University of Bath

PHD

Resource-Oriented Architecture based Scientific Workflow Modelling

Duan, Kewei

Award date:
2016

Awarding institution:
University of Bath

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 22. May. 2019

Resource-Oriented Architecture based Scientific Workflow Modelling

submitted by

Kewei Duan

for the degree of Doctor of Philosophy

of the

University of Bath

Department of Computer Science

11/07/2016

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with its author. This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signature of Author . . . .

Kewei Duan

Abstract

This thesis studies the feasibility and methodology of applying state-of-the-art computer technology in scientific workflow modelling, within a collaborative environment. The collaborative environment also indicates that the people involved include non-computer scientists or engineers from other disciplines. The objective of this research is to provide a systematic methodology based on a web environment for the purpose of lowering the barriers brought by the heterogeneous features of multi-institutions, multi-platforms and geographically distributed resources which are implied in the collaborative environment of scientific workflow.

The central argument of this thesis is that *applying novel design based on Resource-Oriented Architecture (ROA) is able to enhance the efficiency of applications from the perspective of data staging, service deployment and workflow enactment of scientific workflow modelling in the context of a cooperative environment*. Based on this generic solution, the thesis aims to provide approaches and evidence to support the central argument from two perspectives: (i) from the application perspective, as a user of scientific workflow, this thesis aims to validate the feasibility and discover the advantages of applying a new web-based services workflow framework. (ii) from the development perspective, as a computer scientist, this thesis aims to fill the gap between new web services technology and scientific workflow modelling by developing a new framework;

For the depiction and analysis side, this thesis reviews the relevant literature of scientific workflow systems and MDO frameworks and provides the background information about the web services technology and ROA that have been applied within this work. It further reviews the existing solutions on web services composition as well as some relevant solutions about workflow, which also includes the reviews of the related works on declarative languages for REST web service composition and speculative enactment of services. With the exploration on the literature gap of a sys-

tematic ROA based scientific workflow modelling framework, this thesis introduces a new architectural style by showing the potential to overcome the disadvantages inherited from RPC-style. It also explores and researches the advantages that the new architectural style can bring. The reviews on the evolution of MDO frameworks represents the demand from scientists that smart methodology should be adopted in the scientific workflow that bears the features of multi-institutions, multi-platforms and geographically distributed resources.

For the design and implementation side, this thesis is driven by the motivation to enhance the performance of scientific workflow modelling and present the feasibility of implementing ROA and RESTful web services in scientific workflows. Throughout the thesis, several approaches are introduced to present the contributions from both development and application perspectives. They cover *Data Management and Virtualization, Service Management and Virtualization and Service Composition*. The whole scientific workflow modelling framework is implemented based on *the ROA Based Distributed Data-flow Model, the Cloud Based Online Web Services Management System, the Declarative Services Composition Description and the Speculative Enactment Engine*. Related evaluations are included based on user experience experiments, model simulations and case studies.

Based on this research and corresponding implementation, this framework is able to achieve the four following points:

- It proposes a data staging mechanism for ROA based distributed system that utilises the features and follows the design constraints of RESTful web service without introducing an extra layer of protocol. This mechanism can provide better data staging performance by its distributed and asynchronous features.

- It proposes a SaaS based mechanism for REST web services deployment and management. It simplifies the design and architecture based on ROA. Thus, it can be integrated with existing WMSs simply through the HTTP/HTTPS protocol. Additionally, it simplifies the process for non-specialist users to turn legacy code and programs into web services.

- It proposes a scientific workflow modelling framework based on virtualization technology. The implementation of this framework can enhance the efficiency of computing resources utilisation and increase the reproducibility of the computing environment for heterogenous scientific applications.

- It proposes a speculative services description to describe composite web services. Based on this description, an speculative enactment runtime is proposed as well to execute the composite web services. This speculative enactment runtime enables failure handling of composite web services. Also, these improve the workflow execution performance in the scenario of web service failure.

Acknowledgements

First, I would like to thank my parents, grandparents and all my family members. The journey of seeking truth and knowledge is never easy. Their supports are always my strongest backing in this journey. Also, I would like to give my special thanks to my wife Shirui Chai. As my constant companion, her encouragement gives me the force to go forward.

I also would like to thank my supervisors Dr. Julian Padget and Dr. Alicia Kim , who play the most important roles in my journey of seeking truth and knowledge as a PhD student. They give me great help and show me the directions.

Thanks to my internal examiner Dr Marina De Vos and external examiner Prof Maozhen Li. Thanks for the time they spend, their patience and the helpful advices from them.

Bath is a beautiful city. My lovely friends made my life in Bath more colorful. Thank them.

I also spent half an year for an internship in NII Tokyo in my PhD study. In this period, Prof. Ken Satoh, who was my supervisor, provided great helps not only to my researches, but also to my life in Tokyo. Thanks to him and also to all the friends I met in that journey.

As the last thank, I would like to thank a special character, Mr Xiaopin Deng. This former overseas student brought back college education in China. He advocated and granted the basic state policies of *Opening To The Outside World* and *Family Planning*. Although, it is hard to say what results will those policies finally bring to us. But without them, I will not be able write such a thesis in English, study in the UK and meet all the people in this journey.

Finally, I want to say something to myself. I write a poem in the form of *Ci* with the particular title (*Cipai*) of *Ru Meng Lin*:

伐得幹劈枝裂
唯剩根盤如鐵
任雪蓋寒襲
心有生生不滅
早鵲，早鵲
鳴喚春生萌蘖

Publications

Chapter 3, Chapter 6:

1. Duan, Kewei, Julian Padget, H. Alicia Kim, and Hiroshi Hosobe. ”Composition of engineering web services with universal distributed data-flows framework based on ROA.” In *Proceedings of the Third International Workshop on RESTful Design*, pp. 41-48. ACM, 2012.

Chapter 4, Chapter 6:

1. Duan, Kewei, YE Vincent Seowy, H. Alicia Kim, and Julian Padget. ”A Resource-Oriented Architecture for MDO Framework.” In *Proceedings of 8th AIAA Multidisciplinary Design Optimization Specialist Conference*. 2012.
2. Duan, Kewei, Julian Padget, and H. Alicia Kim. ”A Light-Weight Framework for Bridge-Building from Desktop to Cloud.” In *Service-Oriented ComputingICSOC 2013 Workshops*, pp. 308-323. Springer International Publishing, 2014.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Nowadays, in-silico experimentation plays an important role in scientific research to overcome physical limitation. With the growing of model complexity and development of cyberinfrastructure, more computing resources, both hardware and software from heterogenous contexts are involved in a series of computational and data manipulation steps for the experiments. The concept of scientific workflow is introduced to represent this series of steps as *useful paradigm to describe, manage, and share complex scientific analyses* Singh and Vouk [1996]. The user and creator of scientific workflows are usually not computer and software specialists. A general trend can be observed that the cyberinfrastructure for scientific workflow is behind the development of the new computer technology and the methodology/approaches based on the technology. The advantages brought by technology progress cannot be reflected in scientific researches, which heavily depends on in-silico experimentation. The forward-looking investigation and research is necessary for the emergence of new infrastructure with better related features, such as performance, reusability, accessibility and so on. Thus, the questions of what out-dated cyberinfrastructure parts influence those features, what and how technology and methodology can be adopted to resolve the issues need to be answered.

This thesis studies the feasibility and methodology of applying state-of-the-art computer technology in scientific workflow modelling, within a collaborative research environment. The object of study is workflow, which is a sequence of connected activities or operations that abstract and model real-world working as an activity. It has emerged as a paradigm for representing and managing complex distributed scientific resources. The collaborative research environment is intended to indicate that the people involved include non-computer scientists or engineers from other disciplines. How to enhance the efficiency of both the applications and users in such an environment drives the general direction in this research. Nowadays, large-scale computations and data analysis characterize much scientific research, meaning there is a need for new approaches to the computation and management of distributed resources in what is also a broader execution environment, namely the Internet. Given this background, the computational resources, such as servers and specialist software, can be established and maintained by multiple organizations at different institutions. The computational resources can be developed by scientists or engineers with different knowledge backgrounds on multiple development platforms. Therefore, the objective of this research is to provide a systematic methodology based on a web environment for the purpose of lowering the barriers brought by the heterogeneous features of multi-institution, multi-platform and geographically distributed resources which are implied in the collaborative environment of scientific workflow.

This thesis will mainly study scientific workflow enactment under web environment from several aspects: data transfer, service deployment, data and service virtualization and service composition. For this purpose, Resource-Oriented Architecture (ROA) is introduced to serve as the firm basis of the proposed scientific workflow modelling framework. A series of state-of-art technologies, such as REST web services, containerization and speculative computing are adapted in this ROA based framework to form the new feasible methodologies and approaches for scientific workflow enactment. The central argument of this thesis is that, *through the application of novel designs based on ROA it is possible to enhance the efficiency of applications and users in the context of a collaborative research environment from the perspective of data staging, service deployment and workflow enactment of scientific workflow modelling*. All the approaches will be proposed and examined from two perspectives: application perspective and user perspective. The goal is that the proposed approaches should not only enhance the performance of scientific workflow enactment, but also it will reduce the difficulty of utilising the adopted technologies for user. Thus, (i) from application perspective, the ROA based framework is proposed to fill the gap between new technology and scientific workflow modelling. This framework serves as the foundation to enhance the efficiency of applications. (ii) from user perspective, this thesis will validate the feasibility and discover the advantages of adopting new technologies in this framework. This framework aims to help users to enhance their efficiency in the collaborative research environment.

From developer's perspective, the efficiency will be enhanced by filling the gap between a new web design pattern (ROA) and scientific workflow modelling by developing new framework. The feasibility will be studied and reflected from two aspects: (i) A series of approaches designed and delivered based on a distributed dataflow architecture and an online service deployment system to enhance the data staging performance, simplifying the service deployment and management process, and (ii) A set of methods can be achieved to build a scientific workflow based on services composition, aiming to enhance the workflow enactment efficiency.

From user's perspective, the efficiency will be enhanced to resolve the issues in collaborative research environment by applying new web service technology in a workflow framework. The feasibility will be studied and reflected from two aspects: (i) a framework that enables the integration of a variety of existing applications and codes based on a generic solution, which in terms of intention also enables and enhances the performance of transfers, storage, sharing of geographically distributed scientific data, and (ii) a framework that can simplify the manipulation of web services deployment so that the barrier for non-specialist users can be lower for the purpose of sharing and integration.

One of the important application domains that drives this thesis is Multidisciplinary Design Optimization (MDO). Multidisciplinary design optimization (MDO) is a field of research that studies the application of numerical optimization techniques to the design of engineering systems involving multiple disciplines or components[Martins and Lambe, 2013]. MDO enables designers to incorporate all relevant disciplines in one design process. MDO have been applied in design processes, such as automobile design, naval architecture, electronics, architecture, computers, electricity distribution and so on. In this research field, the largest number of applications have been in the field of aerospace engineering, such as aircraft and spacecraft design which are related to disciplines like aerodynamics, structural analysis, propulsion, control theory, and economics. The reason of selecting MDO as primary problem domain is that MDO embodies all the aforementioned characteristics of a collaborative research environment. They are the heterogeneous features of multi-institution, multi-platform and geographically distributed resources. To support the research on MDO, various MDO frameworks are developed. The benefits brought by the development are defined as to provide the capability to enable interdisciplinary interaction and communication via the use of sophisticated computational procedures combined with state-of-the-art optimization or design improvement techniques[Salas and Townsend, 1998]. In this development trend, it is shown that the integration with new computer technology gradually also improves the efficiency of MDO framework. The improvement allows the framework to enact an automatic process for engineering computation in an more easy-to-use manner. Therefore, this research also follows this trend, and aims to provide a scientific workflow framework that can be used for deployment, sharing and enactment of MDO workflow.

In summary, this thesis will study the new features of a newly proposed web-service based framework for scientific workflow modelling. New technologies, which are popular in cyberinfrastructure nowadays, such as REST web service, containerization, speculative computing as well as new approaches like distributed dataflows, SaaS and PaaS based web service management are adopted. The pros and cons by adopting these technologies in scientific workflow modelling will be discussed and examined based on a series of experiments on workflow enactment performance and user experience. MDO will serves as an representative and important application domain, which embodies some typical application scenarios as the challenges for scientific workflow.

1.1 Motivation

This research has the feature of cross-discipline. On one hand, it requires computer science knowledge to fill the gap between new web services technology and scientific workflow modelling. On the other hand, it needs to investigate the problem domain from the point of view of the user of the scientific workflow modelling framework. Thus, considering the primary problem domain defined previously, the motivation of this research is mainly driven from two perspectives: computer science and mechanical engineering.

1.1.1 A Computer Science Perspective

In recent years, a number of scientific workflow management systems (WMS) have been developed to support the creation of scientific workflow [Ludäscher et al., 2006, Oinn et al., 2004, Taylor et al., 2007, Van Der Aalst and ter Hofstede, 2005]. They all possess the basic functions of workflow compositions, enactment and monitoring.

In these workflow management systems, there are common features that function to access distributed computing resources. The main driving force is that scientific workflows require more computing power given their growing complexity. A key element in the construction of workflows is the web service, which is defined by W3C as "a software system designed to support interoperable machine-to-machine interaction over a network"[Haas and Brown, 2004]. This interoperable machine-to-machine interaction over a network allows resources to be accessed in a geographically distributed cooperative environment.

To enable scientific reproducibility, result publication and sharing, nowadays, scientific workflow is often defined by co-operation and web services technology. However, technically, there exist different web service styles and communication protocols, each with a set of methods built on a specific style and protocol – and being incompatible with each other. For example, arbitrary web services, which are categorized by W3C [W3C, 2004], include a series of web services styles including RPC (Remote Procedure Call), CORBA (Common Object Request Broker Architecture), RMI (Java Remote Method Invocation), SOAP (Simple Object Access Protocol). These are also known as RPC-style web services. They all have different and incompatible communication protocols, which severely limit access to service components created by different systems, and this restriction reduces the overall interoperability between systems. Additionally, new users may have to learn heavyweight specifications in order to use systems effectively, which can increase the cost of introducing of new application tools. Based on this, there is also the risk that non-specialist users are locked inside one particular system because the complexity of each web service specification.

In the meantime, a relatively new class of web services emerged and gained its popularity rapidly, namely REST-compliant web services [W3C, 2004]. The main difference between REST (Representative State Transfer) and the RPC-style web services is the protocol employed between client and server. RPC-style requires each application designer to define a new and arbitrary protocol comprising vocabulary of nouns and verbs that is usually overlaid on the HTTP [Pautasso et al., 2008]. In contrast, RESTful web services work directly with HTTP, so they have one less protocol layer, simplifying and making it easy to deploy and use web services, as well as reducing the overall cost of computation and knowledge preparation. Pautasso and Wilde [2009] conclude that for the objective of designing IT architectures, REST is a better choice in the context of a co-operative environment in that it is intrinsically loosely coupled because of its asynchronous communication nature, universal addressability and uniform interfaces. From the emerging of RESTful web services, many new public web services from large vendors, such as Google, Yahoo and Amazon, are now REST based. REST's simplicity and being lightweight has brought significant changes to business markets and has been broadly accepted.

A question is raised together with the emergence of a new service style, can scientific workflow modelling efficiency be enhanced further, by comparing with the existing service styles, based on the features of new web services technology? The data flow and control flow are the two important aspects of workflow[Russell et al., 2005]. In practice, they represent the steps of date transfer (data staging) from one service to another and a sequence of web services invocations. Also, the composition of these steps finally forms the executable workflow. The efficiency of these practical activities are closely related to the web services styles that are applied and used to build the bricks in the scientific workflow modelling. From a computer science perspective, questions can be asked tangibly from bottom to top through three aspects: How to improve *data staging* performance in the interaction of RESTful web services for a scientific workflow situation? How to provide a more convenient RESTful *service deployment/management* mechanism with higher reusability? How to support RESTful *services composition* for the co-operation of web services in a scientific workflow?

1.1.2 A Mechanical Engineering Perspective

It is a common practise that optimization program for specific discipline is written by the expert on this discipline. For example, optimization program for aerodynamics will be written by an aerodynamics expert, structure optimization program will be written by a structure expert. They may come from different institute or even different country. In a MDO method, these programs have to work together for one overall goal and the methods might be developed by people who work under different organisational standards, adopt different computer platforms and coding techniques. Moreover, all relevant experts and computing resources can be distributed around the world. The features of being multi-institutional, multi-platform and geographically distributed set the backgrounds for designing MDO framework. Various computing technologies have been adopted in the design of MDO frameworks to overcome these barriers. In hindsight, it can be anticipated that more new computer technology can be brought in to further lower the barriers based on the observation of MDO framework research history.

In the history of MDO framework research, the advances of computer hardware and software is correlated with its development. In their review of the history of MDO framework research[Padula and Gillian, 2006], S. L. Padula and R. R. Gillian summarize that in the period 1984-1994, frameworks tended to be presented in monolithic codes written in procedural languages such as FORTRAN and C. It was difficult for researchers to embed their own code into the framework, which was formed by fixed analysis codes. They also mention that in the following decade (1994-2004), a wider variety of framework architectures emerged, taking advantage of hardware and software developments. Web technology is adopted in MDO framework in the course of development.

By considering the features of MDO and the close relationship between the development of MDO frameworks and computer technology, a number of ideas have emerged. For example, the idea of modularity is raised by Padula and Gillian [2006] to isolate machine-dependent parts of code, to improve the readability and to expedite testing. A similar idea is also proposed by Kodiyalam and Sobieszczanski-Sobieski [2001] as a requirement of a MDO framework, which is that the framework should provide a quick and easy linking of analysis tools which serve as slaves to the MDO framework. In an earlier paper, Salas and Townsend [1998] discussing the requirements of MDO framework development, the requirements for architectural design also include the opinions that a framework should be designed using object-oriented principles to allow switching of analysis or optimization methods at run time. In this process, plenty of MDO frameworks proposed various solutions based on different computing technologies to build up a modularized architecture. Based on those frameworks, a tendency emerges for frameworks to evolve from monolithic codes to modularized components, from local programs to online services. Computer technology improvements are closely correlated with the future development directions of MDO framework.

The adoption of web service technology can further support the concept of modularization in the form of web service and parallel processing among services. In the realm of computer science research, for the purpose of supporting interpretable machine-to-machine interaction mechanism through the Internet, web services have been an active research and standardization topic for several years now, reinforced by the emergence of the concept of service-oriented architecture. Remote Procedure Call

(RPC) is an early style of web services. The remote invocation through the RPC protocol allows programs to communicate over the network by means of special communication protocols obviating the need for extra coding to account for network details and remote machines. Several analogues of RPC have arisen to provide more sophisticated features, such as the Common Object Request Broker Architecture (CORBA)[Corba, 1995], Java Remote Method Invocation (RMI)[ORACLE, 2011] and Simple Object Access Protocol (SOAP)[Box et al., 2000]. Systems built using these protocols are sometimes referred to as *arbitrary web services*[W3C, 2004] and they have been utilized in the MDO context.

This thesis is motivated, by the tendency that has been shown in the evolution of MDO frameworks, to propose that the features of web services are particularly relevant for MDO. The reason is that MDO requires a co-operative environment across multiple specialist disciplines that not only can access and integrate geographically distributed resources, but also have the capacity to access existing knowledge that resides in legacy code and the latest software across a range of platforms. From user perspective, there is another reason that it also needs to consider the users of MDO frameworks. They are non-computer scientists. Thus, a light-weighted and easy-to-access architecture will be helpful on improving the use efficacy. In this case, REST web service has this potential that it is built based on broadly applied protocols (HTTP/HTTPS) and interfaces (CRUD: create, read, update and delete).

1.2 Contributions

This thesis is driven by the motivation to enhance the efficiency of scientific workflow modelling and present the feasibility of implementing ROA and RESTful web services in scientific workflows. Contributions are embodied in four problems this framework aims tentatively to solve. They are:

- Data management,
- Service management,
- Data and Service Virtualization
- Service composition.

Each of them will introduce related approaches to enhance the efficiency from both application and user point of view. They are introduced in following sections.

1.2.1 Data Management

One key feature of scientific workflow is its dataflow driven nature and the large data transfers between client and server can consume significant bandwidth. In this thesis, reducing the redundant data transfer amount and enabling an asynchronous data transfer style among multiple services are considered as the keys to enhance the data transfer performance. On the other hand, from a computer science perspective, the problem of staging data in workflows has received much attention over the last decade, with a variety of user-directed and automatic solutions. In order to provide an efficient architectural style for this Internet-based data transfer mechanism, this thesis proposes a data staging model. In this model data-flows are distributed to each pair of connected web services, without passing through clients, who act as central controllers. A RESTful web service, which is called Datapool, is designed to enable clients to store, identify and manipulate data elements based on this model. By means of this distributed data transfer mechanism, the amount of data transferred from client-side to server-side and vice-versa can be reduced, by storing data server-side and exposing it as a globally accessible resource. This work around this contribution is presented in Chapter 3.

1.2.2 Service Management

From the perspective of mechanical engineering, there is an obvious gap between scientific applications and web services. The process to wrap and deploy from the former to the latter is a barrier for users who may have limited computer knowledge of the relevant technology. Following on from the aforementioned issues, questions can also be raised about the configuration and management costs of a lightweight (RESTful) web service. Hence, another contribution of this thesis is the combination of the data staging mechanism with a simple service deployment mechanism, that is designed to allow applications developed for the command-line to function as (RESTful) services without modification or (in some cases) without recompilation. Furthermore, this deployment and management mechanism is built based on a Software-as-a-Service (SaaS) manner framework. All the data and application resources can be shared through network by a distributed computing platform. This enables both a flexible and convenient service deployment and migration process. Along with this mechanism, a desktop client system is designed to further reduce the hurdle for non-specialist users. This part of work is mainly presented in Chapter 4.

1.2.3 Data and Service Virtualization

This thesis also presents an more advanced approach for data and service management based on containerization technology. This allows data and service to be virtualised. It can enhance the reproducibility and reusability of application/service. This part of work is built based on the works about data management (Chapter 3) and service management (Chapter 4). In those two parts of works, it is discovered that there are still limits on the approaches proposed for data management as well as service management. In general, there is bottleneck on data transfer and the reproducibility and reusability is still not ideal enough for non-specialists that there is space for further exploration. This part of work filled this space with a tangible solution that a ROA based scientific workflow enactment framework is also built in Python. A series experiments are carried out to present the feasibility and the enhancement on data staging and workflow enactment. This part of work is presented in Chapter 5

1.2.4 Service Composition

A scientific workflow is the combination of a series of data resources and services resources. How to construct them together to form an automatic process is yet another important issue that this thesis aims to resolve. Especially, in the case of ROA based scientific workflow modelling, the question is how to further utilise its specific features to further improve the efficiency. Essentially, this is an issue that rests on service discovery, service description and service enactment. This thesis proposes two sets of methodologies that allow workflow to operate in both imperative and declarative manners. Firstly, a complete set of methods will be presented to build a scientific workflow based on the data and service management systems together with an existing generic Workflow Management System (WMS) that works in an imperative manner. Secondly, a RESTful services-based composite service system is presented based on a Prolog based description language and a knowledge sharing mechanism. With the support of the second solution, the workflow has the ability to deal with faults at run-time and share its knowledge about the computing process. The two methodologies can be applied in different working circumstances and will be discussed later in Section 6.1. This discussion also leads to the introduction of the declarative services composition language and speculative enactment engine. The evaluation of this work is carried out in two parts: (i) A Petri net based model of the composite service mechanism is proposed for simulation. It aims to show the performance enhancement in various scenarios. (ii) An implemented workflow enactment engine based on the Prolog description language is presented. It aims to show the feasibility and the influence of introducing such a mechanism to workflow enactment. This part of work is presented in Chapter 6.

1.3 The Structure of the Thesis

The remainder of the thesis is structured as follows:

Chapter One – Introduction

In which this thesis states the central argument and the motivation. This chapter also gives a brief introduction of the current literature and the contribution this work makes to the subject area.

Chapter Two – Literature Review

In which this thesis presents a comprehensive review of related scientific workflow research, identifies gaps and observes trends. In particular, for the scientific workflow this thesis focuses on the implementation of web services, workflow system architecture, other PaaS applications and identifying the trend of web services application. For the MDO framework this thesis focuses on the literature gap in respect of the implementation of web technology and workflow together with an introduction about the evolution of MDO frameworks.

Chapter Three – Datapool: A ROA Based Distributed Data-flow Model

In which this thesis starts to build the basis for scientific workflow modelling, which is the dataflow. It is argued that the performance of data transfers in a web-based workflow system can be enhanced by the distributive dataflow model by comparing it with a centralized dataflow model. Users can use the Datapool service to easily access and manage the data collection resources, which are supported by ROA. The enhancement is experimentally verified, and is put to use in a workflow application.

Chapter Four – SaaS based Scientific Web Services Management

In which this thesis further discusses the scientific workflow modelling on services management, which includes deployment, configuration and execution. The issue of application wrapping and service management are identified based on the literature review, especially for non-specialist users. A SaaS Cloud system is presented for users to deploy, configure and manage their local application-based web resources. It enables complicated services management jobs to be turned into a series of mouse-clicking operations. A series of service management examples are demonstrated in this chapter. The system also fits into the distributive dataflow model.

Chpater Five – Virtualized Data and Services for Scientific Workflow Modelling

In which this thesis proposed a scientific workflow enactment framework based on containerization technology. It further enhances the performance on data staging and workflow enactment. It also enhance the reusability and reproducibility of application/service. This chapter starts with an introduction on the technology foundation and basic methodology. Then it elaborates the design of this framework. Following that, the chapter evaluates the framework with a series experiments. This chapter also discusses the suitable solution for scientific workflow in different scenarios as well as the impact of adoption of containerization in scientific workflow modelling.

Chapter Six – Services Composition Language and Enactment Engine

In which this thesis introduces the methodologies for building a scientific workflow based on the composition of Datapool services and application services. Firstly, an imperative method is introduced based on the application of an existing generic workflow management system. Then, a declarative method is introduced, which includes a novel declarative description for composite services and an online description knowledge system. The advantage of this system and the advantages and disadvantages of imperative and declarative methods are discussed in detail in this chapter.

Chapter Seven – Case Studies

This Chapter present several case studies. The first is the main study subject MDO. The whole system demonstrates its ability to solve MDO problems by building a web services composition based workflow. The implementation of Datapool, online web service management system and service composition is shown step-by-step to deliver a complete image of the systematic methodology when solving the issues in scientific workflow modelling. Additionally, the methodology shows its capacity to work with other existing MDO frameworks. Furthermore, other exemplar applications are also demonstrated based on this modelling system to show its extensibility.

Chapter Eight – Conclusions

This chapter summarises the contributions of the thesis, and examines how this series of methodologies have developed to support the central argument for scientific workflow modelling. This thesis also discusses the inspiration for a broader scope and suggests possible avenues for the future development of this work.

Chapter 2

Literature Review

This chapter reviews literature on scientific workflow systems and MDO frameworks and provides some background information about web services technology, resource-oriented architecture (ROA), cloud computing and speculative computing, which have been applied within this work. This chapter first reviews and discusses the main technology used in scientific workflow systems and web services, which serves as a basis and follows that is the review of MDO frameworks and scientific workflow system. Afterwards, this chapter presents a review of the literature on the use of web services in MDO frameworks. Following it this chapter reviews some key aspects of scientific workflow systems which concern this research. These aforementioned two reviews aims to present the gaps between two different areas in different development stages. In the reviews on scientific workflow systems, more detailed background knowledge is reviewed from three aspects: data staging, service management and service composition. In the review of service composition, one important technique for service composition, workflow description, is also reviewed. The aim is to draw a clear picture of what can be resolved by state-of-the-art approaches, by identifying the current gaps in existing cases and how these can be resolved tangibly. The summary identifies research trends and current literature gaps.

2.1 Web Services

For the purpose of exploring the implementation of ROA in scientific workflow, one essential step is to have a panoramic view of the state-of-the-art web services technology. As introduced in Section 1.1.1, W3C categorises web services in two ways: arbitrary web services and REST-compliant web services. Nowadays, SOAP and RESTful web services are the representative services of the two aforementioned classes that have been widely implemented. Briefly, they are described as:

- SOAP/WSDL based web Services (also known as “Big web services”) that provide an arbitrary (i.e. developer defined) set of operations, and
- REST type web services, providing a uniform interface, that is considered more “web-friendly”. For example, an HTTP(S) REST web service will use some or all of the standard HTTP protocol operations: GET, POST, HEAD, PUT, DELETE

There are pros and cons to both of them as discussed by Spies [2008], which are summarised in Table 2.1 for comparison and discussed in depth in the following sections, leading on to a comparison between service-oriented and resource-oriented architecture.

2.1.1 Advantages and Disadvantages of SOAP

Simple Object Access Protocol (SOAP) is used to defined as a web service protocol using XML technologies, an extensible messaging framework containing a message construct that can be exchanged over a variety of underlying protocols. SOAP communicates mainly based on HTTP, but can also communicate through FTP, SMTP, etc. Thus, SOAP supports communication in going around firewalls. SOAP contains three main parts [Box et al., 2000] (i) an envelope that defines a framework for describing what is in a message and how to process it, (ii) a set of encoding rules for expressing instances of application-defined datatypes, (iii) a convention for representing procedure calls and responses. SOAP is a W3C recommendation. W3C describes the set of interrelated technologies that can be utilised to construct and consume SOAP web services, as illustrated in Figure 2-1.

SOAP Message

By observing the SOAP message framework, a better understanding of the characteristics of heavy-weight and extra payload can be acquired. A SOAP message is specified

SOAP	REST
Pros	**Pros**
• Transfer over multiple protocols, not only HTTP	• Simpler to develop than SOAP
• Designed to handle distributed computing environments by SOAP intermediary	• Easier to learn, less reliance on tools
• Has better support from other standards (WSDL, WS-*) and tooling from vendors	• No additional messaging layer
	• Closer in design and philosophy to the web
	• Built-in error handling
Cons	**Cons**
• Conceptually more difficult, more ”heavy-weight” than REST	• Transfer only through HTTP
• More verbose, extra payload	• Point-to-point communication model, no intermediary in the message exchange
• Harder to develop, requires tools and higher level of knowledge preparation	• Lack of standards support for security, policy, reliable messaging, etc., may require extra development

Table 2.1: Pros and Cons of SOAP and REST[Spies, 2008]

as an XML information set whose comment, element, attribute, namespace and character information items are able to be serialized as XML 1.0. It is defined in Box et al. [2000] as comprising:

SOAP Envelope The SOAP envelope element information item has: A *local name* of Envelope; A *namespace name* of http://www.w3.org/2003/05/soapenvelope; Zero or multiple namespace-qualified attribute information items in its *attributes* property; One or two element information items in its *children* property in the following order: An optional *Header* element information item; A mandatory Body element information item. Figure 2-2 shows the structure of the SOAP Envelope.

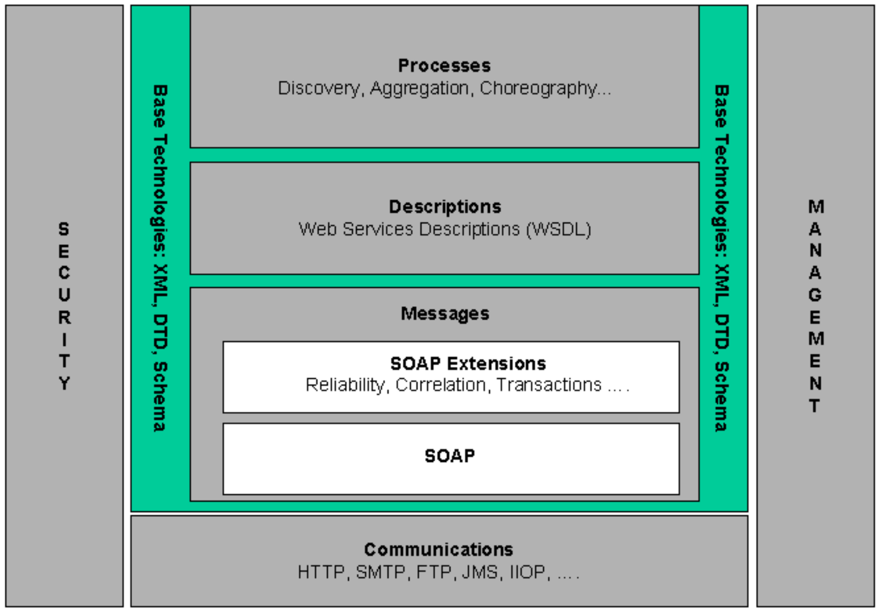

SECURITY
Base Technologies: XML, DTD, Schema
Processes
Discovery, Aggregation, Choreography...
Descriptions
Web Services Descriptions (WSDL)
Messages
SOAP Extensions
Reliability, Correlation, Transactions ....
SOAP
Base Technologies: XML, DTD, Schema
MANAGEMENT
Communications
HTTP, SMTP, FTP, JMS, IIOP, ...

Figure 2-1: SOAP Web Services Architecture Stack [Booth et al., 2004]

SOAP Header The SOAP header element information item provides a mechanism for extending a SOAP message in a decentralised and modular way. The SOAP Header element is optional. A SOAP Header serves as an extension mechanism to provide a way to exchange information in SOAP messages that is not an application payload. This information includes passing directives, that is contextual information related to the processing of the message. By this mechanism, SOAP message can be extended in an application-specific manner.

SOAP Body A SOAP body provides a mechanism for transmitting information to a SOAP receiver. The SOAP body is a mandatory element within the SOAP Envelope. It includes the main information conveyed in a SOAP message that is closest related to the functions the web service provides.

SOAP Fault A SOAP fault is used to carry error information within a SOAP message. To be recognised as carrying SOAP error information, a SOAP message *must* contain a single SOAP Fault element information item as the only child element information item of the SOAP Body.

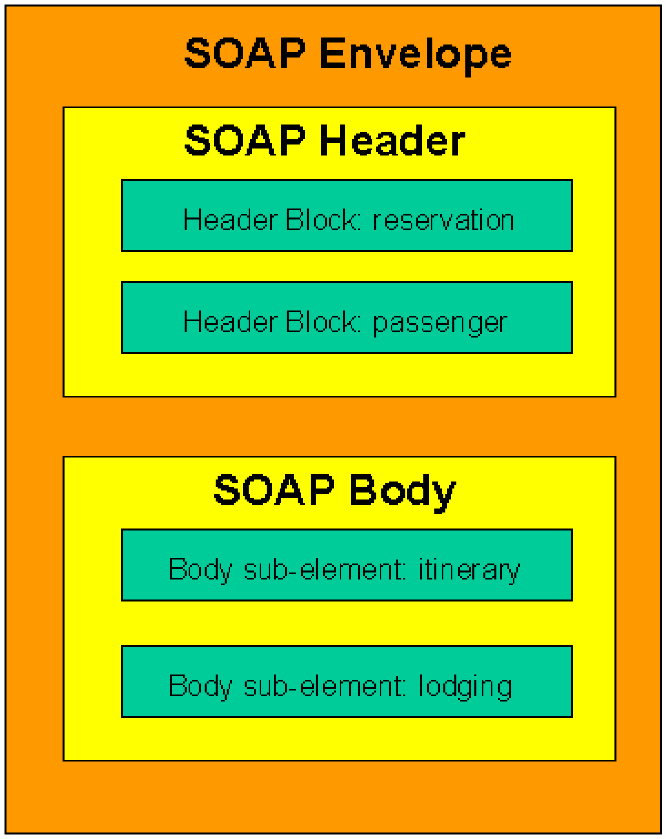

SOAP Envelope
SOAP Header
Header Block: reservation
Header Block: passenger
SOAP Body
Body sub-element: itinerary
Body sub-element: lodging

Figure 2-2: The structure of SOAP Envelop Mitra et al. [2003]

2.1.2 REST Web Service

The term REST is now widely used to refer to REST-complaint web services. However, when REST was originally coined by Fielding, it denoted an architectural style. The fundamental features of this architectural style are summarized clearly by Fielding [2000] as:

> *The Representational State Transfer (REST) style is an abstraction of architectural elements within a distributed hypermedia system. The REST ignores the details of component implementation and protocol syntax to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the web architecture, and thus the essence of its behaviour as a network-based application.*

In the Chapter 5 of Fielding's thesis [Fielding, 2000], he elaborates on the architectural constraints of REST. They also form the basis for the guidelines of Resource Oriented Architecture which will be further discussed in Section 2.1.3. One basic constraint is that REST is based on the principle of a client-server architecture. Client and server are separated with their certain concerns. Clients only initiate requests based on user state to services. Services only process requests and return responses based on data stored without reference to user state. They communicate with each other through a uniform interface and this is the only matter of concern to both client and server. Both request and response are built around the transfer of a representations of a resource. In REST, the representation of a resource means a document in a certain format that captures the current or intended state of a resource. State transfer means that, when clients send requests, they are ready to transit from current state to a new state. In the representation of application state returned by servers, there are also links in the form of URIs that can be chosen by the client to lead to a new state transition. With such an client-server architecture, clients and servers have a separation of concerns. Thus, servers and clients may also be designed, developed and configured independently, as long as the uniform interface between them is not altered.

In HTTP(S)-based RESTful web services, the emphasis is on simple point-to-point communication over HTTP(S). As designed by Fielding [2000], REST does not have to be coupled with HTTP only, but in real implementations, only HTTP is applied. REST architectures that use the HTTP application protocol can be summed up as using five verbs (GET, HEAD, POST, PUT, and DELETE methods from HTTP 1.1) and a collection of nouns, which are the resources available on the network, referenced through Universal Resource Identifier (URI). The verbs have the following operational equivalents, shown in Table 2.2. Here CRUD means Create, Read, Update and Delete. They are the the four basic functions of persistent storage, which normally used to describe user interface conventions such as the verbs based interface for REST web service.

In the REST uniform interface, which is based on the HTTP verbs, the convention has been established that the GET and HEAD methods should not have side effects that it should only take the action of retrieval as the operation READ in the CRUD. These are the so-called safe methods of HTTP[Fielding et al., 1999]. Methods that only take the action of read should be considered as "safe". Methods can also have the property of "idempotence" that the side-effects of two or more identical requests are

HTTP	CRUD Equivalent	Safe	Idempotent
GET	Read	YES	YES
HEAD	*Get metadata*	YES	YES
POST	Create, Update, Delete	NO	NO
PUT	Create, Update	NO	YES
DELETE	Delete	NO	YES

Table 2.2: HTTP verbs

the same as, or cannot be distinguished from that for a single request. The methods GET, HEAD, PUT and DELETE have this property.

As shown above, REST web services can work directly with HTTP, so they have one less protocol layer (by comparing with SOAP and web service protocols based on application layer protocol). Based on such a fact, tt potentially can reduce the overall cost of computation and knowledge preparation. Pautasso and Wilde [2009] concludes that for the objective of designing IT architectures, REST is a better choice in the context of a co-operative environment in that it is intrinsically loosely coupled because of its asynchronous communication nature, universal addressability and uniform interfaces. Evidence of the recognition of these properties is visible in the many new public web services from large vendors, such as Google, Yahoo and Amazon, that are now REST based.

2.1.3 A Comparison between SOA and ROA

Service Oriented Architecture and Resource Oriented Architecture are commonly applied in the architecture design of web services based frameworks. In SOA, a service is designed and organized on the basis of verbs, which correspond to operations in SOAP. In ROA, a service is designed and organized on the basis of nouns, which corresponds to resources (referenced by URI) in REST. Thus, SOAP is often associated with SOA while REST is often associated with ROA. This leads to decision-making on both technology and architectural style.

Based on SOAP, SOA tends to model the workflow as service end-points or so-called actions. Every service serves as a distinct unit, a black box of functionality. Therefore, SOA focuses on interfaces like *method* and *operation*. It is geared towards the applications that are activity-based, which is strongly related to the RPC-style, thus it can better model transaction-based business systems. Based on such an architectural design, it also heavily depends on an extra layer of protocol for interface description. Thus, there is tight coupling between client and server because of object reference semantics, object serialization and early binding to interfaces defined in the interface description.

ROA, in contrast, is normally considered as a set of constraints or guidelines for the implementation of REST-based web application. Other than the technical features discussed in Section 2.1.2, ROA summarises the guidelines as four properties [Richardson and Ruby, 2007]:

Addressibility Addressability means to expose the resources, applications or data, through URIs that can be located and accessed universally. In the prototype system that will be introduced in the following chapters, it means all related resources should be allocated appropriate URIs. In addition, it would be desirable that the URI could self-explain the function and access method to some extent. For example, the URIs can be designed to include the type of the requested resources in accordance with the W3C note on "cool URIs" Sauermann et al. [2011].

Statelessness Statelessness means that every HTTP request should be isolated from any user session information associated with previous requests. The HTTP request made by a user must contain all the information to fulfil this request. The application state should be placed at the client side and the resource state at the server side. Essentially, the application state is different for each user, such as some session information about individual visitors. The resource state is the same for every user.

Connectedness RESTful web services usually present resources as hypermedia, which means the resources not only contain data but also links to other resources. This is built on the addressability of all the resources. This feature can increase the extendibility of the system and save storage space in the case of multiple data instances. Therefore, a web service is connected to the extent that you can put the service in different stages just by following links and completing forms.

Uniform Interface Create, Read, Update and Delete (CRUD) are the four basic operations on persistent storage. The HTTP uniform interface covers the four basic operations by POST, PUT, GET and DELETE methods. For example, given a URI of a resource, every user knows that to retrieve the resource, a GET request should be sent to that URI.

Briefly, the language analogy is that the components in SOA serve as arbitrary verbs (actions) and in ROA serve as a series of nouns (resources). There are also verbs exposed by ROA, but they are constrained in that they are based on the uniform interface. Furthermore, ROA as an approach aims to describe a SOA implementation that follows the guidelines stated above. In other words, ROA is more particular in its dependence on the technical architecture of the services, which is REST.

2.2 The Implementation of Web Services and Workflow in Engineering Framework

A question, when employing network technology, is which of the many protocols and frameworks to choose, and indeed which will last. Others have faced the same question. For example, FIDO [Weston and Townsend, 1994] is proposed with functions that offer control over processes, monitoring of execution status and visualization of problem data in a networked system of heterogeneous computers. Another early framework, DARWIN 1 and its successor DARWIN 2 [Walton et al., 2000], allows users to access data through web browsers. The data are dynamically generated by means of Common Gateway Interface (CGI) scripts and visualized with Java applets embedded in the web pages. The common themes here are the integration of hardware platforms and remote process control and monitoring, but are intrinsically techno-centric, rather than problem-centric: resources can only be accessed by users through specific interfaces, and frameworks are constructed based on a bespoke deployment and communication mechanism. These both limit the sharing of resources and seamless access to substantial computing resources.

For the purpose of supporting an interoperable machine-to-machine interaction mechanism through the Internet, web services have been an active research and standardisation topic for several years now, reinforced by the emergence of the concept of the service-oriented architecture (SOA). Remote Procedure Call (RPC) is an early style of web services, still in limited existence albeit with disadvantages. The remote invocation through the RPC protocol allows programs to communicate over the network by means of special communication protocols that obviate the need for extra coding to account for network details and remote machines. Several analogues of RPC have arisen to provide more sophisticated features, such as the Common Object Request Broker Architecture (CORBA) [Corba, 1995], Java Remote Method Invocation (RMI) [ORACLE, 2011] and Simple Object Access Protocol (SOAP) [Box et al., 2000]. Systems built using these protocols are sometimes referred to as arbitrary web services [W3C, 2004] and they have (all) been utilised in the MDO context.

For example, the ModelCenter software utilises RPC-style web services to deploy legacy programs remotely that are then accessible through a special applications program interface (API) via ModelCenter [Ng et al., 2003]. Analysis Server [Phoenix Integration, 2001], a web services container, is used to publish web services and allow remote access from ModelCenter. ModelCenter and the Analysis Server together provide a client/server environment for distributed resource access. In this environment, Quick-Wrap is the local tool used for automating command line programs and generating services for Analysis Server. Based on this local tool solution, local machine administrator permissions and relevant administration skills for software installation and web services deployment are mandatory.

The VADOR framework uses a similar client-server architecture but is mediated by a Java RMI package to provide remote access [Alzubbi et al., 2000]. In VADOR [Mahdavi, 2002], another wrapping mechanism is implemented. The CPU server, which is a JAVA server program, is designed to act as a wrapper and remotely execute the analysis task. To run a task, the CPU server performs the following steps [Zhou et al., 2003]:

1. Locates the executable program, application, or script file;
2. Builds a temporary directory;
3. Transfers all input files to the temporary directory;
4. Runs the program, application, or script file;
5. Transfers the created output files;
6. Cleans up and destroys the temporary directory.

These are the typical and basic steps shared in the process of wrapping scientific applications. CPU servers run the legacy scripts and programs locally. Although this server can only wrap the programs or scripts that already exist in the local machine, these steps still can be the guidelines for the design of web services based scientific application frameworks.

The Java RMI interface has been used in iSIGHT and Fiper [Sobolewski, 2002], which provide a web representation of the analysis and results accessible via a web browser. The Fiper framework utilises web-based components for the purpose of remote execution. Analysis components are executed and monitored through a web browser which provides an open API and a Component Generator for the job of development of components. It offers a JAVA based mechanism to wrap applications. In this process, coding work of application provider is unavoidable.

In recent years, SOAP has gained wide acceptance as a web services protocol and Lee et al. [2009] demonstrates a MDO framework based on SOAP. They deploy and execute MDO methods using the Globus Toolkit [Foster and Kesselman, 1999], however it is, as with ModelCenter, only accessible through a particular client. In this case, it is also accessible via a web browser.

The aforementioned existing MDO frameworks are typically implemented in a way that services are mapped directly to language-specific function calls. Thus, the limitation on access to each of the RPC analogues arises. This means the services provided by a server can be accessed and executed only by specified clients. For example, a SOAP-based design optimization process [Crick et al., 2009] and the Soaplab tool [Senger et al., 2008b] have been used to deploy SOAP services. In that case, client tools can only access the services based on integration with the Soaplab plug-in that has particular library and protocol. Indeed, the interfaces of the RPC analogues (e.g. RMI and SOAP) have different and incompatible communication protocols which inhibits access to service components created by different frameworks. This restriction is inherent to arbitrary web services and reduces the overall interoperability between systems.

This *tight coupling of components* characteristic of SOAP can potentially increase the costs of migrating from one client tool to another, or of introducing new application tools to existing frameworks that operate using bespoke protocols. The question that naturally arises then is whether there are any other service types that can resolve the restriction issue.

There are also some MDO frameworks supporting the extension of functions through external code. For example, the MDO framework DAKOTA [Sandia National Laboratories, 2010] simulation code can be deployed as components on networks of workstations (NOWs) or desktop multiprocessors. It requires further coding work involving scripts or programs as local components based on provided APIs.

2.3 Scientific Workflow System

With the development of computer technology, the trend can be observed that the network plays a more important role to support large-scale science research through geographically distributed co-operation. The issue of enabling individual domain scientists to access large scale data collections and computing resources in this Internet environment has become key to defining research in e-Science [Deelman et al., 2009]. Large-scale computations and data analysis characterize much scientific research now, meaning there is a need for new approaches to emerge in order to support computation and management of distributed resources in what is also a new execution environment. Workflow is a sequence of connected activities or operations that abstract and model as an activity. In business applications, it is defined as an orchestrated and repeatable pattern of business activity, enabled by the systematic organization of resources into processes that transform materials, provide services, or process informationwebmaster@ftb.ca.gov [2009]. In the area of scientific computation, it has emerged as a paradigm for representing and managing complex heterogeneous scientific resources. Scientific activities can also be systematically organized and processed in such a paradigm. For example, each of the data management and computation process can be abstracted and modeled as activities in workflow. This brings new challenges in the context of e-Science research, such as application requirements, data and workflow description, dynamic workflows, user steering and system level workflow management [Gil et al., 2007].

As mentioned in Chapter 1, a number of workflow management systems (WMS) [Ludäscher et al., 2006, Microsoft, 2011, Oinn et al., 2004, Taylor et al., 2007, The YAWL Foundation, 2004] have been developed to provide the functions of workflow composition, enactment and monitoring. As basic functions, users can use these WMSs to build their workflow, execute the computation process and retrieve the outputs from it. In this section, the review of WMSs will be presented through the three main issues of this thesis, that we aim to study in scientific workflow modelling working from bottom to top: data staging, web service management and service composition. Based on this review, the adaptivity of each workflow management system in a cooperative environment can be discerned. Furthermore, this chapter aims to determine these WMSs' advantages and disadvantages in respect of handling data management, service deployment and service composition. These are the key points in order to lower the barriers brought by the features of multiple institutions, multiple platforms and geographically distributed resources that are central to the cooperative environment of scientific workflow.

2.3.1 Data Staging

Conventionally, web service composition frameworks have centrally coordinated control-flows and data-flows, so the data associated with the execution process has to be staged through a client-side management system. This data staging activity handles the data processing activities, such as data I/O, on an intermediate storage area used in the middle of the computational processes. The staged data in the area is used for the execution of corresponding computing activity in the whole (workflow) process. The centralized data-flow approach can easily increase the level of data traffic and the situation is exacerbated where large data sets are concerned[Alonso et al., 1997]. In general, centralised approaches are easily implemented and suit workflow applications in which large-scale data flow is not required. Taverna [Oinn et al., 2004] is a typical WMS that implements centralised data staging.

Data staging and how to control it are not new problems. In 1997 [Alonso et al., 1997], adopted the idea of distributed dataflows in a service composition framework to improve data transfer performance, as did [Liu et al., 2005] some years later. Similar ideas are embodied in some distributed program execution engines, such as [Isard et al., 2007, Murray et al., 2011], to overcome bottlenecks in data transfers. Meanwhile, several workflow management systems utilise a peer-to-peer style mechanism for intermediate data movement Cao et al. [2003], Ludäscher et al. [2006], Taylor et al. [2007]. Although there are differences in detail between the aforementioned solutions, there is one common aspect, namely the use of a private – by which is meant internal, or closed – mechanism (functions are exposed by a set of developer-defined specific interfaces and operations) to handle data transfer.

In early research on peer-to-peer data staging solutions, a distributed data management architecture for workflow management system was adopted in the distributed workflow environment Exotica/FMQM [Alonso et al., 1997]. The motivation was to re-solve the problem of poor performance when data-flow is embedded with control-flow. The solution implemented only works on a local-area network (LAN), by means of a set of loosely synchronized replicated databases. Several other workflow management systems have adopted such an architecture, but using more recent technology, namely Triana [Taylor et al., 2007] and Kepler [Ludäscher et al., 2006], both of which apply JXTA [Gong, 2001] – a peer-to-peer protocol – to allow services to exchange messages. Both use an internal mechanism to identify resources in the network based on a unique identifier. GridFlow [Cao et al., 2003] offers a slightly different solution at the implementation level, utilising an agent-based resource management system, whose task it is to transfer between peers realized as agents. The Globus Replica Location Service (RLS) [Chervenak et al., 2009], used in some Grid-based workflow management systems for peer-to-peer data transfers, takes a similar approach, but with different components, maintaining a simple registry that keeps track of where data resides in the Grid environment. RLS also has a web service incarnation, called WS RLS, which is based on the Web Services Resource Framework (WSRF). This works with the WS-addressing specification to identify data at a messaging level. The common feature in all of these data staging solutions, is either the use of an extra layer of private protocols, above the Internet application layer protocols, or the introduction of a potentially complicated (internal) system to achieve peer-to-peer data transfer.

2.3.2 Service Deployment and Management

For the purpose of considering service deployment and management in a co-operative environment, one key point that should be emphasized is the high possibility that users are non-specialists with limited knowledge and experience of relevant computer technology. This determines whether the system provides a relatively simple architecture and interface for installation as well as manipulation for those non-specialists users.

Web Service Wrapping Tools

In (e-)science research, web services are commonly created for and invoked within workflows. Web service end-users can utilise workflow management systems to compose and execute them. Taverna [Oinn et al., 2004] is a popular platform for this purpose. It can invoke Soaplab [Senger et al., 2008b] and Opal [Krishnan et al., 2006] web services through Taverna's plugin mechanism. The Kepler workflow tool [Ludäscher et al., 2006] can also invoke services created by these two application wrapper tools. However, both of these workflow tools need to develop extra plugins, using Soaplab's and Opal's programmatic APIs, in order to invoke web services seamlessly. Specific

APIs for each scientific application's wrapping tool may place an extra barrier for workflow engines to invoke corresponding web services. For example, both gRAVI [Chard et al., 2009], a WSRF web service wrapping tool and Generic Factory Service (GFac)[Kandaswamy et al., 2006], a web service wrapping tool, have three ways to invoke services: Java Client API, web interface and command line. However, neither wrapping tool provides a method for invocation from workflow engines directly. Therefore, a plugin mechanism is necessary to handle the interface between workflow and service. Thus, there always appear to be hurdles for non-specialists to deploy scientific applications for use within workflows. Although Taverna and Kepler can support Soaplab and Opal services via plugins, because programmatic access to those services needs specific APIs, this creates a restriction on the programming languages it is able to support. For instance, Soaplab2 supports Java and Perl whilst Opal supports Python and Java.

Soaplab and Opal are widely used and build on the SOAP specification. In Soaplab, access to the underlying processes is provided by some generically named operations like *createJob, run, destroy, getSomeResults* etc., but data is not sent to the server through distinct SOAP endpoints. More precisely, each data object must be packed into a Map data structure and sent by invoking the *runAndWaitFor* method. These technical details are handled by a mature plugin in Taverna and are transparent to workflow management tool users. The Soaplab service in Taverna can expose several interfaces allowing users to extract each data item directly. Each output from a service can be directed to all the input ports for following services. However, this is also the reason why specific APIs must be included in order to invoke a Soaplab service, which also happens on the other SOAP-based tools. We also observe that Soaplab does not have any security mechanisms to encrypt data and perform role-based access control over users.

Opal, in contrast, uses a more ad-hoc solution to address the payload format issue. For example, in the Opal plugin for Taverna [University of Manchester, 2010], an example of an Opal web service is presented, which shows the response from the Opal service as a string of comma-separated URLs of output files plus the content of the standard output stream from the server-side application. Some regular expression services must be created by the user locally in order to select the necessary URL, which will be fed into the next Opal service as input. This would appear to raise the (user) cost of workflow creation and linking of web services based on Opal, requiring knowledge of regular expressions, as well as the data returned being a URL pointing to the data rather than the data itself. Although this RESTful-like feature allows the URL to be used as the input URL for compatible web services [Ren et al., 2010], an incompatible web service needs extra steps and data transfer to obtain and utilise the output. Compared to Soaplab, Opal does however provide a security mechanism for data encryption, authentication and authorization. It is based on the GSI security system, which uses the Secure Sockets Layer (SSL) for its mutual authentication protocol.

Cloud Based Web Services Deployment and Management

Cloud computing is commonly categorized into three service models [Mell and Grance, 2011] known as {Infrastructure, Platform, Software} as a Service (IaaS, PaaS and SaaS, respectively), of which PaaS is the service model which provides consumers with the capability to deploy consumer-created or acquired applications onto cloud infrastructure, thus creating a service. One aim of this research is to provide ready access to cloud services so that regular users can deploy their own applications as services, share them with others and utilise them in service workflows. This is done through the provision of a platform (i.e. PaaS) that provides: (i) deployment services, and (ii) data storage and transfer services.

PaaS cloud has been used to deploy science and engineering applications, in which the platform enables applications to appear as web services, creating a SaaS for public invocation. There are several generic PaaS platforms like Google's APP Engine [Google, 2008] and Heroku [Lindenbaum et al., 2008], both of which provide the means for users to deploy web applications on the providers' public cloud infrastructure. However, both work via programming language APIs. For the purpose of deploying an application into their infrastructures, users must either write applications in specific languages or modify original code in those languages. Other potential platforms providing command-line interfaces are: (i) CloudFoundry [GoPivotal, Inc., 2011], which provides an open-source mechanism for application deployment. However, it uses its own API, implemented for a range of popular languages for service interaction, and (ii) Openshift [Red Hat, Inc., 2011], which only appears to provide a platform for running applications using cloud resources, rather than actually deploying them as services that others may use. This is a very rapidly evolving area in terms of service provision and what was true and current at the time of writing, may not hold for very long. It is clear PaaS cloud is receiving much attention through the development of experimental deployment stacks, but our focus aims as much as is feasible to stay on architectural issues that underpin the design and implementation of such frameworks.

The Generic Worker framework [Simmhan et al., 2010] has similar goals to our framework: it provides a PaaS service based on Microsoft's Azure Cloud platform and services can be deployed by the client using command-line tools. They also adopt a distributed data transfer mechanism for performance enhancement. However, their services are tightly connected to Azure service elements, such as Azure's REST web service API and the Azure blob store.

2.3.3 Web Services Composition

There are two main types of solutions for web services composition that have been developed independently of one another. The first approach comes from the business world, which formalizes a series of XML-based standards as the specification for web services, their flow composition and execution. It has primarily syntactic interfaces and takes the view that web services work as remote procedure calls. The other approach stems from the Semantic Web community designed series of ontologies whose purpose is the explicit and precise declaration of the preconditions and effects of a web service. Thereby, a composition of web services can reason about web resources and plan flow by means of a goal-oriented inferencing.

Concretely, there are many web services flow specification languages like BPEL4WS [Khalaf et al., 2003] and WSCI [Arkin et al., 2002] that exemplify the first approach. Semantic Web community has widely discussed semantic annotations for a number of years, arriving at a solution in which preconditions and effects of services are explicitly declared in the Resource Description Format (RDF), based on the terms from widely-accepted ontologies. Not only those web services flow specification languages, but also the description languages of workflow can provide approaches for web services composition. On this basis, this section aims to give an overview on the existing technologies for web services composition, as well as related technologies on workflow description.

Service Composition and WSDL

Web services, which are machine-readable, need an appropriately formatted description of additional information on either their functions and/or the methods to interact with them. WSDL specifies only the syntax of messages that enter or leave a single SOAP service. In the composition of a group of SOAP services described by WSDL, the flow of message exchange between SOAP services must be described by a specification out of WSDL.

As aforementioned, in the business world, there are many web services flow specification languages, but there is not an obvious common standards stack. Thus, each organisation is free to build up their own proprietary business protocols. There are languages, such as BPEL4WS and WSCI, both of which use WSDL as the interface with web services. The composition of the flow is typically constructed manually. Semantic annotations have been widely discussed in the Semantic Web community where preconditions and effects of services are explicitly declared in the Resource Description Format (RDF) using terms from pre-agreed ontologies. There is also work on light-weight annotation of WSDL[Kopeckỳ et al., 2007]. All those concrete implementations are introduced and discussed in following text.

BPEL4WS BPEL4WS (commonly also known as WS-BPEL or BPEL) is the de facto standard for web services orchestration and business process modelling. BPEL4WS is an OASIS standard executable language for specifying actions within business processes with web services. Its primary proponents and supporters are Microsoft and IBM. It describes the control logic for web services coordination in a business process. The process is interpreted and executed by a BPEL [Curbera et al., 2003] engine. It includes a set of control structures like those in imperative programming languages (if/else, repeat/until, while, etc.), but additionally it also offers parallel loops and XML processing.

BPEL4WS uses SOAP-style web services as a base. This fact is embodied from three points: (i) Every BPEL4WS based composite service is exposed as a web service using WSDL. For example, the public entry and exit points of the process are described in WSDL. (ii) A BPEL process communicates with external web services through WSDL interfaces. (iii) The information in the BPEL process is described by WSDL data types directly.

WSCI WSCI [Arkin et al., 2002] is a language for describing the sequence of web service invocations. More precisely, it is a language for specifying the conditions under which a particular operation can be invoked. It is proposed by Sun, SAP, BEA,[1] etc., and has been submitted to W3C as a *technical note*. In compar-

[1]Sun:https://www.oracle.com/sun/index.html, SAP:http://go.sap.com/index.html,

ison to BPEL4WS, BPEL4WS primarily focuses on the creation of executable business processes in a centrally-controlled manner, while WSCI is concerned with public message exchange between web services that in the context of a distributed system. This is also the primary difference between an orchestration language and a choreography language. Similar to BPEL language, WSCI is also XML-based. About the services grounding in WSCI, actions, which are tagged by `<action>`, represents a unit of work and would typically map to a WSDL defined service invocation. Thus, WSDL describes the entry points of each service, while WSCI describes the interactions between these services.

WSCI does not concern itself with the creation of executable business process. So WSCI uses Business Process Management Language (BPML), a metalanguage, to describe business processes. In other words, WSCI only defines the interactions between services, and BPML defines the big image of the business processes. BPML provides similar process flow constructs and activities as BPEL.

SAWSDL Semantic Annotations for WSDL and XML Schema[Kopeckỳ et al., 2007] was published as a technical recommendation of W3C in 2007. It describes the abstract functionalities and other semantic information of a service by defining a set of extension attributes for WSDL. It is not yet another language for representing the semantic model, such as ontologies. On the contrary, the mechanisms it has enable it to reference to the concepts from the semantic models, typically defined outside the WSDL document, from within WSDL and XML Schema components by annotations. For example, it uses the the extension attribute *modelReference* to represent the URI link to the semantic model. To specify the mappings between semantic data and XML, two other extension attributes, *liftingSchemaMapping* and *loweringSchemaMapping*, are added to XML Schema element declarations and type definitions. In a nutshell, SAWSDL does not describe the web services composition directly. It makes WSDL-based web services amenable to Semantic Web Services automation for the purpose of forming a composite web services (CWS).

BEA:http://www.oracle.com/us/corporate/acquisitions/bea/index.html

Web Services Composition in the Semantic Web

In semantic web, web resources are described or annotated by semantic marks that the whole World Wide Web can be globally interlinked as an database of semantic data. There are large amounts of data spread across the web that is only readable to humans: the goal of semantic web is to process and share the knowledge implied in the data automatically as machine-readable. The semantic web works as a representation of the knowledge, which is well-defined in a format that is machine-readable. Web Ontology Language for Services (OWL-S)[Martin et al., 2004] and Web Service Modeling Ontology (WSMO)[De Bruijn et al., 2005a] are two prominent techniques used for services composition in the domain of semantic web.

OWL-S OWL-S, which was formerly called DAML-S, is a language based on OWL (Web Ontology Language) used for describing semantic web service properties. The OWL-S ontology has three main parts: *service profile*, *process model* and *grounding*. They have the following usages:

1. The service profile is used to describe what the service does. This part includes the service name and description, limitations on applicability and quality of service, publisher and contact information. It mainly aims to provide human-readable information about the web service.
2. The process model is used to describe how to perform the web service's functions, which includes the sets of inputs, outputs, pre-conditions and effects (IOPE) of the service execution.
3. The grounding specifies the details of how to interact with the web service based on web service protocols, which includes communication protocols, message formats, port numbers, etc.

The grounding part is filled with a description of the actual web service, for which WSDL is the commonly-used description language. OWL-S is mapped to WSDL through the following three aspects:

1. An OWL-S atomic process is grounded as a WSDL operation.
2. The I/O of an OWL-S atomic process are grounded as WSDL messages.
3. The I/O types of an OWL-S atomic process are grounded as WSDL abstract types.

Therefore, in practice, OWL-S has a high degree of coupling with SOAP. Based on OWL-S and WSDL, it is possible to describe web services declaratively, which in turn enables the automatation of web services composition[Chapman et al., 2007].

WSMO Besides OWL-S, there is another conceptual model used for describing the semantic web services called the Web Services Modelling Onology (WSMO) [Feier et al., 2005]. WSMO has four main components:

Goals *The client's objectives when consulting a web Service.*

Ontologies *A formal semantic description of the information used by all other components.*

Mediators *Connectors between components with mediation facilities. Provides interoperability between different ontologies.*

Web Services *Semantic description of web Services. May include functional (Capability) and usage (Interface) descriptions.*

As presented in [Kopeckỳ et al., 2005], WSMO is grounded to WSDL. The W3C member submission document [De Bruijn et al., 2005b] also shows that WSMO only specifies the grounding in WSDL. Therefore, as with OWL-S, WSMO also targets the so-called "Big" web services [Pautasso et al., 2008], which mainly operate under the paradigm of arbitrary web services [W3C, 2004] or Remote Procedure Call (RPC). Both OWL-S and WSMO use Semantic Web elements such as ontologies that pre-date the concept of Linked Data or Connectedness in ROA. Neither OWL-S or WSMO appear to have stood the test of time, in that despite extensive web search, it seems there is currently (2014) no substantial real-world usage.

Discussion of Web Services Composition

In the past decade, great efforts have been expended on semantic web service models. They are not limited to those mentioned above[Akkiraju et al., 2005, Kopeckỳ et al., 2007]. However, there is no substantial real-world usage of them. In this process, it can be observed that there is a strong binding between SOAP+WSDL and semantic web service models. One important reason is that SOAP became popular as a web service protocol and has been widely implemented since it became a W3C recommendation in

2003. There is semantic annotation for REST web resources, called SA-REST[Sheth et al., 2007], that supports semantic description in a /emphmashup, which is the web application hybrid to create a single new service from more than one source. However, its annotations are based on human-readable HTML or XHTML, which has limited utility if the resources are machine-readable only web services. Thus, it is difficult to apply the languages in previous approaches to a new type of web service, namely the RESTful web service. The difficulties are reflected from the following two perspectives:

1. There is no binding from semantic web services languages to RESTful web services description. Semantic web services cannot be grounded to the RESTful web services in an straightforward way.

2. There are fundamental differences between SOAP and REST, in that SOAP is operation-based and REST is resource-based. In the design of semantic web service models, it naturally follows this operation-based style. This design allows those semantic web serivce models to be adaptable to their grounding targets. This basic architectural difference means that the key concept of resource cannot be reflected in those models.

2.3.4 Workflow Description

Nowadays, scientific workflows are largely built based on distributed resources, which rely heavily on the support of web services. Some of the Workflow Management System (WMS) involve standard or proprietary workflow definition language that can sew web services and other resources together into one workflow; nevertheless, the final goal is to enable the workflow enactment. This section investigates the existing solutions for workflow description and workflow enactment, especially focussing on their implementation in WMS. This section aims to bring the ideas from those solutions to bear on the design of a novel service composition language.

Two Major Phases for Workflows

The Workflow Management Coalition (WfMC)[Coalition, 1996] defines workflow as the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a

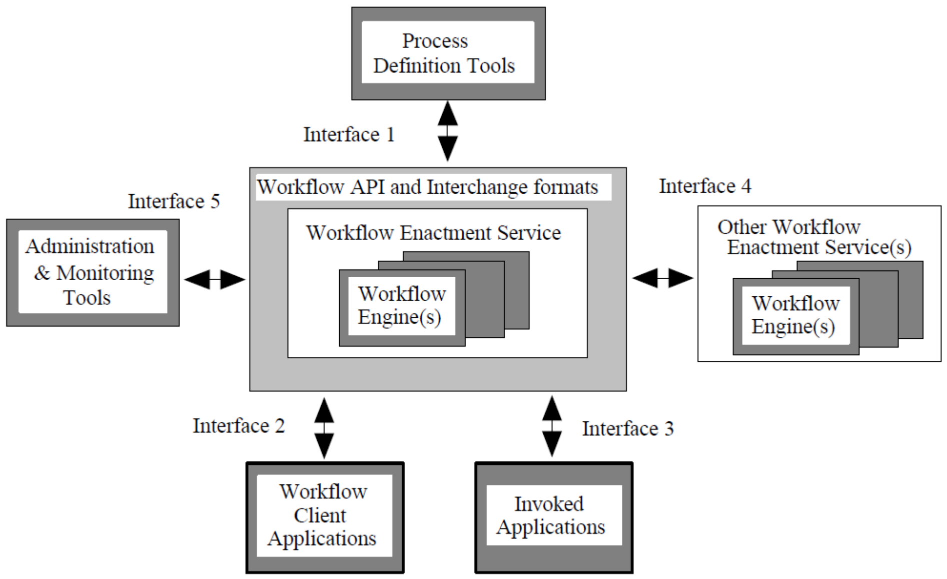

Process Definition Tools
Interface 1
Workflow API and Interchange formats
Interface 5
Interface 4
Administration & Monitoring Tools
Workflow Enactment Service
Workflow Engine(s)
Other Workflow Enactment Service(s)
Workflow Engine(s)
Interface 2
Interface 3
Workflow Client Applications
Invoked Applications

Figure 2-3: Workflow Reference Model - Components & Interfaces [Hollingsworth and Hampshire, 1993]

set of procedural rules. In other words a workflow consists of all the steps that should be executed in order to deliver an output or achieve a goal. These steps (tasks) can have a variety of complexities and usually are connected in a non-linear way, forming a directed acyclic graph (DAG). The dependencies among the processing steps can be temporal (for example, that step should run after this one), data, or control semantics.

A Workflow Management System defines, manages and executes workflows through the execution of software that is driven by a computer representation of the workflow logic. The description of a workflow includes the definition of different tasks, their interconnection structure and their dependencies and relative order. This description of a workflow's operational aspects can be expressed in textual (e.g. XML) or graphical form (e.g. as a graph in Business Process Modelling Notation [White, 2004] or Petri nets [Van Der Aalst, 1998]).

The Workflow Reference Model[Hollingsworth and Hampshire, 1993] proposes the model shown in Figure 2-3. This model defines two major phases for workflows:

1. The *build phase* where the workflow is defined in terms of a textual or graphical language using some Process Definition Tools. Various modeling languages that have been proposed including Web Services Flow Language (WSFL)[Leymann et al., 2001], Microsoft's XLANG[Thatte, 2001] for BizTalk, and Business Pro-

cess Execution Language for Web Services (BPEL4WS), which is the merging of WSFL and XLANG for web services orchestration.

2. The *run phase* where the workflow is enacted according to its definition by a workflow execution (enactment) component or a workflow engine. At this phase, the execution of some tasks may require the interaction with users or other software applications and tools.

Workflow Description Solutions for Workflow Management System (WMS)

This section surveys the workflow description solutions in some of the WMS that have been popular in recent years in the scientific user communities.

Taverna SCUFL (Simple Conceptual Unified Flow Language) is the workflow language used by Taverna 1.7.x. This language is essentially a data-flow-centric language[Oinn et al., 2006], defining a graph of data interactions between different services. It is based on IBM's WSFL[Leymann et al., 2001] which is the one of the predecessors of BPEL, including support for specifying control-flow and data-flow. The SCUFL model contains a set of inputs, outputs, processors, data links and control links. The processor represents a logical service that is an individual step within a workflow. It is specified in the design stage of workflow.

In Taverna 2.x, a new workflow language *t2flow* is adopted. It is a serialization format that is very close to the Java object model. It contains various items that are simply Java beans that serialized using XMLBeans. From the functional point of view, it is similar to SCUFL. However, its drawback is being a serialization format that can be difficult for third party software to operate on. In Taverna 3.x, it will be replaced by SCUFL2 [Williams and Soiland-Reyes, 2012]. Its beta release was in 2012. Now, Taverna 3.x is still not released and there is no clue it has been involved in current version of Taverna. It mainly aims to overcome the problem in the *t2flow* that its serialization format suffers from being very close to the Java object model, and contains various items that are simply Java beans serialized using XMLBeans. As the t2flow format is very verbose, it can be difficult to deal with for third party software. The workflow structure of SCUFL2 is defined using an OWL ontology written by the Taverna developers.

Kepler Kepler adopts a non-standard workflow description language [Barker and Van Hemert, 2008] called Modeling Markup Language (MoML)[Lee and Neuendorffer, 2000]. It is based on XML and specifies interconnections of parameterized, hierarchical components. Based on this language, the workflow in Kepler is a composition of independent components (actors) that communicate with each other through interfaces called ports. Each actor is parameterized that it exhibits a concrete behaviour, which is specified at design time. The execution of a workflow is controlled by a director, which is the object that orchestrates the actors, initializing them and controlling their execution.

The ACGT Workflow Editor The ACGT Workflow Environment[Martin et al., 2011] supports BPEL as its standard workflow definition and enactment language. BPEL is introduced in more detail as an language for web services composition in 2.3.3.

YAWL YAWL [Van Der Aalst and ter Hofstede, 2005] is a workflow system, based on a modelling language. The language has the same name as the system, which stands for "Yet Another Workflow Language". The YAWL language design is based on Petri nets with the aim of preserving their strengths for the specification of control-flow dependencies in workflows, with extensions that allows for straightforward specification of workflow patterns[Van Der Aalst et al., 2003]. The tasks specified in the workflow are associated with SOAP web services are design time. It is enacted through the Web Service Invoker Service in YAWL workflow system[Adams and ter Hofstede, 2009].

There are also some WMS do not have an explicit workflow languages to support the design of workflow, such as Triana[Taylor et al., 2007]. The workflow in Triana is written based on XML, which is proprietary. Its workflow file is saved as plain text without a postfix. Galaxy[Galaxy Team, 2008] saves its workflow in a proprietary JSON-like format, which is named as *ga*.

Discussion on Workflow Description

Section 2.3.4 provides an overview of how workflow is orchestrated for the purpose of workflow enactment. The importance of a workflow description language is reflected in both the two major phases and functions as a bridge between them. Considering the situation that all the components described in a workflow are web services, then such a workflow is in effect a composite web service (CWS). Therefore, based on the two major phases noted in 2.3.4, it can be further observed that, the following are important aspects in the design of web services composition language:

1. a description language that can define the execution dependencies among web services.
2. a runtime to support enactment of CWS derived from the description language.

2.4 Summary

Existing MDO frameworks provide different mechanisms for the job of applications wrapping. Some can deploy analysis components as web services, others either do not provide a web-based component or utilise web pages rather than web services for component invocation. A web page is more person-friendly, but lacks machine accessibility. Meanwhile, all of these popularly adopted wrappers are executed locally, which need local installation processes, administrator permissions and most importantly are without/lack remote accessibility. It is also significant that the web services adopted by them are RPC-style, so all the drawbacks of RPC-style are also inherited, which may incur extra hurdles to user experience and data staging.

A new architectural style, in REST, has emerged showing the potential to overcome the disadvantages inherited from the RPC-style. It is also worth exploring and researching the advantages that the new architectural style can bring. The evolution of MDO frameworks represents the demand from scientists that smart methodology should be adopted in the scientific workflow that bears the features of multi-institutions, multi-platforms and geographically distributed resources.

The following chapters in this thesis aim to analyse those issues and propose the a systematic solution as a framework to be applied in the context of scientific workflow, based on ROA. From the perspectives of data staging, service management and service compostion, the improvements based on this framework will be reflected in the following aspects:

1. The framework considers the deployment of web services in a broader context, assuming services will typically be composed. Consequently a data staging mechanism is provided to assist in the effective enactment of composition of services. This is in direct contrast to the above tools, which do not consider data communication as part of their concern, and which can, in the worst case, result in centralised data transfer, when deployed as web services.

2. The framework allow hot-plug style program uploading, deployment and system deployment. In contrast, some of the wrapping techniques, such as Soaplab and Opal, mentioned in this chapter require the wrapper to be installed on the server and work as local tools on a server that needs to be set up and configured every time a new service is deployed. The framework we propose also aims to provide a desktop GUI tool for clients to deploy web services based on command-line programs. This avoids the need to learn and use the description languages adopted in existing tools, as well as the overheads involved in authoring, debugging and maintaining such descriptions in parallel with the application.

3. Based on the new methodology for data staging and service management, this framework will further explore the feasibility of introducing cloud computing into this framework. It will further enable the ability of distributed computation and resources with the supported of enhanced reusability and reproducibility. This aims to overcome the technical hurdles and management complexity. These hurdles and complexity are brought by various legacy code, which require specific execution environments, such as heterogeneous operating systems and software dependencies.

4. The framework also needs to address support for the construction and deployment of composite services. A possible issue is the dependence on specific services, meaning there is a reliance on a service provided at a specific location, as against a specification of a service by, say, its profile (in OWL-S terminology), and the late binding identification of suitable available candidate services close to enactment time. We propose and construct a mechanism that based on the works in this thesis to support workflow description for RESTful web services based composite service, web service composition and composite service execution.

Chapter 3

Datapool: A ROA Based Distributed Data-flow Model

3.1 Introduction

The problem of staging data in workflows has received much attention over the last decade, with a variety of user-directed and automatic solutions. In Section 2.3.1, the existing solutions and researches were introduced. A trend of distributed management of data and separation of data-flow and control-flow emerges, which aims to resolve the performance overhead when the two flows are embedded within each other. The first challenge in the design for this ROA based WMS emerges as, how to design a data staging mechanism that can utilise the light-weight feature of less layers of protocol, follow the constraints of ROA and also enable the data staging in a distributed manner with separation of data-flow and control-flow. This chapter addresses this challenge. The resulting data-flow model has following features:

1. The model is designed based on mature and widely applied technology. It does not use a private – by which we mean internal, or closed – mechanisms (functions are exposed by a set of developer defined specific interfaces and operations) to handle data transfer.

2. The model supports universal access through network to data objects, which are viewed as online resources. By this, this mechanism enables the access to provenance data.

3. The model provides a seamless interface for application services to access individual data objects or a collection of data objects.

4. In this model, data-flow is separated from control-flow to avoid central coordination, which forms the basis for distributed data transfers.

5. It supports a distributed data staging mechanism. Data-flow can be distributed among services without passing through a central controller.

6. It supports asynchronous data staging. Data transfers can be initiated asynchronously before the actual execution of a service when data comes from different sources.

The rest of this chapter will first provide a detailed design of this model as well as its comparative advantages in Section 3.2; then implementation of this model based on ROA in Section 3.3; evaluation of performance is provided in Section 3.4; there will be a summary in Section 3.5. Furthermore, there is a discussion of limitations in Section 8.2.1.

3.2 Distributed Data-flow

Inspired by the principle of distributed data transfer, this thesis presents a data-flow model for web services composition, which is able to alleviate the communication bottleneck between client and server. There are two quite obvious remarks about data-flows among multiple services: (i) for a given service invocation, the data-flow rarely involves the client or central controller, which means that data-flows can (normally) be distributed, and (ii) it is not uncommon that the necessary data objects (inputs) may come from different sources, suggesting that data transfers can be initiated asynchronously before the actual execution of a service. These constitute the two properties that our data staging mechanism needs to satisfy.

3.2.1 Distributed Data Staging

Figures 3-1 and 3-2 illustrate the essential difference between a centralized and a distributed mechanism for data transfer. Figure 3-1 shows that both control-flow and data-flow are centrally coordinated for each web service invocation. There is a high

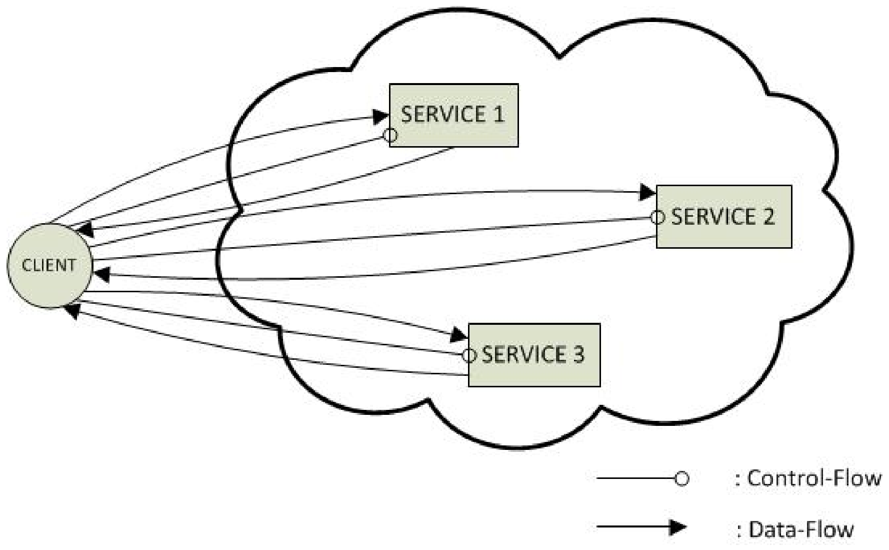


Figure 3-1: Centralized Data-Flows in Web Services Composition

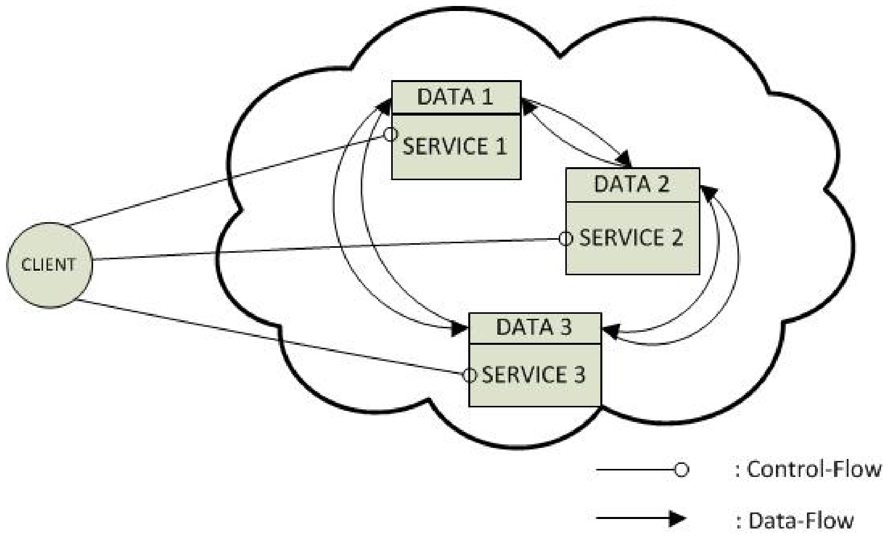


Figure 3-2: Distributed Data-flows in Web Services Composition

risk that the client or central controller becomes a bottleneck for data communication among computation components. In Figure 3-2, the data-flows are distributed among web services directly rather than passing through a central controller, which also allows for the concurrent transfer of data objects from different resources. The I/O data communication carries out in network between each related web service. The client can also obtain the complete set of data objects from the network whenever it needs.

Hence, each service provider takes care of the task of data storage instead of the client.

This comparison also illustrates the additional technical requirements on the distributed data staging mechanism. Two key functions are identified: (i) an universal addressing system that allows each individual data object can be identified and accessed in network, (ii) an server-side data management service that allows data object to be stored, grouped and accessed based on that universal address system. Chapter 2 showed that some other distributed data transfer system to achieve these two requirements either through the use of an extra layer of private protocols, above the Internet application layer protocols, or the introduction of a potentially complicated (internal) system to achieve distributed data transfer. This thesis aims to present that an open solution can be achieved by simpler web services based architecture like ROA and in which data objects can be addressed just by URIs. More design details are presented in Section 3.3.

3.2.2 Asynchronous Data Staging

As aforementioned, it is common that necessary data objects come from different sources. In a centralized way, control-flow and data-flow are centralized coordinated through client. Therefore, the inputs from different sources for next service have to be synchronized at client in advance. In an arbitrary web service, it is common to have multiple data objects compacted into one data object, such as XML, JSON file, which naturally makes the data staging among services working in a synchronous style. On the other hand, in REST-compliant web services, it is also common phenomenon that multiple data objects are compacted together into a data structure, which is either contains in URI or HTTP body data.

The comparison between asynchronous and synchronous data staging mechanisms are demonstrated in Figure 3-3, where there are three data objects from three different web services that need to be transferred to another service as inputs through three different connections. We make the not entirely realistic assumption that in the two situations, the same data object transfer takes the same time. Under synchronous data transfer, because the data references are controlled through the client, data transfer only starts when the last service finishes and the next service invocation happens. However, in the asynchronous method, the transfers start asynchronously when each previous service finishes. The transfers are not synchronized with the invocation of next service.

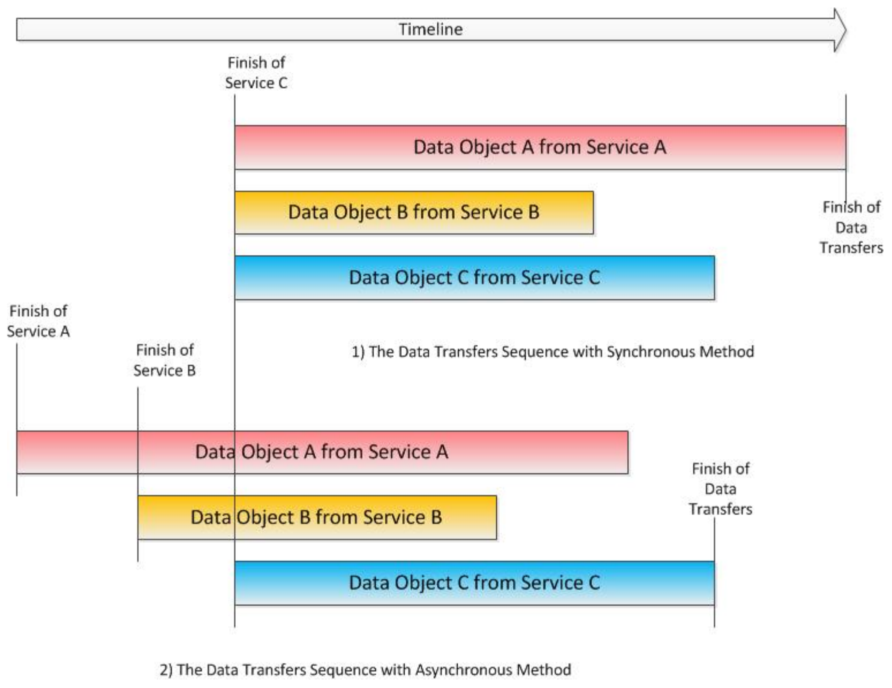

Timeline
Finish of Service C
Data Object A from Service A
Data Object B from Service B
Data Object C from Service C
Finish of Data Transfers
Finish of Service A
Finish of Service B
1) The Data Transfers Sequence with Synchronous Method
Data Object A from Service A
Data Object B from Service B
Data Object C from Service C
Finish of Data Transfers
2) The Data Transfers Sequence with Asynchronous Method

Figure 3-3: Comparison of two different data staging mechanisms

Data objects will be transferred in advance. From Figure 3-3, we can clearly see that the asynchronous method may be able to bring about an earlier completion of the whole data transfer process.

The advantages of asynchronous data staging are mainly embodied in the enactment of workflow, such as multiple steps in sequence. This is shown in Figure 3-4 by a simple workflow data staging example. In this example, there is a data dependency that Service 3 depends on both Service 1 and Service 2 directly. D2 is generated after Service 1's execution. In synchronous data staging mechanism, D2 has to wait for Service 2's execution and is submitted to Service 3 together with D4. In asynchronous mechanism, D2 can be transferred to Service 3 in advance in spite of the execution of Service 2. Obviously, with the increase of number of sequential steps in workflow, there are higher possibility that the overhead on data staging can be saved by asynchronous mechanism.

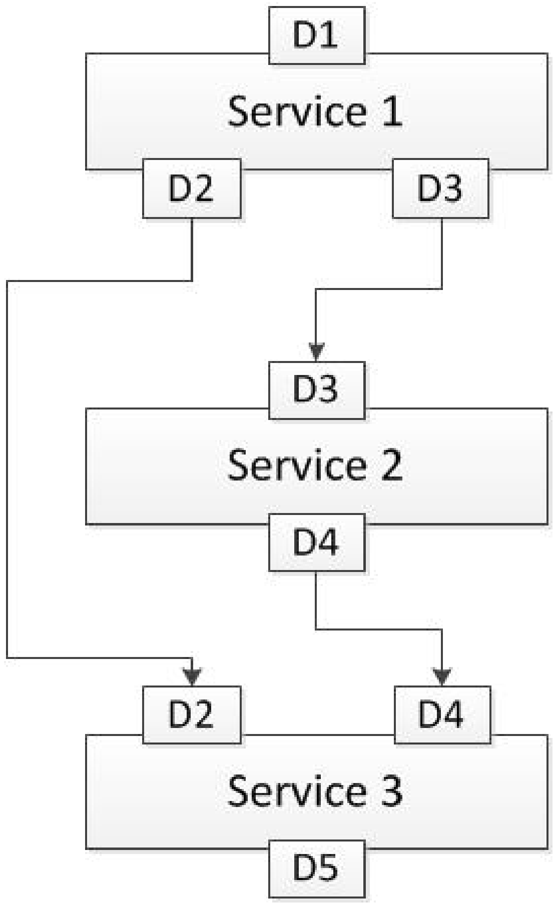


Figure 3-4: The data dependency in a simple example of multiple steps of workflow

3.3 The Implementation of Distributed Data-flow Model

In order to implement the distributed data-flow model in this framework, there are several basic design requirements to be considered. First of all, the addressability of data objects. By following the design principles of ROA and the URI protocol, we are straightforwardly able to achieve the requirement of enabling data objects being identified and accessed universally on Internet. Secondly, it is also required for users to be able to deploy services that can be joined up via this implementation, without the need to take any special action or need to modify existing applications. Finally, it is also a requirement that an appropriately authorized user should be able to manipulate only their own data objects, in order both to secure a given workflow's data, but also to avoid unintentional conflicts with other users.

3.3.1 Datapool Service

The Datapool service is the RESTful web service for I/O data items manipulation (uploading, retrieval, etc.). Datapools are components that used to connect data with application services, which implement the distributed data-flow model. Each Datapool represents a collection of data items each with a unique URI. Each data item inside one Datapool is allocated with unique URI to enable *addressability* as well.

There are two advantages to organising data in this way. First, because all the data items and data collections are directly allocated with URIs, they are all web resources that can be re-accessed through HTTP at any time rather than merely a data stream in the form of an extra layer of XML or other structure. Furthermore, each data item can also be transferred and kept in their original textual or binary format. Second, in the execution of an application service, the URI of one Datapool that contains all the input data is provided to the service. The application will pull the necessary data automatically from the provided local or remote Datapool. In this way, the interfaces are unified for different application services in the form of one URI, of which the Datapool URI is a constituent as a query string.

Distribution of data between services is greatly simplified through the use of URIs. Each Datapool service is collocated with application services in the same host as shown in Figure 3-2. Hence the basic function of the Datapool is to provide data storage for association with a particular service, whereby data objects are uploaded into a particular Datapool, such as during initialization, or are transferred from one Datapool to another as a result of the control-flow. In each case, the object will be given a name derived from the Datapool's URI.

Asynchronous data staging is also achieved by the existing of Datapool. We identify that the essence of asynchronous data staging is the separation of control-flow and data-flow. In our case, data-flows among services can be totally controlled through Datapool service. Control-flow merely needs to pass through the application services that associated with the Datapool service which is collocated in the same host. Each data object can be submitted to or retrieved from Datapool service independently from control-flow.

	Methods	URIs
Datapool Services	PUT	http://.../datapool/{Datapool_Instance_Name}/{Data_Object_Name}
	PUT	http://.../datapool/{Datapool_Instance_Name}?DO_URI={Data_Object_URI}
	GET	http://.../datapool/{Datapool_Instance_Name}/{Data_Object_Name}
	GET	http://.../datapool/{Datapool_Instance_Name}
	DELETE	http://.../datapool/{Datapool_Instance_Name}/{Data_Object_Name}
	DELETE	http://.../datapool/{Datapool_Instance_Name}

Table 3.1: URIs of Datapool Service

3.3.2 User Interface

For the purpose of enabling data being transferred from client to Datapool or between two Datapools, firstly, it exposes an interface for users to inject documents of any format into the repository as either textual data or arbitrary binary data. Datapool service automatically allocate URIs and organize them in the form of a collection of URIs. Secondly, clients can submit the URI of the data object to Datapool. Datapool will automatically obtain and allocate a new URI to it. Afterwards, before the execution of service, client merely need to provide the URI of the corresponding local Datapool to web service, which contains all the necessary inputs. Web services will automatically extract and consume the data it needs based on data object's name. The whole interface is built based on the HTTP protocol to allow the *uniform interface* of ROA.

The services of Datapool are accessible through URIs as shown in Table 3.1. Users can maintain several Datapool instances for different processes or for different stages in one process, and only the owner of the Datapool instance can control the data objects stored in it and decide who can access them, which provides a degree of security for user data. Data resources in the Datapool instance have unique URIs so that different users cannot unintentionally conflict each other.

The HTTP methods - GET, PUT, DELETE - are used in these URIs to represent corresponding functions of the Datapool service. The Datapool service aims to provide following functions: (i) *create*, *update*, *retrieve*, *delete* of Datapool instances, (ii) *create*, *update*, *retrieve*, *delete* of data object in Datapool instance. The number of functions can be further reduced to simplify the method vocabulary. The *update* job of Datapool instance actually equals the *create*, *update* and *delete* jobs of data objects. The *create* job of Datapool instance is automatically activated by the *create* job of data object in a new Datapool instance. There is no necessity to create an empty Datapool instance. Meanwhile, the *create* job of data object can share the same method of *update*. The Datapool system can distinguish them by checking if the requested data object is existing. Therefore, there are five jobs that needed to be achieved by the those URIs and the HTTP methods. (i) *retrieve*, *delete* of Datapool instance, (ii) *update*, *retrieve*, *delete* of data object in Datapool instance.

The first two PUT methods in Table 3.1 are used for *update* job of data object. First is used for uploading raw data directly. Second is used to upload from another Datapool by providing the URI of data object. The first GET method in Table 3.1 is used to retrieve the raw data of the data object. The second GET method is used to retrieve the list of the data object URIs in the retrieved Datapool instance. The last two DELETE methods are used to delete the corresponding data object or the whole Datapool instance.

3.3.3 Usages and Technical Details

Datapool Service Datapool service is deployed in each of the machine where applications reside in. This Datapool service is always associated with the data staging activities of the applications in the machine it is deployed. When there are multiple machines, there will be same number of Datapool services deployed on those machines, one for each. Cross-machine data staging happens between two Datapool services in two different machines. In Table 3.1, the URI prefix *http://.../datapool/* represents Datapool Service. Datapool Service can contain multiple Datapool instance, where the data object are categorised based on their usages and owners. At this level, only the system admin have the permission to do configuration and manipulation. In the OS file system of the Datapool service, their will be a designated folder to hold all the data instances as sub-folders. The path of the designated folder can be customised in the configuration file for the Datapool service.

Datapool Instance Datapool instance is the data collection in Datapool service. There can be multiple Datapool instances in one Datapool instance. Each of the Datapool instances can have associated data owner or group. Only the owner or the member in the group has the access to the data objects in the corresponding Datapool instance. Contrarily, one owner or one group can have multiple Datapool instances in one Datapool service for different usages. For example, Datapool instance X can be used only for Service X's data staging activities and instance Y is only used for Service Y. In Table 3.1, the URI prefix *http://.../datapool/*{*Datapool_Instance_Name*} represents the Datapool instance. Each of them can have it own unique name. As mentioned, each Datapool instance appears as sub-folder in the designated folder for Datapool Service in OS file system. Their folder name is the same as the *Datapool_Instance_Name* appeared in the URI.

Data Object in Datapool Instance Data object is the real raw I/O data from applications. Their accessibility is decided by the Datapool instance where they are put in. Only the Datapool instance owner or owner group can access the data objects. In the OS file system, they are saved in their raw formats in each of the corresponding Datapool instance sub-folder. When there is data staging activities happen between two Datapool instance in the same machine, the real background activity in the OS is just to change the original path of the file to let it points into the new Datapool instance sub-folder. If it is cross-machine data transfer, the original data will be deleted, and a new file will be created in the new sub-folder in the other machine OS. In Table 3.1, *Data_Object_Name* in one complete URI *http://.../datapool/*{*Datapool_Instance_Name*}/{*Data_Object_Name*} means the name of the Data object. It is also the file's name in the Datapool instance's sub-folder.

3.3.4 Authentication and Authorization

Authentication and authorization are key functions that should be applied to the Datapool system. One of the important reasons of applying ROA is that RESTful web service enforces security based on the HTTP/HTTPS protocol. There is no necessity to implement an extra security layer for such purpose. Based on that, an authorization system is implemented to enable a role-based control authorization. There are two obvious benefits brought by it: firstly, developers do not need to design and embed a new security mechanism into the system, which can save developing time and reduce knowledge preparation; secondly, users also do not need to extra knowledge to get familiar with the security mechanism as well as do not need to install extra layer of software or plug-ins to use it, because of the wide application of the HTTP/HTTPS protocol.

HTTPS is a communications protocol for secure communication over a computer network. It simply layers the HTTP protocol on top of the SSL/TLS protocol, thus

	Methods	URIs
Datapool Services	PUT	https://.../datapool/{Datapool_Name}/{Data_Object_Name}
	GET	https://.../datapool/{Datapool_Name}/{Data_Object_Name}

Table 3.2: Secure URIs of Datapool Service

adding the security capabilities of SSL/TLS to standard HTTP communications. By applying HTTPS, the data object that is transferred among services and between service and client can be encrypted and protected. This does not lay any extra burden to users. The only difference from the point of view of operation is that the URI used for data manipulation starts with *https* rather than *http*. Table 3.2 shows two examples that used to PUT and GET data object.

Multiple Datapool instances can be generated and customized through the Datapool service by multiple users. Necessary authorization policies are applied to Datapool service to regulate the operations of users. The user of this framework, which includes Datapool service, is enforced to register a pair of username and password. The principle of the policies is only the creator the of Datapool instance or anyone who is given the authorization by the creator has the access to it. This is applied to all the operations: *update*, *delete*, *retrieve*. For example, in the case of the second URI is invoked for data object uploading in Table 3.1. The user has to make sure that he/she does not only have the access to the Datapool instance where the data object is uploaded to, but also the access to the other Datapool where the data object is stored.

3.4 Evaluation

In order to evaluate the performance of services deployed using the new framework, a wing optimization process workflow, which is written in Taverna, (more details about this workflow are introduced in Section 7.1) is executed in two network-based ways: (i) with all the programs deployed as SOAP services and controlled through a centralized client, including all the data transfers, constituting in effect a worst case scenario for transfer overheads, and (ii) with the programs deployed as REST services, using a centralized client for control, but the distributed data-flow model for data. Firstly, these two modes are compared, where the programs or services are executed in the same machine environment and the network environment is the same.

To provide preliminary evidence that the REST web services with distributed data-

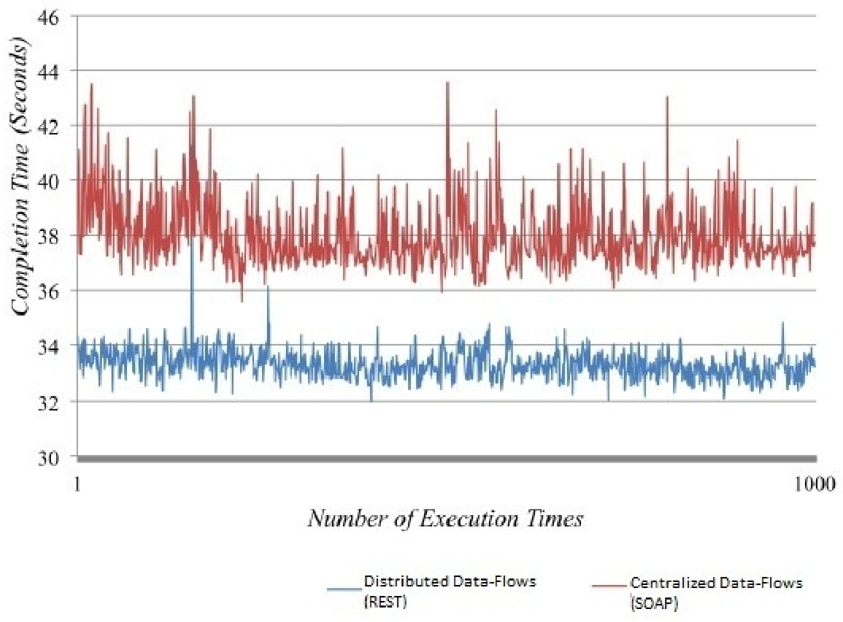


Figure 3-5: Comparison of 1000 continuous executions

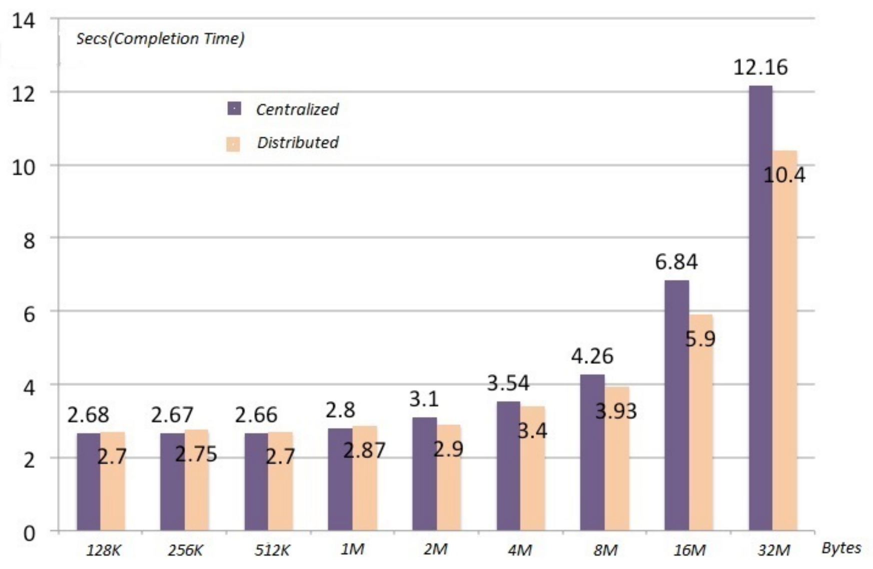


Figure 3-6: Results of simple workflows work with both Centralized and Distributed Data-flows

flows performs better than the centralized approach, we ran an experiment of 1000 consecutive executions for both processes in the same environment with workflows written in Taverna. The result is presented in Figure 3-5[1]. There is fluctuation caused by the real world scenario, but the figure shows that the REST workflow is faster by a clear margin. In order to show a comparison between a centralized method and a distributed method in the same situation, we wrote a workflow in Taverna based only on RESTful services that just moves data from client to one service, on to another, then back to the client. These two services are distributed in two different VMs on the same LAN as the client. Client and services access each other by URIs. The data-flow between these two services does not go back to the centralized controller - client. It only exists between those two distributed services.

We set up two scenarios both using RESTful services. The first scenario uses centralized transfer, which means the data-flow goes back to client when data is transferred from one service to the other. The second scenario uses the distributed method. In first scenario, the data transferred from the first service to the second is included in the HTTP body, while in the second just the URIs are transferred and data is transferred in the background by Datapool. The results are shown by Figure 3-6. Each workflow was run 10 times for the two scenarios and different data sizes to obtain the mean value. The results suggest an expected trend, in that gains increase with the size of data to be transferred.

3.5 Summary

This chapter presented a systematic mechanism for a ROA based distributed data-flow model. It describes the design and implementation of a distributed data-staging mechanism, that itself is RESTful, by following REST design principles for the support of data-intensive (RESTful) workflows. It is evaluated by means of an engineering workflow developed for multi-disciplinary design optimization. The workflow is specified in Taverna, which is a conventional centralized data-staging enactment system. However, by virtue of the underlying services and staging mechanisms described here, the resulting enactment is distributed, which furthermore permits asynchronous staging, with benefits for network utilization and end-to-end execution time. Therefore, the

[1]The x-axis only denotes the number of the run: it does not signify concurrent execution of the two modes. The data from the two sets of runs is overlaid to facilitate comparison of the execution times.

contributions of this work can be summarised as:

- It provides a data staging mechanism for ROA based distributed system that it utilises the features and follows the design constraints of RESTful web service without introducing extra layer of protocol.
- This data staging mechanism has the ability to separate the data-flow and control-flow in workflow enactment. By having the asynchronous feature, it is also able to further separate the data-flow based on the data source and data destination. Also, this mechanism can provide better data staging performance by its distributed and asynchronous features.

Chapter 4

SaaS based Scientific Web Services Management

4.1 Introduction

For at least the past decade, a significant trend in scientific research, which includes significant amount of computation for the purposes of modelling, simulation, verification and evaluation, has been the increasing uptake of computational techniques (modelling) for in-silico experimentation. This trend trickles down from the grand challenges that require capability computing to smaller-scale problems suited to capacity computing. Such virtual experiments also establish an opportunity for collaboration at a distance. At the same time, we believe the development of web service and cloud computing technology, is providing a potential platform to support these activities. This defines the general topic and issues we aim to study in this thesis. In Chapter 2, a series of issues are identified on existing web service deployment toolkits. The problem is the technical hurdles for users without specific and enough knowledge about web services, workflow and other related mechanisms – in a word, 'accessibility'. Specifically: (i) the heavy weight and diversity of infrastructures that inhibits shareability and collaboration between services, (ii) the relatively complicated processes associated with deployment and management of web services for non-disciplinary specialists, and (iii) the relative technical difficulty in packaging the legacy software that encapsulates key discipline knowledge for web-service environments.

The aforementioned hurdles are determined by the nature of the end-user-scientist and the resources that need to be deployed in the cloud. Most scientists who have limited knowledge of web services or cloud infrastructure may need to face the need to learn new programming languages or system administrative skills for the purpose to build scientific applications in the cloud or as web services. For example, an engineer possibily has the skill to develop desktop applications based on Fortran or Matlab, but rarely has knowledge of or experience with web application development based on languages like Java or Python. On the other hand, with years of development, numerous legacy programs in which real domain-specific knowledge resides, may face the predicament that a new round of coding and translating work is needed or they simply lose the ability to be re-developed because of the lack of source code, documents or language support[1].

Therefore, this chapter aims to propose a web services management framework that makes differences from following aspects:

1. A Software-as-a-Service (SaaS) cloud model is applied to provide the function of *service deployment as web services*, which allows a hot-plug style of program uploading and deployment. The existing methods for service deployment assume the programs or the libraries to support the service have been pre-installed on the server. They work as local tools on a server that needs to be set up and configured by system administrators every time a new service is deployed.

2. The deployment of web services is considered for a certain demand, which assumes services will be composed as one web service. Consequently, a data staging mechanism - Datapool, which was proposed in Chapter 3 is provided to assist the composition of services. The web service management mechanism in this chapter will inherit the interfaces and be able to communicate with Datapool without further configuration from users. The existing tools do not consider the ease of use for service composition. Data staging among services have to be orchestrated by a third-party application through a centrally controlled data staging space, which may result in the extra data traffic mentioned in Chapter 3.

3. A desktop GUI tool is provided for clients to deploy web services based on command-line programs. This avoids the need to learn and use the description languages adopted in these tools (such as "*Ajax Command Definition*" in

[1]In the worst case, only a binary of the program may exist, which happens to be executable due to backwards hardware compatibility.

Soaplab, “*serviceMap*” in GFac and “*Metadata*” in Opal), as well as the overheads involved in authoring, debugging and maintaining such descriptions in parallel with the applicaiton.

To be compatible and consistent with the Datapool service adopted, the proposed web service management framework adopts REST web services. At the same time, because this framework is also based on REST/ROA, it also inherently has the benefits of flexibility, simplicity as well as the easy-to-use feature for non-specialist users as discussed in Chapter 2. Furthermore, this framework lowers the barriers by providing a set of GUI based client tools and a set of REST web services which serve as both portal for service deployment and service execution by following the SaaS service model[Mell and Grance, 2011]. The framework is able to deploy legacy code and command-line programs as RESTful services, which can support a wide range of languages and tools, such as C/C++, Fortran, Matlab, Python, Unix shell, JAVA, and some engineering design optimization frameworks, specifically OpenMDAO[The OpenMDAO development team , 2010] and Dakota[Sandia National Laboratories, 2010]. In this case, because the framework follows RESTful principles, it can be directly accessed from a wide range of programming languages (such as a command line scripts/applications) or a generic workflow management system (such as *Taverna*[Oinn et al., 2004], see section 4.4) without any additional library support or tools. The services are made into web applications, based on easily obtainable, free, open-source tools, such as Apache-Tomcat and MySQL. Embedded within the framework is a distributed data-flow mechanism, that can enhance data-staging performance in the execution of composite services. Through the desktop GUI tool, inexperienced users can learn about, create, and use web services. We demonstrate the framework operating both in the context of a private server and the Amazon EC2 service, in order to show compatibility with both private and public cloud provisioning. Hence, we believe it should be readily deployable on top of other IaaS services with little change.

4.2 The Wrapping Technology for Applications

This section provides more detailed insight of the wrapping technology for applications. The wrapping technology discussed in this section refer to the technology that to wrap an piece of code or compiled binary to web services. They mainly have two types: local deployed desktop application and online platform. All these platforms will be analyzed based on some essential features to discover the pros and cons. This information has a directional role in the design of the SaaS based online web services management proposed in this chapter. The wrapping tools will be categorized into local service deployment tools and online service deployment platforms.

4.2.1 Local Application Wrapping Technology

In scientific workflows, web services are created and invoked as components. Workflow management systems enables end-users to compose, execute and monitor them in a centralized manner. Taverna[Oinn et al., 2004] is one of the most popular platforms for this purpose. Soaplab [Senger et al., 2008a] and Opal [Krishnan et al., 2006] are two typical local application wrapping tools. Their services can be accessed through Taverna's plugins mechanism. The Kepler workflow management system (WMS) [Altintas et al., 2004] can also invoke services created by these two application wrapping tools. Both of these WMSs need to develop extra plugins by using Soaplab or Opal programmatic APIs for the purpose of invoking web services seamlessly. This indicates that if there is new software, a WMS or any one needs to invoke such web services, it needs a new round of development for this extra plugins. Specific APIs for each scientific applications wrapping tool may place an extra barrier for workflow engine to invoke corresponding web services. For example, both gRAVI [Chard et al., 2009], a WSRF(Web Services Resource Framework) web service wrapping tool and Generic Service Toolkit [Kandaswamy et al., 2006], a web service wrapping tool, each have three ways to invoke services: Java Client API, web interface and command line. But neither provides a method to enable the access from the WMSs aforementioned: a plug-in mechanism is always necessary to handle the interface between workflow and service. Because programmatic access to those services needs specific APIs, this creates a restriction on the programming languages supported. For an instance, Soaplab2 only supports Java and Perl. Likewise with Opal, except the languages are Python and Java [Ren et al., 2010]. It means that if one user wants to work with these tools, he/she does not only need to have certain knowledge of the specific APIs, but also the certain knowledge for a programming language. This can be a hurdle for non-specialist users. Thus, based on the analysis on web services in Chapter 2, we can see two hurdles brought by the existing implementations of local application wrapping tools: *a*) The hurdle of extra knowledge preparation and tools are needed to access local application wrapper based web services; *b*) The hurdle of reduction of interoperability incurred by heterogenous service interfaces. For the purpose of identifying the points that should be improved, more details of local application wrapping technology are categorized as follows:

Soaplab

Originally, Soaplab was designed for bioinformatics applications. However, it can be used to deploy any command-line based application as a SOAP web service. In Soaplab, access to the underlying processes is provided by some generically-named operations (verbs in the protocol), but data is not sent to the server through distinct SOAP endpoints (not distinct URLs for each operation). More precisely, each data object must be packed into a data collection and sent by invoking specific method exposed by the endpoint. These technical details are handled by a mature plug-in in WMS, such as Taverna, and transparent to workflow management tool users. The Soaplab service in Taverna can expose several interfaces in order to let users extract each data item directly. Each output from a service can be directed to all the input ports for following services. However, this is also the reason why specific APIs must be included in order to invoke a Soaplab service, which also applies to SOAP based tools. It is observed that Soaplab does not have any security mechanisms to encrypt data and perform role-based access control over users.

Opal

In Opal [Krishnan et al., 2006], the application services that wrap the scientific applications are accessible via programmatic APIs. In their design, they do not mandate a single user interface to be used by all end-users - instead, the APIs for the services are provided to enable access via a number of different clients. This provides an innovative idea for the design of user interface. However, to deploy the application, every application needs to be described by an application configuration file, which is configured locally in the server. The programmatic APIs based on WSDL and SOAP are generated from this configuration. Significantly, the developers of the scientific applications are required to have some perquisite knowledge on the proprietary configuration files and web services. Opal does however provide a security mechanism for data encryption, authentication and authorization. It is based on the GSI security system that uses the Secure Sockets

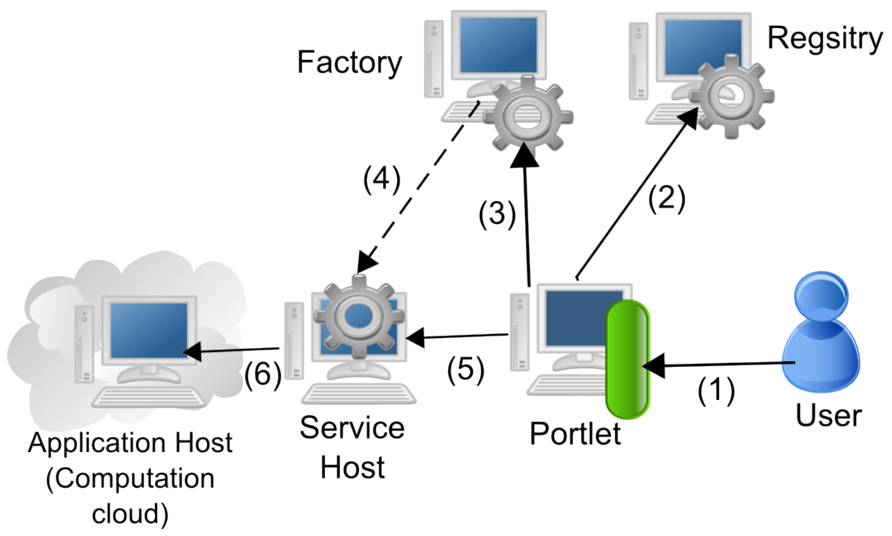


Figure 4-1: GFac Architecture [The Trustees of Indiana University, 2006]

Layer (SSL) for its mutual authentication protocol.

GFac

The latest version of GFac implements a more sophisticated architecture compared with the previous two tools. Figure 4-1 depicts the architecture of GFac. It contains a series modularized services, which brought the benefits of enabling distribution of computing resources. Naturally, this raises the complexity of deployment, configuration, management for the system administrator. However, it has novel architectural designs can significantly reduce the burden on users and system administrators. For example, it provides a portal service which can reduce the difficulty to services description and deployment. The design of separation of application and data hosts is able to further reduce the system's redundance. However, in their data staging design, data is not necessarily resided within the application and it has no separation of control-flow and data-flow. Fundamentally, the whole system is built based on SOAP+WSDL web service development tool stack. It unavoidably brings in the features of arbitrary web services. Purely form a user point of view to observe that this system still need a private programmatic APIs, plug-in or tool to invoke the web services deployed by GFac [The Trustees of Indiana University, 2007].

4.2.2 Online Application Wrapping Technology

With the development of Cloud technology, it is not surprising to observe the trend that various Cloud infrastructures or platforms start to provide the service for hosting of web applications in an on-demand manner. People start to utilise the advantages of shareability, ability to easily migrate and scale services for their web applications. In this section, the technology related to wrapping application as services through Cloud platforms will be discussed in order to provide a more complete perspective on the development trend of wrapping technology for web applications.

Cloud computing is commonly categorized into three service models[Mell and Grance, 2011] known as {Infrastructure, Platform, Software} as a Service (IaaS, PaaS and SaaS, respectively), of which PaaS is the service model that provides the consumer with the capability to deploy consumer-created or acquired applications onto infrastructure, thus creating an instance of a service. IaaS aims to provide a Cloud infrastructure such as a Virtual Machine (VM), whereas SaaS aims to provide services without the capacity to deploy consumer-created or acquired applications. Based on the definition, strictly speaking, the framework proposed in this chapter sits in the middle of PaaS and SaaS. On one hand, it provides services with simplicity to enable users to create their application services. On the other hand, it does not give complete control on software dependencies in the OS to be deployed. This chapter mainly proposes the methods on how to enable an easy-to-use framework to wrap and deploy application services. The topic about giving more controls on software dependencies based on this chapter will be discussed in Chapter 5.

Nowadays, there are plenty of commercial and free Cloud vendors online, which can provide multiple levels of quality and various functions. The functions basically are the applications and programming languages they can support. This section selects some of the typical, well-known platforms to give an brief understanding on the advantages/disadvantages of existing wrapping and deployment technology for web applications.

Heroku [Lindenbaum et al., 2008]

Heroku is one of the first PaaS vendors in market. It provides its own hardware infrastructure. Like most of the other PaaS vendors, it mainly supports web applications to be deployed onto its supported platforms, such as Ruby, Java, Node.js, Scala, Clojure, Python, (undocumented) PHP and Perl. Applications are deployed through commend-line client tool based on Git server for data push. There is no indication that it can support command-line programs to be deployed as web services directly, or written in languages other than aforementioned ones.

Google APP Engine [Google, 2008]

Google APP Engine provides the means for users to develop and host web applications in Google-managed data centers. It works via programming language APIs, such as Python, Java, Go, and PHP. For the purpose of deploying an application into their infrastructures, users must either write applications in specific languages or modify original codes in those languages. Along with it, there are also a list of restriction on the applications in Google APP Engine, such as (i) developers have read-only access to the filesystem on App Engine. Applications can use only virtual filesystems, like GAE(Goolge App Engine)-filestore. (ii) App Engine can only execute code called from an HTTP request (scheduled background tasks allow for self calling HTTP requests), etc. Compare with other application hosting, App Engine provides more infrastructure to make it easy to write scalable applications, but can only run a limited range of applications designed for that infrastructure, to say nothing of a legacy command-line programs or codes.

Cloud Foundry [GoPivotal, Inc., 2011]

Cloud Foundry is an open source cloud computing platform as a service (PaaS) software. The main difference from the others is that it can be used to build PaaS host either public or private without a binding with certain infrastructure provider. For example, AppFog [AppFog, Inc., 2013] is built based on Cloud Foundry based on the IaaS provided by Amazon Web Service(AWS). As well as being an open source software project, Cloud Foundry is also a hosted service offered at cloudfoundry.com. It supports Java, Ruby and Node.js. In AppFog, it can further support PHP, Python, etc. On the basis of supports to applications, there are not much difference from aforementioned PaaS vendors. It needs sophisticated programming and computer skills for the purpose of deploying services.

Openshift [Red Hat, Inc., 2011]

Openshift is yet another PaaS product from Red Hat. It supports JavaScript, Ruby, Python, PHP, Perl, Java language environments. The main difference from the other PaaS products is that OpenShift also supports binary programs that are web applications, so long as they can run on Red Hat Enterprise Linux. This allows the use of arbitrary languages and frameworks. If the binary programs is merely a local application, it cannot be accessed as web services or applications.

Generic Worker

Generic Worker framework [Simmhan et al., 2010] has similar goals to the framework presented in this thesis. It aims to bridge the gap between desktop and the cloud for eScience applications. It is an implementation for Windows Azure that eases deployment, instantiation, and remote invocation of existing .NET applications within Azure without changing their source code. It enables multiple .NET methods and dependency libraries to be registered as applications and dynamically downloaded into a virtual machine on demand when a request to run an application is received. There are also multiple ways to invoke the registered applications by means of a command-line client, .NET APIs, and a Trident workflow activity.

Other Services

There are also plenty of other PaaS products, such as AWS Elastic Beanstalk [Amazon Web Services, Inc., 2011], which are quite similar to Heroku. Openshift Origin [Red hat, Inc., 2011] is released as an open source software project as Cloud Foundry. There are not only PaaS products hosted on AWS IaaS, but on Openstack [OpenStack project, 2012] (Stackato [ActiveState Software Inc., 2014]) as well as on Google (Jelastic [Jelastic, Inc., 2011]), etc. Besides, Microsoft Windows Azure [Microsoft, 2010] is totally built based on the Windows platform. However, all of these features do not make them out of the range covered by aforementioned typical PaaS products.

Both related local and online application wrapping technologies are analyzed in this part. It can be observed from two perspectives about the trend of development, which has significant guidance to the design of the framework for Cloud Based Online Web Services Management. They are architecture design and user experience. The improvements should focus on: (i) The framework architecture should be built on a distributed manner, not only among computing resources, but also between application and data. (ii) A user friendly interface should be provided. It is better to be able to clearly distinguish the difference among the roles of administrator, developer and user.

Functionally, the aim is to provide access to on-line services so that regular users can deploy their own (command-line) applications as services, share them with others and utilise them in service workflows. Thus, this can be achieved through the provision of a platform that provides: (i) deployment services, and (ii) data storage and transfer services.

4.3 SaaS model based Web Service Deployment and Management Mechanism

4.3.1 Service Deployment

The scientific applications which serve as the understructure for the web services must be uploaded and registered with the framework before they are available for invocation and execution in the cloud. There are three tasks at this stage: (i) to upload and store the application and its dependencies in the cloud repository, (ii) to write and upload the description of the application to cloud for subsequent configuration and deployment, (iii) the configuration of authorization information that controls who may access the service once deployed. These tasks are all performed through the client GUI tool. More details of the GUI are presented in Section 4.3.3.

The basic operation is that the service provider can upload relative files of the application and its configuration file through RESTful web services. The configuration file, which includes the description of the service, is written in the form of XML for the purpose of machine processing. Two examples are presented in Figure 4-2. The configuration file is designed for application command generation, service interface generation and service permit initialization. This series of operations simplifies the process of resource deployment and execution. Because the tools used to deploy web services are web services as well, uploading of application related data can be carried out remotely and under the control of the user rather than local system administrator. All of the entries in the XML, such as service name, application command, I/O setting, have their representation in the GUI tool. They can be generated automatically from the GUI tool.

Lastly, users also need functions to remove, modify or redeploy the service from the local machine, which requires the service description. During the deployment process, the description – represented as a XML file – is uploaded as a cloud resource.

```
<WS_Definition>
  <Service_Name ifPublic="no">Aerosolve</Service_Name>
  <Application_Name>aerosolve</Application_Name>
  <Inputs>
    <Input name="aero_input_dat"/>
  </Inputs>
  <Outputs>
    <Output name="aero_output_dat"/>
    <Output name="aero_forces_dat"/>
  </Outputs>
  <STDOut name="stdout"/>
</WS_Definition>

<WS_Definition>
  <Service_Name ifPublic="yes">Instal2asp</Service_Name>
  <Application_Name>instal2asp</Application_Name>
  <Inputs>
    <Input name="DomainFile" qualifier="-d"/>
    <Input name="IAL"/>
  </Inputs>
  <Outputs></Outputs>
  <STDOut name="lp"/>
</WS_Definition>
```

Figure 4-2: Configuration files of application services

At the same time, a copy is stored in a designated local folder, which aims to reduce the data traffic when user wants to modify the configuration. Users can start the GUI tool from the menu obtained by right-clicking on the description file to access the deployment functions.

4.3.2 Service Invocation and Execution

Table 4.1 shows all the URIs of the two types of services. Datapool services are the services for I/O data item manipulation (uploading, retrieval, etc.), which has be introduced in Chpater 3. Application services include the services for application service deployment and execution. Uniform methods based on the HTTP protocol are allocated to each URI for each specific operation. For example, the first and third service in the application services list have the same URI, which denotes one appli-

	Methods	URIs
Datapool Services	PUT	http://.../datapool/{Datapool_Name}/{Data_Object_Name}
	PUT	http://.../datapool/{Datapool_Name}?DO_URI={Data_Object_URI}
	GET	http://.../datapool/{Datapool_Name}/{Data_Object_Name}
	GET	http://.../datapool/{Datapool_Name}
	DELETE	http://.../datapool/{Datapool_Name}/{Data_Object_Name}
	DELETE	http://.../datapool/{Datapool_Name}
Application Services	PUT	http://.../APP_service/{Service_Name}
	GET	http://.../APP_service/{Service_Name}?DP_URI={Datapool_URI}
	DELETE	http://.../APP_service/{Service_Name}
	GET	http://.../APP_service/Service_Info/{Service_Name}

Table 4.1: URIs of Datapool and Application Services

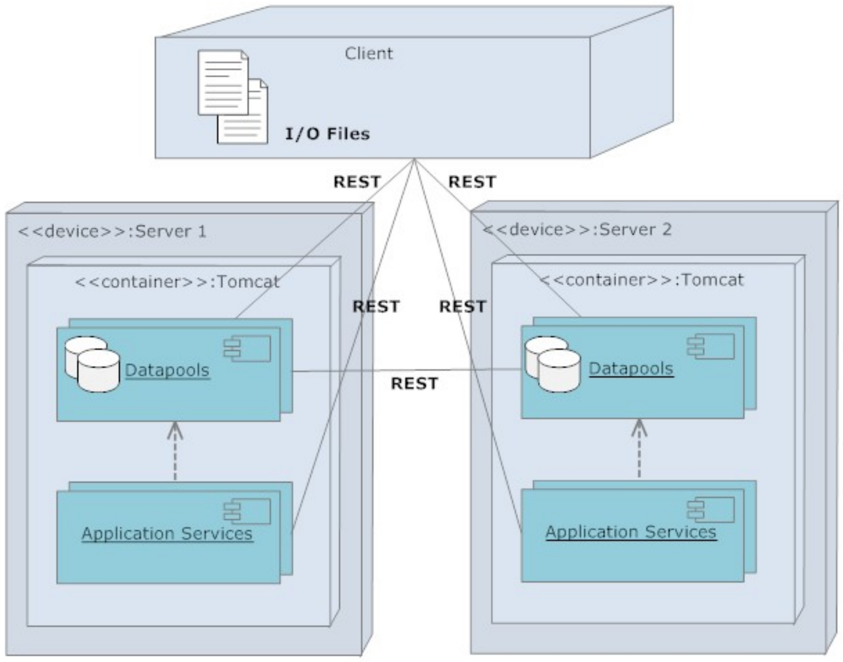


Figure 4-3: UML Deployment Diagram of the Framework Deployment Example

cation resource. The PUT method denotes a service deployment/update operation, while DELETE denotes a service removal operation. These services also support a role-based authorization system so that only an authenticated and authorized user can access those services. Authentication is carried out over HTTP and communication can be further encrypted and secured by HTTPS through the Transport Layer Security (TLS) protocol.

Figure 4-3 shows an example deployment using the framework. It contains one client and two servers. Each server is composed of a pair of a Datapool and an Ap-

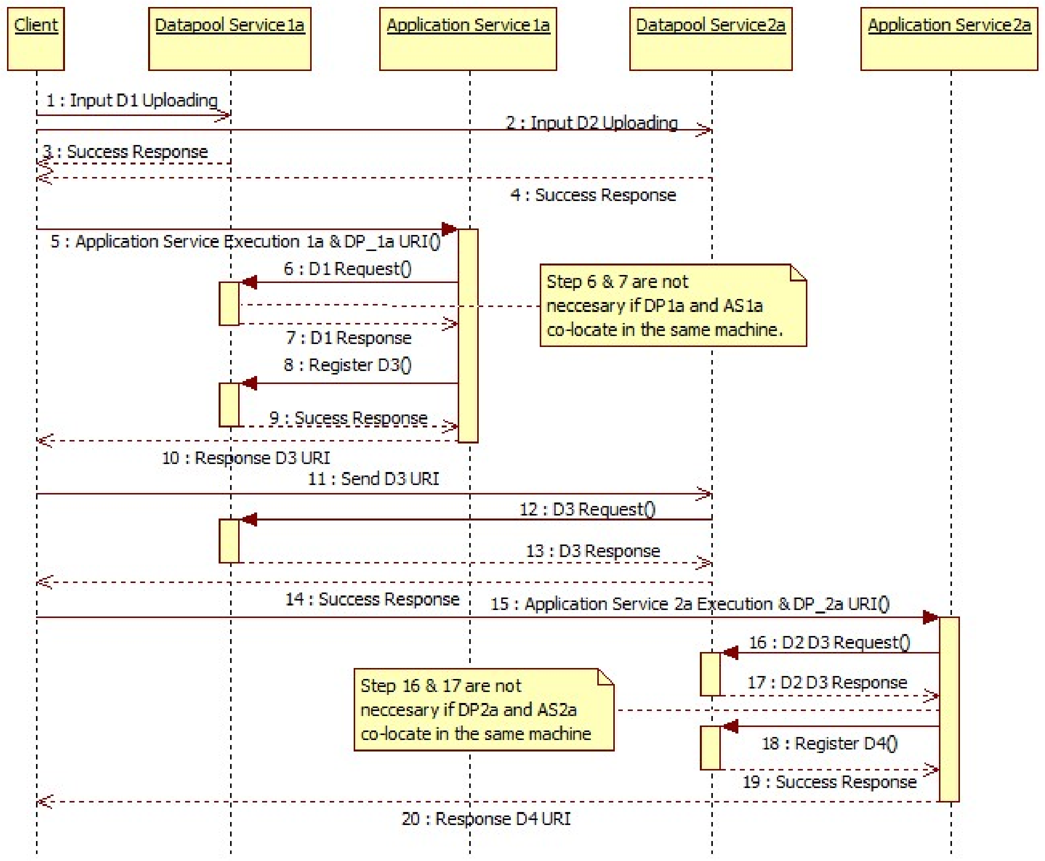
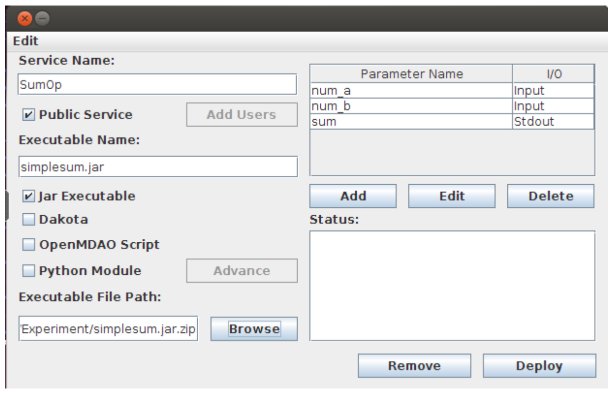


Figure 4-4: UML Sequence Diagram of the Execution of Workflow Example

plication service, both of whose implementation is based on Apache-Tomcat. All the components communicate with each other through REST services invocations. The execution of application service depends on the data provided by its local Datapool, which are fed through a file system. Figure 4-4 shows more details about the execution sequence in an example workflow based on the framework in Figure 4-3. In this example, Application Service 1a(AS1a) consumes input D1. AS2a needs D2 and D3, which are the output generated by AS1a, as inputs. As depicted, client's duties are simplified to initializing input data and dispatching control signals to Datapool and Application Services. There are two essential features here, namely: (i) inputs are uploaded to Datapool separately and in advance, so that Step 1 and Step 2 are able to execute concurrently (ii) DP2a can retrieve the input directly for AS2a in Step 12 and 13 from the other Datapool service without data needing to pass via the client.



Service Name: SumOp

[✓] Public Service

Executable Name: simplesum.jar

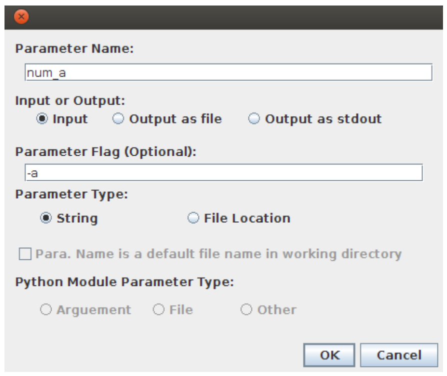
[✓] Jar Executable
[ ] Dakota
[ ] OpenMDAO Script
[ ] Python Module

Executable File Path: Experiment/simplesum.jar.zip

Parameter Name	I/O
num_a	Input
num_b	Input
sum	Stdout

Status:

(a) The main window of GUI tool



Parameter Name: num_a

Input or Output: [✓ Input] Output as file Output as stdout

Parameter Flag (Optional): -a

Parameter Type: [✓ String] File Location

[ ] Para. Name is a default file name in working directory

Python Module Parameter Type: Arguement File Other

(b) The parameter window of GUI tool

Figure 4-5: Windows of GUI tool

In Figure 4-3, the integration of the Datapool and the Application Service is also demonstrated. They are both developed in Java and Jersey library as REST web services. In particular, each Datapool denotes a collection of data items, addressable through an unique URI. Multiple Datapool instances can be generated and customized through the Datapool service by the user. The GET method of Application Service shown in Table 4.1 includes an parameter for the Datapool address. There are two advantages to organizing data in this way. First, because all the data items and the data collection are directly associated with URIs, they are all web resources that can be accessed over HTTP at any time rather than merely a data stream in the form of an extra layer of XML or other structure. Therefore, each data item can also be transferred and kept in their original textual or binary format. Second, in the execution of an application service, the URI of one Datapool that contains all the input data is provided to the service. The application will pull the necessary data automatically from the provided local or remote Datapool. In this way, the interfaces are unified for different application services in the form of a URI, of which the Datapool URI is a constituent as a query string.

4.3.3 GUI Based Interface

To illustrate the features of the deployment service, we use the screenshots shown in Figure 4-5, where Figure 4-5(a) shows the main window of the GUI tool. Our aim here is to make deployment tasks fit within the familiar range of operations of a desktop window manager. The GUI tool is set up to connect with the delpoyment service

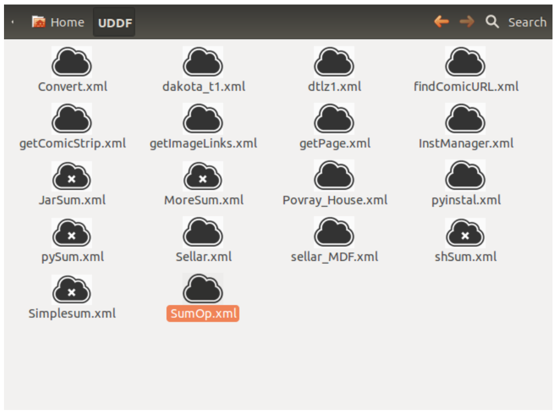


Figure 4-6: Local folder for service description

in the cloud through a URI with user authentication information. For the application uploading task, the user packs the binary and dependencies into a self-contained folder as a compressed file and uploads it cloud side through the deployment service. The uploader can be started from the menu when the user right-clicks on the compressed file[2]. In this case, a Java executable which has two inputs and one output is uploaded. The Java runtime is a special case that can be specified by ticking "Jar executable". One another notable feature shown in Figure 4-5(a) is the access permission setting. The user can choose whether a service can be accessed by all users as a *public service* or by selected users. Permitted users can be added in a separate window by the service owner clicking the *Add Users* button.

Figure 4-5(b) shows the parameter window of the GUI tool. In the deployment process of web service, the framework needs the information for mapping each command-line argument into a parameter for the web service. At the same, the framework also needs to generate a command-line for the invocation of the program. This window allows the description of a wide range of command-line I/O types, such as an argument flag, file path, standard I/O stream, etc. The framework identifies the binary file type through the extension name of file name entered here as well.

One another notable feature shown in Figure 4-5(a) is the access permission setting. The user can choose whether a service can be accessed by all users as a public service or by selected users. Permitted users can be added in a separate window by the service

[2]Thanks to integration with the file manager. Although, in this case, the integration is with the Nautilus file manager on Ubuntu, such overlays are common interface extensions on other operating systems, so we view this as a generic technique.

owner clicking the *Add Users* button. After all the information is entered via this tool, the deployment task is completed by clicking the *Deploy* button.

4.3.4 Implementation Details

The *Service Deployment* and *Service Invocation and Execution* related components are written in Java. JDK6 is used. All the REST web services are written based on the framework *Jersey*. Jersey RESTful Web Services framework is open source, production quality, framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation. The whole framework is developed and deployed in Ubuntu 12.04 VBox VM. It is allocated with 2 threads of 3.2 GHz Intel Core i3 and 4096MB memory. Local file system is used for the storage of all configuration files, libraries and executable files.

The integration of the GUI with the file system is implemented by using Nautilus file manager on Ubuntu. By using Nautilus, it will allow the GUI to have a better integration with the OS that it can be started from the menu by right-clicking on the configuration file. The GUI tool itself is written in JAVA with the support of JWT libraries. By using the same programming language, it is able to reuse some code in existing development. The output of the GUI tool is the XML configuration files that can be recognized by the scientific workflow framework. The GUI tool communicates with the framework, which is remotely deployed in server side, only by REST web service invocation.

4.4 Evaluation

4.4.1 Experiment on Usefulness and Usability

A formal experiment with an after-experiment survey was carried out to collect evidence for the usefulness of the GUI tool-based service management mechanism. The objective was to assess usage of the tool for users who do not have any experience of building or deploying web services. A secondary aim was to collect evidence for the usability of the GUI. In this experiment, four programs were provided to the evaluators, who were PhD students randomly chosen with different research backgrounds from

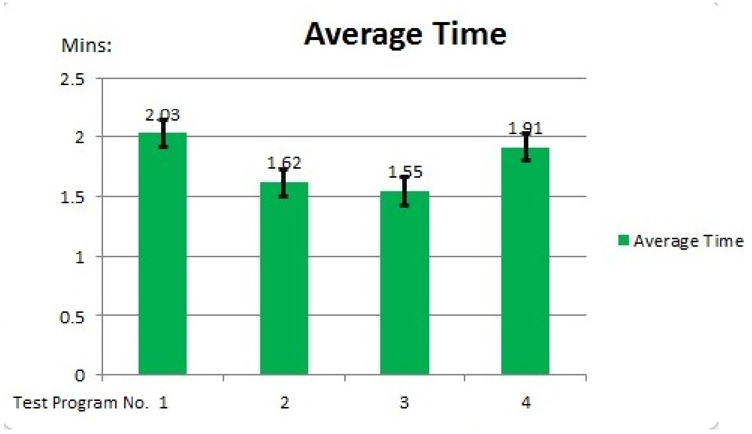


Figure 4-7: The average time of deployment operations

Mechanical Engineering, Computer Vision, Computer Graphics, Human-Computer Interaction and AI. The after-experiment survey has confirmed they do not have any any experience of building or deploying web services. Three of programs had two inputs and one output, and were written in Java, Python and Unix shell, respectively. The other had three inputs and two outputs and is written in Python. The experiment had four stages: (i) a 3–5 minute training stage, which included a tutorial video and question time, (ii) three simple programs were provided to participants to deploy in an order that they decided, while the time to complete the operation was recorded, (iii) a more complicated program for which deployment time was also recorded, and (iv) completing the survey.

Figure 4-7 shows the mean deployment time and with error bars based on data collected from 9 participants. In the first three simple programs, a significant reduction on deployment time is recorded, which indicates a fast learning speed. Obviously, with the increasing number of the application's I/O interfaces, the deployment time grows. However, it is even faster than the first simple deployment. The sufficient training and the familiarity to the GUI tool can further reduce the deployment time. This can be shown in just four rounds of deployment jobs. In this experiment, none of the subjects claimed any prior experience of building or deploying web services.

In a question about their subjective views on simplicity with 5-point scales from very easy (1) to very difficult (5), 2 out 9 said very easy (1), and the rest said easy (2). All the participants successfully deployed web services in around 2 minutes. In the randomly ordered simpler cases, there is a significant fall in the time taken. It also can

```
executeworkflow.sh -inputdoc input.xml \\
  -outputdoc output.xml \\
  -cmdir <Credential Manager's directory path> \\
  -cmpassword <password for Credential Manager>
```

Figure 4-8: The command for the cancer model workflow

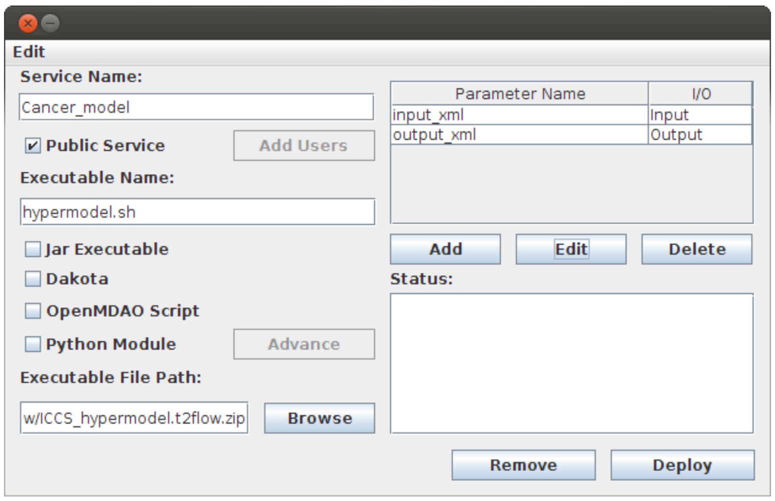
be noticed that after three test cases, the time taken for the more difficult case is less than the first of the simple ones. The objective evidence obtained from this experiment is that the GUI based mechanism is easy to learn and use for single service deployment.

4.4.2 Preliminary Case Study

The case study is specifically designed for the purpose of demonstrating this service management framework. In this preliminary case study, a single component (a single web service) will be deployed to demonstrate the framework's basic functions in a real case on application service deployment, the integration with the Datapool service and the integration with existing scientific workflow system.

In this case, a cancer modelling workflow is deployed as a web service. This workflow includes three models which collaborate together to provide the function as a hypermodel. They are wrapped and invoked through the Taverna workbench. However, it is not a public web service and user can only access it through Taverna. The details about the cancer model itself and relevant terms will not be presented here and the reader is refered the Appendix A.4.

Thanks to the Taverna command-line tool, the workflow can be executed as a whole from command-line. Figure 4-8 shows the structure of the command-line. The rest of the deployment job is straightforward based on our GUI tool. By filling in the necessary command name and I/O list in the GUI as shown in Figure 4-9. The workflow is successfully deployed as a RESTful web service which can be accessed through any form of REST client. For example, as shown in Figure 4-10, a python based client is used to invoke the web service together with the support of the Datapool Service.


Edit

Service Name: Cancer_model

[✓ Public Service]

Executable Name: hypermodel.sh

Executable File Path: w/ICCS_hypermodel.t2flow.zip

Parameter Name	I/O
input_xml	Input
output_xml	Output

Figure 4-9: The GUI for the deployment of cancer hypermodel

```
import libUDDF

service = libUDDF.RESTful_Services()
indate = {'input_xml':<the path to the xml file>}
service.input_file = indata
service.username = <username>
service.password = <password>
service.datapool_URI =
  "<host_URL>/Datapool/Cancer"
service.service_URI =
  "<host_URL>/APP_Services/Cancer_model"
service.execute()
print service.outputs['output_xml']
```

Figure 4-10: The python script to execute the RESTful web service

4.5 Discussion

4.5.1 The Integration with Existing Workflow Management Systems (WMSs)

In Section 4.4.2, an example of deploying a scientific workflow as a web service was presented. This shows the applying range of this ROA based web services management system can be further extended. Considering the WMS as a reference point, the web service can be:

1. the wrapped components in various WMSs that are described either by open or proprietary workflow description language.

2. the wrapped workflows based on various WMSs that can be furthered published and reused for higher level of workflow.

In Chapter 8, some examples about wrapped components in WMSs are presented as well.

Therefore, this ROA based light-weight framework as shown in Chapter 3 and 4 is able to serve as a generic interface for all the resources residing in various WMSs. For example, there is forum like myExperiment[Goble and De Roure, 2007] that is used to makes it easy to find, use and share scientific workflows. The workflow shown in the case study can be uploaded and shared easily through it. There are also local services in each of the WMSs that the other WMSs may not have. All these resources can be re-deployed as web services and integrated with other WMSs or even proprietary platforms. In a nutshell, two benefits can be brought by the integration based on this ROA based web services management:

1. the lower technical barrier to access the resources residing in the WMSs that brought by this ROA based light-weight framework,

2. the generic interface to enable build a higher level of workflow based on various resources, which includes existing workflows.

4.5.2 A Generic Interface for Building Hyper-workflow

Following the discussion in Section 4.5.1, in this section, the benefits of having a generic interface will be further discussed. Firstly, I would like to coin a new term and introduce a concept of Hyper-workflow here. What is Hyper-workflow? Hyper-workflow is still a workflow, the components of this hyper-workflow can be:

1. any scripts, programs, software or their components;

2. web services or other online resources;

3. internal tools and resources in WMS;

4. an existing workflow built by WMS.

The fourth component is the key element that defines the Hyper-workflow. In a hyper-workflow modelling, a composed workflow will never be the final product. It can be reused and integrated into a new workflow when needed. Workflow, especially the scientific workflow, regardless how it is called by its applying purposes, is a collection of computational processes. This is the same as all those programs, web services, etc, which may act in a black-box way. Thus, it naturally allows all the existing workflows to be the bricks of building up more abstract steps for a higher level workflows, which are called Hyper-workflow here. Therefore, it highlights the necessity to keep the compatibility with these WMSs to allow the users to compose user-defined black-box.

There are already such demands and necessity. For example, in the EU project *CHIC* (Computational Horizons in Cancer)[3], scientists and clinicians aim to create multiscale cancer hypermodels (integrative models) based on existing cancer models. Some of the cancer models are workflows by themselves already, such as the workflow shown in Section 4.4.2. These workflows can be built by different technologies and WMSs. At the same time, there are also models in a more traditional forms, such as command-line programs. Obviously, to bring them together in one computational procedure, a generic interface/protocol (is called a Generic Stub in CHIC) is necessary. In such cases, to utilise the existing popular infrastructure (tools, interface, protocol, etc.) rather than reinvent the wheel and introduce new complexity can bring more benefits. This is where the benefits of applying ROA (HTTP/REST stack) based infrastructure can be found. It is believed that the Hyper-workflow in CHIC will not be the only case when more and more workflows, which are potentially able to link each other, are created and published on the platforms like *myExperiment*. By such a generic interface/protocol, not only heterogeneous resources can be integrated into one Hyper-workflow, but also the hyper-workflow can be created and managed by heterogeneous workflow management systems. The iterative process of creating new workflows/hyper-workflows can be even in one existing WMS without introducing new technologies and tools.

[3]This project has received funding from the European Unions Seventh Framework Programme for research, technological development and demonstration under grant agreement No 600841.

4.6 Summary

This chapter presented the evidence for the benefits arising from this light-weight framework for the deployment and execution of scientific application in the cloud. With the GUI based deployment mechanism, the technical barriers are lowered for non-specialist usage of web services and cloud resources. The framework reduces the efforts and enhances the efficiency for users to turn legacy code and programs into web services and hence collaborate with each other. The integration with the Datapool service helps reduce the end-to-end times by hiding the costs of data staging between services as well as between client and service. This chapter also evaluates the usefulness and usability of the framework through a simple user study and case study, showing how different types of legacy programs and tools can cooperate seamlessly in workflows with the support of our framework. The ability to deploy the cancer model workflow also shows the framework can integrate with existing scientific workflow management systems. It builds a bridge to form more complex scientific workflow. In conclusion, the contributions of this Chapter is:

- The system introduced in this Chapter provides a SaaS based mechanism for REST web services deployment and management in a server on per user basis. The system is built in ROA, which simplified the design and architecture. Thus, it can be integrated with existing WMSs simply through the HTTP/HTTPS protocol.

- A GUI based deployment mechanism is provided to further simplify the process for non-specialist users to turn legacy code and programs into web services. This fills the gap between desktop working environment, which is familiar to non-specialist users, and the cloud resources for e-Applications.

- By integrating and be compatible with the Datapool service mechanism introduced in Chatper 3, the direct outputs (in the form of URIs) of one service not only can be retrieved by end-user, but also can be the inputs of next web service. Thus, the web services deployed by this system can be linked together through data-flow without extra protocols and controllers.

Chapter 5

Virtualized Data and Services for Scientific Workflows

5.1 Introduction

In Chapter 3 and Chapter 4, this thesis presented the methods to benefit network utilization on data staging in web services based scientific workflow. These chapters also showed how light-weight ROA based web services management system can overcome the technical hurdle of service deployment for user and assist with legacy code reusability. The works in Chapter 4 enable each component of the scientific workflow to work in a SaaS (Software-as-a-Service) manner. However, there are still constraints on the data staging process as well as reusability of legacy code. For example, it is assumed in Chapter 3 and Chpater 4 that legacy code co-located under one management system will depend on the same stack of OS and libraries. Although multiple versions of libraries can be sophisticatedly pre-installed and configured in one machine to support various legacy code/applications, this level of OS administration activities is normally not supposed to be exposed to public users. Besides, because web services/applications are deployed on multiple machines/OS rather than in one machine/OS, there will be more overhead in the data staging process in workflow execution to transfer data among multiple possibly geographically distributed machines.

There are following constraints co-located with the external dependencies of web services/applications. They will lead to the main issues to study in this chapter. First, modern scientific workflows often includes large number of external dependencies (e.g., requires a certain version of Python/JAVA, or particular versions of subject specific software). It is reasonable to expect that the components of scientific workflow are distributed and deployed on multiple machines. Therefore, in the atomic sequential part of workflow enactment, the data transfer between two machines/OS, or more if a separate data repository is involved, is unavoidable. Second, the execution of certain components will highly depend on the machine/OS with pre-installed library stack. To re-deploy, re-configure and finally reproduce certain component in another machine/OS could incur a series of OS adminstration activities. Because of this hurdle, in the scenario of high scalability demand, the costs on manpower and time can be high. Therefore, the constraints caused by this external dependencies raise two issues to overcome in this chapter. First, the data staging overhead issue. Second, the issue of workflow component reusability reduction, which will further reduce the reproducibility of workflow results.

In this chapter, the virtualization of data and services will be introduced to overcome the two issues on data staging and reproducibility as aforementioned. It presents a scientific workflow enactment framework based on Linux containerization technology. This chapter will start with introducing existing virtualization based workflow platforms and the containerization technology and why this technology can help to overcome the issues, then, in Section 5.3, the architecture of container-based scientific workflow framework is introduced in detail. Then, Section 5.4 will assess the performance costs due to data staging, and to assess to what extent the containerization technology can improve on this situation. Section 5.5 discusses the benefits of introducing virtualization and containerization into scientific workflow enactment. It also discuess what are the ideal situations for the application of such a framework.

5.2 Scientific Workflow and Virtualization Technology

5.2.1 The Challenge of Scientific Workflows in the Cloud

With the emergence of cloud computing, commercial information technology and infrastructure has been greatly influenced and changed. Cloud technology like Software-as-a-service(SaaS) enables remote access of computing- and data-intensive services from home or even mobile devices that cannot be provided with in-house facilities and expertise, such as government public services, financial services and so on. In addition, Platform-as-a-service (PaaS) service provided by cloud providers simplify the development and operation of SaaS applications. They bring the benefits of costs reduction, lower technical hurdle and more applicable scenarios that innovates new services. However, as revealed in Foster [2015], the current scientific cyberinfrastructure is outdated. In contrast, the technology seems stuck in the 20th Century. Just as in this thesis, the challenge is rooted in the issues that scientific applications need specialized expertise to install, configure, maintain and execute the scientific applications. This issue also raises the hurdle to bring resources (applications and data) together in an composite manner such as scientific workflow. As a result, it will also make the resources hard to be accessed and reused. In Chapter 3 and Chapter 4, this thesis introduced how ROA and SaaS can alleviate the situation in a simpler and more sustainable way. However, as introduced in Section 5.1, they are still not simple enough to cope with all scenarios and have drawbacks on reproducibility. Thus, the challenge is how to achieve a simple to use scientific workflow framework for non-specialist as well as enhance the reproducibility of applications/services based on Cloud technology.

More specifically, in the SaaS based framework proposed in Chapter 3, there are two essential issues. First, although the Datapool service can enhance the data staging performance by asynchronous transfers and shared data storage, it cannot avoid the data passing through networks supported by a stack of protocols, which may increase the complexity of framework. The ideal situation is that all the data is kept locally and can be shared by all the scientific applications. The second issue is that users do not have the permission to configure the OS where their applications are executed on. Each OS needs sophisticate system administrator to configure and maintain. Thus, the concrete challenge is how to enable an ideal situation that data can stay locally or with a minimum transmission distance and at the same time the heterogenous dependencies of scientific applications can be supported. As a more ideal situation, the dependency of each application can be directly configured in the OS by the user. It is believed that Virtualization technology has the potential to achieve this. First, all the VMs are naturally stay on one host machine/OS. Second, each of the VM can be directly deployed and configured by the user.

Figure 5-1 and 5-2 shows the difference between the data staging in an existing framework and the data staging in a framework based on a virtual machine(VM). This comparison aims to provide an intuitive understanding on how to resolve the issue on data staging and at the same time simplify the architecture of the framework based on

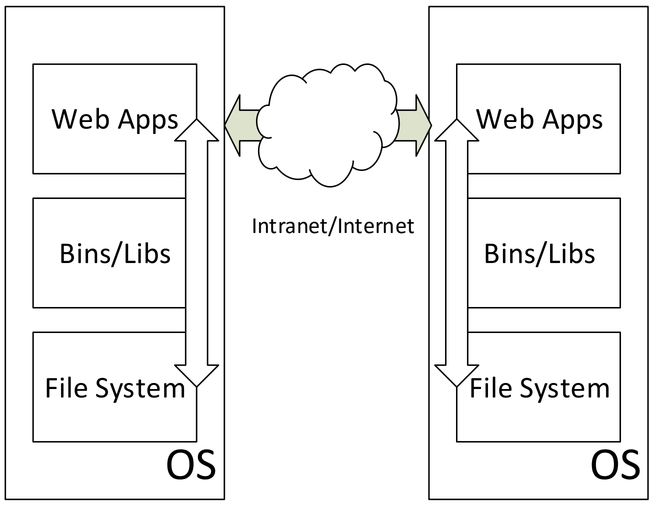


Figure 5-1: Data-flow between two host OSs

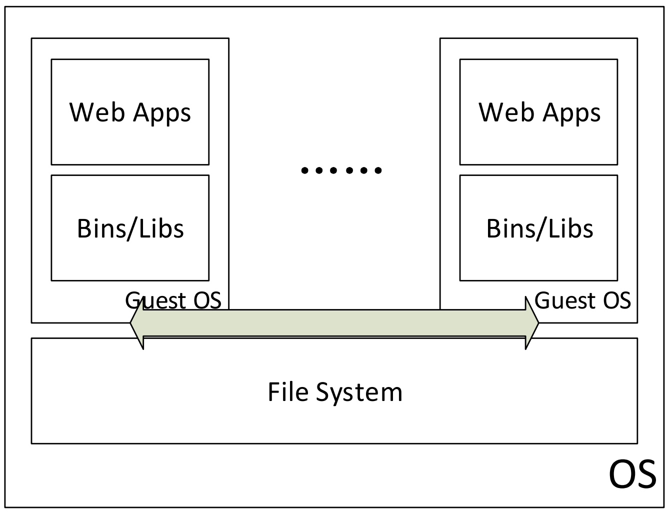


Figure 5-2: Data-flow between two guest OSs

virtualization. In the environment depicted in Figure 5-1, data-flow has to go through the whole stack of dependencies and network. Comparing with the it, Figure 5-2 does not need to have such a data staging process. The data staging can be processed through the local file system. There is one concern on the architecture presented in 5-2. To keep the service always active for users, the VMs need to be in a standby status.

In an extreme situation, the number of services is equal to the number of VMs in the host machine. It means one VM only host one service. All the VMs must be active to make sure all the services are active. This may incur the situation that large amount of hardware resource is occupied or even wasted. Also, these guest OSs cannot work in an efficient on-demand manner as their start-up costs too long time. Based on these facts, a conflict emerges that either hardware resources are occupied to provide a fast reaction on service invocation or users need to wait for VM start-up time when calling services in order to save hardware resources. This is an concrete challenge needs to be conquered.

5.2.2 PaaS and Linux Container Virtualization

PaaS is a type of cloud computing service that provides a platform allowing customers to develop, run, and manage web applications created using programming languages, libraries, sevices and tools provided by the service provider [Mell and Grance, 2011]. It can reduce the complexity of building, configuring and maintaining the infrastructure, which the web application depends on [Chang et al., 2010]. Nowadays, the implementation of PaaS closely links with virtualization technology. For example, hardware virtualization allows a complete abstraction between the operating system and the actual hardware that it was running on. This serves as the foundation for web application with different dependencies to run on one host machines.

There are a series of hardware virtualization implementations based on hypervisors, such as the Kernel Virtual Machine (KVM) [Kivity et al., 2007], Xen [Barham et al., 2003], Hyper-V [Velte and Velte, 2009], or VMware ESX/ESXi [vmware, 2012]. A hypervisor runs multiple virtual machines on a host machine. Each of the virtual machines runs as a guest machine. A hypervisor enables the guest machine to run as independent computer system with its own resources and operating system. Based on hypervisor, multiple instances of a variety of OS can share the same hardware resource. Theses hypervisors are all based on emulating virtual hardware, which means they make substantial demand on system resources.

There is also a different level of virtualization, called containerization, which takes place at the operating system-level. It enables the kernel of the operating system to provide mechanisms to isolate processes in system or application containers. The container is further isolated from the infrastructure level of virtualization. So it enables the mechanism that different web applications with heterogeneous dependencies on OS-/libraries to run in one operating system. Those containers are independent from other, but share the same operating system, which includes the kernel and device drivers. This level of abstraction between the web application dependency stack and the actual operating system perfectly matches the service product abstracted as PaaS. Early implementation was introduced in 1982, which is based on chroot operation that changes the apparent root directory for the current running process and their children. Some other implementation examples include Linux-VServer [lin], Solaris Container [Lageman and Solutions, 2005], openVZ [ope], LXC [lxc] and Docker [Merkel, 2014] for different operating systems.

The potential benefits that can be brought to scientific workflow are: *a*) although all the applications/services can be executed in one host OS, each of them can have independent libraries and software as dependencies in a light-weight working environment; *b*) based on the nature that all of the applications/services are co-located on one physical machine, there is the potential that the data staging among them can take less time.

5.2.3 Docker

Docker Merkel [2014] is an open source project, which was first released in 2013, that has seen increasing popularity in recent years. It builds on LXC containers, virtualization of the OS, and a hash-based or git-like versioning and differencing system. Docker currently supports all major Linux distributions. This versioning and differencing system provides an easy-to-access interface for users to store, manage and deploy customized OS images. This makes it an important reason why Docker is chosen to be adopted in our framework. This important feature is based on that, first, Docker provides mechanisms to deploy applications into the containers with a Dockerfile. It allows system to builds images automatically by reading the instructions from a Dockerfile, which is a text document that contains all the commands a user could call on the command line to assemble an ready-to-use image. Second, it has an important difference from the other Linux containers - the layered filesystems. A docker container is made up of a base image plus layers of committed Docker images. These enable Docker to do image versioning and 'commit' in a Git style approach. With the support of Docker Hub, which is like Github to Git that makes it easy to share the im-

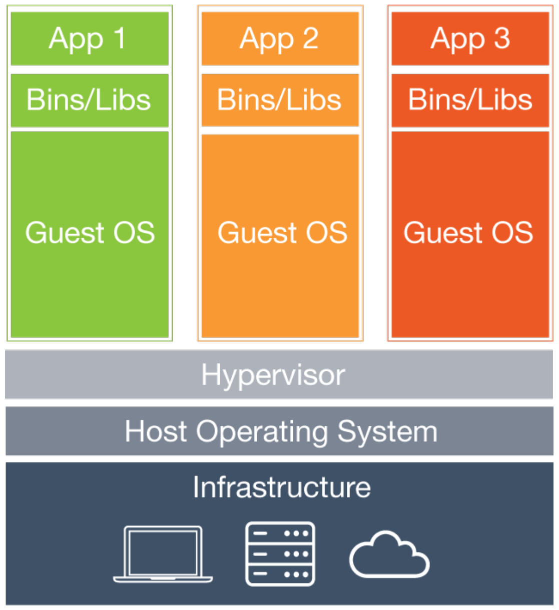


Figure 5-3: VM architecture[Docker, 2016]

age publicly. The reasons why these features are so important to the containerization technology this framework would like to adopt are:

- Independence: it allows the scientific application providers to provide specific working environment without the necessity to be aware of the dependencies and configurations of other applications possibly will be used in a scientific workflow;
- Shareability: this git-like platform enables an simple interface for scientific application providers and users to share these customized working environment together with the scientific application;
- Reproducibility: this mechanism allows users to easily and quickly reproduce a working environment for all the scientific applications and workflows in an out-of-the-box manner based on Docker Engine.

Figure 5-3 and Figure 5-4 Docker depict the architectural difference between VM and Docker container. Each VM includes the application, the binaries and dependencies, and an entire guest operating system, which reach tens of Gbs in size. On the other hand, containers include the application and all of its dependencies, but share the kernel with other containers. Thus, because of the light-weight architecture, the start

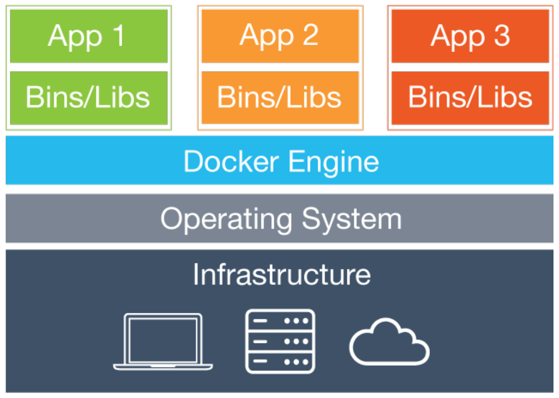


Figure 5-4: Docker architecture[Docker, 2016]

and stop time are much less than the start and stop time of VMs. In Hussain [2014], Docker container are compared with KVM on a test platform with Inter(R) Core(TM) i7 CPU, 47 GB memory with 23GB swap and 2 drive raid1 disk configuration. Test results are 50ms by comparing with 30-45 seconds start time and 5-10 seconds stop time of VMs. By this feature, it is possible to alleviate the issues that VMs take too much resource to be on idle status and the start and stop time are too long if keeping them off when they are idle.

Based on Docker, the potential benefits of adopting containerization technology in scientific workflow enactment can overcome the issues raised in Section 5.2.1. Some of the key issues, such as shareability, reproducibility and long start and stop time of VMs , which are raises in the thinking about how to utilise virtualization technology in scientific workflow have the chance to be resolved. In the following sections, the approach of how to implemented such an infrastructure by combining Docker together with ROA and SaaS based scientific workflow modelling will be introduced.

5.3 Container based Workflow Enactment

5.3.1 Architecture

In the architecture of the scientific workflow framework proposed in Chapter 3 and Chapter 4, when the dependencies changes, the applications and services have to be deployed in a different OS with different dependencies pre-installed. The services communicate with each in a distributed manner through the network. Figure 5-5 depicts this process. This chapter will investigate the potential improvement from technical perspective by adopting containerization technology. With this container-base architecture, services may be deployed in multiple containers but co-located on the same physical machina/OS. The first question in the process to design this architecture is: when services have to be deployed in the distributed manner?

There are following reasons, such as:

- Limited hardware resource on one machine
- Licensing or legal requirement that certain service has to be deployed in certain network or machine
- Legacy services for which migration can be costly on both manpower and time

The above are hurdles with a more non-technical basis. For example, considering there is enough investment, the first reason can be resolved. In some cases, to purchase a new machine can be more feasible than upgrading hardware on a machine in duty. Before the design of framework architecture, one certain thing is that the architecture will not aim to resolve any of the above hurdles.

Then it will be introduced that what exactly this architecture aims to resolve. From technical perspective, in Section 5.1, two issues have been identified: workflow component reusability and data staging overhead. In previous discussion, Docker, as an implementation of containerization technology, will be adopted to resolve these issues. Thus, three requirements emerge. First, based on Docker, a container for each of the workflow component will be created. Here a mechanism is needed to deploy and manage all the containers. Second, based on Docker and all the containers, a mechanism is needed to manage the I/O data for workflow in local file system in the host OS where Docker is deployed. One more important thing is, with these new mechanisms introduced, the complexity of operations will increase. Just as emphasized through out this thesis, it is still important to have a mechanism to allow non-specialist users to use it, in this case, without knowing details about how Docker and containers are organised in this architecture.

Following last paragraph's discussion and three requirements, a containerized scientific workflow framework is proposed in this chapter. Figure 5-6 shows the architecture of this framework. Following are the three new components corresponding to the three requirements. They are built based on the work from Chapter 3 and Chapter 4.

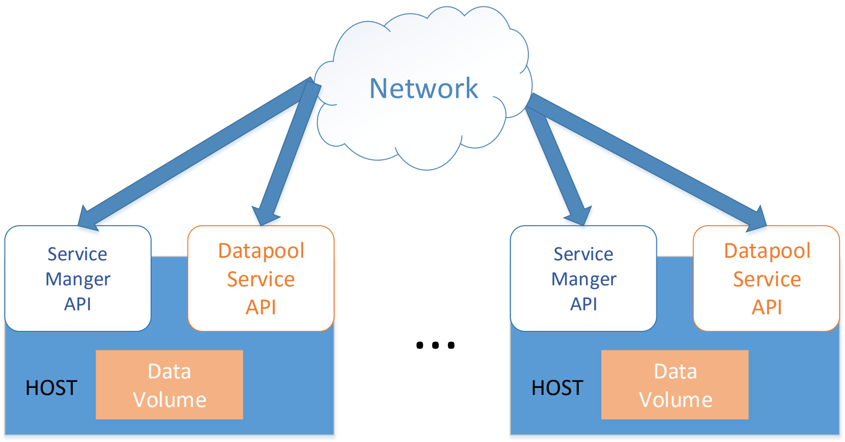


Figure 5-5: The Architecture of Multiple Framework Instances

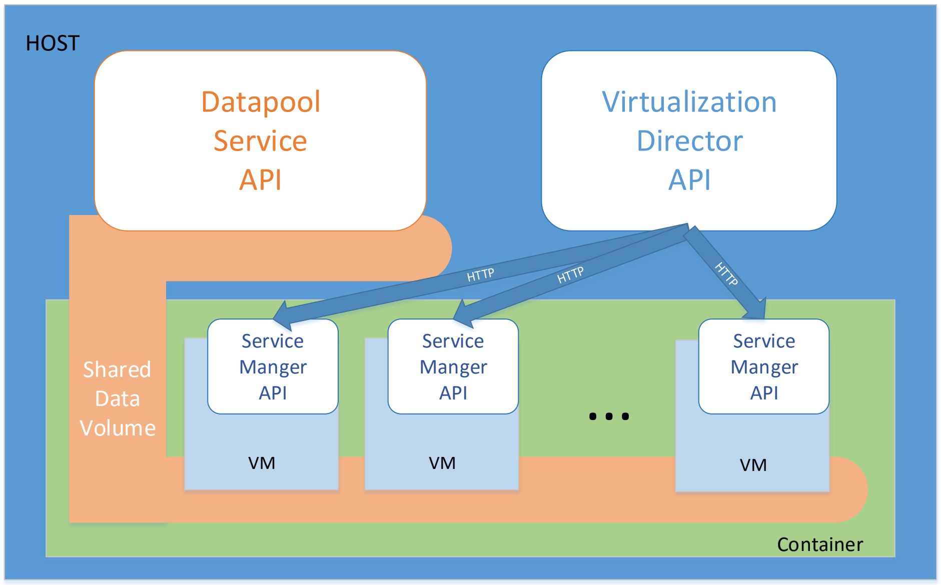


Figure 5-6: The Architecture of Containerized Framework

Virtualization Director

The virtualization director API is located in the host, where Docker is also installed. It serves as a gateway to all the services deployed in containers (the Service Manger is pre-installed on the Docker images). It has the following three functions: *a*) matching the appropriate image for service deployment and execution; *b*) start container based matched imaged and maintain the lifecycle of the container; and *c*) execute the service within the container through allocated local ports. The design of Virtualization Director aims to abstract the web services invocation activities. The management of images and containers are hidden from users. The service creators need to provide the information of what dependencies the services need. In Section 5.3.2, the schema for this configuration information is introduced.

Shared Datapool Service

To enable the seamless data staging, the shared Data Volume function provided by Docker is adopted in this architecture. This design brings two advantages: *a*) it reduces the redundant data that distributed on multiple machines; *b*) it eliminates the unnecessary data staging process between two services deployed in two operating systems. In Figure 5-6, the Datapool Service, as in Chapter 3, is exposed to external network. All the data, which includes services information, applications, I/O data are all stored in the shared data volume. All the containers can access the data volume with the permission from corresponding users.

Transparent Service Manager

In this architecture, the original Service Manager is hidden from external APIs. The interfaces are mapped to the Virtualization Director. In Figure 5-6, the Service Manager API in each container is exposed to Virtualization Director. When the user invokes the API of Virtualization Director, it will look for the matching image, start it as a container and invoke the Service Manager API in this container. In this way, the details of managing the Docker system will be transparent to users.

5.3.2 Virtual Service Configuration File (Metadata) Schema

In Section 4.3.1, a description file in the form of XML is introduced for application description and service configuration. With the adoption of Docker, more information is needed for container configuration. In this section, an upgraded configuration description language for virtual services is proposed in the form of JSON schema[Galiegue and Zyp, 2013]. This is used to describe the application, virtual service and the working environment for all of them. By using this, the Virtualization Director can automatically allocate the appropriate container for applications. Additionally, this is the basis for how the new Service Manager can keep the details of Docker implementation and container transparent to non-specialist users.

List 5.1 is the JSON schema for virtual service deployment. It serves as the metadata of the service. With the refactoring work on the whole scientific workflow framework to fit the virtualized platform, the configuration file for service is also updated as JSON. The Service Manager uses a similar configuration schema for service deployment. The only difference between the Service Manager configuration file and the one for Virtualization Director one is that the latter has two more properties: *neededOS, neededSoftware*. The *neededOS* is a string to indicate the OS of the image on which the service will be deployed and executed. The *neededSoftware* is an array of strings to indicate all the dependencies names and versions which will be used to match the *supportedSoftware* field in the metadata of the Docker image. This matching step is introduced in Section 5.3.4.

Two fields need to be discussed in detail. Both *workingDirectoryInputNameList* and *workingDirectoryOutputNameList* are dictionary objects, so each contains key-value pairs, of which the keys are the I/O file names in the working directory. In some specific circumstances, the application consumes and generates I/O files from default locations in the working directory. Those file names can be only temporary symbols that are not self-explaining. In this case, the values in these pairs are the external file names that will be used by the service output as well as Datapool.

Listing 5.1: Virtual Service Configuration Schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://virtual_configurable.com/
     app_descritpion_schema",
"type": "object",
"properties": {
  "createUser": {
    "type": "string"
  },
  "serviceName": {
    "type": "string"
  },
  "commandString": {
    "type": "string"
  },
  "inputNameList": {
    "type": "array",
    "items": {
      "type": "string"
    }
  },
  "outputNameList": {
    "type": "array",
    "items": {
      "type": "string"
    }
  },
  "inCommandArgumentInputList": {
    "type": "array",
    "items": {
      "type": "string"
    }
  },
  "workingDirectoryInputNameList": {
    "type": "object"
  },
  "workingDirectoryOutputNameList": {
    "type": "object"
    },
    "isStdin": {
      "type": "boolean"
    },
    "isStdout": {
      "type": "boolean"
    },
    "isPublicAccess": {
      "type": "boolean"
    },
    "authorizedGroup": {
      "type": "array"
    },
    "neededOS":{
      "type": "string"
    }
    "neededSoftware": {
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  },
  "required": [
    "createUser",
    "serviceName",
    "commandString",
    "inputNameList",
    "outputNameList",
    "neededOS",
    "neededSoftware"
  ]
}
```

5.3.3 Authentication and Authorization

Now there is one more layer of APIs, the Virtualization Director. In addition, multiple services are deployed in multiple containers. To organize the authentication and authorization of all these components, a more sophisticated mechanism is needed for Single Sign-on(SSO). For this purpose, JSON Web Token (JWT) Sakimura [2015] is adopted in this framework.

JSON Web Token (JWT) is a JSON-based open standard (RFC 7519), which is designed to enable compact, URL-safe means of representing claims to be transferred between two parties in a SSO context. JWT claims can be used to pass the identity of authenticated users between an identity provider and a service provider in the web applications environment. This matches the demands on authentication in our framework. The framework's Virtualization Director and Datapool needs to communicate and manipulate services as well as data on behalf of framework users. So, JWT is mainly used in this virtualized scientific workflow framework for two scenarios:

- The Virtualization Director invokes the Service Manager on behalf of the user.
- The Datapool invokes another Datapool service on different machine to collect data (for example, the output from last step as the input for next step) on behalf of the user.

The JSON identifier service can be a standalone service or co-located with the virtualization director. All the services can be accessed by either username password pair or JWT, but all requests on behalf of a user from any service are done with JWT only. The service requests the JWT by calling the identifier service. On the server side, JWT stores a single key in memory (or in config file) - called a secret key. That key has two purposes, it enables creating new encrypted tokens and it also verifies all tokens. The benefits of using JWT are:

- The Service Manager in each container does not need to have the full knowledge of authentication information or access to authentication information database. New container can be created without user authentication information, or even remote access to authentication information. This enhances the scalability by means of faster container deployment and configuration speed and smaller size of container.

- Based on JWT authentication, the responses are much faster than normal authentication requests. The reason is there is only one token to be checked for authentication within local JWT system. In contrast, in a centralized authentication service, container still needs to send request remotely. Each of the requests contains username password pair or token for verification.

With the support of this authentication method, the authorization information associated with each service can be stored together with the service information. For example, the usernames who can access the service. This list of usernames can only created or modified by the creator of the service.

5.3.4 The Execution Process of Virtualized Application in the Framework

Figure 5-7 depicts the process of one round of service execution that is directed by Virtualization Director. It starts from getting inputs or references (URIs) of the inputs and ends with sending back the reference (URI) to the outputs. More details are as follows:

Authentication & Authorization Checking for Datapool Service Both the Datapool service and Virtualization Director service are protected by the authentication and authorization mechanism described in Section 5.3.3. A user needs to have both permissions to access the data storage and the service. If either of them fails, status 403 is be returned.

Is the Input in the Host? When execute the following application service, the argument Datagroup URL needs to be sent. It points to the independent data sandbox for the user and service. If the inputs are not in the Datagroup, the user needs to upload it at this step.

Matching Image Virtualization Director will find the matching image to create the container in following step here. The matching process in a way the the required dependencies (recorded in service configuration information) must be the subset of the available dependencies that provided by the image (recorded in the image information in virtualization container). The first matched image name will be returned for the following process.

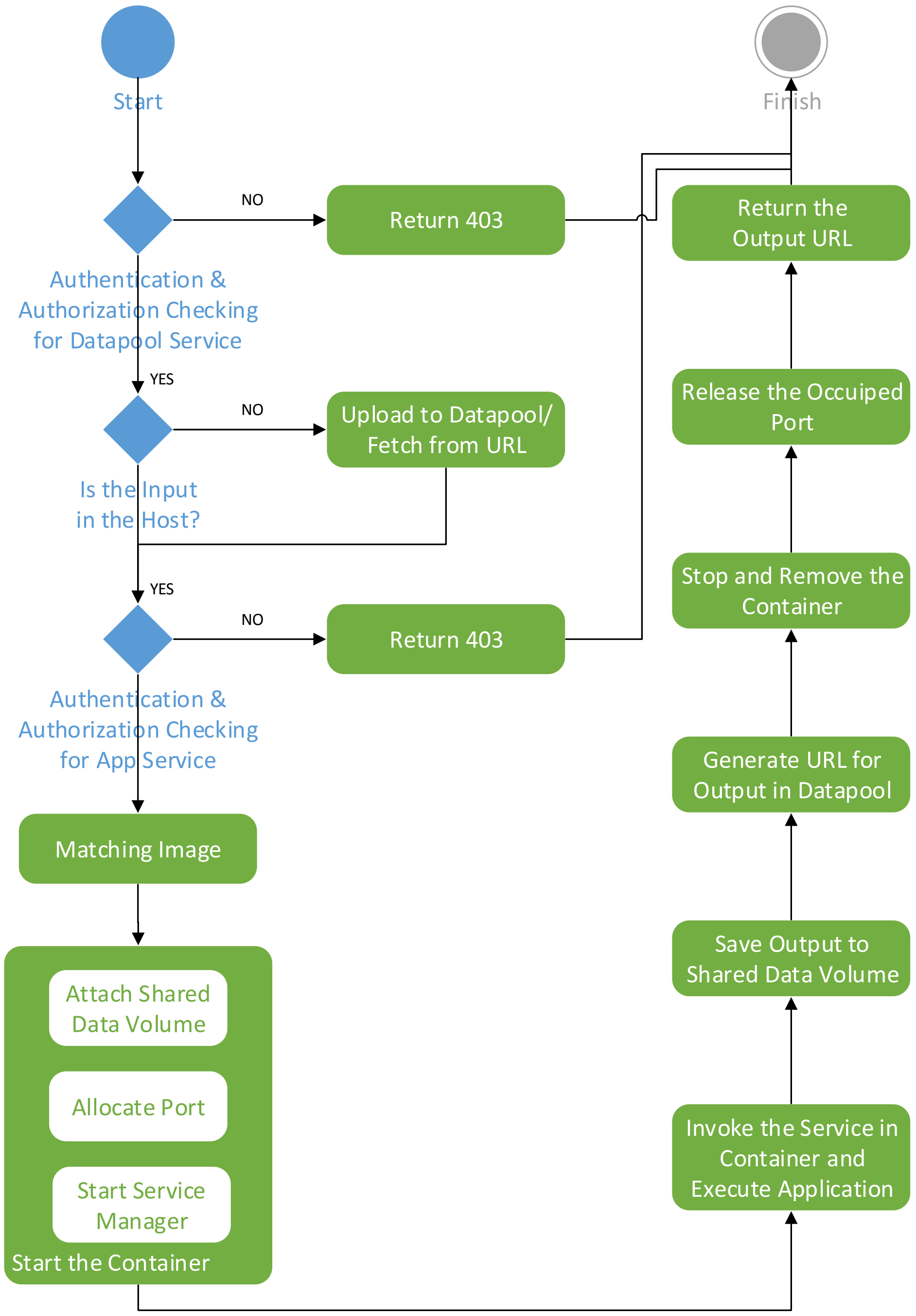

Start
Finish
NO
Return 403
Return the Output URL
Authentication & Authorization Checking for Datapool Service
YES
NO
Upload to Datapool/ Fetch from URL
Release the Occuiped Port
Is the Input in the Host?
YES
NO
Return 403
Stop and Remove the Container
Authentication & Authorization Checking for App Service
Generate URL for Output in Datapool
Matching Image
Save Output to Shared Data Volume
Attach Shared Data Volume
Allocate Port
Start Service Manager
Start the Container
Invoke the Service in Container and Execute Application

Figure 5-7: Activity diagram of virtualization director

Start the Container Before start the container through the Remote API provided by Docker, there are three steps as prerequisites. They are: (i) The shared data volume, which also used by Datapool service, needs to be attached and mounted to the container. (ii) The port needs to be picked from ports pool and allocated to the new container for internal access from Virtualization Director to Service Manager in the container. This port will be returned to the pool in following step after the execution. If the pool is empty, the starting job needs to wait until new port is available. (iii) The service manager needs to be started when the container is created.

Invoke the Service in Container and Execute Application In this step the Virtualization Director will invoke the corresponding container and service to execute the application through the pre-allocated port on localhost.

Save Output to Shared Data Volume The output data will be saved to the designated Datagroup, which is in the Shared Data Volume. Thus, the Datapool service on the host on top will be able to access the output data after the life cycle of the container.

Generate URL for Output in Datapool The Datapool service will generate a URL for the output, which points to the host address.

Stop and Remove the Container The container is stopped and removed in this step.

Release the Occupied Port The occupied port resource will be returned to the port pool.

Return the Output URL The URL is returned to client together with status code 200.

5.4 Experiments and Results

5.4.1 Framework Implementation

The whole framework is written in Python based on the Django REST framework, which is well maintained and has large user group. It also has a good integration with JWT system. There are three web applications included in the system: *a*) Datapool; *b*) Service Management; and *c*) Virtualization Director. Both Datapool and Service

Management are re-implementations of the Java based framework introduced in previous chapters to accommodate the new features adopted in the new framework, such as JWT.

The Docker engine version 1.8.0 is installed on the host platform. Communication between Docker and the framework is carried out through the Docker Remote API (v1.18) whose operations can be invoked as RESTful web services. The input for the API is a series of configuration files written in JSON. The template for this configuration file is listed in Appendix C. Through the template, the Virtualization Director can fill the content for the image name, the port number for service in container, the shared data volume path. These are basic variables for runnable Docker container.

All the configuration information is stored in a Sqlite database. The application binaries are stored in an designated folder called *appRepo*. The I/O data are categorized based on username and data group name given be the user in an designated folder called *sandbox*. These files can be placed at any directory in the OS. Because all three data locations can be customized through fields in setting.py file in this Django project's root directory. In the configuration for experiments, they are all stored in the shared data volume, whose path stored in the corresponding field in the JSON configuration file of the proceeding paragraph.

5.4.2 Experiment Design

The experiments firstly aim to demonstrate the feasibility of implementing such a framework as described in Section 5.3 based on Docker containers. More importantly, the experiments aim to provide evidence that the scientific workflow will get better data staging performance in execution without sacrificing performance on service start-up in which the container mechanism plays an important role. To examine this issue, the experiments need first to show that the network-based and web-service based data staging solutions have significant impact on data staging performance. Subsequently, based on the experiment results, we discuss the which is suitable solution for scientific application/workflow.

Workflow Scenarios Setting

The workflow simulates the process of data staging from one process/service to another process/service. This data staging step is the atomic unit of a more complex data staging process. There are five versions of the data staging process:

1. local file system copy operation;
2. pipe between processes;
3. socket (localhost) between two processes;
4. HTTP (REST web service) between two services; and
5. HTTP (REST web service) between services on two different host machines, but shared data volume between two services in containers/host on one host machine.

Of the above the HTTP experiments are carried out using both localhost and LAN for comparison. When the LAN environment introduced, a virtual network between two VMs is set up to simulate this scenario in order to eliminate the influence from other network traffic. Furthermore, all the local file system I/O is carried out on SSD disk to eliminate the influence from hard disk caching mechanism and possible reading speed difference between inner and outer cylinders. In addition, RAM disk is used based on SSD disk to further eliminate the influence of cache in harddisk. The hardware and software used to host the VMs are listed below:

- CPU: E5-1650 v3 @3.5GHz
- RAM: 32GB DDR4 ECC
- Hard disk: 256GB SSD
- VM: Virtual Box 4.3.26

All the experiments programs are written in Python and the source codes can be found in Bitbucket[1]. To record and examine the influence of different data size, a series of different sizes of input data are used for the experiments. Data ranges from 100 MB to 1 GB with a 100 MB interval among them.

Experiments Procedure

The data-flow for each of the scenarios is depicted in Figure 5-8.

Copy The 1) workflow in Figure 5-8 is a simple copy operation carried out by application APP_A1. The input and output are the same, which also applies to the following workflows.

[1]In branch *Develop* folder *Experiments* of https://bitbucket.org/me1kd/virtual_configurable

Pipe The 2) workflow depicts the procedure that the data is transferred from one process to the other process through pipe. Both of the processes are controlled by APP_B1.

Socket The 3) workflow is the data staging workflow between two applications through socket. In this experiment, they are located in the same localhost.

Web service The 4) workflow is the Web service based data staging. This workflow brings in the concept of separation of data-flow and control-flow. In previous three workflows, the receiving of the data and the start of following application which consumes the data are both activated in one step. Thus, there is one step to upload the data as Service_D1 at server side. The Service_D2 simulates the actual execution/consuming on the data. In this experiment, it only checks the data availability. The App_D1 acts as client application.

Containerized web service The 5) workflow is similar to 4). The only difference is the Service_E2 that consumes the data is deployed in container.

Each of the workflows is ran ten times with ten different sizes of input data from 100MB to 1GB. Thus, in total 100 runs for each workflow. The whole process is carried out automatically by a Python script. In all the executions, the origin and destination of data-flow are in ramdisk. The framework and its database are located and executed from ramdisk as well. In is the client-server experiments in 4) and 5), both of the client and server hosts are set up on ramdisk as well.

Experiments 4) and 5) in Figure 5-8 are done under several different conditions for further comparison. Experiment 4) is run in both localhost and LAN network environments. Experiment 5) has one more variable, in that all the containers created in the experiment are either left active but idle, or deleted after each round of workflow finishes. In Figure 5-14, 5-15 and 5-16, A indicates Active but idle, D indicates Deleted.

5.4.3 Results

The result of the experiments are shown in Figure 5-9 - Figure 5-16. All the numbers from Figure 5-9 to Figure 5-16 shows the average of workflow execution time. The x axis is the execution time with seconds as unit. The y axis is the size of the I/O data

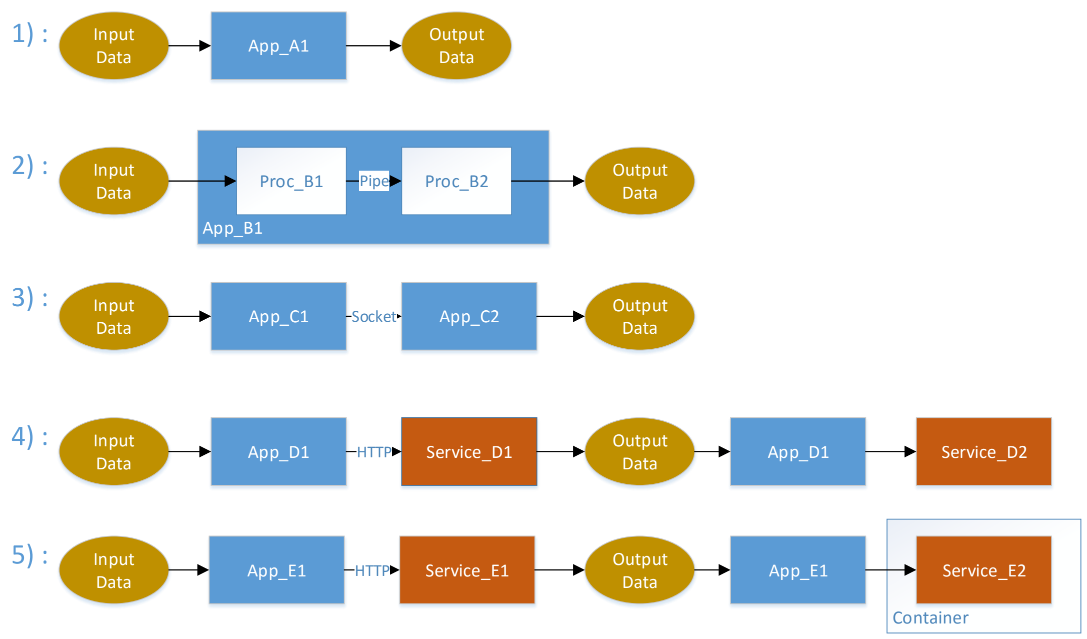


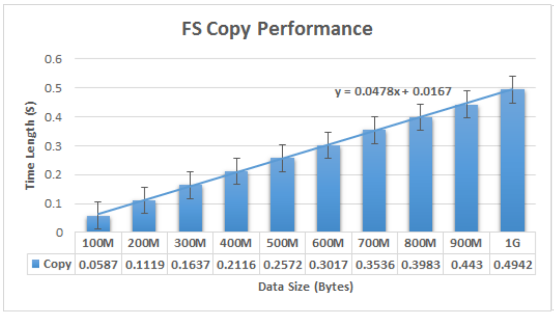
Figure 5-8: Experiments Procedure

from 100MB to 1GB. The error bar in each figure indicates the standard deviation. The accurate execution time is also listed in the form at the bottom of each figure with second as unit. The general trend depicted in the results from Figure 5-9 to Figure 5-16 shows that all the experiments generate almost linear results. This indicates that with the changes on data transfer method and service deployment location, there do not appear to be any hidden factors: there is only the parameter change. All the formulas for the linear regression models of them are also listed in the figures. They will provide an intuitive comparison of the changes brought by different methods.

The comparisons among all of them are demonstrated through following aspects:

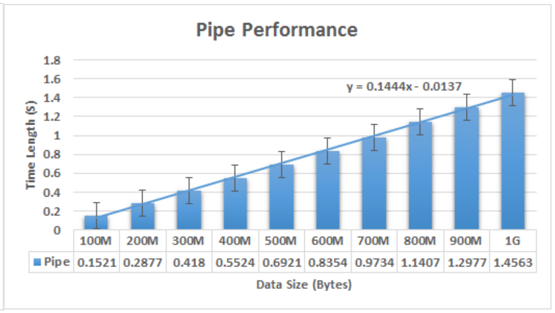
Local methods VS Network based methods

Copy workflow has the fastest speed considering it is only one process and local operation. The Coefficient Of X (COX) is 0.0478. Both pipe (COX=0.14) and socket (COX=0.09) workflow are almost at the same level. Socket (in this case TCP based) is already a network based method with one layer less by comparing with HTTP, which is used by the web service in this experiment, in OSI model. By one more layer of protocol, it can be observed that there is a significant jump. The COX of localhost WS based workflow is 1.2739. The distance between client and server impacts the speed significantly. When client and server are put



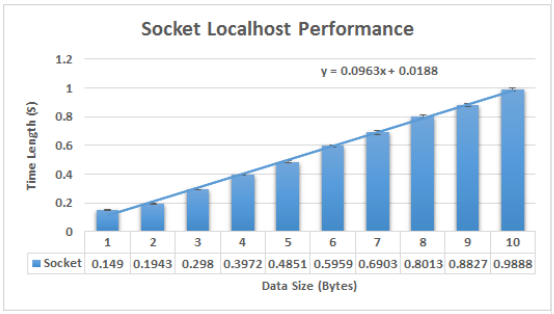
	100M	200M	300M	400M	500M	600M	700M	800M	900M	1G
Copy	0.0587	0.1119	0.1637	0.2116	0.2572	0.3017	0.3536	0.3983	0.443	0.4942

Figure 5-9: Copy Workflow



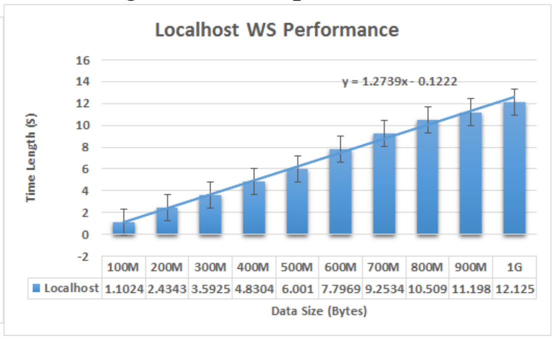
	100M	200M	300M	400M	500M	600M	700M	800M	900M	1G
Pipe	0.1521	0.2877	0.418	0.5524	0.6921	0.8354	0.9734	1.1407	1.2977	1.4563

Figure 5-10: Pipe Workflow



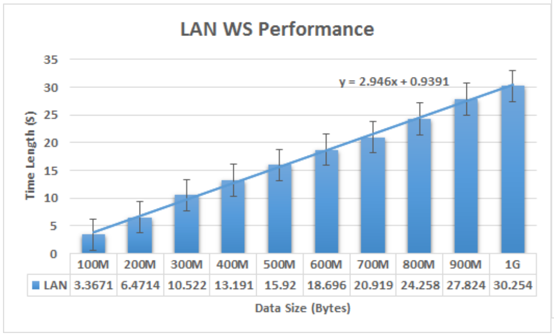
	1	2	3	4	5	6	7	8	9	10
Socket	0.149	0.1943	0.298	0.3972	0.4851	0.5959	0.6903	0.8013	0.8827	0.9888

Figure 5-11: Socket Workflow



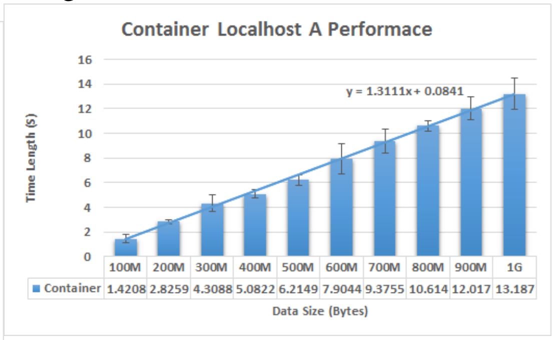
	100M	200M	300M	400M	500M	600M	700M	800M	900M	1G
Localhost	1.1024	2.4343	3.5925	4.8304	6.001	7.7969	9.2534	10.509	11.198	12.125

Figure 5-12: WS Localhost Workflow



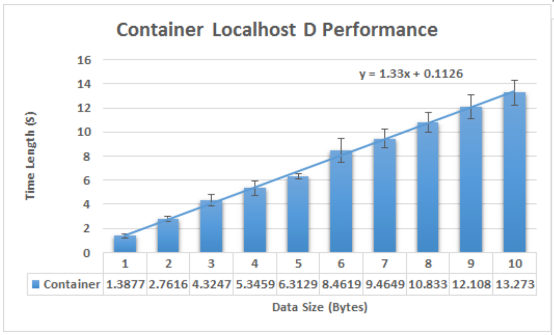
	100M	200M	300M	400M	500M	600M	700M	800M	900M	1G
LAN	3.3671	6.4714	10.522	13.191	15.92	18.696	20.919	24.258	27.824	30.254

Figure 5-13: WS LAN Workflow



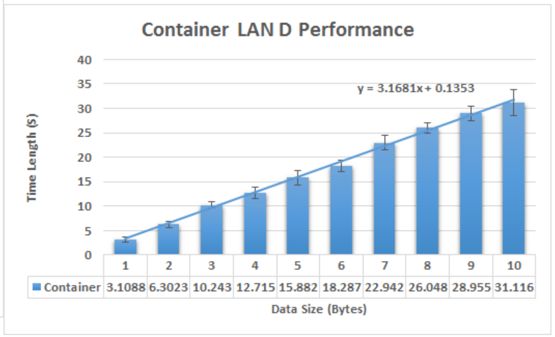
	100M	200M	300M	400M	500M	600M	700M	800M	900M	1G
Container	1.4208	2.8259	4.3088	5.0822	6.2149	7.9044	9.3755	10.614	12.017	13.187

Figure 5-14: Container Localhost A Workflow



	1	2	3	4	5	6	7	8	9	10
Container	1.3877	2.7616	4.3247	5.3459	6.3129	8.4619	9.4649	10.833	12.108	13.273

Figure 5-15: Container Localhost D Workflow



	1	2	3	4	5	6	7	8	9	10
Container	3.1088	6.3023	10.243	12.715	15.882	18.287	22.942	26.048	28.955	31.116

Figure 5-16: Container LAN D Workflow

in two hosts that communicates through LAN, the COX grows to 2.946.

Web service based methods VS Container based methods

Considering one of the web services is hosted in a VM in the container based methods, the overhead is expected in the results. However, it is not significant especially by comparing with the change brought by different network condition. For WS localhost workflow, the COX is 1.2739. The COX of container localhost D workflow is 1.33. On the other hand, the COX of WS LAN workflow is 2.946, and the COX of container LAN workflow D is 3.1681. By applying containerization, there is slight growth on COX. By changing the network setting, the COX grows more than double.

Between two container based methods

In the container based experiments, two different modes are brought in for localhost experiments. All the containers created in the experiment are either left active but idle (indicated as A), or deleted after each round of workflow finishes (indicated as D). For normal VMs, even they are kept as idle, they will still occupy CPU and memory. By deleting the running normal VM instance, the start-up time of a VM instance in next run will extend the overall execution time. However, this is not observed in this container VM experiment environment. The COX of container localhost A workflow is 1.3111. The COX of container localhost D is almost the same at 1.33.

5.5 Discussion

5.5.1 The Suitable Solution for Scientific Workflow

In the experiments, the comparison is not only performed between web service based method and container based method. The traditional methods used in the single process, multiple processes and multiple applications across the network are also tested in the experiments. It aims to demonstrate that by introducing new methods, what has been traded off. It also leads to a general discussion on the reason why new methods need to be brought in for scientific workflow. It also leads to a more specific discussion on what is the suitable solution for a certain type of scientific workflow.

The key advantage of having scientific computing process operated in a network environment is that it allows the integration of multiple scientific applications with a multi-disciplinary, multi-institutional and multi-platform background. This enables the automatic processing of the scientific computing in the so-called scientific workflow. However, this comes with a price. The experiment clearly shows that the data staging performance is significantly impacted. When a more sophisticated method is introduced, such as the extra application layer protocol HTTP/HTTPS, the data staging performance is further reduced which means a higher price. This one more layer brings the advantage of enabling your service to be publicly searchable and accessible with a public protocol for communication rather than a proprietary protocol that the others do not know. Therefore, the suitable solution for one's scientific computing process can be fitted into the following three categories:

1. Single-disciplinary, single-institutional and single-platform application should follow traditional methods to reach a higher computing performance;

2. When there is a necessity for integration of multiple applications that may be developed separately by people with different knowledge background, from different institutions, a network based workflow architecture should be considered. If it is just for proprietary usage, there is no necessity to introduce one more layer of application layer protocol. For example, socket is enough for this case.

3. When the applications are from last category, and at the same with a demand for public searchable and accessible services, the sophisticated extra layer of protocol should start to play its role, such as HTTP/HTTPS.

It looks like it is impossible to hold the two ends of the stick at the same time: the performance, or the integration ability and public accessibility. The adoption of containerization technology gives the chance to shorten the stick. On one hand, data staging can be carried out through local file system, which brings better performance. On the other hand, heterogenous OS and dependencies can still be maintained for each service at the same time, which allows the reusability of services that they can be integrated into any workflows with lower technical hurdle. Additionally, the ROA based framework proposed in this thesis enables the public access based on a standard protocol.

5.5.2 The Impact of Adoption of Containerization

Technically, the intuitive reflection on the scientific applications from multi-disciplinary, multi-institutional and multi-platform background is that, they have different dependencies on libraries and OS. This is one of the important hurdles why they need to be deployed in different hosts and communicate through a network, especially the data-flow. By adopting containerization technology, this hurdle is overcome. The scientific applications with different dependencies can be hosted on the same physical machine, which becomes the foundation to improve the data-staging performance - because they use the same file system.

In the experiments, it can be observed that by involving the containers, the data staging performance increases with a tiny amount in both of the tested modes for the comparison between Figure 5-12 and Figure 5-15 as well as Figure 5-13 and Figure 5-16. More importantly, this experiment assumes that the input is not in the file system originally. When the I/O data is generated by the applications on the host, there is no extra data staging needed. Inputs are already in the file system. But this atomic data staging step happens all the time in the WS based workflow, which multiplies the overhead. For containerized services, even in the worst case, the data need to be re-located to the sandbox of next application, this will only lead to a copy operation in the file system.

However, there is also a positive side-effect. As mentioned earlier, there are also other non-technical reasons for the applications/data to be distributed on multiple hosts:

- Limited hardware resource on one machine
- Licensing or legal requirement that certain service has to be deployed in certain network or machine
- Legacy services for which migration can be costly on both manpower and time

The adoption of containerization technology can hardly help on the latter two cases. The solutions for such types of issues are out of the technical range. Some of the services have to be deployed in certain machine on a certain place. For example, some NHS services. However, although it did not aim to resolve the issue at the first place, containerization technology may alleviate the issue in first case when the machine has to be deployed as a VM system. In the experiments, it can be seen that the idle VM/-container (without turned off or deleted) will not cost extra hardware resource. They do not influence the overall performance significantly. It means that the computing environment, the container and all the dependencies within, have the same life cycle as the application. They do not occupy extra resource after the application stops. Normal VM systems cannot provide this mechanism, therefor they incur more occupation on hardware resources or longer time on turn on and off the VM.

5.6 Summary

This chapter further proposes a scientific workflow modelling framework based on containerization technology. It also adopts the data staging approach and SaaS platform introduced in Chapter 3 and 4. The experiments compare the data staging performance under different circumstances with various environment setting for workflow execution. The implementation of this framework and the demonstration of the usage in the experiment also shows its ability to enhance the efficiency of computing resources utilisation and increase the reproducibility of the computing environment for heterogenous scientific applications.

This chapter also analyses the limitation of such methodology in scientific workflow modelling. The basic idea is that by introducing cutting-edge technology like containerization, there is no guarantee it will increase the workflow execution performance in each specific circumstance. It also discusses the suitable solution for scientific workflow in different computing environments. However, based on that discussion together with the development status that scientific research becomes more computing and data intensive, the joint efforts of heterogenous of computing resources is the general trend in building scientific workflow. The framework introduced in this Chapter is aiming to improve the execution performance of this new type of heterogenous scientific workflows emerged in this trend.

Chapter 6

Declarative Services Composition Description and Speculative Enactment Engine

6.1 Introduction

In the enactment of scientific workflow, it is necessary to address support for the construction and deployment of composite services: one approach that has been explored as proof-of-concept, is to treat a complete existing Taverna[Oinn et al., 2004] workflow as a service to be executed, where the workflow description is the data and the program is the (Taverna) enactment engine. Similar functionality should also be achievable with Kepler[Ludäscher et al., 2006]. In Section 4.4.2, a case study is shown that composes and executes a workflow in Taverna. Section 4.5.1 also shows the importance of keeping the compatibility with the imperative manner composition, which already resides in those existing WMSs.

A more serious issue however, is the dependence on specific services, meaning there is a reliance on a service provided at a specific URL, as against a specification of a service by, say, its profile (in OWL-S terminology), and the late binding identification of suitable available candidate services close to enactment time. A preliminary effort to address the issue appears in [Chapman et al., 2007], based on a matchmaker that assumes WSDL format service descriptions, but a fresh approach that takes advantage of REST seems desirable when this is revisited. This chapter will show the advantages. Hence, with the support of the ROA based framework from previous chapters, the components of web service composition for framework proposed in this chapter aim to allow more users to build their own services, and take advantage of the power offered by service composition to enable collaboration. Finally, this ROA based framework as a whole, also aims to propose to take advantage of the availability of capacity computing facilities to support speculative enactment of services. Speculative enactment of services allows a step/part of the workflow/composite web service to be executed in advance that it may or may not be needed. This method can further extend the framework's ability on error handling. In case that an alternative step/part needs to be executed for error handling, this step can be taken in advance and the total execution time will be reduced.

Firstly, this chapter briefly revisit the discussion of existing solutions for web services composition as well as some relevant solutions concerned with workflows. Following that is an introduction to related work on declarative description for REST web service composition and speculative enactment of services. This aims to present how declarative description and speculative enactment can benefit web service composition. Deriving from them, the SOA based Speculative Service Description (SSD) and the corresponding runtime implementation are introduced. This chapter ends with an evaluation based on Petri-net simulation and a composite service case study.

6.2 Web Service Composition and Workflow Description

In Section 2.3.3 and 2.3.4, the existing researches and solutions have been introduced and discussed. In the discussion, the following facts are emphasized. For web service composition:

1. There is no binding from semantic web services languages to RESTful web services description. Semantic web services cannot be grounded to the RESTful web services in an straightforward way.

2. The are fundamental differences between SOAP and REST, in that SOAP is operation-based and REST is resource-based. In the design of semantic web service models, it naturally follows this operation-based style to be adaptable

to its grounding target. This basic architectural difference means that the key concept of resource cannot be reflected in those models.

For workflow description:

1. a description language that can define the execution dependencies among web services.

2. a runtime to support enactment of CWS derived from the description language.

Therefore, it is can be observed those solutions for web service composition and workflow description have significant limits for the ROA based architecture. They are categorized as *imperative* description that the algorithm to decide the order of the task execution is precisely prescribed[Pesic and Van der Aalst, 2006]. In another word, the execution order has been pre-defined before the execution of the service composition/workflow. In some cases, the workflow systems allow the changes on component instance or part of the execution order in runtime based on certain models. But the results are still imperative. It will result in many efforts to implement various changes over and over again. A complete flexibility on execution order cannot be obtained.

6.3 Declarative Description

In the earlier reviews in Section 2.3.3 and 2.3.4, a series of web service composition and workflow description languages were introduced. They work in a procedural way so that the language has to express the logic of the computation by describing its control flow: the explicit steps. With the review of existing technology on web services composition and workflow description languages, the topic of web services composition based on ROA, practically based on RESTful web services emerge consequently in this Section. To enable a more flexible composition description, which naturally supports late-binding and failure handling, we present an approach to declarative description based on ROA. In a declarative description, the functions and execution dependencies of services are defined in a declarative way, by organising its functionality as 1) a collection of resources and 2) relationships among them. Thus, a declarative description naturally aligns with the connectedness feature of ROA, in that the relationship among web services are explicitly described in a peer-to-peer manner. This feature has been noticed and there are some existing researches on the declarative description for composite RESTful web services. In this section, several solutions based on ROA are discussed, which inspire the design of languages for RESTful web services composition in this work.

6.3.1 Declarative Description Solutions for REST

Particularly, in this section, three declarative description solution will be introduced.

Resource Linking Language Alarcon et al. [2011] introduces a hypermedia-centric REST service description, the Resource Linking Language (ReLL). This work is motivated by the fact that few works have been proposed to support a RESTful web services composition model. Existing ones are mainly operation-centric and do not acknowledge the feature of connectedness in REST. For the purpose of creating REST web services composition, this paperAlarcon et al. [2011] works out the approach by consolidating the important REST constraint of HATEOAS (Hypermedia As The Engine Of Application State).

A two layer composition engine is proposed, which contains a description language and a composition model. An example of the description is shown in Listing 6.1. It contains links for following resource operations, which can be dynamically generated at run-time, that serves as a late-binding mechanism in this engine. All the essential properties can be described, such as hyperlink, protocol, method, MIME-type, etc. The description is distributively embedded inside each of the resource representations. In this model, the machine-clients will be in charge of the parsing and generating work of the composition, based on the resource description, which makes it a rich client. Also in [Alarcon et al., 2011], a Petri-net model is demonstrated to evaluate a specific social network service composition execution.

Web Logic Programming [Piancastelli and Omicini, 2008] propose Web Logic Programming as a Prolog-based language for the World Wide Web embedding the core REST and ROA principles. It intends to represent the foundation of a logic programming framework for prototyping and engineering applications on the web as to follow its architectural principles and design criteria. The links, or relationship among web services or resources can be mapped and defined as axioms, which can be maintained as a knowledge base in the form of a Prolog

Listing 6.1: A ReLL Description Example[Brzeziński et al., 2012]

```
<resource xml:id="requestToken">
  <uri match="https://api.linkedin.com/uas/oauth/
     requestToken" type="regex"/>
  <representation xml:id="requestToken-text" type="iana:
     text/plain">
    <name>oauth_token request parameters</name>
    <link xml:id="authorizeRq" type="request" target="
       authorize" minOccurs="0" maxOccurs="1">
      <selector name="oauthUri" select="
         oauth_request_auth_url=[a-zA-Z0-9\-\%\.]+" type="
         regex"/>
      <selector name="oauthToken" select="oauth_token=[a-
         zA-Z0-9\-\%\.]+" type="regex"/>
      <generate-uri xpath="concat($oauth_url,'?',
         $oauth_token)"/>
      <protocol type="http">
        <request method="get"/>
        <response media="iana:text/plain"/>
      </protocol>
    </link>
  </representation>
</resource>
<resource xml:id="authorize">
  <uri match="https://api.linkedin.com/uas/oauth/
     authorize\?oauth_token=[a-zA-Z0-9\-\%\.]*" type="
     regex"/>
</resource>
```

program.

For the purpose of reflecting the connectedness property of REST, each resource is mapped as a predicate and the links among resources are defined as axioms in Prolog. The execution of an application is driven by predicates, axioms and inference rules in Prolog. [Piancastelli and Omicini, 2008] define a key concept, called Context, to organize the theories and provide a rule search space in one web application. The concept of Context is built based on the hierarchical naming structure of the Uniform Resource Identifier (URI), which implies that the links among resources or web services are mainly inside one web application or

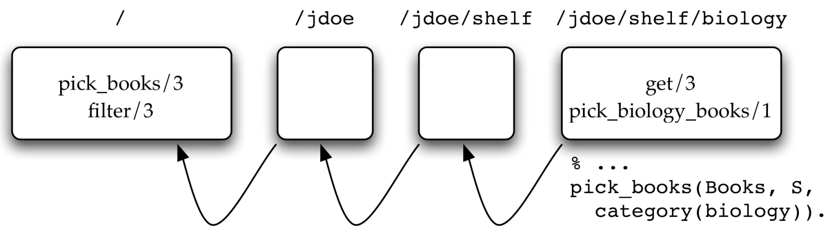


Figure 6-1: The /jdoe/shelf/biology resource responds to a HTTP GET request by eventually invoking the pick_biology_book/1 predicate, which in turn calls pick books/3; the context is traversed until a proper definition for it is found in the resource. [Piancastelli and Omicini, 2008]

Listing 6.2: ROsWeL Entry Rules[Brzeziński et al., 2012]

```
onGet | onPut | onPost | onDelet (<process_URI>,
      <entry_req_struct>, <entry_res_struct>) :-
            [any language instructions].
```

Listing 6.3: ROsWel Invocation Rules[Brzeziński et al., 2012]

```
// syntax of invocation
get | put | post | delete (<service_URI>,
      <inv_req_struct>, <inv_res_struct>).
```

the same domain name. Figure 6-1 shows an example of how the resource is found based on context traversal. Namely, the system searches for the matching predicts by traverse all the branches in the URI. In the workflow environment, which needs support from multiple resources across a variety of domain names, this context traversal based approach may confront new issues, or at least, an extension or example is needed to show its ability to search for other resources across a variety of domains.

ROsWeL Workflow Language [Brzeziński et al., 2012] propose a declarative business process language to support web services compatible with the REST paradigm. They call it the RESTful Oriented Workflow Language (RoSWeL). It introduces a declarative, goal-oriented approach to describe service composition and business processes. Language has a syntax similar to Prolog, augmented with operations supporting the REST paradigm.

The core of this language is the entry rules and invocation rules based on REST uniform interface, which are shown in Figure 6.2 and 6.3. Entry rules serve as the initial point of the business process, which may contain more invocation rules. Invocation rules drive the process by invoking the resource through a specific HTTP method. Business processes can be described in an way that maps directly to the REST uniform interface, which is the main innovative point and difference from other solutions. Based on these rules, the business process engine works server side, which is the centre for both control- and data- flow, to execute the business process of the CWS written in RoSWeL. This language not only aims to provide a web services composition description, but also to embed complete business process code, such as printing, HTML form reading, XML parsing shown, etc. in the form of Java language, as shown in examples in List 6.4 from Brzeziński et al. [2012]. The Java code is embedded in the position from line 18 to 21. In this case, it increases the complexity of the description language. It does not only describe the relationship among services. Obviously, the implementation of this language needs sophisticated run-time support for multiple programming languages to support the business process codes. However, one key feature of REST, MIME-type based representation, is not reflected in this language. In REST web services, one resource can be presented in multiple representations or formats, such as a workflow result can be expressed in say either XML or JSON. They are all defined by Multi-Purpose Internet Mail Extensions (MIME), which works as a protocol between client and server.

Other Related Solutions Besides the declarative description language solutions and Semantic web composition based solutions described above, there is also another workflow-based solution called *DECLARE*[Van Der Aalst and Pesic, 2006] that enables late-binding of web services in workflow at run-time. However, its application is dependant on YAWL, which is a procedural solution that specifies compute procedures at design-time. The workflow's main procedure is still described in YAWL. Some of the components are described in *DECLARE* to provide a form of late-binding support. This hybrid solution is primarily designed to serve SOAP based web services.

6.3.2 Discussion on Declarative Description Languages

In the preceding sections, we have reviewed and analysed web services composition, workflow description languages and declarative description languages. Based on these, three design requirements for REST web services composition can be further identified for the case of scientific workflow:

1. For the purpose of building a practical web services composition system, two essential components are indispensable. They are *declarative description language for web services composition* and *a runtime for composite web services enactment*. It is derived from the two major phases for workflows, which is also reflected in those practical WMS examples discussed in Section 2.3.4.

2. The design of declarative description language must embody the guidelines of ROA, which are *Addressibility, Statelessness, Connectedness, Uniform Interface*, as introduced in Section 2.1.3. The language should be able to deal with the key elements in the implementation of ROA, such as URI links, HTTP interfaces, representation type, etc.

3. The design aims to enable failure handling under complex network conditions, so that the significant performance impact arising from the introduction of the new design can be mitigated. As part of failure handling, the design of runtime should enable late-binding of web service. Furthermore, in scientific workflow, it is not rare to have computations that occupy huge spatio-temporal resources. To overcome this new challenge, it is also worth further exploring the performance potential of declarative description in the circumstance of scientific workflow.

First of all, to meet the requirement on dealing with the new challenge, the speculative enactment engine is introduced into the design of the runtime, which is discussed in Section 6.4.

6.4 Web Services Based on Speculative Computation

Speculative execution is a tentative computation to improve performance of an application by performing some task that may not be actually needed. When apply this technique, another important idea is that, there is necessity that the speculative component not have a side-effect that (incorrectly) affects the main thread in the case the speculative computation is not needed. This technique has been employed in a wide range of applications at a variety of levels, including branch prediction in pipelined processors, prefetching memory and files, and optimistic concurrency control in database systems [Kung and Robinson, 1981, Lampson, 2006, Raghavan et al., 1998]. For example, similar concepts also appear in the time warp mechanism[Jefferson, 1985] for organising and synchronising distributed systems. It is a lookahead-rollback synchronization mechanism by rolling back to the time just before the conflict caused by the side-effect.

[Fukuta et al., 2008] propose the concept of "Kiga-kiku" service that drives web-based business workflow in a speculative manner. The service is able to predict users implicit demands and proactively protect the users from potential failures in the service processes. The primary goal is to use speculative computation for preventing potential failures that would be caused by 'late decisions, which may also caused by human intervention in business workflow. In such case, multiple web services, some of which may not be needed, can be invoked concurrently to prevent the potential failure of some of the web services. In their case study, time can be saved because decisions do not need to be delayed until after failures occur. They define the "kiga-kiku" service in natural language that can be summarized as:

1. The user has initiative and the background speculative computations are invisible to user.
2. The service predicts the users intention, preferences, and possible failures with respect to the current goal.
3. The service performs the proactive action with best predicted value.
4. The service can seamlessly guide the user to switch to an alternative way to accomplish the goal when failures happen.
5. The service can revert inappropriate side-effects caused by proactive actions.

[Fukuta et al., 2008] mainly focus on the formal definition of the "Kiga-kiku" service and an implementation that is oriented towards a particular case study. It does not address how the relationship between web services should be formally described declaratively nor how the description should be processed with a general purpose run-time, such as that needed for the scientific workflow scenario. These shortcomings are what this thesis aims to address, based on the work introduced in Chapter 3 and 4.

In the work of Fukuta et al. [2008], as a conceptual computation model, they assume the existence of appropriate services, service descriptions, service composition mechanism, etc. These are the key components for the realization of such a speculative computation based web services composition mechanism.

In a scientific workflow environment, scientific applications may take considerable length of time to complete one task. In addition, the networked environment cannot be absolutely reliable, so there is also risk that key data or a step in the workflow may fail in this process. It is highly desirable to reduce the risk of and save time from web service failure in scientific workflow – and indeed any workflow. Compared with other solutions in the literature above, the "Kiga-kiku" method provides a solution not only for how to compose and execute services based on the declarative description, it also enables a composite service to perform the proactive action, as well as to handle failure in scientific workflow. The next section introduces the design and discusses the implications of the declarative description and the speculative enactment runtime.

6.5 Design and Implementation of Speculative Service Description (SSD)

In above sections, a series of related techniques and approaches have been analysed. It is practical to narrow down the design requirements of this Speculative Service Description (SSD) based on the analysis on their advantages and disadvantages in the context of a REST based scientific web services composition. The requirements are consolidated as:

1. SSD realizes the declarative description of web services and links between services to provide a formal declarative description of the relationship between web services.
2. SSD enables the description of RESTful resources and is compliant with REST constraints to provide a ROA based systematic solution that also benefits from the works introduced in Chapter 3 and 4.
3. SSD supports the implementation of speculative services, that the workflow enactment runtime can be built based on SSD.

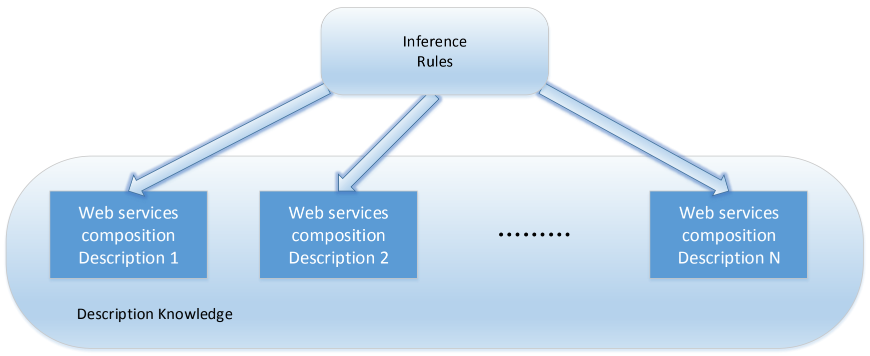

Inference Rules
Web services composition Description 1
Web services composition Description 2
....
Web services composition Description N
Description Knowledge

Figure 6-2: Modularization of SSD

SSD is built on top of Prolog language, through which the function of inference at runtime is provided. There are further two levels of modularization in the language design, as shown in Figure 6-2, where there are two components: inference rules and description knowledge. Inference rules are the static part in the description, while description knowledge is the dynamic part that contains the web service description and links. This aims to simplify the description syntax and ease the process of describe new web service compositions.

In SSD, the execution engine is separated from the description, in these cases, the Prolog-based description. The Prolog-based description itself will not drive the execution or call any external codes in its inference process. It only return the inference result to a separate execution engine. Based on such a design, one other benefit is that the connection between execution engine and CWS description is decoupled. Therefore, the connection can be a system call on the same physical machine or itself a web service. This design aligns with the statelessness constraint of REST.

6.5.1 A Description Language for Stateless Web Services

Both of the languages in [Brzeziński et al., 2012] and [Piancastelli and Omicini, 2008] adopt a design in which the links between the web services in the service composition

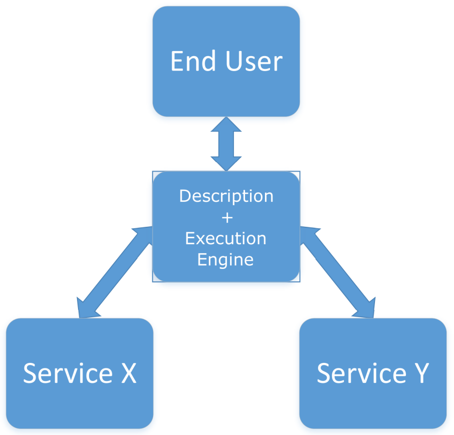


Figure 6-3: Description resides in Execution Engine

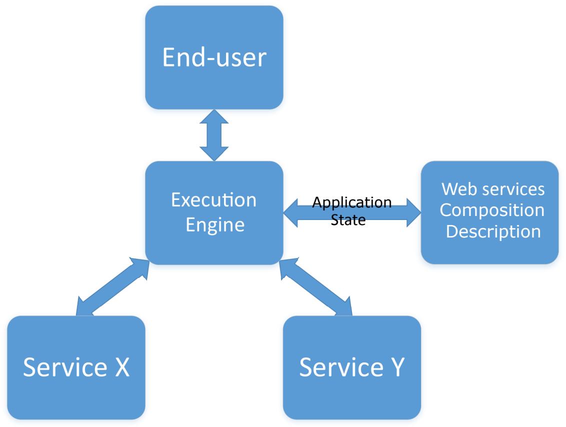


Figure 6-4: Description is Separated with Execution Engine

are invisible to end-users, because the composition description resides in the execution engine. From a user's point of view, it is merely a black box. This black box service is the reason for the lack of the ability to provide interfaces to end-users for the purpose of obtaining the information about the web application's state, such as the overall goal, achieved tasks, following tasks, etc. It can be observed that there is a high-coupling between the execution engine and the knowledge about the descriptions. This implies that the application state which drives the execution of CWS or web application are stored and maintained at server-side. There is no channel to feed the state of the service composition or web application back to the end-user.

Listing 6.4 shows a segment of ROsWeL description [Brzeziński et al., 2012]. It illustrates a business process connected with the visit of a patient in the clinic laboratory. In this example, to achieve the *onPost("Doctor/fSSNg", WFReq, WFResp)* goal, several tasks must be achieved, such as *print, readForm, makeTest*, etc. They are executed either by embedded Java code or other services (*put* predicate in this example) directly in this description. It is lack of mechanism to expose the application state to end-users who call the service, for an instance, "Doctor/fSSNg" in this example.

Figure 6-3 shows the architecture of web service composition with declarative description residing in the execution engine. In contrast, the declarative description of the CWS is separated from the execution engine in Figure 6-4. In Figure 6-4, each declarative description instance of one workflow is transferred in full between the execution engine and declarative description knowledge base. In Figure 6-4, if the *execution*

Listing 6.4: Segment of ROsWeL description example

```
onPost("Doctor/fSSNg", WFReq, WFResp):-
print(SSN),
readForm("../name.html", Form),
sendToClient(Form,Res),
makeTest(Res->FORM->NAME, TestResult)
......

print(X):-
<I-- --I>
<J--
Variable v = X;
System.out.println(v.toString());
return v ;
--J>.

readForm(FilePath, Result):-
<I-- --I>
<J--
// Java code used to read Html form from
// file and assign it to 'Result' variable
--J>.

makeTest (PatientName , Result ):-
put("http://labTest.com/fPatientNameg" ,
PutReq, PutRes),
Result = PutRes.
......
```

engine is located on server-side, the architecture appears to be similar to Figure 6-3 from the end-user's point of view. However, by virtue of the separation, the web services composition description can be shared and accessed explicitly (for each of the description instances, it has its URI).

In this design, some essential data of application state are transferred from execution engine to description knowledge base as a query for inference. They are:

1. The initial input data types for the CWS execution;

2. The service as the goal of the CWS execution;

Query	URI
Initial	http://<domain>/Init?KB_URL=<Knowledge Base> &G_URL=<Goal service>&I_Names=<Input types list>
Next	http://<domain>/Next?KB_URL=<Knowledge Base> &L_URL=<Last successful service>
Alternative	http://<domain>/Next?KB_URL=<Knowledge Base> &L_URL=<Last successful service>&N_URL=<The last failed service>

Table 6.1: The URI templates for inference query

3. The last service which just executed successfully in the CWS execution;

4. The service which failed in last step in the CWS execution.

Correspondingly, the application state that drives the execution are returned as an inference result from description knowledge base. They are:

1. The initial service of the web service composition execution;

2. The next service to be executed in the execution;

3. The alternative service to replace the failed one in the execution.

Table 6.1 shows the URI template for the query from the execution engine to SSD knowledge base. After get the execution request from end-user, the execution engine will launch these queries to steer the execution. There are three types of queries: (i) the query to initialise the composition execution. It needs a goal and the types of the input the execution engine have at this stage; (ii) the query to get the next step in execution. It needs to know that which is the last successfully executed service in the workflow or the current state of this CWS execution; (iii) the query to get an alternative step in execution. It needs the last successfully executed service as well as the last failed service that needs to be replaced. In all three types, the SSD knowledge base needs an identifier for the knowledge base which the execution engine is querying. Each knowledge base represents a specific workflow with all the related web services and links being described. The details of the syntax of this Prolog-based knowledge base is introduced in section 6.5.2. The reason to choose Prolog is: (i) by nature, it is a declarative language which fits for the design of SSD; (ii) there are examples of using Prolog for composite REST web services description; (iii) as a general purpose logic programming language, it has good implementation and can be used together with

Listing 6.5: Inference Rules in SSD

```
<query tag>( [ [optional parameters of web services] , ] ) :-
   [ [expression of relations between services]
   , [optional features of web service] ] .
```

Listing 6.6: Relation Facts in SSD

```
child_of(<Child_Service_URI>, <Child_Service_Method>,
         <Parent_Service_URI>, <Parent_Service_Method>,
         <Child_Service_Priority>).
```

Listing 6.7: Detail of Web service Facts in SSD

```
details_of(<Service_URI>, <Service_Method>,
         <Input_MIME>, <Output_MIME>,
         [Input Local Name], [Input Global Name],
         [Output Local Name], [Output Global Name],
         [Input Workflow Tokens], [Output Workflow Tokens]).
```

some OO languages, such as Java in this thesis's implementation, through existing libraries as well.

6.5.2 First-order Logic in SSD

The main component of the SSD is a set of rules. It contains two types of rule: (i) the rules for the inference process, which is generic to all the CWS description; (ii) the rules to describe certain properties and relations, which is specific description to each of the CWS.

The rules for the inference process consists of a header, and a set of bodies. The left side of the rule (the header) defines a goal, which represents the query from the execution engine. As described in Table 6.1, there are three types of queries, which can be represented by three tags: *initial, next and alternative*. The right side of the rule (a body) specify the conditions, which has to be fulfilled to reach the goal. Listing 6.5 shows the template for the inference rules. For different tags, the rule includes different parameters or expression on constraining relations between services and features of web service. The feature can be the URI of a web service, the I/O MIME types and so on. The core part of the Prolog code is presented in Appendix B.

In Listing 6.5, the inference based on both of the types of rules can be true or false. It implies that the conditions in the rule are fulfilled (the body is true) then the goal is reached. There are results that fulfilled the query. The results will be returned to the execution engine for further execution of the CWS. The web services are constrained by the relations between certain web services, such as Service A is Service B's next step in workflow. They are also constrained by some certain features of the web services, such as the Service A with certain function must be the final step of this workflow. These constraints are all written Prolog.

Each of the CWS description consists two parts: properties of the web service and relations among them. The goal in these rules can always be reached because it has only header without body. Such rules are called facts. Listing 6.6 shows the template for the facts to describe the relations between web services. It contains the basic information about the directed link between two web services. Additionally, it also contains an integer (1...n) to represent the priority of this link, which comes from the knowledge of the workflow developer. In the case of multiple links from one web service, the highest priority link is matched in the inference process. The priority also important for inference engine to find the alternative link.

Listing 6.7 shows the fact template to describe the properties of a web service, such as the URI, method, I/O MIME, I/O names. The I/O names have two sets, one is the so-called local name to identify I/O data within the scope of one instance of web service. Global names can be used to identify I/O data among multiple instances of one web service. For example, in one simple weather forecast web service, input is date time, output is temperature. There is only one web service. However, in a workflow, there can be multiple instances of this web service to query multiple days of temperature. In this case, *DataTime* and *Temperature* will be local name to describe the data type. Some names like *DT_Day1, DT_Day2, Temp_Day1, Temp_Day2* can be the global names to identify each data item in a workflow. There are also workflow tokens to control the workflow execution. This is will be further discussed in Section 6.6

6.5.3 Ranking for Alternatives

The declarative description of workflow and its runtime, which will be introduced in Section 6.6, do not directly provide a ranking algorithm for web services. There are active researches on ranking algorithm for web services and this study is out of the scope of this thesis. However, our description can provide an internal ranking mechanism and be able to provide an interface for external ranking information. The ranking information can be the result generated based on any ranking algorithm. In this section, the internal ranking mechanism will be introduced and the ability to link with and utilize external ranking information will be discussed as well.

The ranking description in this declarative description is designed in a *local ranking* manner, not a *global ranking* manner. It means, for each group of the web services that are ranked together, is only a rank for all possible children of a certain service. This also matches with the way how generally web services ranking system works that query for a certain service requirements is sent, then the all possible services information is returned with a rank for them [Al-Masri and Mahmoud, 2007a,b, Guinard et al., 2010, Panziera et al., 2011]. The ranking information is associated with each *Relation Fact*, rather than *Web service Facts*. All the services in one declarative description instance will not be ranked together in a global manner. Thus, there are multiple ranking tables in one declarative description instance. The purposes of applying this manner is that each of the group can be populated independently based on the external ranking information. The other services' ranks do not need to be recalculated and repopulated in a global manner.

The example for the ranking mechanism in SSD is shown as List 6.8. The first two *Relation Facts* in List 6.8 represents the two available services as the direct children of the service https://example_1.com/parent/ in this workflow DAG(directed acyclic graph). One has priority 1 and the other has 2. Thus, the service with priority 2 is the alternative of service with 1 in this case. If there is a priority 3 service, it will be the alternative of the priority 2 service. The ranks are in ascending order. The last two facts are two available services as the direct children of https://example_2.com/parent/. Priority 1 and 2 are also allocated to each of them, which are isolated from the first two *Relation Facts*. In such a design, if one of the services groups need to repopulate or swap the priority, the other groups priority will not be influenced. In the case of *Relation Facts*, service https://example_1.com/child_1/ is shared by both of the ranking groups. Even in this case, it will be influenced as well.

A concrete example based on existing ranking framework is shown in this paragraph. Here we take the ranking approach introduced in Panziera et al. [2011] because it is designed for RESTful web services. It works in the manner of one way request/re-

Listing 6.8: Ranking example in Relation Facts

```
...
child_of(https://example_1.com/child_1/, GET,
         https://example_1.com/parent/, GET,
         1).

child_of(https://example_2.com/child_2/, GET,
         https://example_1.com/parent/, GET,
         2).

child_of(https://example_3.com/child_3/, GET,
         https://example_2.com/parent/, GET,
         1).

child_of(https://example_1.com/child_1/, GET,
         https://example_2.com/parent/, GET,
         2).
...
```

spond. Clients send request with service query. Server returns the response with services and ranking information. The communication is also carried out via RESTful web services. The ranking information of certain type of services can be referred to an URI as http://SMEendpoint?rpuri=URI. In this case the SMEendpoint points to the ranking service resource. The *rprui* represents the so-called *requested policy* compiled by the user. The *requested policy* contains the requirements to the needed service. Finally, the service will return a response with a list of matching services and *Global Degree(GD)* of them. The GDs of matching for all the services are used to deliver a ranked list of web services. The ranked list can be used to populate the priority field in *Relation Facts*.

6.6 A Runtime System for Speculative Enactment of Workflow

6.6.1 System Architecture

In Section 6.5, three types of queries were introduced, which form the core to drive the enactment of CWS based workflow. Figure 6-5 depicts the basic relations among these three types of queries in the enactment of web service composition as described by SSD. The composition runtime works normally without speculative enactment, so that no web service will be invoked speculatively, but rather only when they become necessary in the workflow.

In Figure 6-5, the runtime starts with a query to find and execute the web services to initialize. After the execution of each of the web services, there will be a decision to be made on whether this service has executed successfully. If it has not, the runtime will try to find an alternative web service and execute it. If it is, the process will check if the whole CWS has been completed. If the whole process is not complete, the runtime will try to find the next web service. A different process happens on the search and execution of alternative web service. If it can be successfully found, it will be checked as all the other web services on whether it has been successfully executed. The other result is that the whole process will be halted, because no further alternative solution is available.

The runtime in Figure 6-5 provides an exception handling mechanism by searching for an alternative solution. However, the feature of speculative enactment is not reflected by this single-thread runtime. A speculative runtime must provide the following basic functions:

1. the ability to execute a web service and all the alternative ones speculatively in parallel;

2. the ability to collect the result or final result from the first completed solution;

3. the ability to stop the redundant services and release the resources when the result is already collected

In addition, in order to manage the a concurrent system that can achieve the above requirements, a control mechanism is need to serve as a bridge between threads to avoid conflicts, such as serializing tokens.

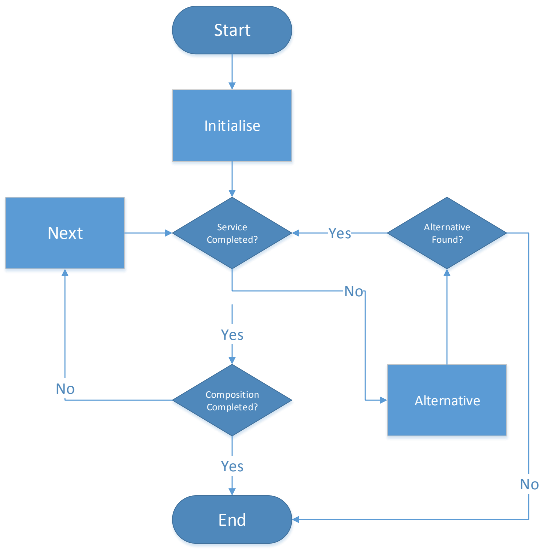


Figure 6-5: The flow chart of runtime based on SSD without speculative enactment

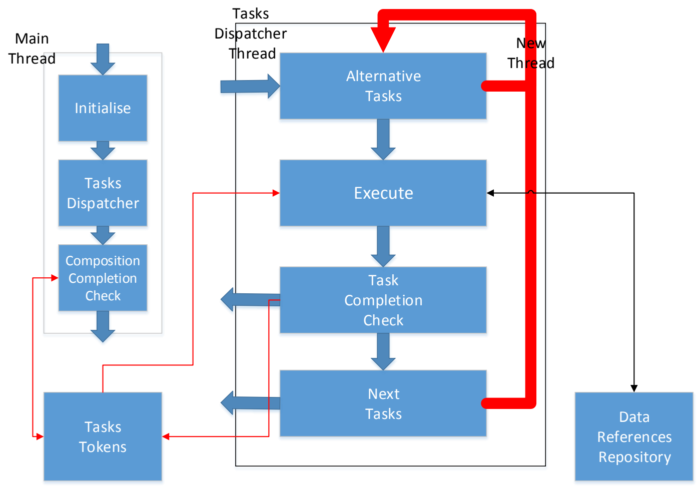


Figure 6-6: The flow chart of runtime based on SSD with speculative enactment

Figure 6-6 depicts the multi-thread based speculative enactment runtime. There are two types of threads: the main thread and service dispatcher thread. The main thread can initialise and finish the enactment of a web service composition. The service dispatcher thread is in charge of the management of web services, which can be initialised either by the main thread or another service dispatcher thread. The first task dispatcher thread, which is initialised by the main thread, starts the management thread of web services with all the alternative management threads of web services concurrently in multiple threads manner. Each of these threads will search for next web services and execute them by more threads recursively. The service dispatcher thread can kill itself in two ways: (i) the web service execution is unsuccessful; (ii) there is no next web service to be executed based on the query from knowledge base. It can also be killed by the main thread when the final result has been collected by the main thread. This means the whole composition has been completed and resources should be released.

The management of threads is supported by the task token mechanism in SSD. The design borrows the concepts in Petri-net [Peterson, 1981], which is a mathematical modelling languages for the description of distributed systems. A Petri net consists of places, transitions, and arcs. Arcs run from a place to a transition or vice versa, never between places or between transitions. The places from which an arc runs to a transition are called the input places of the transition. Correspondingly, the place linked from a transition are called output places of the transition. Places in a Petri net may contain a discrete number of marks called tokens. A transition of a Petri net may fire (executed) if there are sufficient tokens in all of its input places. In the case of speculative enactment runtime, each web service can only be executed when specific tokens have been collected. Those tokens are described as [Input Workflow Tokens] in List 6.7. And it will only generate specific tokens after it is successfully executed, which are described as [Output Workflow Tokens] in List 6.7. Each token can be represented by any string, which can be defined by users for identification purpose. Therefore, this task token mechanism has such two essential functions: (i) it maintains the dependency between web services that web services can be executed concurrently in the right sequence; (ii) the resources occupied by speculative web services can be collected when it is discovered they are not needed, such as when another web service has successfully done their work. Those speculative web services can be killed in two ways: (i) the main thread has collected the final result; it then kills all the web services threads that are still executing; (ii) the service dispatcher thread kills itself when it discovers the task token it aims to generate is already in the task token repository.

The details of the token checking mechanism is shown in Figure 6-6. The Execute component checks the token repository when it starts the execution. The execution will start only when the necessary tokens are in place. The Task Completion Check component checks if the tokens this web service generated have been collected. If it is so, this service dispatcher thread will kill itself. On the other hand, whenever the task token repository is updated, the main thread will check it to determine if it needs to complete the whole process and kill the other ongoing web services. In addition, there is an independent data references repository. All the execution components communicate with it to retrieve and store data references. Each data item is labelled by the I/O global name introduced in Section 6.5.2.

In reality, the enactment runtime is exposed as REST web services, which are written in Java and support by Jersey library[1]. Users can call the web service to start the workflow execution[2]. The configuration file for this service contains the URI of the composite service's description, which is stored as plain text. The threads mentioned in last paragraphs are actual threads in the Java based speculative enactment runtime. *Data References Repository* is written and deployed separately as an REST web service. It uses the URIs presented in Table 6.1. In the workflow execution, the runtime invokes the exposed web services of *Data References Repository* to collect information about execution sequences. This invocation is invisible to end-users. The so-called token repository is a simple List structure in Java to store string at its basic level, which is instantiated and associated with each workflow execution when the workflow is started.

6.6.2 Speculative Execution on Parts of the Workflow

Based on the token mechanism in Section 6.6.1, the runtime can speculatively execute alternatives for one service. In this case, all the services, which include the priority 1 service and all the other alternatives, will generate the same output tokens. The runtime will ensure if one of the service is successful, the other alternatives will be killed to save resources. Additionally, this token mechanism can be applied on parts of one workflow to enable the speculative execution of certain part of the workflow, which has more than one service.

[1]Jersey: https://jersey.java.net/

[2]The URI template is http://.../composite/{Service_Name}

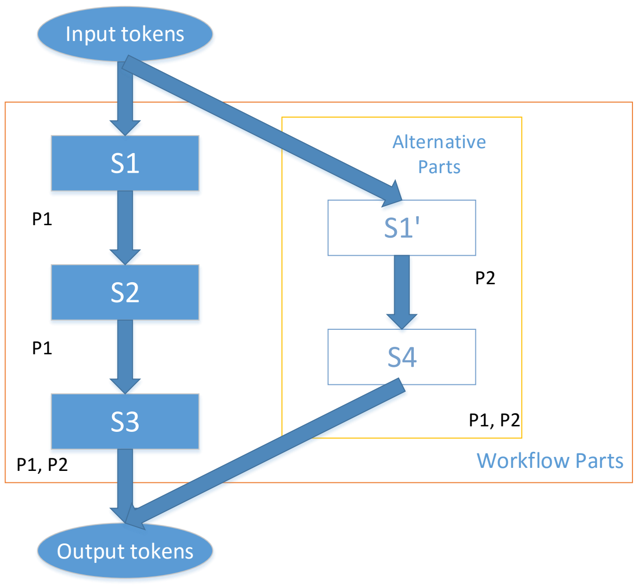


Figure 6-7: Speculative Execution of Parts of Workflow

As shown in Figure 6-7, one part of workflow is extracted to demonstrate the mechanism. The thread S1'+S4 is the alternative part of S1+S2+S3. First, S1' is the alternative service of S1 and it leads to the alternative part. They need to consume the same set of tokens in order to be executed. In such case S4 has no alternative relationship with the other following services S2 and S3 in the main part. The second important setting that makes they are alternative to each other is, S3 and S4 generate the same output tokens. If there are more alternative parts of S1+S2+S3, their final services generate the same tokens as S3 and their first services also are the alternative services of S1. In this speculative execution environment, all the parts start to execute at the same time.

As introduced in Section 6.6.1, the service dispatcher thread will kill itself, which contains a running service or a service of certain part in this case, *when* it discovers the task tokens it aims to generate is already in the task token repository as shown in Figure 6-6. Thus, in the example of Figure 6-7, when S3 or S4 finishes, they other alternative thread will be killed. However, there is a possible disadvantage on resources. The alternative thread will only be killed when it arrives the beginning of the final service of its part. This can be sorted out by further customise the tokens generated by the services for this very part of workflow.

The black marks locate beside the bottom corner of each service in Figure 6-7 represents their output tokens. Each thread of the part has its own unique and unified output tokens except the final service. The final service will generate the same tokens and they are the collection of all the unique tokens from each part. For example, main thread has token P1 and alternative thread has token P2. The final service of each thread will generate the same token P1+P2. If there is a third thread or even more thread, they will have the unique token P3 or Pn and the final token will be P1+P2+P3+...+Pn. Each of these Pn tokens can also represent a collection of tokens as long as they are unique from each other. In this case, it guarantees that when any of the final service is finished, all the other thread will be killed regardless which step they are in. Besides that, it will not kill any of the thread if the final service is not executed. Based on such a mechanism, the original token checking algorithm does not need to be modified. By setting the tokens carefully, speculative execution of parts of workflow can be processed without occupying unnecessary resources.

6.6.3 A Discussion on Time/Compute Resource Decisions

To have a speculative execution system for a web services based workflow, the primary goal is to save time. However, the saved time come with a price. The web services speculative execution needs extra computing power. The simplest model is that the user need to pay for the service for a certain price. The price can be calculated in an one-off or pay-as-you-go manner based on the final executing time. Surely, if we do not put the resource related decisions in a context of "Low Carbon Earth"(which is obviously out of the scope this thesis), and believe the market and economics will work out the best solution for service providers, we do not need to consider *Time/Compute Resource Decision* when the services are free. The workflow just need to run as many as possible web services to save time.

Then the question is simplified as, *how much are you going to pay for your time?* The speculative execution system can be designed to have such a blank field to let the user to fill in this number. However, the final decision has to be decided before this filling action by the user. Then, at the system level, in the case that the price is known, the question will be, *how much time indeed can be saved by this speculative computing*? This has to be answer from a more basic question, *why we need a speculative execution system?* If all the services can be executed in one go without failure and can serve for our purpose perfectly, there is no necessity to have such a system. It is designed for all the unexpected conditions. Then the question can be simplified as, *how unexpected all these services are*? Or by asking, *what is the occurrence probability that a web service will fail or its result has to be discarded?* This will be a probability calculation based on experience. This may has been included in the service-level agreement(SLA) provided by the service provider. Additionally, this can be counted by user based on experience data to get a more customised result.

With the existence of the following three numbers: (i) the price the user can give for his/her time; (ii) the probability that one alternative service's result will be adopted; (iii) the price needs to pay for the execution of this alternative service, a formula can be worked out for the decision of speculative execution of an alternative service.

$$D = M_t - M_s/(P_s * T_s) \tag{6.1}$$

In Equation 6.1, M_t means the price the user would pay for time. M_s represents the price needs to pay for the execution of a certain alternative service. P_s is the probability that this alternative service will be adopted, which also means the probability of all the higher priority services will fail. T_s is the service execution time of the alternative service. If T_s is longer than the execution time of the service with priority one, the execution time of the priority one will be used to replace T_s here. The reason to have such a replacement is that even when the longer alternative service is executed in parallel, the maximum time can be saved is only equal to the execution time of the priority one service. When D is equal or greater than 0, the alternative service should be executed as speculation. Or it should not. This equation can be also interpreted in such a way that, the higher price the user can set for his/her time, the higher chance the alternative should be executed. The lower the price to pay for the execution of alternative service, the higher chance it should be executed. The higher the probability this alternative service will be used, which also means all the previous services are failed, the higher chance it should be executed. And finally, the more time can be saved, the higher chance the alternative should be executed.

$$D = M_t - S * M_s/(P_s * T_s) \tag{6.2}$$

This discussion can lead to an interesting future work. We can further update the equation as Equation 6.2. The S in the equation is a coefficient for the system sensitivity to financial cost. For example, if the S is higher, the lower chance the alternative service will be executed. Users can adjust their sensitivity to financial cost case by case by setting the M_t. The system administrator can set the coefficient S for the whole system in a more macroscopic view. To work out the appropriate coefficient and find out the appropriate price for certain user's price for time will finally decide the efficiency of the whole speculative execution system.

6.7 Evaluation

6.7.1 A Petri Net Model

This model aims to provide a formal foundation based on Petri Net [Peterson, 1981] for the speculative enactment runtime. It contains variables that can used to simulate certain workflow enactment environments, such as workflow procedure, relationship among web services, failure rate of each web service and execution priority of each web service, etc. By performing experiments on it, simulation results can be generated,

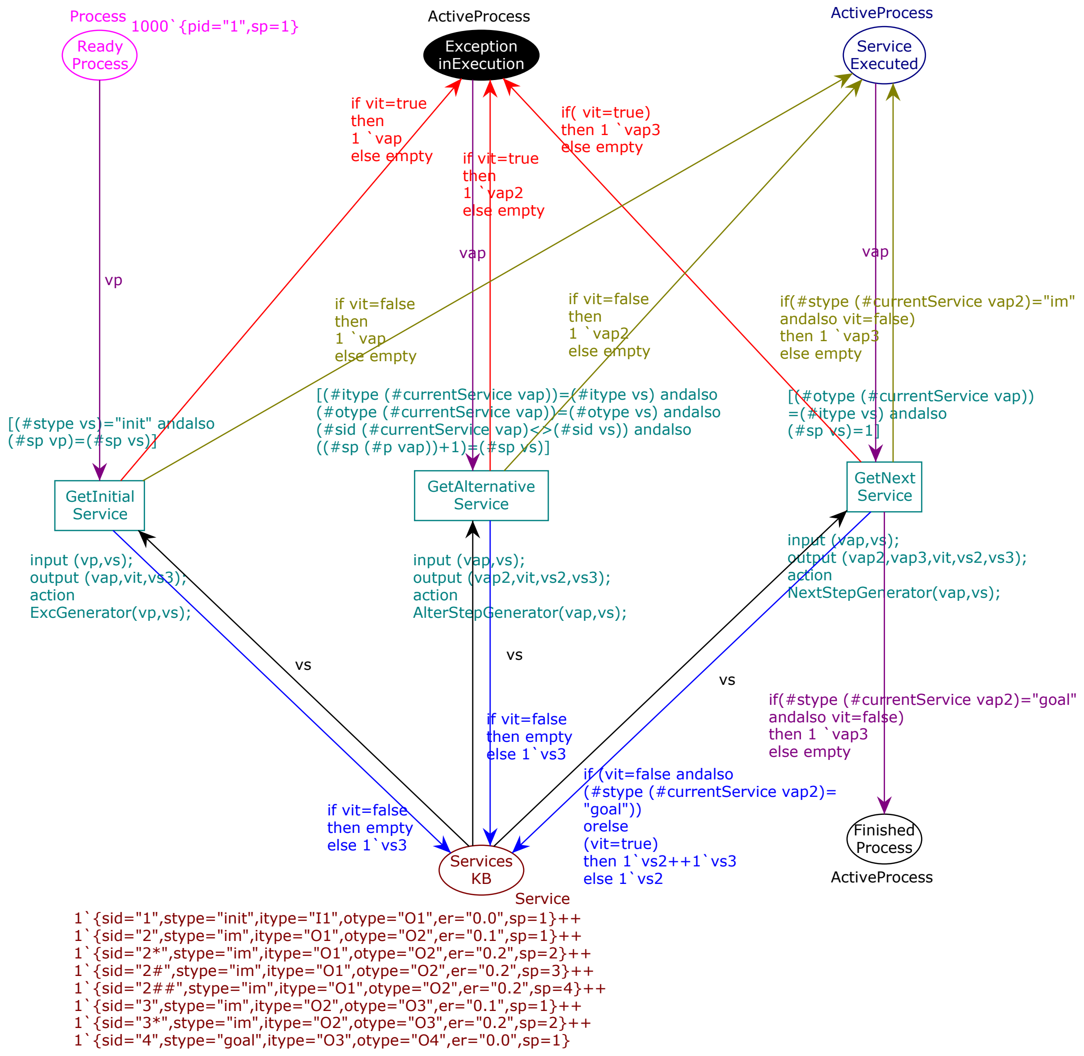

Process
1000`{pid="1",sp=1}
Ready Process
ActiveProcess
Exception inExecution
ActiveProcess
Service Executed
vp
if vit=true then 1 `vap else empty
if vit=true then 1 `vap2 else empty
if(vit=true) then 1 `vap3 else empty
vap
vap
if vit=false then 1 `vap else empty
if vit=false then 1 `vap2 else empty
if(#stype (#currentService vap2)="im" andalso vit=false) then 1 `vap3 else empty
[(#itype (#currentService vap))=(#itype vs) andalso (#otype (#currentService vap))=(#otype vs) andalso (#sid (#currentService vap)<>(#sid vs)) andalso ((#sp (#p vap))+1)=(#sp vs)]
[(#stype vs)="init" andalso (#sp vp)=(#sp vs)]
[(#otype (#currentService vap)) =(#itype vs) andalso (#sp vs)=1]
GetInitial Service
GetAlternative Service
GetNext Service
input (vp,vs); output (vap,vit,vs3); action ExcGenerator(vp,vs);
input (vap,vs); output (vap2,vit,vs2,vs3); action AlterStepGenerator(vap,vs);
input (vap,vs); output (vap2,vap3,vit,vs2,vs3); action NextStepGenerator(vap,vs);
vs
vs
vs
if(#stype (#currentService vap2)="goal" andalso vit=false) then 1 `vap3 else empty
if vit=false then empty else 1`vs3
if (vit=false andalso (#stype (#currentService vap2)= "goal")) orelse (vit=true) then 1`vs2++1`vs3 else 1`vs2
if vit=false then empty else 1`vs3
Services KB
Finished Process
ActiveProcess
Service
1`{sid="1",stype="init",itype="I1",otype="O1",er="0.0",sp=1}++
1`{sid="2",stype="im",itype="O1",otype="O2",er="0.1",sp=1}++
1`{sid="2*",stype="im",itype="O1",otype="O2",er="0.2",sp=2}++
1`{sid="2#",stype="im",itype="O1",otype="O2",er="0.2",sp=3}++
1`{sid="2##",stype="im",itype="O1",otype="O2",er="0.2",sp=4}++
1`{sid="3",stype="im",itype="O2",otype="O3",er="0.1",sp=1}++
1`{sid="3*",stype="im",itype="O2",otype="O3",er="0.2",sp=2}++
1`{sid="4",stype="goal",itype="O3",otype="O4",er="0.0",sp=1}

Figure 6-8: A Petri Net Model of Speculative Enactment Runtime

observed and analysed. It can serve as an evaluation tool for the design concepts based on the experiment designs, which simulate the target execution environments of scientific workflow.

Figure 6-8 shows the formalization of a speculative enactment runtime model of web service composition. A runtime model is a colored Petri Net represented by a tuple $\mathscr{S}$ =($\mathscr{P}$, $\mathscr{T}$, $\mathscr{F}$, $\sum$, $\mathscr{M}_0$, $\mathscr{G}$, $\mathscr{C}$), where $\mathscr{P}$ and $\mathscr{T}$ are disjoint sets of *places* and *transitions*. Places represent states that contain tokens with multiple attributes, and transitions represent activities that can be guarded; transitions are fired when all the tokens in the corresponding input place arrive. Different token types with multiple attributes are defined as color set. Places and transitions are linked through arcs. $\mathscr{F}$ represents all the arcs the link them. $\mathscr{M}_0$ is the initial marking in which some tokens are initially assigned to places. $\sum$ represents all the color sets, which allows tokens to have data values attached to them like properties of the tokens. $\mathscr{G}$ represents a set of guard functions assigned to each transition, which are Boolean expressions evaluating to *true* or *false*. Guard functions encode token restrictions controlling whether they can be fired. $\mathscr{C}$ is a set of functions for a transition to generate output tokens in certain color sets from the input tokens that is defined by other color sets.

The execution of the runtime model starts from the *Ready Process* place. This place can be customised by the initial marking, which represents all of the lements of the composite web services(CWS) based workflow that is to be executed. The other important place in the initial marking is the place *Service KB*. It contains all the knowledge about each of the web services, such as I/O types, priority, service type (initial, goal, others). The aforementioned are the essential information that is defined also in the SSD. In addition, for the purpose of driving this model, a pre-defined fault rate of each service is embedded as well. The possibility of activating alternative solutions is determined based on uniform distribution. The three transitions are the query services. They are also provided by actual speculative enactment runtime. Place *Service Executed* holds the intermediate web services, which activate the following queries. After finishing the whole execution, there will be no tokens in this place. The remaining two places *Exception In Execution* and *Finished Process* contain tokens that represent failed workflow and successful workflow after the execution. All the failed service instances are kept in *Exception* as a record, as well as to activate the alternative solution query. All successful processes are be recorded in *Finished Process*.

To demonstrate the feasibility of SSD for CWS execution, as well as the intuitive

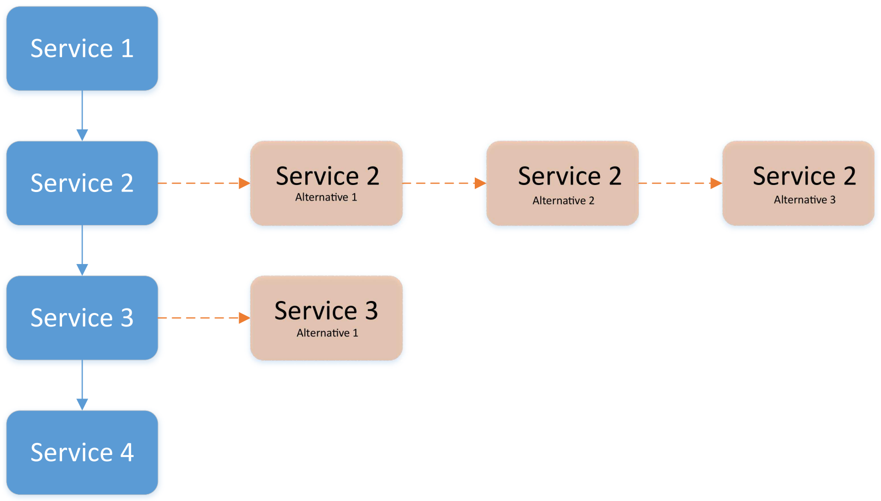


Figure 6-9: An exemplar composite web service for the Petri net model

	No. of completed	No. of total transitions	Extra steps on alternative solution
No substitute	821/1000	4000	N/A
1 substitute for S2 & S3	965/1000	4116	0.81(**116**/144)
2 sub. for S2 & 1 for S3	978/1000	4182	6(**56**/13)
3 sub. for S2 & 1 for S3	984/1000	4225	7.16(**43**/6)

Table 6.2: The Simulation Results of Petri Net Model

understanding on how the SSD supports exception handling, an exemplar CWS is created based on the Petri net model. Figure 6-9 depicts the workflow of the CWS. It is a four step sequential workflow. The second and third web service have substitutes to provide alternative solutions.

This experiment is designed and simulated by using the CPN Tools[Jensen et al., 2007]. The Petri net model is designed as shown in Figure 6-8. There are 1000 CWS instances of the same workflow being enacted in each of the simulation. They are all initiated from place *Ready Process*. In all the scenarios, Service 2, Service 3 and their substitutes all have a success rate of 90%. It means 10% of the executions need to be handled by error handling mechanism. The variables are the number of services that have substitutes (the alternatives) and the number of substitutes for one service. This simulation aims to show the trend that how the declarative description, based on the three query tags (Initial, Next, Alternative), can influence the number of completed CWS from a macroscopic perspective.

Table 6.2 shows the simulation results based on a series simulations under different conditions. It aims to show that this model is capable of increasing the number of completed CWS. It also aims to show with the increasing of the number of completed CWS, there is more overhead on finding and calling the alternative solutions.

The second column is the number of the completed workflows. For example, in the scenario of no substitute, 821/1000 is approximately equal to the product (81%) of the simulated success rates of Service 2 and Service 3. The third column shows the overall transition firing steps in the simulation. When there are alternative solutions, the CWS needs more steps to complete. The difference between the numbers of other rows with the number of first row represents the extra execution steps on web service execution. The fourth column shows the extra average overhead on each of the CWS that being recovered by new alternative solution. For example, in the row of “One substitute for S2/S3”, 0.81 equals to (4116-4000)/(965-821). Obviously, it can be observed that, with more alternative web service as substitute, more CWS are able to finish successfully. On the other hand, more successful enactment means more overhead, especially when there is no speculative process is introduced so that all the primary and alternative solutions are executed in a single-thread manner. This issue is rooted in the time wasted on unsuccessful execution and service invocation timeout. This timeout can be a very long time that varies because of different client tools. The situation could be even worse by considering that some scientific web services may take very long time to finish in real-world scenario. It means to redo them, by either the original service or the alternative one, it may double this long time.

In fourth column, the ratio between extra steps on alternative solutions and the number of extra successful CWS is increasing from 0.81 to 7.16. It means have 1 more successful CWS, the system needs more queries on alternative solutions and more executions of services. As explained in last paragraph, in a single-thread execution model, the more alternative solutions one web service have, the more time will be spent on long time execution and timeout. They are piled up in a sequential manner. In this simulation experiment, it can be observed that the design of having alternative solutions for failures in this declarative description is able to raise the number of successful CWS in the environment that web services have a certain failure rate. However, it also shows that when you apply more alternative solutions, the time resources that spend on the each successful CWS is more. It is believed by introducing the speculative enactment runtime, this trend can be mitigated that all the alternative solutions

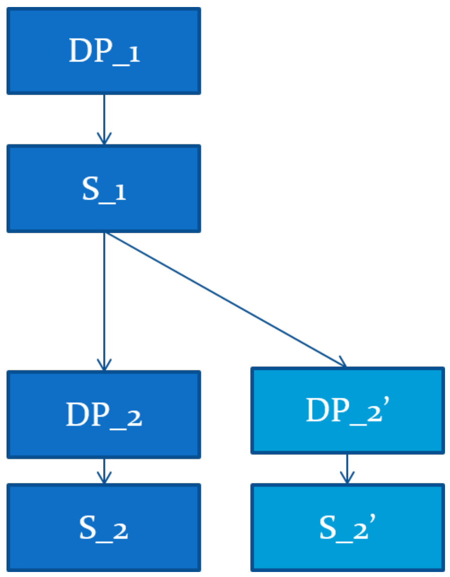


Figure 6-10: The test composite web service

will be executed in a speculative and parallel way to save time.

6.7.2 A Implementation of the Speculative Enactment Runtime

In Section 6.6, the design of the runtime for speculative enactment has been introduced. It is implemented as web services based on a series tools, such as Java language, Prolog, Apache Tomcat. The reasoning process for finding candidates (alternative solutions) for speculative enactment is written in Prolog as shown in Appendix B and connected with web service component in Java by JPL [Singleton et al., 2004]. In the Petri Net evaluation, it is hard to simulate the speculative process, especially the real performance effect taking full account of the overhead of speculative enactment, such as the query for speculative solution as well as the impact of network bandwidth. By evaluating the declarative description in Section 6.7.1, it also exposes the issue that more time will be spent on one successful CWS when more alternative solutions are applied. In this section, an example CWS is created based on the real runtime, which allows the customisation of the amount of data being transferred between services. This allow conclusions to be drawn on the general trend of network bandwidth impact.

Figure 6-10 depicts the CWS for the test. It contains two applications. Each of them has the function to replicate the input data and output it for the following web service. The CWS is built based on Datapool service and web service management

	Non-spec.(ms)	Spec. Normal(ms)/ Difference with Non-spec.	Spec. Alter.(ms)/ Difference with Non-spec.
128KB	1127.2	2090.0/963.8	2233.2/1105
256KB	1169.8	2118.6/948.8	2189.6/1119.8
512KB	1101.4	2105.0/1004	2339.0/1237.6
1MB	1339.2	2196.0/856.8	2327.8/988.6
2MB	1784.0	2416.2/632.2	2526.8/742.8
4MB	1872.0	2720.3/848.3	2850.9/978.9
8MB	2776.8	3499.0/722.2	3785.1/1008.3

Table 6.3: The simulation results of the composite web service in real runtime

systems that were introduced in Chapter 3 and Chapter 4. *DP_1* and *DP_2* represent the Datapool service. *S_1* and *S_2* represent the application based Services. *DP_2'* and *S_2'* are the alternative solution of *DP_2* and *S_2*. They are invoked concurrently by the speculative enactment runtime. The input data size can be customised to observe the impact of network bandwidth on the overall performance. The alternative solutions *DP_2'* and *S_2'* are deployed in different machine. It is in the same intranet of the main server. In this experiment, *DP_2* and *S_2* are disabled to simulate the situation that an alternative solution has to be activated. The environments are listed below:

1. Network: VLAN over gigbit (copper) ethernet;
2. Main server: Linux 2.6.32-31-server: Processor 2.27 GHz Intel Xeon E5520 (*16);
3. Alternative server: Mac OS X Version 10.6.7: Processor 3.2 GHz Intel Core i3, Memory 4GB;
4. VMs (the services are deployed in two VMs on each server): each VM is allocated with 2 CPUs and 2GB Memory; Operating System: Linux 2.6.32-28-generic;
5. Client: the client is co-located with the VM on the Mac machine. This VM also holds the alternative service.
6. Service container: Apache Tomcat 7.0.28

Table 6.3 shows the results of a series execution of the CWS shown in Figure 6-10. They are executed in three modes. *Non-spec.* means speculative enactment runtime

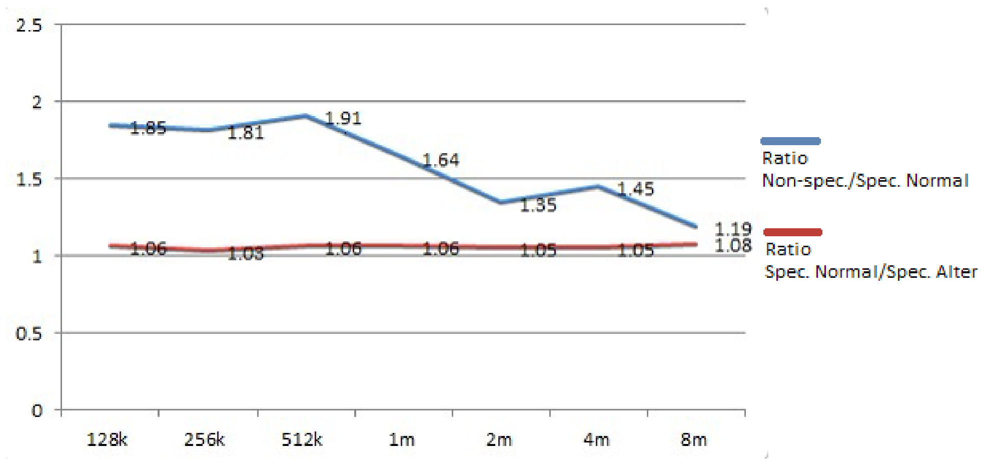


Figure 6-11: The ratio between the performance of execution modes

is not used in the execution. The services are all executed in the normal way. *Spec. Normal* means speculative enactment runtime is applied. The alternative solution is executed by the runtime. However, it is not activated to generate the output for the CWS. *Spec. Alter.* means that the alternative solution is executed concurrently, and the output from it is adopted. The relatively simple CWS allows the observation on the trend of performance when various size of data is applied.

Based on the data from Table 6.3, Figure 6-11 is drawn to show the trend of performance variance. The blue line shows the ratio between *Non-spec.* and *Spec. Normal*. It can be observed that there is still difference between *Non-spec.* and *Spec. Normal*. It can be understood as the overhead of the speculative enactment runtime, which includes all the process for query, reasoning, etc. Whereas, the ratio reduces when data size grows. In this situation, because the overhead is relatively static, the more time spent on other activities in this CWS, the lower the ratio for the overhead of the speculative enactment runtime. The red line, which shows the ratio between *Spec. Normal* and *Spec. Alter.*, provides another evidence on the efficiency of the runtime. Across the range of data transfer sizes, the ratio hardly alters, as would be expected. In the situation that primary solution disabled, it only takes slightly longer time to switch to alternative solution.

Based on this experiment, it can be observed that, by applying the speculative runtime, a relatively static overhead is recorded which is not varied much based on the changes of the time a web service costs. It is reflected by that the coefficient of variation(CV) of the difference between *Spec. Normal* and *Non-spec.* is 15.87%. On the other hand, the CV of the Non-spec. is 38.10%, which is larger. At the same time, the correlation value between *Difference with Non-spec.(ms)* in fourth column and *Non-spec.* is -0.4588. There is no positive correlation and it is only weakly correlated. So the overhead is not caused by the extra alternative execution of the service. On the contrast, this positive correlation can be observed in Table 6.2, that the model runs on single thread and no speculative mechanism applied. When the number of substitute(s) increases from 1 to 3, the extra steps spent on new alternative solutions increases from 0.81 to 7.16. In the experiment, each step can be deemed as a unit time that spent on one service when all the services are assumed to cost the same time.

6.7.3 The Simulation for Time/Compute Resource Decisions

This simulation aims to evaluate the effectiveness of Equation 6.2 for time/compute resource decisions. An speculative enactment runtime model is built based on Colored Petri Net(CPN) in CPN Tools[Jensen et al., 2007]. The the simulation largely reused the structure of the model shown in Figure 6-8. The differences are mainly on:

- The colorset for service is changed. Now it includes the extra information about the price and estimated execution time of each service.
- On more function IfAlternative() is added for transition GetAlternativeService, which is shown as List 6.9. This function is written based on Equation 6.2, as Line 8 in List 6.9, to decide if an alternative should be executed speculatively based on the equation.
- The initial services are changed to fit for this workflow simulation. More details about this workflow is introduced in following text.

Listing 6.9: IfAlternative Function

```
1 {
2   fun IfAlternative(vs:Service)=
3     let
4         val price_ins=(#price vs);
5         val et_ins=(#et vs);
6         val rr_ins=(#rr vs);
7     in
```

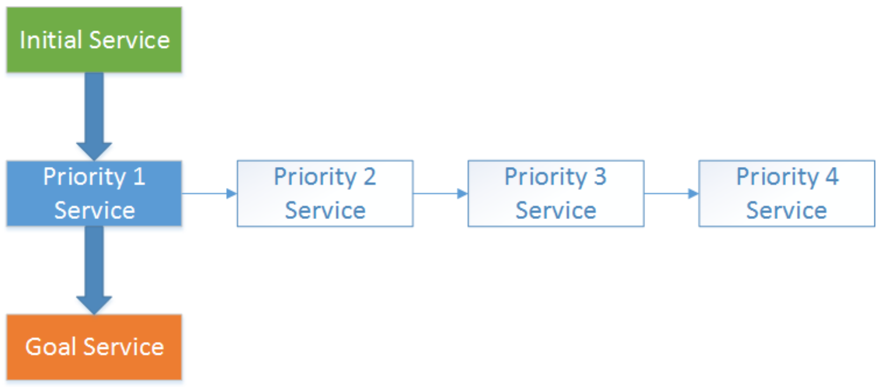


Figure 6-12: Equation simulation workflow

```
        if Cons_mt-Cons_s*price_ins/(rr_ins*et_ins)>0.0
            then true
        else false
    end
}
```

Workflow shown in Figure 6-12 is used for the test. It includes three services. The initial service and goal service will be executed with error rate 0. Their executions will not be influenced by the speculative enactment runtime and Equation 6.2. In order to show the direct influence of the equation to the execution of this workflow, one service (priority 1 service in Figure 6-12) in the workflow will be speculatively enacted. It has three alternative services in sequence with priority from 2 to 4. All the services have the same error rate 10%. In this workflow all the services have the same price, 10 price units. They also have the same estimated execution time(EET), 10 time units. In this simulation, the coefficient S, which is the Cons_s in List 6.9, is set as 1. The only variable is M_t. It is also the only designated number that can be adjusted by workflow runtime users in the design.

With simple calculation, it can be realised that, by setting M_t in certain range, different level of speculation will be achieved. By setting in the range (1,10], only priority 1 service will be executed. By setting in the range (10,100], priority 1 and 2 services will be executed and speculated. Range (100,1000] can have priority 1 to 3 services being executed and speculated. Range (1000,∞) can have priority 1 to 4 services, all of them, being executed in a speculative manner. Thus, to monitor the influence of the equation, four scenarios are set with M_t set to 2, 11, 101 and 1001.

M_t	Round 1(Successed/Failed)	Round 2	Round 3	Money
2	4506/494	4508/492	4515/485	50000
11	4945/55	4944/56	4951/49	100000
101	4993/7	4995/5	4992/8	150000
1001	5000/0	4998/2	4999/1	200000

Table 6.4: The three rounds results of simulation

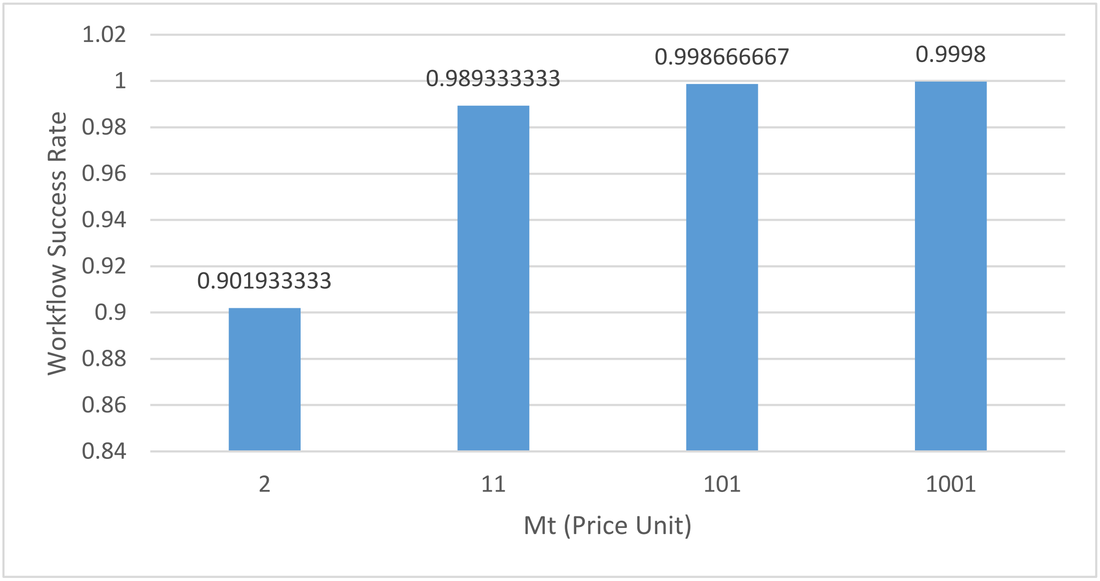


Figure 6-13: Results of simulation

Workflow will be executed in each of the scenarios for 5000 times with 3 rounds, 15000 times in total.

From Table 6.4, it can be seen that for all the scenarios, they have almost the consistent success rate in all the three rounds. The chart in Figure 6-13 further shows the overall success rate by accumulating the three rounds. The general trend is when the order of magnitude of M_t grows, more nines the success rate gets. In Table 6.4, the fourth column shows the costs on the second step execution in the workflow. It grows linearly from 50000 to 200000. Surely, by setting M_t at 2, the user can save 150000 in speculative enactment by comparing with setting it to 1001. But in the circumstance that every fail can incur great loss and every lost time unit may worth more than 1000 price unit, the extra 150000 will worth it.

In a real-world scenario, users can set their psychological prices for M_t. The distribution of service prices and EET will be more complex for the whole workflow. The basic theory about how the equation can be effective and works is already shown in this simulation. Besides, to make the equation more effective, all the variables can even be set in a dynamic way. This idea will lead to more research. This is further discussed in Section 8.3.

6.8 Summary

From the evaluation on the Petri net model and the runtime implementation, two points can be related as summary:

1. Based on the simulation on the formal model, the design of enactment model based on the declarative description is able to increase the number of completed CWS, in the scenario that there is failure rates applied on each web service.

2. Furthermore, based on the results from the experiment on the real runtime, the speculative enactment model shows that the overhead spent on the process of error handling is not correlated with the the execution time of alternative service. The overhead is relatively static when the execution time of alternative service increases.

This chapter gave an overview on web service composition, related description languages and especially the declarative description. SSD is proposed to describe CWS declaratively that are based on and implementable with the ROA based Datapool services and the application services introduced in Chapter 3 and 4. For the purpose of enabling SSD to be implementable in CWS enactment and real world scenario, a detailed design on the declarative enactment runtime is proposed as well. Together with the evaluation based on both a formal Petri-net model and real runtime, this chapter shows the capacity of the ROA based runtime for: (i) REST web services failure handling, which is supported by late-binding feature of SSD and the runtime; (ii) improved performance in the scenario of web service failure, which is supported by the speculative workflow enactment and SSD.

Chapter 7

Case Studies

This chapter presents case studies on some applications in real scenarios. There are three types of cases: (i) a workflow for optimization of the wing internal stiffness distribution. This workflow is based on *in-house* code written by non-computer scientists. The workflow is composed in the Taverna. (ii) the integration with local program based on MDO frameworks. This workflow is built based on existing MDO frameworks, which do not have the ability to expose themselves as web services and composite web service. (iii) a image processing workflow that built based on existing desktop programs.

7.1 Optimization of the Wing Internal Stiffness Distribution

7.1.1 Scenario

The optimization process is formed by three main steps: aerogynamics analysis, structural load transfer, structural optimization. In these three steps, there are four programs which are *DoubletLattice, LoadTransfer, FiniteElement and LevelSetOptimizer*. In our services composition case we integrate *FiniteElement* and *LevelSetOptimizer* as one program. The tangible functions of each program are listed below:

DoubletLattice

The DoubletLattice program models the aircraft wing as an infinitely thin layer to compute aerodynamic lift and induced drag. The program reads in the input file, aero input.dat that consists of the geometrical definition of the wing, such as span and chord, as well as the flight conditions. By simulating the flow vortices around the wing, the aerodynamic load distribution is extracted to an output file, aero load.dat. This program is written in FORTRAN 77 and FORTRAN 95.

LoadTransfer

Fluid-structural interaction is an important study when integrating the aerodynamic and the structural discipline. This is because of the two different meshes between aerodynamic (DoubletLattice) and structural (FiniteElement) models. The input script, structural input.dat details the element sizes of the wing which is also going to be modelled in the FiniteElement program. The LoadTransfer program transfers the calculated aerodynamic loads to the finite element mesh via surface spline and computes the structural loads (structural load.dat). The LoadTransfer program is written in FORTRAN 95.

FiniteElement

The FiniteElement program reads in the structural loads input and calculates the stiffness of the wing structure. The deflection of the wing is computed based on a linear static equation. Therefore, the strain energy, which is the objective function to be minimized, can be calculated. This program is written in C-program.

LevelSetOptimizer

A level-set method is implemented in C-program to optimize the wing structure. The objective is to minimize the strain energy subject to a volume constraint. During the optimization, the structural boundary is moving inward, hence, the volume is reduced. These involves a number of iterations and in each iteration, the wing deflection and the strain energy are re-calculated as the overall structure stiffness has changed. The solution is converged when the convergence criteria are met, i.e. the targeted volume is achieved and the objective values of the last 5 iterations are less than 0.1

7.1.2 Solution

For the purpose of demonstrating a MDO process in a web services composition, the Taverna workbench is utilized for the jobs of services composition, execution and monitoring with the support of the services located in Datapool and SaaS based framework

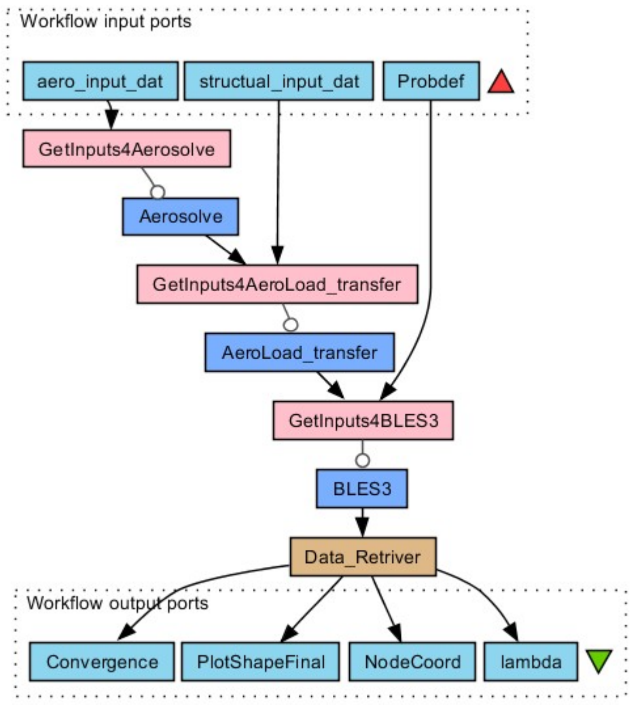


Figure 7-1: The MDO process built in Taverna based on REST web services

proposed in Chapter 3 and 4. We demonstrate all the functional modules with an example created in Taverna. Figure 7-1 is the screenshot of the services compostion design example, which intends to optimize the topology of the internal layout of an aircraft wing structure. All the dark blue boxes *(Aerosolve, AeroLoad_transfer, BLES3)* are RESTful web services built by the web services management system upon three command line programs, *DoubletLattice, LoadTransfer and FiniteElement and LevelSetOptimizer as BLES3*, which are written in either Fortran or C languages. All the pink boxes *(PutInputs4Aerosolve, PutInputs4AeroLoa transfer, PutInputsBLES3)* are the RESTful web services for the Datapool, whose functions are uploading data as web resources for application services. The input ports built in Taverna based on RESTful web services are located at the top of Figure 7-1, and output ports are located at the bottom. One local service Data Extractor is utilised to retrieve the data based on the URIs collection return by the last application service.

A simple rectangular wing example is provided and tested by the non-specialist to demonstrate the feasibility of this ROA-based framework. In the aero input.dat, a wing structure of 20 strips and 5 chord boxes with a 0 dihedral angle and 4 incidence

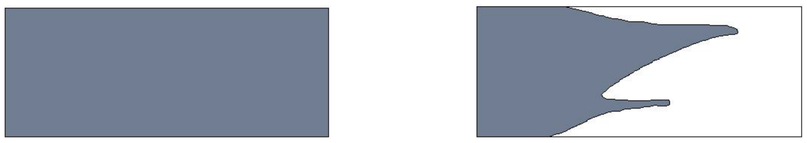

(a) The initial wing topology (b) The final wing topology

Figure 7-2: The wing topology

is defined. The wings root chord is 8 meters and span is 20 meters. Meanwhile, the ratio of root chord and tip chord is 1. The flight condition is set as 0.7 MACH and 35000 feet. After the execution of aerodynamics analysis and load transfer steps, the structural loads that are applied on each finite element nodes are interpolated from aerodynamic loads using spline method. By using the structural loads transferred from aerodynamics loads as input, final topology solution is generated by the structural optimization step. Figure 7-2 depicts the initial topology and final optimum solution. The solution is generated based on the numerical result from PlotShapeFinal output port.

7.2 The Integration with local program based MDO frameworks

7.2.1 Scenario

The framework also has the ability to integrate with existing MDO frameworks. As reviewed in Chapter 2, there are some frameworks that already include a web service function or a server/client architecture. However, some frameworks can only be executed as local programs. A learning curve is needed for the non-specialists to fill the gap between those local frameworks and online service based functions. In this case, the ability to deploy legacy MDO workflows based on existing MDO frameworks like OpenMDAO[The OpenMDAO development team , 2010] and Dakota[Sandia National Laboratories, 2010] is presented.

With the support of the OpenMDAO runtime installed in the server, the deployment process can be achieved as easily as for any other command-line program. On the other hand, Dakota has a different execution approach in that the workflow is defined as a input file, which is then executed by the Dakota runtime. With the Dakota

```
<WS_Definition>
<Service_Name ifPublic="no">sellar_MDF</Service_Name>
<Application_Name IfDakota="no"
  IfExecutableJar="no"
  IfOpenMDAO="yes"
  IfPythonModule="no">
   sellar_MDF_solver.py</Application_Name>
<Inputs>
<Input IsValue="yes"
  NotInArgs="no" name="z1" qualifier="-zi"/>
<Input IsValue="yes"
  NotInArgs="no" name="z2" qualifier="-z2"/>
<Input IsValue="yes"
  NotInArgs="no" name="x1" qualifier="-x1"/>
</Inputs>
<STDOut name="sellar_result"/>
</WS_Definition>
```

Figure 7-3: The description file for openMDAO MDF service

runtime installed on the server, the workflow can be executed as a web service by simply uploading the input file through the Datapool service.

7.2.2 Solution

openMDAO Service

The executable based on the openMDAO framework is a python script, which is created by users. The openMDAO works as a python library to support the functionality of the MDO workflow. In this case a Multidisciplinary Design Feasible (MDF) workflow [The openMDAO development team, 2013] is deployed as a RESTful web service based on the deployment GUI tool and the Datapool service. In this case, the workflow executables have to be deployed to the server as a web service. If other MDO workflows need to be deployed, the corresponding python script should be deployed in advance as web services through the GUI tool as well. The generated web service description is presented in Figure 7-3.

In the description in Figure 7-3, the three parameters of the optimizer are configured as inputs. The output stream from the openMDAO framework is configured

```
<WS_Definition>
<Service_Name ifPublic="yes">dakota_t1</Service_Name>
<Application_Name IfDakota="yes"
  IfExecutableJar="no"
  IfOpenMDAO="no"
  IfPythonModule="no"/>
<Inputs>
<Input IsValue="no" NotInArgs="no"
  name="dakota_rosenbrock_2d_in" qualifier="-i"/>
</Inputs>
<Outputs>
<Output IsValue="no" NotInArgs="no"
  name="out_2d" qualifier="-o"/>
</Outputs>
</WS_Definition>
```

Figure 7-4: The description file for Dakota service

as the output of the web service. They are all stored in the Datapool service and accessible through allocated URIs.

Dakota Service

The Dakota has a different architecture as the openMDAO. It is a software package that reads in an *.in* as a description to the MDOproblem. The user writes the description rather than a script to solve the problem as what the openMDAO directs users to do. Thus, to deploy this framework as a web service, first it is assumed the Dakota software is already installed in the server. The generated description of the RESTful web service is presented in Figure 7-4. This description is generated by the GUI based interface introduced in Section 4.3.3. Through the GUI interface, the command-line entry for this software can be deployed as web service on server side. Then, to solve a specific problem, user just needs to send the problem description *.in* input as he/she does for the desktop version of Dakota software to the web service. In this case, a problem called *rosenbrock* is used as an example, which is extracted from its User Manual [Eldred et al., 2007].

In Figure 7-4, the necessary information to deploy the Dakota script as web

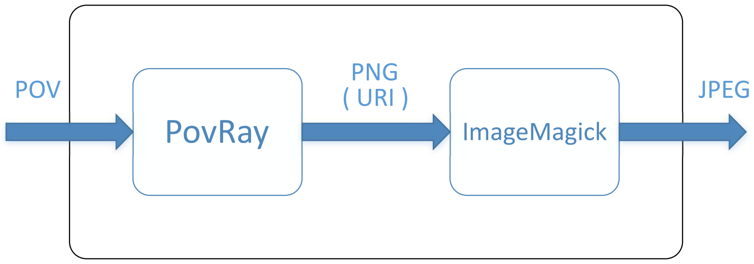


Figure 7-5: The image processing workflow

service is described for the whole framework. In it, the input is the *.in* file. The output of the web service is the output file from the Dakota framework. It can be noticed that, in this case, the service description does not have an application name, because the application that this web service aims to wrap has been deployed at the server side. The problem itself is not an executable script or program. In other words, all the Dakota related service actually have the same application name.

7.3 Image Processing Workflow

7.3.1 Scenario

PovRay[Persistence of Vision Raytracer Pty. Ltd., 2003] is a ray tracing program which generates images from a text-based scene description, which is written in the POV description language. ImageMagick[ImageMagick Studio, 1999] is a software suite to create, edit, compose, or convert images. Conventionally, the user needs to manually operate these two different pieces of software to achieve a workflow, such as generating an image by following an editing step on the image. Surely, there is learning curve for both of the software.

In this case, a workflow, as shown in Figure 7-5 based on web services and the ROA framework will be created to generate image from a POV file and then convert the 3-D image from the *png* format to the *jpeg* format. Both of the command-line executables are written as shell scripts. The PovRay executable also has files reside that serve as libraries for 3-D image generation. Both of the PovRay and ImageMagick have been installed in the target server.

```
<WS_Definition>
<Service_Name ifPublic="yes">Povray_House</Service_Name>
<Application_Name
  IfDakota="no" IfExecutableJar="no"
  IfOpenMDAO="no" IfPythonModule="no">
    povray.sh</Application_Name>
<Inputs>
<Input IsValue="no" NotInArgs="no" name="pov"
  qualifier=""/>
</Inputs>
<Outputs>
<Output IsValue="no" NotInArgs="no" name="png"
  qualifier=""/>
</Outputs>
</WS_Definition>
```

Figure 7-6: The description file for PovRay service

```
<WS_Definition>
<Service_Name ifPublic="yes">Convert</Service_Name>
<Application_Name IfDakota="no" IfExecutableJar="no"
  IfOpenMDAO="no" IfPythonModule="no">
    convert.sh</Application_Name>
<Inputs>
<Input IsValue="no" NotInArgs="no" name="png"
  qualifier=""/>
</Inputs>
<Outputs>
<Output IsValue="no" NotInArgs="no" name="result_jpeg"
  qualifier=""/>
</Outputs>
</WS_Definition>
```

Figure 7-7: The description file for ImageMagick service

7.3.2 Solution

The related executables and libraries for each program are packed as two zip files. They are deployed through the GUI tool as RESTful web services. The generated web service description files are listed in Figure 7-6 and 7-7.

The workflow also contains two Datapool services. The first Datapool service is used for initial input data uploading. The second one is the one to store intermediate data, the *png* file. The Datapool service for ImageMagick receives this *png* file as a URI reference (step 6 in Figure 4-4). They are invoked from the client-side by an executable script written in Python, which supports the invocation of RESTful web services. One syntax example is shown in Figure 4-10 on Page 90.

7.4 Summary

This chapter covers three types of scenarios:

- in-house legacy code based command line programs, which are composed in existing WMS;
- legacy workflows in existing software framework, which is wrapped and exposed as web service;
- legacy software based command line programs, which is composed programmatically by new code.

Therefore, this framework shows its flexibility to integrate with various types of workflow platforms. It also fits with the requirements from different skill levels of users. The components of the composite web service can be a piece of code, software or even the workflow created by existing workflow framework. The output can be workflow in existing WMS, web services and in-house customized code.

Chapter 8

Conclusion

8.1 Contribution

This thesis studied the *feasibility* and *advantages* of applying state-of-the-art computer technology in scientific workflow modeling in a collaborative environment. Based on this study, a *reference implementation* for ROA based workflow modelling was proposed, which covered the topics of data management, service management, data and service virtualisation and service composition. This thesis started with a series of reviews on related researches and technologies related to those four topics. Among them, the data management and web service management formed the foundation of this modelling framework. The data and service virtualisation enabled the reusability of scientific applications in heterogenous computing environments with various machines/operating systems. The study on web service composition carried out an exploration to further release the potentials of this framework, which enabled the description of the composite service declaratively and to execute it by a speculative computing enactment runtime. Practically, the contributions of this thesis were achieved in three stages:

- A light-weight framework for workflow data management and service management that is based on RESTful web service to bridge the gap between desktop scientific application and online scientific workflow;
- A data and service virtualisation framework to further enhance the efficiency on computing resource and increase the reproducibility of the computing environ-

ment for heterogenous scientific applications based on virtualisation/containerization technology;

- A speculative enactment runtime for scientific workflow, which is based on the light-weight framework.

A Light-Weight Framework for Bridge-Building from Desktop to Cloud

This study has presented an indication for the benefits arising from our light-weight framework for the deployment and execution of scientific application in the cloud. With our GUI based deployment mechanism, the technical barriers are lowered for non-specialist usage of web services and cloud resources. The framework **reduces the effort and enhances the efficiency** for users to turn legacy code and programs into web services and hence collaborate with each other. The distributed and asynchronous data staging mechanism helps **reduce end-to-end times** by hiding the costs of data staging between services as well as between client and service, which works as the foundation to enhance the efficiency of data transfers in the ROA based scientific workflow modelling. Practically, this thesis also evaluates the usefulness and usability of the framework through a user study and some experiments, showing how different types of legacy programs and tools can cooperate seamlessly in workflow with the support of our framework.

A PaaS Scientific Workflow Framework based on Data and Services Virtualisation

Chapter 5 proposed a scientific workflow modelling framework based on containerization technology. It also adopted the data staging approach and SaaS platform introduced in Chapter 3 and 4. By adopting latest technologies, this work proposed a new idea about how to overcome the issue of data staging overhead and the compatibility issues brought by heterogenous computing platforms in a scientific workflow. This framework showed its ability to **enhance the efficiency of computing resources utilisation** and to **increase the reproducibility of the computing environment for heterogenous scientific applications**. This chapter also discussed the limitation of such methodology in scientific workflow modelling. This discussion yielded the guidelines about the suitable solution for scientific workflow in different computing environments.

A Speculative Enactment Runtime for Scientific Workflow

In Chapter 6, a speculative enactment runtime was introduced together with the Speculative Service Description(SSD). SSD is the first declarative workflow description language that is designed for a speculative enactment runtime. It further presented the capability of declarative description language in the implementation of scientific workflow, which **enhances the fault-tolerance ability of scientific workflow**. It supports the composition of web services that generates the workflow as autonomous services. Additionally, it utilizes and benefits from the architectural specification of hypermedia distributed systems as described by REST, and from the insights and guidelines of ROA.

On the basis of the SSD and enactment runtime model, this work introduced the prototype implementation of the speculative enactment runtime. The implementation brings in the idea of integrating both objective-oriented and logic programming languages to create the web services as the interfaces for the speculative enactment process. This prototype allows the validation of this speculative enactment runtime in real application scenarios.

As a summary, this work enhances the efficiency of scientific workflow modelling and presents the feasibility of implementing ROA and RESTful web services in scientific workflows with the introducing of new approaches on data management, service management and service composition in aforementioned two stages. Specifically, this can be observed from two perspectives:

1. from the application perspective, as a user of scientific workflows, this work validates the feasibility and demonstrates the advantages of applying a new ROA based workflow framework.

2. from the development perspective, as a computer scientist, this work fills the gap between new web services technology and scientific workflow modelling by developing a more efficient framework based on ROA and RESTful web services;

8.2 Limitations

8.2.1 The Overheads in Web Services based Scientific Workflow

In Section 3.4, Figure 3-5 shows the impact of overheads in network communication. The fluctuation of execution times is caused by the varied network situation. It can be assumed that, in workflows with different execution time and I/O data size, this overhead can impact the total performance. Meanwhile, in Section 6.7, Figure 6-11 also shows the overhead and its trend caused by the introducing of speculative enactment runtime. This section will discuss the overhead in web services based scientific workflow and its limitation.

To further investigate the impact of overhead on the performance, first of all, two experiments are set up based on a workflow with (i) scalable computing time, (ii) and transferred data size , which are composed based on the framework. It is the sellar workflow script sellar_MDF_solver.py in OpenMDAO in this experiment, which is introduced in Chapter 7. The results are discussed in two parts:

Scalable Computing Time

The code in sellar.py has a very short compute time (one line of script), which makes it easy to observe the communication costs. With the addition of a line of code to make it sleep for a specified period to simulate a longer computing time. The comparison is between the local version of the prolonged script and the web service version of the prolonged script. In them, the web service version has two scenarios that one is deployed on the Amazon EC2 machine, one is deployed on the intranet of UoB.

In this experiment, the workflow is executed for 139 times to get the mean of total execution time. In Figure 8-1, the execution time for three conditions are presented: local workflow, web service workflow (Intranet environment), web service workflow (Internet environment). The x axis represents the sleep time in the script, which can be deemed as the computing time of the original local script. From Figure 8-1, it is not surprising to see generally the execution time of Internet workflow is the slowest and the local workflow is the fastest. We can also observe that the variation in Internet traffic has a significant impact on workflow execution time.

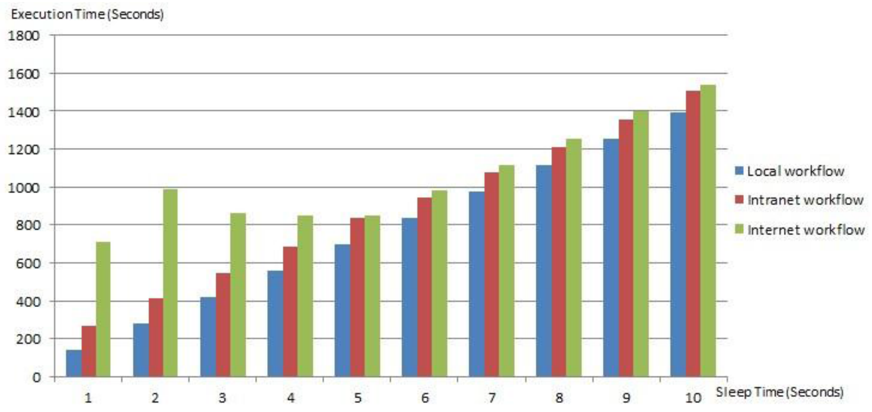


Figure 8-1: Workflow execution time with different sleep time in three situations (Local workflow, Intranet workflow and Internet workflow)

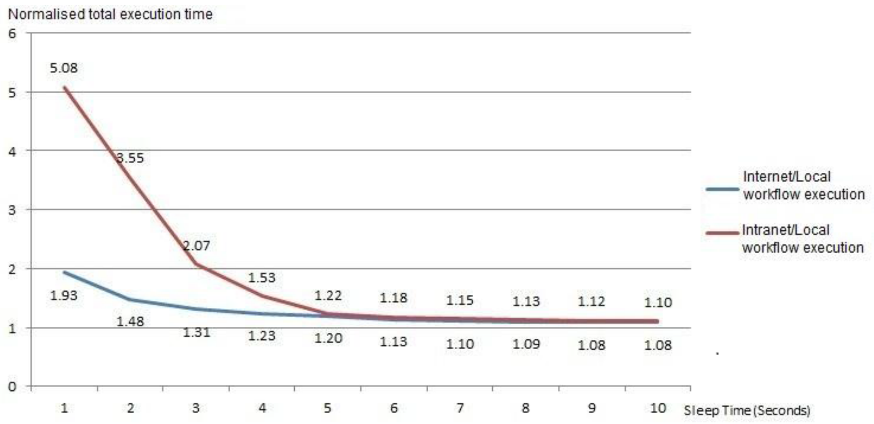


Figure 8-2: Web service based workflow execution time normalised by the local execution time

Based on the results from Figure 8-1, we derive Figure 8-2 which plots the ratio between Internet and Local workflow as well as the ratio between Intranet and Local workflow. The latter has a clear downward trend as the computational time grows: from 1.93 at 1s to 1.20 at 5s, and to 1.08 at 10s. In the Internet workflow, we can observe a nonmonotonic trend at the beginning in Figure 8-1, which is simply due to traffic variations in the Internet environment and configuration at server-side. In this experiment, the server is located on the east coast of America. The server is a AWS free-usage tier one, which is allocated with limited and low priority computing and communication resources. However, from Figure

8-1, we can still see the general trend that the ratio between Internet and Local workflow drops from 5.08 at 0.5s to 1.22 at 5s. After 10s, we observe that the differences between the three conditions have stabilized. At 10s, the ratio between Internet workflow and local one reduces to 1.10. It can be observed the Internet/Local ratio is getting closer or even similar to the Intranet/Local ratio when computing time grows. This indicates that, in the circumstance of that there is only control flow (or the dataflow's size is stable), the influence of network situation on total execution time is getting smaller when computing time of web service grows. After certain level, such as the 5s computing time in this case, the influence of network situation on overhead is almost negligible.

Scalable Transferred Data Size

A further experiment will show that how the execution time scales when large-scale data communication is involved in the workflow. It also aims to observe sensitivity to the increasing of data size.

As a basis, the framework has a stream style communication design for data handling that no matter how large the data is, it is never fully loaded into the memory. The data is divided into a stream of small sections. The memory only loads one section from the hard drive every time and handles the data in a stream, with the objective of reducing the overall memory footprint. Even in some extreme situation when the machine only has very limited memory, it can still deal with large amounts of data. This is also one of our test that will be shown. The user of this framework also does not need to manually configure the memory heap size for the framework (it is based on Java Virtual Machine), the default size is enough.

In this experiment, different sizes of data ranging from 128K to 256M are submitted to web service deployed through the framework. The web service on server side will send back a success response when file is successfully received. The whole execution finishes with a further request to a web service to fetch an HTTP response with empty body. The web service simply simulates the process to consume the data uploaded previously. Two network environments are tested here to provide a comparison. In the first scenario, the service is deployed in an

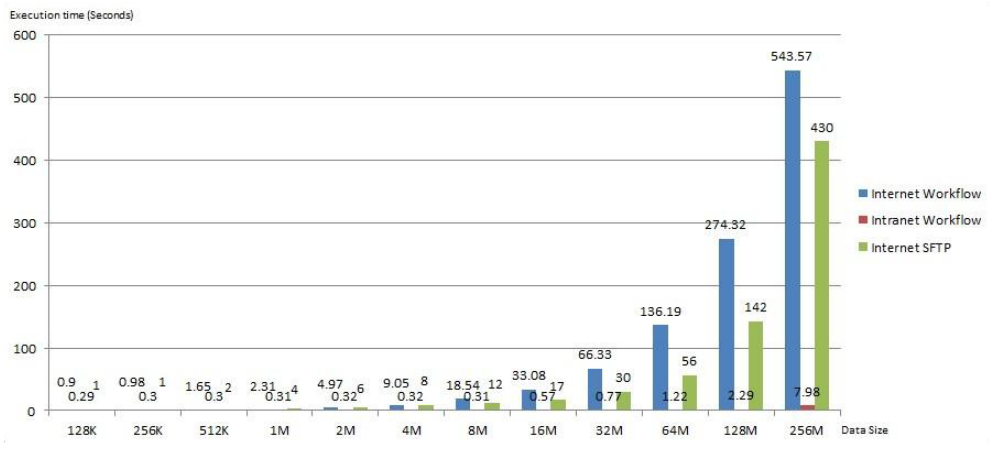


Figure 8-3: Experiment results of data communication

Amazon EC2 machine and the client side is deployed within the University of Bath intranet. In the second, both server and client are within the intranet. Both servers and the client machine have a single core CPU.

To establish a baseline cost for data transfer alone, we ran an experiment using the Secure File Transfer Protocol (SFTP), in order to get a better picture of the influence of the bandwidth and network environment on transmission costs. This data is plotted on Figure 8-3 as well. From Figure 8-3, it can be observed that data transmission time rises in line with data size as expected. The useful observation however, once the transfer size outweighs the overhead costs, is that we can observe the overhead for the WS data transfer over SFTP by comparing the blue (WS) and green (SFTP) bars. As usual, small volumes are noisy and uninformative, but above 4Mb, there is a clear differential between WS and SFTP, with the former taking longer. The study here is not sufficiently detailed to be anything more than indicative, but it suggests confirmation of the intuition that data transfer over HTTP incurs a small but notable overhead compared to SFTP, which is tuned for data transfer. Without more detailed investigation it is hard to say whether the factor is additive or multiplicative, but the ratio is falling between 16Mb and 256Mb. The second useful observation, which helps to contextualize these experiments, is that the approximate bandwidth between UoB and the EC2 machine is in the range 0.8-1.0Mb/s. A higher performance network should provide higher bandwidth and hence contribute to a lower performance crossover point in terms of the minimum data size necessary before it is worth using a WS.

The limitation can be concluded as, (i) the performance of data staging is still significantly impacted by the network; (ii) the network overhead is a stable factor that, with the growth of computing time and data size, the impact turns smaller; (iii) in better network situations, the network overhead takes a smaller part of overall overhead.

On the other hand, the overhead on data staging is not only limited by network latency. The hardware limitation is yet another influential factor. The mechanical hard disk can reduce the I/O speed signicantly. A possible direction to explore in the future is the adoption of RAM disk. On implementation level, a series solutions can be adopted, such as *ramdiskadm* [Oracle, 2011], *RapidDisk* [Koutoupis, 2010], etc.

8.2.2 The Synchronized Connection based on HTTP

In this framework, the HTTP connections in the workflow are synchronized. That means at network level, for each HTTP connection, it will not be disconnected from server until the end of the computing process. For example, if there is an application that needs a long time of execution, the connection will be maintained as long as the execution.

At client side, the Taverna workflow has a multi-process mechanism in its execution runtime. Therefore, the synchronized connection will not block the execution of other available web services invocation. In the Python based and Java based client side, multi-thread code can be involved as well to avoid the impact of synchronization. In both client types, timeout length can be customized. It allows the client to wait for the connection, rather than disconnect it automatically. In a nut shell, in this research work and prototype, the asynchronized connection is not necessary.

In future implementation, the disadvantages of a synchronized connection potentially can be reflected from following aspects: (i) the client with uncustomisable timeout length, such as some of the browsers, will have a problem if the execution time is too long; (ii) the client machine may experience shutdown or restart in this process, or the user wants to retrieve the results from another machine; (iii) if there are multiple instances of a connection, to release the resources as soon as possible will also be a positive factor to the system performance.

The asynchronised connections can be supported by pull or push technology. By implementing pull technology, such as web feeds, the client can maintain an aggregator to subscribe each of the remote executions. Besides, the latest push technology can also provide an option as solution. Such as in HTML5, WebSocket [Fette and Melnikov, 2011] protocol provides the full-duplex communications channels over a single TCP connection. Because it uses TCP port number 80, it can be used in the environments that firewalls block non-web Internet connections.

8.3 Future Works

This framework, as an implementation, shows its ability and potential in the application of scientific workflow. As an outcome of this research, it still can have its potentials to be further released in future implementation beyond a prototype. The future works will be carried out from two aspects as well: web services management and speculative enactment runtime.

Web Service Management

The future works on web service management will mainly focus on the following aspects:

- In the future, the framework can be further developed to be deployed on more platforms, such as Windows and Mac OS.
- The user interface can be migrated from a Client/Server architecture to Browser/Server architecture.

Speculative Enactment runtime

Currently, the speculative enactment runtime is used mainly to deal with fault-tolerance or error-handling issues. It has the potential to work as a mechanism to provide alternative solutions in more scenarios. For example, when there are multiple options of the inputs for a computing process, it will be helpful to provide a single service that can take multiple input options and generate one optimal result. This can be implemented and tested for some cases in specific research areas. Currently, there is no such a study case in hand in order to build the test workflow and collect the test data. This research and implementation can only be meaningful when there is such a demand to work out the optimal

output from multiple inputs. The possible amount of the inputs and the affordable computing resources can potentially influence the efficiency of the system. These will lead to the needs for more case studies and researches.
At the implementation level, the Prolog based speculative enactment runtime can be refactored and written in the OO languages, such as C++, Java because the current solution does not have an ideal link between OO languages and Prolog. For example, JPL does not support multi-process requests. Furthermore, with an OO languages based system, it will reduce the costs to deploy the system with less knowledge preparation and software/library support. Besides, the SSD can be designed and written based on more user-friendly grammar or even align with other description language standards.

Integration of Virtualized Scientific Workflow and Speculative Enactment Runtime

In this thesis, both the virtualized scientific workflow platform and the speculative enactment runtime are independently developed based on the works introduced in Chapter 3 and Chapter 4. They are both good examples to show the extendibility of the ROA based scientific workflow platform. Furthermore, these two independently developed systems can be integrated together to gain more potential.
For example, now the speculative enactment runtime is more like a tool that only serve for client-side users. The whole speculative process can be explicitly programmed and monitored from user's point of view. The whole process is not transparent to users. They are aware of which services are speculatively executed and what alternative services are invoked. By integrating the virtualized scientific workflow and speculative enactment runtime, this process can be carried out entirely at server-side such that the users do not to be aware that there is even a speculative process. The speculative enactment process will be transparent to users. At server-side, for one service, there can be multiple algorithms, multiple instances or multiple inputs that are executed in parallel in a speculative manner to serve for one purpose. They are all controlled by the server. From the user's point of view, the enactment of this type of *server-side speculative service* does not need to be different from a normal service. Virtual machines can be dynamically started or stopped for each service execution job based on the number of service requests and idle resources. Algorithms can be developed for

this dynamic allocation.

Future Works on Speculative Enactment Runtime

In the discussion about time/compute resource decisions in Section 6.6.3, an equation is proposed about how to make the decision. Equation 8.1 is also proposed to further reveal the research potential based on the speculative enactment in scientific workflow.

$$D = M_t - S * M_s/(P_s * T_s) \tag{8.1}$$

It is discussed that, users or system administrators can adjust their sensitivity to financial cost case by case by setting the M_t. This setting can be a static number. It can also be dynamically reflected and configured based on some factors at runtime. Finally, it will lead to a system that not only automatically and efficiently, but also economically can achieve the workflow tasks in terms of time/compute resource costs. The factor can be the idle resources that left in the HPC system. The coefficient can also be set dynamically based on the change of power price or the daily generated power from the local solar panels. The factors can be more. The data about those factors can be collected from a sensor network. This combined consideration of efficiency and economy will lead to a wide range of future works.

Appendix A

The Description of the Cancer Modelling Workflow

A.1 Introduction

The cancer modeling workflow is composed of three models that concern breast cancer. They are the deliverables from the research in Kolokotroni et al. [2008a,b, 2011]. The workflow is built based on Taverna workbench as depicted in Figure A-1. The workflow contains three sub-models. Their input data are read in through a XML parser. The parser is also used to connect models. The parser collects data from last step in workflow and combine it with the raw input to generate the input file for next step in the workflow. The functions of each model is briefly introduced in following sections.

A.2 Epirubicin Pharmacokinetcs

The model simulates the pharmacokinetics of Epirubicin and calculates the Area Under Curve of the plot plasma concentration of the drug against time after administration. More specifically, the pharmacokinetics of Epirubicin is modeled by a three compartment open model with elimination from the central compartment

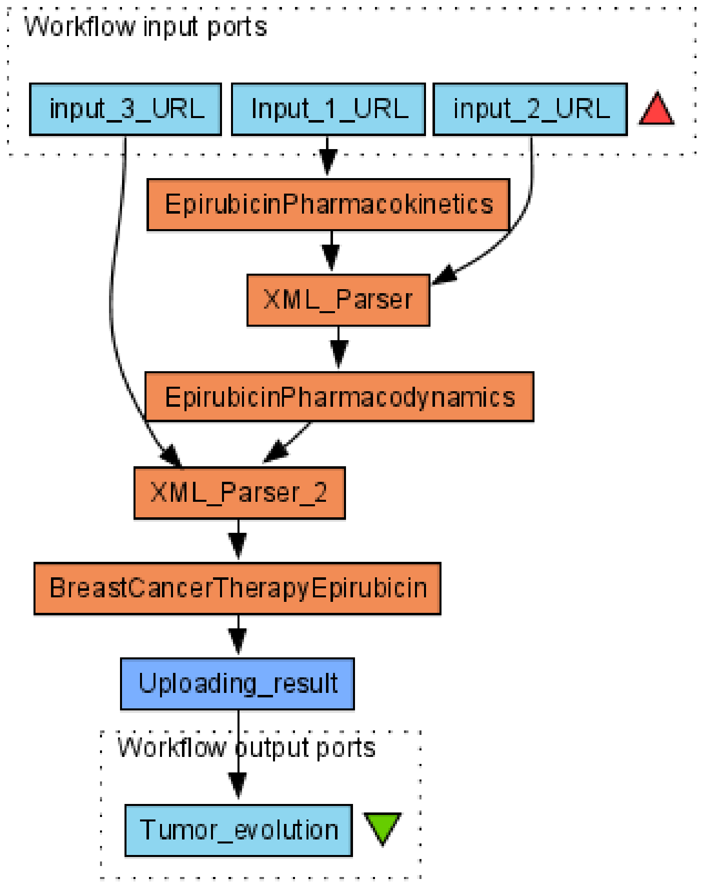


Figure A-1: Taverna workflow of cancer model

A.3 Epirubicin Pharmacodynamics

The model calculates the cell kill ratio (CKR) of Epirubicin assuming an exponential dependence of CKR from area under curve under the plot of plasma concentration of drug (not logarithm of the concentration) against time after drug administration

A.4 Breast Cancer Therapy: Epirubicin

A generic four-dimensional simulation model of breast cancer response to chemotherapy with single agent Epirubicin. The model is based on the consideration of a discrete time and space stochastic cellular automata.

Appendix B

The Core Prolog Code of the Enactment Engine

```
% Definition on how to find out the descendent of service.
% The usage of PX and P here is to define the priority of the child service or descendent service,
% When a descendent priority is P, the priority of all the children on this branch that lead to this
    descendent should be higher. (1 is highest)
% The meaning is that when you can not find a priority 1 branch (all children with priority 1) to the
    descendent, try less priority route which does not need all the children on this route should be
    priority 1.
% X,Y : Service identification (URI). XM, YM: Service Method: {PUT, POST, GET, DELETE}

descendent_of(X,MX,Y,MY,P):−child_of(X,MX,Y,MY,PX),PX=<P.
descendent_of(X,MX,Y,MY,P):−child_of(Z,MZ,Y,MY,PX),PX=<P,descendent_of(X,MX,Z,MZ,P).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% Definition on how to match the initial service based on appropriate inputs.
% Vars: input names; URI_D/O: The URI of the destination service and the original service. MD/O:
    Method of the service.

initial_steps(Vars,URI_D,MD,URI_O,MO):−details_of(URI_O,MO,_,_,_,List,_,_,_,_),inVars(List,Vars),
    descendent_of(URI_D,MD,URI_O,MO,1).
initial_steps(Vars,URI_D,MD,URI_O,MO,X):−details_of(URI_O,MO,_,_,_,List,_,_,_,_),inVars(List,Vars
    ),descendent_of(URI_D,MD,URI_O,MO,X).

```

```
% This function can match the initial service not only based on inputs but outputs as well. This function is not implemented at Java level.
initial_steps_IO(IVars,OVars,URI_O,MO,URI_D,MD):-details_of(URI_O,MO,_,_,_,ListI,_,_,_,_),details_of(URI_D,MD,_,_,_,_,_,ListO,_,_),inVars(ListI,IVars),(OVars=ListO),descendent_of(URI_D,MD,URI_O,MO,1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Utilities: definition on how to match the input name to the input lists contained by some service.

inVars([List|[]],Vars):-inlist(List,Vars).
inVars([Item|Tail],Vars):-inlist(Item,Vars),inVars(Tail,Vars).

inlist(Item,[Item|Rest]).
inlist(Item,[DisregardHead|Tail]):-inlist(Item,Tail).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Definition on how to match the appropriate next step.
% URI_N: the URI of next service; MN: the method of the next service.

% Find the next step based on not only the present service, but also the destination service. This can be
% be used in the circumstances that service composite description has more than one goals.
next_step(URI_O,MO,URI_D,MD,URI_N,MN):-child_of(URI_N,MN,URI_O,MO,1),descendent_of(URI_D,MD,URI_N,MN,1).
next_step(URI_O,MO,URI_D,MD,URI_N,MN):-child_of(URI_N,MN,URI_O,MO,1),URI_D==URI_N,MD==MN.

% Find the next step based on the present service when there is only one goal in the service composite description.
next_step(URI_O,MO,URI_N,MN):-child_of(URI_N,MN,URI_O,MO,1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Find the alternative next service. The priority is added by one.
% URI_E: The failed service (with Error).

```

```
add_1(X,Y):-Y is X+1.
% Find the alternative service when there are multiple goals in the service composite description.
next_alternative(URI_O,MO,URI_D,MD,URI_E,ME,URI_N,MN):-child_of(URI_E,ME,URI_O,MO,X
    ),add_1(X,Y),child_of(URI_N,MN,URI_O,MO,Y),descendent_of(URI_D,MD,URI_N,MN,Y).
% Find the alternative service when there is only one goal in the description.
next_alternative(URI_O,MO,URI_E,ME,URI_N,MN):-child_of(URI_E,ME,URI_O,MO,X),add_1(X,Y
    ),child_of(URI_N,MN,URI_O,MO,Y).
```

Appendix C

The Template for Docker Configuration File

```
{
        ”Hostname”: ””,
        ”Domainname”: ””,
        ”User”: ””,
        ”AttachStdin”: true,
        ”AttachStdout”: true,
        ”AttachStderr”: true,
        ”PortSpecs”: null,
        ”ExposedPorts”: {
          ”8000/tcp”: {}
        },
        ”Tty”: true,
        ”OpenStdin”: true,
        ”StdinOnce”: true,
        ”Env”: null,
        ”Cmd”: [
          ”/home/virtual_configurable/virtual_configurable/env/bin/python”,
          ”/home/virtual_configurable/virtual_configurable/virtual_configurable/manage.py”,
          ”runserver”,
          ”0.0.0.0:8000”
        ],
        ”Image”: ”me1kd/virtual_configurable”,
        ”Volumes”: {
                    ”/tmp”: {}
          },
        "WorkingDir": "",
        "Entrypoint": null,
        "NetworkDisabled": false,
        "MacAddress": "",
        "OnBuild": null,
        "Labels": {},
         "HostConfig": {
            "Binds": ["/tmp:/tmp"],
            "Links": [],
            "LxcConf": {},
            "Memory": 0,
            "MemorySwap": 0,
            "CpuShares": 0,
            "CpuPeriod": 0,
            "CpusetCpus": "",
            "CpusetMems": "",
            "BlkioWeight": 300,
            "OomKillDisable": false,
            "PortBindings": {
                "8000/tcp": [
                    {
                        "HostIp": "127.0.0.1",
                        "HostPort": "<port>"
                    }
                ]
            },
            "PublishAllPorts": false,
            "Privileged": false,
            "ReadonlyRootfs": false,
            "Dns": null,
            "DnsSearch": null,
            "ExtraHosts": null,
            "VolumesFrom": [],
            "CapAdd": null,
            "CapDrop": null,
            "RestartPolicy": { "Name": "", "MaximumRetryCount": 0 },
            "NetworkMode": "bridge",
            "Devices": [],
            "Ulimits": null,
            "LogConfig": { "Type": "json-file", "Config": {} },
            "SecurityOpt": [],
            "CgroupParent": ""
        }
}
```

Bibliography

Linux-VServer. http://http://linux-vserver.org/. URL http://linux-vserver.org/. [Online; accessed 26-Sept-2015]. 99

LXC. https://linuxcontainers.org/. URL https://linuxcontainers.org/. [Online; accessed 26-Sept-2015]. 99

openvz. http://openvz.org/. URL http://openvz.org/. [Online; accessed 26-Sept-2015]. 99

ActiveState Software Inc. stackato. http://www.activestate.com/stackato, 2014. URL http://www.activestate.com/stackato. [Accessed: 30/12/2013]. 80

Michael Adams and Arthur ter Hofstede. Yawl user manual. *User manual, The YAWL Foundation*, 2009. 55

Rama Akkiraju, Joel Farrell, John A Miller, Meenakshi Nagarajan, Amit Sheth, and Kunal Verma. Web service semantics-wsdl-s. 2005. 51

Eyhab Al-Masri and Qusay H Mahmoud. Discovering the best web service. In *Proceedings of the 16th international conference on World Wide Web*, pages 1257–1258. ACM, 2007a. 137

Eyhab Al-Masri and Qusay H Mahmoud. Qos-based discovery and ranking of web services. In *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, pages 529–534. IEEE, 2007b. 137

Rosa Alarcon, Erik Wilde, and Jesus Bellido. Hypermedia-driven restful service composition. In *Service-Oriented Computing*, pages 111–120. Springer, 2011. 124

G. Alonso, B. Reinwald, and C. Mohan. Distributed data management in workflow environments. In *Research Issues in Data Engineering, 1997. Proceedings. Seventh International Workshop on*, pages 82 –90, apr 1997. doi: 10.1109/RIDE.1997.583708. 43

Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludaescher, and Steve Mock. Kepler: Towards a grid-enabled system for scientific workflows. In *the Workflow in Grid Systems Workshop in GGF10-The Tenth Global Grid Forum, Berlin, Germany*, 2004. 75

Abdulsalam Alzubbi, Amadou Ndiaye, Babak Mahdavi, Francois Guibault, Benoit Ozell, and Jean-Yves Trepanier. On the use of JAVA and RMI in the development of a computer framework for MDO. In *8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, 2000. 40

Amazon Web Services, Inc. AWS Elastic Beanstalk. `http://aws.amazon.com/elasticbeanstalk/`, 2011. URL `http://aws.amazon.com/elasticbeanstalk/`. [Accessed: 30/12/2013]. 80

AppFog, Inc. appfog. `https://www.appfog.com/`, 2013. URL `https://www.appfog.com/`. [Accessed: 30/12/2013]. 79

Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, et al. Web service choreography interface (wsci) 1.0. *Standards proposal by BEA Systems, Intalio, SAP, and Sun Microsystems*, 2002. 47, 48

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003. 98

Adam Barker and Jano Van Hemert. Scientific workflow: a survey and research directions. In *Parallel Processing and Applied Mathematics*, pages 746–753. Springer, 2008. 54

David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. 2004. 13, 34

Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2000. 25, 32, 33, 40

Jerzy Brzeziński, Arkadiusz Danilecki, Jakub Flotyński, Anna Kobusińska, and Andrzej Stroiński. Roswel workflow language: A declarative, resource-oriented approach. *New Generation Computing*, 30(2-3):141–164, 2012. 17, 125, 126, 127, 131, 132

Junwei Cao, Stephen A Jarvis, Subhash Saini, and Graham R Nudd. Gridflow: Workflow management for grid computing. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 198–205. IEEE, 2003. 43, 44

William Y Chang, Hosame Abu-Amara, and Jessica Feng Sanford. *Transforming enterprise cloud services*. Springer Science & Business Media, 2010. 98

Neil Chapman, Simone Ludwig, William Naylor, Julian Padget, and Omer Rana. Matchmaking support for dynamic workflow composition. In *Proceedings of 3rd IEEE International Conference on eScience and Grid Computing*, pages 371–378, Bangalore, India, December 2007. IEEE, IEEE. DOI:10.1109/E-SCIENCE.2007.48. 51, 121

Kyle Chard, Wei Tan, Joshua Boverhof, Ravi Madduri, and Ian Foster. Wrap scientific applications as wsrf grid services using gravi. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 83–90. IEEE, 2009. 45, 75

Ann L Chervenak, Robert Schuler, Matei Ripeanu, Muhammad Ali Amer, Shishir Bharathi, Ian Foster, Adriana Iamnitchi, and Carl Kesselman. The globus replica location service: design and experience. *Parallel and Distributed Systems, IEEE Transactions on*, 20(9):1260–1272, 2009. 44

Workflow Manage Coalition. Terminology & glossary. Technical report, Technical Report WFMC-TC-1011, 1996. 52

OMG Corba. Common object request broker architecture, 1995. 25, 40

Tom Crick, Peter Dunning, Hyunsun Kim, and Julian Padget. Engineering design optimization using services and workflows. *Phil. Trans. R. Soc. A*, 367(1898):2741–2751, 2009. 41

Francisco Curbera, Yaron Goland, Johannes Klein, Frank Leymann, S Weerawarana, et al. Business process execution language for web services, version 1.1. 2003. 48

Jos De Bruijn, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, M Kifer, B König-Ries, J Kopecky, R Lara, E Oren, et al. Web service modeling ontology (wsmo). *Interface*, 5:1, 2005a. 50

Jos De Bruijn, Dieter Fensel, M Kifer, J Kopeck, R Lara, H Lausen, A Polleres, D Roman, J Scicluna, and I Toma. Relationship of wsmo to other relevant technologies. *W3C Member Submission*, 2005b. 51

Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, 2009. 42

Docker. What is Docker. `https://www.docker.com/whatisdocker`. URL `https://www.docker.com/whatisdocker`. [Online; accessed 29-Sept-2015]. 100

Docker. What is Docker? `https://www.docker.com/what-docker`, 2016. URL `https://www.docker.com/what-docker`. [Online; accessed 17-June-2016]. 13, 14, 100, 101

MS Eldred, BM Adams, DM Gay, LP Swiler, K Haskell, WJ Bohnhoff, JP Eddy, WE Hart, JP Watson, JD Griffin, et al. Dakota version 4.1 users manual. Technical report, Sandia Technical Report SAND2006-6337, Sandia National Laboratories, Albuquerque, NM, 2007. 49, 2007. 163

Cristina Feier, John Domingue, Jos de Bruijn, Dieter Fensel, Holger Lausen, and Axel Polleres. D3. 1v0. 2 wsmo primer. *http://www. wsmo. org/TR/d3/d3*, 1:v0, 2005. 51

Ian Fette and Alexey Melnikov. The websocket protocol. 2011. 175

Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999. 36

Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. 35, 36

Ian Foster. Lessons from industry for science cyberinfrastructure: Simplicity, scale, and sustainability via SaaS/PaaS. 06 2015. URL http://dx.doi.org/10.6084/m9.figshare.1449018. 96

Ian Foster and Carl Kesselman. The globus toolkit. *The grid: blueprint for a new computing infrastructure*, pages 259–278, 1999. 41

Naoki Fukuta, Ken Satoh, and Takahira Yamaguchi. Towards kiga-kiku services on speculative computation. In *Practical Aspects of Knowledge Management*, pages 256–267. Springer, Yokohama, Japan, 2008. 129, 130

Galaxy Team. Galaxy optimization. http://galaxy.psu.edu/, 2008. URL http://galaxy.psu.edu/. 55

Francis Galiegue and Kris Zyp. Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, 2013. 105

Yolanda Gil, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers. Examining the challenges of scientific workflows. *Ieee computer*, 40(12):26–34, 2007. 42

Carole Anne Goble and David Charles De Roure. myexperiment: social networking for workflow-using e-scientists. In *Proceedings of the 2nd workshop on Workflows in support of large-scale science*, pages 1–2. ACM, 2007. 91

Li Gong. Jxta: A network programming environment. *Internet Computing, IEEE*, 5 (3):88–95, 2001. 44

Google. Google App Engine. http://developers.google.com/appengine/, 2008. URL http://developers.google.com/appengine/. [Online; accessed 02-May-2013]. 46, 79

GoPivotal, Inc. Cloud Foundry. http://www.cloudfoundry.com/, 2011. URL http://www.cloudfoundry.com/. [Accessed: 24/08/2013]. 46, 79

Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *Services Computing, IEEE Transactions on*, 3(3):223–235, 2010. 137

Hugo Haas and Allen Brown. Web Services Glossary. `http://www.w3.org/TR/ws-gloss/`, 2004. URL `http://www.w3.org/TR/ws-gloss/`. [Online; accessed 11-February-2011]. 22

David Hollingsworth and UK Hampshire. Workflow management coalition the workflow reference model. *Workflow Management Coalition*, 68, 1993. 13, 53

Ali Hussain. Performance of Docker vs VMs. `http://www.slideshare.net/Flux7Labs/performance-of-docker-vs-vms`, 2014. URL `http://www.slideshare.net/Flux7Labs/performance-of-docker-vs-vms`. [Online; accessed 29-Sept-2015]. 101

ImageMagick Studio. Imagemagick. `http://www.imagemagick.org`, 1999. URL `http://www.imagemagick.org`. [Online; accessed 15-Jan-2014]. 164

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: http://doi.acm.org/10.1145/1272996.1273005. URL `http://doi.acm.org/10.1145/1272996.1273005`. 43

David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985. 129

Jelastic, Inc. Jelastic. `http://jelastic.com//`, 2011. URL `http://jelastic.com/`. [Accessed: 30/12/2013]. 80

Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007. 149, 154

Gopi Kandaswamy, Liang Fang, Yi Huang, Satoshi Shirasuna, Suresh Marru, and Dennis Gannon. Building web services for scientific grid applications. *Ibm Journal of Research and Development*, 50:249–260, 2006. doi: 10.1147/rd.502.0249. 45, 75

Rania Khalaf, Nirmal Mukhi, and Sanjiva Weerawarana. Service-oriented composition in bpel4ws. In *WWW (Alternate Paper Tracks)*, 2003. 47

Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007. 98

S. Kodiyalam and J. Sobieszczanski-Sobieski. Multidisciplinary design optimisation-some formal methods, framework requirements, and application to vehicle design. *International Journal of Vehicle Design*, 25(1):3–22, 2001. URL http://inderscience.metapress.com/index/903X7U03T201VDTG.pdf. 24

Eleni A Kolokotroni, Eleni Ch Georgiadi, Dimitra D Dionysiou, and Georgios S Stamatakos. A 4-d simulation model of tumor free growth and response to chemotherapy in vivo: The breast cancer case. *Zografeio Lyceum, Istanbul, Turkey September 23-24, 2008*, page 31, 2008a. 178

Eleni A Kolokotroni, Georgios S Stamatakos, Dimitra D Dionysiou, Eleni Ch Georgiadi, Christine Desmedt, and Norbert M Graf. Translating multiscale cancer models into clinical trials: Simulating breast cancer tumor dynamics within the framework of the trial of principle clinical trial and the acgt project. In *BioInformatics and BioEngineering, 2008. BIBE 2008. 8th IEEE International Conference on*, pages 1–8. IEEE, 2008b. 178

Eleni A Kolokotroni, Dimitra D Dionysiou, Nikolaos K Uzunoglu, and Georgios S Stamatakos. Studying the growth kinetics of untreated clinical tumors by using an advanced discrete simulation model. *Mathematical and Computer Modelling*, 54 (9):1989–2006, 2011. 178

Jacek Kopeckỳ, Matthew Moran, Tomas Vitvar, Dumitru Roman, and Adrian Mocan. D24. 2v0. 1. wsmo grounding. *WSMO Working Draft. http://www. wsmo. org/TR/d24/d24*, 2:v0, 2005. 51

Jacek Kopeckỳ, Tomas Vitvar, Carine Bournez, and Joel Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *Internet Computing, IEEE*, 11(6):60–67, 2007. 48, 49, 51

Petros Koutoupis. RapidDisk. `http://www.rapiddisk.org/`, 2010. URL `http://www.rapiddisk.org/`. [Online; accessed 12-August-2014]. 174

Sriram Krishnan, Brent Stearn, Karan Bhatia, Kim K. Baldridge, Wilfred W. Li, and Peter Arzberger. Opal: SimpleWeb Services Wrappers for Scientific Applications. *Web Services, IEEE International Conference on*, 0:823–832, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/ICWS.2006.96. 44, 75, 76

Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981. 129

Menno Lageman and Sun Client Solutions. Solaris containerswhat they are and how to use them. 2005. 99

Butler Lampson. Lazy and speculative execution in computer systems. In *Principles of Distributed Systems*, pages 1–2. Springer, 2006. 129

Edward A Lee and Steve Neuendorffer. *MoML: A Modeling Markup Language in SML: Version 0.4*. Citeseer, 2000. 55

Ho-Jun Lee, Jae-Woo Lee, and Jeong-Oog Lee. Development of web services-based multidisciplinary design optimization framework. *Advances in Engineering Software*, 40(3):176–183, 2009. 41

Frank Leymann et al. Web services flow language (wsfl 1.0), 2001. 53, 54

J. Lindenbaum, A. Wiggins, and O. Henry. Heroku. `http://www.heroku.com`, 2008. URL `http://http://www.heroku.com`. [Online; accessed 02-May-2013]. 46, 78

David Liu, Jun Peng, Gio Wiederhold, Ram D. Sriram, Corresponding Aruthor, Kincho H. Law, and Kincho H. Law. Composition of engineering web services with distributed data flows and computations, 2005. 43

Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. 21, 42, 43, 44, 121

Bakak Mahdavi. The design of a distributed, object-oriented, component-based framework in multidisciplinary design optimization. Master's thesis, School of Computer Science, McGill University, Montreal, 2002. 40

David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, et al. Owl-s: Semantic markup for web services. *W3C member submission*, 22:2007–04, 2004. 50

Luis Martin, Alberto Anguita, Norbert M Graf, Manolis Tsiknakis, Mathias Brochhausen, Stefan Rüping, Anca ID Bucur, Stelios Sfakianakis, Thierry Sengstag, Francesca Buffa, et al. Acgt: advancing clinico-genomic trials on cancer-four years of experience. In *MIE*, pages 734–738, 2011. 55

Joaquim R. R. A. Martins and Andrew B. Lambe. Multidisciplinary design optimization: A survey of architectures. *AIAA Journal*, 51:2049–2075, 2013. doi: 10.2514/1.J051895. 20

P. Mell and T. Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology, Special Publication 800-145*, 2011. URL http://developers.google.com/appengine/. [Online; accessed 02-May-2013]. 46, 74, 78, 98

Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014. 99

Microsoft. Windows Azure. http://www.windowsazure.com/, 2010. URL http://www.windowsazure.com/. [Accessed: 30/12/2013]. 80

Microsoft. Trident project. http://tridentworkflow.codeplex.com/, 2011. URL http://tridentworkflow.codeplex.com/. 42

Nilo Mitra, Yves Lafon, et al. Soap version 1.2 part 0: Primer. *W3C recommendation*, 24:12, 2003. 13, 35

Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, page 9, Berkeley, CA, USA, 2011. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1972470. 43

Henry Ng, Suleyman Geluyupoglu, Frank Segaria, Brett Malone, Scott Woyak, and Greg Salow. Collaborative Engineering Enterprise with Integrated Modeling Environment. http://www.phoenix-int.com/resources/CollaborativeEngineering.php, 2003. URL http://www.phoenix-int.com/resources/CollaborativeEngineering.php. [Online; accessed 08-August-2011]. 40

Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004. 21, 42, 43, 44, 74, 75, 121

Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: Lessons in creating a workflow environment for the life sciences: Research articles. *Concurr. Comput. : Pract. Exper.*, 18 (10):1067–1100, August 2006. ISSN 1532-0626. doi: 10.1002/cpe.v18:10. URL http://dx.doi.org/10.1002/cpe.v18:10. 54

OpenStack project. Openstack. http://www.openstack.org/, 2012. URL http://www.openstack.org/. [Accessed: 30/12/2013]. 80

Oracle. ramdiskadm - administer ramdisk pseudo device. http://docs.oracle.com/cd/E23824_01/html/821-1462/ramdiskadm-1m.html, 2011. URL http://docs.oracle.com/cd/E23824_01/html/821-1462/ramdiskadm-1m.html. [Online; accessed 12-August-2014]. 174

ORACLE. Remote Method Invocation Home. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html, 2011.

URL http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html. [Online; accessed 07-August-2011]. 25, 40

Sharon L. Padula and Ronnie E. Gillian. Multidisciplinary environments: A history of engineering framework development. In *11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 2006. 24

Luca Panziera, Marco Comerio, Matteo Palmonari, and Flavio De Paoli. Distributed matchmaking and ranking of web apis exploiting descriptions from web sources. In *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*, pages 1–8. IEEE, 2011. 137

Cesare Pautasso and Erik Wilde. Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. In *Proc. of the 18th International World Wide Web Conference (WWW2009)*, pages 911–920, Madrid, Spain, April 2009. URL http://dret.net/netdret/docs/loosely-coupled-www2009/. 22, 37

Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. In *17th International World Wide Web Conference (WWW2008)*, pages 805–814, Beijing, China, April 2008 2008. URL http://www2008.org/. 22, 51

Persistence of Vision Raytracer Pty. Ltd. Povray. http://www.povray.org/, 2003. URL http://www.povray.org/. [Online; accessed 15-Jan-2013]. 164

Maja Pesic and Wil MP Van der Aalst. A declarative approach for flexible business processes management. In *Business Process Management Workshops*, pages 169–180. Springer, 2006. 123

James L Peterson. Petri net theory and the modeling of systems. 1981. 141, 146

Phoenix Integration. Analysis Server User Manual. http://www.phoenix-int.com/~AnalysisServer/main.html, 2001. URL http://www.phoenix-int.com/~AnalysisServer/. [Online; accessed 23-October-2014]. 40

Giulio Piancastelli and Andrea Omicini. A multi-theory logic language for the world wide web. In *Logic Programming*, pages 769–773. Springer, 2008. 14, 124, 125, 126, 131

Prabhakar Raghavan, Hadas Shachnai, and Mira Yaniv. Dynamic schemes for speculative execution of code. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1998. Proceedings. Sixth International Symposium on*, pages 309–314. IEEE, 1998. 129

Red Hat, Inc. Openshift. https://www.openshift.com/, 2011. URL https://www.openshift.com/. [Accessed: 24/08/2013]. 46, 79

Red hat, Inc. Openshift Origin. http://openshift.github.io/, 2011. URL http://openshift.github.io/. [Accessed: 30/12/2013]. 80

Jingyuan Ren, Nadya Williams, Luca Clementi, Sriram Krishnan, and Wilfred W Li. Opal web services for biomedical applications. *Nucleic acids research*, 38(suppl 2): W724–W731, 2010. 46, 75

Leonard Richardson and Sam Ruby. *The Resource Oriented Architecture*, chapter 4, page 79. O'REILLY, 2007. 38

Nick Russell, Arthur HM Ter Hofstede, David Edmond, and Wil MP van der Aalst. Workflow data patterns: Identification, representation and tool support. In *Conceptual Modeling–ER 2005*, pages 353–368. Springer, 2005. 23

N Sakimura. Json web token (jwt). Technical report, 2015. No. RFC 7519. 108

A. O. Salas and J. C. Townsend. Framework requirements for mdo application development. In *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, pages 98–4740, 1998. 21, 24

Sandia National Laboratories. The DAKOTA Project. http://dakota.sandia.gov/, 2010. URL http://dakota.sandia.gov/. [Online; accessed 08-March-2013]. 41, 74, 161

Leo Sauermann, Richard Cyganiak, and Max Völkel. Cool uris for the semantic web. 2011. 38

M. Senger, A. Rice, P. andBleasby, T. Oinn, and M. Uludag. Soaplab2: more reliable Sesame door to bioinformatics programs. In *Proc. 9th Bioinformatics Open Source Conference (BOSC 2008)*, 2008a. 75

Martin Senger, Peter Rice, Alan Bleasby, Tom Oinn, and Mahmut Uludag. Soaplab2: more reliable Sesame door to bioinformatics programs, 2008b. 41, 44

Amit P Sheth, Karthik Gomadam, and Jon Lathem. Sa-rest: semantically interoperable and easier-to-use services and mashups. *IEEE Internet Computing*, 11(6):91–94, 2007. 52

Y. Simmhan, C. van Ingen, G. Subramanian, and Jie Li. Bridging the gap between desktop and the cloud for escience applications. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 474–481, Chengdu, China, July 2010. IEEE. doi: 10.1109/CLOUD.2010.72. 47, 80

Munindar P. Singh and Mladen A. Vouk. Scientific Workflows: Scientific Computing Meets Transactional Workflows, 1996. URL http://www.csc.ncsu.edu/faculty/mpsingh/papers/databases/workflows/sciworkflows.html. 18

Jan Wielemaker Paul Singleton, Fred Dushin, and Jan Wielemaker. Jpl: A bidirectional prolog/java interface, 2004. 151

M. Sobolewski. FIPER: The Federated S2S Environment. In *JavaOne, Sun's 2002 Worldwide Java Developer Conference*, 2002. 41

Brennan Spies. Web services, part 1: Soap vs. rest. *Retrieved March*, 12:2010, 2008. 16, 32, 33

Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. The triana workflow environment: Architecture and applications. In *Workflows for e-Science*, pages 320–339. Springer, 2007. 21, 42, 43, 44, 55

Satish Thatte. Xlang: Web services for business process design. *Microsoft Corporation*, 2001, 2001. 53

The OpenMDAO development team . OpenMDAO. http://openmdao.org, 2010. URL http://openmdao.org/. [Online; accessed 08-March-2013]. 74, 161

The openMDAO development team. Multidisciplinary Design Feasible (MDF). http://openmdao.org/releases/0.5.0/docs/mdao/mdf.html,

2013. URL http://openmdao.org/releases/0.5.0/docs/mdao/mdf.html. [Online; accessed 14-August-2014]. 162

The Trustees of Indiana University. GFac User Manual. http://www.extreme.indiana.edu/gfac/userguide.html, 2006. URL http://www.extreme.indiana.edu/gfac/userguide.html. [Online; accessed 28-December-2013]. 13, 77

The Trustees of Indiana University. Invokeing Gfac Web Service. http://www.extreme.indiana.edu/gfac/invoke.html, 2007. URL http://www.extreme.indiana.edu/gfac/invoke.html. [Online; accessed 28-December-2013]. 77

The YAWL Foundation. YAWL Project. http://www.yawlfoundation.org/, 2004. URL http://www.yawlfoundation.org/. [Online; accessed 15-August-2014]. 42

University of Manchester. Opal plugin for Taverna 2.2. http://www.taverna.org.uk/2011/02/15/opal-plugin-for-taverna-2-2/, 2010. URL http://www.taverna.org.uk/2011/02/15/opal-plugin-for-taverna-2-2/. [Online; accessed 6-August-2014]. 45

Wil MP Van Der Aalst. The application of petri nets to workflow management. *Journal of circuits, systems, and computers*, 8(01):21–66, 1998. 53

Wil MP Van Der Aalst and Maja Pesic. *DecSerFlow: Towards a truly declarative service flow language*. Springer, 2006. 127

Wil MP Van Der Aalst and Arthur H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005. 21, 55

Wil MP Van Der Aalst, Arthur HM Ter Hofstede, Bartek Kiepuszewski, and Alistair P Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003. 55

Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009. 98

vmware. ESX and ESXi. http://www.vmware.com/uk/products/esxi-and-esx/overview, 2012. URL http://www.vmware.com/uk/products/esxi-and-esx/overview. [Online; accessed 26-Sept-2015]. 98

W3C. Web Services Architecture. http://www.w3.org/TR/ws-arch/#relwwwrest, 2004. URL http://www.w3.org/TR/ws-arch/#relwwwrest. [Online; accessed 08-August-2011]. 22, 25, 40, 51

Joan D. Walton, Robert E. Filman, and David J. Korsmeyer. The evolution of the DARWIN system. In *Proceedings of the 2000 ACM symposium on Applied computing - Volume 2*, SAC '00, pages 971–977, New York, NY, USA, 2000. ACM. ISBN 1-58113-240-9. doi: http://doi.acm.org/10.1145/338407.338709. URL http://doi.acm.org/10.1145/338407.338709. 39

webmaster@ftb.ca.gov. Business Process Management Center of Excellence Glossary. https://www.ftb.ca.gov/aboutFTB/Projects/ITSP/BPM_Glossary.pdf, 2009. URL https://www.ftb.ca.gov/aboutFTB/Projects/ITSP/BPM_Glossary.pdf. [Online; accessed 02-Jan-2016]. 42

R. P. Weston and J. C. Townsend. A Distributed Computing Environment for Multidisciplinary Design. In *5th AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City Beach, FL*, pages 94–4372, 1994. 39

Stephen A White. Introduction to bpmn. *IBM Cooperation*, 2(0):0, 2004. 53

Alan Williams and Stian Soiland-Reyes. SCUFL2. http://dev.mygrid.org.uk/wiki/display/developer/SCUFL2, 2012. URL http://dev.mygrid.org.uk/wiki/display/developer/SCUFL2. [Online; accessed 06-March-2014]. 54

Qun Zhou, Babak Mahdavi, Dao Jun Liu, Francois Guibault, Benoit Ozell, and Jean-Yves Trepanier. A web-based distribution protocol for large scale analysis and optimization applications. In *High Performance Computing Systems and Applications*, pages 257–270. Springer, 2003. 40