purposes in this thesis.

- **Biology:** Geneticists use workflows to perform genetic sequencing, which represents a process that determines the precise order of nucleotides within a genetic molecule. *Epigenomics* is an example of such workflows, which is used to automate the execution of various gene sequencing operations [12]. Figure 1.1 shows the structure of a small *Epigenomics* workflow as depicted in [12]. In this workflow, a generated dataset

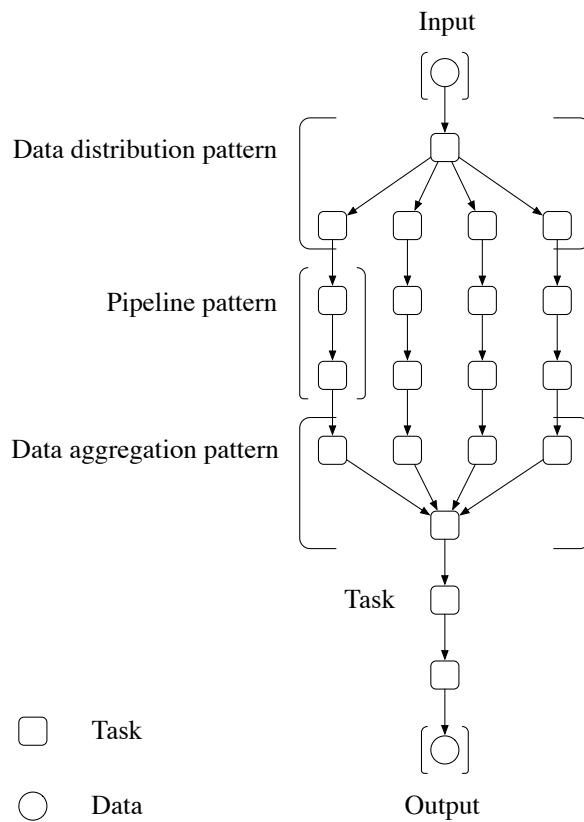


Fig. 1.1 Example of a small *Epigenomics* workflow.

by a genetic analyser component is split into parts that can be processed in parallel. Each part is then converted into an appropriate format for processing in a series of tasks that include filtering out erroneous sequences, mapping sequences into the correct location in a reference genome, generating a global map and then determining the sequence density at each position in the genome.

- **Astronomy:** Modern astronomers use workflows to generate sky mosaics from images collected using infrared telescopes to examine celestial objects [14]. *Montage* represents an example of such workflows which consists of tasks that process, analyse and manipulate input images to produce a single mosaic image. Figure 1.2 shows a relatively small *Montage* workflow as depicted in [12]. During the production of

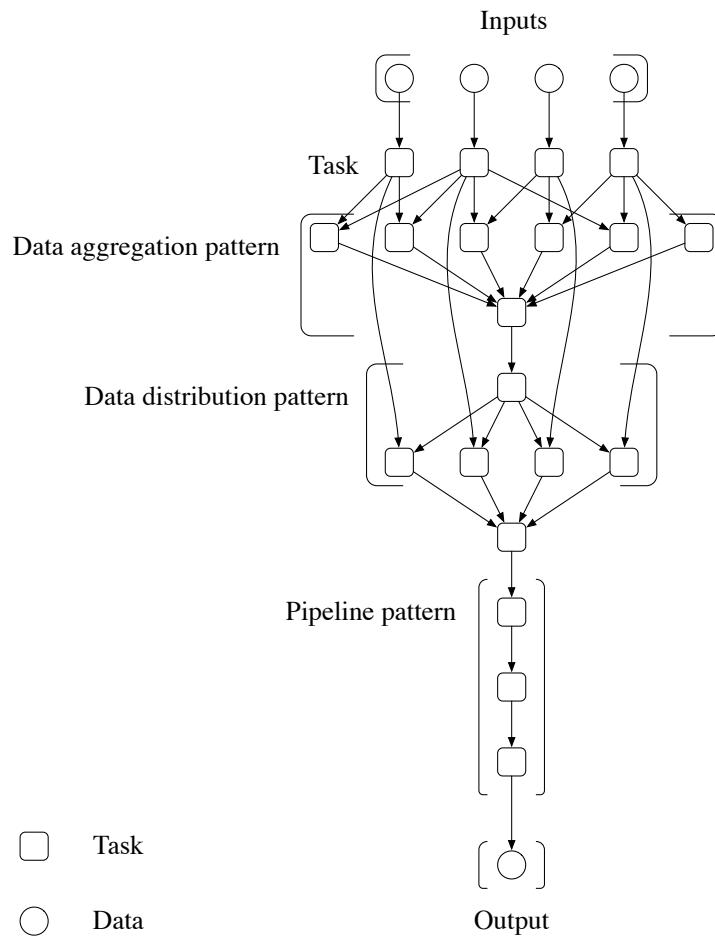


Fig. 1.2 Example of a small *Montage* workflow.

the mosaic image, its geometry is calculated from the geometry of the input images. These input images are then re-projected to be of the same spatial scale and rotated appropriately. Then, the background emissions (e.g. noise) in the images are then modified in a manner such that level of these emissions is the same in all images. Finally, the re-projected images are combined to form the final mosaic image. *Montage* is executed in grid environments [15], and can be highly data-intensive. For example, it can be used to produce a mosaic of the galactic plane [16] from input images provided by NASA's *Telescope Missions* (e.g. *Spitzer*). This involves 15.5 million tasks and produces results of approximately 86 Terabytes.

- **Gravitational physics:** Physicists use the *Laser Interferometer Gravitational Wave Observatory* (LIGO) inspiral analysis workflow [17] shown in Figure 1.3 as depicted in [12] to detect gravitational waves produced by certain cosmic events. These gravitational waves were first predicted by Einstein's general theory of relativity at a time when the technology needed for their detection did not exist. For example, binary sys-

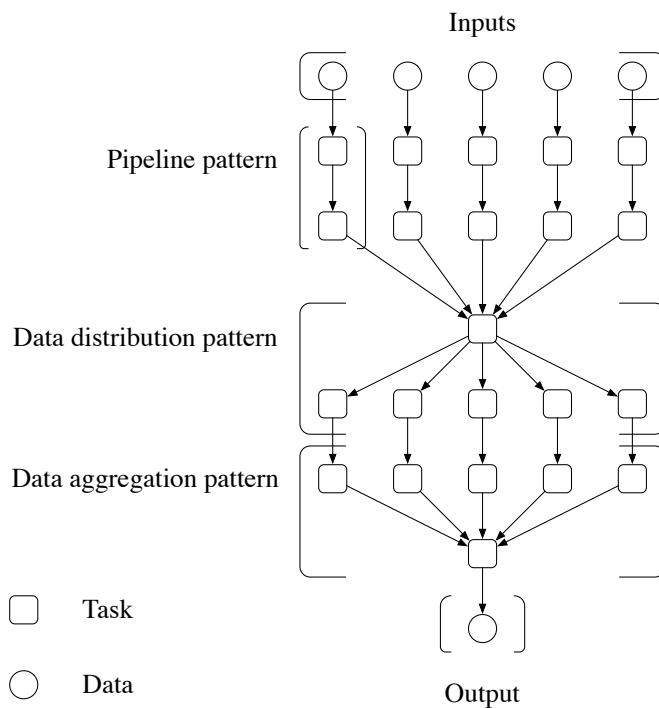


Fig. 1.3 Example of a small inspiral analysis workflow.

tems (e.g. binary neutron stars) slowly inspiral as gravitational waves carry away energy and momentum, and consequently merge together forming bursts of short gamma rays. This workflow takes a dataset obtained from the coalescing of binary systems and splits it into smaller parts, where each part is analysed to generate data that represents waveforms belonging to a certain parameter space. During the workflow, several tests may be performed over the data to detect a true inspiral.

- **Seismology:** Seismologists employ workflows to characterise earthquake hazards in a specific geographic region using probabilistic techniques. *CyberShake* is an example of such workflows which produces synthetic seismograms from analysing data collected by scientific instruments. Figure 1.4 shows the structure of a relatively small *CyberShake* workflow as depicted in [12]. Once the data is analysed from the scientific instruments, spectral acceleration and probabilistic hazard curves are generated. This workflow commonly involves thousands of individual tasks that are typically executed in a grid-based environment [18].

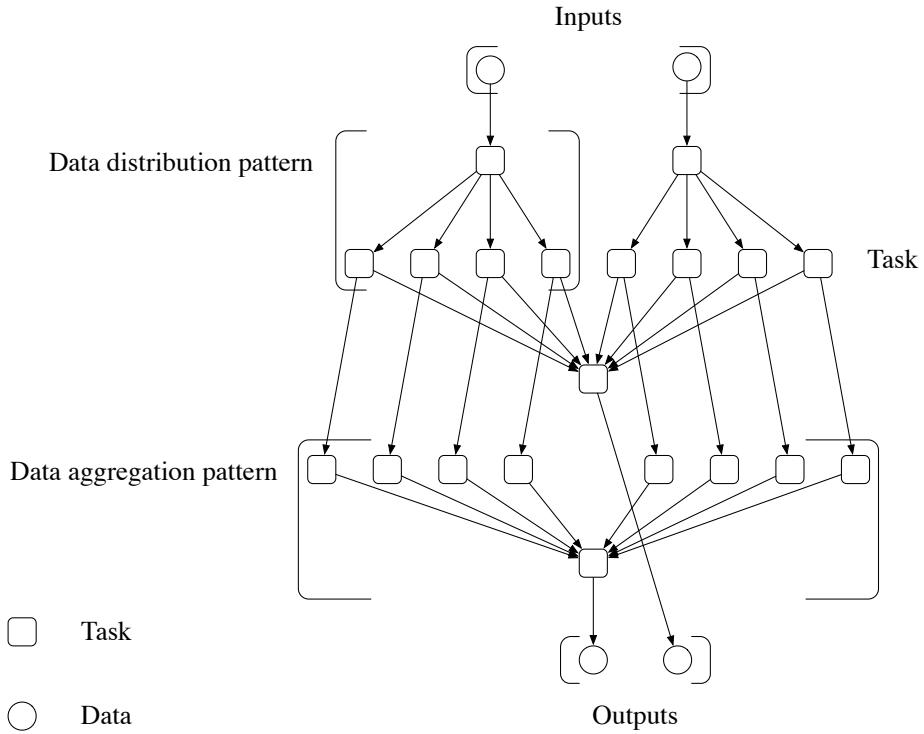


Fig. 1.4 Example of a small *CyberShake* workflow.

Service computing provides an approach to constructing and managing workflows by composing loosely-coupled services together and orchestrating their work. Services are the manifestations of integrated software components that can be exposed to a network by logical interfaces, which permit services to interact with their clients by exchanging messages in a standard format. Creating a service involves describing the operations it supports, defining the communication protocol used to invoke those operations and give results, and operating a server machine to process incoming requests to the service. Service-oriented technology can be beneficial to scientific applications. It is gaining wide acceptance in the industrial domain, and therefore different services exist which may be distributed across many service providers and hosted in different geographic locations. There are many commercial and open-source tools for creating, deploying and operating services. Therefore, different service implementations can exist and may be distributed across many providers, which permits a scientist to select candidate services to fulfil the workflow tasks.

Parastatidis [19] discusses the potential of using service-oriented architectures in establishing a global knowledge-driven infrastructure that improves the process of scientific research. This infrastructure must provide the right set of services to not only enable scientists to access research data but also to expose computational services which operate on that data. Scientists would be able to synthesise experiments using existing computational services and data provided by this infrastructure. This infrastructure should also enable scientists to ask questions related to their domain expertise, which may be answered by other scientists or service-oriented knowledge engines. The ^{my}Experiment [20] project is an example of a service-oriented infrastructure that demonstrates the benefits of capturing and sharing scientific workflows, in addition to information about how these workflows are used. However, an explicit limitation of using services is that their execution is bound to the resource machines hosting them. This means that workflows must be mapped onto services statically. Services are also agnostic of the existence of other services. This does not permit services to collaborate with each other automatically. Hence, service-oriented workflows must be composed from a high-level design perspective that supports the interactions between services needed to conduct a scientific experiment.

Scientists commonly compose workflows using a *workflow language* that provides a set of abstractions for describing the workflow tasks (e.g. jobs) and the dependencies between them (e.g. control and data flows). The result of this composition is a *workflow specification* that provides a logical model (e.g. abstract workflow) which defines structured tasks to be executed in a specific order [21], [22], [23]. This workflow specification does not include information about the physical resources used in the execution of the workflow, and therefore it is processed and analysed using a tool that generates an execution plan (e.g. concrete workflow), which maps each task in the workflow onto a computation service for execution [21], [24], [25]. Such an execution plan describes the data transfer amongst the services to support their collaboration. This thesis refers to the execution and management of a workflow specification as *service orchestration*, which relies typically on a centralised compute server known as the *workflow engine*. The centralised workflow engine oversees the interactions between services, provides control over the workflow, supports process automation, and encapsulates the workflow logic at a single location according to Erl [26]. It exploits connectivity to the services by invoking their functionality using input data, and collecting the invocation results. Furthermore, the engine is responsible for composing the services participating in the workflow by routing data from one service to another based on the workflow specification.

The remainder of this chapter is organised as follows: Section 1.1 presents the research statement which describes the central research problem in centralised service orchestration, and states the research question that is covered in this thesis. Section 1.2 identifies a set of research challenges that this thesis attempts to address. Section 1.3 proposes a research hypothesis that describes a possible solution based on a decentralised service orchestration approach. Section 1.4 describes the research contributions presented in this thesis to the body of knowledge in the area of service computing generally, and service orchestration specifically. This section classifies the research contributions into primary and secondary contributions. Section 1.5 provides a preview of the decentralised service orchestration approach presented in this thesis. Section 1.6 provides the glossary of terms. Section 1.7 presents the organisation of this thesis. Finally, Section 1.8 concludes this chapter.

1.1 Research Statement

This thesis attempts to address a research problem that focuses on a number of issues: the design of scientific service-oriented workflows, and the scalability and optimisation of their orchestration. These concepts are discussed as follows:

1. Scientists must be able to design (e.g. model) a service-oriented workflow easily.
There are a plethora of workflow languages for the specification of workflow tasks (e.g. jobs) but most of which are focused on business-oriented workflows that are ill suited to compose scientific workflows, and may require previous knowledge of how the workflow is executed.
2. Most existing workflow management systems are based on a centralised orchestration model to execute workflows. Centralised orchestration presents a significant scalability problem as the workflow engine may become a performance bottleneck. During the workflow's execution, all data going to or coming from the services pass through the engine. This leads to unnecessary network traffic that can consume the engine and consequently disable it. For example, the engine may invoke a service that produces a large data set which may be required by other services to perform further computational tasks. Such data will be received by the engine, which neglects the data after forwarding it to the services that require it. This causes unnecessary network traffic, and introduces additional load on the engine that leads to a single point of failure as the number of services and data in the workflow increase. Hence, a scalable decentralised solution is required to address the limitations of centralised service orchestration.

Based on the issues described above, an alternative approach must be discovered for the specification, execution, and optimisation of service-oriented workflows. This thesis investigates if it is possible to address the research problem by creating a high-level language that permits service-oriented workflows to be specified easily, and a scalable system for executing workflows that supports the optimisation of the data transfer amongst the services.

1.2 Research Challenges

Jim Gray laid out a vision for a *fourth paradigm* of science that is concerned with the construction of new technology needed to perform data-intensive research as discussed in Hey et al. [3]. This paradigm's principal objective is to create a data-driven world in which scientists can collaborate to solve research challenges using global computing and data resources. Despite the huge potential of data-intensive science, it has been progressing slowly due to the general lack of understanding of the essential means needed for its realisation by the research community. This has motivated the author of this thesis to rethink the design and construction of service orchestration systems by considering a set of challenges that are classified as follows:

1. Challenges in designing scientific service-oriented workflows easily.
2. Challenges in decentralised service-oriented orchestration.
3. Challenges in global computation involving geographically distributed services.

1.2.1 Design Challenges

Ludäscher et al. argues that the principal objective of using workflows is to enable scientists to compose and orchestrate workflows easily by reducing the effort required to construct them [27]. Designing service-oriented workflows requires expertise that is often beyond the skills of many researchers, which include programming knowledge, understanding of service behaviour, the execution mechanism of the workflow. For example, researchers are typically forced to use scripting languages to compose functionality provided by independent software modules to perform tasks in a workflow context. Such modules may be written in different programming languages, and this forces the researcher to think how these modules need to be interfaced instead of describing what functionality needs to be executed. Due to this rigid programming approach, workflows may need to be written entirely from scratch or modified when the implementation of the independent modules changes. Existing workflow languages provide abstractions that permit workflows to be composed using

control-flow structures. This increases the difficulty of composing large-scale workflows as the researcher will be more concerned with the application-specific behaviour of the tasks to be executed instead of describing the data flow dependencies required to execute the tasks. Hence, workflow languages must provide high-level abstractions that permit scientists to compose and orchestrate workflows easily without knowledge of how the workflow is executed.

1.2.2 Decentralised Service Orchestration Challenges

Decentralisation is inherently more complicated to implement than centralised service orchestration due to the interactions between many distributed, asynchronous, and concurrent service processes as according to Barker et al. [28]. For example, it presents a set of research challenges that include partitioning a centralised workflow specification into smaller executable sub workflows, mapping and deploying these sub workflows onto appropriate engines, executing these sub workflows and monitoring the overall workflow execution. These challenges require analysing the workflow to detect its intricate data dependencies, and decomposing it in a manner such that the data transfer is minimised between the services and engines while parallelism is maximised amongst the engines. Placement analysis is required to determine the most appropriate engines onto which the sub workflows are deployed, but selecting a candidate engine to execute a particular sub workflow is challenging as the engine instance is typically determined at run-time. Furthermore, a mechanism is required to manage the order in which the sub workflows are executed and coordinate the data movement between the engines. These challenges need to be addressed properly to exploit the full potential of decentralisation.

1.2.3 Global Computation Challenges

Service-oriented architectures can be used to create new technology that enables the linkage of data and computing resources around the globe. Service computing infrastructures (e.g. clouds) have already affected the manner in which scientific experiments are performed

and managed due to their support for processing and storing research data. It is certainly appealing to compose and orchestrate workflow application that can exploit such global network infrastructures. Unfortunately, such infrastructures present significant challenges that undermine common assumptions about the behaviour and performance of workflows executed in local networks. These challenges are described as follows:

1. Workflow tasks may be executed on heterogeneous machines that may be physically or virtually present at network locations in different geographic regions [29]. This may influence the overall speed of transmitting information from one network location to another across different administrative domains. Such physical limitation has drastic consequences for distributed applications relying on global resources.
2. Network latency and bandwidth suddenly become direct factors that limit the overall performance of such applications regardless of processing power and storage capacity as argued by Cardelli [30]. The condition of a global network infrastructure is susceptible to change due to unpredictable congestion (e.g. competing traffic), which results in temporary fluctuations of bandwidth that may create performance bottlenecks.
3. Both service failures and long delays become indistinguishable in a global network environment according to Cardelli [31]. Services may fail suddenly (e.g. become unavailable) and therefore cannot be reached (e.g. accessed or invoked), and the network latency can affect the speed in transmitting data between different endpoints. Such behaviour may be frequent and unpredictable.

Therefore adequate measures must be taken to control the distribution of computation over the locality of data, and react to the uncertainty of the network to avoid underperformance issues. This thesis presents a research solution that determines the most appropriate network locations at which to execute the workflow based on Quality of Service (QoS) metrics collected from the execution environment (e.g. network latency and bandwidth) to optimise the performance of the workflow. However, the issue of dealing with service failures during the workflow execution could not be addressed due to time constraints and therefore it is beyond the scope of this thesis.

1.3 Research Hypothesis

This thesis argues that centralised orchestration is inadequate to support large-scale service-oriented workflows as it relies on a single coordination and so under-performs. Therefore, this thesis offers the following hypothesis:

“Decentralisation provides a solution that addresses some of the scalability and performance limitations of centralised service orchestration. It can be achieved by decomposing the workflow logic into smaller parts that may be distributed across multiple execution engines at appropriate locations with short network distance to the services providing the data. This reduces the overall data transfer in the workflow and improves its execution time.”

This research hypothesis is based on a number of tenets or beliefs which are described as follows:

- **Lack of a centralised coordinator supports scalability:** The research hypothesis suggests that decentralised orchestration can overcome the scalability limitations of a completely centralised orchestration approach. This is because there is no centralised engine acting as a “middle-man” to coordinate the data movement between the services. This can help in avoiding performance bottlenecks and can potentially improve scalability as the number of services participating in the workflow increases.
- **Distribution of the specification of the workflow logic supports parallelism:** The distribution of the specification of the workflow logic supports parallelism, and permits the workflow partitions (e.g. specifications of sub workflows) to be executed independently by multiple engines at network locations nearer to the services providing the data. This can improve the overall service response time (e.g. round-trip times between the services and engines) and the overall throughput.
- **Distribution of the intermediate data improves the overall performance:** the distribution of the intermediate data across the engines can help in reducing the network

traffic and the overall time of data transfer in the workflow. This is because the intermediate data in the workflow is forwarded directly by the engines to network locations where the data is required, without passing through a centralised engine.

The scope of this research hypothesis covers service-oriented workflows and takes into consideration the study of the scalability and performance benefits of a decentralised service orchestration approach over completely centralised service orchestration approach only. This research hypothesis is examined in Section 6.2 which describes the thesis scope, rationale, and scalability dimensions in detail.

1.4 Research Contributions

This thesis addresses the research hypothesis and challenges discussed in Section 1.2 by providing a set of contributions in the area of service computing in general, and service orchestration specifically. These contributions are classified into primary, secondary, and tertiary contributions.

1.4.1 Primary Contributions

The primary contributions of this thesis include a high-level, functional, data coordination language for the specification of workflows, and a decentralised service-oriented orchestration system architecture for the execution of workflows.

- **High-level data coordination language:** This thesis presents a high-level, functional data coordination language called *Orchestra* for the specification of service-oriented workflows. This language separates the workflow logic from its execution, supports parallelism, determinism, and data-driven execution. It provides simple abstractions for defining a set of services, composing them, and regulating the data movement between them.
- **Decentralised service-oriented orchestration system architecture:** This thesis presents a decentralised architecture that is designed to orchestrate a workflow specified in the

Orchestra language using multiple compute servers called *workflow engines*. Each engine is responsible for analysing and executing part of the overall workflow logic by exploiting connectivity to the services, and forwarding the intermediate data directly to locations where it is required.

1.4.2 Secondary Contributions

The secondary contributions of this thesis include a workflow partitioning approach for decomposing the workflow into smaller parts that may be executed in parallel, a placement analysis mechanism that determines the most appropriate network locations at which to execute the workflow parts, and experimental evaluation of the presented decentralised solution.

- **Workflow partitioning approach:** This thesis presents a novel approach that decentralises a workflow by transforming its specification into smaller partitions, each represents an independent sub workflow. These partitions can be executed in parallel to improve the overall workflow performance. This approach analyses the data dependencies in a workflow to detect its intricate parallel parts that can be isolated, and relies on placement heuristics to determine a set of appropriate workflow engines to execute these parts.
- **Placement analysis mechanism:** This thesis presents a computation placement analysis mechanism that is responsible for determining a candidate engine to execute a particular workflow partition. It relies on *Quality of Service* (QoS) metrics such as the network latency and bandwidth between the services and the engines. These metrics are used to select the nearest engine to a group of services for executing a relevant workflow partition. Consequently, all workflow partitions are deployed onto a set of designated workflow engines for execution, and their deployment activity is transparent to the user. Following deployment, real-time distributed monitoring is used to watch the workflow execution to address emergent run-time issues such as unexpected failures.

- **Experimental evaluation:** This thesis evaluates the solution implementation in terms of scalability and performance by orchestrating experimental workflows over *Infrastructure as a Service* (IaaS) clouds across several geographical locations.

1.4.3 Tertiary Contributions

The tertiary contributions of this thesis includes a literature survey that explores the area of workflows and related technologies, a concrete set of requirements needed for composing and orchestrating scientific workflows based on the limitations of existing workflow technologies, and future research directions in this area.

- **Literature review:** This thesis presents a comprehensive study of workflows and related management approaches in grid-based and service-oriented computing. By analysing these approaches this thesis has been able to highlight the limitations of existing works, and identify the research challenges in composing and orchestrating workflows.
- **Decentralised service orchestration requirements:** This thesis defines a set of concrete requirements for composing services and their orchestration in a decentralised manner. These requirements are not satisfied by existing workflow specification languages and execution architectures.
- **Future research directions identification:** This thesis identifies future research directions based on the current state-of-art in workflow management approaches, and the evaluation results of the proposed decentralised orchestration solution.

1.5 Preview of Research Solution

This thesis presents a novel decentralised service-oriented orchestration system architecture that relies on a high-level, data coordination language called *Orchestra* for the execution of workflows. This language provides a limited set of abstractions that are needed only to

invoke or compose services. Therefore, a scientist can focus on composing a workflow without knowledge of how it is executed, and pass the responsibility of executing the workflow to a decentralised orchestration system. Decentralised orchestration is achieved through decomposing a workflow specified based on *Orchestra* into smaller partitions (e.g. sub workflows) that may be executed in parallel, and determining the most appropriate engines at which these partitions may be executed efficiently based on placement analysis. This permits the workflow logic to be moved towards the services providing the data in the workflow. For example, each workflow partition is executed by an engine that maintains short network distance (e.g. latency) and high bandwidth with a group of services participating in the workflow. This can reduce the round-trip time required for the engine to invoke the services, and receive the invocation results, which improves the overall execution time of the workflow. Multiple engines can collaborate with each other by transferring data relating to the workflow execution directly to locations where they are required. This thesis provides an evaluation of the presented approach, which concludes that decentralised orchestration improves the overall execution time of a workflow, and provides scalability over centralised orchestration.

Figure 1.5 shows a preview of the presented approach. Based on this diagram, a scientist provides a workflow specification to an initial engine (E). This engine is usually running on the scientist’s local machine, and it is responsible for processing and analysing the workflow to ensure its correctness. It uses a workflow partitioning approach that decomposes the workflow logic into smaller partitions that can be executed in parallel, and performs placement analysis to determine the most appropriate engines onto which these partitions may be deployed for execution. Typically, placement analysis relies on gathering *Quality of Service* (QoS) metrics such as the network latency and bandwidth, which indicate the geographical distance between the services and the engines. This helps in determining a set of candidate engines hosted “closer” to the services to execute the workflow. Before the deployment process, each partition is encoded in a form that is suitable for network transmission. The initial engine (E) deploys the workflow partitions by transmitting them to engines (E_1) and (E_2), which in turn execute the partitions by invoking a group of services. For example,

engine (E_1) invokes services (S_1) and (S_2), whereas engine (E_2) invokes services (S_3) and (S_4). Engines (E_1) and (E_2) may collaborate with each other by exchanging data segments needed to execute the workflow partitions.

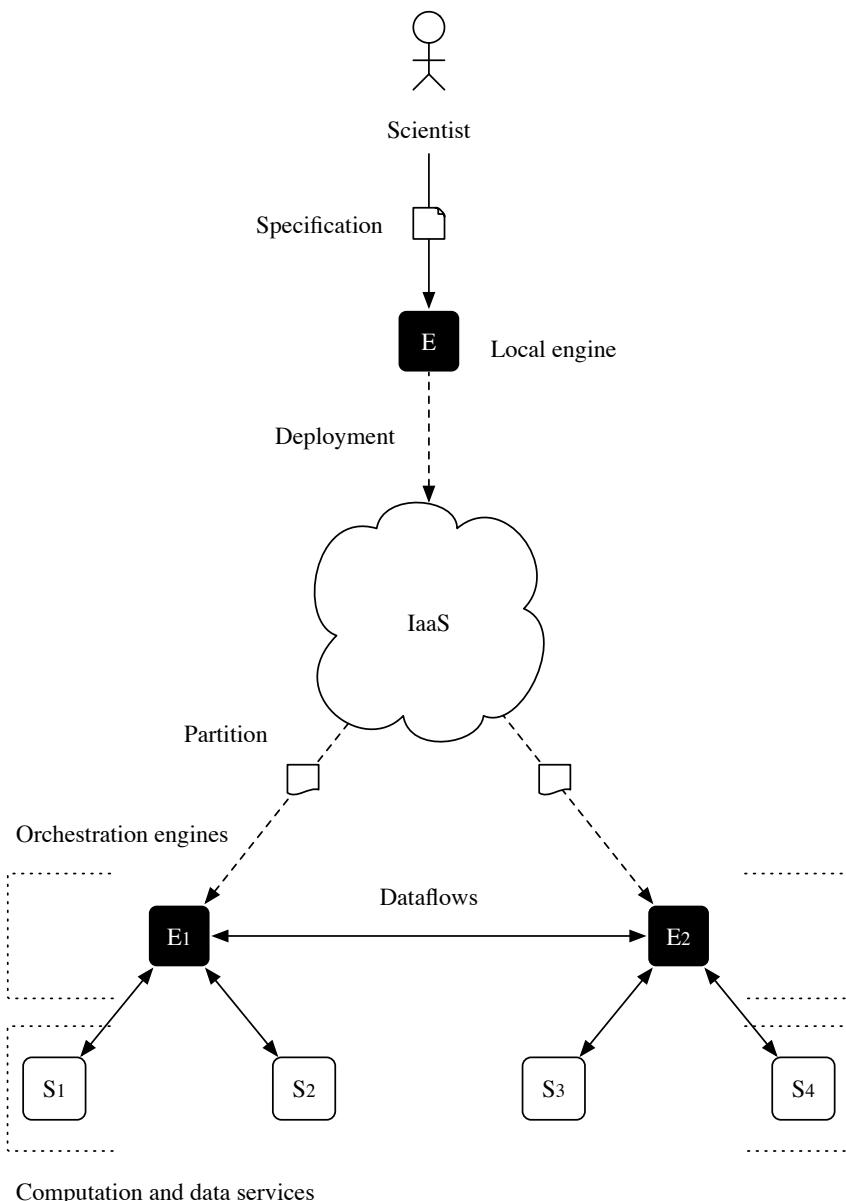


Fig. 1.5 High-level preview of the research solution.

1.6 Glossary of Terms

Researchers often use different terminology to describe the same concepts, and to avoid ambiguity the terminology introduced here is used consistently throughout this thesis.

- **Grid:** This term refers to a network of computing resources that are hosted on the Internet by different providers.
- **Resource:** This term refers to hardware that is used to host software components and services.
- **Service:** This term refers to a software component that can be exposed to a network using a standard interface, which describes a set of functionality compliant with the web service architectural model defined by the *W3C Web Services Architecture Working Group* [32].
- **Interface:** This term refers to a named set of operations that characterise the behaviour of a software component or a service.
- **Service-oriented computing:** This term refers to the use of loosely-coupled web services to construct and execute large-scale distributed applications.
- **Task:** This term refers to a computational step in a business-oriented or scientific process, and many steps can be composed together to construct workflows.
- **Workflow:** This term refers to a distributed application that is composed of a sequence of tasks that are structured based on their data dependencies.
- **Partition:** This term refers to part of an overall workflow (e.g. sub workflow) that is composed of a sequence of tasks organised according to data dependencies.
- **Orchestration:** This term refers to a design pattern authored by Erl and Loesgen¹ that is dedicated the effective maintenance and execution of business process logic.

¹See the definition of the term “Orchestration” at <http://serviceorientation.com/soaglossary/orchestration>

- **Service orchestration:** Service orchestration refers to the enactment or execution of a workflow and its management in a distributed system, which involves coordinating the data movement across a set of computing services to support their collaboration.
- **Execution engine:** This term refers to a compute server that is responsible for executing a workflow or part of it. It provides capabilities for invoking a set of services participating in a workflow, collecting and storing the invocation results, and coordinating the data movement between them.
- **Data deluge:** This term was first coined by Hey and Trefethen [5] that represents the unprecedented explosion of data resulting from conducting scientific experiments.
- **Data-intensive:** This term is an adjectival phrase which denotes that a particular item to which the term is applied requires attention to the data and the manner in which the data is handled.
- **Dataflow patterns:** This term refers to a set of design patterns that can be used to organise a workflow as a *dataflow graph*, which is a *Directed Acyclic Graph* (DAG) that consists of vertices as workflow tasks with the edges between them as data dependencies.
- **Data coordination languages:** This term refers to a type of functional languages that are based on a model of coordination or communication among several primitives (e.g. tasks or processes) that operate over data objects based on some type system.
- **Decentralised system architecture:** This term refers to a type of software system architecture that involves peer-to-peer collaboration between its components, and constitutes equality between them as explained by Taylor and Harrison [33].
- **Network region:** This term refers to a distinct geographic area in the world where hosting servers of execution engines or web services are based.

1.7 Thesis Organisation

The rest of this thesis is organised as follows:

- **Chapter 2:** This chapter presents a general background of service computing, service-oriented architecture, web services and related standards. It provides an overview of scientific workflows, discusses the design patterns used for composing them, presents the lifecycle of scientific workflows, highlights the common features of scientific workflow management systems, describes service orchestration in general and explains the execution models of service-oriented scientific workflows based on centralised and decentralised orchestration approaches. Furthermore, it provides a comprehensive survey of workflow technologies including workflow languages, management systems, dataflow optimisation system architectures, and workflow scheduling heuristics.
- **Chapter 3:** This chapter presents a high-level, functional data coordination language for the specification and execution of service-oriented workflows called *Orchestra*. It presents the features of this language, describes the general syntax of the language, provides its design specification, introduces a set of examples that demonstrate the language ability to compose services using common dataflow patterns, discusses the language support for distributed computation, and its type system.
- **Chapter 4:** This chapter presents the architectural design of a decentralised service-oriented orchestration system that is responsible for executing workflows based on the *Orchestra* language. It describes the interactions between the distributed components of the system architecture such as execution engines and services, discusses the design of the execution engines and their internal modules, and presents the steps of decentralised orchestration from the compilation of a workflow specification to the deployment of workflow partitions and their execution. Furthermore, it provides an example that is used to explain these steps in detail.

- **Chapter 5:** This chapter discusses the implementation of the presented orchestration system architecture. It presents the implementation of an execution engine’s interface, discusses the computational states that an execution engine can exhibit before, during, and after the workflow execution, and discuss the construction of the engine’s internal modules.
- **Chapter 6:** This chapter provides an evaluation of the presented decentralised service-oriented orchestration architecture to test the hypothesis proposed in the thesis introduction.
- **Chapter 7:** This chapter finally concludes this thesis by summarising the work and presenting future research directions to extend the work covered in this thesis.

1.8 Conclusion

This thesis focuses on the construction of a decentralised service-oriented orchestration system that permits the execution of scientific workflows based on a simple data coordination language. This chapter has presented a general introduction to workflows, described the significance of workflows in scientific research, and provided examples of scientific workflows. It provided a research statement that discussed the central research question that has guided this research. Furthermore, it discussed some of the research challenges addressed in this thesis, presented the research hypothesis and explained its rationale. This chapter has also listed the research contributions of this thesis, and provided a preview of the research solution which is a decentralised service-oriented orchestration system that relies on a simple data coordination language for the execution of scientific workflows.

Chapter 2

Literature Review

2.1 Introduction

This chapter establishes general understanding of the subject matter, and focuses at the aspects that led to the realisation of this thesis. It begins with Section 2.2 which provides an overview of the service computing paradigm, and describes *Service-oriented Architecture* (SOA) and its fundamental concepts. This section also describes what constitutes a web service, and discusses the course of evolution that led to the creation and standardisation of web service technology. Then it presents the web service architecture model and associated standards, the central limitations of this model and describes the efforts of the research community and industrial organisations to address these limitations using web service specifications. Section 2.3 describes what constitutes a workflow in general and discusses scientific workflows specifically. It describes concepts that relate to the construction and enactment of workflows including the workflow lifecycle, service composition. Section 2.4 defines service-oriented orchestration and describes centralised and decentralised service-oriented orchestration system architectures, whereas Section 2.5 provides the execution cost model of these architectures. Section 2.6 provides a comprehensive review of existing workflow management systems, workflow languages, dataflow optimisation system architectures, workflow scheduling techniques used for mapping distributed workflows onto computing resources, and summarises related surveys published in this area. Section

2.7 analyses and states the requirements that need to be addressed by modern workflow management systems and workflow languages. Section 2.8 provides a general discussion and identifies the limitations of existing workflow technologies. Section 2.9 finally concludes this chapter by summarising the presented work.

2.2 Service Computing

Service computing is a paradigm for the assembly of application components into networks of loosely-coupled services to create flexible, dynamic business processes, and agile applications that span organisations and computing platforms as defined by Papazoglou et al. in [34]. It can be realised through *Service Oriented Architecture* (SOA) which is defined by the *Organisation for the Advancement of Structured Information Standards* (OASIS) as follows:

“*Service Oriented Architecture* (SOA) is an architectural approach for organising and utilising distributed capabilities that may be under the control of different organisational domains.” [35]

From a software engineering perspective, this approach represents a design pattern for separating concerns. This means that the computation logic required to solve a particular problem can be managed easily when decomposed into a set of smaller components, where each component addresses a particular concern (e.g. part of the problem). Service-oriented architecture is distinguished by the manner in which it achieves this separation. It enables the application logic of distinct pieces of software to be encapsulated within loosely-coupled *web services* as explained by Papazoglou et al. in [34]. However, there is no precise definition of the term “loose-coupling”, and many researchers have attempted to determine the degree (e.g. level) of coupling needed within a software system such as Kaye [36], Chatterjee and Webber [37], Pautasso and Wilde [38]. For matters of simplicity, this thesis adopts the definition provided by Woods and Mattern [39] which states the following:

“the phrase loosely coupled describes enterprise services’ characteristic of interacting in well-defined ways without needing to know each other’s inner workings. This means that the service’s functionality can change without affecting the services that use it, as long as the behaviour described in its interface remains the same – that is, as long as it continues providing the functionality it provides.”

This thesis describes the fundamental concepts embodied in the *Four Tenets of Service Orientation* provided by Don Box [29] to explain service-orientation more clearly. These fundamental concepts include the following:

- **Boundaries are explicit:** Services may be hosted on heterogeneous machines across many geographic locations, and managed by different providers. Such physical boundaries are difficult to cross in terms of performance. Hence, inter-service communication must be distinguished from local method invocations by making these boundaries explicit. For example, the service endpoint location must be publicly accessible to be identified by client applications and remote services. Furthermore, the origin of messages communicated to and from a particular service must be indicated in the content of these messages clearly. This can be useful in implementing message correlation mechanisms for distributed models of computation that involve multiple services.
- **Services are autonomous:** Services are autonomous software components that can control the computation logic they encapsulate. This supports execution transparency as the knowledge of the internal workings of a particular service and its underlying resources is not required to access its functionality.
- **Services share schemas and contracts, not classes:** Services do not share classes publicly but service contracts instead. These contracts are based on structured schemas that describe the service behaviour, its functionality and supported data types. Service providers can advertise these contracts to describe the structure of messages their services can send or receive, and the organisation in which the messages are communicated to and from the services. Furthermore, a service implementation can be

modified without introducing changes to its public contract being shared with other entities (e.g. client applications and remote services).

- **Compatibility is determined based on policy:** Services may have specific requirements, conditions or guarantees (e.g. assertions) that must be true before their operations are executed. Such conditions can be specified in a machine-readable form called a policy, which extends the public service contract. For example, a policy may describe service compatibility in terms of the messages and message exchange sequences it supports. This permits a service functionality to be accessed and executed based on simple propositional logic that determines the service compatibility for certain requirements or conditions.

2.2.1 Web Services

A web service is a platform-independent software component that encapsulates computational logic, and provides functionality that can be exposed to a network by a logical interface. This interface can be specified and published by the service provider to enable consumers (e.g. client applications and other services) to discover the service, and interact with it. Service interaction is based on standards that define communication protocols, and the format of messages communicated to and from the service. Figure 2.1 shows a generic

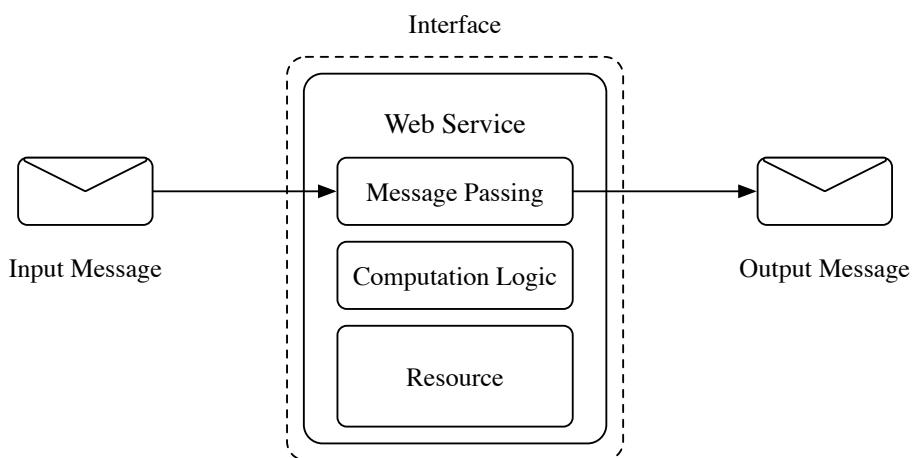


Fig. 2.1 A web service consuming, analysing, and producing messages.

model of a web service which encapsulates computation logic and relies on resources to process incoming (e.g. input) messages and produce outgoing (e.g. output) messages through a message passing facility. Based on this model, a web service can be regarded as a message processor that is capable of consuming, analysing, and producing network messages as argued by Vogels [40]. Papazoglou [41] explains that a service interface provides a set of operations that can be invoked. This permits multiple services to be combined into a single service known as a *composite service*. This can be achieved by passing the output of a particular service operation as an input to another. The technology behind web services has evolved from the continuous improvement of grid-based technology over the past few decades. This section presents the evolution of key technologies that led to the creation and adoption of web services and it discusses the following:

1. General background of early grid-based resource management systems.
2. Development of grid-based service technology.
3. Overview of web service technology proposals.

Early Grid Resource Management Systems

Grid systems emerged in the mid 1990s to meet the needs of the community to share resources, and enable collaboration across organisational boundaries. This has led to the adoption of scheduling systems including *Condor* and its counterparts [42], [43], [44] throughout academia and business. *Condor* [45], a high-throughput distributed batch computing system, was used for creating jobs, and scheduling their execution on computational resources [46], [47]. Jobs represent executable programs that operate over some data. Each job is typically created by the user and passed to *Condor*, which then chooses an appropriate pre-configured machine to execute the job. *Condor* provides the following capabilities:

- **Resource allocation description:** Computational resources are described using a schema-free resource allocation language [48], [49], [50] that permits resource requests (e.g. jobs) to be matched with resource offers (e.g. machines).

- **Job checkpoint and migration:** *Condor* performs periodic monitoring to record checkpoints during the execution of jobs. This provides a form of fault tolerance and safeguards the accumulated computation time of a particular job, and permits a job to migrate from one machine to another to resume its execution upon failure [51].
- **Remote system calls:** *Condor* employs remote system calls for redirecting the inputs and outputs of jobs between machines.

Condor monitors the execution of the job and informs the user of its completion, after which the job is discarded. However, job scheduling systems of this kind are limited by the need to repeatedly transmit the code needed to re-execute each job after it has been discarded, and perform system calls. Furthermore, the user may need to re-configure a hosting machine in order to process a particular job that requires specific software components to be installed (e.g. programming libraries). The key distinction between using job scheduling systems and web service technology, is that a job can be deployed and executed once, whereas a web service is deployed once but its operations can be invoked many times.

Development of Grid Services

Prominent distributed computing projects such as SETI@Home [52] raised awareness in the research community about the power of distributed computing, and the need to share resources across organisational boundaries [53]. This awareness alongside the rapid increase of data needed for scientific experiments, motivated scientists to build a new infrastructure to provide efficient management of compute and data resources. Foster et al. proposed a blueprint for a new computing architecture in [54], [55] which is referred to as the *Open Grid Services Architecture* (OGSA). This architecture's specification defines a set of requirements to support loose-coupling, and interoperability between heterogeneous components in a distributed grid-based system. These requirements state that the basic functionality of a service must be defined to support service re-use in different applications, and the manner in which the service's behaviour and functionality are defined must be clearly expressed. Services must also be identified and treated in a uniform manner, and their communication must be

supported through well-defined interfaces. Furthermore, existing service standards, application platforms, and development tools must be leveraged to facilitate the creation, discovery, and management of distributed services.

The *Open Grid Services Infrastructure* (OGSI) [56] was implemented to support these requirements using *Grid Services*. Frey et al. [57] describe a *Grid Service* as a software component that conforms to a set of conventions which are expressed as interfaces for purposes such as the discovery of service characteristics, lifetime management, and notification. OGSI provides controlled management of distributed grid services using factory and registration facilities for creating and discovering new services, and binding clients with existing services. It relied on the marriage of existing grid-based technologies from the *Condor* and *Globus* [58], [59] projects. *Condor* provides mechanisms for creating jobs, allocating them onto computational resources and managing their execution, whereas *Globus* provides a set of protocols that support communication and standard access to a variety of remote batch systems based on procedural middleware technology [60]. The concept of remote procedural calls has been widely used in distributed computing systems for many years. For example, previous work in this area has focused on remote procedural call mechanisms for tightly-coupled homogeneous processors, and distributed objects including the *Common Object Request Broker Architecture* (CORBA) [61] and *Remote Method Invocation* [62]. GridRPC [63] was introduced to standardise this technology for grid-based scientific applications according to Ho et al. [64], and Shirasuna et al. [65].

The implementation of the OGSI infrastructure relies on an approach that binds application clients with service instances but it presents significant problems. For example, a factory facility creates a service instance based on a client's request, and returns a service *handle* to the client. This handle is then resolved by the client to obtain a service reference (e.g. pointer), which holds binding information required to establish communication with the service and invoke its functionality [56]. This is considered as a rigid approach as the binding between a service and its client relies on a pointer which may become useless if the service becomes fails. Furthermore, this pointer may be used to expose the underlying resources upon which the service operates within the client's application. This contradicts a

fundamental tenet of service-orientation, and has led the research community to revise the design and implementation of this architecture and investigate alternative technology [66]. Banavar et al. [67] and Vinoski [68] argue that procedural middleware technology is inadequate for distributed computing applications. The implementation of procedural calls is based on the assumption that distributed objects are all available for each other to function properly. This can be problematic in large-scale network environments such as the Internet as an object reference may become invalid due to the object's unavailability. Communication channels between objects must also remain open and this introduces additional network traffic. Waldo et al. [69] argue that communication in a distributed system needs to be dealt with intrinsically different from the communication in a single address space.

Web Service Technology Proposals

Parastatidis et al. [70] proposed the *Web Service Grid Application Framework* (WS-GAF) which provides an alternative approach to OGSI that supports loosely-coupled web services. This framework does not pass a service reference to its client, but instead it exposes a web service by publishing its network address (e.g. web service endpoint). Using the service address, a client can communicate with the service by sending it a message directly. This means that the binding between a client and a particular service is transparent, and does not expose the underlying resources used by the service. The emergence of this technology has motivated the creators of OGSI to reconsider its design, which has led to the creation of the *Web Services Resource Framework* (WS-RF) [66] which was proposed by Foster et al. [71]. This framework addresses the relationship between *stateful* resources and web services. It permits a programmer to implement the association between a web service and a particular resource without compromising the ability to implement web services as stateless message processors. This technology bridges the gap between web services and grid-based systems. However, the research community has widely adopted the web service technology (WS-I⁺) recommended by Atkinson et al. [72] which is based on the technology originally proposed by Parastatidis et al. [70] due to its simplicity, and support for *loose-coupling* and existing standards that address the manner in which distributed services and resources are managed.

2.2.2 Web Service Model and Related Standards

The *W3C Web Services Architecture Working Group* [32] defines a reference architectural model for web services, which describes the interactions between web services based on a set of web standards. Figure 2.2 shows the three primary components of this model and the interactions between them. These components include the service provider, consumer, and registry. The service provider is a component that is responsible for deploying a web service and making it available to service consumers. The service consumer is a component that interacts with a web service by invoking its functionality. It relies on the service registry to discover services of interest and locate their endpoints. The service registry is a component that provides a searchable directory of web service endpoints. Service providers use this registry to publish information about active web services they are hosting. The remainder of this section provides a general overview of the web standards used in this architectural model, which include:

1. *eXtensible Markup Language* (XML).
2. *Simple Object Access Protocol* (SOAP).
3. *Web Service Description Language* (WSDL).
4. *Universal Description, Discovery and Integration* (UDDI).

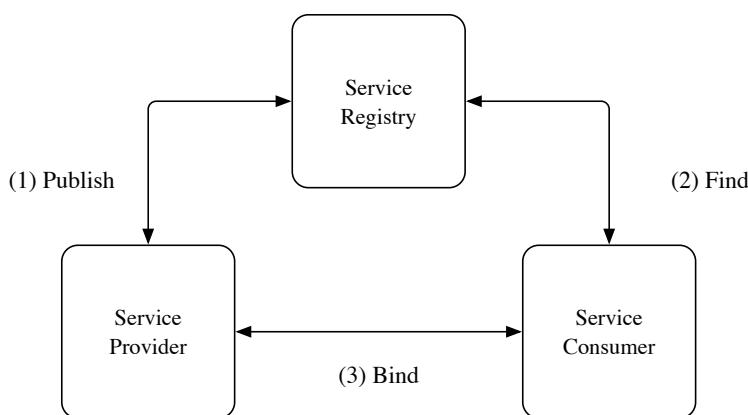


Fig. 2.2 The publish-find-bind web service architecture model.

SOAP

The *Simple Object Access Protocol* (SOAP) [73] is a protocol for exchanging information between computers. It permits a remote procedure call to be specified in an *envelope* that is then encapsulated within a *HyperText Transfer Protocol* (HTTP) [74] message to be transported to a particular service. This permits clients to communicate with remote services and invoke their operations. SOAP messages are platform independent as they are written entirely in XML [75], and are both human and machine readable [76]. This enables heterogeneous distributed applications to exchange data easily. SOAP specifies rules for encapsulating and encoding data that is being transferred including application-specific data and information about accessing the envelope contents (e.g. name of method to invoke, method parameters, or return values). There are different standards of this protocol that include SOAP 1.1 [73] and SOAP 1.2 [77] which are based on XML and XML Information Set (XML-InfoSet) [78] respectively. The implementation of the research solution presented in Chapter 5 supports SOAP 1.1 due to its simplicity and service interoperability support. This thesis does not consider the differences between SOAP 1.1 and SOAP 1.2. It also does not address the performance limitations inherent in the SOAP protocol as they are addressed by Abu-Ghazaleh et al. [79], and Chiu et al. [80].

WSDL

The *Web Service Description Language* WSDL [81] is an XML-based language for describing a web service interface that provides information about the service operations, supported data types, binding details based on specific transport protocols, and the service network address. Such information allows service clients to locate a service and invoke any of its functions. The service interface defines a number of elements which are listed in Table 2.1. These elements are typically structured in a service description document. Figure 2.3 provides a UML diagram showing the relationships between these elements, where a service defines one or more ports, each of which provides a set of operations that consist of input, output, and fault messages. Each message may consist of one or more parts. For example, an input message defines one or more parts that represent parameters to invoke the service.

Each output message has one or more parts that represent the results of invoking a service operation. Each part has a type that is defined based on the XML Schema standard [82]. There are two versions of this standard which are WSDL 1.1 [83] and WSDL 2.0 [84]. The UML diagram shown in Figure 2.3 models the service description document based on WSDL 1.1 because it is more commonly used than WSDL 2.0. There are few toolkits that support the implementation of web services using WSDL 2.0 although it supports RESTful web services [85]. The research solution presented in this thesis supports both standards.

Table 2.1 Elements of a service description document.

Name	Description
Types	A container for data type definitions used by the web service.
Message	A typed definition of the data being communicated.
Port Type	A set of operations supported by one or more endpoints.
Operation	An operation that defines input and output messages.
Binding	A protocol and data format specification for a particular port type.
Service	A container that holds the service name and other elements.

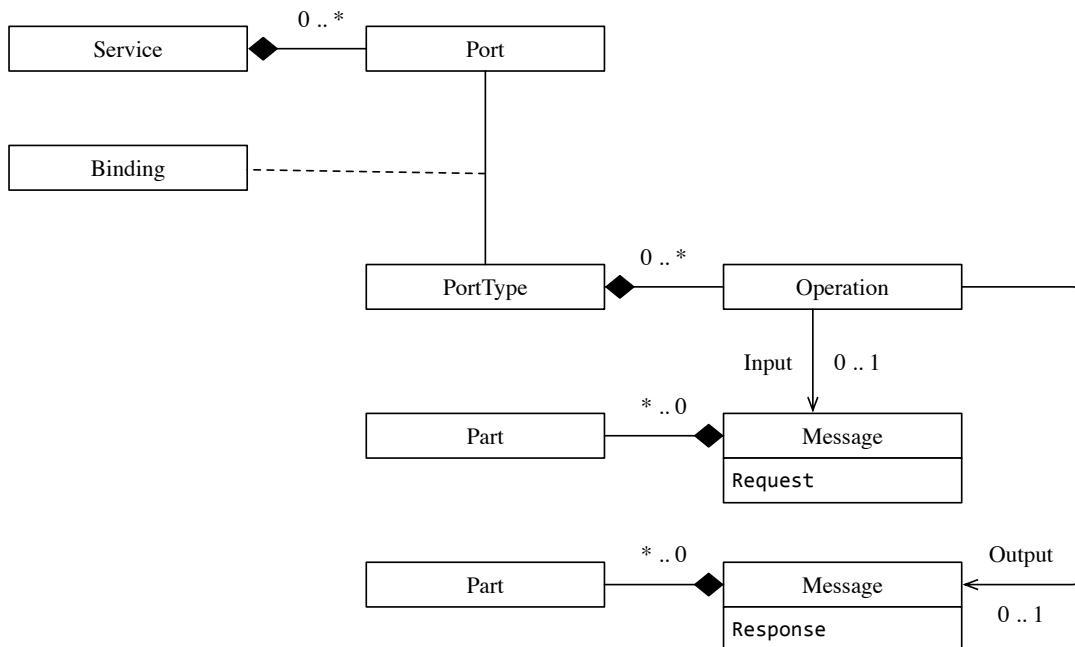


Fig. 2.3 UML diagram of the service description document based on WSDL.

Service operations can be accessed based on pre-defined message exchange sequences between the service and its clients, which are called *Message Exchange Patterns* (MEPs) [86]. Figure 2.4 presents these patterns, which are described as follows:

- **Request-response pattern:** This pattern defines a message sequence in which the service responds to a client's request message with a return or fault message.
- **Solicit-response pattern:** This pattern defines a message sequence in which a service expects a response or fault message from a client upon sending the client a message.
- **One-way pattern:** This pattern defines a unidirectional message exchange behaviour in which a message is sent from a client to a service, but the service is not obliged to respond to the client with a return message.
- **Notification pattern:** This pattern defines a message exchange behaviour in which the service sends a message to an arbitrary client but the service expects no response.

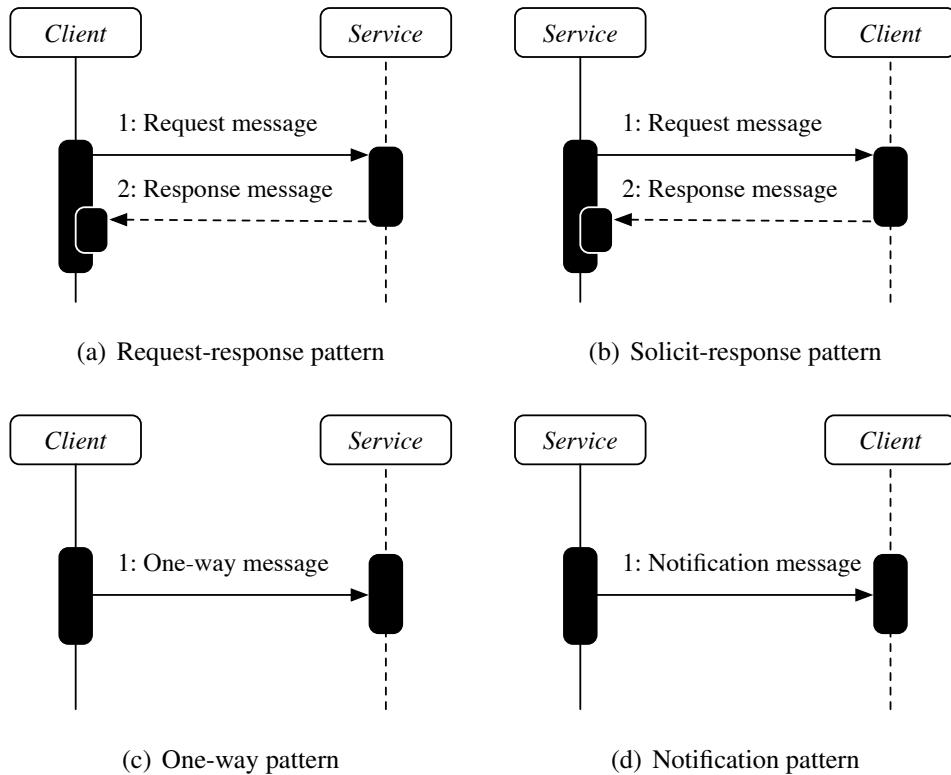


Fig. 2.4 Sequence diagrams of Message Exchange Patterns (MEPs).

Nitzsche et al. [87] discuss the limitations of message exchange patterns, and argue that they are not sufficient for describing the interactions between multiple services. Furthermore, most existing web service technology used for creating web services and client applications such as JAX-WS [88] do not support the solicit-response and notification patterns¹. There are no use case applications that demonstrate the ability of a service to initiate communication with its client. This is because services are agnostic and are not aware of the existence of service clients until these clients communicate with the services by sending request messages.

UDDI

The *Universal Description, Discovery and Integration* (UDDI) [89] represents a technical specification for publishing and finding businesses and web services. This specification states that a centralised *service registry* can be used for storing information about existing services, which enables users to search for and discover services that provide specific functionality. However, this thesis does not focus on service discovery; Dustdar and Treiber [90] explore this area in greater depth.

2.2.3 Central Limitations of the Web Service Model

This section describes the central limitations of the web service architectural model. These limitations include the following:

1. Firstly, the web service architecture model can only support the interactions between a service, its provider and consumer. It does not support service composition in which multiple services interact with each other to provide a new functionality.
2. Secondly, it provides a rigid web service addressing approach that is appropriate in reliable networks only. It does not take into consideration unreliable networks that intrinsically consist of geographically distributed services, which require robust composition and management.

¹See the Java™ API for XML-Based Web Services 2.2 Specification at <https://jax-ws.java.net/>

3. Thirdly, it does not support service composition and orchestration.
4. Fourthly, the wide adoption of this architectural model has created a landscape of web services built based on different requirements.
5. Finally, service providers host many proprietary services that provide different interfaces which increases the difficulty of composing services together.

2.2.4 Web Service Specifications

Web services and their limitations have been the focus of standardisation committees and industrial enterprises over the past few decades [81], [91], [92], [37], [93]. These specifications represent proposals that may become standards, which address the limitations of the web service architecture model. Nezhad et al. [94] reviews many of these specifications. This section only focuses on those specifications that cover issues related to service composition and coordination.

Cabrera et al. [95] proposed the WS-Coordination specification to support the coordination of multiple service interactions. This specification defines a general framework that consists of several elements which include a *coordination context*, an *activation service*, a *registration service*, and a *participant service*. The coordination context element represents an element that is propagated to the distributed participants (e.g. services). The activation service element is used by clients to create a coordination context. The registration service element is used by participants to register resources to be used in the coordination context. The participant service element represents a service that is involved in an activity. There can be multiple services participating in a single activity. Before coordinating a set of participant services, a client must send a request message to the activation service to create a coordination context. This context contains a global identifier, and information about the registration service endpoint (e.g. network address, port reference). Once this context has been created, the client can embed this context within a service invocation message. This permits the service being invoked to find the port reference of the registration service, and register for coordination. However, there are some problems associated with this approach.

It is prone to failure as it is based on a centralised entity (e.g. registration service), and the failure of the registration service can hinder the coordination useless. Salas et al. [96] discuss this problem and attempt to solve it using replicated services to eliminate the dependence on a centralised registry. Furthermore, this approach is intrusive as it requires the modification of services. Each service implementation must be altered to analyse additional information embedded in incoming messages, which indicate some action to be performed.

There are many competing, and overlapping web service specifications that have been created by different organisations to address certain needs. For example, the WS-Transaction proposed by Cabrera et al. [97] to support short-running, and long-running service transactions for business applications, and the *Web Services Composite Application Framework* (WS-CAF) proposed by Bunting et al. [98] to compose and coordinate web services. However, the WS-Transaction is not mature enough to be implemented [99], and WS-CAF relies on a centralised a centralised coordination entity that presents an inherent weakness; if this entity becomes unavailable then the coordination process fails. This thesis does not consider the study of further web service specifications due to problematic issues discussed in [100]; there are no concise guidelines for reviewing web service specifications, and most of which are poorly-written and have no concrete implementations.

2.3 Workflows

Early researchers including Clarence and Nutt [101] attempted to integrate workflows as part of automated information systems in the early 1980s to enhance the overall efficiency of office applications. The business community was the first to adopt workflows, which has led to a space crowded with competing specifications and technologies [102]. This has prompted the *Workflow Management Coalition* (WfMC), a standardisation consortium, to define a workflow as follows:

“A workflow is the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant (a resource either human or machine) to another for action, according to a set of

procedural rules.” [103]

From this definition, a workflow can be regarded as a distributed application in which a structured set of tasks can be specified and executed using computing resources. Most existing workflow management systems [104], [105], [106], [107], [108] used for the specification and execution of workflows mirror a generic architecture that is specified in the *Workflow Management Reference Model* [103] which is discussed in Section 2.3.4. This architectural model and its variants [109] focus on executing business-oriented workflows, and therefore are not suitable for scientific workflows. Business workflows aim to reduce human resources and automate the execution of tasks to support business requirements (e.g. increase revenue), whereas scientific workflows have different goals that include reducing the effort to model and orchestrate an experiment, reducing the computation costs for executing an experiment, and improving the execution performance of experiments. Section 2.3.1 defines and discusses scientific workflows.

2.3.1 Scientific Workflows

Scientific workflows have recently emerged as a new paradigm for scientists to integrate, structure, and orchestrate a wide range of local and remote heterogeneous services and software tools into complex scientific processes to enable and accelerate many scientific discoveries [110]. There are many definitions of what constitutes a scientific workflows provided by Ludäscher et al. [111], and Taylor et al. [112]. This thesis uses a simple definition provided by Romano [113] and iterated by Goble and De Roure [4] which describes a scientific workflow as:

“a precise description of an experiment (e.g. a multi-step process) in which multiple tasks are coordinated based on the dataflow between them.”

Based on this definition, scientists design workflows by precisely describing or modelling data-centric experiments that can be executed to process and analyse data gathered from sophisticated scientific instruments. For example, a scientist can design a workflow

that consists of multiple computational tasks. These tasks can be ordered and interlinked in a specific manner that permits the overall workflow to be carried out by executing each task, and transferring data from one task to subsequent ones. This process continues until there are no more tasks that need to be executed. Each task can only be executed when all the necessary requisites are fulfilled (e.g. inputs), and may produce intermediate outputs (e.g. data used as inputs to execute other tasks) or final outputs. Goble and De Roure [4] consider a workflow task as a computational process that may involve invoking a service over the web to use a remote resource, and that the output data from one task is consumed by subsequent tasks according to a predefined graph topology that “orchestrates” the flow of data.

2.3.2 Design Patterns of Scientific Workflows

Several scientific workflow examples have been presented in Section 1 of this thesis. This section presents a set of design patterns that are used to compose these examples. These patterns are used to organise the order in which tasks are executed, and how they are inter-linked, and may be used to model the behaviour of tasks in the workflow application. For example, the workflow architect (e.g. scientist) can select a specific set of patterns that can be combined to synthesise a solution for a recurring design problem. Scientific workflows are commonly modelled as *Directed Acyclic Graph* (DAG) in which the vertices are tasks with directed edges between them as data dependencies that indicate the data movement in and out of tasks. This requires a set of design patterns which are also known as *dataflow patterns*. Bharathi et al. [12] provides a characterisation of scientific workflows and discusses these patterns which are used to compose them. This section provides a general overview of these patterns which include:

1. The process pattern.
2. The pipeline (sequential) pattern.
3. The data aggregation (fan-in) pattern.
4. The data distribution (fan-out) pattern.

There is a large amount of research work that focuses on design patterns to model both control and data flows in the workflow [114]. van Der Aalst et al. [115], [116] provide design patterns called *workflow patterns* for modelling constraints, and representing control behaviour in the workflow. These patterns provide abstractions for describing control-flow structures such as conditionals, arbitrary cycles (e.g. loops). Similarly, Barros et al. [117] provides a set of *Service Interaction Patterns* that address recurring design issues derived from transactional business process applications. These patterns and are commonly used in business-oriented workflows and their discussion is beyond the scope of this thesis.

Process Pattern

This pattern is used to perform a single computational task which takes one or more inputs and produces a single output. Figure 2.5 provides an example of this pattern, in which the directed arrows represent the data movement into and out of the computational task.

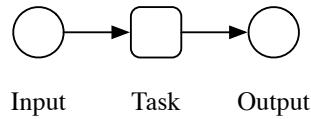


Fig. 2.5 The process pattern.

Pipeline Pattern

This pattern is used for chaining several tasks, where the output of a particular task is passed as an input to a subsequent task in a sequential manner. Figure 2.6 provides an example of this pattern which consists of three tasks including an initial task that accepts input data, an intermediate task, and a final task that produces the workflow output. Based on this example, the output of the initial task is passed directly to the intermediate task which in turn produces an *intermediate output* that is implicitly passed to the final task for further computation.

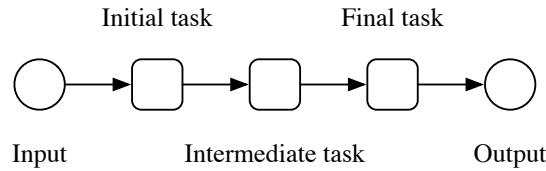


Fig. 2.6 The pipeline pattern.

Data Aggregation Pattern

This pattern is used for collecting outputs from different tasks, and passing them all as inputs to a single task that acts as a data *sink*. Figure 2.7 shows an example of this pattern, in which two tasks pass their outputs directly to a third task, which in turn produces a single output. This pattern can be used to create reductive workflows in which the data is consumed gradually towards a single sink.

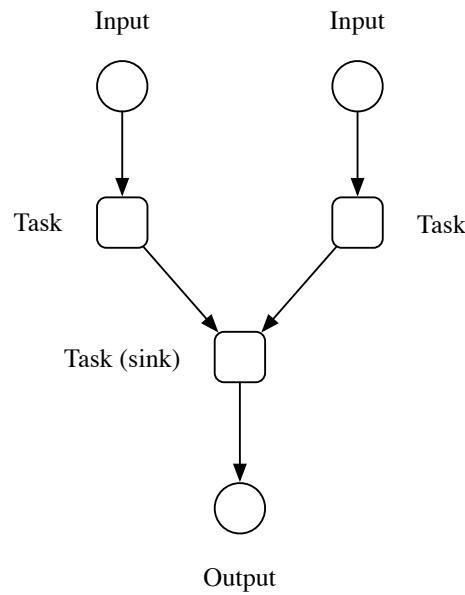


Fig. 2.7 The data aggregation pattern.

Data Distribution Pattern

This pattern is used for distributing data to multiple tasks. Figure 2.8 shows an example of this pattern, in which a single task acts as a data *source* that distributes identical copies of

its output to subsequent tasks. This pattern can be used to create expansive workflows in which more data is produced gradually towards multiple sinks.

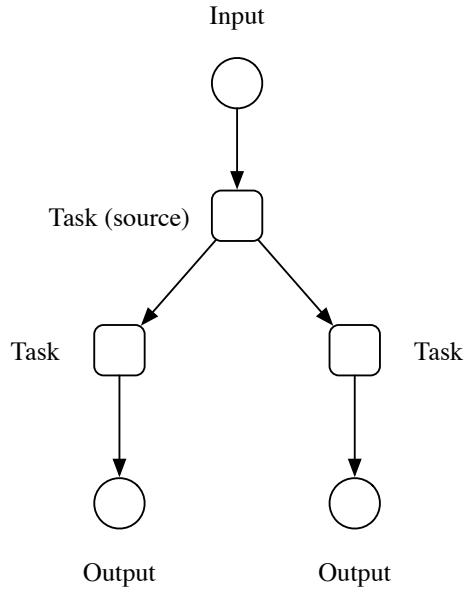


Fig. 2.8 The data distribution pattern.

2.3.3 Lifecycle of Scientific Workflows

The lifecycle of a workflow is important to understand the steps needed to set up and run workflows. Laymann and Roller [118] provide a workflow lifecycle that is widely accepted by the research community, but focuses on business-oriented workflow applications. However, the lifecycle of scientific workflows are distinguished from its business counterpart as explained by Görlich et al. [119]. Unlike business-oriented workflows that may involve different user groups (e.g. business specialists, administrators, and analysts), scientific workflows involve a single user group (e.g. scientists) that play different roles in designing, composing, executing, managing workflows and analysing their results. Scientists are typically interested in a single workflow instance that is developed based on a trial-and-error method. Hence, there is no rigid organisation in the steps required to conduct a workflow as it may be repeatedly modelled and executed using different datasets. There are many studies focused on domain-specific workflows and attempted to define the lifecycle of scientific

workflows. For example, Görlach et al. [119] proposed a scientific workflow lifecycle based on observations relating to the manner in which scientists create and conduct experiments, and from common characteristics of standard data analyses and simulations that are described by Barga and Gannon [120]. This lifecycle consists of modelling, execution, monitoring, and analysis steps. Ludäscher et al. [27] proposed a similar scientific workflow lifecycle that consists of steps needed for workflow design, preparation, execution, and post-execution analysis. This lifecycle requires the datasets needed for the workflow execution to be staged into computational resources before the workflow execution begins. Deelman et al. [121] provides a comprehensive study that focuses on the lifecycle of scientific workflows and highlights the significance of provenance, and the ability to reproduce scientific experiments. Based on this study, this thesis discusses the scientific workflow lifecycle which consists of the following steps:

1. Composition.
2. Resource mapping.
3. Execution.
4. Provenance.

Figure 2.9 provides an adaptation of this scientific workflow lifecycle from Deelman et al. [121]. Typically, a set of different tools are used in handling each step in the workflow lifecycle. Section 2.3.4 provides a general background of workflow management systems, which provide the tools needed for composing workflow, mapping workflow tasks onto computation resources, executing workflow tasks, and capturing provenance.

Composition

Scientists use workflows to precisely compose experiments to process, and analyse data gathered from sophisticated scientific instruments. During composition, workflow tasks are specified and combined from a high-level perspective using a *workflow language*. This step produces a *workflow specification* document encoded in a human-readable format, which

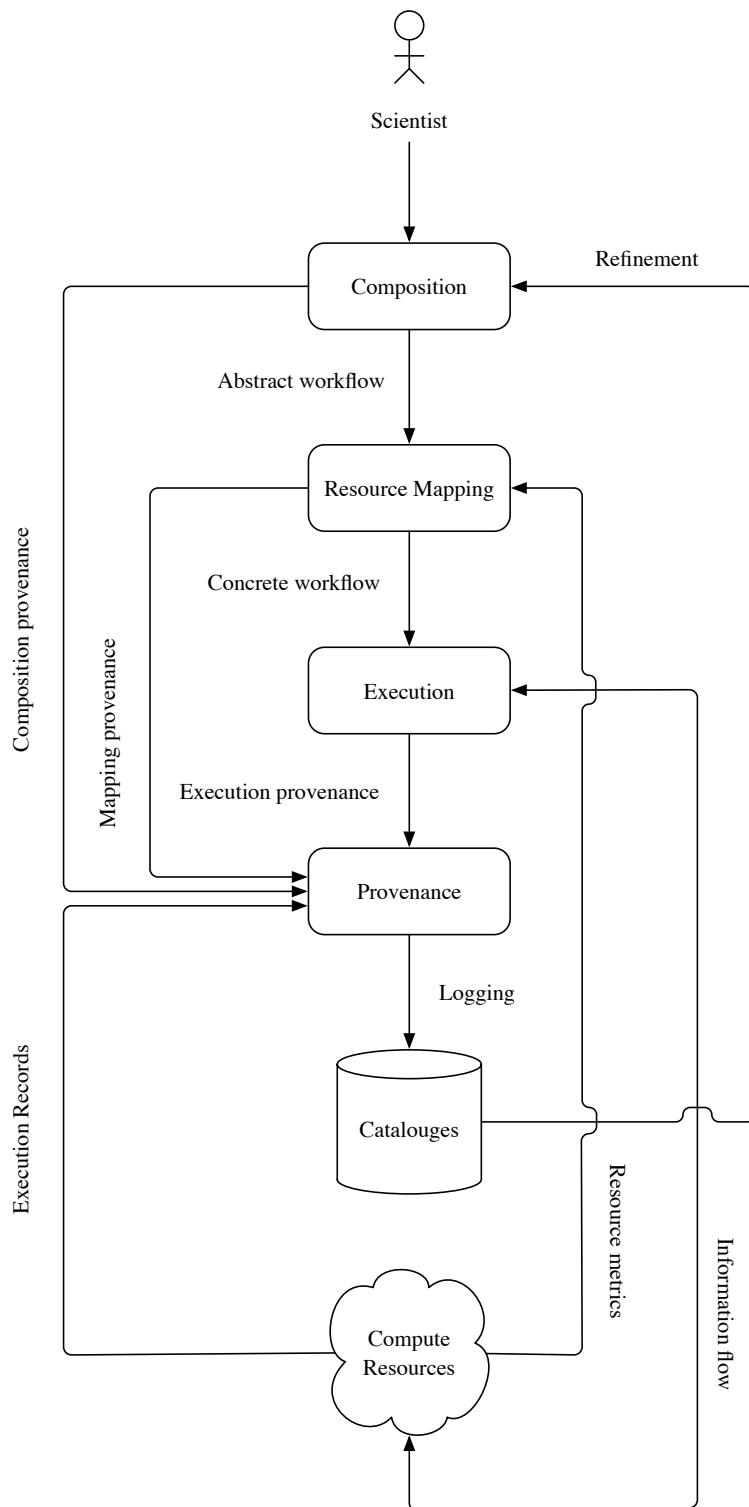


Fig. 2.9 Scientific workflow lifecycle adapted from Deelman et al. [121].

represents a logical model that describes the workflow tasks to be executed, and the order in which to execute them. This logical model is commonly known as an *abstract workflow* [21], [22], [23] because it does not describe the mapping of tasks onto computing resources. Following composition, the workflow provenance is captured and encoded in a suitable format for storage in a *workflow catalogue* or *repository*. This permits scientists to select existing workflows from a storage facility to be used again, or refined by introducing changes to the workflow specification. Scientists may alternatively choose to use workflows from public repositories such as myExperiment [20].

Resource Mapping

Resource mapping is an essential step in the workflow lifecycle because it is responsible for finding a group of appropriate services to execute the workflow tasks. It generates an executable plan called a *concrete workflow* [21], [24], [25] which may affect the overall workflow performance. Typically, a concrete workflow is encoded in a *machine-readable* form that describes the mapping of tasks onto computing resources, and how the data is transferred between the resources to support their collaboration. Resource information may be collected from a *resource catalogue* based on high-level specifications of desired *Quality of Service* (QoS) properties [122] needed to execute the workflow, or may be gathered from available computing resources in the execution environment automatically.

Execution

Based on the outcome of the resource mapping step, the workflow tasks are deployed onto computing resources for execution. Typically, a *workflow engine* is used to manage the overall workflow execution. This engine is responsible for executing tasks in a particular order, and coordinating the data movement between computing resources that execute the workflow tasks. Multiple workflow engines may collaborate with each other to execute the workflow tasks. During execution, a particular engine may be responsible for monitoring the overall workflow execution progress, and the execution environment by collecting information about the computing resources and the network condition. Such information can

be analysed to optimise the workflow performance, and adapt to dynamic changes in the execution environment if necessary. This thesis presents different approaches for executing workflows in Section 2.4, and reviews existing workflow engine technology.

Provenance

Provenance records the history the workflow data including initial inputs, intermediate data, and final outputs. This history can be used to predict and improve the workflow performance before execution using different data sets, and may be used to refine the workflow structure. There are many studies that focus on provenance issues [123], [124], [125], [126] and highlight related challenges [127]. However, the discussion of this area is beyond the scope of this thesis.

2.3.4 Workflow Management Systems

Most existing workflow management systems mirror a generic design model that is specified in the *Workflow Management Reference Model* [103]. This model depicts a workflow management system as having one or more engines that can exploit connectivity to computation services, and coordinating the data movement between them. Figure 2.10 shows an architectural diagram of a workflow management system based on this model, which shows the interactions between the system components and their interfaces. This model provides *workflow enactment service* consists of a single or multiple workflow engines, where each engine provides a run-time environment to execute workflows. Typically, the workflow enactment service presents an interface that permits its workflow engines to interact with other components including process definition tools, client and external applications, administration and monitoring tools, and other workflow enactment services. Based on this model, process definition tools can be used to design and compose the workflow into a specification that can be interpreted at run-time by the workflow enactment service. Client applications can be used to issue requests to the workflow enactment service (e.g. creating, suspending, or terminating workflows), and permit users to obtain certain data relating to the workflow execution from the workflow enactment service. Such data may be ob-

tained by the workflow engine(s) from invoking external applications (e.g. computation resources or data resources), routing data between services, or collaborating with remote workflow enactment services that may be executing different workflows. The administration and monitoring tools permit the user to monitor the progress of the workflow and gather general information about the workflow, and metrics relating to the workflow performance. However, scientific workflow management systems may not adhere to this design model.

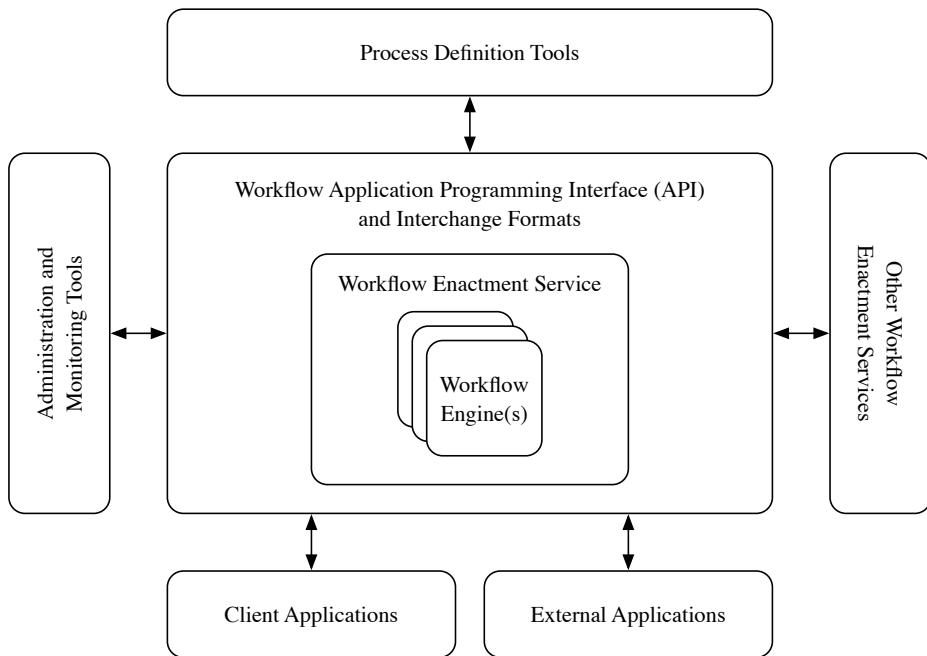


Fig. 2.10 Generic workflow management system.

Deelman et al. [121] state that scientific workflow managements may be based on different execution models, and provide their own execution environment and development kit that enable scientists to conduct their experiments. Such workflow management systems include *Pegasus* [13], *Kepler* [111], *Taverna* [128], *Triana* [129] and others which are reviewed in Section 2.6. Goble and De Roure [4] explain that scientific workflow management systems are designed to handle common concerns such as invoking service applications, providing the ability to interface with different computing platforms, monitoring and recovery from failures, optimisation of storage resources, supporting concurrency, handling data management activities and dealing with heterogeneous data types, etc.

2.4 Service Orchestration

Originally, the term “orchestration” refers to a design pattern that is dedicated to the effective maintenance and execution of business process logic [26]. Peltz [130], [131] describes *service orchestration* as a centralised executable business process that can interact with web services at the message level. Barker and van Hemert [102] provide a similar definition of service orchestration. However, existing definitions are inaccurate and do not capture the true meaning of service orchestration. Therefore, this thesis provides its own definition of service orchestration as follows:

“Service orchestration is an approach for the enactment, and management of a service-oriented workflow that involves the coordination of control, and data messages across distributed web services based on a precise high-level specification of service interactions.”

Service orchestration is typically carried out using a workflow management system. Existing workflow management systems provide their own development environments, and tools for managing the execution of the workflow [121]. The architectural design of these management systems affects the manner in which services are orchestrated. The remainder of this section provides a description of service orchestration system architectures which include centralised and decentralised service orchestration system architectures.

2.4.1 Centralised Service Orchestration System Architectures

Centralised service orchestration refers to an approach for the execution and management of a workflow using a centralised system architecture [132]. There are two kinds of centralised service orchestration system architectures which include:

1. Completely centralised service orchestration system architectures.
2. Hierarchical service orchestration system architectures (e.g. partially centralised).

The remainder of this section describes these architectures, their components and the interactions between them, and highlights the advantages and disadvantages of each architecture.

Completely Centralised Service Orchestration System Architecture

This approach relies on a single compute server (e.g. workflow engine) which is responsible for coordinating control and data flows amongst the services. This engine provides *complete control* over the workflow, supports process automation, and encapsulates the overall workflow logic [26]. This is useful because all the information about the workflow, its tasks and relevant data are persistent and resident at the server that hosts the engine. The engine provides capabilities to interpret a given workflow specification, invoke a set of services, collect the invocation results, and forward these results to other services as necessary. Using a centralised engine, however, is not considered a scalable approach as discussed in Section 1.1. This is because all data and control messages are routed through a single point of coordination which risks in causing the engine to become a performance bottleneck. The advantages and disadvantages of this architecture are identified as follows:

- **Advantages:** This architecture permits the workflow logic, and all system status information to be maintained at a single location which is useful for monitoring the progress of the workflow execution.
- **Disadvantages:** This architecture makes it difficult to change the workflow structure, and employ performance optimisation techniques. The centralised engine also presents a single point of failure (e.g. potential performance bottleneck) in the workflow.

Hierarchical Service Orchestration System Architecture

Centralised orchestration can be partial when it is based on a system architecture that employs centralised control, and distributed data transport mechanisms to execute a workflow. For example, a centralised workflow engine is used to control low-level *workers* to execute

the overall workflow. Each *worker* executes part of the workflow on behalf of the engine by invoking a particular group of services, and collecting the invocation results. Typically, the engine initiates the execution of the workers and receives notification messages from the workers regarding the execution progress of the workflow. For example, a worker can transmit a notification message to the centralised engine which indicates that the worker has completed the execution of a particular set of workflow tasks. Based on the contents of these notification messages, the centralised workflow engine may choose to change the behaviour of the workers. For example, the engine may instruct the workers to forward the results of the executed workflow tasks to particular locations, or execute a new set of workflow tasks based on these results. Multiple workers can collaborate with each other by transferring data directly to locations where they are required, or exchanging references to the data that permit workers to obtain the data from particular network locations when necessary. However, the failure of the central engine will result in entire system failure. The advantages and disadvantages of this architecture are identified as follows:

- **Advantages:** This architecture permits system status information to be obtained from low-level workers, and maintained at a centralised location (e.g. workflow engine). The centralised engine is only responsible for controlling the behaviour of the workers instead of executing the overall workflow logic. This reduces the amount of data to be passed through the engine. Low-level workers have some degree of autonomy, and can collaborate with each other directly.
- **Disadvantages:** Control messages between the engine and the low-level workers (e.g. inter-level communication) may increase and influence the performance of the workers. Low-level workers have limited capabilities to execute the workflow tasks. It is also difficult to employ optimisation techniques to adapt to changes in the execution environment. The centralised engine presents a single point of failure (e.g. the failure of the engine can interrupt the execution of the workers and halt the overall workflow execution).

2.4.2 Decentralised Service Orchestration System Architecture

Decentralised orchestration system architectures consist of multiple engines that may collaborate with each other without the need for a centralised coordination entity. Each engine plays an equal role in the workflow by executing part of it, and may transfer data directly to remote engines that require the data to execute other workflow parts. Unlike centralised and hierarchical service orchestration system architectures, there is no single entity that controls the workflow execution. Decentralised orchestration architectures must guarantee that a sufficient number engines provide the required capabilities that can be provided by a centralised engine. Typically, decentralised orchestration typically employs self-management techniques that determine how to establish communication between engines, and how communication messages are routed across them. However, decentralised orchestration presents a set of research challenges which were described earlier in Section 1.2.2. The advantages and disadvantages of this architecture are identified as follows:

- **Advantages:** There is no single point of coordination that may become a performance bottleneck in this architecture as it relies on multiple collaborative engines. Each engine controls its own behaviour (e.g. local autonomy). Each engine is also identical in design and implementation (e.g provides the same capabilities). This architecture supports optimisation techniques for adapting to changes in the execution environment.
- **Disadvantages:** In this architecture, it is typically difficult to change the workflow structure as its logic is distributed across different engines. There is no standard protocol that describes the manner in which engines collaborate with each other. The engines may operate in an execution environment that may be unstable, and may be running on unreliable hosting machines. It is difficult to implement failure handling mechanisms due to lack of a centralised coordinator, and it is also difficult to distinguish between different types of failures (e.g. engine failure, broken communication link between two engines) during the workflow execution.

2.5 Service Orchestration Cost Models

This thesis presents the cost models for orchestrating a workflow using centralised and decentralised approaches. These cost models are only used to understand the behaviour of a completely centralised service orchestration approach and its scalability problem which was identified in Section 1.1 more clearly, and the differentiate between centralised and decentralised service orchestration. This thesis provides a generalised cost model for orchestrating a workflow using centralised and decentralised approaches, after which it presents a detailed cost model for orchestrating pre-defined patterns of communication that focus on the data movement amongst the engine and service entities. These patterns were discussed earlier in Section 2.3.2. The remainder of this section presents:

1. Mathematical notation for representing service orchestration cost models.
2. Preliminaries.
3. Orchestration cost model for:
 - (a) Centralised service orchestration system architectures.
 - (b) Decentralised service orchestration system architectures.

2.5.1 Mathematical Notation

This thesis introduces the following mathematical notation which is used to provide a concrete representation of the centralised orchestration computation model:

- E : Engine entity.
- S : Service entity.
- $\vec{I}_{x,y}$: Cost of transmitting input data from entity x to entity y .
- $\vec{O}_{x,y}$: Cost of transmitting output data from entity x to entity y .
- $\overleftrightarrow{C}(x,y)$: Round-trip communication cost between entities x and y .

- $\vec{C}(x,y)$: Unidirectional (e.g. one way) communication cost between entities x and y .
- $C_{workflow}$: Total communication cost of orchestrating the workflow.

2.5.2 Preliminaries

Before presenting the general cost model of service orchestration, this section describes the round-trip communication cost for invoking a service using an arbitrary engine entity. This section provides the communication cost for service invocations based on message exchange patterns that include request-response, and one-way service invocation patterns.

Request-response Service Invocation Pattern Cost Model

This communication cost typically involves the cost of transmitting an input to the service, which is used to execute a particular service operation, and the cost of transmitting the service operation's output to the engine. It is expressed in Equation 2.1, where $\vec{I}_{E,S}$ is the cost of transferring the input data from engine E to service S , and $\vec{O}_{S,E}$ is the cost of returning the service invocation output to the same engine. Figure 2.11(a) shows the interactions between the engine E and service S based on this communication cost.

$$\vec{C}(E,S) = \vec{I}_{E,S} + \vec{O}_{S,E} \quad (2.1)$$

One-way Service Invocation Pattern Cost Model

The communication cost of a unidirectional service invocation in which an arbitrary engine invokes a service operation without expecting a response from the service is defined in Equation 2.2, where $\vec{I}_{E,S}$ is the cost of transferring the input data from engine E to service S . Figure 2.11(b) shows the interaction between engine E and service S based on this cost model.

$$\vec{C}(E,S) = \vec{I}_{E,S} \quad (2.2)$$

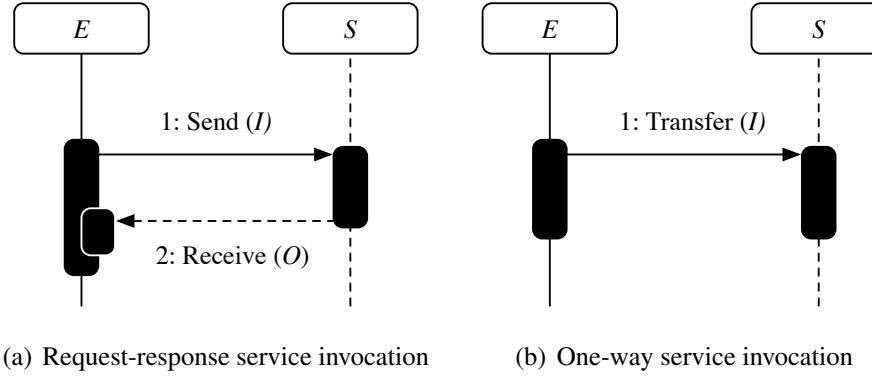


Fig. 2.11 Sequence diagrams of service invocations.

2.5.3 Cost Model for Centralised Service Orchestration

Based on Equations 2.1 and 2.2, a general cost model for orchestrating an entire workflow is devised. This general cost model is shown in Equation 2.3, where E represents the centralised engine, S_i is a service that is expected to respond to the engine where n is the number of services that need to be invoked in a request-response fashion, S_k is a service that needs to be invoked but does not respond to the engine where r is the number of services. In practice, the services are not aware of the notion that they are part of a larger collaboration. The workflow engine invokes each service individually as soon as the data that is required for its invocation becomes available from other services, and multiple service invocations may be carried out in parallel. Hence, centralised orchestration can be expressed generally as the total cost of invoking all the services participating in a workflow.

$$C_{workflow} = \sum_{S_i=1}^n \overleftarrow{C}(E, S_i) + \sum_{S_k=1}^r \overrightarrow{C}(E, S_k) \quad (2.3)$$

The general centralised orchestration cost model provided in Equation 2.3 does not take into consideration the cost of orchestrating common dataflow patterns. This section provides the cost model for orchestrating these patterns using a centralised engine as follows:

1. Centralised orchestration cost model of the pipeline pattern.
2. Centralised orchestration cost model of the data aggregation pattern.
3. Centralised orchestration cost model of the data distribution pattern.

Centralised Orchestration Cost Model of the Pipeline Pattern

The pipeline pattern is used for chaining several services, where the output of a particular service operation is passed as an input to another service operation in a sequential manner. This pattern can be modelled in Equation 2.4, which represents the total cost of all data transfers between an engine E and n number of services.

$$C_{pipeline} = \sum_{S_i=1}^n (\overrightarrow{C}_{E,S_i}) \quad (2.4)$$

Figure 2.12 provides an example of the pipeline pattern, in which an engine (E) is used to invoke services (S_1) and (S_2). It invokes (S_1) with an input in step (1), receives the invocation result in step (2), then it invokes service (S_2) with the result obtained from (S_1) in step (3), and receives the result of this invocation from (S_2) in step (4). In this example, additional transfer costs are incurred from passing the output of service (S_1) to service (S_2) through the engine instead of transmitting the output directly between the services.

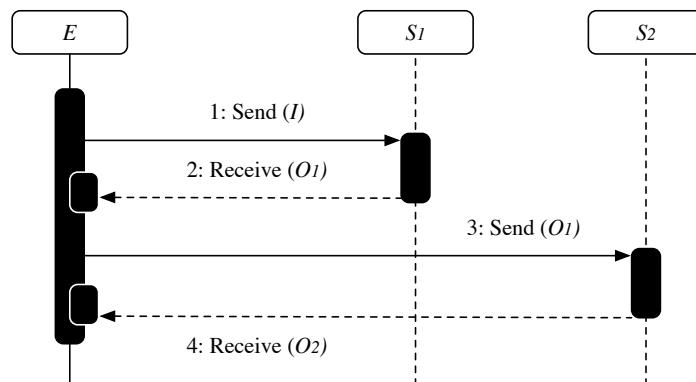


Fig. 2.12 Sequence diagram of the pipeline pattern based on centralised orchestration.

Centralised Orchestration Cost Model of the Data Aggregation Pattern

The data aggregation pattern represents the collection of multiple outputs from different service operations which are all used as input parameters to a single service operation. It can be modelled in Equation 2.5. It represents the total cost of all outgoing data transfers from each service S_i to the centralised engine E , the cost of invoking a service S_δ which acts as a data sink, and n represents the number of services.

$$C_{aggregation} = \sum_{S_i=1}^n (\vec{C}_{S_i,E}) + \vec{\vec{C}}_{E,S_\delta} \quad (2.5)$$

Figure 2.13 provides an example of this pattern. In this example, both services (S_1) and (S_2) provide the engine with invocation results in steps (1) and (4), that are used to invoke service (S_3) in step (5) and retrieve the invocation result in step (6). There are additional transfer costs incurred in this pattern represented by steps (2, 4 and 5). This additional cost could have been avoided if the data transferred from services (S_1) and (S_2) to the engine could have been transferred directly to service (S_3) instead.

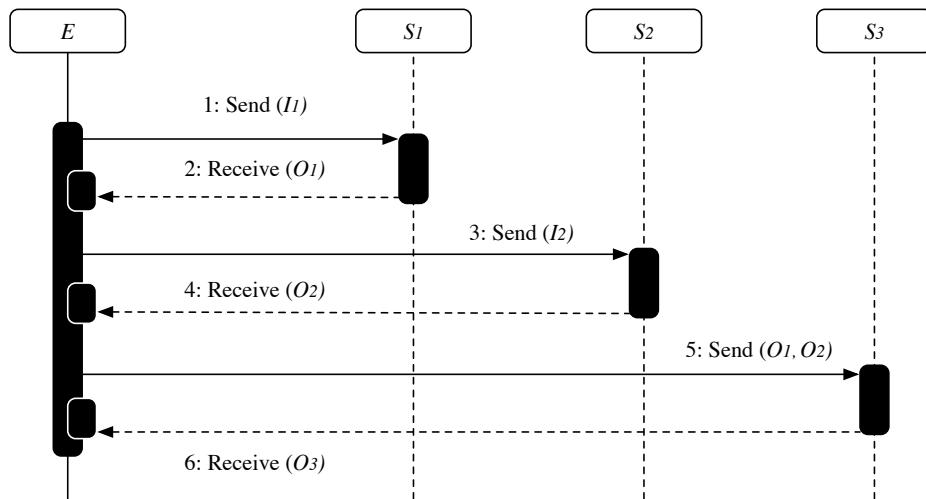


Fig. 2.13 Sequence diagram of the data aggregation pattern based on centralised orchestration.

Centralised Orchestration Cost Model of the Data Distribution Pattern

The data distribution pattern represents the passing of a service operation output to multiple service operations as an input parameter to these operations. It can be expressed in Equation 2.6, which represents the total cost of all data transfers between an engine E and n number of services, in addition to the cost of data transfer from data providing service S_σ to the centralised engine E .

$$C_{distribution} = \vec{C}_{S_\sigma, E} + \sum_{S_i=1}^n (\vec{C}_{E, S_i}) \quad (2.6)$$

Figure 2.14 provides an example of orchestrating pattern, where service (S_1) provides data to the engine (E) in (2), which in turn invokes services (S_2) and (S_3) concurrently in (3) and (5). The invocation results from both services are returned to the engine as demonstrated in steps (4) and (6). Centralised orchestration is responsible for introducing additional data transfer costs in this pattern. For example, instead of passing the data used to invoke the services (S_2) and (S_3) through the engine, the same data could have been transmitted directly from service (S_1) to the services.

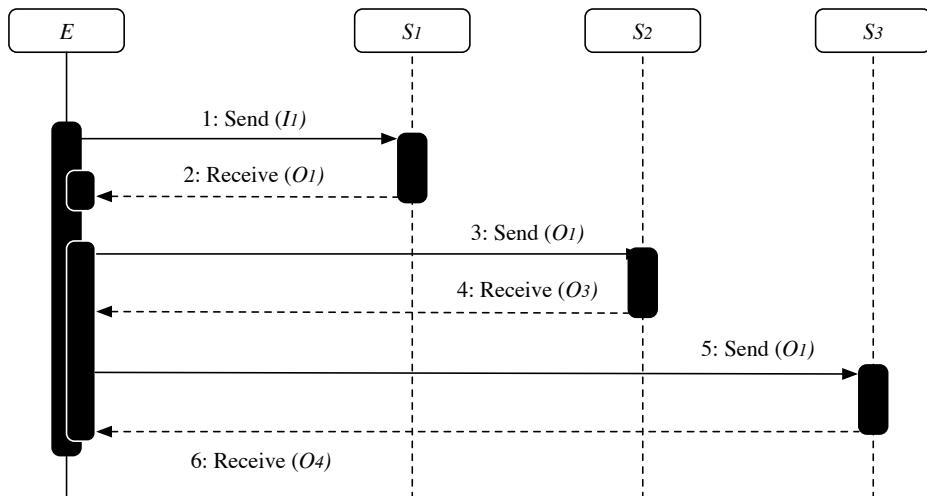


Fig. 2.14 Sequence diagram of the data distribution pattern based on centralised orchestration.

2.5.4 Cost Model for Decentralised Service Orchestration

This section presents a general cost model for orchestrating a workflow in a decentralised manner. This model is expressed using Equation 2.7. Based on this equation, the overall cost of executing a workflow using our decentralised approach represents the total cost of executing each workflow partition (e.g. sub workflow) represented by P_i , and the cost of routing data between the distributed engines that execute these partitions. The communication cost for executing a workflow partition using a single engine is expressed in Equation 2.8, where as cost of communication between the engines is expressed in Equation 2.9.

$$C_{workflow} = \sum_{P_i=1}^n (C_{partition}(P_i) + C_{routing}(P_i)) \quad (2.7)$$

$$C_{partition}(P) = \sum_{S_i=1}^n \overrightarrow{C}_{invocation}(E(P), S_i) + \sum_{S_k=1}^k \overrightarrow{C}_{invocation}(E(P), S_k) \quad (2.8)$$

$$C_{routing}(P) = \sum_{E_k=1}^m (\overrightarrow{C}_{E(P), E_k}) + \overrightarrow{O}_{E(P), E_\delta} \quad (2.9)$$

Based on Equation 2.8, the cost of executing a workflow partition, where P is the workflow partition which is executing using engine $E(P)$. The specification of this partition defines a group of services (e.g. S_i and S_k) to be invoked by the engine. The total cost of transferring the intermediate workflow data from engine $E(P)$ to each remote engine E_k that requires the data is expressed by \overrightarrow{C} , in addition to transferring the partition's execution outputs from engine $E(P)$ to an engine that acts as an ultimate data sink E_δ as shown in Equation 2.9. The general decentralised orchestration cost model provided in Equation 2.7 does not take into consideration the cost of orchestrating common dataflow patterns. The remainder of this section discusses the following decentralised orchestration cost models:

1. Decentralised orchestration cost model of the pipeline pattern.
2. Decentralised orchestration cost model of the data aggregation pattern.
3. Decentralised orchestration cost model of the data distribution pattern.

Decentralised Orchestration Cost Model of the Pipeline Pattern

The pipeline pattern is used for chaining several services, where the output of particular service operation is passed as an input to another service operation in a sequential manner. This pattern can be modelled in Equation 2.10 for decentralised orchestration, which represents the addition of the data transfer costs for invoking the first service S_1 by the initial engine E_α , invoking each service S_i by its engine $E(S_i)$, forwarding the intermediate data between engines $E(S_i)$ and $E(S_{i+1})$, and returning the final output of the workflow from engine $E(S_n)$ to the final engine E_ω which represents the ultimate data sink.

$$C_{\text{pipeline}} = \vec{C}_{E_\alpha, S_1} + \sum_{S_i=1}^n (\vec{C}_{E(S_i), S_i}) + \sum_{S_i=1}^{n-1} (\vec{C}_{E(S_i), E(S_{i+1})}) + \vec{C}_{E(S_n), E_\omega} \quad (2.10)$$

Figure 2.15 provides an example of orchestrating this pattern, which involves two engines and two services. Engine (E_1) transmits sends a request message containing input (I) to service (S_1). In this example, service (S_1) responds to engine (E_1) with data (O_1) in (2), which in turn passes (O_1) as an input to a remote engine (E_2). Engine (E_2) invokes service (S_2) with the data (O_1). The invocation result of service (S_2) is then forwarded by engine (E_2) to engine (E_1).

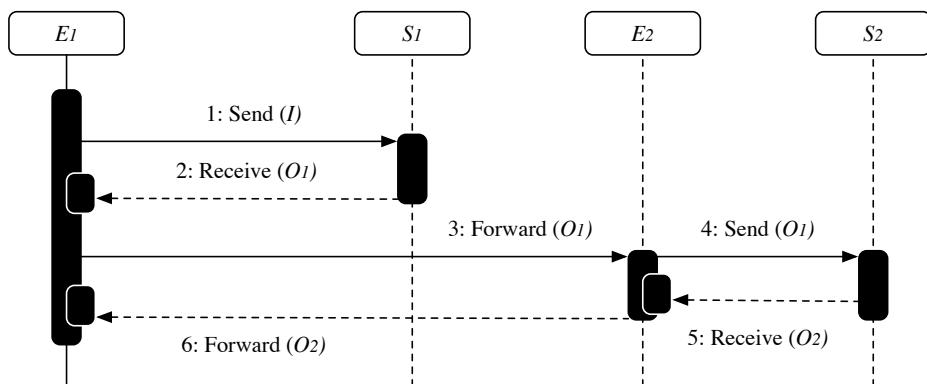


Fig. 2.15 Sequence diagram of the pipeline pattern based on decentralised orchestration.

Decentralised Orchestration Cost Model of the Data Aggregation Pattern

This pattern represents the collection of multiple outputs from different service operations which are all used as input parameters to a single service operation. This pattern can be modelled in Equation 2.11 for decentralised orchestration, which represents the sum of data transfer costs for invoking each service S_i by engine $E(S_i)$, the sum of forwarding the invocation results from each engine $E(S_i)$ to engine $E(S_\delta)$, the cost of invoking service S_δ by engine $E(S_\delta)$, and returning the invocation result finally to engine E_ω .

$$C_{aggregation} = \sum_{S_i=1}^n (\overrightarrow{C}_{E(S_i), S_i}) + \sum_{S_i=1}^n (\overrightarrow{C}_{E(S_i), E(S_\delta)}) + \overleftarrow{C}_{E(S_\delta), S_\delta} + \overrightarrow{C}_{E(S_\delta), E_\omega} \quad (2.11)$$

Figure 2.17 provides an example of orchestrating this pattern, where services (S_1) and (S_2) provide data to the engine (E_1), which in turn forwards this data collection to a remote engine (E_2) that invokes service (S_3). The invocation result of service (S_3) is then forwarded by engine (E_2) to engine (E_1).

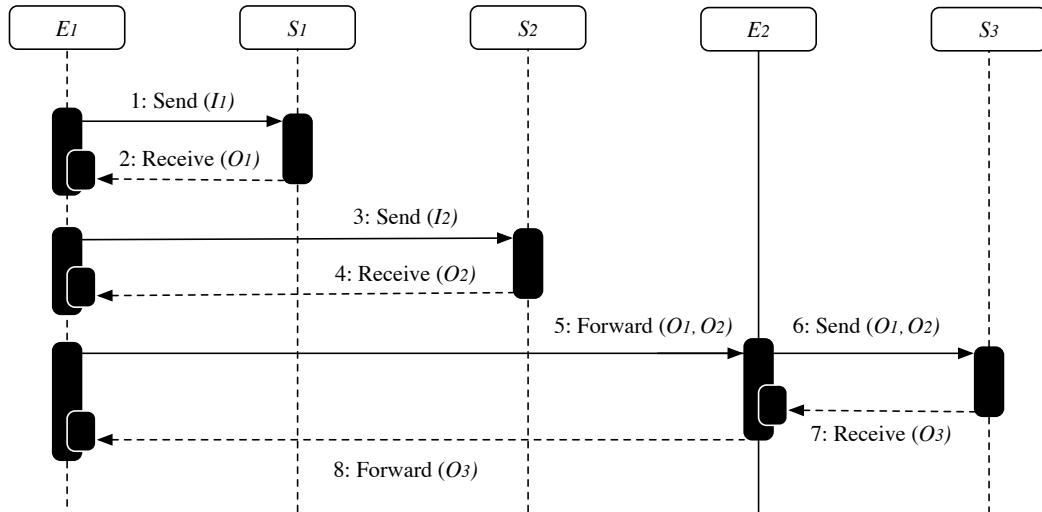


Fig. 2.16 Sequence diagram of the data aggregation pattern based on decentralised orchestration.

Decentralised Orchestration Cost Model of the Data Distribution Pattern

This pattern represents the passing of a service operation output to multiple service operations as an input parameter to these operations. This pattern can be modelled in Equation 2.12 for decentralised orchestration, which represents the addition of the data transfer costs for invoking the source service S_σ by engine $E(S_\sigma)$, forwarding the invocation result from engine $E(S_\sigma)$ to every other engine $E(S_i)$, invoking every service S_i by its engine $E(S_i)$, and returning the invocation result of each service S_i to the final engine E_ω .

$$C_{distribution} = \overleftarrow{C}_{E(S_\sigma), S_\sigma} + \sum_{S_i=1}^n (\overrightarrow{C}_{E(S_\sigma), E(S_i)}) + \sum_{S_i=1}^n (\overleftarrow{C}_{E(S_i), S_i}) + \sum_{S_i=1}^n (\overrightarrow{C}_{E(S_i), E_\omega}) \quad (2.12)$$

Figure 2.17 provides an example of orchestrating this pattern, where service (S_1) provides data to the engine (E_1), which in turn forwards this data to a remote engine (E_2) that distributes identical copies of the data to services (S_2) and (S_3). The invocation results of these services are then forwarded by engine (E_2) to engine (E_1).

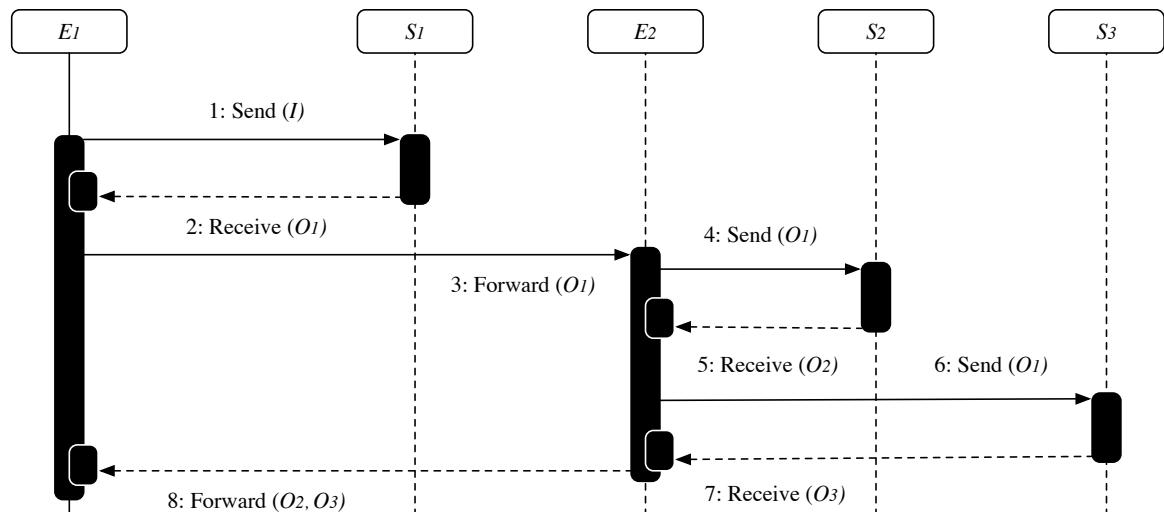


Fig. 2.17 Sequence diagram of the data distribution pattern based on decentralised orchestration.

2.6 Review of Workflow Technologies

Researchers have studied workflow management systems and discussed their importance in the scientific domain [4], [119], and have examined their challenges [18], [133], [134]. These systems aim to support the collaboration between scientists by providing software instruments that can be used to compose workflows and execute them automatically. This section provides a comprehensive review of existing workflow technology. Such technology includes specification languages used for describing scientific workflows, orchestration system architectures used for mapping workflow tasks onto computation resources, executing the workflow tasks and monitoring the execution of the workflow. In addition to dataflow optimisation system architectures that employ scheduling, and data routing techniques to improve the performance of executing workflows. The remainder of this section reviews the state-of-the-art in scientific workflow technology from different application domains.

2.6.1 Workflow Management Systems

There are many workflow management systems, but it is not feasible to review all of these systems due to time constraints. This section therefore reviews notable, and well-established workflow management systems such as *Sedna* [135], *Taverna* [136], *Kepler* [111], *Triana* [137], *Pegasus* [13], *ASKALON* [138], and others. It also provides a review of workflow specification or execution languages that are related to these workflow management systems.

Sedna

Sedna [135] represents an example of a scientific workflow project that relies on the BPEL [139] language discussed in Section 2.6.2. It aims to simplify the process of constructing workflows by domain experts with limited technical knowledge. This project has resulted in an open-source graphical-based editor called the BPEL Designer [135] that can be used to compose service-oriented workflows. Goderis et al. [140] highlight the significance of *Sedna* project in orchestrating hundreds of concurrent services for chemistry applications. Existing scientific workflow management systems need to meet a set of requirements im-

posed by different domains of science. Hence, they are often built from scratch and do not depend on technology that is established in the business domain. Barker and van Hemert [102] highlight this issue and argue that business-oriented workflow technology provide rich constructs to compose workflows but are difficult to use by scientists due to their complexity. Scientists therefore require a higher level of abstraction for composing workflow tasks together to solve a research problem.

myGrid and Taverna

Bioinformaticians often conduct *in silico* (e.g. dry lab) experiments by composing different analytical tools and distributed data resources manually, and performing low-level tasks relating to the configuration of these tools and the preparation of data needed to run their experiments. This methodology is inefficient as scientists need to compose and orchestrate scientific experiments easily without dealing with technical details relating to the execution of the experiments. The *"myGrid* [141] project team has developed an open source service-oriented middleware called *Taverna* [136] to support such experiments on grid-based infrastructures. *Taverna* is a workflow management system that provides a workbench for visually creating, editing and browsing workflows. This workbench permits external web service description documents to be imported and used in workflows directly, and provides an *enactor* that coordinates the execution of workflows.

Taverna allows the user to capture details about the workflow execution including the services used, and the results of these services. It uses an XML-based proprietary language, the *Simple Conceptual Unified Flow Language* (SCUFL) [142], [136]. This language permits a scientist to define a workflow involving a group of local or remote services which can be wired together using data and control links. The language abstractions allow inputs and outputs to be defined and used in the workflow such that inputs can be obtained from *sources*, whereas outputs are provided to *sinks*. Each source or sink resource has a unique name, and can be associated with metadata to aid the visualisation of the workflow. The basic computation units in this language are called *processors* that can be regarded as func-

tions, each of which may accept a set of inputs and produces a set of outputs using specific ports. The language provides *data links* to compose different ports, and *control links* to organise the computational units. Typically, the execution of a workflow based on SCUFL begins from the sources and completes when all the sinks have successfully produced their outputs or failed to do so. *Taverna* provides the user with the ability to select specific parts of the workflow to be executed. However, *Taverna* executes a workflow using a centralised engine that represents a performance bottleneck.

Triana

Triana [137] is an open-source problem solving environment developed for the GridLab [143] project. It provides a graphical modelling environment for constructing workflows with no explicit support for control structures [129]. This modelling environment permits users to drag programming components called *units* or tools onto a workspace canvas. These components can be wired together using *data links* and *control links*. *Triana* uses plugins to support different languages such as BPEL. Its execution model is capable of distributing parts of the workflow to remote machines using a peer-to-peer network.

Triana's execution model is capable of distributing parts of the workflow to remote machines using a peer-to-peer network based on the JXTA [144] project architecture which supports the discovery of available peers and resources on the network, sharing files with peers, managing groups of peers and supporting their communication in a secure manner across different networks. The workflow distribution mechanism is achieved by migrating the code required to execute part of the workflow to a resource where execution is desired. This is achieved by discovering peers that offer particular computational capability, and then downloading executable code to these peers to be run [145], [146]. However, *Triana* has a number of drawbacks. Firstly, *Triana* is implemented in Java but makes use of JXTA libraries that are used to build a significant layer of services to enable the distribution of executable workflow parts and the collaboration between peers. Hence, it is not lightweight solution for executing workflows according to Jurczyk et al. [147] that

may require deployment and configuration of *Triana*'s JXTA peer-to-peer components on hosting machines before execution. Secondly, *Triana* does not provide compact mechanisms for expressing workflows with complicated structures according to Fahringer et al. [138]. Finally, *Triana* performs scheduling during run-time to determine where to execute parts of the workflow, and it does not provide performance optimisation estimates according to Fahringer et al. [138]. This thesis presents a lightweight solution for orchestrating service-oriented workflows. This approach is non-intrusive to the implementation of the services involved in the workflow, and supports the decomposition of the workflow logic into smaller partitions (e.g. sub workflows) for execution using lightweight execution engines at appropriate locations within short network distance to the services. These locations are determined based on placement analysis performed before the execution of the workflow. Unlike *Triana* which performs scheduling at run-time, the solution proposed by this thesis does not require scheduling as it is based on a data-driven execution model. This allows each workflow task to be executed as soon as the input data that is required for its execution becomes available from remote services and resources.

Kepler

Kepler [111] is an open-source workflow project that aims to build a system for sharing computation services and data storage resources to compose scientific workflows. It is built upon the *Ptolemy* II system [148] developed at the *University of California at Berkeley*. *Ptolemy* II is based on an *actor-oriented* design model in which actors are independent computational component (e.g. services) that provide some functionality and can be reused. In this model, *actors* interact with each other through message passing using well-defined interfaces (e.g. ports). For example, actors can consume data from *inports* and produce data to *outports*. These actors can be specified using the *Modeling Markup Language* (MoML) [149]. *Kepler* separates the communication of between actors from their coordination using a *director* component. Hence, the actors define the computational tasks that need to be executed whereas the directors determine when the computation happens during the workflow execution. This supports reusability as the same workflow can be orchestrated using

different execution semantics. *Kepler* provides the ability to compose workflows using a graphical user interface by dragging components onto a canvas and wiring them together. Altintas et al. [150] discuss *Kepler*'s ability to monitor the workflow execution with access to the provenance catalogues. However, *Kepler*'s workflow execution mechanism is centralised.

Pegasus

Pegasus [13], [22], [24] is a notable workflow planner for scientific workflow applications in astronomy, biology, etc. It has no capability to execute workflows but can run on the *Condor* [45], [47] workflow engine. *Condor* provides a *High Throughput Computing* (HTC) environment based on a collections of distributed computing resources, and uses the *Directed Acyclic Graph Manager* (DAGMan) [151] as a meta-scheduler for executing distributed jobs. DAGMan enables *Pegasus* can discover available machines for the execution of a particular job, and performs matchmaking to select resource machine for executing a job if it satisfies some performance requirements.

Pegasus translates abstract workflows composed of tasks (e.g. application components) and their data dependencies into concrete workflows that represent the mapping of tasks to resources [21]. These concrete workflows are then passed to a centralised workflow engine for execution. It generates concrete workflows based on user demands for certain data products. This is achieved by querying a virtual data catalogue [152] for available application components that can produce the required data, the initial inputs and intermediate data replicas in the Grid. Searching for and composing application components presents a significant challenge as it is difficult to capture the functionality provided by these components and the data types being used [136]. *Pegasus* uses an approach to decompose a workflow into a sequence of *layer-partitioned* sub workflows, each of which can be scheduled for execution only if its immediate predecessor has completed its execution. There are some drawbacks of *Pegasus*. *Condor*'s computational jobs and their requirements must be specified manually by the user and pre-knowledge about the resource machines and the network condition is

also required to do so. *Pegasus* may produce inefficient scheduling plans as demonstrated in [153]. This thesis presents a solution that handles the partitioning and mapping of workflows onto machines automatically by collecting information about the network condition, and performing placement analysis. Hence, no pre-knowledge about the resource machines or the network condition is required during the design of the workflow. Furthermore, no scheduling mechanism is required for executing the workflow.

ASKALON

ASKALON [138] is a workflow execution environment that was created to simplify the development and optimisation of application that can harness grid-based and cloud-based computing resources. It relies on the *Abstract Grid Workflow Language* (AGWL) [154], which is an XML-based language that is used to describe abstract workflows. This language permits the user to compose atomic units of work known as *activities*, which are interconnected using control and data flow dependencies. Control flow structures that can be specified using basic constructs such as *sequences*, *for*, *foreach*, *while* loops, and *switch*, and advanced constructs such as *parallel*, *parallel loops*, and *collection iterators*. For any pair of activities, the dataflow between them can be specified by connecting the output of one activity to the input of the other. This language supports the specification of *constraints* or *properties* to activities, which provide functional and non-functional information that may be used to optimise the workflow execution, or describe the workflow activities respectively. ASKALON also supports graphical specification of workflows based on the UML activity diagram which are translated to workflow specifications based on AGWL [155].

The ASKALON execution environment provides a grid-based resource manager, a scheduler, and performance prediction components. The resource manager handles the allocation of available resources and deployment of services onto these resources. The scheduler is responsible for mapping workflows and monitoring their execution, while the performance prediction component estimates the execution time of the workflow. These components provide cooperate together to optimise the workflow performance by adjusting the execution

plans based on Quality of Service (QoS) information obtained from the execution infrastructure [153]. However, the main drawback of using AGWL and ASKALON is that it is difficult to detect parallelism in the workflow efficiently before its execution. The language permits the specification of loops that may be evaluated differently during run-time, and therefore the exact number of parallel activities to be executed cannot be identified before the execution of the workflow. These parallel activities must be determined before the execution of a distributed workflow to enable the scheduler to produce an efficient execution plan. This thesis presents a solution that determines parallelism in the workflow automatically during the compilation of the workflow specification, and its execution model does not depend on a scheduling mechanism.

ICENI

The *Imperial College e-Science Networked Infrastructure* (ICENI) [156] is middleware for the management of service-oriented resources to support administrators, application developers, and end users to compose and orchestrate workflow applications. ICENI provides a graphical toolkit to compose services which supports *spatial* and *temporal* composition. Spatial composition permits all the components that make up an application to be displayed simultaneously, with information that describe how they interact with each other. Temporal composition permits components to be ordered with based on their explicit dependencies. Each component includes graph-based representation in which the directed edges represent temporal dependence and each vertex represents some computation. Once the components and their links are designed, an *execution plan* can be generated and orchestrated using a scheduling service according to Furmento et al. [157], McGough et al. [158], and Young et al. [159]. This scheduling service is responsible for matching specified components with their implementation, and mapping them onto resources. ICENI permits running application components to be monitored, and captures performance data related to these components and stores them in a repository system. Such data can be used by schedulers in the future to estimate the execution times of each component. Mayer et al. [156] discuss the features of this system in more detail. However, ICENI relies on a centralised mechanism to orchestrate

workflows, and the user (e.g. scientist) must indicate manually the QoS constraints required to execute a particular workflow.

Swift

Swift [160] is a workflow management system that was originally designed to automate the processing of large datasets used in high energy physics experiments. It relies on a scripting language called *SwiftScript* [161] for composing workflows, and the *CoG Karajan* [162] run-time system for executing them. The *SwiftScript* language provides rich abstractions, built-in functions and data types to support data-oriented distributed workflow applications. It permits the specification of applications and their interactions to process collections of files, and provides the ability to map the contents of a file system into variables in the language. Furthermore, it supports iterations and branching control-flow structures. *Swift* is distinguished from other languages by its support for implicit parallelism, and location transparency. It automatically determines the parts of the workflow that need to be executed in parallel, selects execution *sites*, and handles data staging activities, and oversees the overall execution of the specification workflow. Similar to the approach presented in this thesis, the *SwiftScript* language supports a data-driven execution model that generates concurrent tasks by interpreting the workflow script (e.g. specification), and distributing these tasks onto sites for execution where each task is executed as soon as the inputs required for its execution become available. However, *Swift* is not directed at service-oriented workflow applications and focuses on coordinating the execution of legacy applications coded in various programming languages. Furthermore, it permits users to include implementation-specific details that makes the language complicated and reduces its platform independence.

DynaFlow

DynaFlow [163] is a peer-to-peer, agent-based system architecture that aims to support dynamic workflows that consist of a set of computational *activities*. Each peer can assume a specific role in the workflow as a *publisher* or an *executor*. These peers are described as follows:

- **Publisher peer:** This peer is responsible for defining a set of activities, and is capable of publishing these activities to neighbouring peers.
- **Executor peer:** This peer volunteers to execute one or more available activities.

Each peer is managed by a group of agents that control the workflow. However, the current implementation of this system architecture permits only a single peer to be exhibit the role of a workflow *publisher*. This peer may become a potential centralised performance bottleneck. The authors of this work recognise this problem and suggest that multiple peers can become *publishers* [164], but highlight related research questions that need to be addressed to support multiple publisher peers. These questions include for example how to identify peers that might share a workflow, how to define a criteria for the selection of executors for a workflow, or the execution order of the workflow activities. There is no evidence of the authors attempt to support multiple publisher peers. This thesis provides a solution that attempts to solve these questions that relies on a high-level language for the specification of peers (e.g. services and engines) that may be involved in a workflow, and relies on a computation placement algorithm that determines a set of workflow engines to execute specific parts of the workflow based on a data-driven execution model.

GridFlow

GridFlow [165] is a workflow management system that relies on an agent-based methodology for managing grid-based which is described by Cao et al. [166], [167], and [168]. It integrates a resource scheduling systems based on the work by Spooner et al. [169] that utilises an iterative heuristic algorithm to reduce the idle time of a particular grid resource, and relies on performance prediction capabilities introduced by Nudd et al. [170]. *GridFlow* provides different layers of abstractions for the specification of a distributed application that include *tasks*, *sub workflows*, and *workflow*. These layers are described as follows:

- **Tasks layer:** This layer defines a set of tasks that represent the smallest workflow elements. These tasks are *Message Passing Interface* (MPI) [171] jobs that run on

multiple machines and may be responsible for transferring data between different machines.

- **Sub workflow layer:** This layer defines a set of sub workflows. Each workflow represents a group of tasks that may be executed in a specific sequence on computation resources in a local cluster environment.
- **Workflow layer:** This layer defines a workflow which consists of several *activities*, each of which represents a sub workflow. These activities are loosely-coupled and may be executed using different computation resources.

This workflow management system provides a graphical modelling tool that facilitates the composition of workflows and supports access to grid-based resources. The execution model of this workflow management system relies on a centralised manager that is responsible for controlling the workflow execution, and assigning sub workflows to low-level schedulers as explained by Yu and Buyya [172]. Each low-level scheduler is responsible for mapping tasks in a sub workflow onto resources for execution. This execution model permits different scheduling policies can be employed by the centralised manager for mapping the sub workflow tasks onto resources by low-level schedulers for execution as described by Hamscher et al. [173]. However, the failure of the centralised manager in this execution model will result in entire system failure.

2.6.2 Workflow Languages

This thesis has so far reviewed a number of workflow management systems in Section 2.6.1, and workflow languages specially designed to support these systems such as SCUFL [136], MoML [149], *SwiftScript* [161], and AGWL [154]. This section describes early grid-based and service-oriented workflow languages. These languages include WSFL [174], XLANG [175], BPEL [139], GSFL [176], GWEL [177], GEL [178], GALE [179], WS-CI [180], WS-CDL [181], and YAWL [182]. These languages may provide support for graphical modelling tools that permit workflows to be composed by dragging and dropping different

software components onto a canvas and connecting them. Some of the languages described in this section may not be popular in the research community, and may not be currently used in business or scientific applications.

WSFL

The *Web Services Flow Language* (WSFL) [174] is based on XML and it was proposed by IBM in 2001 for describing service composition. It provides abstractions for organising business process activities and relevant data exchanges based on *flow* and *global* models which are described as follows:

- **Flow model:** This model represents the execution sequence of the business process activities and the data exchange between the web services involved in these activities. It uses data and control *links* as constructs to separate data from control in service interactions.
- **Global model:** This model represents how the web services interact with each other. These interactions are modelled as links between the service endpoints, where each link corresponds to the interaction between one web service with another's interface based on WSDL.

This language supports nested block structures and the iteration of an activity until it meets an *exit* condition. WSFL was adopted initially to compose and execute workflow applications in grid-based systems. However, it was no longer a suitable choice for describing workflows as they became more complicated [142].

XLANG

XLANG [175] a language based on XML that was proposed by *Microsoft* in 2001. This language describes business processes and the interactions between service providers, and relies on the service interfaces based on WSDL to compose them together. It supports block structures with basic control flow constructs such as *sequence* for sequential routing, *switch* for conditional routing, *while* for cycles, *all* for parallel routing, and *pick* for racing

conditions that are based on timers or triggers. Furthermore, it supports exception handling and long-running transactions. However, this language is no longer used for describing workflows as it has been abandoned by the business community. Some of the language features provided by this XLANG and WSFL have been combined in the design of the BPEL [139] language.

BPEL

The business community was the first to adopt the use of workflows. This has led to the creation of different workflow technologies based on existing standards, and specifications tailored for business requirements. The *Business Process Execution Language* (BPEL) [139] is considered the standard language for the specification and execution of workflows, and it has broad industrial support from companies including IBM, Microsoft, and Oracle. It was created by combining features of early pioneering web service composition languages including the *Web Services Flow Language* (WSFL) [174] and XLANG [175]. These languages were adopted initially to compose and execute workflow applications in grid-based systems, but are no longer suitable for describing workflows as they became more complicated [142]. The BPEL language provides abstractions for defining *activities* to support synchronous and asynchronous business interactions between services. These *activities* can be organised using control flow structures to express sequential or parallel execution of business processes, if-else statements, cycles, etc. There are two kinds of activities that include *basic* and *structured* activities which are described as follows:

- **Basic activities:** These activities represent the basic computational steps in a business process such as the *invoke* activity which describes an invocation (e.g. request) to a particular web service.
- **Structured activities:** These activities are used to describe the order in which a collection of basic activities are executed using control flow abstractions like *if* and *while* that permit activities to be executed based on some conditions.

The BPEL language is typically executed using a centralised workflow engine. There have been many attempts to decentralise workflows based on this language including a technique that was originally proposed by Nanda et al. [183], which splits a given workflow specification into a set of processes. These processes are then executed by different servers without a centralised coordination point. Similar approaches are proposed by Baresi et al. [184], Yildiz and Godart [185], but require hosting heavyweight proprietary engines at distributed locations that may not be affordable for scientific research purposes. Decker et al. [186] has also proposed an extension to the BPEL language to facilitate seamless integration between multiple workflows and supporting decentralised behaviour but this extension has no concrete implementations for executing workflows.

GSFL

The *Grid Services Flow Language* (GSFL) [176] is an XML-based language that was developed for the service workflows compliant with the *Open Grid Services Architecture* (OGSA) [54] framework. This language describes a list of *Service Providers*, each of which has a unique name and type and a specific role in the workflow. It also provides an *activity model*, a *composition model*, and a *lifecycle* model which are described as follows:

- **Activity model:** This model describes all the service provider operations that are used in the workflow.
- **Composition model:** This model describes the interactions between the services using control and data flows.
- **Lifecycle model:** This model describes the order in which the activities are executed.

There are other grid-based workflow languages that provide similar features to GSFL. These languages include the *Service Workflow Language* (SWFL) [187] and the *Grid Workflow Execution Language* (GWEL) [177]. The main drawback of these languages is that the configuration and implementation of the services specified in the workflow need to be

altered in most situations before the workflow execution begins. However, there is a general lack of technical material that describe the features of these languages in detail, and therefore their discussion is beyond the scope of this thesis.

GWEL

The *Grid Workflow Execution Language* (GWEL) [177] is an XML-based language that has been proposed to reuse ideas from BPEL to describe the interactions between services defined with WSDL. The specification of a workflow can be seen as a template for creating Grid service instances, performing operations on the instances, and destroying them. It features a set of constructs such as *factory links*, *data links*, *variables*, *fault handlers*, a *life cycle* element and *control flow* element. These constructs are described as follows:

- **Factory links:** These constructs represent the listing of all services that take part in the workflow, each of which is identified by a unique name.
- **Data links:** These constructs represent the listing of any data sources or storage locations that will be used throughout the workflow.
- **Variables:** These constructs are used to identify data entities of specific types that are exchanged during the workflow between services.
- **Fault handlers:** These constructs are used to specify exceptions that should be caught and the activities that should be executed once an exception has been caught.
- **Life cycle element:** This element is used to instantiate or destroy certain instances, and contains the instance name and the name of the instance *factory*.
- **Control flow element:** This element describes control in the workflow within a structured set of activities like BPEL.

However, although this language provides rich abstractions for describing business-oriented workflows in general, it may not be a suitable choice for composing web services used in scientific workflow applications. Typically, scientific workflows are expressed as

DAGs and therefore do not require control-flow constructs to construct them. Control-flow constructs may increase the size and complexity of the specification of a scientific workflow. The authors of this language did not investigate the performance of the implementation of this language [177].

GEL

The *Grid Execution Language* (GEL) [178] is a succinct language for executing programs on a heterogeneous distributed system. It permits the user to define programs as a set of *jobs* to execute and their execution order, and supports *cyclic* control structures (e.g. loops). Unlike other grid-based languages, this language does not require any modifications to be made to the services being used in the workflow. Chua et al. [188] present a scientific workflow application that is specified using this language, which is used to analyse how genes are transcribed in different tissues. This workflow application involves hundreds of jobs that extract thousands of annotated exons from the human genome and compares them against records stored in a large database of transcripts. Similar to *SwiftScript*, this language can handle the execution of legacy software and the data transfer between *jobs*. Unlike other grid-based workflow languages such as GSFL [176], SWFL [187], and GWEL [177], this language does not require any modifications to be made to the grid services being used in the workflow. However, this language is not directed at executing web service workflows and it does not support automatic detection of data dependencies as used in *Swift*. For example, the user must define jobs that need to be executed in parallel in an explicit manner. This increases the difficulty of composing the workflow and forces the workflow architect (e.g. scientist) to think about *how* the workflow jobs need to be executed, instead of composing the workflow based on *what* jobs need to be executed.

GALE

The *Grid Access Language for High-Performance Computing Environments* (GALE) [179] is a language that is used to describe sequences of tasks to be performed on Grid resources. It features a *resource query*, *computation activity*, and *data transfer* instructions which are

described as follows:

- **Resource query:** This query is used to determine the availability of resources advertised in a grid-based information service.
- **Computation activity:** This activity specifies a number of environment settings, application arguments, and computation attributes.
- **Data transfer instructions:** These instructions command the workflow engine to transfer data between resources.

Despite the fact that this language provides abstractions for describing workflows using key grid-based services, it does not support the specification of standard web service interactions.

YAWL

The *Yet Another Workflow Language* (YAWL) [182] and its extended version *newYAWL* [189] are workflow languages that were developed to show that comprehensive support for business-oriented workflow patterns [115] is achievable. This language is based on *Petri nets* [190] which provide formal semantics for modelling workflows [191] using different control-flow structures for execution [192]. It is argued that *Petri-nets* semantics may not be sufficient to represent workflows properly and therefore these semantics require to be extended using UML [193] activity diagrams according to Eshuis and Wieringa [194], Pallana et al. [195], Dumas and Ter Hofstede [196], and Bastos et al. [197]. Unlike *Petri nets*, YAWL permits the composition of different tasks (e.g. atomic and composite) directly to each other without using conditions. This helps in reducing the size and complexity of the workflow during its construction. The execution environment of this language conforms with service-oriented architectures. It provides a workflow engine that is responsible for creating tasks and coordinating their execution based. This engine delegates responsibility for task execution to a service that may be chosen from a pool of available services. These services may be able to invoke external web services or applications, perform data transformations, etc. Curcin and Ghanem [198] discuss the features of this language and argue

that YAWL is not adequate for expressing scientific workflows as it was designed to support business-oriented workflows. This language also has limited industrial support.

WS-CI

The *Web Services Choreography Interface* (WS-CI) [180] language is based on XML and it was created by Sun, SAP, BEA and *Intalio* to describe messages exchanged between web services in a collaborative activity, which is referred to as *service choreography*. Each web service is associated with a set of WS-CI interfaces, and each of these interfaces describes the behaviour of the service with another service. This permits each service to interact with other services independently without the need for a single entity that controls the overall collaboration between the services. Unlike WSDL which describes the entry points for each web service, WS-CI describes the interactions among the operations specified in WSDL. WS-CI supports *basic* and *structured* activities, and provides a set of constructs such as *action* which maps to a specific service operation and describes its request or response message, *call* for invoking the operations of remote services, *all* for indicating that a set of actions can be performed in parallel and without any specific order, and *switch* for conditional execution. However, this language is considered a non-executable specification language that describes the observable behaviour of web services and their interactions to support design decisions. There is no concrete implementation of this language and it no longer has industrial support as it was superseded by the WS-CDL language.

WS-CDL

The *Web Services Choreography Description Language* (WS-CDL) [181] is a non-executable XML-based language that describes web service interactions from a high-level perspective. This language was originally proposed to support design decisions. It can be used to check for conformance and ensure service interoperability in a distributed system. Based on WS-CDL, a web service is known as a *participant* and can perform one or more predefined *roles*. For any collaborative pair of services, a *relationship* defines their actions. Every *action* that takes place is broken down into a set of public message exchanges between the

services. This language permits *channels* to be specified, which represent communication between participants and define the characteristics of message exchanges. This thesis does not attempt to analyse the design of this language as this research has already been done by Decker et al. [199]. There are few implementations that support the specification of workflows based on WS-CDL. Current implementations include a syntax analyser [200] by Fredlund, and a prototype of WS-CDL+ [201] by Kang et al. which is a proposed language extension to support its execution. Ross-Talbot et al. provide Pi4SOA [202] which is a graphical modelling tool that permits a user to model and verify web service choreographies. This tool is capable of generating web service choreography specifications based on WS-CDL that can be reused in different projects. Mendling and Hafner [203], [204] focused on transforming service choreography specifications based on WS-CDL to BPEL to support their execution. However, the authors do not demonstrate the effectiveness of their approach. There are ongoing efforts to formalise WS-CDL based on π -Calculus led by Carbone et al. [205], [206], and Hongli et al. [207].

Let's Dance

Decker et al. [208] provide an alternative language to WS-CDL known as *Let's Dance*, which captures models of service interactions from a behavioural perspective using a graphical modelling tool called *Maestro* [209]. These models include a *global* model in which interactions are described from the perspective of a centralised controller that oversees all the interactions between a set of services, and *local* models that focus on a particular service and captures its interactions. Such interactions are represented as public message exchanges between the services. This language is used for conformance checking of service behaviour [210]. For example, global models can be produced and studied by analysts to agree on appropriate service interactions, while local models can be produced for developers who refine the local models or use them to generate templates of workflow specifications [211]. However, this language requires additional information relating to the configuration of the services to support the execution of its models .

2.6.3 Dataflow Optimisation System Architectures

This section presents distributed system architectures that aim to improve the performance of executing a workflow by employing techniques that optimise the data movement between the services. These distributed systems include *Dryad* [212], the *Circulate Approach Architecture* [213], *Service Invocation Triggers* [214], and the *Flow-based Infrastructure for Composing Autonomous Services* (FICAS) [215].

Dryad

Dryad [212] a general-purpose distributed execution engine for coarse-grain data-parallel applications that was developed by *Microsoft*. In *Dryad*, a distributed application that consists computational tasks which communicate with each other using channels. *Dryad* runs the application by executing its computational tasks on a set of available computing machines, communicating as appropriate by exchanging data through TCP pipelines. *Dryad* allows each computational task to be executed in a concurrent manner, and employs scheduling techniques to produce data placement plans at run-time to optimise the use of available resources. Similar to *Condor* [45], *Dryad* leverages the resources of many workstations using batch processing but it is only designed to focus on high-throughput local area networks [212]. *Dryad* also relies on a simple language called *DryadLINQ* [216] for writing large-scale applications that can be executed on a group of clusters. However, all data dependencies relating to the computational tasks in this language must be specified in an explicit manner. *Dryad* does not provide an automatic mechanism to detect these dependencies implicitly during compilation of the specification of the distributed application.

The Circulate Architecture

The *Circulate* architecture [213] is a dataflow optimisation architecture. This architecture relies on proxies can be deployed within short network distance to the services involved in the workflow. The proxies exploit connectivity to the services by performing service invocations. Furthermore, the proxies exchange references to the data with a centralised execution

engine and pass data values directly to locations where it is required. This allows the execution engine to monitor the progress of the execution. The authors of this architecture evaluate its performance by orchestrating a workflow that simulates the a scientific workflow application called *Montage* which has been described earlier in Chapter 1. Although this architecture provides some performance benefits over a fully centralised orchestration architecture, the *Circulate* architecture has a number of disadvantages. Firstly, it relies on a centralised mechanism to facilitate the collaboration between proxies. The centralised execution engine may become a potential performance bottleneck. It holds information about the workflow execution state, and maintains references to all the data relating to the workflow which may be used by the proxies. These proxies would most certainly be unable to collaborate with each other if the engine suddenly disappears due to an unexpected failure or unforeseen network problems. Secondly, if a proxy that holds some data fails then all the references to that data become invalid and it would be impossible for other proxies to obtain the data. Finally, there seems to be no automated mechanism for partitioning the workflow in this architecture as the interactions between the proxies must be specified manually.

Flow-based Infrastructure for Composing Autonomous Services

The *Flow-based Infrastructure for Composing Autonomous Services* (FICAS) [215] is a distributed dataflow architecture for composing software services into workflows described as “mega-structures” by its authors. Composition of the services in this architecture is specified using the *Compositional Language for Autonomous Services* (CLAS). This language supports the specification of interactions among collaborating services in a sequential manner. The specification of a workflow based on this language is translated into an executable control sequence using a build-time component. Barker et al. [213] argue that FICAS does not deal with modern web standards, and that it is only considered as a prototype and a proof of concept. Balasooriya et al [217] provide a similar approach to FICAS that relies on self-managing proxies that embed the coordination logic of the workflow. However, the main drawback of these works is that the implementation is not platform independent and requires the modifications of the applications and services being coordinated.

Service Invocation Triggers

Service Invocation Triggers [214] provide triggers (e.g. proxy) to invoke particular services, collect the invocation results and forward data related to these results directly to remote triggers that require the data. Each trigger is executed upon the availability of the input data that is required for its execution which supports decentralised service orchestration. The implementation of these triggers permits the workflow logic to be deconstructed into a set of sequential fragments that does not consist of control structures (e.g. loops and conditionals), and encodes the data dependencies within the triggers themselves. This solution, however, does not support standard web service technology according to Barker et al. [213].

2.6.4 Workflow Scheduling

It is difficult to discover a single appropriate solution for mapping tasks onto resources for all workflow applications. Hence, a workflow management system must be able to make mapping decisions based on some criteria (e.g. workflow structure, performance metrics, or high-level constraints). Deelman et al. [23] argues that decisions about a workflow can be made with regard to information about the whole workflow or its individual tasks. Such decisions can be made before the execution of a workflow using a *planner* component, or during the workflow execution using an *executor* component in the workflow management system. For example, the *planner* typically generates a scheduling plan before execution that maps all the tasks in a workflow onto a particular set of resources, which leads to making a few decisions about scheduling during execution. During the workflow execution, the *executor* may have the choice to select which resource should be used to execute a particular task. Planners and executors often rely on user-directed techniques or heuristics to make such decisions. Before the execution of a workflow, the user can define a set of rules for mapping particular tasks onto resources based on their knowledge (e.g. a user may prefer to map tasks onto specific resources that are more reliable than others based on the user experience) according to Yu and Buyya [172]. Scheduling heuristics are used to predict the most appropriate resources to execute the workflow based on the analysis of some information

that can be obtained at run-time from the execution environment. Such information may represent metrics (e.g. processing power, storage capacity, network latency, network bandwidth) that may be gathered using resource and network monitoring mechanisms, and can be recorded in a datastore and used in the future to estimate the performance of executing a particular task according to Mayer et al. [156], Jang et al. [218], and Smith et al. [219]. These heuristics are essential for mapping tasks to resources in a workflow, and affect the performance of the system according to Hamscher et al. [173], and the differences amongst which persist in how tasks are given execution priorities (e.g. the execution order of the workflow tasks). This thesis considers heuristic workflow scheduling approaches.

Some notable examples of heuristics used in scheduling workflows include the *Heterogeneous Earliest-Finish-Time* (HEFT) algorithm which is discussed by Topcuoglu et al. [220]. This algorithm attempts to select a task with the highest rank at each step, where each rank is defined by the latency between resources including the computation and communication costs. This algorithm is employed in the ASKALON system to support the execution of grid-based workflow applications. Similar strategies attempt to solve the workflow mapping problem in grid environments including *Min-Min* according to Blythe et al. [221], *Max-Min* and MCT according to Braun et al. [222]. However, the value of a rank may be different when assigned to different resource in a heterogeneous environment and it can ultimately affect the execution performance of a workflow.

Duan et al. [223] propose a strategy for mapping tasks onto grid sites in which weights are assigned to vertices and edges in the workflow graph by predicting the execution time for each task, and the time for transferring data between the resources. Each task is then mapped onto a resource that provides the earliest expected time to complete its execution. However, the time for executing a service operation cannot be predicted efficiently in a service-oriented environment as it typically depends on the application logic, and the underlying protocols and infrastructure.

Several other heuristic methods were proposed and compared by Sakellariou and Zhao [224], and a partitioning technique is proposed for provisioning resources into execution sites by Kumar et al. [225] but does not support decomposing the actual dataflow graph.

2.6.5 Related Surveys

This thesis does not cover all aspects related to workflow management systems as they have been discussed thoroughly in many survey studies, and therefore this thesis summarises these studies which are classified into *general*, and *specialised* surveys.

General Surveys

There are three general studies that focus on numerous aspects of workflows and workflow management systems. These studies are summarised as follows:

1. Deelman et al. [121] provide a study that classifies workflow management systems based on characteristics identified by scientists and domain experts. This study describes in more detail the workflow lifecycle and compare the functionality provided by different workflow management systems to support each workflow lifecycle stage.
2. Barker and van Hemert [102] present a detailed survey of existing workflow languages and management systems from both the business and scientific domains. It provides a number of suggestions towards the future development of scientific workflow systems.
3. Yu and Buyya [172] provide a study that classifies workflows according to their design structure, specification, composition method using textual languages and graphical modelling toolkits, workflow scheduling and execution, and fault tolerance.

Specialised Surveys

There are a number of surveys that focus a specific aspects of workflow management. These surveys are summarised as follows:

1. Wieczorek et al. [226] provide a survey that addresses the problem workflow scheduling based on several aspects related to the modelling of a workflow, its tasks and resources.
2. Chen and Yang [227] provide a survey that concentrates on ensuring the correctness of workflow specifications (e.g. issues relating to workflow verification and validation).

3. Han et al. [228] provide a survey that focuses on adaptive workflow management. They provide their own classification of different types of workflow adaptation, and discuss mechanisms to achieve adaptation based on dynamic composition of workflow resources, workflow models with support for exception handling.
4. Wu et al. [229] provide a comprehensive survey of workflow scheduling techniques relating to the mapping of tasks onto computational resources, on-demand resource provisioning, performance fluctuation and failure handling in cloud-based environments. The authors of this survey provide a set of future research directions in the area of workflow scheduling.
5. Simmhan et al. [123] provide a survey that focuses on data provenance in *e-science*. This survey provides a taxonomy that classifies workflow systems that support provenance based on their objectives (e.g. the reasons for recording provenance), the description of provenance that is typically stored using these systems, and the techniques these systems employ for representing, storing, and disseminating provenance information. It also identifies a set of open research problems related to provenance.
6. Ramakrishnan and Gannon [230] provide a survey of distributed workflow applications and describe their characteristics and resource requirements.

This thesis may not cover all existing workflow technologies including workflow management systems and languages, workflow scheduling techniques, and specialised survey articles. However, the author firmly believes that the knowledge presented in this chapter is sufficient to identify the limitations of existing workflow technologies, and derive a set of requirements to support conducting scientific workflows. Section 2.7 describes and analyses scientific workflow requirements, whereas Section 2.8 summarises the limitations of existing workflow technologies.

2.7 Requirements Analysis

Existing technologies address fundamental issues in workflow management, but the scientific community is demanding specialised languages to improve the productivity and efficiency in composing workflows, and new infrastructures for orchestrating workflows to address the challenges in data deluge. This section presents a set of requirements needed to address the the limitations of existing works, and current and future challenges in workflow management stated in different studies by Goble and De Roure [4], Deelman and Gil [18], Zhao et al. [160], and Gil et al. [133]. These requirements include:

1. Global scientific collaboration.
2. Usability support.
3. Service interoperability support.
4. Parallelism support for executing computational tasks.
5. Scalability support.
6. Deployment and optimisation support.
7. Reusability support.
8. Fault-tolerance support.
9. Reconfiguration support.

2.7.1 Global Scientific Collaboration

Scientific research requires the collaboration between scientists who work in different institutions around the globe. This collaboration involves sharing data and computation resources (e.g. services) to conduct experiments and simulations. Chapter 1 has provided examples of large-scale workflows that involve the collaboration of scientists across geographical boundaries. Nowadays, there are a few *Infrastructure as a Service* (IaaS) clouds

that make it easy for scientists to perform data-centric computations. Parastatidis [19] argues that it will be essential in the future to develop programming patterns to support data-centric scientific research activities that involve the aggregation and acquisition of data. Scientists must be able to build a large-scale application capable of exploiting the world's computer-represented scientific knowledge according to Parastatidis [19]. Maechling et al. [231] describe the characteristics of their research which depends on workflow technology. Such characteristics include the collaboration between scientists across many organisations, and the integration of scientific techniques from different disciplines. Hence, a workflow management system must therefore support global scientific collaboration providing means for defining resources that may be hosted in different geographic locations, the computational tasks needed to operate on data obtained from these resources, and the organisation of the execution of these tasks effectively.

2.7.2 Usability Support

Composing workflows is often difficult and requires a unique set of programming and technical skills that are beyond the skills of many researchers. For example, a research may use a programming language to compose functionality provided by independent software modules to perform a set of computational tasks in a workflow context (e.g. experiment). This requires the researcher to think about how these modules need to be interfaced instead of describing what functionality needs to be executed. Furthermore, a researcher may be required to configure the execution environment manually. This increases the difficulty of composing large-scale workflows as the researcher will be more concerned with the application-specific behaviour of the tasks to be executed instead of describing the data flow dependencies required to execute the tasks. Ludäscher et al. states that scientists must be able to compose and orchestrate workflows easily by reducing the effort required to construct them [27]. Designing a workflow must be simple and intuitive, without the need to deal with low-level details related to the workflow execution. Barker and van Hemert [102] highlight this issue and argue that existing workflow languages support rich constructs to compose workflows but are difficult to use by scientists due to their complexity. Scientists therefore require a

higher level of abstraction for composing workflow tasks together to solve a research problem, without knowledge of how the workflow is executed.

2.7.3 Service Interoperability Support

Based on the literature work, workflow management systems must support the integration, and access of heterogeneous data and computation services to support scientific experiments. Sonntag et al. [232] argues that scientific workflows are difficult to model due to many factors among which is the workflow size (e.g. number of services involved) and complexity (e.g. the manner in which the services are composed together). For example, services must be composed together correctly to construct workflows and it is difficult to ensure the correctness of the workflow when a large number of services are involved. Such services may be hosted by different service providers on heterogeneous machines with unique configurations and application server technologies that cannot be modified by researchers. Hence, the following requirements must be addressed to support service interoperability in a workflow language:

1. The workflow language must support the composition and interoperability of web services using high-level abstractions that coordinate the data movement between them, and by capturing the functionality and typed properties of these services.
2. Service composition must not involve modifying existing services or reconfiguring their hosting environments. The workflow language must therefore rely on neutral technology mechanisms to support service composition.

2.7.4 Parallelism Support

There is an increasing demand for a scientific workflow language that supports scalability for composing a large number of tasks, and exposes parallelism in a manner such that the data dependencies in the workflow can be detected automatically by the language compiler as discussed by Zhao et al. [134]. This means that the workflow language does not need to

support all the constructs featured in conventional workflow languages, and the workflow can be automatically decomposed into smaller parts that may be mapped for execution onto distributed machines to improve performance. Therefore, the following requirements must be addressed to support parallelism in the workflow:

1. The workflow language must provide a set of abstractions to express a workflow in a manner such that the compiler can effortlessly produce highly scalable, distributed form of the workflow.
2. Scientists must also be able to compose services in a workflow using common dataflow patterns, which represent the basic building blocks for composing large-scale scientific workflows.

2.7.5 Scalability Support

Most existing workflow technologies are based on a centralised orchestration approach that does not support scalable execution of scientific workflows. This is because centralised orchestration relies on a single execution engine that may become a performance bottleneck especially as both the workflow size (e.g. number of tasks) and the number of services increase. Gil et al. [133] argue that scalability must be considered in terms of the number of tasks that need to be executed in a workflow, the number of data or computation resources involved in the workflow execution, and the size of the data involved in the workflow. Therefore, a workflow management system must satisfy the following requirements:

1. The system must support a decentralised service orchestration approach that addresses the limitations of completely centralised service orchestration approaches.
2. The system must support scalability in terms of the number of services involved in a workflow and the overall size of the data produced in the workflow.

2.7.6 Deployment and Optimisation Support

Scientists must be free from dealing with administrative tasks such as deployment (e.g. mapping tasks onto resources for execution) and optimisation tasks needed to improve the performance of the workflow execution. Typically, it is possible to manage a workflow application effectively up to a certain size and complexity, beyond which manual management using software tools becomes unrealistic. It becomes necessary to employ a system that can make decisions without human guidance to manage the enactment and execution efficiently. This can be achieved through automatically decomposing a workflow into smaller independent parts that can be executed in parallel. Placement analysis is needed to determine the most appropriate engines onto which the workflow parts can be deployed for execution. It relies on network resources monitoring to gather QoS information related to the services participating in the workflow, and the available engines to produce an efficient plan a deployment plan. Deployment refers to the activity that makes the workflow partitions available for use by the workflow engines. The system must therefore be able to do satisfy the following requirements:

1. The system must be able to partition the workflow logic into smaller parts that can be executed in parallel, where each part represents an independent self-contained sub workflow.
2. The system must be able to partition the workflow automatically without user intervention (e.g. users should not have to manually partition the workflow).
3. The system must be able to monitor the network bandwidth and latency of connections between its execution engines and services in the workflow.
4. The system must be able to use resource monitoring information to manage the placement of the workflow partitions for initial deployment.
5. The system must be able to deploy the workflow partitions onto candidate execution engines transparently, and the deployment activity must not require direct user intervention.

2.7.7 Reusability Support

Scientific workflows represent experiments that may be executed repeatedly using different data sets to produce new results that can be then studied and analysed by researchers. Barga and Gannon [120], and Ludäscher et al. [27] describe the nature of constructing scientific workflows, reusing them, and changing them frequently to support different experimental requirements. Ludäscher et al. [111] introduced the notion of repeating the execution of a workflow in the Kepler workflow management system. Hence, a workflow management system must permit scientists to reuse existing workflows using different datasets.

2.7.8 Fault-tolerance Support

Fault-tolerance support is an essential requirement in a workflow management system. Many events such as sudden service failures, and network disruptions can halt the workflow execution. Gil et al. [133] examine the challenges of scientific workflows and indicates the need for mechanisms to detect and deal with failures that happen due to “dynamic” changes in the execution infrastructure. There are a few research contributions that focus on handling failures in workflows, and some of these contributions are discussed by Eder and Liebhart [233]. Hwang and Kesselman [234] also attempt to address failures encountered when executing grid-based workflows using a flexible failure handling framework that relies on a high-level specification of recovery policies. Therefore, a workflow management system must be resilient to failure of individual machines running workflow instances. It is expected that multiple machines may fail too often to depend on manual recovery. The system must not require manual intervention to recover from machine failure and it must be able to continue operating normally up to the failure of a specific number of machines. This number should be configurable by the system, but it is limited by the number of machines that are executing the workflow at a certain time.

2.7.9 Dynamic Reconfiguration Support

During the execution of a workflow, its performance can be affected by sudden changes in the execution environment and therefore the workflow must be reconfigured accordingly to adapt to such changes. It is likely that the network condition will change over time, meaning that it is not possible to rely on the same deployment configuration of the workflow partitions and the system must adapt to the dynamic changes in the network environment. Therefore the system must be able to monitor the resources in the execution environment, and obtain information about their current state, and employ new data or computation resources (e.g. services or engines) as necessary. Large-scale, and long-running workflows commonly require support for dynamic configuration which involves changing the structure of the workflow based on changes in the execution environment (e.g. failures) that are discussed in Section 2.7.8, or high-level specified goals that may be specified by the workflow architect (e.g. scientist) to improve the overall workflow performance. Existing research on dynamic configuration is focused on distributed systems whose components can be expressed using an architectural description language, and include works provided by Magee and Kramer [235], Oueichek and De Pina [236], Shrivastava et al. [237], and Bellissard et al. [238]. However, there are few works that attempt to address dynamic reconfiguration for scientific workflow applications. Baek et al. [239] proposes a self-adaptive system that permits a workflow to be executed based on specific goals specified by the user, but the applicability of this system has not been evaluated in different application domains as the authors indicate in their paper. Sadiq et al. [240] discuss a set of challenges related to dynamic configuration but only in the context of business-oriented workflow applications.

The requirements described above may not cover all the requirements that need to be satisfied by a workflow management system or a workflow language. These requirements are used only to highlight the essential challenges that may not be addressed at all or adequately addressed by existing workflow technologies. This thesis presents an experimental workflow language, and a decentralised service-oriented orchestration system that attempt to address these requirements.

2.8 General Discussion

This thesis draws the main benefits of using workflows in scientific research based on studies provided by Goble and De Roure [4], Görlich et al. [119], and Taylor et al. [112]. These benefits include the following:

1. Firstly, a workflow management system aims to support collaborative research and enable scientists from different communities to share computation services and data resources to conduct experiments.
2. Secondly, it provides the ability to construct workflows without dealing with technical details relating to their execution.
3. Thirdly, a workflow management system aims to automate the execution of scientific experiments. This is achieved by performing a wide range of activities without the direct intervention of the scientist such as mapping workflow tasks onto distributed and heterogeneous computational resources, executing these tasks and monitoring the overall workflow execution.
4. Finally, a workflow management system may employ optimisation mechanisms that improve the execution performance.

Existing workflow management systems may be based on different execution models, and provide their own development kit for composing workflows and executing them as discussed in Section 2.3.4. Most notable workflow management systems such as *Sedna*, *Taverna*, *Kepler* are based on an execution model that relies on a centralised engine for orchestrating the workflow tasks. Centralised workflow management systems may employ planners that produce concrete workflows from abstract workflows such as *Pegasus*. Hamscher et al. [173] argues that centralised approaches can produce efficient execution plans. However, these approaches are not scalable as both the workflow size (e.g. number of tasks) and number resources increase according to Yu and Buyya [172]. They argue that these

approaches are only suitable for small-scale workflows or a large-scale workflows in which every task has the same objective as those seen in business-oriented applications. Barker et al. [213] also discuss the limitations of centralised orchestration and proposes a dataflow optimisation system architecture to address these limitations called the *Circulate* architecture. This architecture employs distributed proxies to execute the workflow. These proxies are responsible for invoking services, and collaborate with each other by exchanging information about data (e.g. data references) rather than the actual data in the workflow. This permits a particular proxy to use the a reference to obtain some data needed for executing part of the workflow directly from other proxies when necessary, and without the need for the data to pass by a centralised workflow engine. However, this approach has a number of limitations that are discussed in Section 2.6.3. It relies on a single coordination engine that oversees the behaviour of the distributed proxies. This coordination engine represents a single point of failure, and its failure would affect the overall progress of the workflow execution. Furthermore, if a proxy that holds some data fails then all the references to that data become invalid and it would be impossible for other proxies to obtain the data. Yu and Buyya [172] highlight the disadvantages of using a centralised approach for executing workflows, and partially centralised approaches (e.g. hierarchical) that employ a single coordination engine and multiple distributed low-level workers.

Decentralised peer-to-peer workflow management systems such as *Triana* enable parts of the workflow to be executed by each peer. Compared to centralised workflow management systems, decentralisation is more scalable in terms of its support to large-scale workflows. However, existing decentralised workflow management systems may not produce efficient plans for executing different parts of the workflow and may suffer from resource conflict problems according to Mateescu [241]. Chafle et al. [242] proposed a decentralised orchestration approach for services to support the execution business process workflows, which decomposes a workflow into smaller parts that can be executed by multiple engines, but the authors do not explain how the decomposition process is performed. Decentralisation is not a new concept, but it is not well-researched in the area of orchestrating scientific workflow applications that involve data-centric computation and data services.

2.9 Conclusion

This chapter has presented a general background of service-oriented computing, workflow technology, scientific workflows and the design patterns needed to compose them, the scientific workflow lifecycle, and the workflow management system reference model. It provided a definition of service-oriented orchestration, and discussed different kinds of orchestration models including decentralised and completely centralised orchestration model. Furthermore, it has reviewed existing workflow technologies including workflow specification languages, workflow management systems, dataflow optimisation system architectures, workflow scheduling approaches, and listed a number of related general and specialised surveys in the area. Based on this chapter's discussion of existing workflow technologies, this section provides the following conclusions:

1. Firstly, the increasing difficulty of designing, executing, and maintaining workflows manually have created a demand for a high-level approach that separates the workflow design from execution, and an automated orchestration mechanism.
2. Secondly, centralised orchestration suffers from a performance bottleneck when dealing with data-centric workflow applications.
3. Finally, decentralisation may hold the key to solving the problems of centralised orchestration. However, there is limited research that have investigated this area.

This chapter has also identified a set of requirements that must be satisfied by the design and implementation of a workflow management system, and provided a general discussion that highlights the benefits of using workflows in scientific research and summarised the limitations of existing workflow technologies.

Chapter 3

The *Web Service Orchestra* Language

3.1 Introduction

This chapter presents a high-level, function, and strongly-typed data coordination language called the *Web Service Orchestra*. This language aims to express a workflow in a simple and intuitive manner such that the compiler can produce highly scalable, distributed form of the workflow effortlessly. *Orchestra*'s workflow logic can be partitioned into smaller computational units (e.g. sub workflows) that can be distributed across the network, and may be executed at different sites (e.g. network locations). Therefore, there is no single locus of control as the workflow execution may involve multiple threads of control within a single address space, multiple address spaces on a single machine, or several machines over a network.

The remainder of this chapter is organised as follows: Section 3.1.1 provides a general overview of *Orchestra* and describes its characteristics. Section 3.1.2 presents the syntax of the language using a simple example. Section 3.2 discusses the language design and its philosophy. Section 3.3 presents high-level abstractions used to compose services based on common dataflow patterns. Section 3.4 presents computer-generated abstractions that support distributed computation of workflows. Section 3.5 discusses the type system and supported data structures of the language. Finally, Section 3.6 concludes this chapter.

3.1.1 Language Overview

Orchestra is a workflow language that is uniquely distinguished by the following characteristics:

1. Simplicity.
2. Parallelism support.
3. Determinism support.
4. Data-driven execution support.
5. Strong type system.

Simplicity

Simplicity is the cornerstone in designing programming languages. *Orchestra* is distinguished from other workflow languages by its ability to separate the workflow logic from its execution using simple abstractions. This permits a workflow architect (e.g. scientist) to concentrate on the inherent problems of designing workflows and not on how they are executed. *Orchestra* can be considered a “Coordination Language” [243]. Coordination languages refer to a class of functional languages based on a communication model that permits primitives (e.g. tasks) to operate over data objects. *Orchestra* provides abstractions needed only for defining the services participating in a workflow, and composing their functionality by coordinating the data movement between them.

Parallelism Support

Parallelism refers to the ability to execute multiple workflow parts concurrently. *Orchestra* is distinguished from other workflow languages by its intrinsic support for parallelism. It permits a workflow to be represented as a *Directed Acyclic Graph* (DAG) that consists of vertices representing service operations with edges between them representing data dependencies. Data dependencies can be detected automatically and analysed during compilation

to determine opportunities for parallelism [244]. This helps in decomposing the workflow into smaller independent parts that can be executed concurrently. Parallelism in *Orchestra* is supported by indicating the data dependencies explicitly in the workflow specification using simple constructs.

Determinism Support

Determinism constitutes that the result of any specified workflow must be the same regardless of its execution manner, and it has been considered as an advantageous feature in data coordination languages as stated by Karp and Miller [245]. *Orchestra*'s deterministic feature supports *referential transparency* for all its operations. Each operation is known to be “referentially transparent” when it is:

1. Free from side-effects. This means that the execution state of the workflow at a particular time depends on the execution of its individual tasks (e.g. service operations), and their completion. For example, the workflow state changes as soon as an operation produces an output but the workflow state does not change onwards unless that output is used in the future in the execution of other operations.
2. Pure. This means that an a workflow task (e.g. service operation) will always produce the same output when provided with the exact set of inputs with needed for its execution, with the condition that this operation is provided by the exact service implementation. Otherwise, if a different service is used then it may produce a different output depending on the functionality being invoked from the service.

Determinism can be useful because a programmer can compose a workflow for execution on a sequential machine, which may then be compiled and debugged to be executed on parallel machines. This permits an operation’s output to be held in a local datastore when obtained from a remote service, and allows the retrieval of the output from the datastore when necessary without executing the operation twice. Section 4.2 describe this datastore and discuss how it is used.

Data-driven Execution Support

Each service operation in *Orchestra* can be invoked as soon as all the inputs required for its invocation become available. This is useful for the following reasons:

1. Firstly, it permits the input data for a particular workflow task (e.g. service invocation) to be obtained from different sources (e.g. user, services, engines).
2. Secondly, it supports parallelism as more than a single workflow task can be executed at once.
3. Thirdly, the order in which the workflow tasks are arranged in the workflow specification is irrelevant to their execution. This is because each workflow task is executed in a concurrent manner unless its execution depends on the outputs of other tasks.
4. Finally, data-driven execution does not require a scheduling mechanism that coordinates the flow of data from one workflow task to another. This is because the output of executing a workflow task is directly passed to a subsequent task that may require it, which is particularly useful as this model does not have additional overhead for sending requests for data from their sources.

Strong Type System

Orchestra is a strongly-typed language that uses a set of types expressed in a type system matching those defined in the XML Schema standard [82]. There are many definitions of what constitutes a strongly-typed language but most of which are inconsistent. This thesis considers a language to be strongly-typed if the type compatibility of all expressions representing values can be determined from the static program representation at compile time according to Wegner [246]. Cardelli [247] discusses different programming styles and introduces his own which advocates static typing as much as possible and dynamic typing only when necessary. He argues that a language can be considered as strongly-typed when there is strict observance of static or dynamic typing that leads to the absence of unchecked run-time type errors. *Orchestra* enforces static typing in a manner such that

all identifiers that represent storage (e.g. variables) and expressions representing values are given a type during compilation that cannot be changed during run-time. This permits *Orchestra*'s compiler to perform type compatibility checks to ensure the correctness of the overall workflow specification. There are other strong reasons to enforce this type system which are summarised as follows:

1. Firstly, the user is forced to make the behaviour of workflow explicit. This makes the workflow specification easier to understand as there is no hidden behaviour.
2. Secondly, static typing provides safety as any attempt to mistreat data entities is automatically captured during compilation. Static typing and safety benefits inexperienced programmers such as scientists that often tend to make type errors. For instance, when a workflow consists of hundreds of data entities of different types, it is easy to get confused and commit mistakes.
3. Thirdly, a strongly-typed workflow specification is more likely to be correct. *Orchestra*'s type information is used by the compiler to check the compatibility of types especially for service invocations and composition.
4. Finally, the implementation of a statically typed workflow language can take advantage of the type information to produce clear-cut workflow specifications that require less maintenance. For example, correctly typed workflow specifications that represent parts of a larger workflow can be produced to be independently executed.

Orchestra's type information can be used to ensure that service invocations are not misused in the workflow specification. For example, type checking can be performed to confirm that each service in the workflow accepts an input, whose data type matches that which is defined in its interface. Similarly, the compiler uses this type information to ensure that services are composed correctly in the workflow specification. This is particularly achieved by comparing the output type of the a service to the type of the input parameter defined the interface of a subsequent service. Furthermore, type information may be used during the partitioning of a workflow which is discussed in Section 4.3.2. During partitioning, the

workflow is decomposed into smaller parts (e.g. sub workflows) and the type information is used to ensure that these sub workflows and their data dependencies are correct. The implementation of the type system is discussed in Section 5.4.1 of this thesis.

3.1.2 General Syntax

This section presents an overview of *Orchestra*'s syntax. Figure 3.1 shows the dataflow graph of a simple service invocation, which is specified in Listing 3.2 using *Orchestra*. The specification of this service invocation consists of several constructs, which include the following:

1. Workflow name.
2. Service description document reference.
3. Service endpoints.
4. I/O Interface.
5. Service operations.
6. Data coordination symbol.

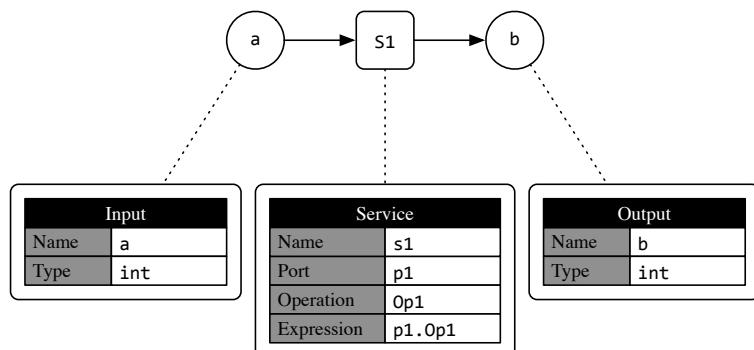


Fig. 3.1 Dataflow graph of a simple service invocation.

```
01 workflow example
02 description d1 is http://ec2-54-80-6-125.compute-1.
   amazonaws.com/services/Service1?wsdl
03 service s1 is d1.Service1
04 port p1 is s1.Port1
05 input:
06   int a
07 output:
08   int b
09 a -> p1.Op1
10 p1.Op1 -> b
```

Listing 3.2 Specification of a simple service invocation based on the example in Figure 3.1.

Workflow Name

The workflow name ‘example’ is defined using the ‘**workflow**’ keyword in Line 1. It is used to distinguish the workflow from others, and may be modified by the language compiler to prepare the workflow for distribution across the network.

Service Description Document Reference

The service description document reference represents an identifier for a service description document based on WSDL which can be located using a URL address. This document contains information about the service, its ports, operations, typed input and output messages, and it is typically based on WSDL. *Orchestra*’s compiler uses this reference identifier to retrieve the service description document, analyse it and obtain information about the service. Such information can be useful during syntax analysis to determine that the service exists, and perform type checking. Furthermore, this reference identifier permits the compiler to determine if the service is available when attempting to retrieve its service description document. If the compiler fails to communicate with the service to retrieve this document then the compilation process fails and the user is notified about the compilation error (e.g. service unavailability). Figure 3.3 shows a UML representation of a service description document used in the workflow example. Based on the workflow specification provided in Listing 3.2, ‘d1’ represents a reference to this document, which is declared using the ‘**description**’

keyword in Line 2. This reference can be used in the workflow specification to define a service endpoint and therefore exposes the location of the service.

Service Endpoints

The service reference defines a set of service endpoints where each endpoint may provide one or more ports. Each port represents a service interface that provides a set of operations that can be invoked. In this example, a service endpoint identifier `'s1'` is declared using the `'service'` keyword in Line 3. It represents a service endpoint defined as `'Service1'` in the service reference identified by `'d1'`. Similarly, a service port identifier `'p1'` is declared using the `'port'` keyword in Line 4. It represents a service interface `'Port1'` provided by the service endpoint `'Service1'` as defined in the service reference.

I/O Interface

The `'input'` and `'output'` keywords are used to define the I/O interface, which indicates the initial inputs that are required to execute the workflow and the final outputs to be obtained

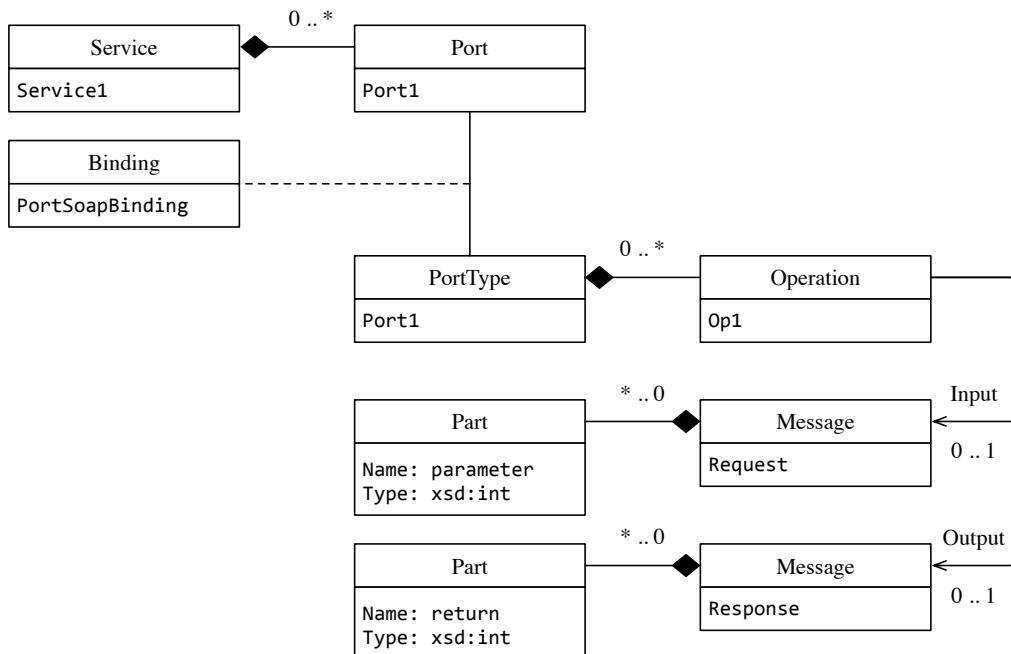


Fig. 3.3 UML diagram of the service description document used in Listing 3.1.

from the workflow execution. This interface permits multiple inputs and outputs of different types to be specified. In this example, the input '`a`' and output '`b`' are of the same type.

Service Operations

Service operations can be manipulated without restriction as they can be passed input parameters for evaluation, returned as results, passed as results in a service composition, or distributed across the network to be executed at remote locations. Operations are expressed as record data structures that may store values which may be evaluated during the execution of the workflow. Typically, an operation's value represents a computational result that is initially unknown but is obtained from a service response when the operation (e.g. service invocation) is executed. Operations can be executed when provided the correct number of inputs, and the value of the operations' results can be used whenever necessary. In Listing 3.2, a service operation is invoked using the variable '`a`' whose value is passed as a default input parameter to a service operation in Line 9. This service operation is expressed as '`p1.Op1`', which consists of two parts separated by the '`.`' symbol. The first part indicates the service port '`p1`' that provides the operation, whereas the second part '`Op1`' indicates the name of the service operation to be invoked. This operation is invoked as soon as the value of the input variable '`a`' becomes available. The result of the service invocation is passed to a storage entity that represents the output variable '`b`'.

Data Coordination Symbol

Data coordination is expressed using the left-arrow symbol '`->`' which indicates the data movement in and out of the service. This symbol acts as a data type constructor when passing values of service invocations to unknown type variables. For example, the user may wish to use an identifier for a variable in the workflow specification whose type may not be defined in the workflow interface. During compilation, a type is inferred automatically for this variable which matches the type of the value being passed to it using the data coordination symbol.

3.2 Design of the *Orchestra* Language

Central to *Orchestra*'s design is that the language itself should be simple. The introduction of abstractions to the language without integrating them into some kind of design specification increases the language complexity, which can make the language difficult to comprehend by the programmer. This issue was first explored by Wirth and Hoare [248], [249] who contended that the power and flexibility of a language should derive from unifying simplicity, rather than from proliferation of poorly integrated features and facilities. Priestley [250] explains the authors' argument; for each purpose in the language there must be exactly one appropriate facility, so that there is minimal scope for erroneous choice and misapplication of facilities whether due to misunderstanding, inadvertence or inexperience. The principal objective pursued in designing *Orchestra* focuses on conceptual economy. *Orchestra* has a small number of grammatical rules with no special features, and *Occam's Razor* is applied wherever possible to preserve the simplicity of the language. The message being conveyed here is that "*power through simplicity, and simplicity through generality*" should be the guiding star in language design as explained by Morrison [251]. The remainder of this section is organised as follows: Section 3.2.1 presents the principles used to guide the design process. Section 3.2.2 discusses the steps taken to design the language and its grammatical rules. Section 3.2.3 provides the grammatical rules of *Orchestra*. Section 3.2.4 finally provides the computation model upon which *Orchestra*'s design is based.

3.2.1 Guiding Principles

This section introduces general principles that guide *Orchestra*'s design process. These principles are listed as follows:

1. Firstly, the language must provide syntactically unique constructs. Programmers can use names based on a certain context in which the names are used, but within a single language different names may exist that may be syntactically similar. The language grammatical rules that govern the use of names must be designed together to avoid inconsistent situations and irregularities in which the names may be used. This is

known as the *principle of correspondence* which has been closely examined by Tennet [252] and its origin has roots in Landin's work [253].

2. Secondly, the language design must define semantically meaningful constructs and allow abstraction over these constructs. This is known as the *principle of abstraction* [251].
3. Finally, the language must define a complete set of data types and general rules that govern their use in a uniform manner to avoid complexity. This is known as the *principle of data type completeness* [251], [254], [255].

3.2.2 Design Methodology

This section discusses the design steps taken to produce the language grammatical rules presented in Section 3.2.3. These steps are described as follows:

1. A set of data types was chosen to support service interoperability. These data types match those defined in the XML Schema standard [82]. Section 3.5 discusses this type system in detail.
2. A set of meaningful syntactic constructs was invented for defining network entities (e.g. services, engines), external references (e.g. service description documents, data type schemas) that may be used in a the workflow.
3. An interface was introduced for defining the initial inputs required to execute a workflow and its final outputs, which supports the data type system.
4. A set of constructs was introduced to invoke services and compose them based on the computation model discussed in Section 3.2.4.
5. A set of constructs was introduced to coordinate the data movement between distributed engines to support their collaboration in workflow execution.

3.2.3 Language Grammatical Rules

This section presents *Orchestra*'s grammatical rules that govern the manner in which the language constructs are used, and discusses the design steps taken to define these rules. Listing 3.4 provides the language grammar based on *Backus Naur Form* (BNF). The language grammar consists of a primary production rule called “specification” that defines sub rules that govern the manner in which a workflow is specified. These rules define the following:

1. Workflow name and unique identification information.
2. Services.
3. Execution engines.
4. External data type schemas.
5. I/O interface.
6. Dataflow coordination (e.g. service invocation, composition, data forwarding).

Namespace Production Rule

The “namespace” rule is responsible for defining the workflow name, and additional information (e.g. unique identifier) that could be used to distinguish the workflow from others with the same name being executed using the decentralised orchestration system presented in Chapter 4. Unique identification information are generated automatically following workflow partitioning, and specified based on the “uid” production rule. Section 4.3.2 provides an example that shows how a unique identifier is used in the orchestration system.

Services Production Rule

The “services” rule relies on sub rules for declaring references to service description documents, service identifiers, and port identifiers for the services. These sub rules include the “description”, “service” and “port” rules respectively.

```

<specification> ::= <namespace> <services>* [<engine>]* [<schema>]* <interface>* <dataflow>*
<namespace>
<workflow>
<uid>
<services>
<description>
<service>
<port>
<engine>
<schema>

<interface>
<inputs>
<outputs>
<var>
<tuple>
<array>
<type>
<schema-type>
<dataflow>
<invocations>
<invocation>
<forwarding>

::= <workflow> [<uid>]
::= workflow <name>
::= uid <workflow-id> [ `.' <sub-workflow-id> ]
::= <description> <service> <port>
::= description <description-id> is <URL>
::= service <service-id> is <description-id> `.' <service-name>
::= port <port-id> is <service-id> `.' <port-name>
::= engine [ `sink` | <engine-id> ] is <URL>
::= schema <schema-id> is <URL>

::= <inputs> <outputs>
::= input `:' [ <var> | <tuple> | <array> ]*
::= output `:' [ <var> | <tuple> | <array> ]*
::= <type> <variable-id> [ `,' <variable-id> ]*
::= `[' <type> [ `,' <type> ]* `]' <tuple-id> [ `,' <tuple-id> ]*
::= <type> `[' <size> `]' <array-id> [ `,' <array-id> ]*
::= any | integer | double | float | decimal
     | byte | boolean | string | long | short
     | <schema-type>
::= <schema-id> <type> <variable-id> [ `,' <variable-id> ]*

::= [<scalar> | <variable-id>] `->' <invocations>
| <invocation> [ `->' <variable-id> | <invocations> ]
| <forwarding>
::= <invocation> [ `,' <invocation> ]*
::= <port-id> `.' <operation-name> [ `.' <input-parameter> ]
::= forward <variable-id> to <engine-id>

```

Listing 3.4 Language grammatical rules based *Backus Naur Form (BNF)*

Engine Production Rule

The “engine” rule is used to define one or more compute servers that may be used to execute the workflow. This rule permits users to define a set of available engines in the workflow specification that may be used for executing the workflow as discussed in Section 4.4. Furthermore, it permits the orchestration system to define a set of execution engines in generated specifications of workflow partitions which may be used during the execution.

Schema Production Rule

The “schema” rule is used to declare a reference to an external data type schema document. Section 3.5 explains how external data type schemas are specified and used in *Orchestra*.

Interface Production Rule

The “interface” rule relies on sub rules for defining the input and output interface used in the workflow. These sub rules include the “inputs” and “outputs” rules respectively. Each input or output may represent a specific data structure (e.g. a single variable, a tuple, or an array) based on a number of sub rules, and must have a type that is defined based on the type system discussed in Section 3.5.

Dataflow Production Rule

The “dataflow” rule relies on sub rules for invoking services or composing them. For example, a scalar or variable value to be passed as an input to one or more service operations. Multiple service operations can be specified based the “invocations” rule. Similarly, the result of a service operation which is specified based on the “invocation” rule can be used to compose services by passing it as an input to multiple service operations. Furthermore, the “forwarding” rule is used to define data segments (e.g. inputs, intermediate data, or outputs) to be forwarded to remote engines as discussed in Section 4.4.

3.2.4 Dataflow Computation Model

Orchestra's design supports a computation model that is dissimilar to the classical *von Neumann* computation model in which data passively resides in a specialised store whilst instructions are each executed in a sequence controlled by a *program counter*. Its computation model permits a workflow to be represented as a *Directed Acyclic Graph* (DAG) in which the vertices represent workflow tasks (e.g. service operations) to be executed with directed edges between them that represent data dependencies. These dependencies indicate the data movement to and from the services. For example, input data may be passed to a particular service by invoking a functionality (e.g. operation) it provides. This service may produce a result which may be passed to another service that provides a different functionality. The workflow tasks in this computation model may be ready to execute simultaneously as they represent asynchronous concurrent events. Each workflow task is executed as soon as all the input parameters required for its execution become available. This execution model must not be confused with other models that provide similar features such as actor-oriented computation models described by Dennis [256], [257], petri nets proposed by Peterson [190], or process networks described by Kahn [258]. This thesis does not consider the differences between these models which are discussed in detail by Johnston et al. [259]. The remainder of this section discusses the following:

1. The structure of the dataflow graph.
2. Dataflow dependencies.
3. Dataflow patterns.

Dataflow Graph Structure

Orchestra's dataflow graph can be represented as a DAG G that consists of vertices V_G and edges E_G , each having the form $(x \rightarrow y)$, where $x, y \in V_G$. Hence, a path in this graph can be expressed as a sequence of edges that share adjacent endpoints as from vertex v_1 to vertex v_n as follows: $(v_1 \rightarrow v_2), (v_2 \rightarrow v_3), \dots, (v_{n-2} \rightarrow v_{n-1}), (v_{n-1} \rightarrow v_n)$. This prevents the

introduction of loops and control structures in the workflow specification. For instance, the vertices v_1 and v_n cannot be identical in the preceding path. The edges that flow toward a particular vertex are input dependencies to that vertex, while those that flow away are its output dependencies. Based on this representation, a service-oriented workflow can be expressed as a graph $W = (S, D)$, where the vertices are denoted by S which can be expressed as a finite set of services: $S = \{s_0, s_1, \dots, s_n\}$, and the edges between them are denoted by D where $D \subset S \times S$. Any pair of vertices such as s_x and s_y are considered neighbours if $(s_x, s_y) \in D$. Each service can be represented as a tuple $s = (E, O)$, where E denotes the service endpoint, and O denotes the operation provided by the service.

Dataflow Dependencies

Dataflow graphs are commonly generated for compiler optimisation purposes, and have been used in the past to capture the relationship between data entities and operators for a particular program [260], [261], [262]. *Orchestra*'s dataflow graph permits services to communicate with each other using messages that contain a typed data set. For instance, an input message may contain one or more parameters to invoke a particular service operation which in turn may produce an output message that contains a result. These messages are required to compose service operations together, where the output of a particular service is passed as an input to another. Service composition is only possible when the input and output types are identical. *Orchestra*'s dataflow dependencies include the following:

- **Service input dependency:** Service input dependency means that a service invocation typically requires one or more inputs for it to be executed.
- **Service output dependency:** Service output dependency means that a service invocation produces a particular output upon its execution, which may represent intermediate or final data in the workflow.
- **Service composition dependency:** Service composition dependency means that a service invocation produces an output that is passed in the workflow as an input parameter to execute other service invocations.

These dependencies can be detected automatically by the language compiler, after which they may be analysed to determine opportunities for parallelism. For example, the analysis outcome can be used to decompose a workflow into smaller parts for execution onto distributed machines, and the dependencies can be used to determine the order in which these parts can be executed efficiently. Some advanced notations based on ontological concepts can be used to capture such dependencies by Cardoso and Sheth [263], and Paolucci et al. [264]. However, the discussion of these notations is beyond the thesis scope.

Dataflow Patterns

Dataflow patterns have been discussed in Section 2.3 in detail, and they are used as basic building blocks to compose workflow tasks for scientific applications as described by Juve et al. [265]. Similarly, *Orchestra* uses dataflow patterns to compose different services where each service is responsible for computing a particular workflow task. Therefore these patterns are re-defined as follows:

- **Process pattern:** This pattern represents a simple service invocation, where the service takes one or more inputs and produces a single output. It has been shown in Figure 3.1.
- **Pipeline pattern:** The pipeline pattern is used for chaining several services, where the output of a particular service operation is passed as an input to another service operation in a sequential manner.
- **Distribution pattern:** This pattern represents the passing of a service operation output to multiple service operations as an input parameter to these operations.
- **Aggregation pattern:** This pattern represents the collection of multiple outputs from different service operations which are all used as input parameters to a single service operation.

3.3 Service Composition

So far this chapter has presented several language constructs needed to specify a service invocation. Service invocations are the simplest computation components in *Orchestra*, and represent the basic building blocks to compose services together in a workflow. This section discusses how these invocations can be used to compose services based on the following dataflow patterns:

1. Pipeline pattern.
2. Data distribution pattern.
3. Data aggregation pattern.

3.3.1 Pipeline Pattern Specification

Orchestra allows service operations to be composed together by directing the output of a particular service operation directly as an input to another service operation. Figure 3.5 shows a pipeline dataflow pattern in which the input '`a`' of integer type is passed to service '`s1`' which produces an output that is passed directly to service '`s2`' which produces the output '`b`' of integer type. Listing 3.6 provides this pattern's specification in a workflow called '`pipeline`' where the input '`a`' is used to invoke '`p1.Op1`' in Line 12. The invocation's output is directed as a default input parameter to '`p2.Op2`' in Line 13, which produces an

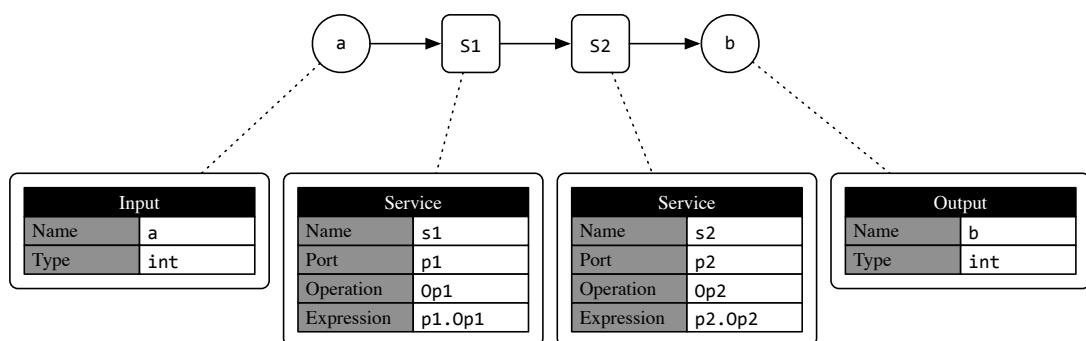


Fig. 3.5 Dataflow graph of services composed in a serial manner.

output obtained in Line 14. Service operations may accept one or more inputs depending on their implementation, and may not require any inputs at all. It is essential to note that the order in which the service invocations are organised through lines 12-14 is irrelevant to their execution. The order in which these service invocations are executed depends on the data dependencies between them, which are detected automatically by the compiler.

```

01 workflow pipeline
02 description d1 is http://ec2-54-80-6-125.compute-1.
   amazonaws.com/services/Service1?wsdl
03 service s1 is d1.Service1
04 port p1 is s1.Port1
05 description d2 is http://ec2-54-80-6-122.compute-1.
   amazonaws.com/services/Service2?wsdl
06 service s2 is d2.Service2
07 port p2 is s2.Port2
08 input:
09   int a
10 output:
11   int b
12 a -> p1.Op1
13 p1.Op1 -> p2.Op2
14 p2.Op2 -> b

```

Listing 3.6 Specification of the pipeline pattern based on Figure 3.5.

3.3.2 Data Aggregation Pattern Specification

Orchestra's design supports data-driven execution by permitting each service invocation to be executed when all its input parameters are satisfied at run-time. This permits a service invocation to be executed as soon as its input parameters become available from different sources. Figure 3.7 shows the data aggregation pattern where the inputs 'a' and 'b' are passed to services 's1' and 's2' respectively in service invocations. The outputs of these service invocations are then used as input parameters to invoke service 's3' which in turn produces the final workflow output 'c'. In this example, both inputs 'a' and 'b', the final workflow output 'c', and the parameters 'par1' and 'par2' which are used to invoke service 's3' are all of integer type.

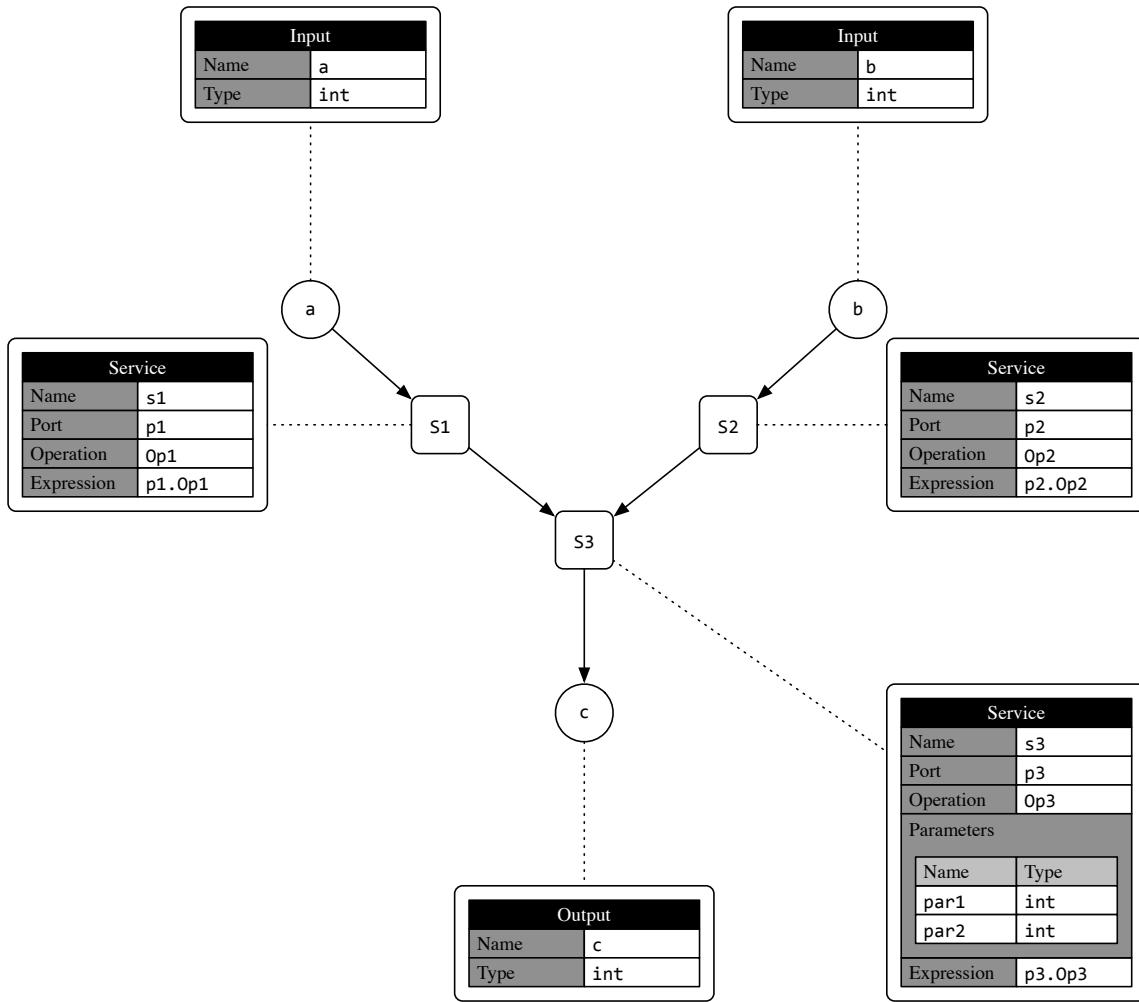


Fig. 3.7 Dataflow graph of services composed using the aggregation pattern.

Listing 3.8 presents the specification called "aggregation" which represents the data aggregation pattern. In this specification, the service description document references, the services and their ports are all defined through Lines 2-10. The workflow interface provides a couple of inputs and a single output. The input "a" is used to invoke "p1.Op1", whereas the input "b" is used to invoke "p2.Op2". The outputs of "Op1" and "Op2" outputs are passed as input parameters "par1" and "par2" to "p3.Op3" as shown through Lines 17-18. In this example, "p3.Op3" can be executed only when all the input parameters required for its execution become available at run-time.

```

01 workflow aggregation
02 description d1 is http://ec2-54-80-6-125.compute-1.
   amazonaws.com/services/Service1?wsdl
03 service s1 is d1.Service1
04 port p1 is s1.Port1
05 description d2 is http://ec2-54-80-6-122.compute-1.
   amazonaws.com/services/Service2?wsdl
06 service s2 is d2.Service2
07 port p2 is s2.Port2
08 description d3 is http://ec2-54-80-6-124.compute-1.
   amazonaws.com/services/Service3?wsdl
09 service s3 is d3.Service3
10 port p3 is s3.Port3
11 input:
12   int a, b
13 output:
14   int c
15 a -> p1.Op1
16 b -> p2.Op2
17 p1.Op1 -> p3.Op3.parameter1
18 p2.Op2 -> p3.Op3.parameter2
19 p3.Op3 -> c

```

Listing 3.8 Specification of the data aggregation pattern based on Figure 3.7.

3.3.3 Data Distribution Pattern Specification

Orchestra's compiler can automatically detect parallelism by analysing the data dependencies indicated explicitly in the workflow specification. Hence, the order of service invocations is irrelevant and control flow structures are not required to describe parallelism. However, the simplest parallel data structure in *Orchestra* represents a finite sequence of service invocations based on the data distribution pattern. Figure 3.9 shows this pattern where an input '*a*' is passed to service '*s*1' which produces an output whose value is passed to services '*s*2' and '*s*3'. These services are executed concurrently, and produce '*b*' and '*c*' respectively. In this example, the initial input '*a*', and the final outputs '*b*' and '*c*' are all of integer type. It is essential to note that the type of the output provided from service '*s*1' which is the same type as that of the input parameters of the services '*s*2' and '*s*3'.

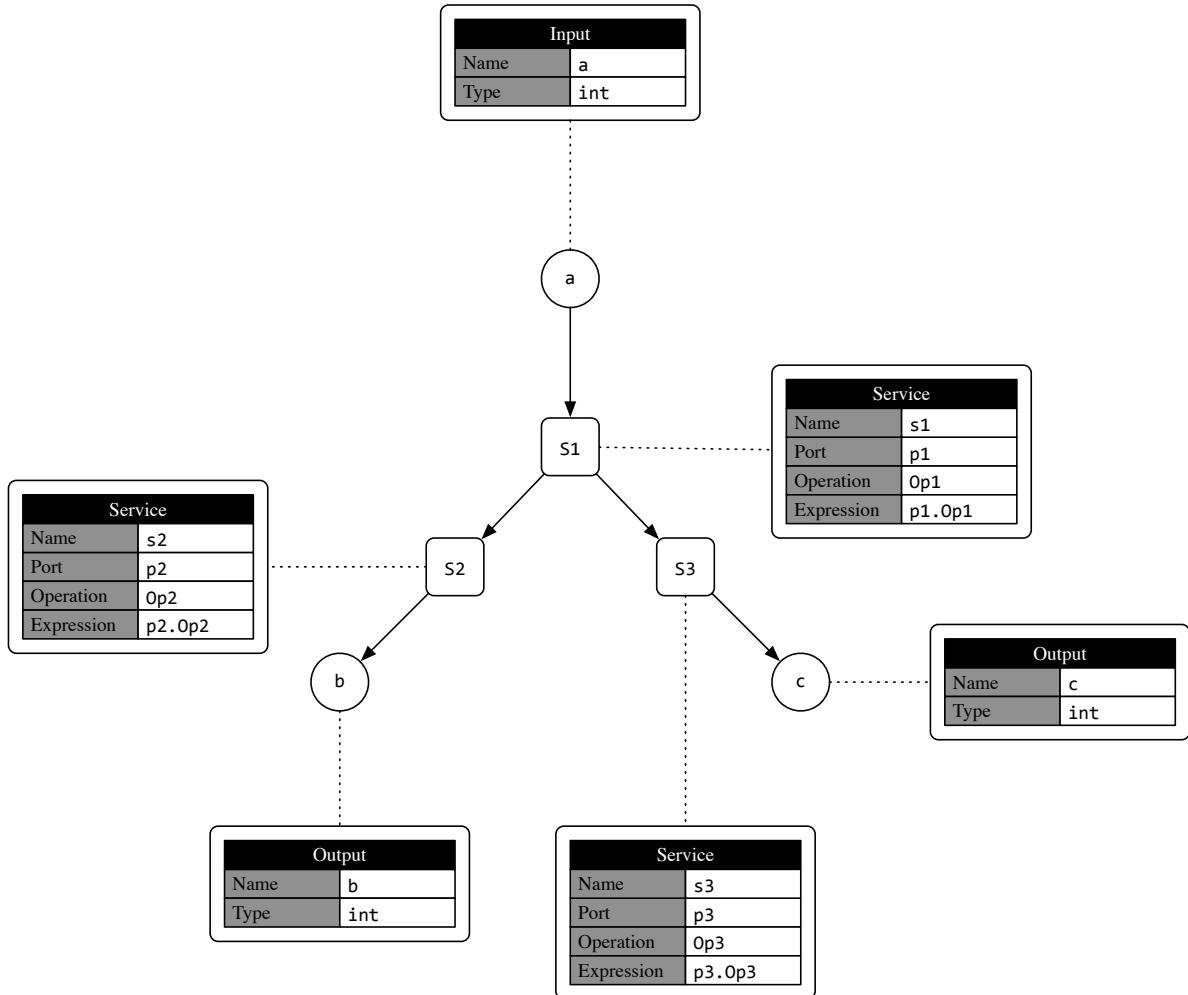


Fig. 3.9 Dataflow graph of services composed using the distribution pattern.

Listing 3.10 provides this pattern's specification in a workflow called 'distribution'. In this specification, service description document references, services, and port identifiers are all defined through Lines 2-10. The workflow interface consists of a single input and a couple of outputs which are all of integer type. The input 'a' is used to invoke 'p1.Op1'. The invocation's output is then used to invoke 'p2.Op2' and 'p3.Op3' separated by a ',' symbol in Line 16. Typically, the operations 'p2.Op2' and 'p3.Op3' are invoked independently of each other and in concurrent fashion. The invocations' results are obtained through Lines 17-18.

```

01 workflow distribution
02 description d1 is http://ec2-54-80-6-125.
    compute-1.amazonaws.com/services/Service1?wsdl
03 service s1 is d1.Service1
04 port p1 is s1.Port1
05 description d2 is http://ec2-54-80-6-122.
    compute-1.amazonaws.com/services/Service2?wsdl
06 service s2 is d2.Service2
07 port p2 is s2.Port2
08 description d3 is http://ec2-54-80-6-123.
    compute-1.amazonaws.com/services/Service3?wsdl
09 service s3 is d3.Service3
10 port p3 is s3.Port3
11 input:
12     int a
13 output:
14     int b, c
15 a -> p1.Op1
16 p1.Op1 -> p2.Op2, p3.Op3
17 p2.Op2 -> b
18 p3.Op3 -> c

```

Listing 3.10 Specification of the data distribution pattern based on Figure 3.9.

3.4 Distributed Computation

This thesis discusses a workflow partitioning approach that is presented in chapter 4 for decomposing a workflow into smaller parts that can be distributed across the network. *Orchestra*'s computation logic can be moved from one network location to another between multiple compute servers called *execution engines*. Execution engines are web services themselves and hence are treated as such by *Orchestra*, and these engines can be uniquely identified by their web address location. Each engine maintains a table that consists of information about the workflows it is executing or has executed, and holds data relevant to these workflows using a persistent datastore. *Orchestra* provides abstractions to define references to execution engines which are discussed in Section 4.4, but these engines are used exclusively by the workflow execution architecture. For instance, information about the ex-

ecution engines may be obtained by the compiler and used during workflow partitioning and placement to determine the appropriate engines to execute the workflow partitions. Hence, the user does not need to indicate which parts of the workflow need to be partitioned and on which engines should they be executed, or how the engines communicate with each other. In order to support distributed computation, *Orchestra* must provide abstractions to support collaboration between execution engines. Such abstractions can be used to form statements in the specifications of workflow partitions, which may indicate the following:

- **Execution engines:** the execution engines may be defined by the user in the original workflow specification, and in the specification of workflow partitions.
- **Data forwarding:** the workflow partitions may provide statements that indicate the dataflow between the execution engines.
- **Unique identification information:** the workflow partitions must provide abstractions to introduce unique information about the workflow in order to distinguish it from others during execution.

These abstractions are discussed in Section 4.4.2 which discuss the workflow partitioning approach in detail.

3.5 Type System and Data Structures

Orchestra is a strongly-typed language that uses a group of base types expressed in a type system matching those defined in the XML Schema standard [82]. It supports record types that combine base types together, which can be obtained from external data type schemas or specified manually by the user. Furthermore, it supports record-like data structures such as arrays and tuples.

3.5.1 Base Types and Scalars

Orchestra provides base and scalar types matching those defined in the XML Schema Specification [82]. Base types include *byte*, *boolean*, *string*, *int*, *float*, *double*, *decimal*, *long*, *short*, *date*, *time*, and the union type *any*. These base types are used to indicate the types of scalars, which contain exactly one value such as a number, a literal string, etc. Typically, they are encoded in network messages between web services to support interoperability. For example, the SOAP [73] protocol includes a built-in set of rules for encoding these data types which enables a message transmitted to and from a service to indicate specific types of the data entities it encapsulates. Hence, *Orchestra* supports the encoding rules of these types automatically to hide the details of communication between web services from the user.

3.5.2 Data Structures

This section presents different kinds of data structures supported by *Orchestra*. These data structures include:

1. Record types (e.g. external data types obtained from XML data type schemas).
2. Arrays.
3. Tuples.

Record Types

Record types can be obtained from external XML data type schemas [82]. *Orchestra* permits references to these schemas to be defined in the workflow specification, from which a schema type can be used to declare an input or output variable in the workflow interface. Listing 3.11 provides the syntax for defining an external data type schema reference '`sch`' using the '`schema`' keyword, where the '`is`' keyword is followed by the location of the data

type schema document in Line 1. In this example, `'var'` is defined as a workflow output of type `'newType'` which is obtained from the schema reference `'sch'` in Line 5.

```

1 schema sch is http://ec2-54-80-3-122.compute-1.amazonaws.com/
  types.xsd
2 input:
3   int a
4 output:
5   sch:newType var

```

Listing 3.11 Defining an external data type schema reference.

Orchestra's compiler obtains the external data type schema document, analyses it and generates a data structure for the output `'var'` that is equivalent to the structure of the type `'newType'` as defined in the data type schema document. Records are treated normally like any other variable, and their elements can be used as input parameters to services invocations or hold their results. Listing 3.12 shows an expression `'var.x'` that consists of two parts separated by a `'. '` symbol. The first part `'var'` represents a variable object of record type, whereas the second part `'x'` represents an element (e.g. parameter) of that object which holds a value. This parameter has a type typically specified in the data type schema.

```
var.x
```

Listing 3.12 Expression of a record and its element based on Listing 3.11

Listing 3.13 shows how to pass a record element as an input to a service operation where `'var.x'` is an expression that indicates the record element to be used as input and `'p1.Op1'` is the service operation.

```
var.x -> p1.Op1
```

Listing 3.13 Service invocation with a record element as input based on Listing 3.11.

Record elements can be evaluated only once. Each record element can be assigned a scalar value, a variable value, or a future that is obtained from a service invocation. Listing 3.14 shows the assignment of a scalar value `'7'` to the record element `'x'`, the assignment of

variable `'a'` value to the record element `'y'`. The record element `'z'` is evaluated once the future result of the service invocation `'p1.Op1'` becomes available.

```
1 var.x = 7
2 var.y = a
3 p1.Op1 -> var.z
```

Listing 3.14 Evaluation of record elements based on Listing 3.11.

Arrays

Orchestra supports *array* records that have fixed size with zero-based indexing once allocated in the workflow interface. Like any other functional data structure, array records are *immutable* which means that their contents cannot be modified but only queried. Furthermore, the elements of an array are evaluated on-demand and this is known as *lazy* evaluation.

Listing 3.15 shows how to create arrays `'a'`, `'b'`, and `'c'` in the workflow interface. In this example, both arrays `'a'` and `'b'` are specified with a fixed size, whereas the size of array `'c'` is not specified. This permits a service output that is structured as an array that contains seven elements of integer type to be stored in array `'c'` for example, whereas a service output that is structured as an array of unknown size whose elements are of float type can be stored in array `'c'`.

```
1 input:
2   double x
3   double y
4   double[7] a
5 output:
6   int[7] b
7   float[] c
```

Listing 3.15 Creating arrays in the workflow interface.

Array elements can be evaluated at once using scalars and the values of other variables as shown in Listing 3.16 where the first five elements of the array `'a'` are given scalar values, and the last two elements are given the values of variables `'x'` and `'y'` respectively.

```
x = 6.0
y = 7.0
a = [1.0, 2.0, 3.0, 4.0, 5.0, x, y]
```

Listing 3.16 Evaluating an array based on Listing 3.15.

Arrays can be passed as inputs to service operations and may be assigned values only once.

For instance, Listing 3.17 shows how to invoke an operation `'p1.Op1'` with an array element `'a[0]'` as input in line 1, and how to store the future result of the operation in array `'b'` in line 2.

```
1 a[0] -> p1.Op1
2 p1.Op1 -> b
```

Listing 3.17 Using arrays in service invocations based on Listing 3.16.

Tuples

Orchestra permits values to be built into record-like structures using *tuples* of a certain length. Tuples consist of *immutable* elements whose values are evaluated on-demand, and can be used when the programmer knows in advance how many values are to be stored. Unlike array elements, the elements of a tuple do not need to be of the same type. Tuples are created within brackets with elements delimited by commas in the workflow interface as shown in Listing 3.18, where `'a'` represents a tuple with two elements of the same type, and `'b'` represents a tuple with three elements each of a different type.

```
1 input:
2   [int, int] a
3 output:
4   [int, double, boolean] b
```

Listing 3.18 Creating tuples in the workflow interface.

Tuples elements can be evaluated and used in the same manner as arrays to provide a uniform approach for treating variables in *Orchestra*.

3.6 Conclusion

This chapter presented a novel high-level data coordination language called *Orchestra* for the specification and execution of service-oriented workflows, which is characterised by its simplicity, support for determinism, parallelism, and data-driven execution, and its strong type system. This language was presented and discussed by Jaradat et al. in [266], [267], and [268]. *Orchestra* provides simple abstractions that are used only for defining services, invoking, and composing them. It is deterministic in the sense that it supports referential transparency (e.g. free from side-effects, and pure), and allows a workflow to be specified in a manner such that the compiler can effortlessly produce highly scalable, distributed form of the workflow. *Orchestra*'s computation model permits each workflow task (e.g. service operation) to be invoked only when all its required inputs become available. Furthermore, *Orchestra*'s type system can be used to ensure the correctness of the workflow specification, prevent mistreatment of data entities, support service interoperability.

Based on the understanding of the existing workflow languages and their limitations, *Orchestra* addresses a unique form of distributed programming that permits the computation logic to be decomposed into smaller parts that may be moved towards the services providing the data in the workflow for execution. The primary aim of this work is to provide greater understanding and hopefully a stronger bases on which to create a new family of workflow languages that support distributed computation. This thesis have given precedence to composing workflows in an intuitive manner such that the workflow specification can be compiled into a highly scalable distributed form. However, there is plenty of room for improvement as not all fundamental issues can be addressed at once. For example, *Orchestra*'s design can be extended to support high-level constraints for deploying specific parts of the workflow based on external requirements specified by the user, enforce security over data transfer between web services and execution engines, and handle failures at run-time. The author of this thesis believes that this programming approach will evolve with experience.

Chapter 4

System Architecture Design

4.1 Introduction

This chapter presents the architectural design of a decentralised service-oriented orchestration system that relies on the workflow specification language presented in Chapter 3. This system permits a workflow to be decomposed into smaller parts, and determines the most appropriate network locations at which these parts may be executed efficiently. It relies on multiple engines to execute the workflow. Each engine is responsible for executing part of the workflow, and exploits connectivity to a set of services by invoking them, collecting and forwarding the invocation results to locations where they are required. The remainder of this chapter is organised as follows: Section 4.1.1 provides an overview of the proposed orchestration system with a high-level architectural description of its components. Section 4.1.2 describes the interactions between these components and their configurations. Section 4.2 details the design of the execution engine by describing its internal modules and their interactions. Section 4.3 discusses the functional aspects of the system relating to the compilation and partitioning of workflows, network resource monitoring, placement analysis, deployment, and execution. Section 4.4 provides a workflow scenario that is used to explain the stages of decentralised service orchestration. Section 4.5 finally concludes this chapter.

4.1.1 Architecture Overview

This architecture presents a service-oriented environment that consists of multiple endpoints that represent workflow participants including distributed services and execution engines. Each engine is responsible for executing part of the overall workflow. It permits a workflow to be decomposed into executable parts, and employs network resource monitoring and computation placement analysis to determine the most appropriate engines onto which these parts are deployed for execution. In this architecture, the execution is data-driven as the workflow tasks (e.g. service operations) are invoked as soon as the input data required for their execution becomes available from other sources (e.g. remote services or engines). This architecture relies on a high-level data coordination language called *Orchestra* for the execution of web service workflows, the design of which is provided in Section 3.2. It aims to automate the orchestration of a web service workflow through:

1. Parallelising the workflow by decomposing it into smaller executable parts that can be deployed onto computing machines.
2. Selecting the most appropriate workflow engines that can execute the workflow parts.
3. Determining the inter-connection topology between the engines and how they communicate with each other.
4. Deploying the individual workflow parts to the chosen workflow engines.
5. Triggering the execution of the workflow parts that are initially required to start the workflow.
6. Monitoring the execution of the overall workflow to deal with emergent run-time issues such as handling unexpected errors. This may involve repeating any or all of the above activities to re-deploy the workflow parts.

Figure 4.1 provides an architectural overview of the decentralised system that shows the interactions amongst multiple engines and services which may be hosted in different network regions represented by a cloud shape. This thesis defines network regions as distinct geographic areas around the world where hosting servers of the execution engines and the web services are based. For example, a particular web service may run in a data center located in California. The directed edges labeled (E-E) represent engines' interactions, whereas those labeled (E-S) represent engine and service interactions. These interactions are discussed in Section 4.1.2.

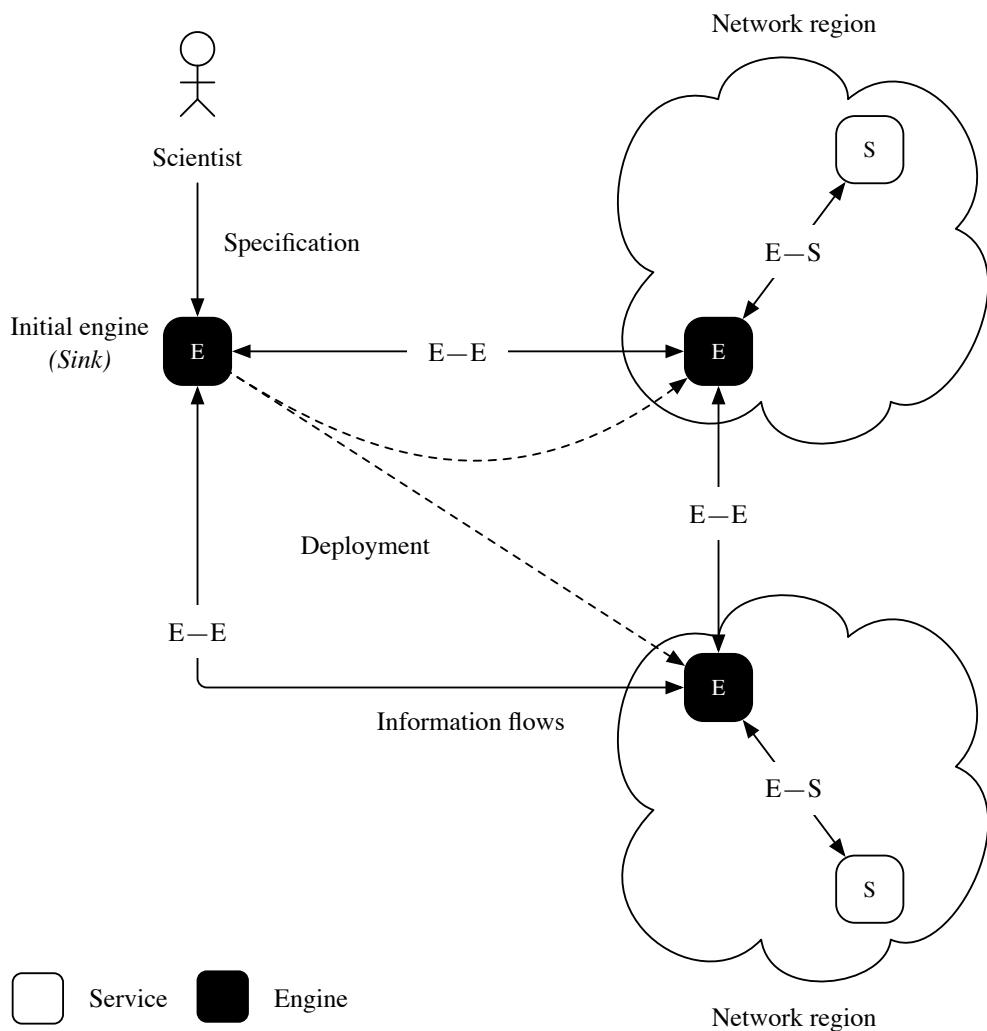


Fig. 4.1 Decentralised orchestration architecture.

4.1.2 Interactions and Configuration of Components

Each engine may interact with one or more services hosted in the same network region, and may interact with multiple engines hosted in remote network regions as shown in Figure 4.1. These interactions are classified into:

1. Engine and service interactions (E-S).
2. Engines' interactions (E-E).

Engine and Service Interactions

Centralised orchestration architectures rely on a single execution engine to interact directly with all services. Using the architecture shown in Figure 4.1, several engines collaborate together to execute the overall workflow. Each engine interacts with one or more services (e.g. *one-to-many* relationship), and may be employed as a *proxy* component to invoke services that are located in the same network region where the engine is hosted. Service invocations take place as soon as all the input data that is required for their invocation becomes available from other sources (e.g. services, or remote engines). For example, this input data may represent the results of invoking other services or intermediate data in the workflow transferred from remote engines. Following a service invocation, the engine collects the invocation results and stores them locally to be used in the future when needed during the workflow execution.

Engines' Interactions

The execution engines interact with each other in different ways. During the workflow execution, multiple engines can communicate with each other (e.g. *many-to-many* relationship) by transferring data segments relevant to the workflow execution. For example, a service invocation result may be forwarded by an engine that collected it to a remote engine that

requires it in order to execute a particular workflow partition. This permits the overall execution of the workflow to progress. Before execution, the *initial engine* which is responsible for compiling and partitioning a workflow may communicate with other engines to retrieve network resource information needed for placement analysis and workflow partitioning. In this architecture, a single engine may act as an ultimate data *sink* that receives the final workflow results from remote engines. Typically, this engine is defined manually in the workflow specification and does not represent a centralised bottleneck point as it is not responsible for invoking services hosted at remote network locations in different geographic regions. Furthermore, this engine may be the initial engine that compiled and partitioned the workflow, and transmitted its partitions to distributed engines for execution.

Configuration of Engines

Execution engines are preferably installed and configured “closer” to the services based on factors that may influence the overall execution performance. These factors include the geographical distance and *Quality of Service* (QoS) properties, and are described as follows:

- **Geographical distance factor:** This factor may be used generally to indicate the most appropriate network region at which to execute a sub workflow efficiently. Based on the analysis of the experiments discussed in Chapter 6, the geographical distance should be considered as a coarse factor that identifies a potential network region for executing a workflow. However, it is insufficient to select a group of candidate engines specifically to execute the workflow.
- **Quality of Service (QoS) factors:** These factors are used to estimate the network distance between a particular engine and a service in the workflow, and include the network latency, and network bandwidth. The network latency represents the length of time required for a request message sent by an engine to reach a service, and the time required for the service to acknowledge the engine’s request by transmitting a

response message. The network bandwidth represents the total amount of information that can be transmitted between a particular engine and a service in a given time.

This thesis argues that the communication overhead between an engine and a particular service can be minimised if the engine is configured “closer” to the service. Depending on the workflow partitioning and placement analysis approach presented in Section 4.3, an engine can be responsible for invoking a single service or multiple services. It may not be possible to install an engine on the same server hosting the service, or within the same network domain due to restrictions imposed by the service provider or administrator. However, performance benefit can still be gained by harnessing the connectivity of available engines installed across different network regions as discussed in Chapter 6.

4.2 Engine Design

The architecture consists of execution engines which are identical in design. Each engine consists of internal modules that provide a unique set of capabilities. These modules cooperates with each other to support the execution of service-oriented workflows. This section presents the design of the engine’s internal modules and describes their interactions, and it is organised as follows: Section 4.2.1 describes the internal modules of the architecture’s execution engine component, whereas Section 4.2.2 and discusses the interactions between these modules.

4.2.1 Description of the Engine’s Internal Modules

Figure 4.2 provides an architectural overview of the engine and its internal modules and the interactions between themselves, the end user (e.g. scientist), and the execution environment. This section provides a concise description of each of these modules which include:

1. Compiler module.

2. Partitioner module.

3. Analyser module.

4. Deployer module.

5. Monitor module.

6. Executor module.

7. Datastore module.

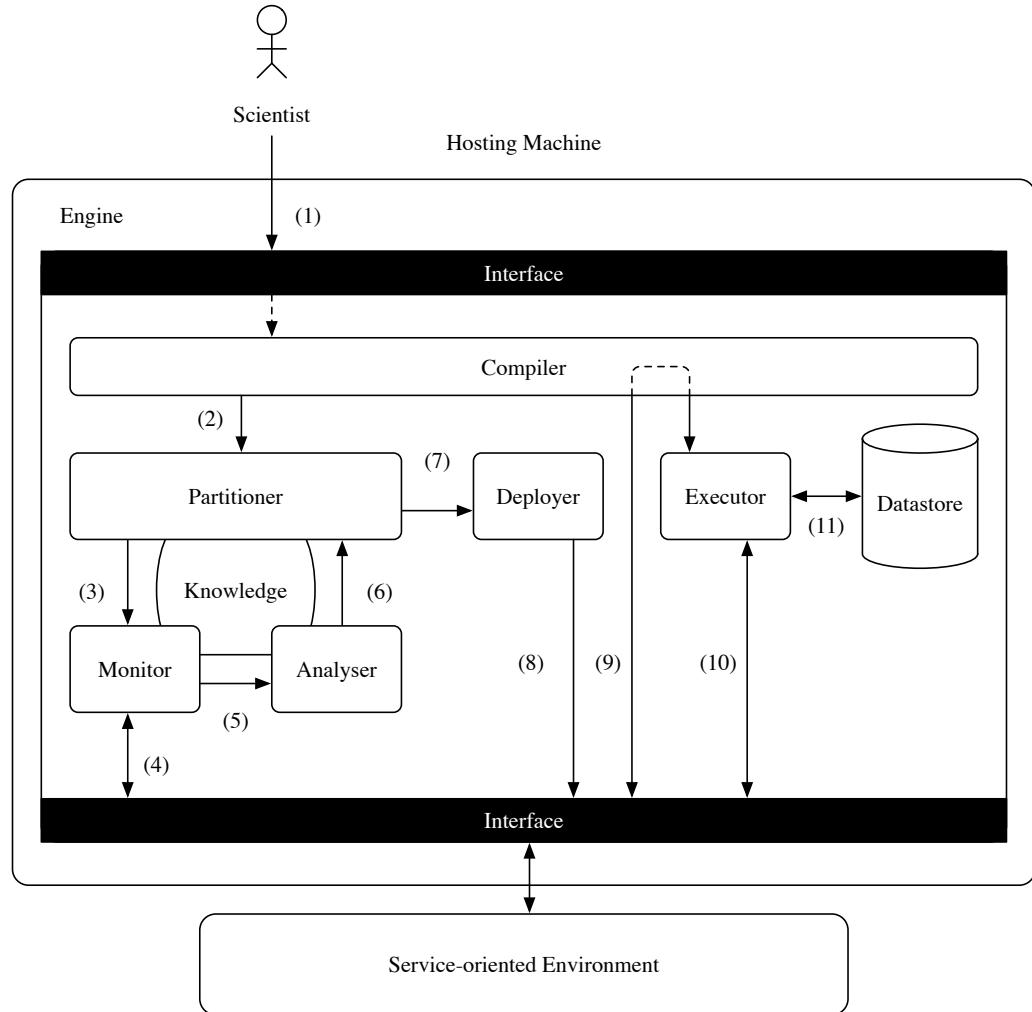


Fig. 4.2 The execution engine's internal modules and their interactions.

Compiler Module

The compiler module is built from a set of procedures matching the production rules of the *Orchestra* language. This compiler ensures the correctness of the workflow specification. It constructs a directed acyclic graph data structure that represents the workflow in which the vertices are computational tasks (e.g. service invocations), with edges between them that represent the data dependencies between the tasks. This data structure can be traversed, analysed, and decomposed into smaller parts that may be distributed to remote engines for execution. This compiler ensures the correctness of the workflow specification, and constructs an executable data structure that represents the workflow which can be analysed and distributed to remote locations.

Partitioner Module

This module is responsible for decomposing the workflow into smaller partitions (e.g. sub workflows) and modifying the overall structure of the workflow or its partitions as necessary. It relies on the analyser module to partition the workflow and produce a plan for the deployment of the overall workflow onto execution engines.

Analyser Module

This module performs placement analysis using information collected from the execution environment to determine candidate engines for executing the workflow partitions. It may cooperate with the partitioner module to restructure the overall workflow by decomposing it into smaller partitions as necessary.

Deployer Module

Based on the placement analysis outcome, this module generates a deployment plan for transmitting the workflow partitions to remote engines and triggering their execution.

Monitor Module

This module monitors network resources such as the remote engines and services involved in the workflow. It collects QoS metrics that relate to these resources such as the network latency and bandwidth metrics. These metrics are typically stored in a *knowledge base* module, and can be used in placement analysis to assist in decisions that relate to the partitioning of the workflow and its deployment.

Executor Module

This module is responsible for invoking services, collecting the invocations' results, and forwarding these results to remote engines as necessary.

Datastore Module

This module is used to maintain information and data relevant to the workflows being executed or have been executed by the engine. The implementation of this module is described in Section 5.4.7.

4.2.2 Interactions of the Engine's Internal Modules

The engine's modules cooperate to compile a workflow specification, partition the workflow into smaller sub workflows, perform placement analysis to determine the most appropriate network locations at which the partitions may be executed, and generate a plan to deploy the workflow partitions onto multiple engines for execution in a monitored service-oriented environment. Based on the illustration provided in Figure 4.2, the interactions between these modules are enumerated as follows:

1. The compiler accepts a workflow specification as input.
2. The compiler generates a dataflow graph and passes it to a partitioner.

3. The partitioner instructs the monitor to collect information from the environment to assist in the workflow partitioning process.
4. The monitor gathers QoS metrics relating only to the network latency and bandwidth between the services and the engines participating in the workflow. These metrics are discussed in section 4.3.3 of this chapter.
5. The information gathered by the monitor are passed to the analyser, which attempts to determine the most appropriate network location at which to execute the partitions.
6. The placement analysis results are passed to the partitioner, which relies on these results to restructure the workflow as necessary and generate a deployment plan of the workflow partitions.
7. The partitioner passes the deployment plan to the deployer component.
8. The deployer transmits workflow partitions to remote engines for execution.
9. The compiler analyses a workflow partition specification and generates its dataflow graph, which is then passed to the executor component.
10. The executor invokes services and communicates with remote engines.
11. The executor reads and writes data from and to a persistent datastore.

4.3 Decentralised Orchestration Stages

Thus far this chapter has presented an architectural overview of a decentralised service-oriented orchestration system, and the design of its workflow engines, their internal modules, and the interactions between them. This section discusses the orchestration stages that relate to the behaviour of the workflow engine. These stages include the following:

1. Compilation.
2. Partitioning.
3. Network resource monitoring.
4. Placement analysis.
5. Deployment and execution.
6. Execution monitoring.

4.3.1 Compilation

The architecture’s workflow engine uses a recursive descent compiler that is built from a set of procedures matching the production rules of the *Orchestra* language grammar. This compiler parses a given workflow specification and analyses it to ensure its correctness. It performs semantic matches against distributed system components including services and execution engines specified in the workflow. It does not generate machine code representation from the workflow specification, instead it constructs a data structure that represents a *dataflow graph* in which the vertices are service operations to be invoked with edges between them as data dependencies. This data-driven workflow representation permits its data structure to be decomposed into independent parts that can be distributed across multiple machines and executed in parallel. Furthermore, it allows the workflow to be maintained upon its distribution such that it can be refactored for re-deployment to deal with emergent run-time issues as discussed in Sections 4.3.5 and 4.3.6.

4.3.2 Partitioning

This section discusses the partitioning and the encoding stages of decentralised orchestration. Partitioning is responsible for decomposing the workflow logic into smaller parts (e.g.

sub workflows) that can be executed in parallel. This permits the workflow logic to be moved “closer” to the services providing the data in the workflow and executed instead of moving the implementation of the services themselves. For example, a workflow partition can be executed using a workflow engine that is hosted within short network distance to a particular set of web services. Encoding is responsible for transforming the workflow partitions to a form that is suitable for distribution across the network, and before they are deployed onto the remote engines for execution. Partitioning the workflow logic into smaller sub workflows and encoding them into an appropriate format for distribution requires gaining insight about the complexity of the workflow to detect its parallel parts following the compilation stage. Hence, a traverser is used to explore the data structure of the workflow generated by the compiler. This traverser obtains useful information relating to the workflow inputs, outputs, services, service operations and the type representations associated with their input parameters and outputs. This information may be used to introduce modifications to the dataflow graph structure as necessary during partitioning. Figure 4.3 shows the partitioning algorithm. Based on this algorithm, workflow partitioning consists of the following steps:

1. Firstly, the dataflow graph information is decomposed into the maximum number of smallest sub workflows that can be isolated for parallel execution. Each isolated sub workflow consists of a single service invocation which represents the smallest unit of computation. Therefore the result of this decomposition is a set of sub workflows based on the process pattern, each of which may require a new inputs set and provides a set of outputs that may be required for executing other sub workflows.
2. Secondly, network resource monitoring is used to determine available engines that can be used to execute the sub workflows and to detect the network condition between the engines and the services involved in the workflow.
3. Thirdly, placement analysis is used to determine the most appropriate execution engine for each sub workflow. Placement analysis relies on the knowledge about the

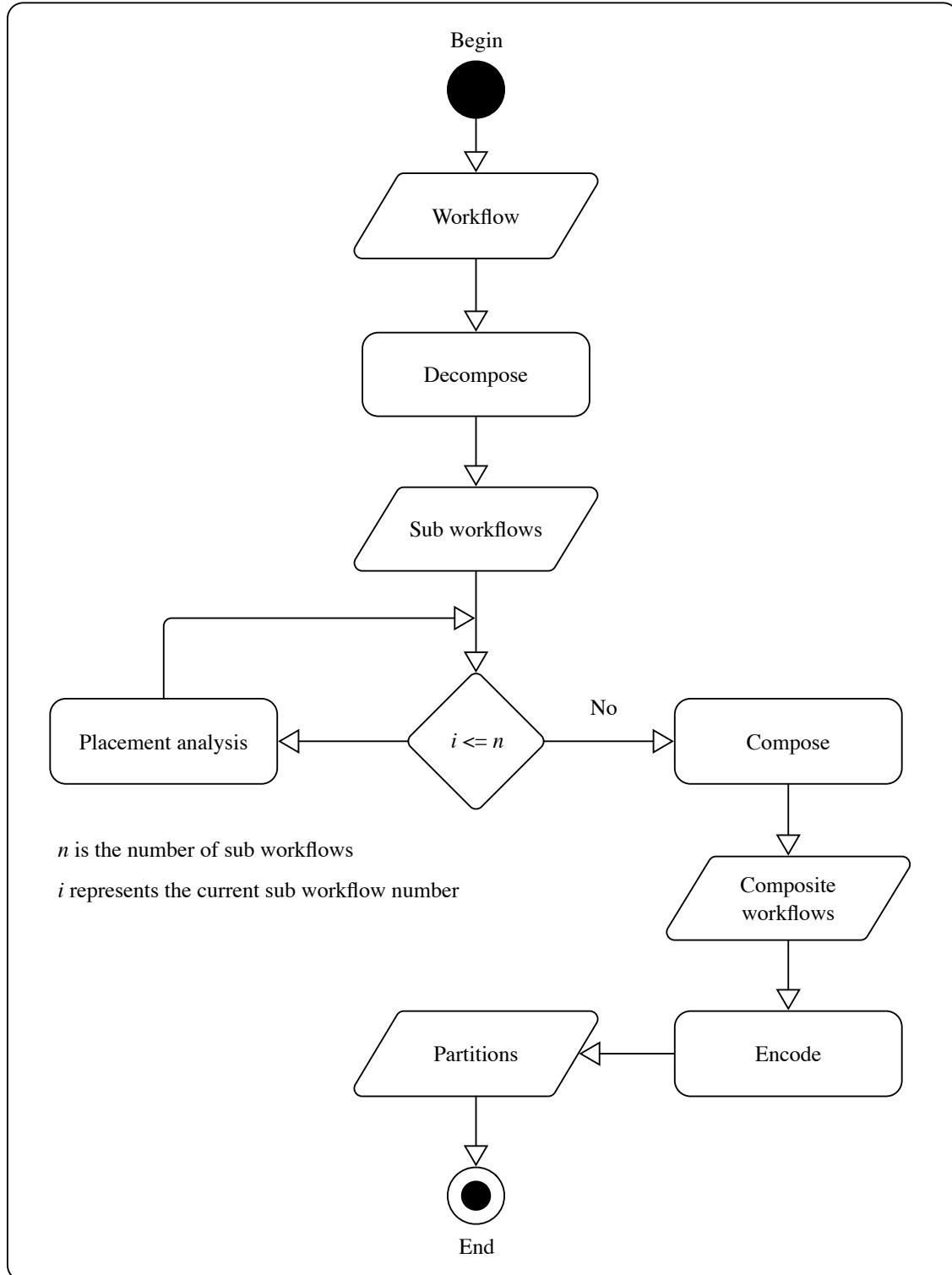


Fig. 4.3 Workflow partitioning algorithm.

network condition and a heuristic algorithm to select a candidate engine. Section 4.3.4 discusses placement analysis.

4. Finally, based on placement analysis the sub workflows may be combined together if the same engine is selected to execute them. This involves introducing directed edges between the sub workflows wherever a data dependency exists. Consequently, the composite workflows are encoded using the same language as used to specify the whole workflow. During recoding, relevant information such as the workflow inputs, outputs, services, service operations, data dependencies and type representations are all captured, and associated with the composite workflows to make each a self-contained stand-alone workflow specification known as a workflow partition. Section 4.4.2 discusses the encoding stage in detail.

4.3.3 Network Resource Monitoring

Network resource monitoring begins following workflow partitioning to construct a logical network topology. This topology represents an indirect graph in which the vertices are network resources, where the edge between any pair of vertices represents the probable network distance between them. Constructing this topological view involves reachability analysis which determines if the engines can communicate with the services and with themselves, after which Quality of Service (QoS) are collected from probing the network resources to calculate the probable network distance between them. These metrics include the following:

- **Network latency:** This metric represents the length of time that takes a request message sent by an engine to reach a service, and the time required to acknowledge the engine's request by transmitting a response message.
- **Network bandwidth:** This metric represents the total amount of information that can be transmitted between a particular engine and a service in a given time.

These QoS metrics can be useful in placement analysis for selecting the nearest engine to a service for executing a particular sub workflow, where the engine has short latency and high bandwidth with the service. Such metrics indicate faster time for transferring data between the engine and the service. Section 5.4.3 discusses the implementation of the network resource monitor module and the mechanisms used for detecting and collecting these metrics. There are a number of reasons for choosing these metrics in particular. These reasons are discussed as follows:

1. Firstly, web services may be hosted on heterogeneous machine that may be physically or virtually present at network locations in different geographic regions. This can influence the overall speed of transmitting information from one network location to another across different administrative domains and such physical limitations has significant consequences for distributed workflow applications relying on global data and computing resources. The network latency between the execution engines and the services must be detected to determine an engine that is "closer" to a particular service in comparison with other engines that may be able to invoke the service.
2. Secondly, network latency and bandwidth suddenly become the principal factors that limit the overall performance of a distributed workflow application regardless of processing power and storage capacity of computing resources. The condition of the global network infrastructure is susceptible to change due to unpredictable congestion that results in temporary fluctuations of bandwidth that may create performance bottlenecks.
3. Finally, service providers may host services with limited accessibility to their hosting machines which may not permit service clients to detect other metrics such as the processing power of the hosting machines and their storage capacity, etc.

4.3.4 Placement Analysis

Placement analysis uses the knowledge about the network with a combination of heuristics for selecting the most appropriate engines onto which the workflow partitions are deployed. Figure 4.4 shows the placement analysis algorithm. This algorithm comprises of the following steps:

1. Firstly, all the available execution engines are organised into groups using a clustering algorithm such as *k-means* that identifies similar engines based on their QoS metrics.
2. Secondly, the groups containing inappropriate engines are eliminated from further analysis by recursively comparing samples obtained from different groups. For instance, the QoS metrics of an engine sample that is selected randomly from one group is compared with the metrics of a sample that is selected from another group. This helps in identifying the engines with high latency and low bandwidth metrics, which are worse than those of engines in other groups. Consequently, a single group remains which represents a collective of candidate engines to execute the workflow partition.
3. Finally, each candidate engine is ranked by predicting the transmission time between the engine and the service to be invoked in the sub workflow using the following simple equation:

$$T = L_{e-s} + S_{input}/B_{e-s} \quad (4.1)$$

where T is the transmission time, L_{e-s} and B_{e-s} are the latency and bandwidth between the engine and the service respectively, and S_{input} is the size of the input that is used to invoke the service. Consequently, the engine with the shortest transmission time is selected. However, if the input data that is required to invoke the service is not available then the engine with the highest bandwidth and lowest latency to the service is selected automatically.

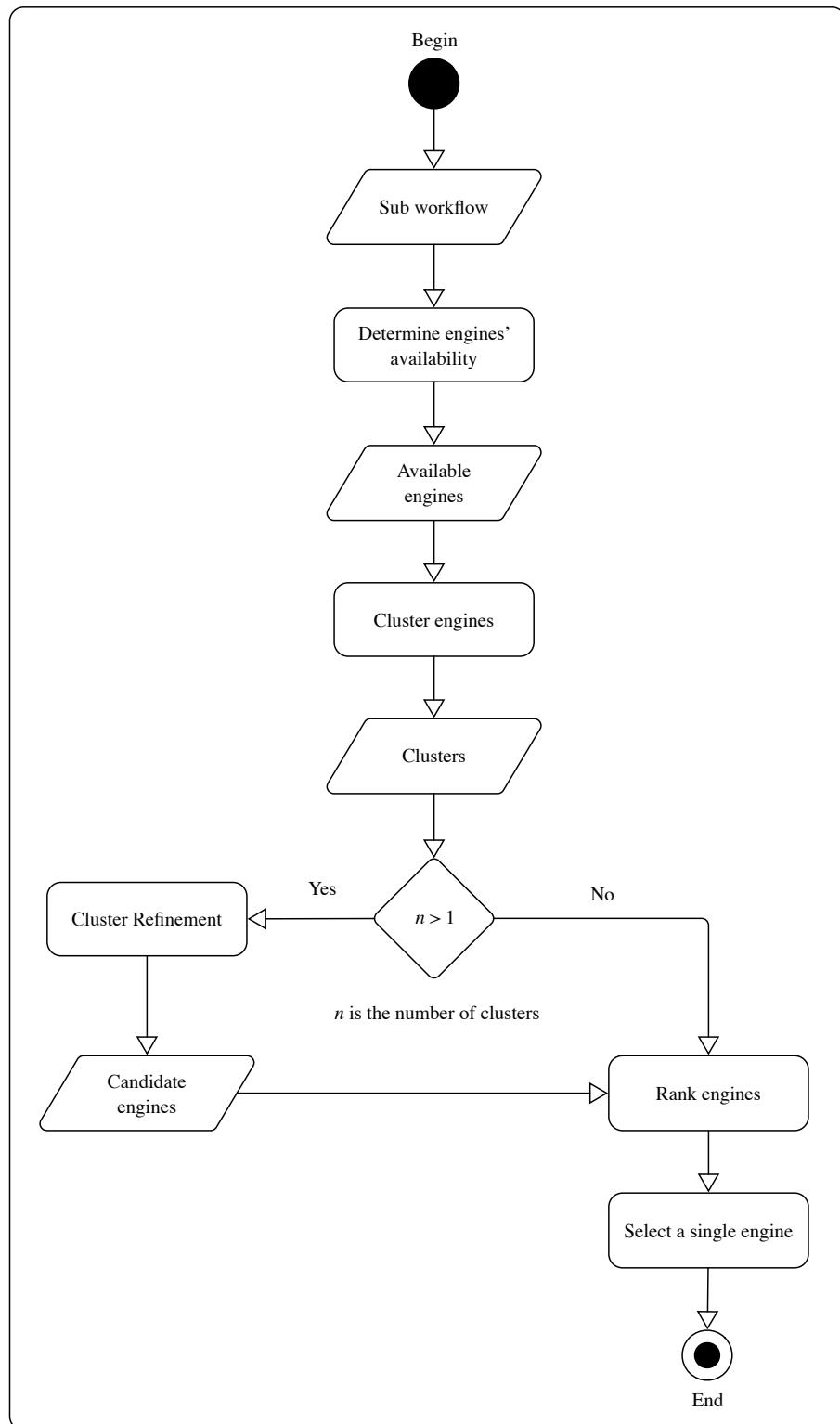


Fig. 4.4 Placement analysis algorithm.

Figure 4.5 shows an arbitrary example of the placement analysis. For each service invocation that needs to be executed in the workflow, all available engines are grouped into clusters as shown in Figure 4.5(a). These clusters are refined by eliminating the engines with inappropriate QoS metrics as shown in Figure 4.5(b). Finally, a single engine is selected to perform the service invocation as shown in Figure 4.5(c).

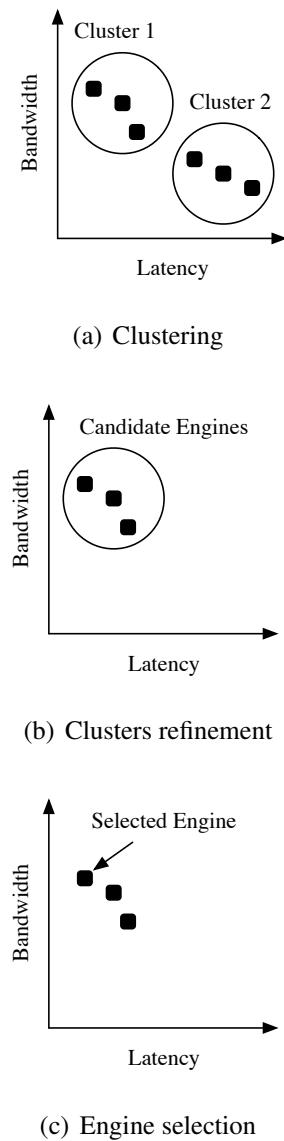


Fig. 4.5 Placement analysis example.

Clustering would be useful for determining the most appropriate workflow engines to execute each workflow task (e.g. service invocation). For example, an engine that has high bandwidth and short network distance (e.g. latency) to a particular service may be selected to execute a particular task by invoking a functionality from that service. This can improve the round-trip time for invoking the service and receiving a response from it. The implementation of this system which is discussed in Chapter 5 employs an iterative *k-means* clustering algorithm that attempts to group engines into a maximum of two clusters where one cluster consists of candidate engines and the other is excluded. The clustering algorithm is performed once only before the execution of the workflow. Currently, the clustering algorithm may take several minutes depending on the number of services participating in the workflow and the available engines (e.g. 5-8 minutes for tens of services and engines). This is because the clustering mechanism relies on gathering QoS metrics relating to these engines and the services from the execution environment. The clustering algorithm's performance has not been evaluated for a large number of services and engines (e.g. exceeding 50) and therefore can be investigated in the future. Future work aims to improve the placement analysis stage by gathering knowledge from the execution environment at run-time, which may be used to optimise the execution of workflows.

4.3.5 Deployment and Execution

Following workflow partitioning, the specification of each workflow partition is dispatched to a designated remote engine that may be located in a different network region. This specification is compiled and analysed by the receiving engine, and may be executed immediately after generating a corresponding dataflow graph of the workflow partition. Executing this dataflow graph involves invoking a set of services as specified in the workflow partition, collecting the invocation results, and forwarding data segments relating to these results to remote engines as necessary. Since this execution process is data-driven, service operations

are only invoked as soon as all the input parameters that are required for their execution become available. This is useful for a number of reasons.

1. Firstly, it permits the inputs to be obtained from different sources (e.g. workflow engines) which automatically forward the values of these inputs to destinations at which they are required.
2. Secondly, there is no communication overhead to request data from remote sources. The workflow engines execute concurrently while waiting for any data relating to their execution from other workflow engines.
3. Finally, centralised scheduling is not required to manage the order in which the workflow partitions are executed.

4.3.6 Execution Monitoring

Decentralisation makes it difficult to deal with emergent run-time issues such as error handling, and failure recovery as the overall workflow state is distributed across multiple engines. The architecture permits the information about the state of all managed workflow partitions to be delivered to and analysed by a centralised entity called the *monitor engine*. Typically, this engine is less resource-constrained than other engines as it is often the initial engine that compiled and partitioned the workflow unless it is specified manually by the user in the workflow specification. Following the execution of a particular workflow partition, its execution engine delivers state information and propagates any unexpected errors encountered to the monitor engine, which in turn analyses this information and performs appropriate actions to handle errors, which include: stopping the execution of the workflow, reporting the error to the user, and reporting the error to the user. Hence, any information about the execution state of each workflow partition including its available and awaiting inputs, final outputs and associated dependencies are all maintained by the monitor engine.

Once an error takes place, this information is used to determine the workflow partitions that have already completed their execution, currently being executed, or awaiting to be executed upon the availability of inputs. This permits the monitor engine to signal all other engines to pause their execution, after which placement analysis may be performed to refactor the workflow partitions whose execution has failed for re-deployment on an alternative execution engine. However, this may require introducing changes to the workflows being executed or awaiting execution. Such changes include the modification of declarative information relating to the engine endpoints, and data routing instructions in the partitions to forward the desired outputs to new locations at which they are required. Consequently, the overall workflow execution resumes upon redeployment.

4.4 Decentralised Service Orchestration Scenario

Based on the architecture presented in Section 4.1.1, decentralised orchestration requires partitioning a workflow into smaller parts (e.g. sub workflows), and deploying the specifications of the sub workflows onto appropriate distributed engines. This section presents an arbitrary workflow orchestration example which is used to explain the system architecture design, and the stages of the decentralised orchestration approach discussed in Section 4.3.

Figure 4.6 shows the structure of a workflow that is used as an example in this section. In

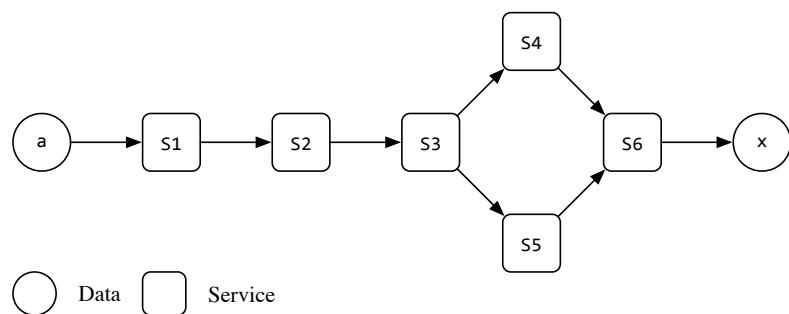


Fig. 4.6 A Directed Acyclic Graph (DAG) workflow.

this example, the input (a) is used to invoke service (S1), which produces an output that is used to invoke (S2) whose output is then passed to (S3). The output of (S3) is used to invoke both (S4) and (S5), whose outputs are used as inputs for (S6), which produces the final workflow output (x).

4.4.1 Workflow Specification Example

Listing 4.7 provides a specification of the workflow shown in Figure 4.6. This specification consists of the following abstractions:

```

01 workflow example
02 description d1 is http://ward.host.cs.st-andrews.ac.uk/
    documents/service1.wsdl
    ..
07 description d6 is http://ward.host.cs.st-andrews.ac.uk/
    documents/service6.wsdl
08 service s1 is d1.Service1
    ..
13 service s6 is d6.Service6
14 port p1 is s1.Port1
    ..
19 port p6 is s6.Port6
20 engine e1 is http://ec2-54-80-3-122.compute-1.amazonaws.com/
    services/Engine?wsdl
21 engine e2 is http://ec2-54-83-2-120.compute-1.amazonaws.com/
    services/Engine?wsdl
22 engine e3 is http://ec2-54-80-6-125.compute-1.amazonaws.com/
    services/Engine?wsdl
23 input:
24     int a
25 output:
26     int x
27 a -> p1.Op1
28 p1.Op1 -> p2.Op2
29 p2.Op2 -> p3.Op3
30 p3.Op3 -> p4.Op4, p5.Op5
31 p4.Op4 -> p6.Op6.par1
32 p5.Op5 -> p6.Op6.par2
33 p6.Op6 -> x

```

Listing 4.7 Specification of the workflow in Figure 4.6.

- **Workflow name:** In this specification, the workflow name `'example'` is defined in line 1 using the keyword `'workflow'`.
- **Service description document references:** Service description document references are declared through Lines 2-7 using the keyword `'description'`. This permits the compiler to retrieve information about the services, their operations and associated types for syntax analysis.
- **Service endpoints:** The `'service'` keyword is used to declare the service endpoint identifiers `'s1'`, `'s2'`, `'s3'`, `'s4'`, `'s5'` and `'s6'` through Lines 8-13. Similarly, the service ports `'p1'`, `'p2'`, `'p3'`, `'p4'`, `'p5'` and `'p6'` are declared using the `'port'` keyword through Lines 14-19.
- **Execution engines:** This workflow specification presents a new abstraction for defining available execution engines that may be used for orchestrating the workflow. The syntax for defining the execution engines `'e1'`, `'e2'`, and `'e3'` is provided through Lines 20-22. These engines are defined using the `'engine'` keyword, where the `'is'` keyword is followed by the location of the engine's web service description document. *Orchestra* provides a default engine identifier called `'sink'` that represents an engine which acts as the ultimate sink for the final workflow outputs. This sink engine can be specified by the user optionally.
- **Workflow interface:** The `'input'` and `'output'` keywords define the workflow interface, which provides an input `a` and an output `x` through Lines 23-26. The number of initial inputs and final outputs and their types depend on the nature of the workflow application specified. In this example, we assume that `a` and `x` types are the same.
- **Service composition:** Service composition is specified through Lines 27-32 the input `'a'` is used to invoke service operation `'p1.Op1'`, whose output is then used to invoke service operation `'p2.Op2'`, which in turn produces an output that is used to

invoke service operation ' $p3.Op3$ '. Following the invocation of ' $p3.Op3$ ', its output is used to invoke both service operations ' $p4.Op4$ ' and ' $p5.Op5$ '. The results of both ' $p4.Op4$ ' and ' $p5.Op5$ ' are used as input parameters to invoke ' $p6.Op6$ '. Finally, the result of ' $p6.Op6$ ' represents the final output ' x '.

The remainder of this chapter provides explains how this example is orchestrated using distributed execution engines.

4.4.2 Workflow Orchestration Example

Based on the illustration provided in Figure 4.8, multiple engines are used to orchestrate this workflow example. This illustration shows the placement of the engines ($E1$, $E2$, and $E3$), and the services ($S1$, $S2$, $S3$, $S4$, $S5$, and $S6$) in different network regions. In this example, engine ($E1$) is responsible for invoking services ($S1$) and ($S2$). Similarly, engine ($E2$) is responsible for invoking services ($S3$) and ($S4$), whereas engine ($E3$) is responsible for invoking services ($S5$) and ($S6$). The orchestration process is conducted as follows:

1. The initial engine ($E1$) compiles the workflow specification in step (1). Some of these partitions may be executed by the initial engine, or distributed across the network to remote engines for execution.
2. Engine ($E1$) may accept some inputs required to execute the workflow from the user such as the input (a) in step (2).
3. Engine ($E1$) transmits a workflow partition to engine ($E2$).
4. Engine ($E1$) transmits a workflow partition to engine ($E3$).
5. The execution begins as engine ($E1$) invokes service ($S1$).

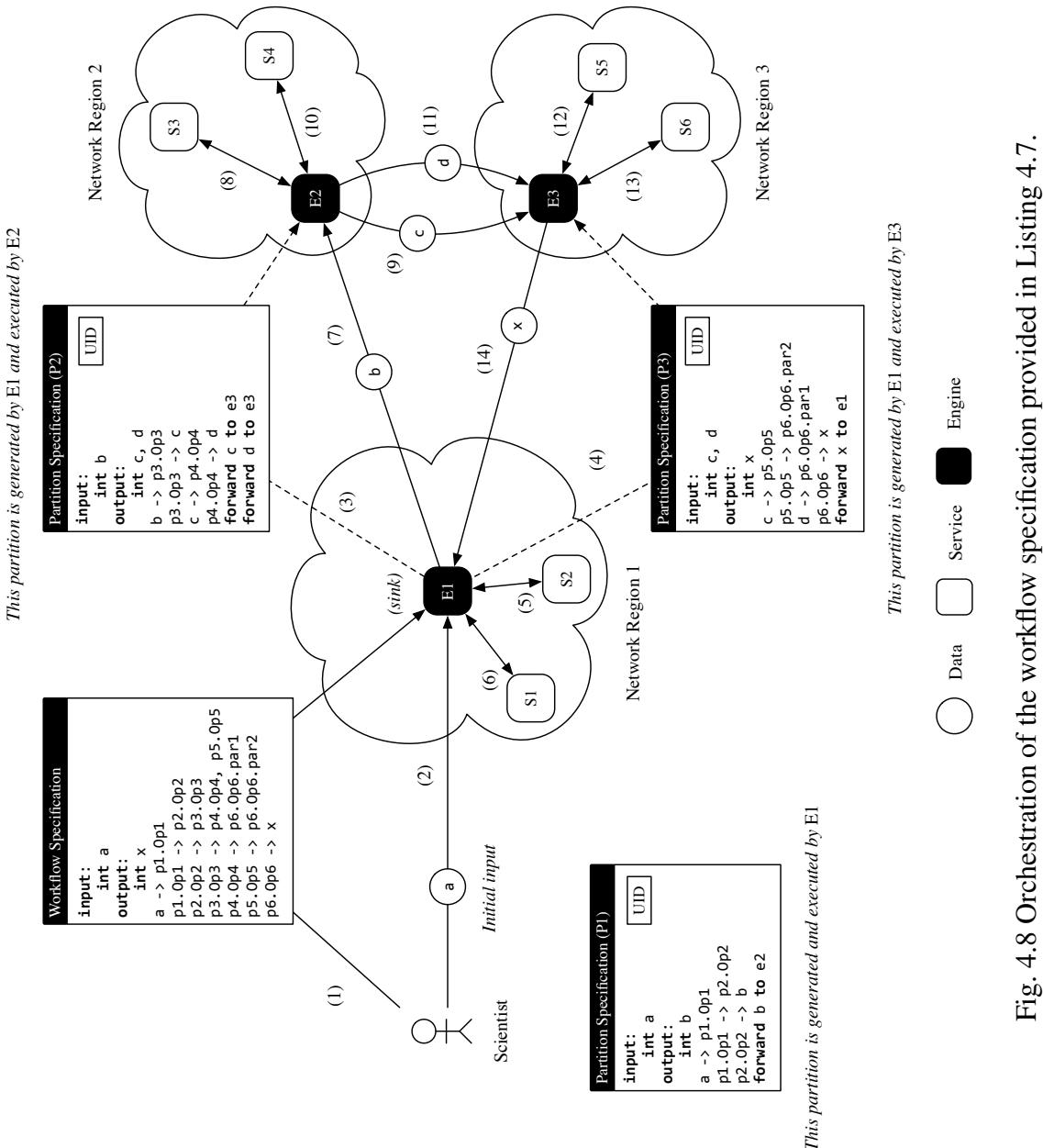


Fig. 4.8 Orchestration of the workflow specification provided in Listing 4.7.

6. The result of invoking service (S1) is used to invoke service (S2).
7. The engine (E1) collects the invocation result obtained in step (6), and transmits a data segment (b) relating to these results to engine (E2).
8. The data segment, (b), is used as an input to invoke service (S3).
9. The result of invoking service (S3) is routed to remote engine (E3). In this example, (c) represents this result.
10. Service (S4) is invoked using engine (E2). In this example, (d) represents the result of this invocation.
11. Engine (E2) forwards the data segment (d) to remote engine (E3).
12. Engine (E3) uses the data segment (c) to invoke services (S5).
13. Engine (E3) uses the data segment (d) and the result of invoking service (S5) as input parameters to invoke service (S6).
14. The result of invoking service (S6) is forwarded to the initial engine which acts as a data sink that stores the final workflow result (x).

Following the decomposition of the workflow into a set of partitions (e.g. sub workflows), the specifications of the partitions are transmitted to remote engines for execution as explained in the steps above. However, the execution of a particular workflow partition may depend on data computed from the execution of other workflow partitions by remote engines. Such data must be automatically transferred once computed to the execution engines that require directly. This is achieved by generating a set of instructions for coordinating the data flows between the engines which may be encoded in the specifications of the workflow partitions. These instructions are generated following the decomposition of the workflow and encoded in the workflow partitions automatically before they are transmitted

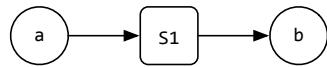
to remote engines for execution. Figure 4.8 shows these computer-generated instructions in the specifications of the workflow partitions. For example, the instruction '**forward c to e3**' means the output 'c' of the workflow partition (P2) must be transferred to engine (E2) which is identified in the specification as 'e2'. Such instructions are based on simple language abstractions that support distributed computation which were briefly discussed earlier in Section 3.4. These abstractions are further discussed in Section 4.4.2 which explains the workflow partitioning approach based on the example provided in this section. The remainder of this section presents discuss the decentralised orchestration stages relating to this example which include:

1. Decomposition of the workflow into smaller sub workflows.
2. Placement of sub workflows.
3. Composition of the sub workflows to form composite workflows.
4. Encoding and related computer-generated abstractions used in the specification of a workflow partition.
5. Specification of the composite workflows.

Workflow Decomposition

This section demonstrates the decomposition of the workflow example presented in Figure 4.6 based on the partitioning algorithm presented in Section 4.3.2. The decomposition process produces a set of independent sub workflows, each of which consists of a single service invocation that accepts a certain amount of inputs and produces a single output. Figure 4.9 shows these sub workflows. Some of these sub workflows require inputs that may be produced by other sub workflows. For example, sub workflow (1) produces an output (b) which can be used as an input to invoke service (S2) in sub workflow (2). Similarly, sub

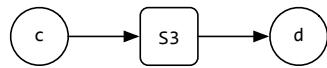
workflow (2) produces an output (c) which can be used as an input to execute sub workflow (3). Sub workflow (6), however, requires both the inputs (e) and (f) which are produced by sub workflows (4) and (5) respectively.



(a) Sub workflow 1



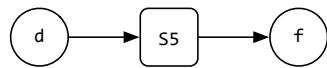
(b) Sub workflow 2



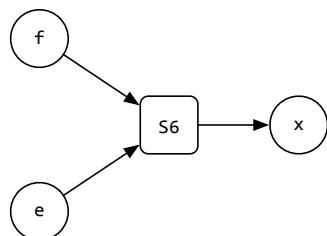
(c) Sub workflow 3



(d) Sub workflow 4



(e) Sub workflow 5



(f) Sub workflow 6



Fig. 4.9 Sub workflows after decomposing the workflow shown in Figure 4.6.

Placement of Sub Workflows

Following the decomposition process, a candidate engine is selected to execute each sub workflow. Table 4.1 provides arbitrary placement analysis results, where three engines (E1, E2, and E3) are used to execute the overall workflow. For example, engine (E1) is responsible for executing sub workflows (1) and (2), engine (E2) is responsible for executing sub workflows (3) and (4), and engine (E3) is responsible for executing sub workflows (5) and (6) respectively.

Table 4.1 Placement of sub workflows shown in Figure 4.9 onto candidate engines.

Sub workflow	Figure	Service	Inputs	Output	Placement
1	4.9(a)	S1	a	b	E1
2	4.9(b)	S2	b	c	E1
3	4.9(c)	S3	c	d	E2
4	4.9(d)	S4	d	e	E2
5	4.9(e)	S5	d	f	E3
6	4.9(f)	S6	f and e	x	E3

Composition of Sub Workflows

Based on the outcome of placement analysis, the sub workflows may be combined to create composite workflows. Table 4.2 shows a summary of arbitrary composite workflows.

Table 4.2 Composition of sub workflows shown in Figure 4.9.

Sub workflow	Figure	Services	Inputs	Intermediate Data	Output	Placement
7	4.10	S1 and S2	a	-	c	E1
8	4.11	S3 and S4	c	d	e	E2
9	4.12	S5 and S6	d and e	-	x	E3

Figures 4.10, 4.11 and 4.12 show the structure of the composite workflows, each of which contains a number of services and accepts a certain amount of inputs and produces a

single output. Some of the outputs of the composite workflows may be used as inputs for others, and it is possible for a composite workflow to produce more than one output. The remainder of this section analyses the structure of these composite workflows:

1. Figure 4.10 shows the structure of a composite workflow that consists of two services (s_1 and s_2), where an input (a) is used to invoke service (s_1). The result of the service invocation is passed directly as an input to service (s_2), which in turn produces (c).

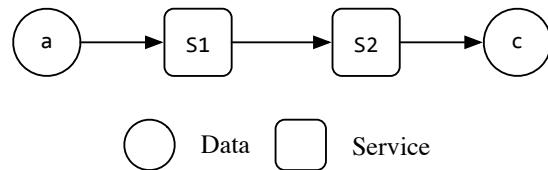


Fig. 4.10 Composite workflow that corresponds to partition (P1) in Figure 4.8.

2. Figure 4.11 shows the structure of a composite workflow that consists of two services (s_3 and s_4). This composite workflow requires (c) as an initial input to invoke service (s_3). The result of invoking service (s_3) is indicated explicitly in the workflow structure as an intermediate output (d). This intermediate output is used as an input to invoke service (s_4), which in turn produces the final output (e).

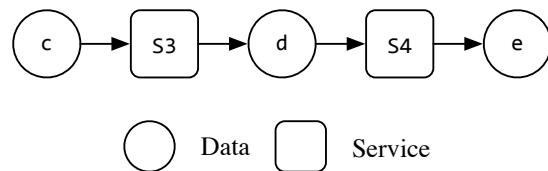


Fig. 4.11 Composite workflow that corresponds to partition (P2) in Figure 4.8.

3. Figure 4.12 shows the structure of a composite workflow that takes the initial inputs (d and e), which are the results of the workflow provided in Figure 4.11. The input (d) is used to invoke service ($S5$), which produces a result that is directly passed as an input parameter to service ($S6$). Similarly, the input (e) is used as a second parameter to invoke service ($S6$). Hence, service ($S6$) cannot be invoked unless the result of invoking service ($S5$) and the initial input (e) both become available. Service ($S6$) produces the final output (x).

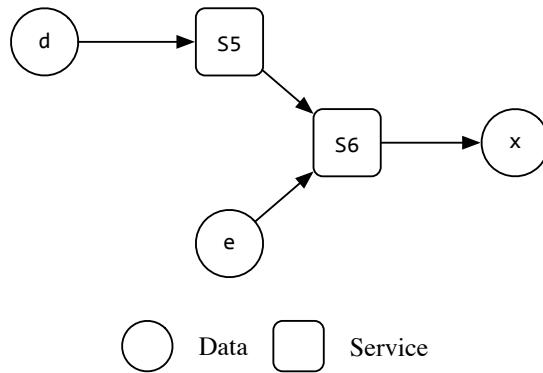


Fig. 4.12 Composite workflow that corresponds to partition (P3) in Figure 4.8.

Encoding and Computer-generated Constructs

This section presents the specification of the composite workflows shown earlier in Figures 4.10, 4.11 and 4.12. These specifications are all encoded automatically following the partitioning process using the *Orchestra* language. *Orchestra*'s language grammar, which is discussed in Section 3.2.3, provides production rules that are used solely by the language compiler to parse computer-generated sub workflow specifications. These computer-generated abstractions are used to indicate the following information:

- **Unique identification information:** Upon partitioning a workflow, each workflow partition is given a globally unique identifier to distinguish it from other distributed

workflows that have identical names. Listing 4.13 shows an example of a unique identifier that may be used in the specification of a workflow partition. This unique identification information is defined using the `'uid'` keyword, and consists of two parts separated by a `'.'` symbol: a numerical identifier, and the workflow partition number.

```
uid 618e65607dc47807a51a4aa3211c3298fd8.1
```

Listing 4.13 Computer-generated identification information of a workflow partition.

- **Data forwarding:** Each workflow partition may contain information for its execution engine about the endpoint locations of the remote engines, and instructions that describe the manner in which to communicate with these engines. Listing 4.14 provides an example of a computer-generated statement that may be defined in the specification of a workflow partition to indicate a particular data segment to be transferred to a remote execution engine that requires it. In this example, the keyword `'forward'` is used to indicate that `'c'` must be transferred to engine `'e2'`.

```
forward c to e2
```

Listing 4.14 Computer-generated data forwarding information.

Based on the partitioning example presented earlier, Listing 4.15 shows a computer generated specification of the composite workflow provided in Figure 4.10. This specification is executed by an engine that is deployed closer to services (S1) and (S2) as shown in Figure 4.8. It shows a universally unique identifier that is specified using the `'uid'` keyword in line 2. This identifier is generated to distinguish the workflow from others with the same name. The `'engine'` keyword declares a remote engine identifier in Line 3. The identifiers relating to the services are all declared through Lines 4-9. The workflow interface is defined through Lines 10-13. The input `'a'` is used to invoke `'p1.Op1'`, whose output is passed to `'p2.Op2'`

that produces '`c`' through lines 14-16. Finally, the '`forward`' keyword is used to indicate that the workflow output needs to be forwarded to '`e2`' to perform further computation as shown in Line 17.

```

01 workflow example
02 uid 618e65607dc47807a51a4aa3211c3298fd8.1
03 engine e2 is http://ec2-54-83-2-120.compute-1.amazonaws.com/
  services/Engine?wsdl
04 description d1 is http://ward.host.cs.st-andrews.ac.uk/
  documents/service1.wsdl
05 description d2 is http://ward.host.cs.st-andrews.ac.uk/
  documents/service2.wsdl
06 service s1 is d1.Service1
07 service s2 is d2.Service2
08 port p1 is s1.Port1
09 port p2 is s2.Port2
10 input:
11   int a
12 output:
13   int c
14 a -> p1.Op1
15 p1.Op1 -> p2.Op2
16 p2.Op2 -> c
17 forward c to e2

```

Listing 4.15 Specification of the first composite workflow shown in Figure 4.10.

Listing 4.16 shows the specification of the second workflow which is shown in Figure 4.11. In this specification, the workflow name and its unique identification information are defined through Lines 1-2. Engine '`e3`' is defined in Line 3 using the '`engine`' keyword. Service description document reference identifiers are all defined through Lines 4-7, followed by the definition of the services used and their ports through Lines 6-9. The workflow interface consists of a single input '`c`', and two outputs: '`d`' and '`e`'. The input '`c`' is used to invoke '`p3.Op3`', which produces the intermediate output '`d`' through Lines 14-15. This intermediate output is then used to invoke '`p4.Op4`' in Line 16, which in turn produces the final output '`e`' in Line 17. Both outputs '`d`' and '`e`' are transferred to engine '`e3`' as indicated through Lines 18-19.

```

01 workflow example
02 uid 618e65607dc47807a51a4aa3211c3298fd8.2
03 engine e3 is http://ec2-54-80-6-125.compute-1.amazonaws.com/
   services/Engine?wsdl
04 description d3 is http://ward.host.cs.st-andrews.ac.uk/
   documents/service3.wsdl
05 description d4 is http://ward.host.cs.st-andrews.ac.uk/
   documents/service4.wsdl
06 service s3 is d3.Service3
07 service s4 is d4.Service4
08 port p3 is s3.Port3
09 port p4 is s4.Port4
10 input:
11   int c
12 output:
13   int d, e
14 c -> p3.Op3
15 p3.Op3 -> d
16 d -> p4.Op4
17 p4.Op4 -> e
18 forward d to e3
19 forward e to e3

```

Listing 4.16 Specification of the second composite workflow shown in Figure 4.11.

Listing 4.17 shows the specification of the workflow in Figure 4.12. Similar to the previous specifications, the workflow name and unique identification information is defined first through Lines 1-2. The engine '`e1`' is defined in Line 3. Service description document reference identifiers are defined through Lines 4-5, whereas the services used and their ports are defined through Lines 6-9. The workflow interface consists of two initial inputs: '`d`' and '`e`' of the same type, and a single output '`x`'. In this specification, the input '`d`' is used to invoke '`p5.Op5`' in Line 14, which produces a result that is passed directly as an input parameter '`par2`' to operation '`p6.Op6`' in Line 15. Similarly, the input '`e`' is used as an input parameter '`par1`' to invoke '`p6.Op6`' in line 16. Hence, this invocation can only take place when both the result of invoking '`p5.Op5`' and '`e`' become available. The operation '`p6.Op6`' produces the final output '`x`' in Line 17. Finally, the workflow output '`x`' is forwarded to engine '`e1`' in Line 18, which acts as a data sink for the workflow outputs.

```
01 workflow example
02 uid 618e65607dc47807a51a4aa3211c3298fd8.3
03 engine e1 is http://ec2-54-80-3-122.compute-1.amazonaws.com/
   services/Engine?wsdl
04 description d5 is http://ward.host.cs.st-andrews.ac.uk/
   documents/service5.wsdl
05 description d6 is http://ward.host.cs.st-andrews.ac.uk/
   documents/service6.wsdl
06 service s5 is d5.Service5
07 service s6 is d6.Service6
08 port p5 is s5.Port5
09 port p6 is s6.Port6
10 input:
11     int d, e
12 output:
13     int x
14 d -> p5.Op5
15 p5.Op5 -> p6.Op6.par2
16 e -> p6.Op6.par1
17 p6.Op6 -> x
18 forward x to e1
```

Listing 4.17 Specification of the third composite workflow shown in Figure 4.12.

These specifications are all executed closer to the services using different execution engines as shown in Figure 4.8.

4.5 Conclusion

This chapter introduced an overview of a decentralised orchestration system and its architectural design which was presented and discussed by Jaradat et al. in [268], [267], and [269]. This system permits a service-oriented workflow specification based on the *Orchestra* language to be decomposed into smaller parts that represent sub workflows (e.g. partitions), and determines the most appropriate network locations at which these sub workflows may be executed. This is achieved by gathering and analysing QoS metrics (e.g. network latency and bandwidth) from the execution environment relating to the services participating in the workflow and available compute servers (e.g. engines). The presented architecture of the

system provides the ability to trigger the execution of the sub workflows and monitor the overall progress of the workflow. It is essential not to confuse this system's architectural design with the design of hierarchical task planning approaches that attempt to generate execution plans for workflow tasks based on requirements and temporal constraints specified by the user. Typically, such requirements or constraints in hierarchical planning approaches are classified as high-level goals (e.g. precedence of tasks, or deadlines) that relate to the context of the workflow, and need to be specified in an explicit manner. Common workflow management systems such as *Pegasus* [22], *Taverna* [128], and *Kepler* [111] permit the user to define the dependencies of the workflow tasks manually. For example, a domain expert user (e.g. bioinformatician) that uses *Taverna* for example is responsible for constructing the workflow based on their goals. *Pegasus* has a distinctive advantage in utilising planning to generate an executable workflow plan based on an abstract specification of the workflow tasks and available resources to support dynamic scheduling. The proposed system in this thesis does not depend on high-level goals specified by the user to execute a workflow application. However, the novelty of the proposed system is its ability to move the computation (e.g. workflow logic) towards the services providing the data. This is particularly achieved by gathering knowledge from the execution environment that may be used to determine the most appropriate engines with short network distance (e.g. latency) from the services to execute the workflow, and decomposing the workflow into smaller partitions that may be transmitted to these engines for execution. Furthermore, it supports automatic detection of the dependencies of the workflow tasks during the compilation of the workflow specification, and the workflow execution does not require a scheduling mechanism as it is explained in Section 4.3.1, and Section 4.3.6 respectively.

Chapter 5

Reference Implementation

5.1 Introduction

This thesis has presented the design of a decentralised service-oriented orchestration architecture that relies on identical distributed execution engines in Chapter 3. These engines are capable of compiling, partitioning, and executing a workflow specification. This section discusses the implementation of these engines. Each engine is implemented as a web service application that provides a standard interface that defines a set of functionality that can be invoked. This permits the engine to be deployed on any physical or virtual server, and enables the engine to communicate with other engines in a uniform manner. This chapter provides a reference implementation of the decentralised service-oriented architecture presented in this thesis. The overall implementation corresponds to the architectural design discussed in Chapter 3, and it is entirely written in the Java programming language which supports portability and allows workflows to be executed on platform-independent machines. This chapter begins with a concise description of the functionality offered by the engine interface in Section 5.2. Section 5.3 describe the execution states of the engine and present a finite state machine that models these states. Section 5.4 discusses the implementation of the internal modules of the engine. Section 5.5 finally concludes this chapter.

5.2 Engine Interface

The engine interface describes a set of functionality (e.g. operations) that can be invoked using standard web service technology. This enables any web service client application (e.g. web browser) to access the engine functionality, and supports interoperability between distributed engines that interact with each other based on uniform message exchanges. Listing 5.1 shows the source code of the engine interface written in the Java programming language.

```
@WebService
public interface IEngine {

    @WebMethod
    public String compile(String specification);

    @WebMethod
    public boolean execute(String workflow, String uid);

    @WebMethod
    public boolean lookup(String workflow, String uid);

    @WebMethod
    public boolean delete(String workflow, String uid);

    @WebMethod
    public boolean insert(String workflow, String uid,
        String identifier, String content);

    @WebMethod
    public String retrieve(String workflow, String uid,
        String identifier);

    @WebMethod
    public String getState(String workflow, String uid);

    @WebMethod
    public float getLatency(String url);

    @WebMethod
    public float getBandwidth(String url);

}
```

Listing 5.1 Source code of the engine web service interface.

Based on Listing 5.1, the engine's interface '`IEngine`' provides a number of operations for compiling a workflow, executing it, manipulating data segments relating to the workflow execution (e.g. inserting, finding, retrieving, and deleting particular workflow inputs or outputs), inquiring about the engine state, and inquiring about Quality of Service (QoS) metrics. These operations include '`compile`', '`execute`', '`lookup`', '`delete`', '`insert`', '`retrieve`', '`getState`', '`getLatency`', and '`getBandwidth`'. The remainder of this section describes these operations, their input parameters and outputs as follows:

- **The '`compile`' operation:** This operation instructs the engine to compile and analyse a workflow specification given as a string. It returns a serialised representation of the compilation log as a string, which is encoded in the *Java Script Object Notation* (JSON) [270] format which contains information about the compilation process. Such information indicates if the workflow specification has compiled successfully, and reports any syntax errors (e.g. position of the error in the workflow specification, and information about that error and how it can be corrected), in addition to warnings that indicate if a particular service or engine is unavailable (e.g. cannot be reached). The compilation log may provide information that may be generated by the engine such as a unique identifier that may be used to identify the workflow, or access its related data. This compilation log is particularly used for testing purposes and to inform the user about the result of the compilation process, whom may choose to inspect the compilation log manually if necessary.
- **The '`execute`' operation:** The execute operation is responsible for executing a workflow specification that has been compiled by the engine. It takes the workflow name and its unique identifier which was generated by the engine as input parameters. These parameters are used by the engine to identify the workflow that needs to be executed. This operation returns a boolean value which indicates if the workflow execution has started.

- **The `'lookup'` operation:** This operation is used to determine if a particular workflow is managed by the engine. It takes the workflow name and its unique identifier as input parameters and returns a boolean value.
- **The `'delete'` operation:** This operation is used to delete a workflow that is managed by the engine and its associated data. It returns a boolean value that indicates if the workflow has been deleted successfully.
- **The `'insert'` operation:** This operation is used to supply the engine with input data of some type that may be used in the execution of a particular workflow managed by the engine. It returns a boolean value that indicates if the engine has received the input data successfully.
- **The `'retrieve'` operation:** This operation is used to retrieve an output data of some type that is related to the execution of a particular workflow managed by the engine. It returns a serialised form of the data based on JSON.
- **The `'getState'` operation:** This operation is used to request information that relate to the execution of a particular workflow from the engine. It accepts the workflow name and its unique identifier as input parameters, and returns a string representation of the current workflow execution state which is encoded in JSON.
- **The `'getLatency'` operation:** This operation instructs the engine to measure the network latency between itself and a remote endpoint, and returns a float value of the network latency.
- **The `'getBandwidth'` operation:** This operation instructs the engine to measure the network bandwidth between itself and a remote endpoint, and returns a float value of the network bandwidth.

Based on the description of the operations provided by the engine, the engine uses the JSON format to encode information related to the workflow compilation, execution state, and associated data. The author of this work decided to use the JSON format to encode such information for implementation and testing purposes. The *Gson*¹ programming library (version 2.3.1) was used for serialising logging information stored in Java objects to JSON string representations. For example, using this format improves the readability of the logs generated by the engine especially that the developer may choose to inspect these logs manually if necessary. The implementation can be modified to support the encoding of logging information in a different format (e.g. SOAP), but the choice of this format does not affect the manner in which the engine operates and communicates with other entities (e.g. remote engines and services).

5.3 Engine States

There are a number of distinct states that an engine can have before, during, and after the execution of a workflow. These states are used for monitoring the overall progress of a workflow execution that involves a group of engines. Figure 5.2 provides a finite-state machine diagram that shows a set of engine states. These states help the user (e.g. scientist) in monitoring the overall execution progress of a workflow amongst a group of distributed engines. Typically, the user runs an engine on his/her local machine which is responsible for analysing a workflow specification, partitioning it, and deploying the specifications of its partitions onto a set of remote engines for execution. Each execution engine enters a compile state and begins the compilation of the workflow partition specification that it receives. Based on the result of the compilation process, the engine may enter a *ready* or an error state if it fails to compile the specification of the workflow partition. The engine enters a *ready* state if the compilation process was successful, and it will wait for a request

¹See the *Gson* project at <https://code.google.com/p/google-gson/>

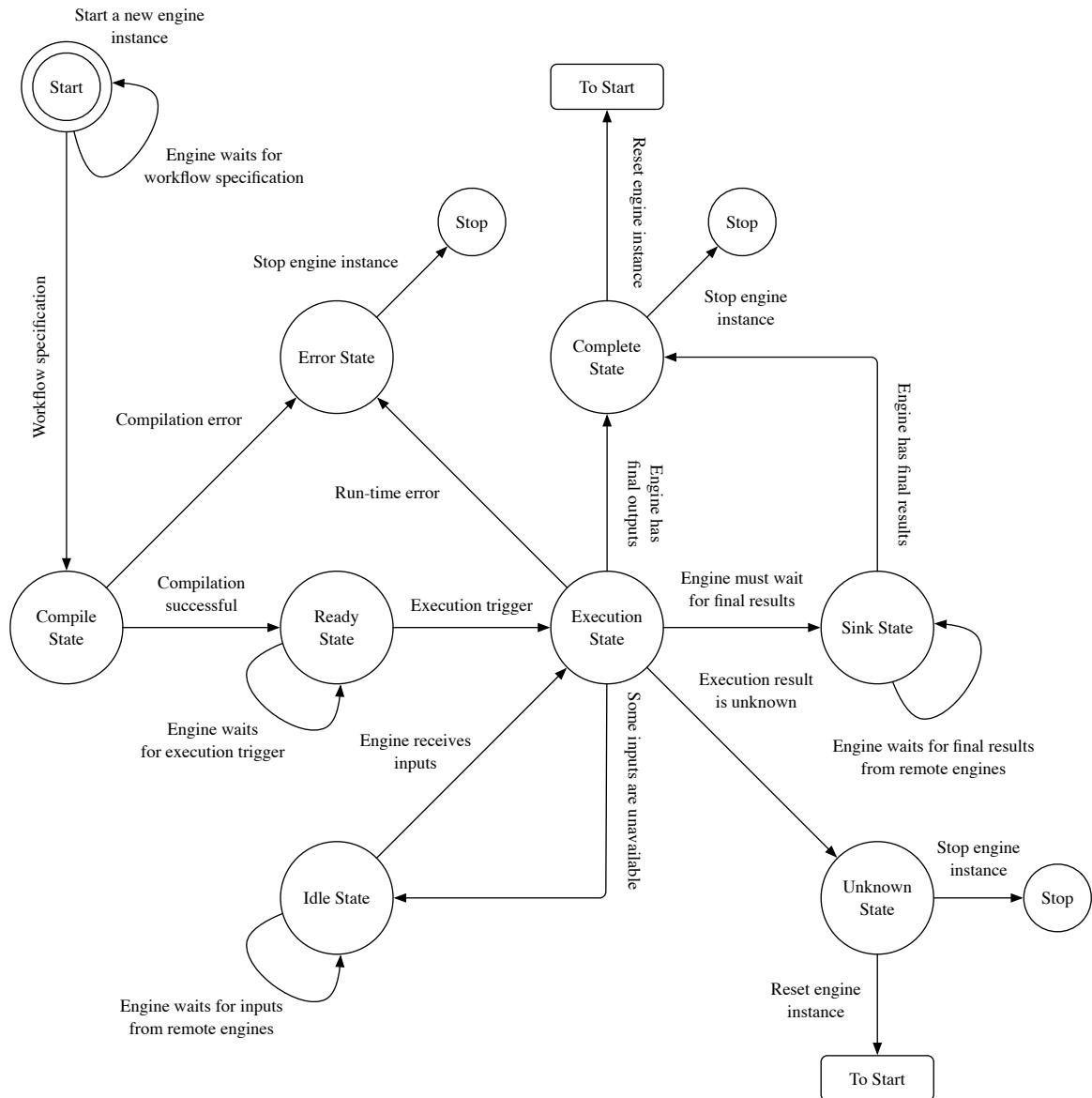


Fig. 5.2 Finite-state machine diagram of the engine states and their transitions.

message to trigger the execution of the workflow. This message typically originates from the user's local machine, and it is transmitted once only to all remote execution engines. Once a trigger message is received by an execution engine, it enters the *execution* state and begins executing the workflow partition. However, if the initial inputs that are required to execute the workflow are unavailable then the engine changes its state from the *execution* state to the *idle* state. This means that the engine will remain idle until it receives these inputs from other

sources (e.g. remote engines). The engine state may also enter the *idle* state suddenly during execution. This indicates that the execution of the workflow has not completed because the engine does not have some input data that is required for the workflow execution to progress. Hence, the engine will remain in an *idle* state until it receives intermediate inputs from other sources to resume the workflow execution. The engine may enter the *sink* state if it is required to receive the final results of the workflow execution, or the *complete* state if it has completed executing a workflow partition successfully. The remainder of this section describes in detail these states which include:

1. Ready state.
2. Compile state.
3. Execution state.
4. Idle state.
5. Complete state.
6. Sink state.
7. Error state.
8. Unknown state.

Compile State

This state indicates that the engine is currently compiling a workflow specification that may have been provided by the user, or a specification of a workflow partition (e.g. sub workflow) that may have been provided by a remote engine. The result of the compilation process determines the transition of the engine from this state to the ready or error state.

Ready State

This state indicates that the engine has successfully compiled the specification of a workflow or a workflow partition, and that it is currently ready to execute it. The engine maintains the state of readiness for execution until it receives a message that triggers the execution, which may be transmitted by an engine that is responsible for the partitioning of the workflow.

Execution State

This state indicates that the engine is currently executing the workflow. During this state, the engine may perform several activities such as invoking services, composing them, and forwarding intermediate outputs to remote engines if necessary. Typically, the engine changes its state to the complete state when it completes the execution of the workflow. However, the engine can change its state suddenly during the execution based on precise circumstances which are described as follows:

- **Temporarily suspension of the execution:** The execution of a workflow may be temporarily suspended if the engine requires further input data to continue the execution. This forces the engine to change its state from execution to idle, and wait for input data to be received from other sources (e.g. engines).
- **Collection of final workflow results:** The engine may change its execution state to the sink state when there is nothing else to do except for collecting the final results of the workflow, which may be received from remote engines. Typically, the user indicates if a particular engine must act as a data sink for the workflow results.
- **Unexpected failure:** The engine changes its state from an execution state to an error state if it encounters unexpected problems such as the failure of remote services or engines that are involved in the workflow execution.

Idle State

This state indicates that the engine is currently not executing the workflow, but it is waiting for one or more inputs from remote engines. The engine can switch back to the execution state when it receives the all the input data needed to resume the workflow execution.

Complete State

This state indicates that the engine has successfully completed the workflow execution. The engine notifies the user that the workflow execution has completed once it enters this state by transmitting a notification message to the initial engine that deployed the workflow which typically runs on the user's local machine. This permits the user to obtain the workflow results from the engine, and control the state of the distributed engines such as resetting their state to wait for further workflow specifications to execute, or terminating them.

Sink State

This state indicates that an engine is currently acting as a data sink. Typically, an engine enters this state suddenly during the workflow execution when there is nothing else to do except to wait for the arrival of final results of the workflow, which may be gathered and transmitted by a group of remote engines. Once all the workflow results are collected, the engine state changes to the complete state.

Error State

This state indicates that the workflow execution has stopped due to an unexpected problem that happened such as the inability to invoke a remote service or engine due to a sudden failure. Typically, an engine notifies the initial engine that deployed the workflow which may be running on the user's local machine of the error. This permits the user to identify the problem and restart the workflow execution.

Unknown State

This state indicates that the result of the workflow execution is unknown. The engine notifies the initial engine that deployed the workflow which may be running on the user's local machine that the workflow execution has suddenly stopped.

5.4 Construction of Engine Modules

The engine modules have been discussed in Section 4.2.1 of this thesis. This section focuses on the implementation of these modules which include the compiler, partitioner and planner, monitor, analyser module, deployer, executor, and datastore modules.

5.4.1 Compiler Module

Programmers can write workflow applications that can be executed across the Internet using *Orchestra*. For such workflows to be of any use, they must first be compiled into a suitable form for execution on distributed machines. *Orchestra*'s compiler is a recursive descent compiler that is built from scratch based on the teachings of Davie and Morrison [271] using the Java programming language. *Orchestra*'s compiler is responsible for checking any syntactic or semantic errors in the workflow specification, and for transforming its specification (e.g. source code) into a data structure that represents a dataflow graph which may be decomposed into smaller executable fragments that can be mapped onto distributed machines for execution. Therefore, *Orchestra*'s compilation process consists of three steps which include lexical analysis, syntax analysis, and the generation of an executable data structure. These steps convert the source code of a particular workflow into an appropriate internal memory representation that the language compiler can understand. Such a representation can be manipulated and refined to generate the executable dataflow graph. For this to be achieved, the source code has to adhere to a set of grammatical rules that describe the

language statements including identifiers, and special symbols. These grammatical rules are presented in Section 3.2.3, and are used to govern the manner in which the constructs of the *Orchestra* language are used. For example, a language is considered syntactically correct if it has followed the grammatical rules of the language which are used to describe the tokens that constitute a particular word in the language for example, and how multiple words can be combined together to form a syntactic statement known as a *clause*. These rules enable the compiler to perform context sensitive analysis to derive some meaning in order to build the internal memory representation of the dataflow graph. Semantic correctness therefore is determined in the language if the combination of words have some sensible meaning. Lexical analysis is performed using the *lexer* module which converts the characters of the workflow specification into a set of basic symbols. Syntax analysis processes the symbols produced by the scanner using a *parser* module, which generates a syntax tree. Finally, this syntax tree is converted into an executable data structure that represents a dataflow graph.

Figure 5.3 presents an overview of the compiler implementation including its classes and associated interfaces. The remainder of this section discusses the following:

1. Lexical analysis.
2. Syntax analysis.
3. Generation of the dataflow graph and its structural elements.
4. Symbol table used for recording relevant information about names (e.g. identifiers) discovered during syntax analysis.
5. Type representations of the symbols parsed during compilation
6. Error diagnosis and reporting facility.

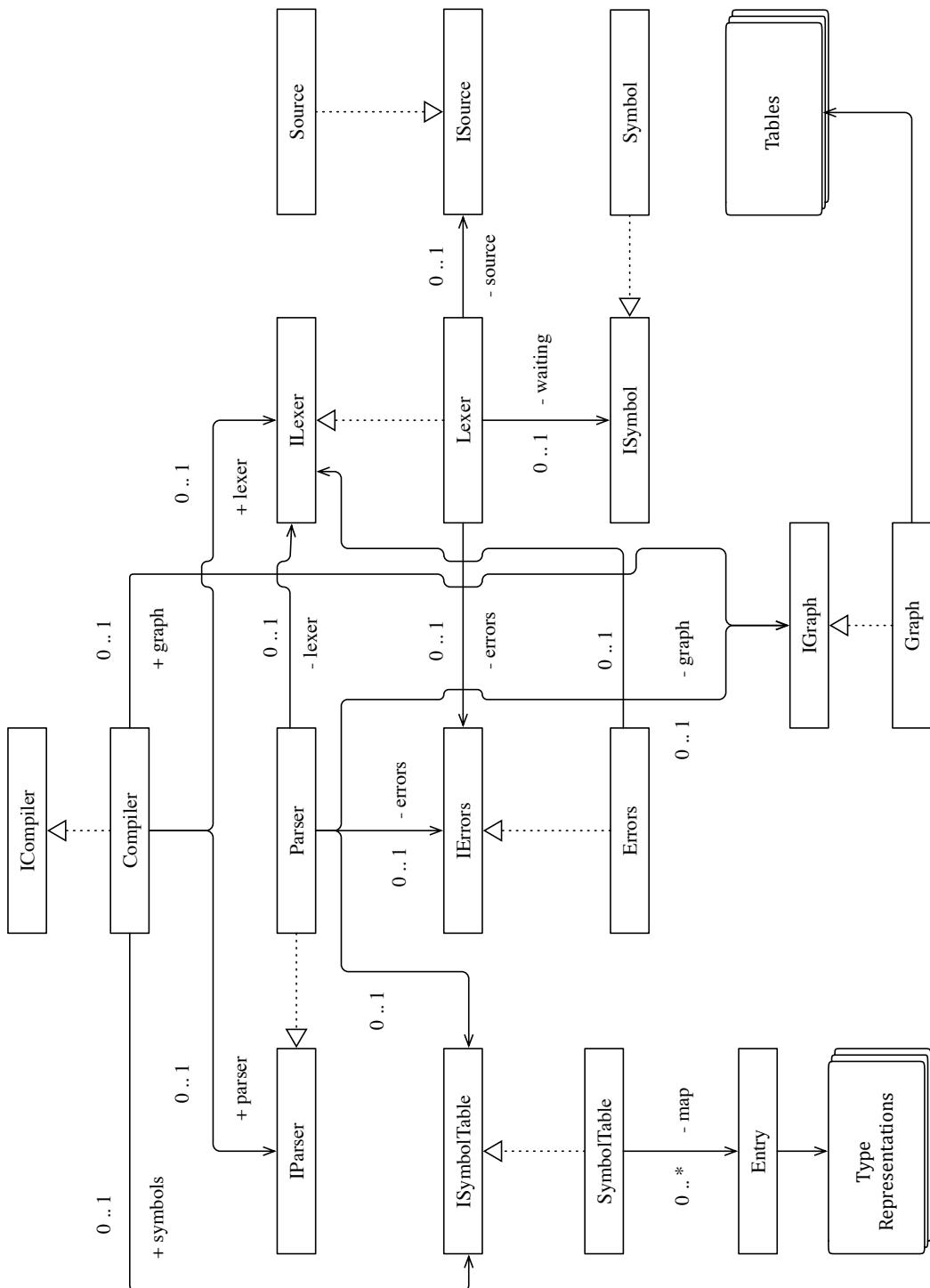


Fig. 5.3 Class diagram showing the main classes and packages used to implement the compiler.

Lexical Analysis

Lexical analysis is performed by the lexer which processes the source code of a workflow in the form of an input stream, and splits it into a stream of tokens which are sometimes known as *lexemes* or *terminals*. During lexical analysis, the tokens are validated to ensure that they are supported by the language grammatical rules. These tokens represent basic textual units of the language including identifiers of services, ports, variables, literals of different types, single and multiple character symbols in addition to punctuation characters. These tokens may represent identifiers of services, ports, variables, or scalar values for example. Lexical analysis involves *scanning* and *screening* which are described as follows:

- **Scanning:** Scanning processes the input stream and produces the tokens stream
- **Screening:** Screening discards some of the tokens in the stream. For example, screening removes spaces, tabs, newlines, comments, and other unnecessary characters such as punctuation symbols. It may convert some identifiers into reserved keywords, basic symbols, and literals.

Both scanning and screening are not independent of each other, as the screening abstractions may depend on those provided for scanning. Errors can be detected during lexical analysis when illegal single characters are found. These characters are passed to the parser to issue a syntax error message to the user.

Syntax Analysis

Syntax analysis is performed using the parser module, which takes a set of tokens produced by the lexer, and determines that they are used in a syntactically correct form. It is responsible for constructing a syntax tree that may be navigated to determine semantic correctness, and to produce an executable dataflow graph. This module's implementation provides a single private procedure for parsing each clause defined in the language grammar, which

will generate an equivalent data structure as part of the overall dataflow graph. This forms a set of recursive procedures to parse the language. For example, a procedure that compiles a particular clause in the language will be called from a procedure that compiles all clauses in the language, and it will then call a set of procedures to compile sub clauses. These procedures may use the scanner to recognise reserved keyword symbols in the language, and may include generative instructions to create each section of the dataflow graph related to the clause being parsed. This eliminates the intermediate encoding of the syntax tree between the parsing and generation steps, which makes the compilation process faster. Despite the fact that the compiler may require more memory storage to execute, this problem is not a significant one as the compiler is small in terms of size and can be written using any high-level language that may be supported by modern computers.

Generation of the Dataflow Graph

The step of generating the dataflow graph is a synthetic process rather than analytic, which relies on traversing the syntax tree constructed by the parser to detect its structural relationships. This permits the compiler to instantiate a set of objects, which may be linked together in a certain way to form a data structure that is equivalent to a dataflow graph. The dataflow graph structure can be manipulated in a manner such that its objects can be altered, removed, and linked with new objects as necessary. These objects represent instantiations of container classes for holding inputs or outputs used in the workflow, and may encapsulate executable logic for invoking services. The purpose of the dataflow graph is to execute the workflow by traversing it and executing the objects that hold computation logic for invoking the services. Service invocations can take place upon the availability of the input data that is required for their execution. This data structure consists of elements that include a hash table, and containers of inputs, outputs, and service invocations. These elements are described as follows:

- **Hash table:** This table is used to hold keys pointing to containers that hold the inputs and outputs of the workflow.
- **Input and output containers:** Each input and output is held in a *node* object that acts as a container for its type and value. Containers of this class hold the initial inputs needed to execute the workflow, and the intermediate and final outputs of the workflow. Each input and output container holds an identifier of the symbol parsed during compilation, its type representation and value. Input containers do not provide pointers to any other containers, whereas output containers point to invocation objects that act as containers for the computation logic needed to invoke the services.
- **Invocation containers:** Objects of this class act as containers that hold information about a particular service endpoint, a service operation that needs to be invoked. Each service invocation container consists of a number of input parameters, which are used to invoke the service operation. These input parameters may hold actual scalar values, hold references to inputs and outputs in the hash table, or point to other service invocation containers. Typically, an invocation container encapsulates the result of the service operation and its type representation. In the situation of a service composition, an invocation node may point to other invocation nodes whose results are required as input parameters for its execution.

Figure 5.4 shows the structure of the dataflow graph generated from compiling the specification of the workflow example provided in Section 4.4. This data structure permits its traversal beginning from the outputs to detect the data dependencies between its components for partitioning purposes, or execution purposes by performing service invocations that are required to produce the outputs. In order to execute the dataflow graph structure, a graph traversal mechanism is used to explore the graph and detect the invocation nodes that are ready to be executed. The traversal begins at the output nodes that have not been evaluated,

and from there the data structure is explored further to detect the invocation nodes that need to be executed to evaluate the output. Each invocation node can only be executed when all the input parameter values needed for its execution become available. Hence, an invocation node is checked when traversing the graph to check its readiness for execution by determining if all its input parameters have been satisfied. Typically, the graph is traversed whenever a new input value has been recorded which may be needed to execute a particular service invocation.

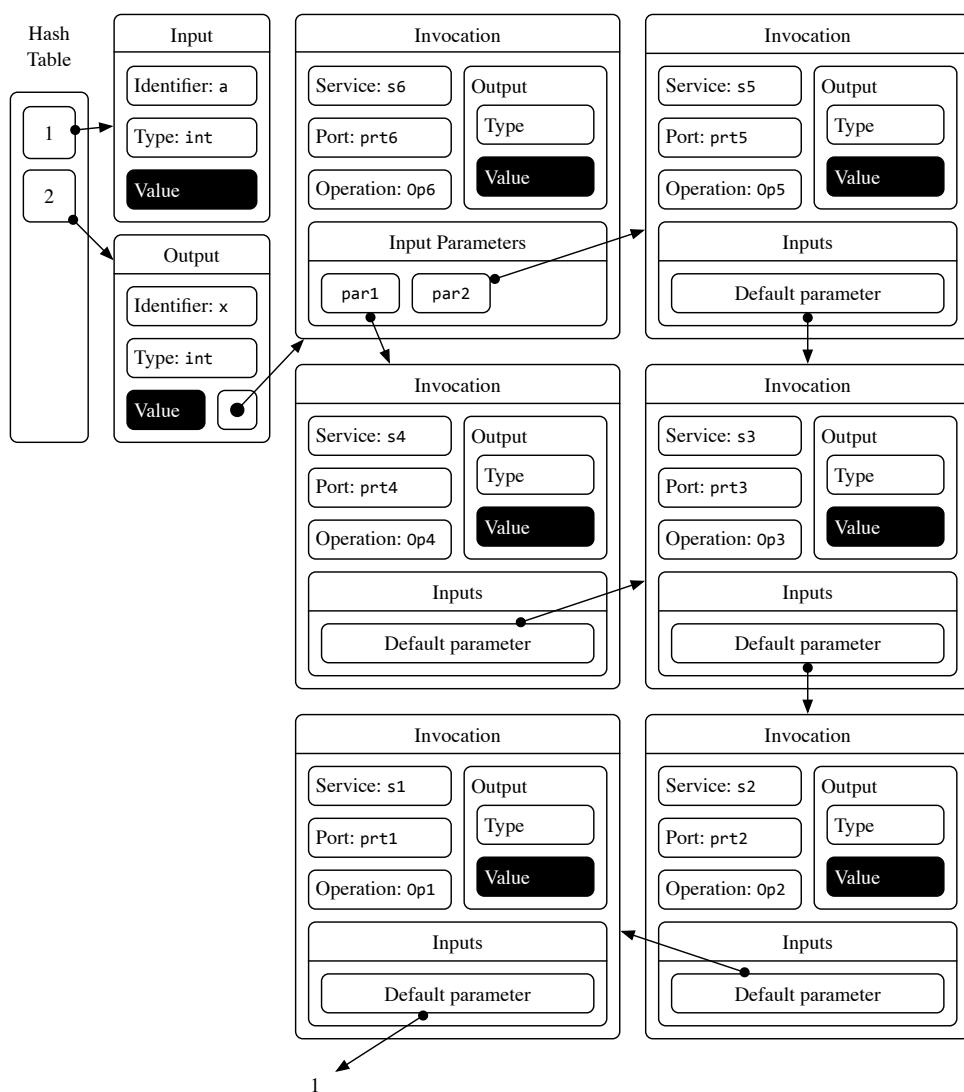


Fig. 5.4 Dataflow graph structure based on the workflow example in Figure 4.6.

Symbol Table

It is necessary to use a data structure for recording relevant information about names (e.g. identifiers) discovered during syntax analysis. Such information is typically recorded in a *symbol table*, which is used when an already declared name is subsequently parsed to determine if it has been used legally. Figure 5.5 shows the classes used to implement the symbol table and its interface which offers basic operations to declare and lookup identifiers. This table can therefore be used to detect errors in the workflow specification. For example, once an identifier name is parsed then a check must be made to ensure that it has not been already registered in the symbol table, after which the compiler must detect the appropriate type that can be associated with the identifier when it is declared. If the same identifier name is found again during parsing, then the parser consults the symbol table to ensure that it has been declared in order to retrieve its type information. This type information is used to determine if the identifier's type has been abused in the workflow specification.

The symbol table information may be extended and used during parsing to generate the

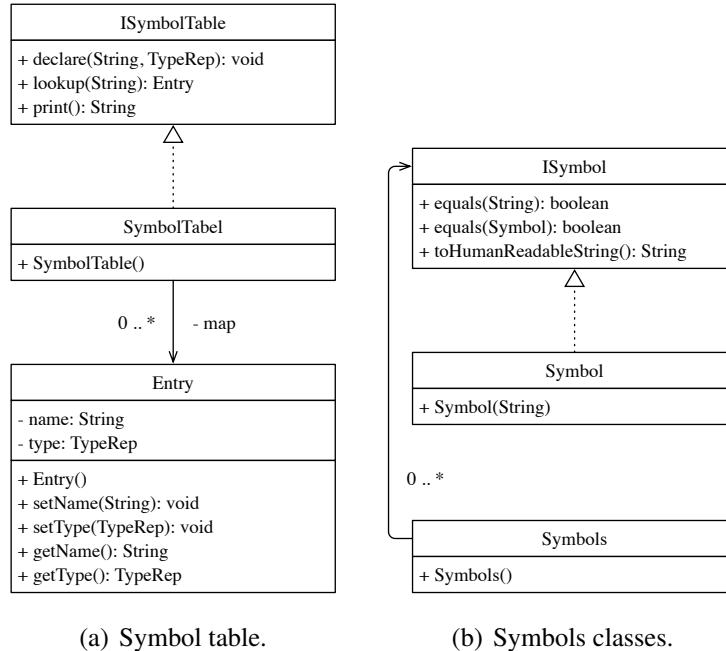


Fig. 5.5 Class diagrams showing the classes of the symbol table package.

dataflow graph. From a software engineering perspective, the symbol table implementation must be simple, easily accessed, and provide fast facilities to insert record entries, lookup records, and access them. For this reason, hashing techniques or binary trees are often used to implement symbol tables. *Orchestra*'s symbol table uses a hash table implementation to maintain records of identifiers and their types, and it may be used during various steps of compilation. It provides a single scope for all symbols discovered during parsing.

For an identifier and its type to be recorded in the symbol table, a procedure is built which takes the identifier's name and its type representation to register it. This procedure is known as `'declare'` and it is typically called while parsing. It returns a boolean value that indicates if the identifier has been declared successfully. For locating a particular symbol in the table, another procedure called `'lookup'` is built. This procedure takes the name of an identifier and attempts to locate it in the symbol table. If the identifier is found then an entry object is returned by the procedure, which represents a record containing the identifier's type representation. Otherwise, it returns a *null* value which indicates that the identifier has not been previously declared in the symbol table. The final procedure is called `'print'`, and it returns a string representation of the symbol table. The `'Entry'` class is used to instantiate an entry object that holds information about the symbol, such as its name and type representation. This entry is recorded into a private hash map in the symbol table once the symbol is declared. The implementation provides a `'Symbol'` abstraction that is used to store information about a symbol when discovered during lexical analysis. Figure 5.5(b) shows the classes of the symbols package and their relationships. The interface `'ISymbol'` provides a set of operations used to determine if a symbol is equivalent to another, and to display comprehensible information about the symbol. I decided to build a dedicated class called `'Symbols'` to store information about special symbols in the language such as keywords and operators. These symbols are defined statically and are given default and final character or string values within the `'Symbols'` class.

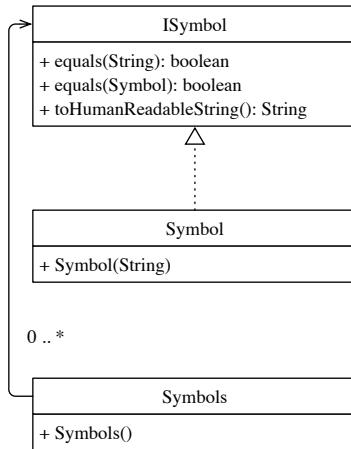


Fig. 5.6 Class diagram showing classes of the symbols package.

Type Representations

Every data type in *Orchestra* must have its own unique representation within the compiler. This is necessary because each parsing procedure typically returns a data type representation of the symbol that it has just parsed. Hence I have implemented a type representation abstraction called 'TypeRep' as shown in Figure 5.7 which provides a UML diagram of the classes implemented in the type representations package. This abstraction implements an interface that provides two procedures. The first procedure is called 'CompareTo' which is used by the parser to compare the type representation of a particular type instance with the type representation of another instance. If the two types do not match, then the parser will have to report an error that indicates the types compared and the reason for their incompatibility. The second procedure is called 'toHumanReadableString' which is used to return a string that displays the type representation in a simple language to the user. This is particularly useful for indicating record types in messages displayed to the user during and following the compilation process. Now that this abstraction is built, it can be used to construct meaningful data type representations. The structural complexity of these representations depends on the type system being supported, which is discussed in Section 3.5 of this thesis.

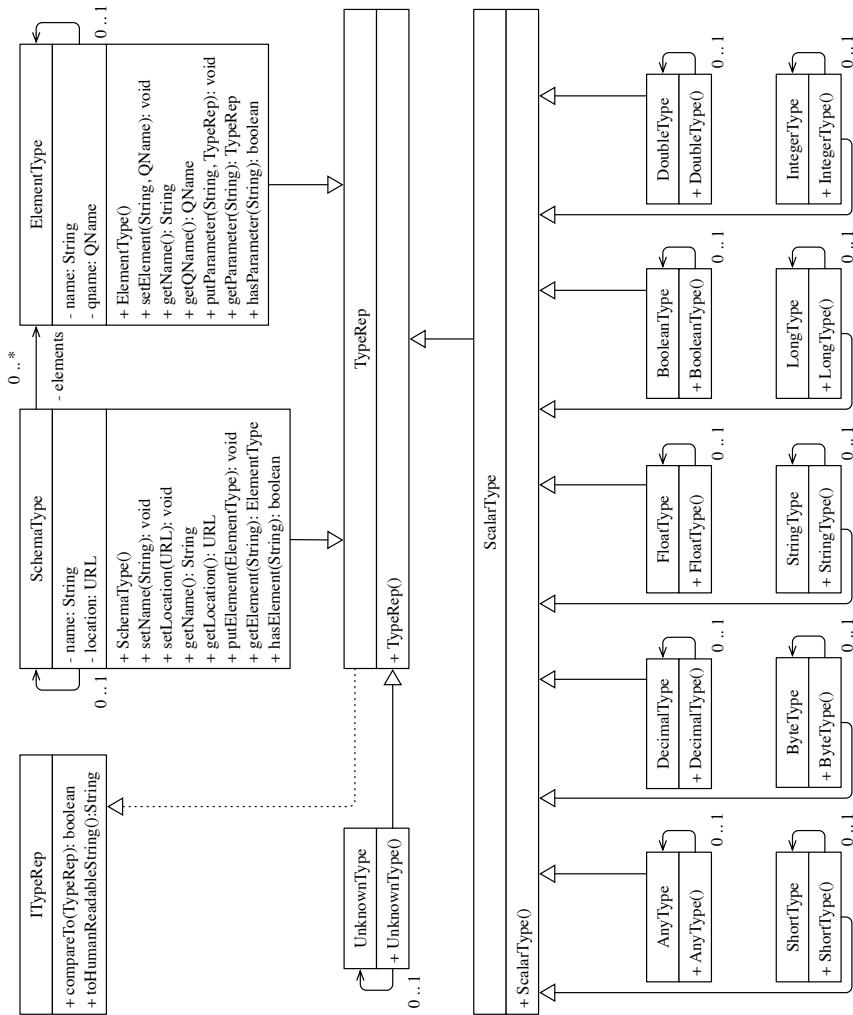


Fig. 5.7 Class diagram showing the classes of simple and record data type representations package.

Based on Figure 5.7 which provides an overview of the type representation classes, the remainder of this section will discuss these classes as follows:

- **Simple data type representation classes:** Simple data type representation classes all extend a class called `'ScalarType'` which is typically associated to a scalar during parsing. These classes include the `'DecimalType'`, `'FloatType'`, `'BooleanType'`, `'DoubleType'`, `'ShortType'`, `'ByteType'`, `'StringType'`, `'LongType'`, and the `'IntegerType'` classes, and the union type class `'AnyType'`. Each of these classes has a *final* string that represents the type name that is used in type matching checks. The compiler also provides the `'UnknownType'` class for an identifier's type that has not been recognised, but may be discovered later during parsing.
- **Record data type representation classes:** There `'SchemaType'`, and `'ElementType'` are both used for representing record data types. The `'SchemaType'` can be used to instantiate an object that may hold a data structure composed of `'ElementType'` objects. Each `'ElementType'` object may contain one or more parameters that have their own names and type representations. These types are usually derived from an external data type schema document identified by a URL. Hence, the parser retrieves the schema document and analyses it to construct these types.

During context sensitive analysis, it is necessary to ensure that the types of inputs used to invoke service operations are compatible with those of the input parameters supported by the operations. The types of the service operation outputs must also be checked for compatibility with the output types defined in the workflow specification. The parser relies on the *Easywsdl Toolbox*² (version 2.1) library to obtain and parse a web service description document. This permits the compiler to construct type representations matching the elements defined within a service description document based on WSDL 1.1 and WSDL 2.0. Figure 5.8 shows the classes implemented for service type representations.

²See the *Easywsdl Toolbox* at <http://easywsdl.ow2.org/>

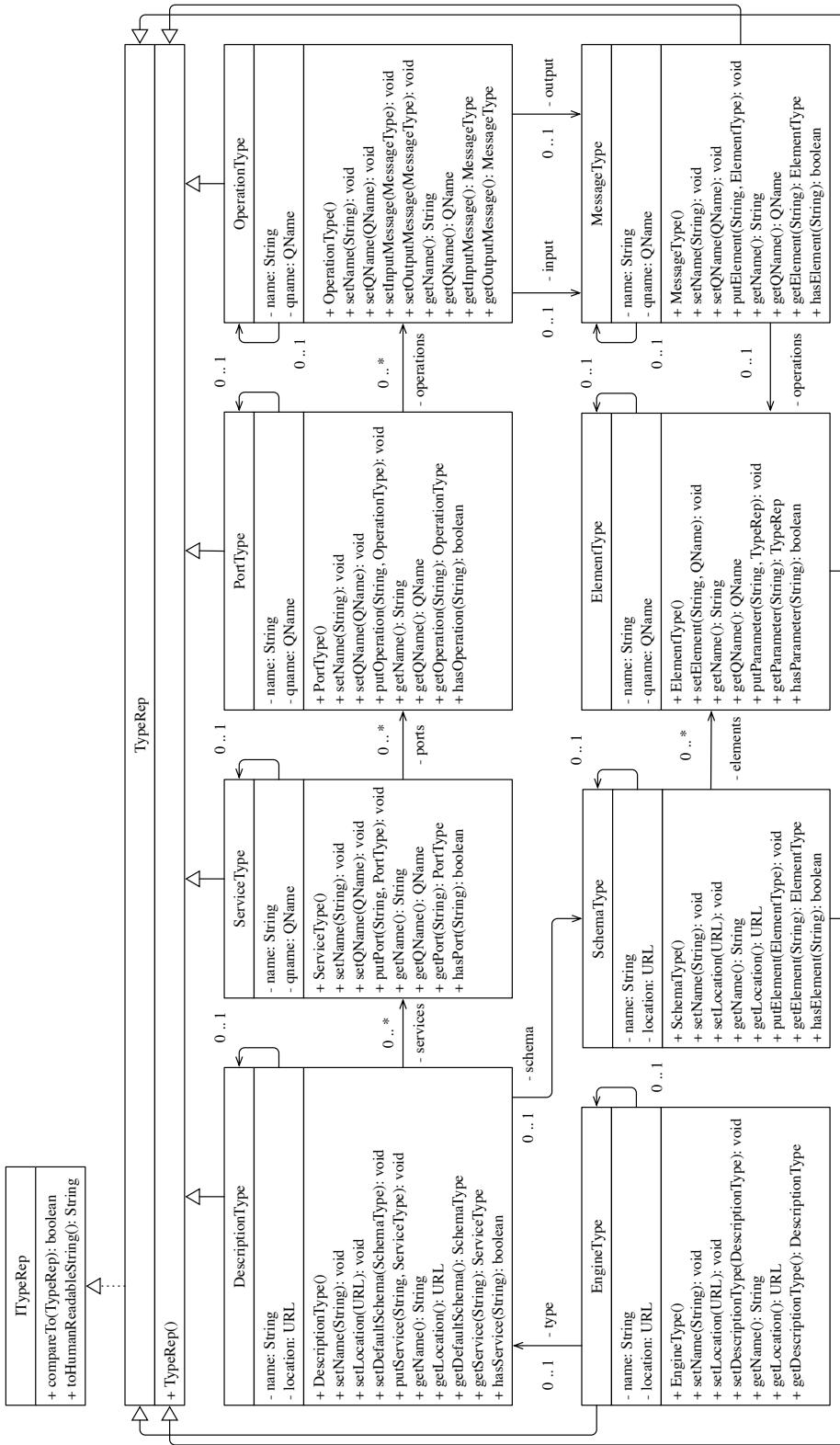


Fig. 5.8 Class diagram showing the classes of service-based type representations package.

There are several classes for representing service types including `'DescriptionType'`, `'ServiceType'`, `'PortType'`, `'OperationType'`, `'MessageType'`, and `'EngineType'`, which are discussed as follows:

- **The `'DescriptionType'` class:** Once an object of this class is initiated, it can hold the name of the description identifier found during parsing, and the location of the service description document. Since the service description document defines one or more services, then this object can maintain multiple objects of the type `'ServiceType'`.
- **The `'ServiceType'` class:** This class defines the name of the service identifier, its qualified name as described in the service description document, and can maintain multiple port objects of the type `'PortType'` when instantiated.
- **The `'PortType'` class:** This class is used to instantiate an object that holds the port type identifier, its qualified name and maintains a set of `'OperationType'` objects.
- **The `'OperationType'` class:** This class is used to instantiate an object that holds the identifier of an operation, its qualified name and contains an input and output `'MessageType'` objects.
- **The `'MessageType'` class:** This class is used to instantiate an object that holds an input or an output message type associated to the service operation. For example, an input `'MessageType'` object holds the input parameters required to invoke the operation. These parameters are each represented by an `'ElementType'` object. The output `'MessageType'` object holds the result parameter of the service operation and its type representation.
- **The `'EngineType'` class:** This class is used for holding information about the engines identified during parsing. Such information includes the name of the engine identifier, its location, and type.

It is important to note that the actual mapping of these types in the operating environment depends on the programming language used to build the compiler which is Java. Table 5.1 provides the mapping of types from the XML Schema standard to the language.

Table 5.1 Mapping of XML schema built-in data types to *Orchestra*.

XML Schema Type	<i>Orchestra</i> 's Type Representation
xsd:string	string
xsd:integer, xsd:int, and xsd:unsignedShort	int
xsd:long, and xsd:unsignedInt	long
xsd:short, and xsd:unsignedByte	short
xsd:decimal	decimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:base64Binary, and xsd:hexBinary	byte[]
xsd:anySimpleType	any

Error Diagnosis and Reporting

Programmers may compose workflow specifications that are syntactically incorrect, and the compiler must be responsible for detecting possible errors in the workflow and react to them accordingly. The compiler must also be able to detect and deal with unknown errors. The implementation uses a simple software engineering approach to deal with errors discovered during the compilation process. Erroneous workflow specifications cannot be executed until the errors are addressed by the programmer. Therefore this approach permits errors to be registered when detected, and reported to the programmer following the compilation process. These errors can be detected during lexical analysis when illegal characters are used, during syntax analysis when a particular clause does not conform to the grammatical rules of the language, or context sensitive analysis when the mismatching type representations are discovered for example. Hence, the compiler provides a dedicated registry for

recording errors during compilation. This registry allows a comprehensible error report to be issued to the programmer, which indicates the number of errors discovered in the workflow, the description of each error and its location in the workflow specification. This report may provide advice to the programmer on how to fix the mistakes found in the workflow specification. Figure 5.9 shows the classes used to implement the error facility. These classes consists of `'Errors'` and its interface `'IErrors'`. This class provides the operation `'setError'` to register an error found at a particular line in the workflow specification, its position in that line, and its description. It provides an operation called `'getCount'` which returns the number of errors registered, and an operation called `'getReport'` which returns a string that represents a summary of the errors discovered.

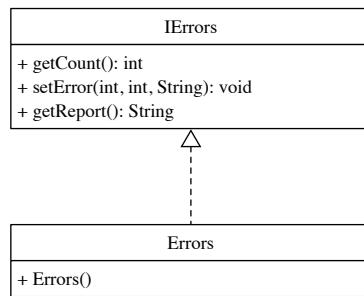


Fig. 5.9 Class diagram showing the classes of error registry package.

5.4.2 Partitioner Module

This section describes the implementation of the partitioner module and the mechanisms used to split the workflow into smaller partitions (e.g. sub workflows), and prepare these partitions for deployment. It provides a description of the following:

1. The general set of rules which are used to assist in partitioning a workflow.
2. The mechanisms required to partition a workflow.
3. The partitioner sub components that are used to implement these mechanisms.

General Partitioning Rules

The partitioner module is implemented based on a general set of rules, which are used to assist in partitioning the workflow into smaller parts (e.g. sub workflows) that can be executed in parallel. These rules are described as follows:

1. The partitioner must be able to detect all the workflow tasks (e.g. service invocations) that can be executed in parallel.
2. The partitioning process must permit the structure of the workflow to be changed as necessary to support the placement of the workflow tasks onto appropriate execution engines to improve the overall execution of the workflow performance.
3. The partitioner must be able to produce a set of workflow partitions (e.g. sub workflows), where each partition may consist of one or more tasks to be executed in some order without affecting the overall workflow results.
4. The partitions must be encoded in a format that is suitable for deployment onto the execution engines.

Partitioning Mechanisms

The implementation of the partitioner module relies on a set of mechanisms that are used to manipulate the structure of the workflow and prepare it for distribution across the network. Such mechanisms should permit the workflow partitioner to perform the following tasks:

1. Decomposing the data structure representation of the workflow which was generated by the compiler into smaller parts (e.g. invocation containers which have been discussed in Section 5.4.1) that can be executed in parallel using distributed engines.
2. Composing a set of invocation containers together to form new data structures that represent sub workflows.

3. Encoding the data structures representing the sub workflows (e.g. partitions) into a suitable format for distribution across the network (e.g. specifications of workflow partitions).

These mechanisms are implemented in a graph traversal mechanism that is used for exploring the data structure representation of the workflow, which was generated by the compiler and discussed in Section 5.4.1. The traversal mechanism begins at the output nodes (e.g. output container objects), and the data structure is explored further to detect the invocation nodes (e.g. invocation container objects) whose execution leads to the evaluation of the outputs, and the input nodes (e.g. input container objects) needed for the invocation nodes to be executed. During each visit, node information is gathered and analysed to build a dependency graph that can be used to decompose or restructure the workflow.

Components of the Partitioner Module

The partitioner module consists of a *composer* and *decomposer* components which are described as follows:

- **Decomposer:** This module is responsible for splitting the dataflow graph generated by the compiler into a set of independent service invocations, each of which may take a different set of input parameters and produces a single output. Such decomposition is achieved by traversing the dataflow graph to detect its dependencies between its intricate parts that can be parallelised. Figure 5.10 shows the output of the decomposer which represents a set of dataflow graphs. These dataflow graphs represent parts of the overall dataflow graph presented earlier in Figure 5.4, and correspond to the sub workflows discussed in the supplementary example provided in Figure 4.9. For example, the dataflow graph shown in Figure 5.10(a) corresponds to the sub workflow shown in Figure 4.9(a).

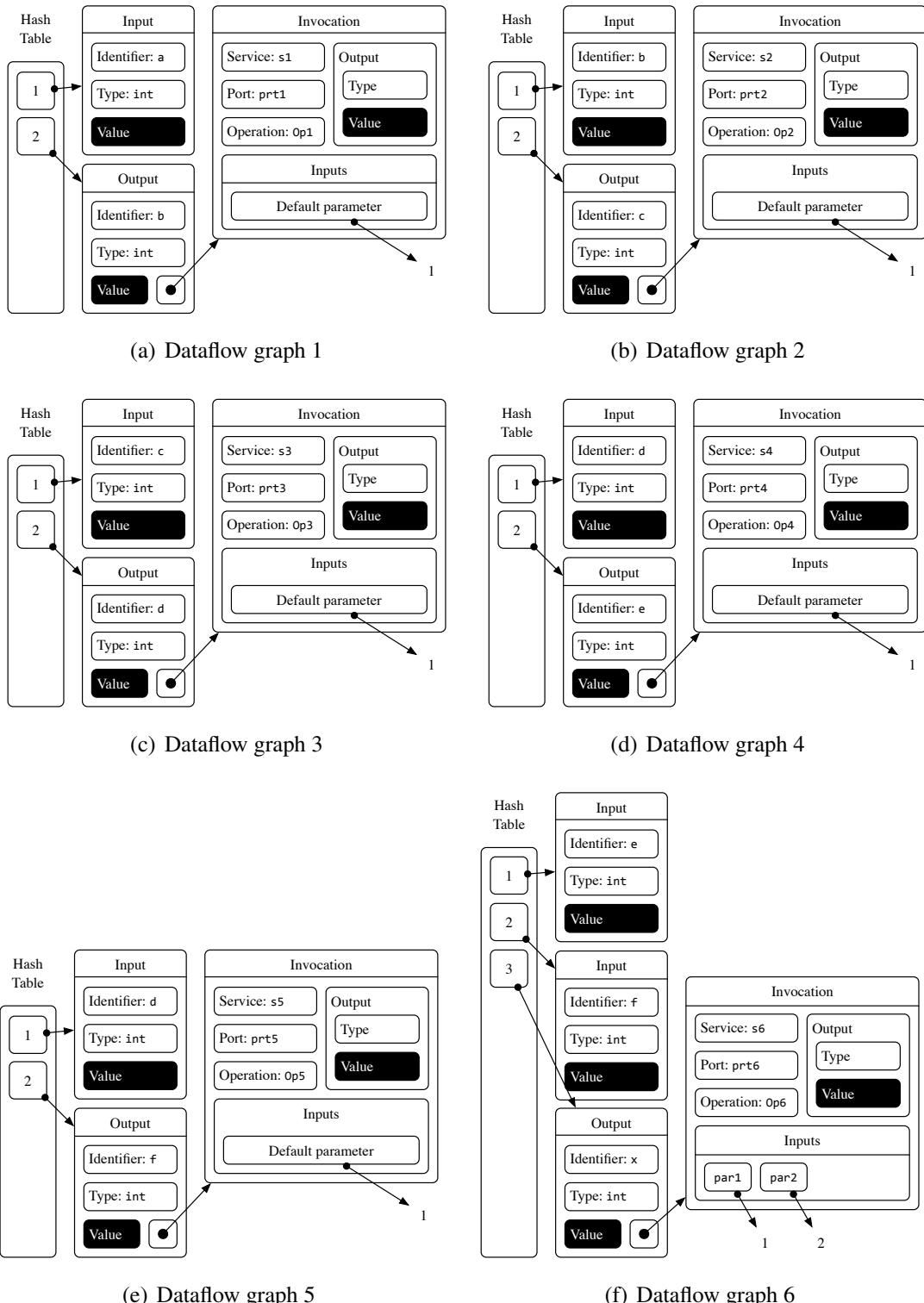


Fig. 5.10 Dataflow graphs based on the sub workflows in Figure 4.9.

- **Composer:** This module is responsible for combining independent service invocations together to form a set of new dataflow graphs, each of which may require a different set of initial inputs, and produces a set of final outputs. For example, this module is capable of composing the dataflow graphs presented in Figures 5.10(a) and 5.10(b) as shown in Figure 5.11. This dataflow graph corresponds to the composite workflow shown in Figure 4.10.

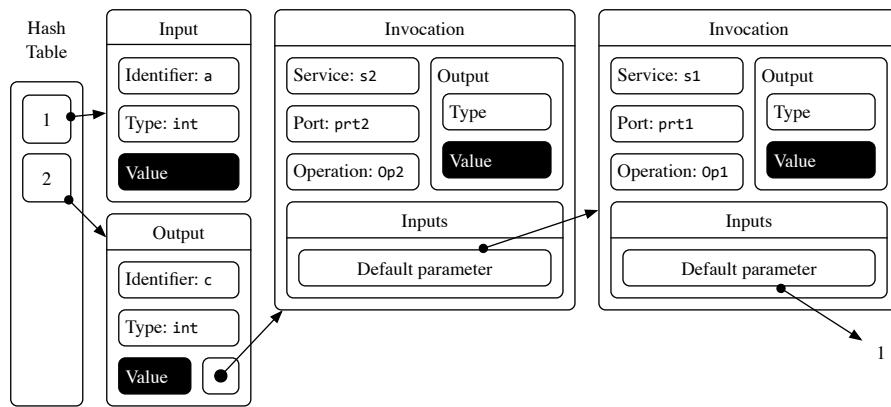


Fig. 5.11 Dataflow graph that represents the workflow in Figure 4.10.

Similarly, the composer is capable of combining the remaining executable dataflow graph structures as shown in Figures 5.12 and 5.13. These composite dataflow graph structures can be traversed to obtain information that can be used to generate specifications of the workflow partitions based on *Orchestra*. For example, the specifications of the workflow partitions presented in Listings 4.15, 4.16 and 4.17 correspond to the composite workflow data structures presented in Figures 5.11, 5.12, and 5.13 respectively. The orchestration of these partitions have been illustrated in Figure 4.8 earlier in the thesis.

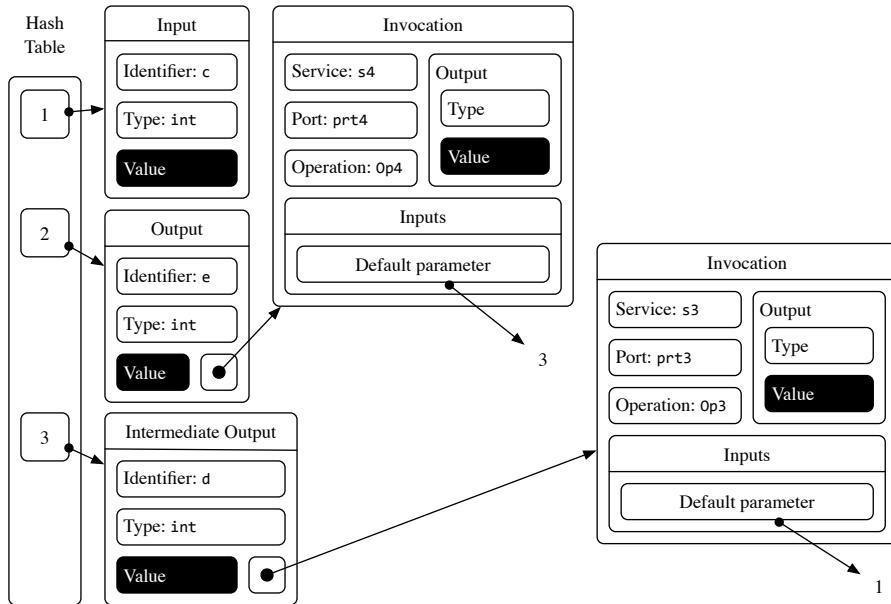


Fig. 5.12 Dataflow graph that represents the workflow shown in Figure 4.11.

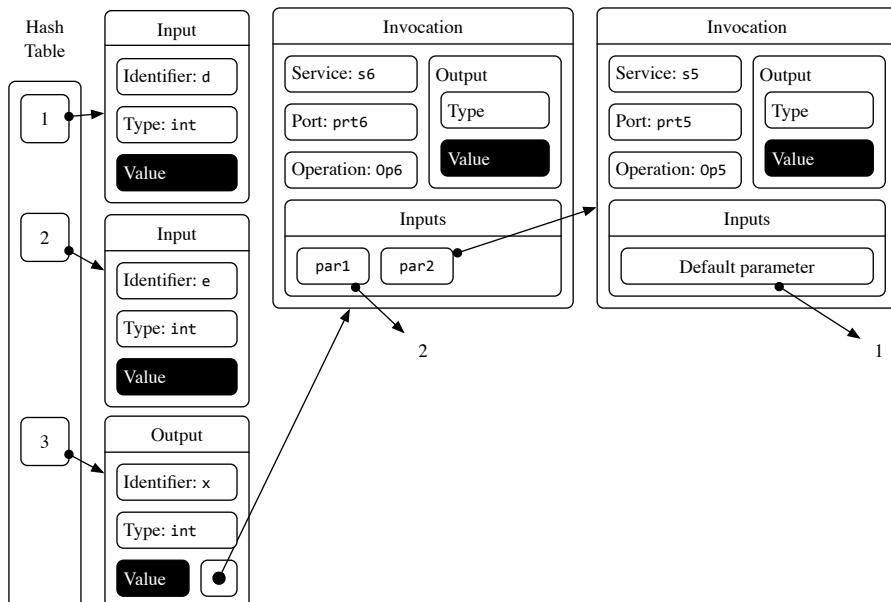


Fig. 5.13 Dataflow graph that represents the workflow shown in Figure 4.12.

5.4.3 Network Resource Monitoring Module

The network resource monitoring module is responsible for gathering information from the execution environment that can be used to measure specific QoS metrics. These metrics include the network latency and bandwidth between the engine and a particular service. The remainder of this section discusses the following:

1. The mechanism used to measure the network latency between the engine and a particular service.
2. The mechanism used to measure the network bandwidth between the engine and a particular service.
3. Creating a network topology of the engines and services participating in the workflow that may be used by the analyser module.
4. Supporting programming libraries.

Measuring the Network Latency

Network latency can be used as a factor to select an appropriate compute server (e.g. engine) to invoke a particular service as discussed in Section 4.3.3. Typically, it is measured by calculating the average *ping* times between a service and its client (e.g. an engine in the context of workflows) using the *Internet Control Message Protocol* (ICMP) [272]. However, there are a number of limitations in using this mechanism as service providers often prohibit clients from interacting with their services using ICMP. This makes it difficult to measure this metric. Hence, this thesis provides a simple and novel approach for detecting the latency between an engine and a particular service using the application layer capabilities. This is achieved using the following steps:

1. The engine probes the service by transmitting a single HTTP HEAD request message to the service to determine if the service is reachable (e.g. online).

2. If a service response message is received then the engine begins transmitting a series of HTTP HEAD request messages to the service and the number of these messages is determined at random ranging from 5 to 10 messages.
3. The round-trip times for sending each HTTP HEAD request to the service and receiving the response message form the service are recorded by the engine accordingly.
4. Finally, the average round-trip time is calculated to estimate the network latency between the engine that issued the HTTP HEAD requests and the service.

This mechanism relies on the HTTP HEAD to determine if the service is reachable, and measure the network latency because the size of the request and response HTTP header is usually small. For example, the maximum size of these messages is 8192 bytes (8 KB) for most services running on *Apache Tomcat*³ (version 7.0.65) servers⁴. Furthermore, this mechanism can be accurate to measure the network latency than *ping* as it takes into consideration the application layer latencies affected by the web service implementation. However, the performance of this approach must be compared with the performance of conventional mechanisms used to detect the network latency like *ping* to assess this claim.

Measuring the Network Bandwidth

Network bandwidth is used to assist in the selection of a candidate execution engines to invoke a particular service in a workflow. Despite the large number of existing tools that estimate the available bandwidth between a couple of network locations such as those reviewed Strauss et al. [273] and Parsad et al. [274], it is still considered more difficult to measure than network latency. This is because existing network bandwidth estimation tools require a program that measures this metric to be running at both network locations, which limits the applicability of using these tools. For example, a developer may not have

³See the *Apache Tomcat* project at <http://tomcat.apache.org>

⁴See the *HTTP Connector* reference at <https://tomcat.apache.org/tomcat-7.0-doc/config/http.html>

access privileges to access a particular machine at a network location to install the measurement program. Furthermore, existing network bandwidth estimation tools often use the *User Datagram Protocol* (UDP) [275] or ICMP probing packets. Service providers often do not allow such traffic to and from their services. Such traffic is also handled differently than HTTP traffic which relies on the *Transmission Control Protocol* (TCP) [276] protocol. Hence, a measurement mechanism that relies on HTTP would be more appropriate than existing tools to estimate the network bandwidth between a client (e.g. an engine in the context of a workflow) and a particular service. This section provides a novel mechanism for estimating the network bandwidth between an engine and a particular service which relies on HTTP. The implementation of this mechanism can be run on a single machine that hosts the service client (e.g. engine), and only requires the endpoint address of the service. This mechanism comprises of the following steps:

1. The service is probed by transmitting a single HTTP HEAD request message from the engine to check if the service is currently reachable.
2. The engine sends a single HTTP GET request message to the service.
3. Following the transmission of the HTTP GET request, the engine does not wait to receive a response and sends another request message to the service.
4. Once the engine receives the HTTP GET response messages from the service, it analyses the interval (e.g. gap) in milliseconds between the arrival of the response messages. This value is then used along with the response message size to estimate the bandwidth using the following simple equation:

$$B_{e-s} = S_{message}/T_{gap} \quad (5.1)$$

where B_{e-s} is the estimated bandwidth between the engine and the service, $S_{message}$

and T_{gap} are the size of the HTTP GET response message and the interval between the arrival of the response messages respectively.

5. Steps 2 through 4 above are all repeated n times to calculate a set of network bandwidth values, where n is an integer generated randomly up to 5.
6. Finally, the average of the all the resulting bandwidth values is calculated.

This mechanism may not be accurate for estimating the network bandwidth, but it gives an estimation of how long it takes to send data using a particular engine which can be particularly useful when comparing between the network bandwidth values obtained for different engines. The rationale behind this mechanism is adapted from a method that is briefly discussed by Spero [277] in a technical paper that focuses on some performance issues relating to HTTP, and demonstrates how to establish some network metrics (e.g the round-trip time and the lower bound on the available network bandwidth).

Creation of the Network Topology

Following the compilation process of a workflow specification using an initial engine (e.g. an engine such as (E1) in Figure 4.8), all information about the services participating in the workflow and available engines are recorded within a hash map data structure that is part of the network resource monitor module. Such recorded information includes the addresses of the service and engine endpoints. The network resource monitor employs a technique that transmits all information about the services to all available engines that may be used for executing the workflow. It instructs each engine to measure the network latency and bandwidth between itself and all the services participating in the workflow. This permits all the engines to gather these metrics concurrently. Each engine transmits these metrics once they are gathered to the initial engine that requested them, which in turn forms a logical network topology that represents a graph data structure in which the nodes represent engine

or service endpoints with edges that represent the network distance between them measured using QoS metrics (e.g. network latency and bandwidth). This network topology represents a *knowledge base* utilised by the analyser module to select candidate engines for invoking specific services.

Supporting Programming Libraries

It is essential to note that the network resource monitor module relies on the *Apache HttpComponents*⁵ programming library (version 4.4.1) for creating and maintaining component objects that provide HTTP protocol capabilities. This library provides an Core and Client components. The Core component is used to build custom client and server side HTTP services with a minimal footprint, whereas the Client component provides support for different functionality including the management of HTTP connections. The implementation of the network resource monitoring module specifically makes use of the Client component to create client objects that are able to send HTTP HEAD or GET request messages to services, and receive service response messages.

5.4.4 Placement Analysis Module

The analyser module is responsible for conducting placement analysis, and determining the most appropriate workflow engines to execute particular service invocations. It relies on the logical network topology constructed by the network resource monitor to obtain QoS metrics about the engines and the services which may be used in placement analysis. The analyser module reports back to the partitioner with its findings, which in turn may instruct its composer component to create composite workflows to be executed using specific candidate engines. The analyser module implements the algorithm that has been previously shown in Figure 4.4. It also relies on the *k-means* clustering algorithm to determine a group

⁵See the *Apache HttpComponents* at <https://hc.apache.org/>

of candidate engines that may be used to execute a particular workflow partition. There are numerous implementations of the *k-means* algorithm. Currently, the implementation of the analyser module relies on the JavaML⁶ (version 0.1.7) programming library that provides this algorithm. This programming library also provides a collection of machine learning algorithms and a common interface to use each algorithm, and it is commonly used for processing and manipulating datasets in scientific applications. The *k-means* algorithm has some known issues which are discussed by Schutt and O’Neil [278]. These issues are listed as follows:

1. Choosing k is considered more of an art than a science although there are bounds such that $1 \leq k \leq n$, where n is the number of data points to be clustered.
2. The algorithm can fail to produce a solution if the execution of the algorithm enters a loop when there is no unique solution.
3. Sometimes the algorithm may not produce a useful answer.

Due to these limitations the implementation of the *k-means* algorithm is configured manually in the implementation of the analyser module to produce a maximum of two clusters for several reasons. This is useful to avoid falling into an infinite loop especially when there are a few number of engines that need to be clustered, as the algorithm may keep going back and forth between two possible solutions. There is no actual need to produce more than two clusters because a single cluster will consist of engines with better QoS metrics than all the remaining engines grouped in other clusters. Future work stated in Section 7.6 include the evaluation of the clustering algorithm employed by the analyser and its improvement. It is also worth studying a recent version of this algorithm called *k-means++* which was developed by Arthur and Vassilvitskii [279]. This algorithm addresses some of the limitations of the standard *k-means* algorithm.

⁶See the JavaML project at <http://java-ml.sourceforge.net/>

5.4.5 Deployment Module

The deployment of the workflow partitions relies on information obtained from the workflow partitioning process. Such information consist of the workflow partitions themselves and the dataflow dependencies between them. For example, the initial input of a particular workflow partition can be the final output of another workflow partition. The deployer module creates a data structure that consists of task objects, where each task is responsible for distributing the specification of a workflow partition to a specific candidate engine that is selected to execute it. Each task can hold a state relevant to the execution of its workflow partition that corresponds to one of the engine states described in Section 5.1. This permits the deployment module to dispatch the specifications of each workflow partition to a designated engine, and trigger the execution of these workflow partitions when the remote engines successfully compile these specifications. Furthermore, this supports monitoring the overall workflow execution state when remote engines transmit notification messages about their execution states to the initial engine.

So far this chapter has discussed the implementation of the engine interface, its execution states, and some of its internal modules, including the partitioner, network resource monitor, and the placement analyser. The remainder of this section provides a high-level view that summarises the implementation of these modules. Figure 5.14 provides a diagram that shows the interface of the dataflow graph '`IGraph`' generated by the compiler can be passed using the partitioner class '`Partitioner`' to the decomposer by accessing its interface '`IDecomposer`'. The decomposition procedure '`decompose`' returns a list of objects of the type '`Partition`'. These objects are then organised within a container object called '`Partitions`' whose interface '`IPartitions`' is passed to the analyser module to determine which engine can execute each partition efficiently. The analyser instructs the network resource monitor using its interface '`INetworkResourceMonitor`' to create a

logical network topology of the services and engines involved in the workflow. Following its construction, the network topology's interface 'INetworkTopology' is returned to the analyser. The information in the network topology (e.g. network latency and bandwidth metrics) relating to each network route 'NetworkRoute' is used by the analyser to decide the placement of each workflow partition. Placement decisions are passed to the partitioner in the form of a hash map, in which the key represents the engine name and its value is a list of partitions to be executed by that engine. Based on this information, the partitioner may instruct the composer to combine different sets of partitions together in a single partition.

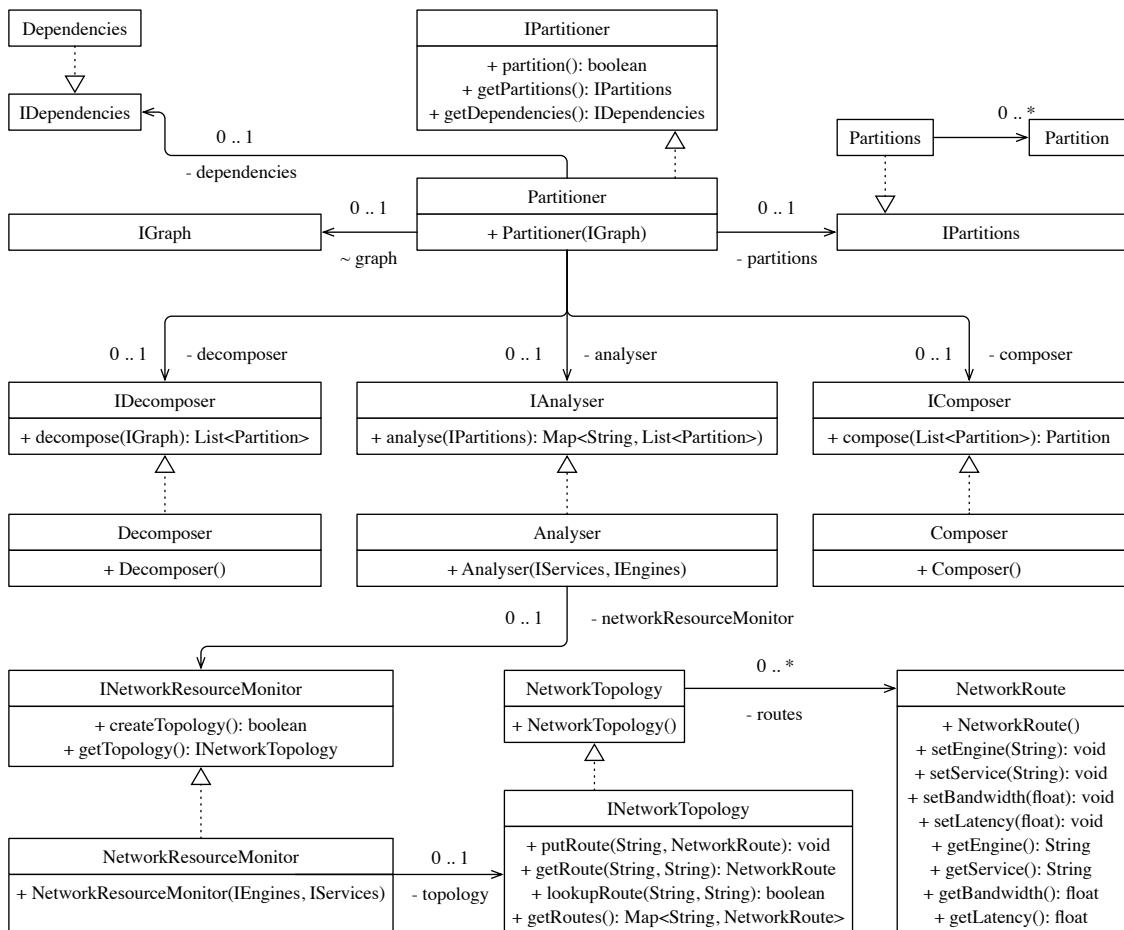


Fig. 5.14 Class diagram showing the workflow partitioning and placement analysis classes.

5.4.6 Execution Module

The execution module is responsible for executing the data structure generated by the compiler module. The mechanism for executing the workflow is has been discussed in Section 4.3.5. It relies on a graph traversal mechanism that explores the data structure to execute *invocation containers* that has also been discussed in Section 5.4.1. The execution module relies on the *Apache CXF*⁷ programming library (version 3.0.4) for creating objects that are used to invoke the services during execution. This library features a service-oriented framework for building and developing front-end programming APIs such as JAX-WS and JAX-RS. It also supports creating web service clients easily without annotations, and a variety of protocols including SOAP, XML/HTTP, RESTful HTTP, CORBA, and others. The execution module also relies on a datastore module for storing information about the workflow being executed and its data.

5.4.7 Datastore Module

The datastore module described in Section 4.2.1 is implemented to maintain information about workflows that have been successfully compiled using the engine, and the data relevant to their execution including initial inputs, intermediate data and final outputs. Following the compilation of a workflow specification, the compiler generates a data structure that represents the workflow which is then serialised into a form that is suitable for storage (e.g. JSON) in the file system of the machine hosting the engine. This permits all information about the workflow such as definitions of service endpoints, the workflow interface (e.g. inputs and outputs), service invocations and associated data dependencies, and typing information to be maintained and retrieved when necessary. The datastore allows the engine to store relevant logging information related to the workflow such as compilation and execution log files which may be used for evaluation purposes.

⁷See the *Apache CXF* project at <https://cxf.apache.org/>

5.5 Conclusion

This thesis has so far presented a decentralised service orchestration approach that is realised through the implementation of the execution engine service which is discussed in this chapter. This engine relies on *Apache Tomcat*⁸ web server (version 7.0) and the Java Run-time Environment (JRE) to work properly. This chapter has presented the implementation of the engine's interface, its execution states, and its internal modules including the compiler of the *Orchestra* language which is discussed in Section 3, the workflow partitioner, the network resource monitor, the placement analyser, the deployment module, the execution module and the datastore. The implementation of this engine meets the requirements outlined in Chapter 2.7 except for fault-tolerance and dynamic reconfiguration support discussed in Sections 2.7.8 and 2.7.9 respectively. The current implementation does not deal with unforeseen problems that can happen at run-time. For example, the system does not provide a mechanism to recover from service failures during the workflow execution. This mechanism requires a reliable server that can monitor failures during the workflow execution and handle these failures by introducing changes in the workflow execution environment. Such changes may involve performing many activities such as deploying services to replace the ones that failed, deploying alternative compute servers (e.g. engines), and reconfiguring the placement of the workflow partitions. Reconfiguration of the workflow partitions requires their execution to be paused, and performing further placement analysis to determine new locations at which the execution of these partitions can continue efficiently. This may involve introducing changes to the specifications of workflow partitions such as changing service definitions, service invocations, service compositions, and data forwarding instructions to transfer data amongst the engines correctly. The study of these issues is beyond the scope of this thesis as it focuses on certain issues relating to the specification of service-oriented workflows and their execution in a decentralised manner.

⁸See the *Apache Tomcat* project at <http://tomcat.apache.org/>

Chapter 6

Evaluation

6.1 Introduction

This chapter provides a detailed evaluation of the decentralised service-oriented orchestration system's implementation presented in this thesis. The purpose of this evaluation is to test the research hypothesis presented in Section 1. This research hypothesis and its scope are examined in Section 6.2, which discusses the rationale behind adopting a decentralised approach to address the limitations of centralised service orchestration and the scalability dimensions used to evaluate the research solution. Section 6.3 discusses the experimental apparatus including the environment used for testing, the hardware specifications of the machines and the software used to conduct different experiments. Section 6.4 describes the methodology used for conducting the experiments (e.g. the specification and orchestration of experimental workflows, and the analysis of their results). Section 6.5 describes the experiments, details the configuration of the services and engines used in the experiments, and presents the results for each experiment. Section 6.6 provides a general discussion of the experimental results, highlights important observations, and identifies different factors that influence the overall performance of orchestrating service-oriented workflows. Section 6.7 concludes this chapter by summarising the evaluation work.

6.2 Examination of Hypothesis

Before proceeding further in this chapter, we closely examine the hypothesis presented in Chapter 1. This hypothesis represents an “educated guess” of a solution that addresses the limitations of completely centralised service-oriented orchestration systems, and it states the following:

“Decentralisation provides a solution that addresses some of the scalability and performance limitations of centralised service orchestration. It can be achieved by decomposing the workflow logic into smaller parts that may be distributed across multiple execution engines at appropriate locations with short network distance to the services providing the data. This reduces the overall data transfer in the workflow and improves its execution time.”

This thesis has argued that decentralised orchestration can be realised using a workflow partitioning approach, which splits the specification of the workflow into smaller parts (e.g. sub workflows), and uses placement analysis to determine the most appropriate engines that can execute the specifications of these parts efficiently. This approach has been discussed in Chapter 4 and it relies on a high-level data coordination language called *Orchestra* for the specification and execution of workflows, which is discussed in Chapter 3. This hypothesis can be evaluated by conducting experiments that attempt to confirm it, and by analysing the experimental results with the predicted consequences of the hypothesis.

6.2.1 Scope

The scope of the research hypothesis covers service-oriented workflows. The research hypothesis takes into consideration the study of decentralised service orchestration and argues that it addresses the limitations of completely centralised service orchestration. It does not

take into consideration the study of a hierarchical (e.g. partially centralised) service orchestration approach and its performance benefits over a completely centralised service orchestration approach. This is because hierarchical service orchestration has been studied and analysed in detail by Barker et al. [213], [280], [28], [281], [282]. The research hypothesis also does not take into consideration the study between hierarchical and decentralised service orchestration approaches in terms of scalability or performance as this is part of future works stated in Section 7.6.

6.2.2 Rationale

This hypothesis suggests that decentralised orchestration can overcome the scalability limitations of a completely centralised orchestration approach based on the following reasons:

1. Firstly, there is no centralised engine acting as a “middle-man” to coordinate the data movement between the services. This can help in avoiding performance bottlenecks and can potentially improve scalability as the number of services participating in the workflow increases.
2. Secondly, the distribution of the specification of the workflow logic supports parallelism, and permits the workflow partitions (e.g. specifications of sub workflows) to be executed independently by multiple engines at network locations nearer to the services providing the data. This can improve the overall service response time (e.g. round-trip times between the services and engines) and the overall throughput.
3. Finally, the distribution of the intermediate data across multiple engines can help in reducing the network traffic and the overall time of data transfer in the workflow. This is because the intermediate data in the workflow is forwarded directly to the network locations where the data is required for computation, without passing through a single point of coordination.

6.2.3 Scalability Dimensions

The scalability of the hypothesised approach is determined by a set of *dimensions* which include the following:

1. Common dataflow patterns.
2. Number of services.
3. Data size.
4. Results of the workflow partitioning and placement analysis algorithms including:
 - (a) Number of partitions.
 - (b) Placement of partitions.

Dataflow Patterns

Dataflow patterns represent the basic building blocks for composing large-scale workflows such as those seen in scientific applications [265], which include the pipeline, data aggregation and distribution patterns. These patterns permit a workflow to be specified in the form of a *Directed Acyclic Graph* (DAG) that indicates which data a particular task in the workflow depends on (e.g. input data required for the task's execution), and the task(s) that produce this data in the first place. The specification of these patterns do not require control-flow structures (e.g. loops, conditionals), which exposes parallelism in the workflow completely. This enables the workflow to detect the workflow tasks that can be executed in parallel automatically, and supports the workflow partitioning process. Therefore, it is important to test the scalability of the presented decentralised approach for orchestrating workflows based on these patterns.

Number of Services

It is important to investigate scalability as the number of services participating in the workflow increases, and how this number can influence the overall workflow performance. For example, the execution time of a workflow may increase as the number of services increase due to the fact that there will be more interactions between the centralised engine and the services. This chapter investigates how decentralised orchestration compares to completely centralised orchestration when executing the same workflow. It presents the evaluation of orchestrating workflows that involve tens of services and engines hosted at different geographic regions in the execution environment (e.g. public cloud) which is discussed in Section 6.3. This thesis does not evaluate the presented decentralised service orchestration approach when executing workflow that involve a larger number of services due to some constraints imposed on the execution environment, which are discussed in Section 6.6.3.

Data Size

Data size refers to the total size of communicated data in the workflow between the engines and the services. It can be used in scalability analysis to compare between completely centralised and decentralised orchestration approaches in terms of performance especially as the data size increases. The introduction of more than a single engine to the workflow may incur additional data transfer between the engines and the services, and between the engines themselves. It is therefore important to investigate the performance of orchestrating a workflow in a decentralised manner as the data size increases. This chapter presents the evaluation of orchestrating experimental workflows which may involve data that range approximately between 10-300 MBs in size. Section 6.6 provides a discussion of the experimental results which clearly indicate the performance benefits of decentralised service orchestration when executing workflows that involve this data of this size range. This thesis does not evaluate the presented decentralised service orchestration approach when executing

experimental workflows that involve data of greater size (e.g. GBs) due to the performance limitations of the web service protocol used for data transfer, and the constraints imposed on the execution environment (e.g. public cloud). These limitations are discussed in Section 6.6.3.

Workflow Partitioning and Placement Analysis Results

The result of the partitioning algorithm can affect the decentralised orchestration performance. It produces a number of partitions that cannot be defined statically by the workflow architect (e.g. programmer or scientist) but it is determined during run-time. Each workflow partition is mapped onto a compute server (e.g. execution engine) that can only be selected during workflow partitioning based on the outcome of placement analysis. Hence, decentralised orchestration can be affected by:

- **Number of partitions or engines:** This refers to the number of workflow partitions, and also to the number of execution engines as each workflow partition is executed by a single engine. During placement analysis, a particular engine can be selected to execute one or more workflow partitions which then would be combined into a single workflow partition. This thesis investigates the behaviour of the workflow partitioning and placement analysis algorithm used to execute the workflow over a series of tests. The outcome of this investigation can be useful to investigate load balancing mechanisms that can be used to enhance the decentralised orchestration approach in future works.
- **Placement of partitions:** It is important to investigate if the network location of the engines responsible for executing the partitions can influence the overall workflow performance. Hence, some environmental factors that relate to the location of the engines must be taken into consideration. These factors include the geographical distance, network latency and network bandwidth between the engines and the services.

It is essential to note that the workflow partitioning process is not necessary when using a centralised service orchestration approach for executing the experimental workflows. This is because each experimental workflow will be executed using a single engine that is responsible for invoking all the services participating in the workflow or composing them.

6.3 Experimental Apparatus

This section describes the experimental environment used for evaluating the presented orchestration approach, and its importance for conducting research. For the past few decades, researchers including Chase et al. [283], Figueiredo et al. [284], Huang et al. [285], and Keahey et al. [286] have been advocating the use of virtual machine environments for service-oriented applications. Much of their work describes the merits of using *Infrastructure as a Service* (IaaS) cloud environments in scientific computing applications. Deelman et al. [287] investigate the effectiveness of clouds in terms of costs for orchestrating scientific workflows. Similar works are presented by Hoffa et al. [288], and Ostermann et al. [289], but none of the works mentioned above consider the actual challenges and technical difficulties of orchestrating workflows in such environments. Unlike grid-based environments, clouds provide provisioning of virtualised computation, storage, and communication resources as explained by Llorente et al. [290], Sotomayor and Montero [291]. This provides flexibility that permits the user to create, launch, and terminate virtual machines on-demand. For example, the user can customise a virtual machine image with a desired operating system, software packages, programming libraries, and a web application platform. Then the user can boot the image using a specialised deployment tool that configures and launches the virtual machine. *Amazon Elastic Compute Cloud* (EC2) was used as an experimental environment for evaluating the decentralised service orchestration system presented in this thesis. Section 6.3.1 describes this environment and the reasons for using it. Section 6.3.2 provides the hardware specifications of the machines used in this environment. Section 6.3.3

provides the specifications of the software installed on these machines for experimentation purposes. Section 6.3.4 describes the setup of the experimental environment.

6.3.1 Experimental Environment

Amazon EC2¹ was used as an experimental environment for evaluating the decentralised service orchestration system presented in this thesis for a number of reasons which include the following:

1. Global service-oriented computing support.
2. Repeatability and reproducibility support for scientific research.
3. Software testing support.

Global Service-oriented Computing Support

Amazon launched EC2 as a global network infrastructure that is specifically designed to give users control over the geographic region at which their services and data can be hosted. *Amazon* EC2 technology are becoming the *de-facto* standard for IaaS management, popularised through their open source re-implementation of *Eucalyptus* [292] according to Srinivasan and Khajeh-Hosseini [293]. *Eucalyptus* is a software for building cloud-based computing environments. There are other alternative cloud-based environments such as *Nimbus*² and *Nebula*³. However, the discussion of these environments is beyond the scope of this thesis and they have been discussed by Keahey et al. [294], and Weissman et al. [295]. Figure 6.1 shows the geographic regions provided by *Amazon* in west and east of North America, in east of South America, in west and central Europe, in east and north east of the Asian pacific.

¹See the *Amazon* EC2 at <http://aws.amazon.com/ec2>

²See the *Nimbus* project at <http://www.nimbusproject.org>

³See the *Nebula* project at <https://www.nebula.com>

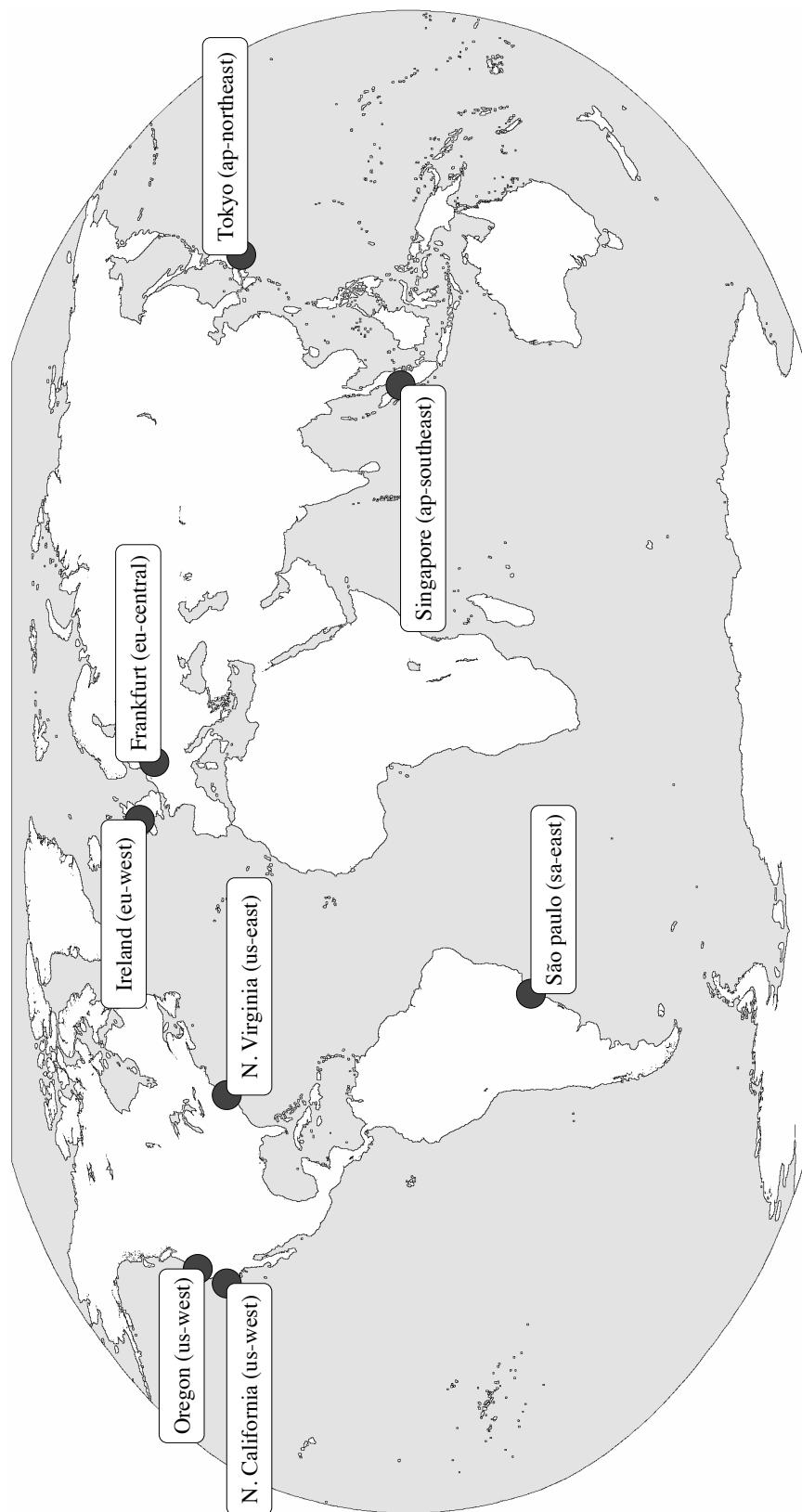


Fig. 6.1 Geographic network regions of *Amazon Elastic Compute Cloud (EC2)*.

Repeatability and Reproducibility Support for Scientific Research

Scientists need to be able to repeat their experiments more than once using difference data sets or reproduce the original results of their experiments for further analysis or perhaps to collaborate with other researchers. For example, scientists often share partial results of their experiments with colleagues to obtain some feedback relating to their research. Dudley and Butte [296] also discuss the cloud computing support for repeatability and reproducibility in scientific research. Cloud computing environments enable data related to experimental research to be published (e.g. stored in public repositories), and shared amongst researchers. *Amazon* has taken the initiative to make scientific data available to researchers to support their collaboration. For example, the *Amazon Public Data Sets*⁴ provide data repositories from many scientific disciplines (e.g. astronomy, biology, chemistry, and environmental science). *Amazon EC2* also enables snapshots (e.g. system images) to be taken of virtualised computer systems that may be used in a scientific experiment. This permits a scientist reconstruct the original execution environment that was used to conduct the experiment. For example, a scientist can use a system image to create instances of virtual machines that run the same scientific applications that were previously used to conduct the original experiment. Several efforts in the field of biomedicine have recently emerged for the development and distribution of cloud-based tools, systems and other resources to support research activities based on the understanding of articles provided by Stein [297] and Schatz [298]. Researchers across a range of computing disciplines have been calling for methodologies to support research reproducibility to assess scientific claims according to Peng [299], whereas Gil et al. [133] examine the challenges in capturing provenance in scientific workflows to support research reproducibility. However, the discussion of these challenges and the construction of workflow provenance tools that support research reproducibility is beyond the scope of this thesis as discussed in Section 2.3.3.

⁴See the *Amazon Public Data Sets* at <http://aws.amazon.com/publicdatasets/>

Software Testing Support

There are significant benefits of using clouds for software testing purposes. Ganon and Zilbershtein [300] discuss such benefits using a use case that was carried out on *Amazon EC2* that resulted in improvements to the software that could not have been highlighted by other feasible means of testing and simulation. Riungu-Kalliosaari et al. [301], [302] reports on similar benefits in using cloud environments for software testing based on interviews conducted with different software development organisations. There have been many research efforts that focus on software testing in cloud-based environments that have been reviewed by Incki et al. [303]. Gao et al. [304] attempt to address common technical difficulties in cloud-based application testing, and examines related challenges. However, the discussion of software testing challenges in cloud computing is beyond the scope of this thesis.

6.3.2 Hardware Specifications

There are different types of virtual machines used in each experiment. *Amazon EC2* refers to a virtual machine as an *instance*⁵, which is capable of running web service applications. Table 6.1 summarises the types of virtual machines used, their processing and memory capabilities, and their network performance. Micro and small instance types have High Frequency Intel Xeon Processors operating at 2.5GHz with Turbo up to 3.3GHz, whereas medium, large and extra large instance types have High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processors or Intel Xeon E5-2670 (Sandy Bridge) Processors running at 2.6 GHz. Several virtual machine instances were used to run a set of services and engines for testing purposes which are discussed in Section 6.3.3. Each virtual machine instance supports a network performance standard that is specified by *Amazon*. This provides some degree of heterogeneity when orchestrating an experimental workflow, as the communication links between different machines may vary in terms of network latency and bandwidth.

⁵See information about *Amazon EC2* instance types at <http://aws.amazon.com/ec2/instance-types>

Table 6.1 Summary of virtual machine types used in the experiments.

Family	Type	Virtual CPUs	Memory (GiB)	Network Performance
Micro	t2.micro	1	1	Low
Small	t2.small	1	2	Low to moderate
Medium	m2.medium	2	4	Low to moderate
Large	m3.large	2	7.5	Moderate
Extra large	m3.xlarge	4	15	High

6.3.3 Software Specifications

Each virtual machine instance has a 64bit *Linux/Unix* operating system installed that runs a pre-configured web server environment based on *Apache Tomcat* version 7, and supported by the *Java SE Runtime Environment* (JRE) version 1.6.0. This enables a web application to be deployed onto a virtual machine running a web server, which handles incoming and outgoing HTTP communication, and performs back-end computational processing. The remainder of this section discusses:

1. Different classes of web service applications including:
 - (a) Engine service.
 - (b) Test service.
2. Deployment of the web service applications onto the virtual machines.

Engine Service

This service represents the engine that is used to orchestrate the workflow, whose implementation has been discussed in Chapter 5 of this thesis. It is responsible for compiling, partitioning, and executing a given workflow specification, and it may collaborate with remote engines. It is essential to note that the computation power and storage resources provided by micro instances are sufficient to run the engine service as it has a small memory footprint and minimal computation power requirements.

Test Service

This service represents a target web service whose operations are invoked by the workflow engine. This target service is used as a generic testing service that produces a random dataset of specific size (e.g. dummy data) based on particular input arguments. It is used to demonstrate the techniques of the workflow engine for orchestrating data-centric workflows without displaying any specific behaviour. For example, a particular test service operation can be invoked with an input parameter of integer value that indicates the number of bytes the service must produce. Some service operations accept datasets as input parameters, which may be combined with resulting data or dropped by the service.

6.3.4 Experimental Environment Setup

The experimental environment must be prepared and configured appropriately before conducting any experiments. This task is not as simple as it sounds because it requires a number of activities that include the creation of virtual machines that may be used to run the engine and test services discussed in Section 6.3.3, installing the implementation of the engine and test services on these machines and any software packages that may be necessary to run these services, configuring and running the services, and testing the services to ensure that they are working properly before conducting any experiments. Furthermore, the deployment of engine and test services onto virtual machine instances at different network regions is a laborious administrative task that requires creating public keys, configuring security groups, and virtual machine templates for each network region which may support a range of virtual machine instance types with different hardware specifications. There is a general lack of tools that provide homogenous access to virtual machine instances in all network regions, and support their deployment in a seamless fashion. Therefore, a considerable amount of research time was directed at finding a solution to this problem. The remainder of this section describes the steps taken to setup the experimental environment.

1. Preparation of engine and test service implementations for deployment purposes.
2. Constructing a couple of base virtual machine snapshots which may be used to instantiate several virtual machines that run the engine and test services in different network regions.
3. Deployment of multiple engine and test services in different network regions, and probing of the services to ensure that they are working properly before conducting experiments.

Preparation of Deployment Files

Before setting up the experimental environment, the engine and test service implementations must be prepared in a form that is suitable for deployment onto virtual machine instances over *Amazon EC2*. Since the implementations of the engine and test services are written in Java, they are wrapped as *Web Application Archive* (WAR) files. This file represents a software bundle of Java classes, server pages, servlets, static web pages and other resources such as programming libraries and files that together constitute a web service application. The advantages of using this file is that it supports the deployment of web service applications easily, and the identification of the web service application version. Typically, this file is generated using a set of tools that are commonly available in most Java-based Integrated Development Environments (IDE) such as *Eclipse*. Hence, *Eclipse* was used to generate the deployment files from the implementations the engine and the test services.

Construction of Base Machine Snapshots

Following the preparation of the deployment files, *Amazon Elastic Beanstalk*⁶ is used to construct a couple of base virtual machines. The first machine runs an engine service,

⁶See the *Amazon Elastic Beanstalk* at <http://aws.amazon.com/elasticbeanstalk>

whereas the second machine runs the test services. *Amazon Elastic Beanstalk* is a cloud-based deployment and resource provisioning service that automates the process of installing applications on virtual machines. This service supports the deployment of web service applications written in many programming languages including Java, Python, Ruby, and others. Developers just have to provide their deployment files to this service and define the configuration of the virtual machines that need to be instantiated, whereas resource provisioning, load-balancing, autoscaling, and resource status monitoring activities are all automatically handled by the *Amazon Elastic Beanstalk* service. However, in order to do this the deployment files must first be uploaded to the *Amazon Simple Storage Service* (S3)⁷, which provides developers with secure, and highly-scalable object storage. The *Amazon Simple Storage Service* provides multiple web service interfaces that enable developers to upload and manipulate different forms of files.

This resulted in creating a tool that permits a web service deployment file to be uploaded to *Amazon Simple Storage Service*, after which this tool interacts with the application programming interface provided by the *Elastic Beanstalk* service to create a new virtual machine instance in a particular network region using the deployment file that has been previously uploaded. This tool uses a set of credentials files (e.g. private keys and security certificates) to access both *Amazon's Simple Storage*, and *Elastic Beanstalk* services. Following the instantiation of virtual machines running the engine and test services, snapshots of these machines are taken and stored using the *Amazon Management Console*⁸. Typically, a snapshot represents a complete system image of the virtual machine and it is also called the *Amazon Machine Image* (AMI). Creating a snapshot of a virtual machine in an *Elastic Beanstalk* environment will produce an image containing the same version of the web service application that was deployed in the virtual machine. However, it is important to make

⁷See the *Amazon Simple Storage Service* at <http://aws.amazon.com/s3>

⁸See the *Amazon Management Console* at <http://aws.amazon.com/console>

sure that the system status of the virtual machine is normal and that the service is working properly before attempting to create a snapshot of that virtual machine instance. This is because *Elastic Beanstalk* makes changes to virtual machine instances during provisioning that can cause issues in the generated snapshot.

Deployment and Probing of Engine and Test Services

Several virtual machines can be instantiated for experimentation purposes using the base machine snapshots created previously. Hence, a tool was developed utilises the programming application interface facilities provided by *Amazon Web Services* to instantiate virtual machines in particular network regions with specific configurations (e.g. HTTP and ICMP support) that run the engine and test services based on the snapshots of the base virtual machines. This process, however, may take several minutes depending on the number of virtual machine instances created in each network region. Following the creation of these virtual machine instances, the tool probes each machine using *ping* to determine that it is reachable (e.g. online) after which it issues a simple HTTP HEAD request to the engine or test service which is hosted on that machine to determine that the service is running properly.

6.4 Experimental Methodology

This section presents the methodology used for orchestrating the experimental workflows discussed in Section 6.5, and analysing the orchestration results. The experimental methodology consists of the following steps:

1. Firstly, the workflows are specified using the *Orchestra* language presented in Chapter 3 of this thesis.
2. Secondly, workflow engines and testing services are deployed onto the experimental environment based on a specific configuration tailored for executing the workflows.

Section 6.5 provides the configuration of engine and test services used for each experiment in detail.

3. Thirdly, the endpoint locations of the engines and services used for testing are typed in the specifications of the experimental workflows.
4. Fourthly, each workflow is executed continuously for a number of iterations using a short-running unit test. For example, a unit test accepts a set of arguments (e.g. number of iterations, workflow specification, and remote engine endpoint used to compile and partition the workflow) to be executed. Unit tests are stopped automatically after detecting a run-time problem, and recording it in a specific log file. This permits the tester to inspect the problem encountered during run-time, and re-starting the unit test after re-configuring the experimental environment if necessary.
5. Fifthly, following the completion of a unit test, relevant logging information is collected and organised accordingly in a spreadsheet to assist in plotting graphs of the performance results. Such logging information includes the compilation time of the workflow in seconds, partitioning details (e.g. time, number of partitions, locations of partitions), and completion time for orchestrating the workflow in seconds, in addition to the total size of data communicated in the workflow in megabytes. Following the documentation of the experimental results, relevant information needed for analysis are computed such as the average execution time over a set of test iterations, standard deviation, and mean speedup rate for decentralised orchestration compared with centralised orchestration.
6. Finally, the experimental results and generated information are used for plotting graphs, and summarised in tables for the purpose of presentation and analysis.

6.5 Experiments

This section describes a set of experimental workflows used to evaluate the implementation of the presented decentralised orchestration approach, and they are also used to investigate the benefits of executing workflows “closer” to the services providing the data. These experiments are classified and described as follows:

1. Exploring the effect of engine locality.
2. Comparing performance between completely centralised and decentralised orchestration approaches as follows:
 - (a) Orchestrating workflows based on common dataflow patterns (e.g. pipeline, data aggregation, and distribution).
 - (b) Orchestrating an end-to-end workflow scenario that combines all common dataflow patterns together.

This section provides a concise description of these experiments as follows:

Exploring the Effect of Engine Locality

This class of experiment aims to investigate the effect of engine locality on performance when accessing an arbitrary service. It measures the average round-trip time required to invoke a service hosted in a particular network region and receive the invocation result using local and remote engines. This experiment is discussed in Section 6.5.2.

Comparing Performance between Centralised and Decentralised Orchestration

This class of experiment aims to evaluate and compare the performance of centralised and decentralised orchestration when orchestrating experimental workflows based on common dataflow patterns (e.g. pipeline, data aggregation and distribution), and end-to-end workflows that combine these patterns together.

6.5.1 Experimental Configuration

The experimental workflows used to compare between centralised and decentralised orchestration may be executed using different configurations based on the placement of services.

The experimental workflows are classified as follows based on the placement of the services:

The classification of experimental workflows based on the placement of services,

1. Regional workflows which are orchestrated as follows:

- (a) Centralised orchestration (e.g. single engine and multiple services in the same network region).
- (b) Decentralised orchestration (e.g. multiple engines and services in the same network region).

2. Inter-regional workflows which are orchestrated as follows:

- (a) Centralised orchestration (e.g. single engine and multiple services in different network regions).
- (b) Decentralised orchestration (e.g. multiple engines and services in different network regions).

The placement of the services determines the manner in which the workflows are orchestrated and the configuration of the engines used to orchestrate them. The remainder of this section describes regional and inter-regional experimental workflows based on the understanding of this classification.

Regional Workflows

This thesis defines a regional configuration as the organisation and placement of services in a workflow, where all the services are hosted in the same network region. Services configured in the same region are orchestrated in both centralised and decentralised manner.

For example, centralised orchestration is carried out using a single engine that is hosted in the same network region (e.g. locally) where the services are hosted, whereas decentralised orchestration is carried out using multiple engines that are hosted locally in the same region.

Inter-regional Workflows

This thesis defines an inter-regional configuration as the organisation and placement of services in a workflow, where the services are hosted in different network regions. Services configured in multiple regions are orchestrated in both centralised and decentralised manner. For example, centralised orchestration is carried out using a single engine that is hosted in an arbitrary network region, whereas decentralised orchestration is carried out using multiple engines that are hosted in different network regions where the services are hosted.

6.5.2 Effect of Engine Locality

This section presents an experiment that aim to evaluate the execution of a workflow based on the process pattern, in which a single service is invoked using a single engine. The purpose of this experiment is to measure the average round-trip time required to invoke a service hosted in a particular network region and receive the invocation result by local and remote engines. The local engine represents a web service that is responsible for invoking a test service, and that is hosted in the same network region where the test service is hosted. The remote engine represents a web service that is responsible for invoking a test service, and that is hosted in a different network region than the network region where the test service is hosted. The remainder of this section discusses the following details:

1. Configuration of the test and engine services.
2. Experimental results.

Configuration

This experiment has been performed using a single test service which has been deployed in N. Virginia. This test service is invoked using a single local engine for a number of iterations. Similarly, the same service is then invoked using remote engines that are deployed in the following regions: N. California, Oregon, Ireland, Frankfurt, Sao Paulo, and Singapore. The average round-trip time for invoking the service is computed for each engine communicating with the service.

Experimental Results and Discussion

Figure 6.2 shows the experimental results for invoking a test service in N. Virginia. The *x-axis* in graph represents the total size of data transmitted to and received from the test service in megabytes, whereas the *y-axis* represents the execution time for invoking the test service and receiving its response in seconds. Based on the experimental results, the best

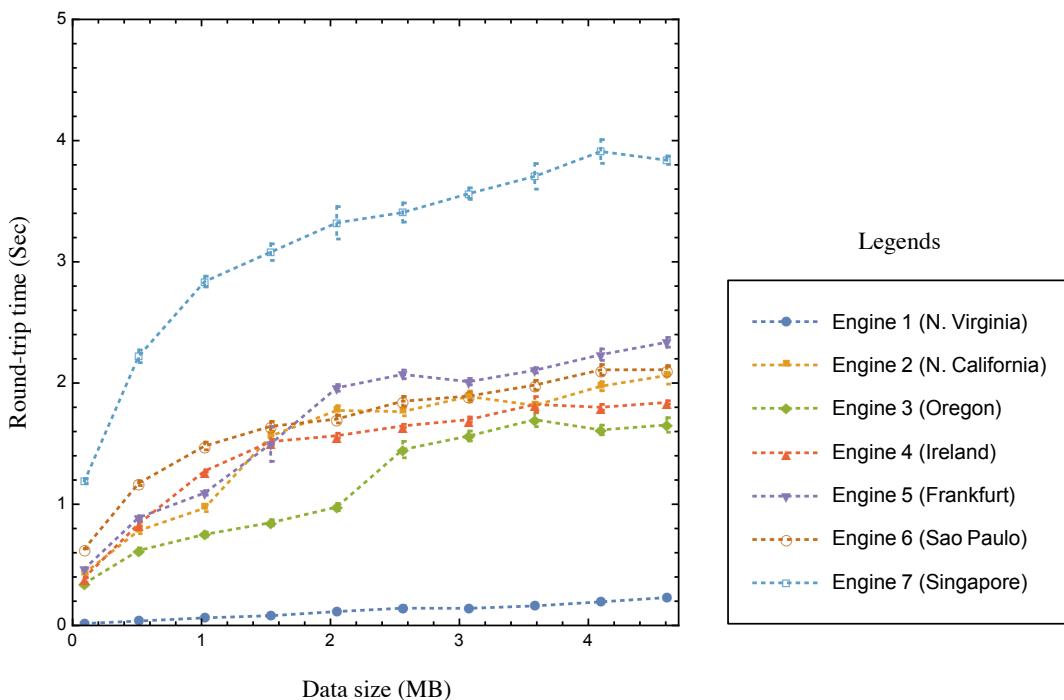


Fig. 6.2 Round-trip times for invoking a service in N. Virginia.

performing engine is hosted in the same network region where the service is hosted. In order to explain these results, the network latency and bandwidth readings between the engines and the service are analysed. Table 6.2 provides a summary of these readings.

Table 6.2 Summary of network metrics between the engines and the service in N. Virginia.

Engine	Location	Avg. Latency (Sec)	Avg. Bandwidth (Kb/Sec)
1	N. Virginia	0.0077	190.57074
2	N. California	0.1708	72.675285
3	Oregon	0.1455	108.552956
4	Ireland	0.1586	79.03856
5	Frankfurt	0.19	65.9668
6	Sao Paulo	0.2574	44.549007
7	Singapore	0.4724	16.523476

Figure 6.3 provides a visualisation of these readings, which demonstrate that the best performing engine which is (1) has the lowest network latency and highest bandwidth with the test service. Such readings indicate that deploying the engine “closer” to the test service improves the round-trip time needed for the whole service invocation to complete.

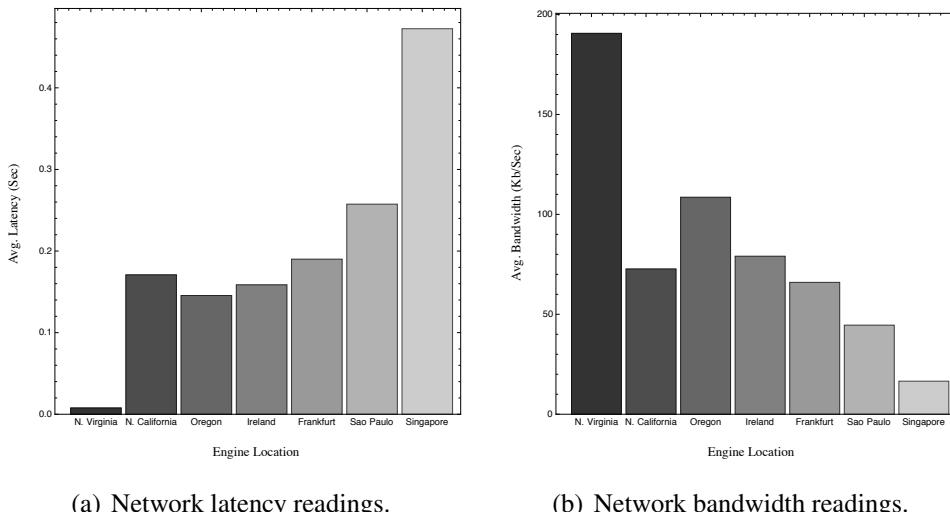


Fig. 6.3 Network latency and bandwidth readings.

6.5.3 Pipeline Pattern Evaluation

These experiments aim to evaluate the execution of workflows based on the pipeline pattern using centralised and decentralised orchestration approaches. They are used to compare the performance between centralised and decentralised orchestration approaches as both the number of services and the total data size communicated in the workflow increase. It is important to investigate this pattern under the presented decentralised orchestration approach. In this pattern, a particular engine would invoke a service operation and obtain its result which may then be forwarded to another engine for a subsequent service invocation. Since the output of one operation is to be used as an input for another, it was not expected that decentralised orchestration may provide significant performance benefits. However, it was not known if the placement of the engines would influence the execution time.

Configuration

This section provides the configuration of the services and engines used in experimental workflows based on the pipeline pattern. Table 6.3 provides a summary of the services configuration for each experimental workflow.

Table 6.3 Configuration of services used in pipeline pattern workflows.

Workflow	Total Number of Services	Configuration	
		Network Region	Num. of Services
1	10	Oregon	10
2	20	N. California	20
3	30	Oregon	10
		N. California	10
		Ireland	10
4	20	N. Virginia	7
		Ireland	3
		Oregon	10

For each network region, there is a number of available engines that may be used to execute the workflows. This number ranges between 10-20 engines.

Experimental Results

Figure 6.4 provides the experimental results of orchestrating the workflows based on the pipeline pattern, where the *x-axis* represents the total size of communicated data between the services in the workflow in megabytes, and the *y-axis* represents the execution time in seconds.

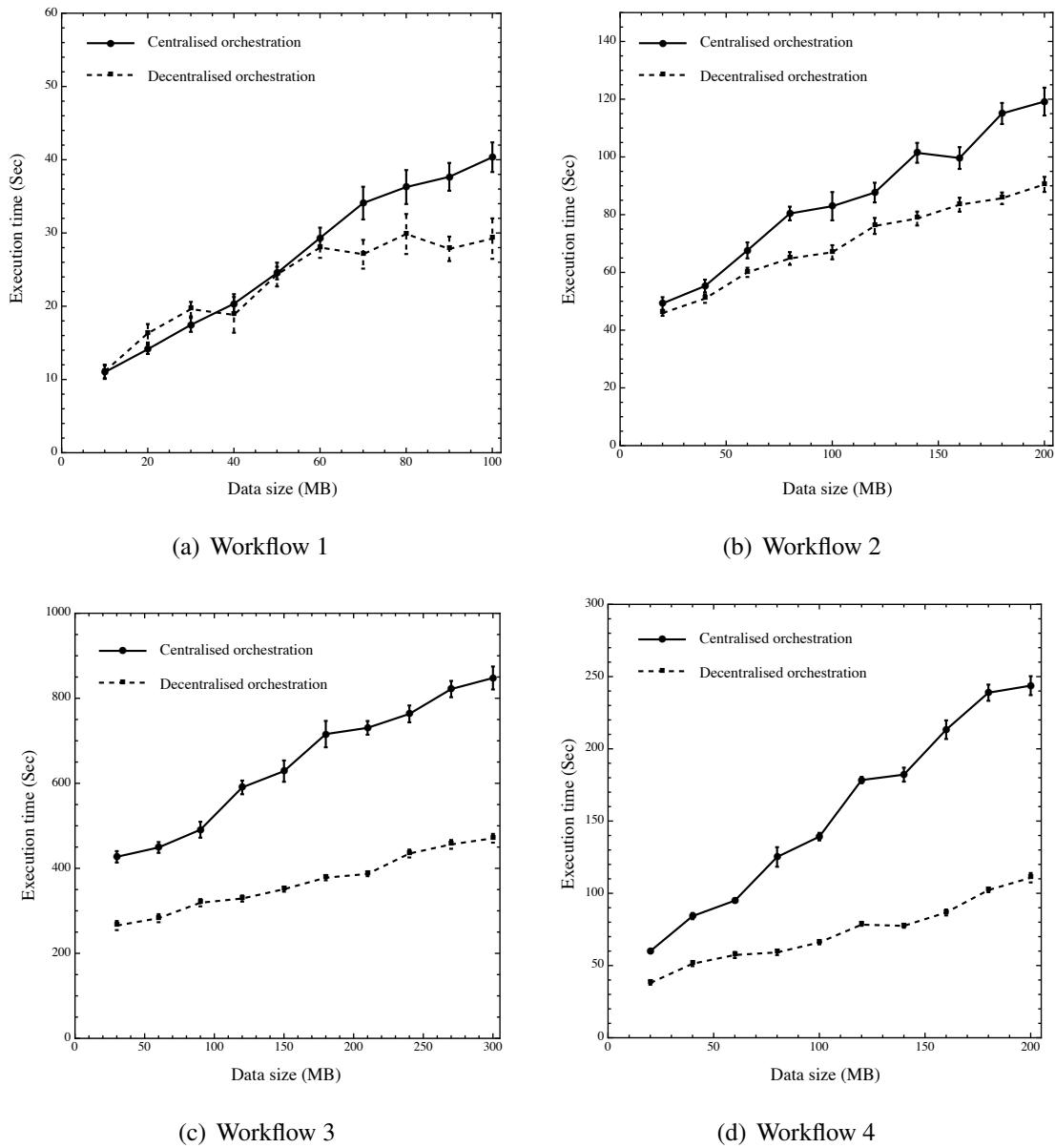


Fig. 6.4 Orchestration of pipeline pattern workflows.

Based on the experimental results provided in Figure 6.4, decentralised orchestration provides some performance improvement over centralised orchestration when executing workflows based on the pipeline pattern. These results are summarised in Table 6.4 which provides the average speedup readings for orchestrating the workflows. It shows the location of the centralised engine used to execute each workflow, and the location of the distributed engines used to execute the workflow in a decentralised manner. This table also shows the average number (e.g. count) of the distributed engines used in each network region, and this number corresponds to the number of partitions being executed as each engine is responsible for executing a single workflow partition.

Table 6.4 Summary of average speedup for orchestrating pipeline pattern workflows.

Workflow	Centralised Engine Location	Decentralised Engines		Avg. Speedup
		Network Regions	Number	
1	Oregon	Oregon	3	1.31
2	N. California	N. California	4	1.23
3	Oregon	Oregon	3	1.74
		N. California	2	
		Ireland	3	
4	Sao Paulo	N. Virginia	3	2.07
		Ireland	1	
		Oregon	4	

6.5.4 Data Aggregation Pattern Experiments

Data aggregation pattern experiments aim to evaluate the orchestration of workflow structured as reductive dataflow graphs, in which the data is consumed gradually as the graph grows inward towards a single sink. Similar to the experimental workflows based on the pipeline pattern, these workflows are used to compare the performance of orchestration using centralised and decentralised approaches as the number of services and the total size of communicated data in the workflow increase. In a centralised orchestration model, the data aggregation pattern would be treated similar to the pipeline dataflow pattern. This is

because although multiple data providing services could send the data directly to a single service that acts as a data sink, they will instead send the data to a centralised engine. The centralised engine will wait for the data from all the services to be received before forwarding it to the final service, and therefore becomes a synchronisation point. The performance of the data aggregation pattern depends on the slowest data providing service from which the engine receives the last data set that is required to invoke the final service. Other data providing services, however, may be invoked concurrently by the centralised engine which may improve the execution time. In this experiment, as the data becomes available from other services, distributed engines acting as proxies to these services may forward such data to the engine that is responsible for invoking the final service concurrently. This helps in reducing the overall time of executing the workflow. However, the choice of the engine that acts as a proxy to the final service is important as placing the engine “closer” to the final service can improve the time for invoking the service and receiving a response from it.

Configuration

This section provides the configuration of the services and engines used in experimental workflows based on the data aggregation pattern. Table 6.5 provides a summary of the services configuration for each experimental workflow.

Table 6.5 Configuration of services used in data aggregation pattern workflows.

Workflow	Total Number of Services	Configuration	
		Network Region	Num. of Services
5	10	Oregon	10
6	20	N. California	20
7	30	N. Virginia	10
		N. California	10
		Frankfurt	10
8	20	N. Virginia	5
		Ireland	5
		Oregon	10

Experimental Results

Figure 6.5 provides the experimental results of orchestrating the workflows based on the data aggregation pattern.

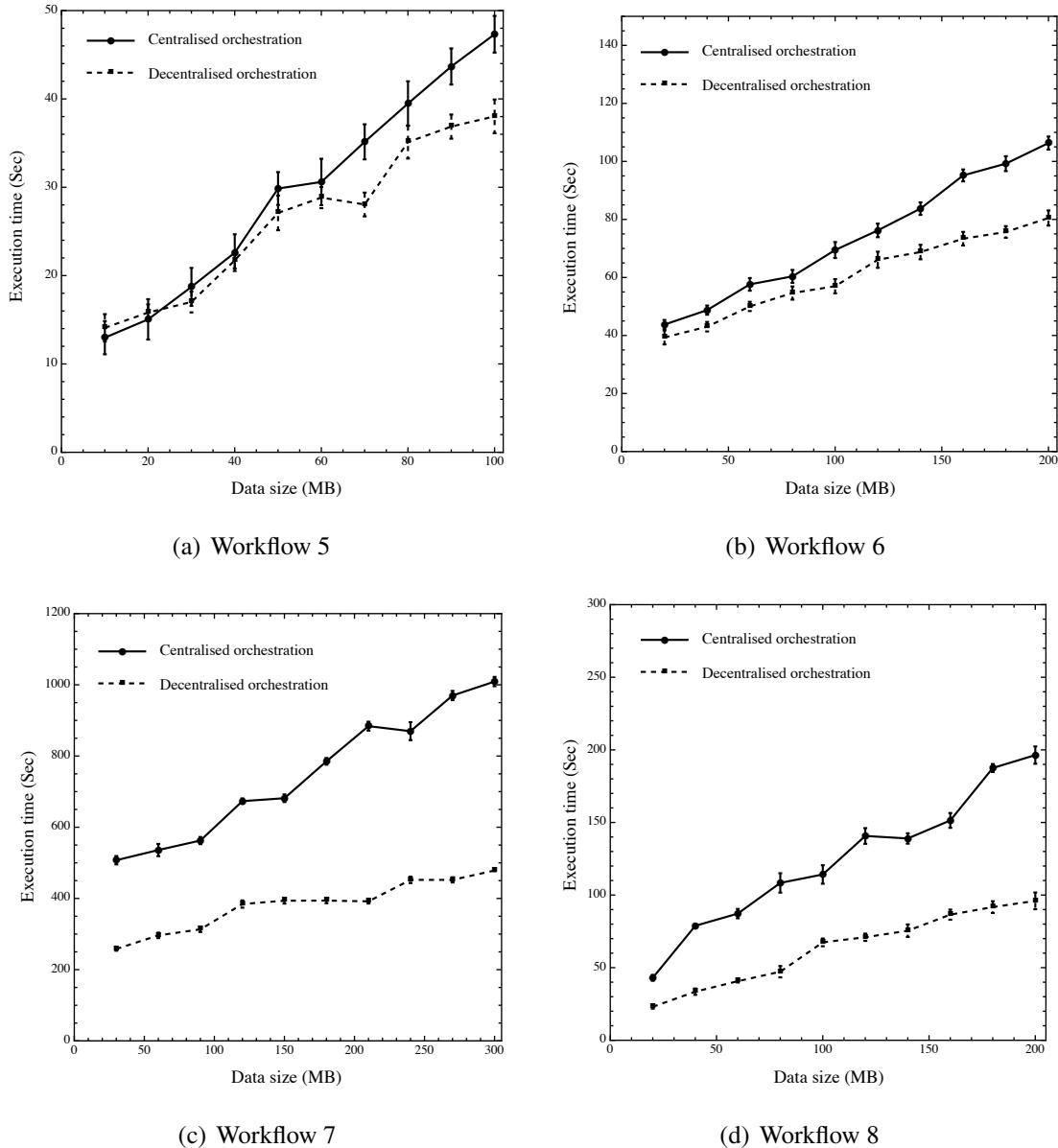


Fig. 6.5 Orchestration of data aggregation pattern workflows.

These results of these experiments are summarised in Table 6.6 which provides the average speedup readings for orchestrating the workflows.

Table 6.6 Summary of average speedup for orchestrating data aggregation workflows.

Workflow	Centralised Engine Location	Decentralised Engines		Avg. Speedup
		Network Regions	Number	
5	Oregon	Oregon	3	1.09
6	N. California	N. California	3	1.20
7	N. California	N. Virginia	4	1.94
		N. California	4	
		Frankfurt	3	
8	N. California	N. Virginia	2	2.00
		Ireland	2	
		Oregon	3	

6.5.5 Data Distribution Pattern Experiments

Data distribution pattern experiments aim to evaluate the orchestration of workflow structures as expansive dataflow graphs, in which more data is produced gradually as the graph grows outward towards multiple sinks. Similar to the experimental workflows based on both the pipeline and data aggregation patterns, these workflows are used to compare the performance of orchestration using centralised and decentralised approaches as the number of services and the total size of communicated data in the workflow increase. This pattern is more complicated than the pipeline pattern but is similar to that of the data aggregation pattern. Each service provides a set of data to be forwarded to one or more services, but unlike the data aggregation pattern there is no single service that acts as a data sink for multiple outputs produced by other services. Hence workflows of this kind can be parallelised by using distributed engines, where each executes part of the workflow concurrently. It is important to remark that using multiple engines necessitates the introduction of additional messages in the workflow to pass intermediate data between them and therefore may increase the overall execution time of the workflow. However, since each engine executes part of the workflow logic “closer” to one or more services the overall time of communicating the data between the engine and the services can be reduced.

Configuration

This section provides the configuration of the services and engines used in experimental workflows based on the data distribution pattern. Table 6.7 provides a summary of the services configuration for each experimental workflow.

Table 6.7 Configuration of services used in data distribution pattern workflows.

Workflow	Total Number of Services	Configuration	
		Network Region	Num. of Services
9	10	Oregon	10
10	20	N. California	20
11	30	N. Virginia	10
		N. California	10
		Frankfurt	10
12	20	N. Virginia	5
		Ireland	7
		Oregon	3

Experimental Results

The experimental results of orchestrating workflows based on the data distribution pattern are summarised in Table 6.8, which provides the average speedup readings for orchestrating the workflows. Figure 6.6 provides a set of graphs relating to these results.

Table 6.8 Summary of average speedup for orchestrating data distribution workflows.

Workflow	Centralised Engine Location	Decentralised Engines		Avg. Speedup
		Network Regions	Number	
9	Oregon	Oregon	3	1.12
10	N. California	N. California	4	1.26
11	N. Virginia	N. Virginia	3	1.90
		N. California	4	
		Frankfurt	3	
12	N. California	N. Virginia	2	1.95
		Ireland	3	
		Oregon	1	

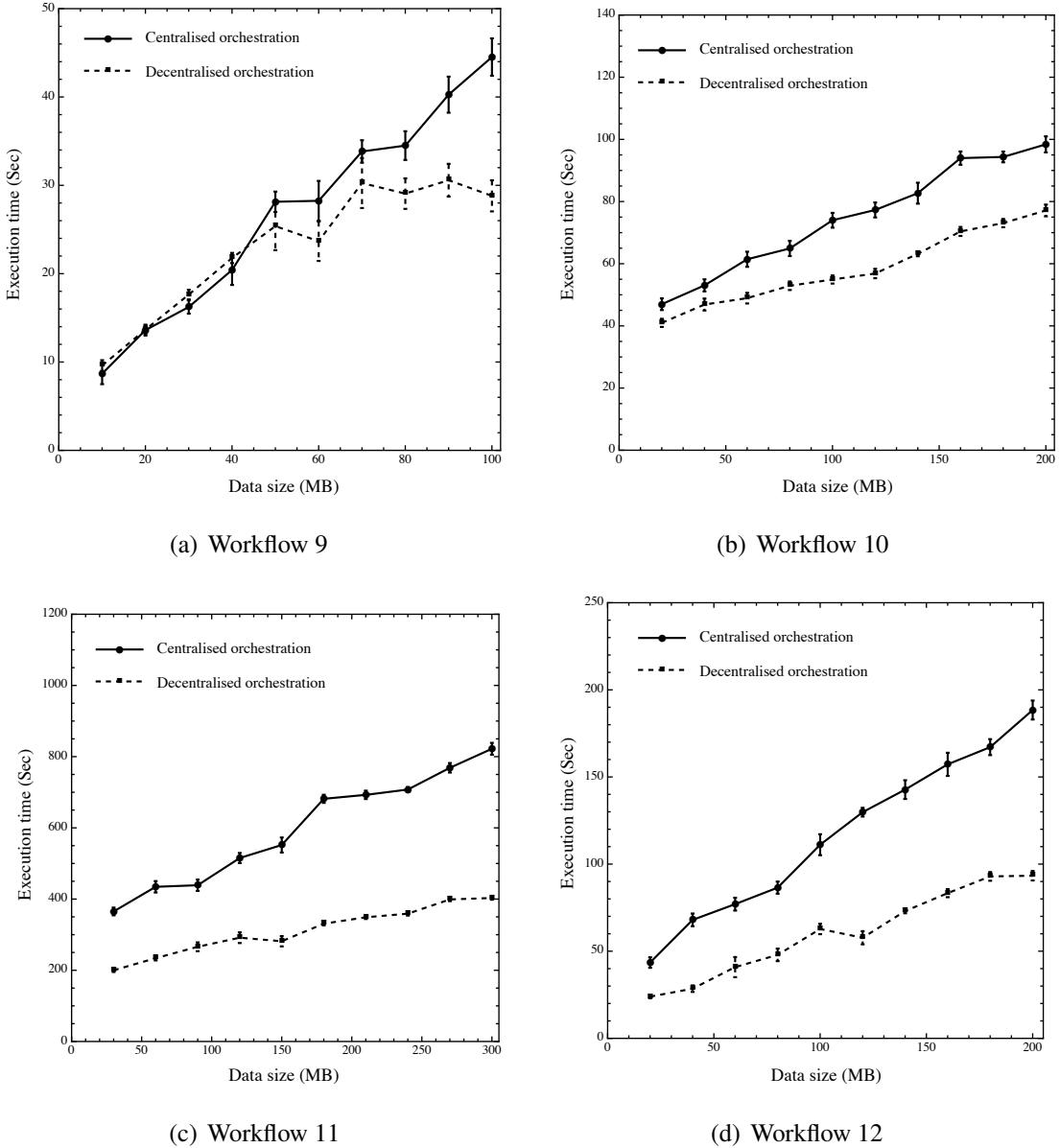


Fig. 6.6 Orchestration of data distribution pattern workflows.

6.5.6 End-to-end Workflow Experiment

Despite the fact that the focus of the experiments has been on pattern-based workflows, it is important to demonstrate the performance of the presented decentralised orchestration approach on an *end-to-end* workflow application that combines all the dataflow patterns presented earlier. This workflow was created by randomly selecting a set of services de-

ployed across different network regions and composing them together in a workflow specification. Figure 6.7 shows the structure of this workflow application which consists of 20 test services. The remainder of this section provides the configuration of testing and engine services, and the experimental results for orchestrating the workflow using centralised and decentralised approaches.

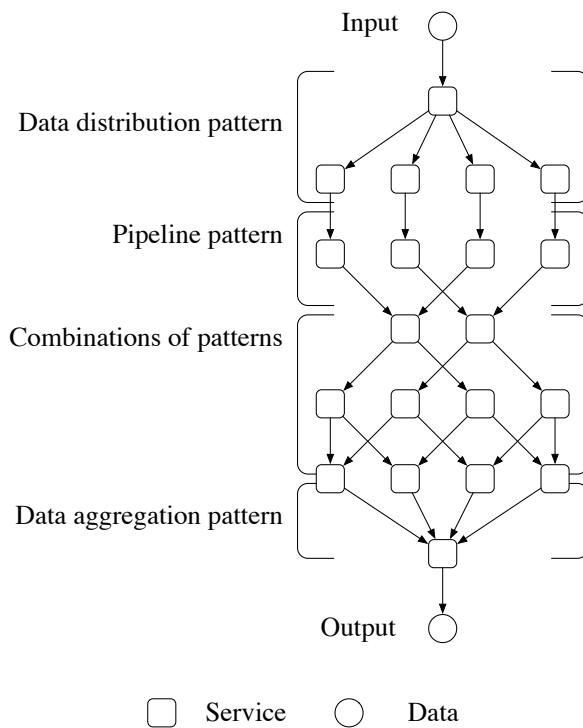


Fig. 6.7 Structure of an end-to-end workflow application.

This workflow involves the interaction of 20 testing services deployed across different regions including N. California, N. Virginia, Oregon, and Ireland where each region consists of 5 testing services and 5 available engines. Figure 6.8 provides the experimental results for orchestrating an end-to-end workflow using centralised and decentralised orchestration. Centralised orchestration of this workflow was performed using a single engine deployed in N. Virginia, whereas several engines were used in each network region to orchestrate the workflow in a decentralised fashion. The average speedup for orchestrating this workflow based on the presented decentralised approach is 1.786.

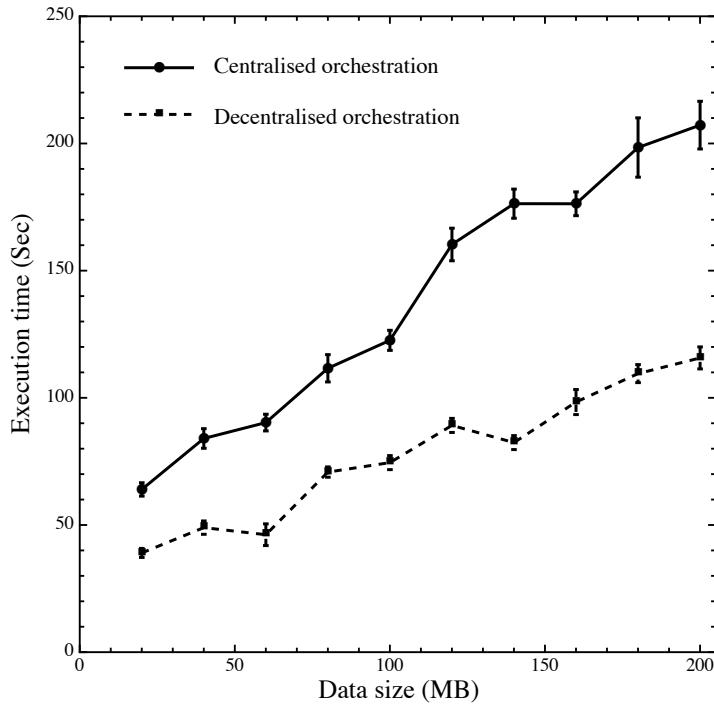


Fig. 6.8 Experimental results for orchestrating an end-to-end workflow that combines common dataflow patterns.

6.5.7 Measurement of Compilation and Partitioning Times

Thus far this chapter has discussed the performance of orchestrating workflows in a decentralised manner. This section provides a brief summary of measurements taken for compiling and partitioning some of the experimental workflows described earlier in this chapter. Table 6.9 provides sample readings of the average time needed to compile and partition a set of experimental workflows. Based on these readings, it is observed that partitioning time normally exceeds compilation time due to the fact that it relies on placement analysis, which takes considerable amount of time due to the following reasons:

1. Placement analysis relies on network resource monitoring to gather information about the services involved in the workflow and the available engines from the network.

2. Partitioning involves traversing the dataflow graph generated by the compiler for decomposition in order to combine them together and encode them in a suitable form for network transport.
3. The compilation process involves obtaining information about the services involved in the workflow by retrieving their service description documents. However, this thesis does not focus on the optimisation of the compilation and partitioning processes because both are performed once for a single workflow.

Table 6.9 Summary of the average compilation and partitioning times for some workflows.

Workflow	Avg. Compilation Time (Sec)	Avg. Partitioning Time (Sec)
1	4.2	14.43
2	9.36	27.71
5	4.86	14.73
6	13.46	30.70
9	7.32	17.12
10	14.34	30.96

6.6 General Discussion

So far this chapter has examined the research hypothesis, defined its scope, explained its rationale and described related scalability dimensions considered for testing the research hypothesis. It has also described the experimental environment, detailed the experimental methodology, and presented a set of experiments to evaluate the decentralised service orchestration approach presented in this thesis. Although the results of these experiments were discussed in previous sections, this section provides a summary of these results in addition to environmental factors that affect the overall performance of experiments, and describes the limitations of the evaluation.

6.6.1 Summary of Experimental Results

The presented decentralised approach provides performance improvement when executing workflows of different structures in different configurations. The experimental results are summarised by the following observations:

1. Typically, a completely centralised engine may take considerable amount of time to execute a workflow often due to passing multiple copies of intermediate data between the engine and the services which may be distributed at different geographic locations. Decentralised orchestration relies on multiple engines that forward the intermediate data directly to the services that require it, without passing through a centralised routing entity.
2. Decentralised orchestration provides some performance improvement over a completely centralised orchestration approach when executing workflows. It provides significant performance improvement over centralised orchestration as both the number of services and the data communicated in the workflow increase. Such performance improvement can be attributed to:
 - (a) The distribution of the workflow logic (e.g. workflow specification). Executing workflow partitions (e.g. specifications of sub workflows) “closer” to the services providing the data reduces the round-trip times for transferring the data between the engines and the services. Consequently, this improves the overall workflow execution time.
 - (b) The distribution of the intermediate data in the workflow. The execution engines permit the intermediate data produced during workflow execution to be transferred directly to network locations where the data is required without a centralised point of coordination. The issue of distributing the intermediate data in the workflow has been discussed in Section 2.5, which provides the execution

cost models for transferring the data in completely centralised and decentralised service orchestration approaches.

This performance improvement is evidenced by the speedup rates summarised in the experimental results.

3. The number of workflow partitions is expected to increase as the number of services in the workflow increases. The frequency with which a specific number of workflow partitions occur depends on the result of placement analysis, which is discussed in Section 4.3.4. From the understanding of the partitioning algorithm discussed in Section 4.3.2 and the experimental results in Section 6.5, it is safe to assume that the partitioning process will attempt to produce as many workflow partitions as necessary to be executed “closer” to the services.
4. The compilation time of any workflow is shorter than the time needed for partitioning the workflow. *Orchestra*’s compiler is lightweight due to the language simplicity, whereas the partitioner component relies on network resource monitoring mechanism that gathers QoS metrics (e.g. network latency and bandwidth) from the execution environment relating to the services participating in the workflow and available execution engines to assist in placement analysis. Hence, partitioning takes more time than compilation.
5. The compilation process usually takes a few seconds to complete as it parses the workflow specification, but may take longer than depending on the number of services specified in the workflow. This is because the compiler attempts to obtain the description documents of these services and analyse the information that these documents contain. The services may be located at remote network locations, and therefore the time needed for the retrieval of their description documents affects the overall compilation time.

6. The QoS metrics (e.g. network latency and bandwidth) between an engine and a service can affect the time required for the service to receive a request message from the engine, and the time for a service response message to be received by the engine (e.g. round-trip time). For example, lower network latency and higher network bandwidth imply faster transfer time between the engine and the service endpoints. This can affect the overall execution time of a workflow that involves multiple service invocations, especially in centralised orchestration as the engine may be hosted in a network region that is geographically distant from the services.

6.6.2 Environmental and Network Factors

This thesis discusses the significance of the environmental and network factors that affect the overall performance of the experiments. These factors include the following:

1. Geographical distance.
2. Network latency.
3. Network bandwidth.

Geographical Distance

Geographical distance is used as a general factor to indicate the most appropriate network region at which to execute a sub workflow efficiently. Based on the analysis of the experiments discussed in this chapter, the geographical distance should be considered as a “coarse factor” that identifies a potential network region for executing a workflow. However, this factor is insufficient to select a group of candidate engines to execute the workflow due to the inaccuracy of its estimation techniques described as follows:

1. Determining the geographical distance between the services and the engines relies on static analysis (e.g. not continuously monitored) that does not take into account the

uncertain nature of the network. The fluctuation of the network latency and bandwidth can cause a particular engine to become inappropriate for orchestrating part of the workflow.

2. Selecting a compute server (e.g. a workflow engine) based on geographical distance may not be scalable. Gwetzman and Seltzer [305] have proposed a technique for placing replicas of compute servers near data sources of high demand by correlating IP addresses and *zip codes*. This establishes a rough estimate of the geographical locations of the compute and data resources, that allows the distance between them to be calculated. This technique relies on a centralised server that calculates the distance in miles to find the closest compute server to serve the client's request, which maintains a large amount of geographical information. The centralised server can become a performance bottleneck as it gathers information from other network endpoints.
3. Predicting the location of a service, or geographical network distance between machines may not be accurate. Padmanabhan and Subramanian [306] provide a tool for predicting the geographic location of a particular service by parsing and analysing information about the service endpoint obtained from domain-name registries and databases using the *whois* protocol, and *ping* from fixed locations. This technique is inefficient as it depends on system administrators' ability to assign meaningful names to servers, and update server information in global registries.
4. The Internet must be modelled as a geometric space in order to estimate geographical distance between machines in accurately, which is not a practical approach. Ng and Zhang [307] proposed this approach in which each machine maintains its own coordinates and network distances to other machines are predicted by computing the Euclidean distance over their coordinates. Similarly, Francis et al. [308] proposes a technique which relies on measurement of the host distance to a set of machines at

landmark locations. However, machines at particular landmarks represent potential bottlenecks because every other host has to measure its distance to the landmarks to compute and update its coordinates. It is also difficult to achieve real-time distance estimation between two endpoints as they may need to obtain measurements to a sufficient number of landmarks, re-compute the coordinates, and share the coordinates information to get the estimation.

5. Jamin et al. [309] examine the network distance prediction problem from a topological perspective. This work proposes an infrastructure service which uses *tracers*, each represents a server that measure the distance between itself and other *tracers* or Internet hosts. This approach, however, requires a global network topological view that relies on the cooperation of network providers. The estimation of the network distance is also inaccurate using this infrastructure as *tracers* need to be placed “closer” to the shortest path between different client hosts, which may not be possible at all times. Theilmann and Rothermel [310] proposed a similar technique that attempts to tackle this problem by constructing a tree of measurement servers, each of which measures the distance to its siblings in the tree. Then hosts are assigned to their closest measurement servers, and the distance between two particular hosts is estimated by the distance between each of their ancestor measurement servers.

Due to these reasons this thesis does not focus on the geographical distance factor for the purpose of placement analysis or experimental evaluation.

Network Latency

Network latency is considered an significant factor in selecting an appropriate engine to invoke a particular service. For example, short network latency between an engine and a particular service implies faster data transfer time between them. Sayal et al. [311] proposed a server selection approach that relies on the network latency estimation between a client

and a set of replicated servers, after which a server with the minimum latency is chosen. However, the authors do not report on the effects of their server selection approach.

Predicting the network latency is important for selecting an appropriate workflow engine to invoke a set of services. Network latency is typically measured by calculating the average *ping* times between a service and its client using the ICMP protocol. Since a workflow engine operates as a client to a service, this factor can be useful to predict the best performing engine in a network region to invoke a particular service. There are a number of shortcomings in determining this metric using *ping*:

1. Service providers may not provide administrative support to access the machines hosting their services using *ping*, which makes it difficult to measure this metric. Furthermore, cloud infrastructures such as *Microsoft Azure* do not permit incoming or outgoing *ping* messages to or from their data centers.
2. Using *ping* does not take into consideration the application layer latencies of the web service.

Therefore, this thesis presents an approach for estimating the network latency between an engine and a service by emulating *ping* using the application layer capabilities. This approach has been discussed in Section 5.4.3 of this thesis.

Network Bandwidth

Round-trip latency does not capture all the information about the quality of the network connection between web services. Network bandwidth is a direct factor that influences the network connection quality. For instance, a higher available bandwidth implies faster data transfer time between a couple of endpoints. Based on Carter and Crovella's works [312], [313], [314], the network bandwidth can be used to select an appropriate compute server, but state that network diagnostic tools are needed to discover this information as

it is not always readily available. The authors provide a tool for detecting the network bandwidth that relies on sending a series of ICMP echo packets from source to destination, and measuring the inter-arrival times between successive packets. This approach is not feasible when orchestrating services that do not permit clients (e.g. engines) to use this protocol to interact with the services. Therefore, the research solution presented in this thesis relies on the HTTP protocol to measure the network bandwidth between an engine and a particular service as discussed in Section 5.4.3. However, the network bandwidth may not be accurate but gives an estimation of how long it takes to send data using a particular engine which can be useful when comparing between the network bandwidth values obtained for various engines.

6.6.3 Evaluation Limitations

This thesis has so far discussed the evaluation of decentralised orchestration scenarios based on data locality for a set of workflows. This section discusses further performance factors that may affect execution time but were not considered in the evaluation including:

1. Service implementation.
2. Hardware capabilities.
3. Testing problems.
4. Lack of experimental scientific services.
5. Limitations of the SOAP protocol.
6. Limitations of *Amazon's EC2* environment.

Service Implementation

Services are developed in the real world differently and therefore the execution time of a service functionality (e.g. operation) depends on the implementation complexity of the service that provides it. For example, different algorithms can be used to design and implement the same operation which may be provided by different services. Furthermore, services may be implemented in a manner such that they interact with back-end resources (e.g. databases) or other remote services. Such interactions are often invisible to the service client (e.g. engine) and influence the overall execution time of the service functionality. This could render the placement of the engine “closer” to the service useless as there may be a long delay in the service response because the service could be waiting for response also from a back-end resource or a remote service. Such issues must be further studied to devise practical solutions to improve service performance, but their discussion is beyond the scope of this thesis. Section 6.3.3 described test services that were used for evaluation purposes. These services are simple and identical in implementation, and provide operations that produce random data of specific size based on input parameters passed to these services during the orchestration of an experimental workflow. However, the evaluation of the presented decentralised orchestration approach does not take into consideration the time needed to produce the data by the test services, as this metric is dependent on the service implementation and the hosting machine’s capabilities. It is essential to note that the evaluation presented in this thesis assumes that the test services are statically deployed over *Amazon EC2* and that the cloud provider does not migrate the implementations of test services for optimisation purposes during the execution of workflows.

Hardware Capabilities

Hardware capabilities including the processing power and memory capacity of machines hosting the services and engines may affect the overall performance of the workflow execu-

tion. However, these metrics are often difficult to detect due to the fact that service providers may not provide access to obtain such information from the machines hosting their services. Furthermore, in cloud-based environments it is difficult to measure the processing power or memory capacity of a particular machine due to automatic management and load balancing mechanisms used by the cloud provider. For example, the hardware specification of a virtual machine hosting a service can change overtime. Furthermore, the cloud provider may not provide complete access to the operating system running on the virtual machine which makes the detection of hardware capabilities impossible. Programming libraries that provide cross-platform, cross-language programming interface to low-level information about the machine hardware and its activities may also become useless when used to detect such information in a virtual machine within a cloud environment. Due to the reasons mentioned above, the presented decentralised orchestration approach does not take into consideration the hardware capabilities.

Testing Problems

The experimental environment of *Amazon EC2* is not infallible. Currently, virtual machines provided by current cloud providers including *Amazon* do not exhibit a stable performance according to Dejun et al. [315], and Iosup et al. [316]. During testing, virtual machine instances can be created and launched on physical machines that may suffer from technical disk or network failures. Such virtual machine instances become inaccessible, and therefore a workflow engine or a target testing service cannot be reached. Troubleshooting these instances is also difficult as it becomes almost impossible to connect to the failed instance. Such a problem can sometimes be detected during the compilation of a workflow specification, which identifies the workflow engines or target web services that cannot be reached. However, virtual machine instances may fail during the execution of the workflow. This halts the workflow execution and forces the user to restart the test after troubleshooting the

problem which is often solved by restarting the virtual machine instances or creating new ones. The testing process becomes more challenging as both the number of services and execution engines participating in the workflow increase. Furthermore, an iterative test for orchestrating an experimental workflow consisting of a hundred services can take hours or even days to complete. This makes it difficult to monitor the workflow at all times, and address any emergent problems when they arise. Therefore, a test must be restarted after inspecting its execution log to determine if any problems were recording during the test's execution.

Lack of Experimental Scientific Services

During the infancy of this research, it was challenging to find existing scientific web services that can be used to demonstrate the benefits of decentralised orchestration in a realistic workflow scenario. Despite the fact that there are existing service-oriented infrastructures for capturing and sharing workflows such as the ^{my}Experiment [20] project, it was difficult to find a working example. For example, existing workflow specifications may depend on services which are no longer available or maintained by service providers. Therefore, the author of this thesis decided to create the test services discussed in Section 6.3.3 to be used in experimental workflows that can be organised to have a structure that is similar to scientific workflows.

Limitations of the SOAP Protocol

Services can suddenly fail during the execution of a workflow due to the limitations of the SOAP protocol used to encode the messages passed to and from the services. For example, a particular test service may not be able to deserialise a SOAP message that contains datasets larger than 20 MBs of data size unless the configuration of the server application (e.g. *Apache Tomcat*) and the memory capacity of the virtual machine instance are modified

accordingly. Chiu et al. [80] state that SOAP requires to be modified or extended to support distributed scientific computing application. During testing, a number of test services used to fail soon after receiving one or more incoming SOAP messages containing large datasets. Furthermore, other services used to fail when attempting to encapsulate the data within an outgoing SOAP message. Due to such issues encountered during testing, it was decided to evaluate the decentralised orchestration approach by executing workflows that involve a small number of services and datasets of moderate sizes. This thesis does not attempt to address the performance limitations of the SOAP protocol as these are addressed by Abu-Ghazaleh et el. [79], and Chiu et al. [80].

Limitations of Amazon's EC2 Environment

Setting up *Amazon's* EC2 environment to conduct experiments is often difficult due to the fact that the tester requires to do a number of activities such as creating virtual machines in different regions, deploying the software applications are related dependencies (e.g. programming language support and libraries) required for testing on the instances of these machines, and configuring these machines if necessary. Such activities and their challenges were discussed in Section 6.3.4 which discusses the steps taken to setup *Amazon's* EC2 environment for evaluation purposes. Furthermore, each network region can host a limited number of virtual machine instances of a particular type based on a pre-defined “quota” by the provider. This has affected the number of engines and test services that can be used in each experimental workflow.

6.7 Conclusion

This chapter has presented an evaluation of the decentralised service orchestration approach presented in this thesis. It has examined the research hypothesis, its scope, rationale, and scalability dimensions. It has also presented the experimental environment and its setup,

and discussed the experimental methodology used to test the research hypothesis based on orchestrating a set of experimental workflows. These experimental workflows comprised of workflows based common dataflow patterns (e.g. pipeline, data aggregation and distribution) and an end-to-end workflow that combines these patterns. The results of these experimental workflows were used to demonstrate the performance benefits of a decentralised service orchestration approach over a completely centralised service orchestration approach. These results were discussed in [268], and other experimental results were presented in [267]. This chapter has also provided a general discussion that summarised the experimental results, discussed the environmental and network factors that influenced the evaluation, and described the limitations of the evaluation.

Chapter 7

Conclusion and Future Research

7.1 Introduction

The rapid expansion of data known as the *data deluge* is transforming scientific research and the manner in which scientific experiments are conducted. These experiments are typically specified or modelled as workflows that involve data and computation services. This thesis has provided a general understanding of scientific workflows, and the challenges in the design (e.g. specification or modelling) and orchestration of these workflows. It has investigated existing workflow technologies including workflow management systems, workflow specification languages, dataflow optimisation system architectures, and workflow scheduling technologies. Based on the comprehensive study of these technologies, the author of this thesis has devised a set of requirements that scientific workflow management systems and languages must address, which led the author to build a decentralised service-oriented orchestration system architecture that supports the execution of scientific workflows. This architecture relies on a simple data coordination language for the specification of workflows that involve data or computation services that may be distributed at different geographic locations. This chapter recalls the motivation of this research in Section 7.2, the research hypothesis in Section 7.3, summarises the thesis in Section 7.4 and its contributions to the

body of knowledge in Section 7.5. It identifies the areas that require further investigation and suggests possible future research directions in Section 7.6. Finally, Section 7.7 concludes the work presented in this thesis with final remarks.

7.2 Thesis Motivation

This thesis was motivated by the notion that service-oriented technology holds great potential for managing scientific workflows, but its promise is undermined by the following challenges:

1. Centralised orchestration is an inadequate approach for executing data-intensive service-oriented workflows such as those seen in scientific domains [12]. It presents a significant scalability problem as it relies on a single compute server that may become a performance bottleneck.
2. Creating workflows is a difficult task and often requires a set of computing skills that are beyond the skills of a scientist. Existing technology used to compose service-oriented workflows force scientists to deal with low-level issues such as describing how to execute the workflow tasks instead of what tasks need to be executed.
3. Current orchestration systems do not support the automation of mapping independent workflow tasks onto computing resources. It is impractical to decompose the workflow tasks and map them onto computing resources manually, especially when the complexity of the workflow increases.
4. It is important to take control of the distribution of computation over the locality of data, and react to the uncertainty of the network to avoid underperformance issues. Therefore, adequate measures must be taken to execute the workflow tasks “closer” to the services providing the data in the workflow.

7.3 Thesis Hypothesis

The tested hypothesis is predicted based on the argument orchestrating workflows needed for modern scientific data analysis presents a significant scalability challenge: they are typically executed in a manner such that all data pass through a centralised engine, which causes unnecessary network traffic that leads to a performance bottleneck. Based on this argument, the research hypothesis was formulated as follows:

“Decentralisation provides a solution that addresses some of the scalability and performance limitations of centralised service orchestration. It can be achieved by decomposing the workflow logic into smaller parts that may be distributed across multiple execution engines at appropriate locations with short network distance to the services providing the data. This reduces the overall data transfer in the workflow and improves its execution time.”

This hypothesis suggests that decentralised orchestration can overcome the scalability limitations of a completely centralised orchestration approach due to the lack of a centralised engine, the distribution of the workflow logic, and the distribution of the intermediate data across multiple engines. The lack of a centralised engine can help in avoiding performance bottlenecks. The distribution of the workflow logic permits the workflow to be decomposed into smaller partitions (e.g. sub workflows) to be executed independently using engines hosted at different network locations “closer” to the services providing the data. This can improve the overall service response time, and the overall throughput. The distribution of the intermediate data across the engines can help in reducing the network traffic and the overall time of data transfer in the workflow. This is because the intermediate data in the workflow is forwarded directly by the engines to network locations where the data is required, without passing through a centralised engine.

7.4 Thesis Summary

This thesis began with an introduction to scientific workflows, and outlined the research motivation, identified the central limitations in orchestration service-oriented workflows in data-centric scientific applications, and proposed a research hypothesis in Chapter 1. It continued in Chapter 2 with a general introduction to service computing and the web service model, and presented a comprehensive review of workflow technologies including workflow management systems, workflow languages, workflow scheduling approaches, and dataflow optimisation system architectures. Based on the comprehensive study of these technologies, a set of requirements for workflow management systems and languages were listed and discussed. Having identified the shortcomings of centralised service orchestration and the limitations of existing technology, but finding none to meet the full set of listed requirements, a new high-level orchestration language called *Orchestra* was presented in Chapter 3 that satisfies these requirements. This language permits a workflow architect (e.g. scientist) to define a set of services and compose them in a workflow specification without previous knowledge of how the workflow is executed. Chapter 4 presented a detailed design of a decentralised service-oriented system architecture for orchestrating workflow specifications based on the presented *Orchestra* language. The implementation of this architecture was provided in Chapter 5. It was used in a series of experiments for the evaluation of the proposed research solution which was provided in Chapter 6. This chapter summarises the research work and contributions, and states future research directions.

7.5 Review of Contributions

This thesis provides a set of contributions to the body of research knowledge in the area of service computing generally, and service orchestration specifically. These contributions include the following:

1. Literature review.
2. Requirements analysis and specification.
3. Data coordination language for the specification of service-oriented workflows.
4. Decentralised service-oriented orchestration system architecture.
5. Optimisation techniques for:
 - (a) Partitioning technique for decomposing a workflow into smaller sub workflows.
 - (b) Placement analysis algorithm mapping workflows onto execution engines).
6. Experimental evaluation.

Literature Review

This thesis presents a comprehensive study of workflow technologies that include workflow management systems, workflow languages, workflow scheduling techniques, and dataflow optimisation system architectures. By analysing these works this thesis has been able to highlight the limitations of existing technologies, and identify the research challenges in the design (e.g. modelling) and orchestration of scientific workflows.

Requirements Specification

This thesis defines a set of concrete requirements for composing and orchestrating scientific service-oriented workflows. These requirements are not currently satisfied by existing workflow specification languages and management systems.

Data Coordination Language

This thesis presents a high-level, functional data coordination language called *Orchestra* for the specification of service-oriented workflows. This language separates the workflow

logic from its execution, supports parallelism, determinism, and data-driven execution. It provides simple abstractions for defining a set of services, composing them and regulating the data movement between them.

Decentralised Service-oriented Orchestration System Architecture

This thesis presents a decentralised architecture that is designed to orchestrate a service-oriented workflows based on the *Orchestra* language using multiple compute servers (e.g. execution engines). Each engine is responsible for analysing and executing part of the overall workflow logic by exploiting connectivity to the services, and forwarding the intermediate data directly to locations where it is required.

Optimisation Techniques

This thesis presents a set of techniques for optimising the workflow execution. These techniques include the following:

- **Partitioning technique:** This thesis provides a novel technique that decentralises a workflow by transforming its specification into smaller partitions, each represents an independent sub workflow. These partitions can be executed in parallel to improve the overall workflow performance. This approach analyses the data dependencies in a workflow to detect its intricate parallel parts that can be isolated, and relies on placement heuristics to determine a set of appropriate workflow engines to execute these partitions.
- **Placement analysis algorithm:** This thesis presents a placement analysis technique that is responsible for determining a candidate engine to execute a particular workflow partition. It relies on QoS metrics that are gathered from the execution environment such as the network latency and bandwidth between the engines and the services. These metrics are used to estimate the network distance between the engines and the

services that may be hosted at different geographic locations. This technique is useful for selecting the nearest engine to a certain service for executing a relevant workflow partition. Consequently, all workflow partitions are deployed onto a set of designated workflow engines for execution, and their deployment activity is transparent to the user (e.g. scientist). Following deployment, real-time distributed monitoring is used to watch the workflow execution to address emergent run-time issues such as unexpected failures.

Experimental Evaluation

This thesis evaluates the implementation of the presented decentralised service-oriented orchestration system architecture in terms of scalability and performance by orchestrating a set of experimental workflows over *Infrastructure as a Service* (IaaS) clouds across several geographical regions.

7.6 Future Research Directions

This thesis provides the implementation of a decentralised orchestration approach as a proof of concept. In the evaluation of this approach, a number of potential research directions that can extend this work were identified. This section presents these directions and provides a discussion for each.

7.6.1 Further Evaluation

This thesis has provided an evaluation of a decentralised service-oriented system architecture that relies on the *Orchestra* language presented in Chapter 3. Further evaluation is required to demonstrate the performance of the presented system in orchestrating realistic scientific workflow scenarios. Future research works also include the evaluation of the *Or-*

chestra language. This can be achieved by comparing it against other existing scientific workflow languages to determine if it can be used to represent realistic scientific workflow scenarios, and to demonstrate that representations of such workflows in other languages can also be mapped to *Orchestra*.

7.6.2 Dynamic Optimisation

Dynamic optimisation is required to re-configure the deployment of executing workflow partitions. From the deployment process onwards in the system implemented, the analyser does not interfere with the execution by producing a new deployment plan in response to dynamic changes in the network environment. Such changes are often unpredictable and may include unexpected service response delays, external load on the engine as it may be responsible for executing multiple workflows, changes in the network latency or bandwidth. Dynamic optimisation steers the execution process and aims to minimise performance losses due to unpredictable changes that disrupt the static deployment plan computed by the analyser. Given the ability to control the state of the executing workflow partitions, a dynamic optimisation mechanism can be used to pause the overall execution, after which the workflow partitions are refactored to be redeployed in new network locations at which their execution can resume and progress efficiently. K. Lee et al. [317] claim that static decisions previously made regarding the mapping of tasks onto resources can be revised during execution to support adaptation to the environment. Dynamic optimisation requires a real-time distributed monitoring approach to track the execution progress of the workflow partitions, and to collect information about the network condition that can be used to devise new deployment plans. Furthermore, dynamic optimisation may change the overall workflow structure by combining workflow partitions or decomposing them further into smaller partitions, and modifying information relating to the engines and instructions to route the data between the engines in the workflow partitions. The features required for implementing dynamic opti-

misation to re-configure the deployment of workflows have been discussed in Chapter 4 in detail. Based on works from the literature discussed in Chapter 2, most existing workflow management systems provide static approaches for mapping tasks onto compute resources. Besides the adaptation to dynamic changes in the environment as discussed earlier, dynamic optimisation can be used in workflow evolution [318] as a business process workflow can change depending on external business requirements, rather than changes in the execution environment and as such are beyond the scope of this thesis.

7.6.3 Real-time Distributed Monitoring of Network Resources

Quality of Service (QoS) information including the network latency and bandwidth metrics are all captured by the network resource monitor from the engines, and maintained in a dedicated datastore in the implementation. Currently, the implementation of the network resource monitor gathers and processes such information in the internal memory as it arrives from the engines before populating the datastore with it. This is not a scalable approach for the following reasons:

1. Firstly, the network resource monitor has to wait for all the engines to return such information, which can take a considerable amount of time.
2. Secondly, as the number of services increase in the workflow more incoming messages from the engines will need to be processed by the network resource monitor, which can result in overloading the memory.
3. Finally, the size of the information collected by the network resource monitor is expected to grow rapidly. Hence, a cleaning mechanism is used to clear outdated information, but this mechanism may need to be performed frequently to provide more space for further information to be stored.

There are a number of possible solutions to these problems. The network resource monitor can employ a mechanism to perform incremental analysis based on a statistical approach, which predicts the overall condition of the network between the engines and the services. This can reduce the overall time needed to perform network resource monitoring, and avoid overloading the internal memory. Such a mechanism, however, may not provide accurate predictions and will require full understanding of the decentralised cost model and the complexity of the workflow. Therefore, it is preferable to design a real-time distributed network resource monitoring mechanism which permits the information collected by the engines to be stored locally, and processed on demand during placement analysis. This improves the overall time required for monitoring the network resources by reducing the overhead of propagating information about these resources to the main network resource monitor.

7.6.4 Constraint-based Partitioning and Deployment

This thesis has focused on the deployment of workflow partitions based on heuristic placement analysis, which takes into consideration the condition of the network environment. It is possible to extend this approach using constraint-based analysis that permits the workflow to be partitioned and deployed based on goals specified by the user. For example, the user can specify a workflow and express constraints over aspects such as the mapping of tasks onto particular engines for execution. Following the compilation of the workflow specification, a constraint solver can be used to find a deployment configuration that satisfies the goals specified by the user. Based on this configuration, the workflow is partitioned into smaller parts which are deployed and executed automatically. During execution, the workflow progress is monitored to detect if the specified goals are no longer being met in order to generate a revised deployment plan that does meet these goals. This can be achieved by providing declarative abstractions in the *Orchestra* language to express constraint-based goals to control the deployment of specific parts of the workflow onto engines. It may be

appropriate to specify such constraint-based goals within an independent placement policy that can be bundled with the workflow specification. This permits the user to modify the placement policy without introducing changes to the original workflow specification, in order to address external requirements related to estimated service usage costs, performance, resilience to failures, etc. This means of course that a specialised constraint solver must be designed to support placement policies. It would be interesting to compare the differences in scalability and performance between the presented decentralised orchestration approach and that which is based on constraint programming techniques in the future. There are very few works that focus on constraint-based approaches for the specification of workflows. Some of these works can be found in [319], [320] and [321], but none of which focus on partitioning the workflow and deploying its parts in specific network locations for execution.

7.6.5 Failure Handling and Recovery

Dealing with failures is an essential requirement in a workflow management system as discussed in Section 2.7.8. Gil et al. [133] examine the challenges of dealing with failures in scientific workflow applications, and indicates the need for mechanisms to detect failures and recover from them. Future research works include investigating methods to handle failures during the execution of workflows and recovering from these failures.

7.7 Concluding Remarks

This thesis has covered the design and orchestration of service-oriented scientific workflows. The objective of this thesis was to provide greater understanding and hopefully a stronger base on which to create a new kind of scientific workflow management technology. This thesis has focused particularly on the scalability and performance challenges of centralised service-oriented orchestration approaches which include the unnecessary consumption of

the network bandwidth, high latency in transmitting data between the services, and potential performance bottlenecks. This thesis has presented and evaluated a novel decentralised service-oriented orchestration system architecture that permits a workflow to be partitioned into smaller sub workflows, which may then be transmitted to appropriate network locations at which their execution takes place using distributed compute servers (e.g engines). These network locations are carefully determined using a placement analysis algorithm (e.g. a heuristic technique) that relies on the knowledge gathered from the execution environment. This permits the specification of workflow logic to be moved across execution engines, and executed within short distance (e.g. short latency) to the services, which improves the overall performance. The end of this thesis does not mark the end of the research work it presented, but an opportunity to address many research problems discussed in Section 7.6.

References

- [1] G. Bell, T. Hey, and A. Szalay, “Beyond the data deluge,” *Science*, vol. 323, no. 5919, pp. 1297–1298, American Association for the Advancement of Science, 2009.
- [2] F. Berman, G. Fox, and A. J. Hey, *Grid computing: making the global infrastructure a reality*, vol. 2. John Wiley and Sons, 2003.
- [3] J. Gray, “Jim Gray on eScience: A transformed scientific method,” *The fourth paradigm: Data-intensive scientific discovery*, pp. xvii–xxxii, Microsoft Research, 2009.
- [4] C. Goble and D. De Roure, “The impact of workflow tools on data-centric research,” *The fourth paradigm: Data-intensive scientific discovery*, pp. 137–145, Microsoft Research, 2009.
- [5] A. J. Hey and A. E. Trefethen, “The data deluge: An e-Science perspective,” *Grid computing: Making the global infrastructure a reality*, pp. 809–824, John Wiley and Sons, 2003.
- [6] S. Callaghan, E. Deelman, D. Gunter, G. Juve, P. Maechling, C. Brooks, K. Vahi, K. Milner, R. Graves, E. Field, D. Okaya, and T. Jordan, “Scaling up workflow-based applications,” *Journal of Computer and System Sciences*, vol. 76, no. 6, pp. 428–446, Elsevier, 2010.
- [7] N. Kaiser, H. Aussel, B. E. Burke, H. Boesgaard, K. Chambers, M. R. Chun, J. N. Heasley, K. W. Hodapp, B. Hunt, R. Jedicke, D. Jewitt, R. Kudritzki, G. Luppino, M. Maberry, E. Magnier, D. G. Monet, P. M. Onaka, A. J. Pickles, P. H. Rhoads, T. Simon, A. Szalay, I. Szapudi, D. J. Tholen, J. L. Tonry, M. Waterson, and J. Wick, “Pan-STARRS: a large synoptic survey telescope array,” in *Astronomical Telescopes and Instrumentation*, pp. 154–164, International Society for Optics and Photonics, 2002.
- [8] Y. Simmhan, R. Barga, C. van Ingen, E. Lazowska, and A. Szalay, “Building the Trident scientific workflow workbench for data management in the Cloud,” in *Proceedings of the 3rd International Conference on Advanced Engineering Computing and Applications in Sciences*, pp. 41–50, IEEE, 2009.
- [9] H. R. Butcher, “LOFAR: First of a new generation of radio telescopes,” in *Proceedings of SPIE*, vol. 5489, pp. 537–544, International Society for Optics and Photonics, 2004.

- [10] G. Almasi, S. Asaad, R. E. Bellofatto, H. R. Bickford, M. A. Blumrich, B. Brezzo, A. A. Bright, J. R. Brunheroto, J. G. Castanos, D. Chen, and G. L. T. Chiu, “Overview of the IBM Blue Gene/P project,” *IBM Journal of Research and Development*, vol. 52, pp. 199–220, IBM Corp., 2008.
- [11] B. Plale, D. Gannon, J. Brotzge, K. Droege, J. Kurose, D. McLaughlin, R. Wilhelmsen, S. J. Graves, M. Ramamurthy, R. D. Clark, S. Yalda, D. A. Reed, E. Joseph, and V. Chandrasekar, “CASA and LEAD: Adaptive cyberinfrastructure for real-time multiscale weather forecasting,” *IEEE Computer*, vol. 39, no. 11, pp. 56–64, IEEE, 2006.
- [12] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su, and K. Vahi, “Characterization of scientific workflows,” in *Proceedings of the 3rd Workshop on Workflows in Support of Large-Scale Science*, pp. 1–10, IEEE, 2008.
- [13] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berrianan, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, “Pegasus: A framework for mapping complex scientific workflows onto distributed systems,” *Scientific Programming*, vol. 13, no. 3, pp. 219–237, IOS Press, 2005.
- [14] G. B. Berrianan, E. Deelman, J. Good, J. C. Jacob, D. S. Katz, A. C. Laity, T. A. Prince, G. Singh, and M. Su, “Generating complex astronomy workflows,” *Workflows in e-Science*, pp. 19–38, Springer, 2006.
- [15] G. B. Berrianan, E. Deelman, J. C. Good, J. C. Jacob, D. S. Katz, C. Kesselman, A. C. Laity, T. A. Prince, G. Singh, and M. Su, “Montage: a Grid-enabled engine for delivering custom science-grade mosaics on demand,” in *Proceedings of SPIE*, vol. 5493, pp. 221–232, International Society for Optics and Photonics, 2004.
- [16] M. Rynge, G. Juve, J. Kinney, J. Good, G. B. Berrianan, A. Merrihew, and E. Deelman, “Producing an infrared multiwavelength galactic plane atlas using Montage, Pegasus and Amazon Web Services,” in *Proceedings of the 23rd Annual Astronomical Data Analysis Software and Systems (ADASS) Conference*, vol. 485, pp. 211–214, Astronomical Society of the Pacific, 2014.
- [17] D. A. Brown, P. R. Brady, A. Dietz, J. Cao, B. Johnson, and J. McNabb, “A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis,” *Workflows in e-Science*, pp. 39–59, Springer, 2007.
- [18] E. Deelman, S. Callaghan, E. Field, H. Francoeur, R. Graves, N. Gupta, V. Gupta, T. H. Jordan, C. Kesselman, P. Maechling, J. Mehringer, G. Mehta, D. Okaya, K. Vahi, and L. Zhao, “Managing large-scale workflow execution from resource provisioning to provenance tracking: The Cybershake example,” in *Proceedings of the 2nd International Conference on e-Science and Grid Computing*, pp. 14–14, IEEE, 2006.
- [19] S. Parastatidis, “A platform for all that we know: creating a knowledge-driven research infrastructure.,” *The fourth paradigm: Data-intensive scientific discovery*, pp. 165–172, Microsoft Research, 2009.

- [20] D. De Roure, C. Goble, and R. Stevens, “The design and realisation of the virtual research environment for social sharing of workflows,” *Future Generation Computer Systems*, vol. 25, no. 5, pp. 561–567, Elsevier, 2009.
- [21] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda, “Mapping abstract complex workflows onto Grid environments,” *Journal of Grid Computing*, vol. 1, no. 1, pp. 25–39, Springer, 2003.
- [22] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny, “Pegasus: Mapping scientific workflows onto the Grid,” in *Grid Computing*, vol. 3165 of *Lecture Notes in Computer Science*, pp. 11–20, Springer, 2004.
- [23] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman, “Workflow management in GridPhyN,” in *Grid Resource Management*, pp. 99–116, Springer, 2004.
- [24] K. Lee, N. W. Paton, R. Sakellariou, E. Deelman, A. A. Fernandes, and G. Mehta, “Adaptive workflow processing and execution in Pegasus,” *Concurrency and Computation: Practice and Experience*, vol. 21, no. 16, pp. 1965–1981, John Wiley and Sons, 2009.
- [25] S. Majithia, D. W. Walker, and W. A. Gray, “Automated composition of semantic Grid services,” in *Proceedings of the 3rd UK e-Science All Hands Meeting*, pp. 363–370, The Engineering and Physical Sciences Research Council (EPSRC), 2004.
- [26] T. Erl, *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2005.
- [27] B. Ludäscher, M. Weske, T. McPhillips, and S. Bowers, “Scientific workflows: Business as usual?,” in *Proceedings of the 7th International Conference on Business Process Management*, pp. 31–47, Springer, 2009.
- [28] A. Barker, C. D. Walton, and D. Robertson, “Choreographing Web services,” *IEEE Transactions on Services Computing*, vol. 2, no. 2, pp. 152–166, IEEE, 2009.
- [29] D. Box, “Code name Indigo: A guide to developing and running connected systems with Indigo,” *Microsoft Developer Network (MSDN) magazine*, 2004.
- [30] L. Cardelli, “Global computation,” *ACM Computing Surveys (CSUR)*, vol. 28, pp. 163–166, ACM, 1996.
- [31] L. Cardelli, “Abstractions for mobile computation,” in *Proceedings of Secure Internet Programming*, pp. 51–94, Springer, 1999.
- [32] H. Haas and A. Brown, “Web services glossary,” *World Wide Web Consortium (W3C) Working Group Note*, 2004.
- [33] I. J. Taylor and A. Harrison, *From P2P to Web services and Grids: peers in a client/server world*. Springer Science & Business Media, 2006.
- [34] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, “Service-oriented computing: a research roadmap,” *International Journal of Cooperative Information Systems*, vol. 17, pp. 223–255, Springer, 2008.

- [35] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, and B. A. Hamilton, “Reference model for service oriented architecture 1.0,” *Organization for the Advancement of Structured Information Standards (OASIS)*, vol. 12, 2006.
- [36] D. Kaye, *Loosely coupled: the missing pieces of Web services*. RDS Strategies LLC, 2003.
- [37] S. Chatterjee and J. Webber, *Developing enterprise Web services: an architect’s guide*. Prentice Hall Professional, 2004.
- [38] C. Pautasso and E. Wilde, “Why is the Web loosely coupled?: a multi-faceted metric for service design,” in *Proceedings of the 18th international conference on World Wide Web*, pp. 911–920, ACM, 2009.
- [39] D. Woods and T. Mattern, *Enterprise SOA: Designing IT for Business Innovation*. O’Reilly Media, Inc., 2006.
- [40] W. Vogels, “Web services are not distributed objects,” *Internet Computing, IEEE*, vol. 7, no. 6, pp. 59–66, IEEE, 2003.
- [41] M. Papazoglou, *Web services: principles and technology*. Pearson Education, 2008.
- [42] S. Zhou, “LSF: Load sharing in large heterogeneous distributed systems,” *Workshop on Cluster Computing*, 1992.
- [43] D. Jackson, Q. Snell, and M. Clement, “Core algorithms of the Maui scheduler,” in *Job Scheduling Strategies for Parallel Processing*, pp. 87–102, Springer, 2001.
- [44] R. Henderson and D. Tweten, “Portable batch system: External reference specification,” *Technical report, NASA*, 1996.
- [45] M. J. Litzkow, M. Livny, and M. W. Mutka, “Condor - a hunter of idle workstations,” in *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 104–111, IEEE, 1988.
- [46] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, “Condor: a distributed job scheduler,” *Beowulf cluster computing with Linux, MIT Press*, pp. 307–350, 2001.
- [47] D. Thain, T. Tannenbaum, and M. Livny, “Condor and the Grid,” *Grid computing: Making the global infrastructure a reality*, pp. 299–335, John Wiley and Sons, 2003.
- [48] R. Raman, M. Livny, and M. Solomon, “Matchmaking: Distributed resource management for high throughput computing,” in *Proceedings of the 7th International Symposium on High Performance Distributed Computing*, pp. 140–146, IEEE, 1998.
- [49] R. Raman, M. Livny, and M. Solomon, “Resource management through multilateral matchmaking,” in *Proceedings of the 9th International Symposium on High-Performance Distributed Computing*, pp. 290–291, IEEE, 2000.
- [50] R. Raman, *Matchmaking frameworks for distributed resource management*. PhD thesis, University of Wisconsin at Madison, 2001.

- [51] P. E. Krueger, *Distributed scheduling for a changing environment*. PhD thesis, University of Wisconsin at Madison, 1988.
- [52] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, “SETI@ home: an experiment in public-resource computing,” *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, ACM, 2002.
- [53] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.
- [54] I. Foster, C. Kesselman, and S. Tuecke, “The open Grid services architecture,” *The Grid: Blueprint for a New Computing Infrastructure*, vol. 1, Morgan Kaufmann Publishers, 2004.
- [55] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, “The physiology of the Grid: An open Grid services architecture for distributed systems integration,” *Globus Project*, pp. 217–249, John Wiley and Sons, 2004.
- [56] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, D. Snelling, and P. Vanderbilt, “Open Grid Services Infrastructure (OGSI): Version 1.0,” 2003.
- [57] J. Frey, S. Graham, and C. Kesselman, “Grid service specification,” *Open Grid Service Infrastructure. Technical Report, Global Grid Forum, Draft*, vol. 2, 2002.
- [58] I. Foster and C. Kesselman, “Globus: A metacomputing infrastructure toolkit,” *International Journal of High Performance Computing Applications*, vol. 11, no. 2, pp. 115–128, SAGE Publications, 1997.
- [59] I. Foster and C. Kesselman, “The Globus toolkit,” *The Grid: blueprint for a new computing infrastructure*, pp. 259–278, Morgan Kaufmann Publishers, 1999.
- [60] R. Srinivasan, “RPC: Remote procedure call protocol specification version 2,” 1995.
- [61] O. M. Group, *The Common Object Request Broker (CORBA): Architecture and Specification*. Object Management Group, 1995.
- [62] R. Java, “Java remote method invocation,” *Sun Microsystems Inc.*, 2010.
- [63] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova, “Overview of GridRPC: A remote procedure call API for Grid computing,” in *Grid Computing*, pp. 274–278, Springer, 2002.
- [64] Q. Ho, W. Cai, and Y. Ong, “Design and implementation of an efficient multi-cluster GridRPC system,” in *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid*, vol. 1, pp. 358–365, IEEE, 2005.
- [65] S. Shirasuna, H. Nakada, S. Matsuoka, and S. Sekiguchi, “Evaluating Web services based implementations of GridRPC,” in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pp. 237–245, IEEE, 2002.

- [66] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke, “From open Grid services infrastructure to WS-Resource framework: Refactoring & evolution,” 2004.
- [67] G. Banavar, T. Chandra, R. Strom, and D. Sturman, “A case for message oriented middleware,” in *Proceedings of the 13th International Symposium on Distributed Computing*, pp. 1–18, Springer, 1999.
- [68] S. Vinoski, “RPC under fire,” *IEEE Internet Computing*, vol. 9, pp. 93–95, IEEE, 2005.
- [69] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, *A note on distributed computing*. Springer, 1997.
- [70] S. Parastatidis, J. Webber, P. Watson, and T. Rischbeck, “WS-GAF: a framework for building Grid applications using Web services,” *Concurrency and Computation: Practice and Experience*, vol. 17, pp. 391–417, Wiley Online Library, 2005.
- [71] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, and S. Weerawarana, “Modeling stateful resources with Web services,” *Globus Alliance*, 2004.
- [72] M. Atkinson, D. DeRoure, A. Dunlop, G. Fox, P. Henderson, T. Hey, N. Paton, S. Newhouse, S. Parastatidis, A. Trefethen, P. Watson, and J. Webber, “Web service Grids: an evolutionary approach,” *Concurrency and Computation: Practice and Experience*, vol. 17, pp. 377–389, Wiley Online Library, 2005.
- [73] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, “Simple Object Access Protocol (SOAP) 1.1,” 2000.
- [74] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol - HTTP/1.1,” 1999.
- [75] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “eXtensible Markup Language (XML) 1.0,” 1998.
- [76] S. Loughran and E. Smith, “Rethinking the Java SOAP stack,” in *Proceedings of the 3rd IEEE International Conference on Web Services*, vol. 5, IEEE, 2005.
- [77] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, “Simple Object Access Protocol (SOAP) 1.2,” *World Wide Web Consortium (W3C)*, 2003.
- [78] J. Cowan and D. Megginson, “XML information set,” *World Wide Web Consortium (W3C) Working Draft*, 1999.
- [79] N. Abu-Ghazaleh, M. Lewis, and M. Govindaraju, “Differential serialization for optimized SOAP performance,” in *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pp. 55–64, IEEE, 2004.
- [80] K. Chiu, M. Govindaraju, and R. Bramley, “Investigating the limits of SOAP performance for scientific computing,” in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pp. 246–254, IEEE, 2002.

- [81] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, “Unraveling the Web Services Web: an introduction to SOAP, WSDL, and UDDI,” *IEEE Internet Computing*, vol. 6, no. 2, pp. 86–93, IEEE Computer Society, 2002.
- [82] P. Biron and A. Malhotra, “XML schema part 2: Datatypes,” *World Wide Web Consortium (W3C) Recommendation*, 2004.
- [83] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, “Web Services Description Language (WSDL) 1.1,” *World Wide Web Consortium (W3C) Note*, 2001.
- [84] D. Booth and C. K. Liu, “Web Services Description Language (WSDL) version 2.0 part 0: Primer,” *World Wide Web Consortium (W3C) Recommendation*, vol. 26, 2007.
- [85] L. Mandel, “Describe REST Web services with WSDL 2.0,” *IBM Rational Software Developer*, 2008.
- [86] M. Gudgin, A. Lewis, and J. Schlimmer, “Web Services Description Language (WSDL) version 2.0 part 2: Message exchange patterns,” *World Wide Web Consortium (W3C) Working Draft*, 2004.
- [87] J. Nitzsche, T. Van Lessen, and F. Leymann, “WSDL 2.0 message exchange patterns: limitations and opportunities,” in *Proceedings of the 3rd International Conference on Internet and Web Applications and Services*, pp. 168–173, IEEE, 2008.
- [88] R. Chinnici, M. Hadley, and R. Mordani, “The Java API for XML-based Web Services (JAX-WS) 2.0,” *Sun Microsystems*, vol. 38, April 2006.
- [89] T. Bellwood, L. Clément, D. Ehnebuske, A. Hately, M. Hondo, Y. Husband, K. Januszewski, S. Lee, B. McKee, and J. Munter, “The Universal Description, Discovery and Integration (UDDI) specification,” *Technical committee report, Organization for the Advancement of Structured Information Standards (OASIS)*, 2002.
- [90] S. Dustdar and M. Treiber, “A view based analysis on Web service registries,” *Distributed and Parallel Databases*, vol. 18, pp. 147–171, Springer, 2005.
- [91] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana, “The next step in Web services,” *Communications of the ACM*, vol. 46, no. 10, pp. 29–34, ACM, 2003.
- [92] L. F. Cabrera, C. Kurt, and D. Box, “An introduction to the Web services architecture and its specifications,” *Microsoft Developer Network (MSDN) Library*, 2004.
- [93] H. Kreger, “Fulfilling the Web services promise,” *Communications of the ACM*, vol. 46, pp. 29–34, ACM, 2003.
- [94] H. R. M. Nezhad, B. Benatallah, F. Casati, and F. Toumani, “Web services interoperability specifications,” *Computer*, vol. 39, no. 5, pp. 24–32, IEEE, 2006.
- [95] F. Cabrera, G. Copeland, T. Freund, J. Klein, D. Langworthy, D. Orchard, J. Shewchuk, and T. Storey, “Web Services Coordination (WS-Coordination),” *BEA, IBM, and Microsoft*, 2002.

- [96] J. Salas, F. Perez-Sorrosal, M. Patiño-Martínez, and R. Jiménez-Peris, “Ws-Replication: a framework for highly available web services,” in *Proceedings of the 15th International Conference on World Wide Web*, pp. 357–366, ACM, 2006.
- [97] F. Cabrera, G. Copeland, B. Cox, T. Freund, J. Klein, T. Storey, and S. Thatte, “Web Services Transaction (WS-Transaction),” *Microsoft and IBM*, 2002.
- [98] D. Bunting, M. Chapman, O. Hurley, M. Little, J. Mischkinsky, E. Newcomer, J. Webber, and K. Swenson, “Web Services Composite Application Framework (WS-CAF),” 2003.
- [99] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed, “Deploying and managing Web services: issues, solutions, and directions,” *The International Journal on Very Large Data Bases*, vol. 17, no. 3, pp. 537–572, Springer, 2008.
- [100] S. Vinoski, “WS-nonexistent standards,” *IEEE Internet Computing*, vol. 8, pp. 94–96, IEEE, 2004.
- [101] C. A. Ellis and G. J. Nutt, “Office information systems and computer science,” *ACM Computing Surveys (CSUR)*, vol. 12, no. 1, pp. 27–60, ACM, 1980.
- [102] A. Barker and J. Van Hemert, “Scientific workflow: a survey and research directions,” in *Parallel Processing and Applied Mathematics*, pp. 746–753, Springer, 2008.
- [103] D. Hollingsworth, “Workflow management coalition specification: the workflow reference model,” *Workflow Management Coalition*, 1994.
- [104] W. M. Van Der Aalst, L. Aldred, M. Dumas, and A. H. ter Hofstede, “Design and implementation of the YAWL system,” in *Proceedings of the 16th International Conference on Advanced Information Systems Engineering*, vol. 3084, pp. 142–159, Springer, 2004.
- [105] L. Liu, C. Pu, and D. Ruiz, “A systematic approach to flexible specifications, composition, and restructuring of workflow activities,” *Journal of Database Management (JDM)*, vol. 15, pp. 1–40, IGI Global, 2004.
- [106] J. A. Miller, D. Palaniswami, A. P. Sheth, K. J. Kochut, and H. Singh, “WebWork: METEOR2’s Web-based workflow management system,” *Journal of Intelligent Information Systems*, vol. 10, no. 2, pp. 185–215, Springer, 1998.
- [107] G. Alonso, R. Günthör, M. Kamath, D. Agrawal, A. El Abbadi, and C. Mohan, “Exotica/fmdc: a workflow management system for mobile and disconnected clients,” in *Databases and Mobile Computing*, pp. 27–45, Springer, 1996.
- [108] F. Leymann and D. Roller, “Business process management with FlowMark,” in *Compcos Spring ’94, Digest of Papers*, pp. 230–234, IEEE, 1994.
- [109] P. Grefen and R. R. de Vries, “A reference architecture for workflow management systems,” *Data & Knowledge Engineering*, vol. 27, pp. 31–57, Elsevier, 1998.

- [110] A. Tsalgatidou, G. Athanasopoulos, M. Pantazoglou, C. Pautasso, T. Heinis, R. Grønmo, H. Hoff, A.-J. Berre, M. Glittum, and S. Topouzidou, “Developing scientific workflows from heterogeneous services,” *ACM Sigmod Record*, vol. 35, no. 2, pp. 22–28, ACM, 2006.
- [111] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. B. Jones, E. A. Lee, J. Tao, and Y. Zhao, “Scientific workflow management and the kepler system,” *Congcurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.
- [112] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, *Workflows for e-Science: scientific workflows for Grids*. Springer, 2014.
- [113] P. Romano, “Automation of in-silico data analysis processes through workflow management systems,” *Briefings in Bioinformatics*, vol. 9, no. 1, pp. 57–68, Oxford University Press, 2008.
- [114] M. Shields, “Control- versus data-driven workflows,” in *Workflows for e-Science*, pp. 167–173, Springer, 2007.
- [115] W. M. van Der Aalst, A. H. ter Hofstede, B. Kiepuszewski, and A. P. Barros, “Workflow patterns,” *Distributed and parallel databases*, vol. 14, no. 1, pp. 5–51, Springer, 2003.
- [116] W. M. van der Aalst, A. P. Barros, A. H. ter Hofstede, and B. Kiepuszewski, “Advanced workflow patterns,” in *Cooperative Information Systems*, pp. 18–29, Springer, 2000.
- [117] A. Barros, M. Dumas, and A. H. ter Hofstede, “Service interaction patterns,” in *Business Process Management*, pp. 302–318, Springer, 2005.
- [118] F. Leymann and D. Roller, *Production workflow: concepts and techniques*. Prentice Hall, 2000.
- [119] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann, and M. Reiter, “Conventional workflow technology for scientific simulation,” in *Guide to e-Science*, pp. 323–352, Springer, 2011.
- [120] R. Barga and D. Gannon, “Scientific versus business workflows,” in *Workflows for e-Science*, pp. 9–16, Springer, 2007.
- [121] E. Deelman, D. Gannon, M. Shields, and I. Taylor, “Workflows and e-Science: An overview of workflow system features and capabilities,” *Future Generation Computer Systems*, vol. 25, no. 5, pp. 528–540, Elsevier, 2009.
- [122] I. Brandic, S. Pllana, and S. Benkner, “An approach for the high-level specification of QoS-aware Grid workflows considering location affinity,” *Scientific Programming*, vol. 14, pp. 231–250, IOS Press, 2006.
- [123] Y. L. Simmhan, B. Plale, and D. Gannon, “A survey of data provenance in e-Science,” *ACM Sigmod Record*, vol. 34, pp. 31–36, ACM, 2005.

- [124] R. Stevens, J. Zhao, and C. Goble, “Using provenance to manage knowledge of in silico experiments,” *Briefings in bioinformatics*, vol. 8, no. 3, pp. 183–194, Oxford University Press, 2007.
- [125] S. Bowers, T. McPhillips, M. Wu, and B. Ludäscher, “Project histories: Managing data provenance across collection-oriented scientific workflow runs,” in *Proceedings of Data Integration in the Life Sciences*, pp. 122–138, Springer, 2007.
- [126] C. Lim, S. Lu, A. Chebotko, and F. Fotouhi, “Prospective and retrospective provenance collection in scientific workflow environments,” in *Proceedings of the IEEE International Conference on Services Computing (SCC)*, pp. 449–456, IEEE, 2010.
- [127] S. B. Davidson and J. Freire, “Provenance and scientific workflows: challenges and opportunities,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1345–1350, ACM, 2008.
- [128] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, and A. Wipat, “Taverna: a tool for the composition and enactment of bioinformatics workflows,” *Bioinformatics*, vol. 20, pp. 3045–3054, Oxford University Press, 2004.
- [129] S. Majithia, M. Shields, I. Taylor, and I. Wang, “Triana: A graphical web service composition and execution toolkit,” in *Proceedings of the 2nd IEEE International Conference on Web Services*, pp. 514–521, IEEE, 2004.
- [130] C. Peltz, “Web services orchestration and choreography,” *IEEE Computer*, vol. 36, no. 10, pp. 46–52, IEEE Computer Society, 2003.
- [131] C. Peltz, “Web services orchestration. a review of emerging technologies, tools and standards,” *White Paper*, pp. 5–37, Hewlett Packard, 2003.
- [132] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*. Pearson Education, 2005.
- [133] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers, “Examining the challenges of scientific workflows,” *IEEE computer*, vol. 40, no. 12, pp. 26–34, IEEE, 2007.
- [134] Y. Zhao, I. Raicu, and I. Foster, “Scientific workflow systems for 21st century, new bottle or new wine?,” in *Proceedings of the IEEE Workshop on Scientific Workflows*, pp. 467–471, IEEE, 2008.
- [135] B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, and J. Patel, “Sedna: A BPEL-based environment for visual scientific workflow modeling,” in *Workflows for e-Science*, pp. 428–449, Springer, 2007.
- [136] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, and D. Marvin, “Taverna: lessons in creating a workflow environment for the life sciences,” *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1067–1100, Wiley Online Library, 2006.

- [137] I. Taylor, M. Shields, I. Wang, and O. Rana, “Triana applications within Grid computing and peer to peer environments,” *Journal of Grid Computing*, vol. 1, no. 2, pp. 199–217, Springer, 2003.
- [138] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wiecekorek, “ASKALON: A Grid application development and computing environment,” in *Proceedings of the 6th IEEE International Workshop on Grid Computing*, pp. 122–131, IEEE, 2005.
- [139] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, and I. Trickovic, “Business process execution language for Web services,” *Technical report, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems*, 2003.
- [140] A. Goderis, U. Sattler, P. Lord, and C. Goble, “Seven bottlenecks to workflow reuse and repurposing,” in *The 4th International Semantic Web Conference*, pp. 323–337, Springer, 2005.
- [141] R. D. Stevens, A. J. Robinson, and C. A. Goble, “myGrid: personalised bioinformatics on the information Grid,” *Bioinformatics*, vol. 19, pp. 302–304, Oxford University Press, 2003.
- [142] M. Addis, J. Ferris, M. Greenwood, P. Li, D. Marvin, T. Oinn, and A. Wipat, “Experiences with e-Science workflow specification and enactment in bioinformatics,” in *Proceedings of 2nd UK e-Science All Hands Meeting*, pp. 459–466, The Engineering and Physical Sciences Research Council (EPSRC), 2003.
- [143] G. Allen, T. Goodale, T. Radke, M. Russell, E. Seidel, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, J. Shalf, and I. Taylor, “Enabling applications on the Grid: A Gridlab overview,” *International Journal of High Performance Computing Applications*, vol. 17, no. 4, pp. 449–466, SAGE Publications, 2003.
- [144] L. Gong, “Project JXTA: A technology overview,” *Technical report, SUN Microsystems*, 2001.
- [145] I. J. Taylor, O. F. Rana, R. Philp, I. Wang, and M. Shields, “Supporting Peer-2-Peer interactions in the consumer Grid,” in *Proceedings of the International Symposium on Parallel and Distributed Processing*, pp. 3–12, IEEE, 2003.
- [146] I. Taylor, M. Shields, and I. Wang, “Distributed P2P computing within Triana: A galaxy visualization test case,” in *Proceedings of the International Symposium on Parallel and Distributed Processing Symposium*, IEEE, 2003.
- [147] P. Jurczyk, M. Golenia, M. Malawski, D. Kurzyniec, M. Bubak, and V. S. Sunderam, “A system for distributed computing based on H2O and JXTA,” in *Proceedings of the Cracow Grid Workshop*, pp. 99–114, 2004.
- [148] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A framework for simulating and prototyping heterogeneous systems,” *International Journal of Computer Simulation*, vol. 4, pp. 155–182, Ablex Publishing Corporation, 1994.

- [149] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, “Kepler: an extensible system for design and execution of scientific workflows,” in *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pp. 423–424, IEEE, 2004.
- [150] I. Altintas, O. Barney, and E. Jaeger-Frank, “Provenance collection support in the Kepler scientific workflow system,” in *Provenance and annotation of data*, pp. 118–132, Springer, 2006.
- [151] J. Frey, “Condor DAGMan: Handling inter-job dependencies,” *Technical Report, University of Wisconsin*, 2002.
- [152] I. Foster, J. Vockler, M. Wilde, and Y. Zhao, “Chimera: A virtual data system for representing, querying, and automating data derivation,” in *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pp. 37–46, IEEE, 2002.
- [153] M. Wieczorek, R. Prodan, and T. Fahringer, “Scheduling of scientific workflows in the ASKALON Grid environment,” *ACM SIGMOD Record*, vol. 34, no. 3, pp. 56–62, ACM, 2005.
- [154] T. Fahringer, S. Pllana, and A. Villazon, “A-GWL: Abstract Grid Workflow Language,” in *Computational Science*, vol. 3038 of *Lecture Notes in Computer Science*, pp. 42–49, Springer, 2004.
- [155] T. Fahringer, S. Pllana, and J. Testori, “Teuta: Tool support for performance modeling of distributed and parallel applications,” in *Proceedings of the International Conference on Computational Science (ICCS)*, pp. 456–463, Springer, 2004.
- [156] A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse, and J. Darlington, “ICENI dataflow and workflow: Composition and scheduling in space and time,” in *Proceedings of the 2nd UK e-Science All Hands Meeting*, vol. 634, pp. 627–634, The Engineering and Physical Sciences Research Council (EPSRC), 2003.
- [157] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington, “ICENI: Optimisation of component applications within a Grid environment,” *Parallel Computing*, vol. 28, no. 12, pp. 1753–1772, Elsevier, 2002.
- [158] S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington, “Workflow enactment in ICENI,” in *Proceedings of the 3rd UK e-Science All Hands Meeting*, pp. 894–900, The Engineering and Physical Sciences Research Council (EPSRC), 2004.
- [159] L. Young, S. McGough, S. Newhouse, and J. Darlington, “Scheduling architecture and algorithms within the ICENI Grid middleware,” in *Proceedings of the 2nd UK e-Science All Hands Meeting*, pp. 5–12, The Engineering and Physical Sciences Research Council (EPSRC), 2003.
- [160] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, “Swift: Fast, reliable, loosely coupled parallel computation,” in *Proceedings of the IEEE Congress on Services*, pp. 199–206, IEEE, 2007.

- [161] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, “Swift: A language for distributed parallel scripting,” *Parallel Computing*, vol. 37, pp. 633–652, Elsevier, 2011.
- [162] G. von Laszewski and M. Hategan, “Workflow concepts of the Java CoG Kit,” *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 239–258, Springer, 2005.
- [163] W. A. Pinheiro, A. S. Vivacqua, R. Barros, A. S. de Mattos, N. M. Cianni, P. C. Monteiro Jr, R. N. De Martino, V. Marques, G. Xexéo, and J. M. de Souza, “Dynamic workflow management for P2P environments using agents,” in *International Conference on Computational Science (ICCS)*, pp. 253–256, Springer, 2007.
- [164] A. S. Vivacqua, W. A. Pinheiro, R. M. Barros, A. S. de Mattos, N. M. Cianni, P. C. Monteiro, R. N. De Martino, V. Marques, G. Xexéo, J. M. de Souza, and D. Schneider, “Dynaflow: Agent-based dynamic workflow management for P2P environments.,” in *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS)*, pp. 275–278, 2007.
- [165] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd, “GridFlow: Workflow management for Grid computing,” in *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 198–205, IEEE, 2003.
- [166] J. Cao, D. J. Kerbyson, and G. R. Nudd, “Performance evaluation of an agent-based resource management infrastructure for Grid computing,” in *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 311–318, IEEE, 2001.
- [167] J. Cao, D. P. Spooner, J. D. Turner, S. Jarvis, D. J. Kerbyson, S. Saini, and G. R. Nudd, “Agent-based resource management for Grid computing,” in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 350–350, IEEE, 2002.
- [168] J. Cao, S. A. Jarvis, S. Saini, D. J. Kerbyson, and G. R. Nudd, “ARMS: An agent-based resource management system for Grid computing,” *Scientific Programming*, vol. 10, no. 2, pp. 135–148, IOS Press, 2002.
- [169] D. P. Spooner, J. Cao, J. D. Turner, H. Lin Choi Keung, S. A. Jarvis, and G. Nudd, “Localised workload management using performance prediction and QoS contracts,” *University of Glasgow, UK*, 2002.
- [170] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox, “PACE - a toolset for the performance prediction of parallel and distributed systems,” *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 228–251, SAGE Publications, 2000.
- [171] E. Lusk, S. Huss, B. Saphir, and M. Snir, “MPI: A message-passing interface standard,” 2009.
- [172] J. Yu and R. Buyya, “A taxonomy of workflow management systems for Grid computing,” *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 171–200, Springer, 2005.

- [173] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, “Evaluation of job-scheduling strategies for Grid computing,” in *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing*, pp. 191–202, Springer, 2000.
- [174] F. Leymann, “Web Services Flow Language (WSFL 1.0),” *IBM Corp.*, 2001.
- [175] S. Thatte, “XLANG: Web services for business process design,” *Microsoft Corp.*, 2001.
- [176] S. Krishnan, P. Wagstrom, and G. Von Laszewski, “GSFL: A workflow framework for Grid services,” *Technical report*, 2002.
- [177] D. Cybok, “A Grid workflow infrastructure,” *Concurrency and Computation: Practice and Experience*, vol. 18, pp. 1243–1254, Wiley Online Library, 2006.
- [178] C. Ching Lian, F. Tang, P. Issac, and A. Krishnan, “GEL: Grid execution language,” *Journal of Parallel and Distributed Computing*, vol. 65, pp. 857–869, Elsevier, 2005.
- [179] H. P. Bivens and J. I. Beiriger, “GALE: Grid access language for HPC environments,” 2002.
- [180] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacs-Nagy, I. Trickovic, and S. Zimek, “Web Service Choreography Interface (WSCI) 1.0,” *Standards proposal by BEA Systems, Intalio, SAP, and Sun Microsystems*, 2002.
- [181] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto, “Web Services Choreography Description Language version 1.0,” *World Wide Web Consortium (W3C) Candidate Recommendation*, 2002.
- [182] W. M. Van der Aalst and A. H. ter Hofstede, “YAWL: Yet Another Workflow Language,” *Information Systems*, vol. 30, pp. 245–275, Elsevier, 2005.
- [183] M. G. Nanda, S. Chandra, and V. Sarkar, “Decentralizing execution of composite Web services,” in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp. 170–187, ACM, 2004.
- [184] L. Baresi, A. Maurino, and S. Modafferri, “Towards distributed BPEL orchestrations,” *Electronic Communications of the European Association of Software Science and Technology (EASST)*, vol. 3, 2007.
- [185] U. Yildiz and C. Godart, “Towards decentralized service orchestrations,” in *Proceedings of the 22nd ACM Symposium on Applied Computing*, pp. 1662–1666, ACM, 2007.
- [186] G. Decker, O. Kopp, F. Leymann, and M. Weske, “BPEL4Chor: Extending BPEL for modeling choreographies,” in *Proceedings of the 5th International Conference on Web Services*, pp. 296–303, IEEE, 2007.
- [187] Y. Huang and D. W. Walker, “Extensions to Web service techniques for integrating Jini into a service-oriented architecture for the Grid,” in *Computational Science*, pp. 254–263, Springer, 2003.

- [188] C. Chua, F. Tang, Y. Lim, L. Ho, and A. Krishnan, “Implementing a bioinformatics workflow in a parallel and distributed environment,” in *Parallel and Distributed Computing: Applications and Technologies*, pp. 1–4, Springer, 2005.
- [189] N. Russell and A. H. ter Hofstede, “newYAWL: Towards workflow 2.0,” in *Transactions on Petri Nets and Other Models of Concurrency II*, pp. 79–97, Springer, 2009.
- [190] J. L. Peterson, “Petri nets,” *ACM Computing Surveys (CSUR)*, vol. 9, pp. 223–252, ACM, 1977.
- [191] W. M. P. van der Aalst, K. Van Hee, and G. Houben, “Modelling and analysing workflow using a Petri-net based approach,” in *Proceedings of the 2nd Workshop on Computer-Supported Cooperative Work, Petri nets and Related Formalisms*, pp. 31–50, 1994.
- [192] A. Hoheisel, “User tools and languages for graph-based Grid workflows,” *Concurrency and Computation: Practice and Experience*, vol. 18, pp. 1101–1113, Wiley Online Library, 2006.
- [193] G. Booch, J. Rumbaugh, and I. Jacobson, “Unified Modeling Language (UML),” *Rational Software Corporation*, 1998.
- [194] R. Eshuis and R. Wieringa, “Comparing Petri net and activity diagram variants for workflow modelling - a quest for reactive Petri nets,” in *Petri net Technology for Communication-Based Systems*, pp. 321–351, Springer, 2003.
- [195] S. Plana, T. Fahringer, J. Testori, S. Benkner, and I. Brandic, “Towards an UML based graphical representation of Grid workflow applications,” in *Grid Computing*, pp. 149–158, Springer, 2004.
- [196] M. Dumas and A. H. ter Hofstede, “UML activity diagrams as a workflow specification language,” in *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pp. 76–90, Springer, 2001.
- [197] R. M. Bastos and D. D. A. Ruiz, “Extending UML activity diagram for workflow modeling in production systems,” in *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, pp. 3786–3795, IEEE, 2002.
- [198] V. Curcin and M. Ghanem, “Scientific workflow systems - can one size fit all?,” in *Proceedings of the Cairo International Biomedical Engineering Conference*, pp. 1–9, IEEE, 2008.
- [199] G. Decker, H. Overdick, and J. M. Zaha, “On the suitability of WS-CDL for choreography modeling,” in *Proceedings of EMISA*, vol. 95, pp. 21–33, 2006.
- [200] L. Fredlund, “Implementing WS-CDL,” in *Proceedings of the 2nd Spanish Workshop on Web Technologies*, 2006.
- [201] Z. Kang, H. Wang, and P. C. Hung, “WS-CDL+: an extended WS-CDL execution engine for Web service collaboration,” in *Proceedings of the IEEE 5th International Conference on Web Services*, pp. 928–935, IEEE, 2007.

- [202] S. Ross-Talbot, G. Brown, K. Honda, N. Yoshida, and M. Carbone, “Pi4 technologies foundation,” <http://pi4soa.sourceforge.net>.
- [203] J. Mendling and M. Hafner, “From WS-CDL choreography to BPEL process orchestration,” *Journal of Enterprise Information Management*, vol. 21, pp. 525–542, Emerald Group Publishing Limited, 2008.
- [204] J. Mendling and M. Hafner, “From inter-organizational workflows to process execution: Generating BPEL from WS-CDL,” in *Proceedings of the OTM 2005 Workshop*, pp. 506–515, Springer, 2005.
- [205] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot, “A theoretical basis of communication-centred concurrent programming,” *Web Services Choreography Working Group Report*, 2006.
- [206] M. Carbone, K. Honda, and N. Yoshida, “Structured communication-centred programming for Web services,” in *Programming Languages and Systems*, pp. 2–17, Springer, 2007.
- [207] Y. Hongli, Z. Xiangpeng, Q. Zongyan, P. Geguang, and W. Shuling, “A formal model for Web Service Choreography Description Language (WS-CDL),” *School of Mathematical Science. Peking University*, 2006.
- [208] J. M. Zaha, A. Barros, M. Dumas, and A. ter Hofstede, “Let’s Dance: A language for service behavior modeling,” in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pp. 145–162, Springer, 2006.
- [209] G. Decker, M. Kirov, J. M. Zaha, and M. Dumas, “Maestro for Let’s Dance: An environment for modeling service interactions,” *Technical Report, Queensland University of Technology*, 2006.
- [210] W. M. Van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek, “Conformance checking of service behavior,” *ACM Transactions on Internet Technology*, vol. 8, pp. 1–30, ACM, 2008.
- [211] J. M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, and G. Decker, “Service interaction modeling: Bridging global and local views,” in *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*, pp. 45–55, IEEE, 2006.
- [212] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 59–72, ACM, 2007.
- [213] A. Barker, J. B. Weissman, and J. I. van Hemert, “Reducing data transfer in service-oriented architectures: The circulate approach,” *IEEE Transactions on Services Computing*, vol. 5, no. 3, pp. 437–449.
- [214] W. Binder, I. Constantinescu, and B. Faltings, “Service invocation triggers: a lightweight routing infrastructure for decentralised workflow orchestration,” *International Journal of High Performance Computing and Networking*, vol. 6, pp. 81–90, 2009.

- [215] D. Liu, K. H. Law, and G. Wiederhold, “Data-flow distribution in FICAS service composition infrastructure,” in *Proceedings of the 15th International Conference on Parallel and Distributed Computing Systems*, 2002.
- [216] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, “DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language.,” in *Proceedings of the Operating System Design and Implementation (OSDI) Symposium*, vol. 8, pp. 1–14, 2008.
- [217] J. Balasooriya, M. Padhye, S. K. Prasad, and S. B. Navathe, “BondFlow: A system for distributed coordination of workflows over Web services,” in *Proceedings of the 19th IEEE International Symposium on Parallel and Distributed Processing*, IEEE, 2005.
- [218] S. Jang, X. Wu, V. Taylor, G. Mehta, K. Vahi, and E. Deelman, “Using performance prediction to allocate Grid resources,” *GriPhyN Technical Report*, vol. 25, Texas A&M University, 2004.
- [219] W. Smith, I. Foster, and V. Taylor, “Predicting application run times using historical information,” in *Proceedings of Job Scheduling Strategies for Parallel Processing*, pp. 122–142, Springer, 1998.
- [220] H. Topcuoglu and S. Hariri, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274, IEEE, 2002.
- [221] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy, “Task scheduling strategies for workflow-based applications in Grids,” in *IEEE International Symposium on Cluster Computing and the Grid*, vol. 2, pp. 759–767, IEEE, 2005.
- [222] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, “A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems,” *Journal of Parallel and Distributed computing*, vol. 61, no. 6, pp. 810–837, Elsevier, 2001.
- [223] R. Duan, R. Prodan, and T. Fahringer, “Run-time optimisation of Grid workflow applications,” in *7th IEEE/ACM International Conference on Grid Computing*, pp. 33–40, IEEE, 2006.
- [224] R. Sakellariou and H. Zhao, “A hybrid heuristic for DAG scheduling on heterogeneous systems,” in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, pp. 111–121, IEEE, 2004.
- [225] S. Kumar, S. K. Das, and R. Biswas, “Graph partitioning for parallel applications in heterogeneous Grid environments,” in *Proceedings of the International Parallel and Distributed Processing Symposium*, pp. 167–174, IEEE, 2002.
- [226] M. Wieczorek, A. Hoheisel, and R. Prodan, “Towards a general model of the multi-criteria workflow scheduling on the Grid,” *Future Generation Computer Systems*, vol. 25, no. 3, pp. 237–256, Elsevier, 2009.

- [227] J. Chen and Y. Yang, "A taxonomy of Grid workflow verification and validation," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 4, pp. 347–360, Wiley Online Library, 2008.
- [228] Y. Han, A. Sheth, and C. Bussler, "A taxonomy of adaptive workflow management," in *Workshop of the 1998 ACM Conference on Computer Supported Cooperative Work*, 1998.
- [229] F. Wu, Q. Wu, and Y. Tan, "Workflow scheduling in Cloud: a survey," *The Journal of Supercomputing*, pp. 1–46, Springer, 2015.
- [230] L. Ramakrishnan and D. Gannon, "A survey of distributed workflow characteristics and resource requirements," *Indiana University*, 2008.
- [231] P. Maechling, E. Deelman, L. Zhao, R. Graves, G. Mehta, N. Gupta, J. Mehringer, C. Kesselman, S. Callaghan, and D. Okaya, "SCEC Cybershake workflows - automating probabilistic seismic hazard analysis calculations," in *Workflows for e-Science*, pp. 143–163, Springer, 2007.
- [232] M. Sonntag, K. Görlach, D. Karastoyanova, F. Leymann, P. Malets, and D. Schumm, "Views on scientific workflows," in *Perspectives in Business Informatics Research*, pp. 321–335, Springer, 2011.
- [233] J. Eder and W. Liebhart, *Contributions to exception handling in workflow management*. 1998.
- [234] S. Hwang and C. Kesselman, "Grid workflow: a flexible failure handling framework for the Grid," in *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, pp. 126–137, IEEE, 2003.
- [235] J. Magee and J. Kramer, "Dynamic structure in software architectures," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6, pp. 3–14, ACM, 1996.
- [236] I. Oueichek and X. R. De Pina, "Dynamic configuration management in the guide object-oriented distributed system," in *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pp. 28–35, IEEE, 1996.
- [237] S. K. Srivastava and S. M. Wheater, "Architectural support for dynamic reconfiguration of large scale distributed applications," in *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pp. 10–17, IEEE, 1998.
- [238] L. Bellissard, S. B. Atallah, F. Boyer, and M. Riveill, "Distributed application configuration," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, pp. 579–585, IEEE, 1996.
- [239] S. Baek, J. Han, and K. Chung, "Dynamic reconfiguration based on goal-scenario by adaptation strategy," *Wireless Personal Communications*, vol. 73, no. 2, pp. 309–318, Springer, 2013.
- [240] S. W. Sadiq, O. Marjanovic, and M. E. Orlowska, "Managing change and time in dynamic workflow processes," *International Journal of Cooperative Information Systems*, vol. 9, pp. 93–116, 2000.

- [241] G. Mateescu, “Quality of service on the Grid via metascheduling with resource co-scheduling and co-reservation,” *International Journal of High Performance Computing Applications*, vol. 17, no. 3, pp. 209–218, 2003.
- [242] G. B. Chafle, S. Chandra, V. Mann, and M. G. Nanda, “Decentralized orchestration of composite Web services,” in *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*, pp. 134–143, ACM, 2004.
- [243] D. Gelernter and N. Carriero, “Coordination languages and their significance,” *Communications of the ACM*, vol. 35, no. 2, pp. 96–107, ACM, 1992.
- [244] W. Ackerman, “Data flow languages,” *Computer*, vol. 15, no. 2, pp. 15–25, 1982.
- [245] R. M. Karp and R. E. Miller, “Properties of a model for parallel computations: Determinacy, termination, queueing,” *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, Society for Industrial and Applied Mathematics (SIAM), 1966.
- [246] P. Wegner, *Dimensions of object-based language design*, vol. 22. ACM, 1987.
- [247] L. Cardelli, *Typeful programming*. Springer, 1991.
- [248] N. Wirth and C. A. R. Hoare, “A contribution to the development of ALGOL,” *Communications of the ACM*, vol. 9, no. 6, pp. 413–432, ACM, 1966.
- [249] N. Wirth, “On the design of programming languages,” in *Proceedings of the IFIP Congress*, vol. 74, pp. 386–393, 1974.
- [250] M. Priestley, *A science of operations: machines, logic and the invention of programming*. Springer Science & Business Media, 2011.
- [251] R. Morrison, *On the development of ALGOL*. PhD thesis, University of St Andrews, UK, 1979.
- [252] R. D. Tennent, “Language design methods based on semantic principles,” *Acta Informatica*, vol. 8, no. 2, pp. 97–112, 1977.
- [253] P. J. Landin, “The next 700 programming languages,” *Communications of the ACM*, vol. 9, no. 3, pp. 157–166, 1966.
- [254] A. Demers and J. Donahue, “Type-completeness as a language principle,” in *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 234–244, ACM, 1980.
- [255] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison, “An approach to persistent programming,” *The Computer Journal*, vol. 26, no. 4, pp. 360–365, 1983.
- [256] J. B. Dennis, “First version of a data flow procedure language,” in *Programming Symposium*, pp. 362–376, Springer, 1974.
- [257] J. B. Dennis, “Data flow supercomputers,” *Computer*, vol. 13, no. 11, pp. 48–56, IEEE, 1980.

- [258] K. Gilles, “The semantics of a simple language for parallel programming,” in *Information Processing'74: Proceedings of the IFIP Congress*, vol. 74, pp. 471–475, 1974.
- [259] W. M. Johnston, J. Hanna, and R. J. Millar, “Advances in dataflow programming languages,” *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, ACM, 2004.
- [260] F. E. Allen and J. Cocke, “A program data flow analysis procedure,” *Communications of the ACM*, vol. 19, no. 3, pp. 137–147, ACM, 1976.
- [261] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, “Dependence graphs and compiler optimizations,” in *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 207–218, ACM, 1981.
- [262] P. G. Whiting and R. S. Pascoe, “A history of data-flow languages,” *Annals of the History of Computing*, vol. 16, no. 4, pp. 38–59, IEEE, 1994.
- [263] J. Cardoso and A. Sheth, “Semantic e-workflow composition,” *Journal of Intelligent Information Systems*, vol. 21, no. 3, pp. 191–225, Springer, 2003.
- [264] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, “Semantic matching of Web services capabilities,” in *The International Semantic Web Conference*, pp. 333–347, Springer, 2002.
- [265] G. Juve, A. L. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, “Characterizing and profiling scientific workflows,” *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.
- [266] W. Jaradat, A. Dearle, and A. Barker, “A dataflow language for decentralised orchestration of Web service workflows,” in *Proceedings of the 9th IEEE World Congress on Services*, pp. 13–20, IEEE, 2013.
- [267] W. Jaradat, A. Dearle, and A. Barker, “Workflow partitioning and deployment on the Cloud using Orchestra,” in *Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing*, pp. 251–260, IEEE, 2014.
- [268] W. Jaradat, A. Dearle, and A. Barker, “Towards an autonomous decentralized orchestration system,” *Concurrency and Computation: Practice and Experience*, 2015.
- [269] W. Jaradat, A. Dearle, and A. Barker, “An architecture for decentralised orchestration of Web service workflows,” in *Proceedings of the IEEE 20th International Conference on Web Services*, pp. 603–604, IEEE, 2013.
- [270] T. Bray, “The JavaScript Object Notation (JSON) data interchange format,” 2014.
- [271] J. Davie and R. Morrison, *Recursive descent compiling*. John Wiley & Sons, 1982.
- [272] R. Rosen, “Internet Control Message Protocol (ICMP),” in *Linux Kernel Networking*, pp. 37–61, Springer, 2014.
- [273] J. Strauss, D. Katabi, and F. Kaashoek, “A measurement study of available bandwidth estimation tools,” in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, pp. 39–44, ACM, 2003.

- [274] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy, “Bandwidth estimation: metrics, measurement techniques, and tools,” *IEEE Network*, vol. 17, no. 6, pp. 27–35, IEEE, 2003.
- [275] J. Postel, “RFC 768,” *User Datagram Protocol*, pp. 1–3, 1980.
- [276] V. G. Cerf and R. E. Icahn, “A protocol for packet network intercommunication,” *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 2, pp. 71–82, 2005.
- [277] S. Spero, “Analysis of HTTP performance problems,” 1994.
- [278] R. Schutt and C. O’Neil, *Doing data science: Straight talk from the frontline*. O'Reilly Media, Inc., 2013.
- [279] D. Arthur and S. Vassilvitskii, “k-means++: The advantages of careful seeding,” in *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1027–1035, Society for Industrial and Applied Mathematics, 2007.
- [280] A. Barker, J. B. Weissman, and J. I. van Hemert, “The circulate architecture: avoiding workflow bottlenecks caused by centralised orchestration,” *Cluster computing*, vol. 12, no. 2, pp. 221–235, Springer, 2009.
- [281] P. Besana, V. Patkar, A. Barker, D. Robertson, and D. Glasspool, “Sharing choreographies in OpenKnowledge: A novel approach to interoperability,” *Journal of Software*, vol. 4, no. 8, pp. 833–842, 2009.
- [282] A. Barker, P. Besana, D. Robertson, and J. B. Weissman, “The benefits of service choreography for data-intensive computing,” in *Proceedings of the 7th International Workshop on Challenges of Large Applications in Distributed Environments*, pp. 1–10, ACM, 2009.
- [283] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle, “Dynamic virtual clusters in a Grid site manager,” in *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, pp. 90–100, IEEE, 2003.
- [284] R. J. Figueiredo, P. A. Dinda, and J. A. Fortes, “A case for Grid computing on virtual machines,” in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pp. 550–559, IEEE, 2003.
- [285] W. Huang, J. Liu, B. Abali, and D. K. Panda, “A case for high performance computing with virtual machines,” in *Proceedings of the 20th Annual International Conference on Supercomputing*, pp. 125–134, ACM, 2006.
- [286] K. Keahey, I. Foster, T. Freeman, and X. Zhang, “Virtual workspaces: Achieving quality of service and quality of life in the Grid,” *Scientific Programming*, vol. 13, no. 4, pp. 265–275, 2005.
- [287] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, “The cost of doing science on the Cloud: the Montage example,” in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, p. 50, IEEE, 2008.

- [288] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Beriman, and J. Good, “On the use of Cloud computing for scientific workflows,” in *Proceedings of the 4th IEEE International Conference on eScience*, pp. 640–645, IEEE, 2008.
- [289] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, “A performance analysis of EC2 Cloud computing services for scientific computing,” in *Cloud Computing*, pp. 115–131, Springer, 2010.
- [290] I. M. Llorente, R. Moreno-Vozmediano, and R. S. Montero, “Cloud computing for on-demand Grid resource provisioning,” *Advances in Parallel Computing*, vol. 18, no. 5, pp. 177–191, 2009.
- [291] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, “Virtual infrastructure management in private and hybrid Clouds,” *IEEE Internet Computing*, vol. 13, no. 5, pp. 14–22, 2009.
- [292] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, “The Eucalyptus open-source Cloud-computing system,” in *Proceedings of the 9th IEEE/ACM International Symposium Cluster Computing and the Grid*, pp. 124–131, IEEE, 2009.
- [293] I. Sriram and A. Khajeh-Hosseini, “Research agenda in Cloud technologies,” *arXiv preprint arXiv:1001.3259*, 2010.
- [294] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, and M. Tsugawa, “Science Clouds: Early experiences in Cloud computing for scientific applications,” *Cloud Computing and Applications*, vol. 2008, pp. 825–830, 2008.
- [295] J. B. Weissman, P. Sundarajan, A. Gupta, M. Ryden, R. Nair, and A. Chandra, “Early experience with the distributed Nebula Cloud,” in *Proceedings of the 4th International Workshop on Data-intensive Distributed Computing*, pp. 17–26, ACM, 2011.
- [296] J. T. Dudley and A. J. Butte, “In silico research in the era of Cloud computing,” *Nature Biotechnology*, vol. 28, no. 11, pp. 1181–1185, Nature Publishing Group, 2010.
- [297] L. D. Stein, “The case for Cloud computing in genome informatics,” *Genome Biology*, vol. 11, no. 5, p. 207, 2010.
- [298] M. C. Schatz, “CloudBurst: highly sensitive read mapping with MapReduce,” *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, Oxford University Press, 2009.
- [299] R. D. Peng, “Reproducible research in computational science,” *Science*, vol. 334, no. 6060, p. 1226, 2011.
- [300] Z. Ganon and I. E. Zilbershtein, “Cloud-based performance testing of network management systems,” in *Proceedings of the IEEE 14th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks*, pp. 1–6, IEEE, 2009.

- [301] L. M. Riungu, O. Taipale, and K. Smolander, “Research issues for software testing in the Cloud,” in *Proceedings of the IEEE 2nd International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 557–564, IEEE, 2010.
- [302] L. Riungu-Kalliosaari, O. Taipale, and K. Smolander, “Testing in the Cloud: Exploring the practice,” *IEEE Software*, vol. 29, no. 2, pp. 46–51, 2012.
- [303] K. Incki, I. Ari, and H. Sözer, “A survey of software testing in the Cloud,” in *Proceedings of the IEEE 6th International Conference on Software Security and Reliability Companion (SERE-C)*, pp. 18–23, IEEE, 2012.
- [304] J. Gao, X. Bai, and W.-T. Tsai, “Cloud testing - issues, challenges, needs and practice,” *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 9–23, 2011.
- [305] J. S. Gwertzman and M. Seltzer, “The case for geographical push-caching,” in *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pp. 51–55, IEEE, 1995.
- [306] V. N. Padmanabhan and L. Subramanian, “An investigation of geographic mapping techniques for Internet hosts,” in *ACM SIGCOMM Computer Communication Review*, vol. 31, pp. 173–185, ACM, 2001.
- [307] T. E. Ng and H. Zhang, “Predicting Internet network distance with coordinates-based approaches,” in *Proceedings of the IEEE 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, pp. 170–179, IEEE, 2002.
- [308] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz, and Y. Jin, “An architecture for a global Internet host distance estimation service,” in *Proceedings of the IEEE 18th Annual Joint Conference of the Computer and Communications Societies*, vol. 1, pp. 210–217, IEEE, 1999.
- [309] S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang, “On the placement of Internet instrumentation,” in *Proceedings of the IEEE 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, pp. 295–304, IEEE, 2000.
- [310] W. Theilmann and K. Rothermel, “Dynamic distance maps of the Internet,” in *Proceedings of the IEEE 19th Annual Joint Conference of the Computer and Communications Societies*, vol. 1, pp. 275–284, IEEE, 2000.
- [311] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek, “Selection algorithms for replicated Web servers,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 3, pp. 44–50, 1998.
- [312] M. E. Crovella and R. L. Carter, “Dynamic server selection in the Internet,” in *Proceedings of the 3rd IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS)*, 1995.
- [313] R. L. Carter and M. E. Crovella, “Measuring bottleneck link speed in packet-switched networks,” *Performance Evaluation*, vol. 27, pp. 297–318, Elsevier, 1996.

- [314] R. L. Carter and M. E. Crovella, “On the network impact of dynamic server selection,” *Computer Networks*, vol. 31, no. 23, pp. 2529–2558, Elsevier, 1999.
- [315] J. Dejun, G. Pierre, and C. Chi, “EC2 performance analysis for resource provisioning of service-oriented applications,” in *Service-Oriented Computing. ICSOC/Service-Wave 2009 Workshops*, pp. 197–207, Springer, 2010.
- [316] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, “Performance analysis of Cloud computing services for many-tasks scientific computing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 931–945, 2011.
- [317] K. Lee, R. Sakellariou, N. W. Paton, and A. A. Fernandes, “Workflow adaptation as an autonomic computing problem,” in *Proceedings of the 2nd Workshop on Workflows in Support of Large-scale Science*, pp. 29–34, ACM, 2007.
- [318] F. Casati, S. Ceri, B. Pernici, and G. Pozzi, “Workflow evolution,” *Data & Knowledge Engineering*, vol. 24, no. 3, pp. 211–238, Elsevier, 1998.
- [319] P. Mangan and S. Sadiq, “On building workflow models for flexible processes,” in *Australian Computer Science Communications*, vol. 24, pp. 103–109, Australian Computer Society, 2002.
- [320] P. J. Mangan and S. Sadiq, “A constraint specification approach to building flexible workflows,” *Journal of Research and Practice in Information Technology*, vol. 35, no. 1, pp. 21–39, 2003.
- [321] P. Dourish, J. Holmes, A. MacLean, P. Marqvardsen, and A. Zbyslaw, “Freeflow: mediating between representation and action in workflow systems,” in *Proceedings of the 1996 ACM conference on Computer Supported Cooperative Work*, pp. 190–198, ACM, 1996.