A Dynamic web service scheduling and deployment framework for grid workflow

Shayan Shahand

2010

Shayan Shahand. (2010). A Dynamic web service scheduling and deployment framework for grid workflow. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/47470>

<https://doi.org/10.32657/10356/47470>

Nanyang Technological University

Downloaded on 26 Apr 2023 23:04:18 SGT

58
2X
9/8he

A DYNAMIC WEB SERVICE SCHEDULING AND DEPLOYMENT FRAMEWORK FOR GRID WORKFLOW

SHAYAN SHAHAND

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Master of Engineering

2010

Abstract

Grid computing boosts productivity by maximizing resource utilization and simplifying access to resources which are shared among virtual organizations. Recently, the Grid and Web Service communities have established a set of common interests and requirements. The latest version of the Globus Toolkit implements the Web Service Resource Framework (WSRF) specifications which have been formulated to cover these interests. However, it has some limitations in supporting the dynamic nature of large-scale Grid and data-intensive workflow executions. Dynamic Web Service deployment fits well into the dynamic nature of the Grid and opens new ways of managing workflow executions on the Grid.

So far, there is little work on deploying Web Services dynamically on Grid computing resources. In this thesis, we design a *dynamic Web Service scheduling and deployment framework* that supports the workflow management of dynamic services. Dynamic Web Service deployment on the Grid enables the system to distribute workflow jobs on the Grid computing resources and allows jobs to be executed on the same site as where the input data is located. The empirical studies show that the designed framework increases resource utilization and decreases workflow execution time. We argue that the framework ensures more flexible, fault-tolerant workflows and also reduces the likelihood and frequency of bottlenecks. The system is based on Open Grid Services Architecture specifications and is WSRF-compliant.

Keywords: Grid computing, Globus Toolkit 4, Web Service, Dynamic deployment, Web Service workflow, Load balancing, Data location awareness.

Acknowledgments

First and foremost, I would like to take this opportunity to express my most sincere gratitude and appreciation to my supervisor, Professor Stephen John Turner, and also to Professor Wentong Cai for their countless efforts in guiding and encouraging me throughout this project. Their knowledgeable suggestions and heuristic advice always inspired me on my way.

Next, I would like to show my appreciation to Ms. Ng-Goh Siew Lai, for her continuous technical support and help in experimental work on the PDCC Grid. In addition, a great deal of gratitude also goes to Dr. Yuhong Feng, Dr. Michael H. Lees, Mr. Heiko Aydt, Mr. Jasleen Singh, Mrs. Maryam Khademi, Mr. Ke Pan. They have provided me with numerous miscellaneous support, without which the realization of many tasks and experiments would not be possible.

Last but not least, I am cordially grateful to my parents, my wife, Narges, and my friends for their unconditional support, endless love and prudent advice.

This project was funded by the Agency for Science, Technology and Research (A*STAR).

Contents

Abstract	i
Acknowledgments	iii
1 Introduction	1
1.1 Background and Motivation	1
1.2 Scope and Objectives	5
1.3 Organization of the Thesis	6
2 Literature Review	9
2.1 Definitions and Taxonomies	9
2.1.1 Web Services	9
Service-Oriented Architecture (SOA)	10
Web Service Technologies	10
Web Services Resource Framework	13
WS-BPEL	15
2.1.2 Taxonomy of Workflows	16
Workflow Structure	16
Workflow Specification	17

2.1.3	Taxonomy of Workflow Management Systems	18
	Execution Control Management	19
	Intermediate Data Movement	21
	Job Code Acquisition	22
	Mapping	23
2.1.4	Summary	24
2.2	Existing Systems Overview	25
2.2.1	Globus Toolkit	26
2.2.2	Condor	30
2.2.3	DAGMan	32
2.2.4	GriPhyN VDS	34
2.2.5	Pegasus	34
2.2.6	Mobile Code Collaboration Framework (MCCF)	36
2.2.7	ServiceGlobe	36
2.2.8	Summary	38
3	Framework Design	41
3.1	Overview	42
3.2	Supporting Components Layer	45
3.2.1	Dynamic Web Service Factory (WS-Factory)	47
	Resource Properties	49
	Operations	50
3.2.2	Web Service Code Repository (WS-CodeR)	51
	Resource Properties	51
	Operations	52
3.2.3	Resource Information System (RIS)	53

GT4 Information Services	53
Ganglia	54
SysMon	54
Speedo	55
3.3 Scheduling Layer	55
3.3.1 Web Service Scheduler (WS-Scheduler)	55
Scheduling Algorithms	57
Load Balancing Algorithm	57
Input Data Location Aware Algorithm	58
Resource Properties	59
Operations	59
3.3.2 Framework Database	62
3.4 WS-Workflow Orchestration Layer	62
3.4.1 WS-Workflow Orchestration Engine (WS-Orc)	63
3.5 Summary	63
4 Empirical Studies	67
4.1 Introduction	68
4.1.1 Experimental Environment	69
4.1.2 Service-Level Dynamic Deployment Emulator	69
4.1.3 WAN Model	70
4.2 Compute Intensive Workflow	72
4.2.1 Determination of Parameters	73
Service-level Dynamic Deployment	74
Container-level Dynamic Deployment	75
Discussion	75

4.2.2	Comparison of Static and Dynamic Deployment	76
4.2.3	Discussion	77
4.3	Data Intensive Workflow	90
4.3.1	Comparison of Static and Dynamic Deployment	91
4.3.2	Discussion	91
4.4	Summary	95
5	Conclusions	97
5.1	Summary of Contributions	97
5.2	Recommendations for Future Work	98
	Bibliography	101
	Appendices	109
A	Publications	111
A.1	Conference Papers	111
B	Glossary	112

List of Figures

1.1	The converging specifications of the Grid community and the Web Service community	4
2.1	Standard Web Service architecture	11
2.2	A Java Web Service component in a GAR file	12
2.3	Server side in Web Service applications	13
2.4	Taxonomy of workflows	16
2.5	Taxonomy of workflow management systems	19
2.6	Relationship between OGSA, GT4, WSRF, and Web Services	27
2.7	Components in the Globus Toolkit	29
2.8	An example of Grid workflow management system layers	33
3.1	DynaSched architecture	44
3.2	Typical topology of the network, underlying resources and location of the framework components	46
3.3	Container-level deployment sequence	48
3.4	Service-level deployment sequence	49
3.5	Dynamic Web Service scheduling and deployment sequence diagram	65
4.1	Normal distribution of measured dynamic deployment time cost	71
4.2	Generated WAN model	72

4.3 Compute-intensive workflow 78

4.4 Average WS-Workflow execution time for various values of p using service-level dynamic deployment 79

4.5 Number of running services on every compute-node for various values of p in one set of experiments when using service-level dynamic deployment . . 80

4.6 Number of running services on every compute-node for various values of p in one set of experiments when using service-level dynamic deployment (Continued) 81

4.7 Standard deviations of number of running services for various values of p in one set of experiments when using service-level dynamic deployment 82

4.8 Average of maximum and average of mean of standard deviations of number of running services for various values of p for all sets of experiments when using service-level dynamic deployment 83

4.9 Average WS-Workflow execution time for various values of p using container-level dynamic deployment 84

4.10 Number of running services on every compute-node for various values of p in one set of experiments when using container-level dynamic deployment . 85

4.11 Number of running services on every compute-node for various values of p in one set of experiments when using container-level dynamic deployment (Continued) 86

4.12 Standard deviations of number of running services for various values of p in one set of experiments when using container-level dynamic deployment . . 87

4.13 Average of maximum and average of mean of standard deviations of number of running services for various values of p for all sets of experiments when using container-level dynamic deployment 88

4.14 Average WS-Workflow execution time for various compute-intensive WS-Workflow sizes 89

4.15 Data-intensive workflow 92

4.16 Average WS-Workflow execution time for various data-intensive WS-Workflow sizes 93

4.17 Average amount of data transferred for various data-intensive WS-Workflow sizes 94

List of Tables

2.1	Mapping DAGMan features to the taxonomy	33
2.2	Mapping Pegasus features to the taxonomy	35
2.3	Mapping MCCF features to the taxonomy	36
2.4	Mapping Service Globe features to the taxonomy	38
2.5	Detailed comparison between Service Globe and the proposed system . . .	39
2.6	Gap analysis of examined systems	40
3.1	Mapping DynaSched features to the taxonomy	64
4.1	Synthetic compute-intensive services used in our experiments	73
4.2	Input files used in our experiments	90

List of Algorithms

1	Load balancing algorithm	58
2	Input data location aware algorithm	60

Chapter 1

Introduction

1.1 Background and Motivation

During the last two decades, computing research has helped scientists apply distributed computing to challenging projects that pushed the limits of what could be done with conventional computing technology. Distributed computing technology has evolved into the Grid, a powerful computing environment to share geographically distributed, heterogeneous and dynamic resources among virtual organizations. The term resource in Grid computing is a general term to denote a computational, storage or instrument resource. Compute intensive applications and data intensive applications seem most likely to benefit from the application of the Grid technology. For instance, data intensive applications, such as experimental data analysis, execute a job on huge data sets, which could be distributed geographically [1].

Workflow is a common technology for developing and executing such applications to harness distributed resources over the Grid. A workflow specifies a series of activities to be executed at a specific time and in a particular sequence. These workflows may either consist of conventional compiled executable jobs written in C/Java or be composed of Web

Services. Workflows which are composed of Web Services are further referred to as *Web Service Workflows (WS-Workflows)* in this thesis. WS-Workflows are either *concrete* or *abstract*. In abstract workflows, jobs are not bound to specific resources for execution while in concrete workflows, jobs are bound to specific resources. Every abstract WS-Workflow must be converted to a concrete one before execution. This means that each job in the workflow should be bound to a particular Web Service which is deployed on a certain machine on the Grid.

The next generation of scientific workflows, if realized as WS-Workflows, can benefit from loosely coupled services¹, using a Service-Oriented Architecture (SOA), where every resource can be accessed as a service without concern about the underlying platform implementation. Thus, any workflow or user requirement can be addressed by these services [2]. Web Services can be used to develop an architecture according to SOA definitions. Nowadays, many applications are composed of different organizations' Web Services. Furthermore, many scientists want to integrate their contribution with those developed by others. Web Services and SOA are essential technologies to facilitate achieving this goal.

A workflow management system is essential to execute workflows on Grid resources, using their computational power, storage capacity and instrument facilities. The workflow management system interacts with the Grid middleware (which is located between the Application level and System level — e.g., Globus Toolkit) to schedule, execute and monitor jobs on the Grid. Workflow management systems can be classified according to: (i) execution control management (ii) intermediate data movement and (iii) job code acquisition [3,4]. In terms of execution control management, the workflow can be executed on the Grid either by a centralized workflow engine or a distributed one. Distributed engines can be classified according to the architecture of the scheduling system. Two main sub categories of distributed engines are collaborative engines (hierarchical) and mobile

¹For simplicity we use the terms service and Web Service interchangeably throughout this thesis.

agent based engines. Secondly, we classify approaches of intermediate data movement as centralized, mediated and peer-to-peer. In the centralized approach intermediate data (i.e., the output of each job in the workflow) is transferred to a centralized repository while in the mediated approach only the locations of the intermediate data is saved in a database. In the peer-to-peer approach, data is transferred between compute-nodes. Finally, in terms of job code acquisition, each job in the WS-Workflow can be either a *static service* or *dynamic service*. Static services are deployed on a specific machine while dynamic services can be deployed on any arbitrary host on demand². Every approach has its own advantages and disadvantages. We explain each approach in detail and discuss its advantages and disadvantages in Section 2.1. There are many workflow management systems available from both the industrial and academic world. Each of these systems uses a combination of the above mentioned approaches and tries to optimize Grid resource utilization and enhance Grid application efficiency. The different capabilities, advantages and disadvantages of some of the most important systems are discussed in Section 2.2.

Open Grid Services Architecture (OGSA) is a standard SOA based on Web Service concepts and technologies which assures interoperability of heterogeneous Grid resources [9]. Recently the Grid community and the Web Service community have established a set of common interests and requirements. A set of specifications, known as Web Services Resource Framework (WSRF), has been formulated to cover those requirements and realize OGSA (see Figure 1.1). The latest release of Globus Toolkit Ver.4 (GT4), an open source software toolkit for building Grids, includes WSRF-compliant components. GT4 is based on OGSA and is a Grid middleware toolkit which is a de facto globally accepted standard.

The converging specifications of the Grid community and the Web Service community reveal an efficient, flexible and powerful Grid environment. Loosely coupled Web Services

²Although “location dependent” and “location independent” are more specific terms, we use “static service” and “dynamic service” because they are more commonly used in the literature (e.g., [5–8]) and therefore are more tangible for the reader.

are being used to implement the SOA of the GT4, which assures interoperability between heterogeneous systems (i.e., different types of resources can communicate and share information). WS-Workflow can be composed of a number of available Web Services provided by different contributors. A scientist can reuse existing Web Services or WS-Workflows to accelerate composing his own WS-Workflow. This could ease the development of service oriented Grid computing applications.

So far there is little work on dynamically deploying Web Services on available computing resources, and the existing approaches do not use standard specifications like OGSA and WSRF (see Section 2.2 for more information). Deploying Web Services dynamically on available computing resources provides important advantages such as load balancing, minimum input/output data movement, high availability of service, reliability, fault tolerance, flexibility and efficient resource utilization. In this project we design, implement and evaluate a set of components that form an environment which is able to schedule abstract WS-Workflow execution on Grid resources. Web Service jobs of the WS-Workflow can be deployed automatically and dynamically on Grid compute-nodes with the help of these components.

Figure 1.1: The converging specifications of the Grid community and the Web Service community [1]

1.2 Scope and Objectives

In order to address dynamic Web Service deployment on the Grid resources and benefit from such a flexible and dynamic environment, several requirements must be taken into consideration. From a bottom-up view, first we need a set of Web Service factories which are able to automatically deploy Web Services on compute-nodes on demand. Then we require a code repository to supply the Web Service factories with the correct undeployed version of the required Web Service. More importantly, we need a scheduler to manage Web Service factories, code repositories and use Grid resources efficiently and intelligently. This scheduler can optimize WS-Workflow job execution by considering input data location and current workload of each compute-node. Finally, at the top most layer, we need a Workflow orchestration engine to parse and execute an abstract WS-Workflow and take care of synchronization and dependency issues.

In this thesis, we explain the design and implementation of a set of components which facilitate scheduling of dynamic Web Services on Grid resources. These components form a framework which is divided into three layers. These layers from a bottom-up view are: (i) Supporting components layer (ii) Scheduling layer and (iii) WS-Workflow Orchestration layer. In the supporting components layer, two components are developed and integrated with other WSRF enabled components of GT4. One of these components addresses automatic and dynamic deployment of Web Service jobs on Grid compute-nodes while the other provides warehouse functionalities for a Web Service code repository. In the scheduling layer, a Web Service scheduler interacts with the components of the supporting components layer to schedule dynamic Web Service deployments. The scheduler decides which deployed Web Service is the most appropriate or where to deploy a Web Service dynamically, according to parameters such as input data location and/or workload of compute-nodes. In the WS-Workflow Orchestration layer, a WS-Workflow orchestration engine orchestrates

execution of abstract WS-Workflows by converting them to concrete ones using the Web Service scheduler.

1.3 Organization of the Thesis

The remainder of the thesis is organized as follow: Chapter 2 is divided into two sections. First, in Section 2.1 we explain a set of fundamental technical definitions about Web Services. We give a taxonomy of workflows and workflow management systems and then discuss pros and cons of each approach. Second, in Section 2.2 we study major existing Grid workflow management systems in order to obtain a detailed view of different system layers and each layer's responsibilities in the Grid environment. We discuss different capabilities of existing Grid workflow management systems and their advantages and disadvantages in this section. Chapter 2 concludes by identifying possible improvements that can be made in this field. Next, in Chapter 3, the dynamic Web Service scheduling and deployment framework is presented in detail including the system architecture, the components and interactions between them. We divide the framework into three layers: (i) Supporting components layer (ii) Scheduling layer and (iii) WS-Workflow Orchestration layer. We describe components of all layers in detail and identify each component's responsibilities and operations. Two dynamic Web Service scheduling algorithms, each suitable for a specific type of Web Service, are introduced in this chapter. In Chapter 4 we describe the experimental studies performed to evaluate the framework. We examine the performance of the framework by carrying out experiments for two types of WS-Workflows: compute-intensive and data-intensive. We discuss the experimental results to justify how the framework increases resource utilization and decreases WS-Workflow execution time. We argue that the framework ensures more flexible, fault-tolerant WS-Workflows and also reduces the likelihood and frequency of bottlenecks. Chapter 5 concludes the thesis with

1.3 Organization of the Thesis

7

a summary and recommendations for future work.

Chapter 2

Literature Review

This chapter consists of two sections: (2.1) Definitions and Taxonomies and (2.2) Existing Systems Overview. In the first section we start by explaining some fundamental aspects and definitions of Web Services. Then we present a taxonomy of different workflows and workflow execution management systems. The taxonomy section concludes with a discussion of the pros and cons of different particular approaches and a comparison between them. In the second section we study some of the major and common existing workflow management systems in detail. Working with the existing systems makes it easier to understand the responsibilities of different Grid workflow management system layers and discuss the pros and cons of possible approaches.

2.1 Definitions and Taxonomies

2.1.1 Web Services

In order to provide a basic knowledge for understanding the concepts of this project, some fundamental Web Service (WS) technologies and related topics are described in this section.

Service-Oriented Architecture (SOA)

SOA is an evolution of distributed computing and modular programming. In SOA, we are able to utilize and organize distributed capabilities under different ownership domains [10]. SOAs are used to build applications out of software services. These services are not bound together by embedded calls. Instead, a process called orchestration is used to tie various services together in predefined patterns to produce a new application. SOA services are loosely coupled and platform independent. To enable such functionality, requirements regarding service description and communication protocols should be addressed.

Web Service Technologies

Web Services can be used to implement a SOA. Web Services are defined by the World Wide Web Consortium (W3C) to address machine to machine interaction over a network. Each Web Service has an interface described in XML format. Other systems interact with the Web Service by standard messages using its description. These messages are in XML format and travel through networks using different network protocols [11].

There are a number of critical core technologies that define Web Services. These technologies are built on top of existing Web technologies and protocols, such as XML, HTTP, FTP, etc., to shape the Web Service architecture. Web Service architecture involves many layered and interrelated technologies. There are many ways to visualize these technologies, just as there are many ways to build and use Web Services. Figure 2.1 (inspired by [11]) provides one possible illustration of some of these technology families. The following paragraph defines the most important core technologies and some related concepts:

- **SOAP:** Simple Object Access Protocol (SOAP), is a lightweight protocol designed for exchanging structured information in a decentralized, distributed environment. SOAP provides a standard, extensible, composable messaging framework for packing and exchanging XML messages [12]. These XML-based message envelopes (SOAP

Figure 2.1: Standard Web Service architecture (inspired by [11])

messages) can be transfered through the network by different protocols like HTTP, SMTP, FTP and so on.

- **WSDL:** Web Service Description Language (WSDL) is an XML format that describes a Web Service interface. This meta-data service description is used by software systems to generate server and client code and dynamically configure the execution environment in order to maintain coherence and integrity of workflow. System designers also use WSDL to choose the appropriate service for their workflow composition [13].
- **UDDI:** Universal Description, Discovery and Integration (UDDI) is a set of specifications to create XML-based directories of Web Services. It is one of the essential parts of the Web Service technologies. UDDI is a protocol for publishing and discovering meta-data about Web Services. This protocol enables software systems and system designers to find an appropriate Web Service, either at design time or runtime [14].
- **URI:** Uniform Resource Identifier (URI) is the unique address of a particular Web Service by which software systems and/or system designers can access a certain Web

Service. Submitting a query to a UDDI results in a URI.

- **WSDD:** Web Service Deployment Descriptor (WSDD) specifies custom Web Service deployment requirements and configuration. The WSDD file together with the WSDL file and the actual implementation files of a Web Service (Jar files if it is a Java Web Service) are packed by Apache Ant¹ to produce a GAR file which is further referred to as an *undeployed Web Service* in the remainder of this thesis (see Figure 2.2). The GAR files are used by the GT4 and contain all the information and files the Web Service container needs to deploy the Web Service and make it available for invocation.

Figure 2.2: A Java Web Service component in a GAR file [15]

Web Services must be deployed in a Web Service container (or WS container for short) to exploit them. The WS container is located on a host server and is responsible for marshalling, execution and de-marshalling of Web Service invocations. According to the Web Service architecture illustrated in Figure 2.1 we need different layers of functionality to communicate with Web Services and invoke them. Figure 2.3 illustrates server side applications which help Web Service invocation. For example, Apache Axis could be used as the SOAP engine, Jakarta Tomcat as the application server and Apache HTTP server

¹Apache Ant is a software tool for automating the software build process. It is similar to `make` but is written in the Java language, requires the Java platform, and is best suited to building Java projects.

to address HTTP Web server functionalities. This software stack is denoted as a WS container. Once a Web Service is deployed on a server we refer to it as a *deployed Web Service*, otherwise it is called an undeplayed Web Service. *Dynamic deployment* refers to the ability of uploading and deployment of a Web Service into a WS container or undeplacement of an existing Web Service from a WS container.

Figure 2.3: Server side in Web Service applications [15]

Web Services Resource Framework

Web Services are normally stateless, i.e., they retain no data between invocations. The WSRF is a collection of five specifications published by the Organization for the Advancement of Structured Information Standards (OASIS). WSRF provides a set of operations by which stateful Web Services can be implemented [16, 17]. Stateful Web Services are required by the OGSA definition. GT4 implements WSRF to meet the requirements of OGSA [18]. We explain the OGSA definition, GT4 and its relation with WSRF and OGSA later in Section 2.2.1.

The five specifications published by OASIS as WSRF are defined as:

- **WS-Resource:** The state of a stateful Web Service is stored in a separate entity called Resource. Note that the term resource used in WSRF should not be confused with the resource in Grid computing. A stateful Web Service can have one or more resources. A composition of a resource and a Web Service is called WS-Resource. This specification defines a WS-Resource and describes the relationship between a Web Service and a resource in the WS-Resource Framework [19]. Each WS-Resource is identified by an Endpoint Reference (EPR) which contains the URI of the Web Service and the identifier (i.e., resource key) of the stateful resource.
- **WS-ResourceProperties:** A resource is composed of zero or more resource properties. The WS-Resource-Properties specification specifies how resource properties are defined and accessed. It is also a representation of the resource's state. The representation is defined in terms of a resource properties document. The resource properties document defines the means by which to access the resource properties through the Web Service interface and query or update the resource property values [20].
- **WS-ResourceLifetime:** Resources can be created and destroyed at any time, not only when the server is started and destroyed. The lifetime of a WS-Resource is defined as the period between its instantiation and its destruction. The WS-ResourceLifetime specifies basic mechanisms to destroy a WS-Resource and monitor its lifetime [21].
- **WS-ServiceGroup:** The WS-ServiceGroup specification defines the means by which Web Services and WS-Resources can be aggregated or grouped together for a domain specific purpose. This specification defines membership rules, membership constraints and classifications. It specifies how to add a new service to a group, remove a service from a group and find a service in a group with specific condition [22].
- **WS-BaseFaults:** The WS-BaseFaults specification defines the means by which faults are reported in order to support problem determination and fault manage-

ment [23].

Although not included in the WSRF, there are two more related specifications to the WSRF's specifications. The summaries of these specifications are as follows:

- **WS-Notification:** Also specified by OASIS, is a specification to define the standard way by which Web Services interact, using Notifications or Events. This specification allows event driven programming between Web Services. It allows some services to be configured as notification producers and some other services as notification consumers. The notification consumers then subscribe to the producers and whenever a change occurs in the notification producer services the notification consumer services are notified [24].
- **WS-Addressing:** This specification by W3C specifies the mechanism to address Web Services in a more versatile way than plain URIs. By using WS-Addressing specification we can address WS-Resources (i.e., a pair of a Web Service and a resource). WS-Addressing defines XML elements to identify Web Service's EPR and to secure end-to-end endpoint identification in messages [25].

WS-BPEL

Web Services Business Process Execution Language (WS-BPEL) is a language for specifying business processes. WS-BPEL is an orchestration language, by which we can define an interoperable interaction model to provide automated process integration. That means, a series of jobs (processes) can be put together to be executed in a certain pattern (e.g., sequential, parallel). These jobs interact with other jobs by sending outputs to and/or receiving inputs from other jobs. These jobs can be in the form of Web Services. Using loosely coupled Web Services allows automated process integration by putting processes from different providers together to form a workflow. WS-BPEL is built on Web Service standards (see Figure 2.1) and uses Web Services as the model for process decomposition

and assembly. WS-BPEL provides XML and WSDL typed variables, structured programming constructs and simple data manipulation functions. In brief, there are two categories of activities identified in WS-BPEL: basic activities (such as receive, reply, invoke, etc.) and structured activities (such as sequence, flow, if-then-else, while, etc.). In addition, WS-BPEL supports process lifecycle management mechanisms and failure recovery features [26].

2.1.2 Taxonomy of Workflows

In this section we present a taxonomy of workflows. This taxonomy is made according to two workflow orthogonal characteristics: structure and specification. We discuss the pros and cons of each approach at the end of each part. The taxonomy is based on [3, 4, 27]. Figure 2.4 illustrates this classification briefly.

Figure 2.4: Taxonomy of workflows

Workflow Structure

A workflow specifies a series of activities to be executed at a specific time and in a particular logical sequence according to their dependencies. There are two different ways for defining workflow structure:

- **DAG based:** Directed Acyclic Graph (DAG) workflows start from a source and end in a sink job. Between the source and the sink there is a set of paths of connected

jobs, which does not contain a directed cyclic pattern. Workflows are categorized into three groups according to the path patterns between the workflow source and the workflow sink: sequenced, parallelized and choice-based. In the sequenced workflow, each job is executed after its predecessor has completed. In the parallelized workflow there are some jobs that can be executed in parallel because they are independent of each other. In the choice-based workflow, different jobs may be selected to execute at runtime if their associated conditions are satisfied.

- **Non-DAG based:** Similar to DAG based workflows, non-DAG based workflows start from a source and finish at a sink job, but there might be a directed cyclic pattern in between. Hence, loops can be realized in these workflows in addition to those structures described for DAG based workflows. Jobs in a loop block may be performed several times during workflow execution.

Although execution management and workflow partitioning of DAG based workflows are easier, non-DAG based workflows provide a vital structure (i.e., loops) which ease workflow designing.

There are two ways of modeling either DAG based or Non-DAG based workflows: language based and graph based. In the language based modeling either an XML language like WS-BPEL or other formats like Condor DAGman could be used. In the graph based modeling, workflow is specified in terms of nodes and edges.

Workflow Specification

In terms of workflow specification, there are two kinds of workflows:

- **Abstract:** In abstract workflows, jobs are not bound to a specific Grid resource for execution.
- **Concrete:** In contrast with abstract workflows, in a concrete model, workflow jobs are bound to specific resources.

The abstract workflow is fully or partially converted to a concrete one before or during runtime, according to the current state of resources. In some systems this conversion is performed for every individual job just before it is scheduled.

In WS-Workflows, jobs which are explicitly bound to specific deployed Web Services on certain servers form a concrete model. In contrast, a WS-Workflow which is defined by implicitly bound jobs has an abstract specification.

Abstract workflows are more flexible and portable, hence they are easier to design and share. In addition, abstract workflows can be mapped to any suitable Grid resource at runtime which enables load balancing and leads to more efficient resource utilization. Furthermore, in the dynamic nature of Grid resources, abstract workflows are more reliable because their jobs are not bound to any specific resource. On the other hand, exploiting concrete workflows might be more suitable for those who want to control execution location and sequence.

2.1.3 Taxonomy of Workflow Management Systems

In this section we present a classification of workflow execution management systems. This taxonomy is based on Yu and Buyya [3, 27] and Feng [4] and is made according to the following orthogonal qualities:

- Execution control management
- Intermediate data movement
- Job code acquisition
- Mapping

We elaborate advantages and disadvantages of each approach at the end of each part. Figure 2.5 illustrates this classification in summary.

Figure 2.5: Taxonomy of workflow management systems

Execution Control Management

The underlying execution control management or scheduling infrastructure plays an important role in performance, efficiency, autonomy and scalability of Grid workflow management systems. In terms of the scheduling infrastructure, workflow management systems can be classified as follows:

- **Centralized:** In this approach a single central workflow scheduler decides the scheduling of all jobs in the workflow. This central scheduler has the information about the entire workflow and up-to-date information of all available Grid resources.
- **Hierarchical:** In this approach one central manager executes the workflow, performs

workflow partitioning and assigns sub-workflows to lower-level schedulers.

- **Distributed:** Unlike the centralized approach, in the distributed approach, workflow jobs can be scheduled by multiple distributed schedulers. Hence, each scheduler is responsible for scheduling a certain number of jobs and maintains only the information related to a sub-workflow. Distributed approaches can be classified further into two groups:
 - **Collaborative engines:** Multiple schedulers without a central manager form a collaborative distributed scheduling infrastructure. Every scheduler can communicate with other schedulers to assign a sub-workflow to a less loaded one.
 - **Mobile agents:** In this approach a number of mobile agents travel among the Grid resources. These agents schedule and execute jobs on Grid resources and collect information about available resources. Mobile agents can generate new agents, communicate and merge with each other to achieve entire workflow execution.

Some researchers believe that the centralized scheduling approach is more efficient [28] because the central scheduler has overall information about workflow jobs and available resources. The centralized approach is particularly suitable for small workflows or batch-job workflows (which consist of a large number of jobs with the same objectives) and is easier to implement. However, in centralized execution control management, a central scheduler means a single point of failure.

In the hierarchical approach, different scheduling policies can be used for different schedulers in different layers. However, a central manager again means a single point of failure.

Distributed approaches are more scalable in terms of the number of jobs and Grid resources. There is no single point of failure in the collaborative engines and mobile agents scheduling infrastructure and these paradigms are more adaptive to an unpredictable and

dynamic environment like the Grid. However, distributed approaches face more challenges to make a decision for overall optimal performance and minimum conflict problems. In addition, distributed approaches impose additional runtime overhead.

Intermediate Data Movement

Before executing a workflow, the input data needs to be staged on the compute-nodes. Similarly the output data of each job is required to be transferred to the resource where successor jobs are located. The data that is produced during workflow execution is called intermediate data. Some systems require users to manage intermediate data movement by specifying it in the workflow specification, rather than using automated mechanisms for intermediate data movement. Automated data movement approaches could affect overall workflow execution time and efficiency. There are three paradigms for automated intermediate data movement:

- **Centralized:** In the centralized approach, intermediate data is uploaded to a central repository and successor jobs download and stage in their required data from the data repository.
- **Mediated:** Mediated data movement is quite similar to the centralized approach except that in this model, intermediate data remains on the compute-node and only its location is posted to mediator databases for later discovery by job successors.
- **Peer-to-Peer:** In the Peer to Peer (P2P) approach, the intermediate data is transferred directly from the predecessor resource (source) to the successor resource (destination) without involving any third party service.

Besides introducing a single point of failure and bottleneck problems, the centralized approach is not scalable in terms of the number of jobs (and consequently the number of intermediate data files). In addition, using the centralized intermediate data movement paradigm imposes additional overhead especially for data intensive workflows. However,

this model is easier to implement and suitable for applications which do not need large scale data flow. The mediated approach is more scalable and more suitable for applications which need to keep intermediate data for later use (e.g., for recovery purposes), but this model still has the problem of single point of failure. The P2P approach does not have the problems of the two previous models and is suitable especially for large scale intermediate data movement. However, this paradigm is more complex to implement and does not implicitly record intermediate data provenance for future verification purposes. Data provenance refers to the identities of the data sets used in the execution of jobs and the locations of intermediate results.

Job Code Acquisition

Workflow job execution can involve either invoking a method of a Web Service or running conventional compiled executables provided by the user. Web Services are platform-independent and language-independent but impose overhead in transmitting data whereas conventional jobs are platform-dependent and language-dependent but more efficient in data transmission. In the first case we have two different types of services:

- **Static service:** If the service is deployed on a particular computational resource, we refer to it as a static service or location dependent service.
- **Dynamic service:** Unlike a static service, if the service is not bound to a specific compute-node and can be downloaded and deployed on any arbitrary compute-node on demand during runtime, we call it a dynamic service or location independent service.

Similarly, regarding conventional compiled executables, we categorize job code acquisition into two categories of static executable job and dynamic executable job. A static executable job can only be executed on a certain compute-node and is location dependent, whereas a dynamic executable job is location independent.

Dynamic services allow a service to be deployed on computational resources on demand. Hence they can be executed near the required input data set, therefore the data transmission over the network is reduced. Moreover, by taking advantage of dynamic services, a higher level scheduler can balance workload over computational resources. In addition, using dynamic services allows workflow execution to be more reliable and fault tolerant because it allows relocation of a service in the case of unavailability or failure of the host resource. Dynamic services are more suitable for data intensive applications. The same conclusion is applicable for conventionally compiled executables.

Mapping

Depending on the mapping time, when abstract workflows are translated to concrete ones (i.e., when workflow jobs are assigned to computational resources), workflow management systems are categorized into two groups [27]:

- **Static mapping:** With static mapping, an abstract workflow is translated to a concrete one before workflow execution. This translation is made according to the current state of the execution environment, hence the dynamically changing state of the resources is not taken into account.
- **Dynamic mapping:** With dynamic mapping, in contrast to static mapping, both dynamic information and static information are used to translate an abstract workflow to a concrete one at run-time. If the mapping decision for every job in the workflow is made only at the time of job execution, it is called just in-time dynamic mapping.

Because of the dynamically changing Grid environment, where utilization and availability of resources varies over time, and a resource can join or leave the Grid at any time, static mapping may produce a poor schedule. However, it is easier to implement.

On the other hand, as the mapping decisions in the dynamic mapping are made according to the current state of the Grid at run-time, it is more accurate and flexible than static mapping. However, dynamic mapping is more challenging to implement and imposes run-time overhead which is needed to make mapping decisions dynamically.

2.1.4 Summary

In this section we described some fundamental concepts regarding Web Services and presented a taxonomy of workflows and workflow management systems in terms of different orthogonal qualities. The following two paragraphs summarize the pros and cons of different models described previously.

Concerning workflow structure, using non-DAG based structures can improve the ease of workflow design and allow better reconstruction of real world processes. We can describe a workflow by a standard XML based language like WS-BPEL. In terms of workflow specification, abstract workflows are more flexible, portable, reliable and fault tolerant because workflow jobs can be bound to any available resource at runtime. Using abstract workflows can improve workflow performance and efficiency.

Regarding workflow management systems, although decentralized approaches are more adaptive to a dynamic environment like the Grid, the centralized paradigm is more efficient for small workflows and easier to implement. However, a central scheduler means a single point of failure. As mentioned earlier, mediated and P2P intermediate data movements are more efficient especially for data intensive workflows. Dynamic services and dynamic mapping approaches are more flexible, reliable and fault tolerant.

2.2 Existing Systems Overview

Getting familiar with existing systems and obtaining a detailed view of different Grid workflow management system layers is not complete without working with some of the most common systems. Working with existing systems makes it easier to understand the responsibilities of different Grid workflow management system layers and the pros and cons of possible approaches.

The following Grid workflow management systems are chosen as they are well known contributions of different research groups or technical committees. Each of these systems is distinguished by different Grid workflow management system layers. The system proposed in this thesis uses the services provided by Globus Toolkit (see Section 2.2.1) and concepts from these systems:

- **Condor**: This is a specialized workload management system for compute-intensive jobs (job scheduler) [29].
- **DAGMan**: Directed Acyclic Graph Manager (DAGMan) is a meta-scheduler for Condor [30].
- **GriPhyN VDS**: GRId PHYsics Network (GriPhyN) Virtual Data System (VDS), formerly Chimera, provides a set of tools for expressing, executing, and tracking the results of workflows [31].
- **Pegasus**: Planning for Execution in Grids (Pegasus) is a workflow mapping engine which maps high-level workflow descriptions onto distributed infrastructures, this system is included in GriPhyN VDS [32].
- **MCCF**: Mobile Code Collaboration Framework (MCCF) is a Lightweight Mobile Agent (LMA) based workflow management system proposed by Feng and Cai. This system utilizes Mobile Agents (MAs) and dynamic services to manage execution of data-intensive workflows on the Grid [4].

- **ServiceGlobe:** This is an open platform for flexible and reliable Web Service specification, execution and deployment on the Internet [5].

The remainder of this section presents some more detail about Globus Toolkit and the above mentioned Grid workflow management systems. Then we discuss the advantages and disadvantages of these systems by mapping them to the taxonomies we made in Section 2.1. Furthermore, we emphasize on the most important components and systems upon which the proposed dynamic Web Service scheduling and deployment framework relies.

2.2.1 Globus Toolkit

The Globus Toolkit is a set of components that act as the Grid middleware. It is an open source project for building Grids and sharing various resources, from databases to computational resources, across corporate, institutional and geographical boundaries without sacrificing local autonomy and security [33,34]. The Globus Toolkit is developed by the Globus Alliance according to OGSA.

The Global Grid Forum (GGF) defines the OGSA, which is an architecture for a service-oriented grid computing environment for business and scientific use. OGSA is based on loosely coupled services to assure interoperability of heterogeneous resources so that various types of resources can communicate, interoperate, interact and share information [9]. OGSA defines a core set of interfaces, bindings, behaviors and resource models of different capabilities to enable building a service oriented distributed system. The capabilities described are security, execution management, data and resource management, self-management and information. These capabilities are independent of each other and it is not necessary to have them all present in an OGSA system. OGSA was first proposed by Foster et al. [9] and later developed by GGF OGSA working group which resulted in [35].

GT4 implements high-level services which are suitable for Grid-based applications. These services provide essential functionalities of a Grid middleware according to the OGSA

specification. OGSA requires stateful Web Services to enable interoperability among Web Services and resources [36]. Figure 2.6 illustrates the relationship between GT4, OGSA, WSRF and Web Services.

Figure 2.6: Relationship between OGSA, GT4, WSRF, and Web Services [15]

As demonstrated in Figure 2.7, GT4 provides standard services based on the OGSA specification to build the Grid. It supports security, data management, execution management, information services and common runtime functionalities for the Grid middleware. GT4 WS container is a WSRF-compliant Java based hosting environment for stateful Web Services. The GT4 WS containers residing on Grid resources eventually form a virtual

organization. Below we summarize the functionality of some of the most important components both from WS components and non-WS ones [37]:

- **GridFTP:** GridFTP provides high-performance, secure, reliable data transfer for high-bandwidth wide-area networks. This component is based on the well-known File Transfer Protocol (FTP).
- **RFT:** Reliable File Transfer (RFT) is a WSRF-compliant component to manage data transfers over the grid using SOAP messages over HTTP. RFT utilizes GridFTP to create, transfer and delete files on the Grid.
- **RLS:** Replica Location Service (RLS) provides a set of repositories for registration and discovery of replicas. It maps between the logical data items and the target names which could be physical locations or another level of logical naming. Each file has one or more physical locations which are known as its replicas. Logical File Name (LFN) and Physical File Name (PFN) entries are stored in Local Replica Catalogs (LRCs).
- **GRAM:** Grid Resource Allocation Manager (GRAM) provides functionalities to submit, monitor and cancel executable jobs on Grid computing resources. WS GRAM is the Web Service implementation of GRAM. GRAM is not a resource scheduler, but it can operate under the control of a scheduler and provides protocols for communicating with a range of different local resource schedulers.
- **MDS:** Monitoring and Discovering System (MDS) or WS-MDS is a set of Web Services to discover and monitor resources and services on Grids. These services provide both static characteristics (e.g., number of processors) and dynamic information (e.g., workload, available memory) of resources.
- **Java/C WS core:** Java/C WS core provides basic toolset and APIs for developing stateful WSRF-enabled Web Services conforming to WS-Resources and WS-Notification specifications.

2.2 Existing Systems Overview

- **GSI Java/C:** Grid Security Infrastructure (GSI) Java/C is an API library to access GT4’s authentication and authorization capabilities which identify users and servers and support temporary delegation of privileges to others.

More detailed information about GT4 components is accessible via [37].

Figure 2.7: Components in the Globus Toolkit [38]

Globus Toolkit is the de facto standard for Grid computing. It includes a complete implementation of the WSRF specifications on which the Grid middleware services are built according to the OGSA. Grid-based applications are developed on top of the GT4 high-level services. The framework proposed, designed, implemented and evaluated in this thesis uses these services and adds further services to enable dynamic deployment of Web

Services on the Grid.

2.2.2 Condor

Condor is a workload management system for compute-intensive jobs. It has features for job queuing and scheduling based on different scheduling policies. Condor harnesses, manages and monitors resources for job execution. It uses a wide range of resources from idle desktop computers to dedicated clusters. A set of resources which are managed by Condor is further referred to as a Condor pool. Once a user submits a batch of jobs to Condor, it puts them in the queue, schedules them based on scheduling policies, and employs resources to execute jobs. During execution, Condor monitors processes as well as resources and also informs the user upon completion of jobs [29].

There are a number of features provided by Condor [39–41]:

- *Harnessing idle desktop computers*: Condor can be configured to harness desktop computers' idle time. If a desktop computer remains idle (i.e., it has no keyboard and mouse activity) for a period of time, Condor uses it for job execution. In addition Condor has features for producing a checkpoint and migrating the job to another resource in the case of the unavailability of the machine (if a key press or mouse move is detected).
- *No shared file system required*: Condor does not require a shared file system across resources. If there is no shared file system available between cluster machines, Condor transfers the required data file(s) to the computing resource(s) on behalf of the user.
- *Transparent I/O redirection*: Condor redirects any I/O request to the submit host transparently.
- *Matchmaking*: Condor matches job requests with resource specifications. Every machine in a Condor pool advertises itself by specifying a set of requirements and preferences for the jobs which it is willing to run. On the other hand, jobs state both

2.2 Existing Systems Overview

31

the job requirements and preferences. The Condor matchmaking mechanism fulfills every job request by matching it with a desired resource.

- *Flocking*: The Condor flocking mechanism allows two or more Condor pools to combine and work together.
- *Condor-G*: Condor can interact with Grid resources which are managed by Globus. Condor-G enables users to submit jobs to resources in the Grid. Condor-G — a Grid-enabled agent for accessing remote batch systems — uses Grid protocols and services like GSI, Global Access to Secondary Storage (GASS) and GRAM to construct a secure remote execution system.
- *Glide-in*: Condor and Condor-G can be combined together to leverage advantages of both strategies. Glide-in builds a traditional Condor pool on top of a Condor-G system.
- *Condor universes*: A universe in Condor defines an execution environment. The Condor universe set includes: Standard, Vanilla, PVM, MPI, Grid, Java, Scheduler, Local and Parallel. The Standard universe provides reliability and migration through checkpointing and remote system calls. The Vanilla universe is for jobs that can not be re-linked with Condor provided libraries. Condor explicitly supports Message Passing Interface and Parallel Virtual Machine applications through Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) universes, respectively. The Grid universe is for submitting jobs to remote resources under the management of Globus and the Java universe is used for jobs that need Java Virtual Machine for execution [30].

Figure 2.8 demonstrates the position of both Condor (as the site scheduler) and Condor-G in the Grid workflow management system layers.

Condor is a powerful scheduler for compute-intensive and batch jobs but it is not an efficient solution for data-intensive jobs because it transfers files to a central machine

(the submit host) upon completion of each job. Its scheduler daemon is a centralized execution manager and it schedules conventional applications. Users can submit jobs without re-linking them via the vanilla universe, but if they need to guarantee reliability with checkpointing and migration features of Condor, they are required to re-link their jobs with the libraries provided by Condor.

Condor can be used with DAGMan, hence the mapping to the taxonomy is given in Section 2.2.3.

2.2.3 DAGMan

DAGMan is a meta scheduler for Condor. Condor schedules jobs based on available resources but not based on dependencies between them. Hence, a meta scheduler is needed to schedule jobs based on inter-job dependencies. DAGMan takes a DAG file describing inter-job dependencies and submits its jobs to Condor for execution while ensuring their dependencies are satisfied. The workflow is described in a DAG-based structure. Every job in the DAG file has its own submit file so it can be submitted to Condor for execution. DAGMan is responsible for high level scheduling, recovery and reporting for the set of submitted jobs [30]. Figure 2.8 illustrates its position in a Grid workflow management system architecture.

DAGMan is a centralized workflow execution management system. It supports DAG-based workflow structures. The effectiveness of DAGMan for executing data-intensive workflows depends on the user because DAGMan transfers intermediate data in a user-directed manner. For example, the user may specify a P2P method for workflow intermediate data movement. Table 2.1 shows the mapping of DAGMan features to the taxonomy.

²Here, concrete workflow means that workflow jobs are explicitly assigned to a specific condor pool

Figure 2.8: An example of Grid workflow management system layers

Table 2.1: Mapping DAGMan features to the taxonomy

Quality/Feature	DAGMan Type
Workflow structure	DAG-based
Workflow specification	Abstract workflow
Workflow execution control management	Centralized
Intermediate data movement	User-directed
Job code acquisition	Dynamic executable job
Mapping	Dynamic (just in-time)

2.2.4 GriPhyN VDS

In GriPhyN VDS, formerly Chimera, workflows are specified independently from details of the location of files and programs. These workflows are composed of applications. Virtual Data Language (VDL) is used to express these location-independent high-level workflows in a distributed environment. The VDS has a primary planner, named Pegasus, which is described in the following section. VDL definitions are stored in a Virtual Data Catalog (VDC) in order to track provenance of all files derived from the workflow execution for fault resilient goals. VDL workflows are converted to DAG in XML (DAX) by Chimera (GriPhyN VDS). The DAX files are used by the workflow planner (Pegasus) in the VDS to generate executable forms of workflows (i.e., DAGs) [31,42]. Figure 2.8 shows the position of GriPhyN in an example of a Grid workflow management system.

GriPhyN VDS supports DAG-based workflows composed of applications (and not services as yet). It uses a centralized execution control management and transfers intermediate data in a mediated model.

Because GriPhyN uses Pegasus as the workflow planner, the mapping to the taxonomy is given in Section 2.2.5.

2.2.5 Pegasus

Pegasus is a workflow planner which converts abstract workflows to concrete ones including computation on specific resources and data management steps. Pegasus can also be identified as a wrapper for Condor and DAGMan to improve their efficiency for data-intensive workflow execution over the Grid. The location of intermediate data is stored in a LRC of Grid's RLS for later reference. This paradigm is useful either for reusing data, reducing workflow (i.e., by using the data produced out of a previous execution of a job instead of re-executing the job when applicable), transferring intermediate data directly

from source (predecessor) to destination (successor) nodes and recovering from workflow execution failures. Pegasus also uses other components of GT4: MDS helps Pegasus in gathering information about resources and mapping workflow jobs to available resources for execution; GridFTP is used for data stage-in and stage-out; and GRAM is used for remote job submission and management. On top of that, Pegasus also provides workflow partitioning for parallel execution of independent sub-workflows and partition-level failure recovery [32, 43]. In addition, the Pegasus Web portal can be used for submitting, monitoring and managing workflows [44]. Figure 2.8 illustrates its functionality in the Grid workflow management system example.

Pegasus maps workflow jobs to the available resources according to the location of input data. In addition the intermediate data is transfered in a mediated manner which is much more efficient for data-intensive workflows. It also combines a number of small granularity jobs into a job cluster to reduce workflow execution time. Pegasus relies on DAGMan and Condor for submitting jobs to the Grid, hence it is a workflow planner for conventional compiled jobs (and not for Web Service jobs). Table 2.4 shows the mapping of Pegasus features to the taxonomy.

Table 2.2: Mapping Pegasus features to the taxonomy

Quality/Feature	Pegasus Type
Workflow structure	DAG-based
Workflow specification	Abstract workflow
Workflow execution control management	Centralized
Intermediate data movement	Mediated
Job code acquisition	Dynamic executable job
Mapping	Static / Dynamic (just in-time)

2.2.6 Mobile Code Collaboration Framework (MCCF)

Simulation studies show that a system exploiting Mobile Agents to control and manage execution of jobs can operate faster than other execution control management approaches in terms of job workflow execution time [45]. In MCCF, mobile agents execute job workflow in a distributed paradigm. Technologies like Lightweight Mobile Agent (LMA) and Code on Demand (CoD)³ are used in the construction of MCCF. This environment principally can help large-scale data-intensive computations over the Grid. In this framework, intermediate data is transfered in a P2P manner and jobs are provided as dynamic executable jobs. LMA is defined using Agent Core (AC) in the MCCF. AC is like a blueprint of job workflow which migrates between computational resources. When an AC arrives at a compute-node, it generates agents which execute the job. Job executable code is acquired by CoD technology and defined as a dynamic executable job [4].

Table 2.3 shows the mapping of MCCF features to the taxonomy.

Table 2.3: Mapping MCCF features to the taxonomy

Quality/Feature	MCCF Type
Supported workflow structure	DAG-based
Supported workflow specification	Abstract workflow
Workflow execution control management	Distributed (Mobile agents)
Intermediate data movement	P2P
Job code acquisition	Dynamic executable job
Mapping	Dynamic

2.2.7 ServiceGlobe

ServiceGlobe is a lightweight infrastructure acting as a distributed and extensible service platform. ServiceGlobe provides dynamic service selection and invocation at runtime. Ser-

³In distributed computing, Code on Demand is a general term for any technology that sends executable software programs from a server computer to a client computer upon request from the client's software (e.g., browser).

2.2 Existing Systems Overview

37

vice selection is performed according to the technical specification of the desired service using UDDI. A generic, modular dispatcher service provides load balancing and high availability of services. This dispatcher replicates new services on idle hosts.

Services are categorized as external services and internal services in this system. The internal services are those services which are implemented in Java using the service API provided by this system and external services are services that are already deployed on the Internet and thus do not use the ServiceGlobe service API. Internal services are classified by two orthogonal attributes: mobility and composition level. In terms of mobility, internal services can be either static (location dependent) or dynamic (location independent). As described in [5], in ServiceGlobe, dynamic services are stateless and do not require special resources or permission. In addition, in terms of composition level, services are classified into three groups: adapters, simple services (basic services) and composite services. Adapters are used for accessing external services inside ServiceGlobe and composite services are those services which are composed of basic ones [5–7].

Runtime loading allows distribution and replication of dynamic services on arbitrary hosts in ServiceGlobe. Although this feature opens up a great optimization potential for ServiceGlobe in terms of load balancing, high availability and parallelism, there are a number of important issues in this system that are not addressed: ServiceGlobe supports composite services but it does not support workflows (this system only supports sequential invocation of services). Composing workflows out of Web Services using a standard XML based language like WS-BPEL can ease Grid application development and service reusability. The internal services which can be loaded at runtime must be implemented in Java using the ServiceGlobe API which confines the set of available services. In addition, in ServiceGlobe, dynamic services are stateless which is not desirable for most Grid applications. In this system, Web Services are loaded on any arbitrary host according to the host's workload and without concern about input data location. ServiceGlobe is a

standalone project and does not rely on any Grid middleware infrastructure. Using standard Grid middleware infrastructures and standards eases secure and controlled resource sharing between virtual organizations. The ServiceGlobe platform project has not been active since 2003. Table 2.4 shows the mapping of Service Globe features to the taxonomy.

Table 2.4: Mapping Service Globe features to the taxonomy

Quality/Feature	Service Globe Type
Workflow structure	DAG-based (limited to sequenced DAG)
Workflow specification	Abstract workflow
Workflow execution control management	Centralized
Intermediate data movement	Centralized
Job code acquisition	Dynamic service (stateless, only available for internal services)
Mapping	Dynamic

2.2.8 Summary

In this section, we examined a number of Grid workflow management systems and mapped these systems to the theoretical taxonomies we gave in Section 2.1 when applicable. Almost all of the existing systems utilize conventional compiled executable jobs as the Grid workflow jobs and those systems which provide WS-workflow execution are not based on standards and globally adopted specifications. Most of the Grid workflow management systems are for high-throughput computing over the Grid and usually are not optimized for data-intensive workflows.

Because Service Globe (see Section 2.2.7) is the most similar workflow management system to the proposed system, we compare these two systems in detail to show the advantages of the proposed system. Table 2.5 shows this comparison.

Based on the advantages and disadvantages of the above mentioned systems, a gap analysis (see Table 2.6) has been performed and a dynamic Web Service scheduling and deployment framework is proposed. The proposed framework is based on standard-based

2.2 Existing Systems Overview

39

existing systems, and conforms with the ongoing convergence of Grid and Web technologies.

Table 2.5: Detailed comparison between Service Globe and the proposed system

Quality/Feature	Service Globe	The proposed system
Support standard workflows	No (sequenced only)	Yes (WS-BPEL)
Support stateful dynamic WS	No (stateless)	Yes
Based on OGSA specifications	No	Yes (using GT4)
Based on WSRF specifications	No	Yes
Load balancing	Yes	Yes
Data location awareness	No	Yes

Table 2.6: Gap analysis of examined systems

Project		Service Globe	DAGMan	Pegasus	MCCF	The Proposed System
Execution Control Management	Centralized	✓	✓	✓		✓
	Hierarchical					
	Collaborative Engines					
	Mobile Agents				✓	
Intermediate Data Movement	User Directed		✓			
	Centralized	✓				
	Mediated			✓		✓
	P2P				✓	
Job Code Acquisition	Static Executable Job					
	Dynamic Executable Job		✓	✓	✓	
	Static Service					
	Dynamic Service	✓				✓
Mapping	Static Mapping			✓		
	Dynamic Mapping	✓	✓	✓	✓	✓

Chapter 3

Framework Design

Based on the literature review described in Chapter 2, the study performed on common Grid workflow management systems discussed in Section 2.2, and the gap analysis (see Table 2.6) we introduce a dynamic Web Service scheduling and deployment framework in this chapter. The framework is designed in line with the onward convergence of Web and Grid technologies. The framework which is further referred to as *Dynamic Web Service Scheduling and Deployment Framework (DynaSched)* is based on the latest standards and technologies such as: SOA, WSRF and OGSA.

This chapter begins with an overview of the framework, followed by a detailed description of the different framework components and their interactions. This includes the components of the supporting components layer. Thereafter we describe the scheduling layer which is located on a higher level and uses the components of the supporting components layer to schedule dynamic Web Service deployments for later invocations. We conclude this chapter with the description of the WS-Workflow orchestration layer and a summary.

3.1 Overview

DynaSched is a modular framework which facilitates dynamic Web Service deployment on Grid resources. This framework consists of various components. Most of the framework components described in this chapter are Web Services themselves and are implemented in Java (i.e., they are Java Web Services developed using the GT4's Java WS Core API). Therefore, these components reside in GT4's WS containers. The framework also uses some of the GT4 components such as GridFTP, WS-MDS, GSI Java, and Java WS Core. GSI authentication and authorization are used to ensure that only authorized parties can invoke the operations provided by the framework components.

To simplify understanding of the framework design, we divide the framework into three layers. The bottom layer, which is further referred to as the supporting components layer is a set of components on which the functionality of the Web Service scheduler relies. On the two higher levels, the Web Service scheduler and the WS-Workflow orchestration engine are located respectively. Figure 3.1 illustrates the DynaSched architecture and its components. The following list summarizes the layers and the components in each layer from a bottom-up view:

1. Supporting Components Layer

- Dynamic Web Service Factory (WS-Factory)
- Web Service Code Repository (WS-CodeR)
- Resource Information System (RIS)
 - GT4's Information Services
 - Ganglia
 - SysMon
 - Speedo (+enhanced ttcp or WANMod)
- GT4's Data Management Services (GridFTP)

2. Scheduling Layer

- Web Service Scheduler (WS-Scheduler)
- Framework Database / UDDI

3. WS-Workflow Orchestration Layer

- WS-Workflow Orchestration Engine (WS-Orc)

Some of the components in the supporting components layer are provided by GT4 or other projects while some others are developed to realize the functionality of the framework. Two WS components, further referred to as WS-Factory and WS-CodeR are developed and integrated with other components provided by GT4. WS-Factory—which is a reimplementation of GT4's `DeployService`—deploys a Web Service dynamically and returns the invocable URI of that Web Service (see Section 3.2.1 for more details). WS-CodeR is a repository for storing undeployed Web Services (see Section 3.2.2 for more details). These components interoperate with other GT4 components to support CoD technology in a Web Service enabled Grid. In addition, the components in the RIS provide the latest information about the available resources such as the current workload of every resource and the available bandwidth between them. Some of the components in the RIS, which are SysMon, Speedo and WANMod are developed in this project (see Section 3.2.3 for more details).

In the scheduling layer, a Web Service scheduler, further referred to as WS-Scheduler, is developed to interact with the WS components in the supporting components layer. The WS-Scheduler uses the information provided by the RIS to decide which resource is a better option to use for dynamic Web Service deployment in order to increase resource utilization and decrease Web Service execution time (see Section 3.3.1 for more detail). The scheduler stores the information about the dynamically deployed Web Services together with the URI of WS-Factories and WS-CodeRs in a database.

In the WS-Workflow orchestration layer, the WS-Orc uses the WS-Scheduler to convert

Figure 3.1: DynaSched architecture

3.2 Supporting Components Layer

45

an abstract Web Service workflow to an executable Web Service workflow by converting every abstract Web Service in the workflow to an invocable Web Service (see Section 3.4.1 for more detail).

Different components are executed on different type of nodes. There are three types of nodes assumed:

- **User machine:** The machine to which the user has direct access. The WS-Orc is usually running on this machine.
- **Head-Node:** A node which is accessible by the user and primarily responsible for receiving and scheduling user jobs on compute nodes. This node is usually the gateway to the resources on which the Web Services are being executed.
- **Compute-Node:** Any node which is not typically accessible directly by the user. Compute-nodes provide the computing power for the Web Services to be executed and are specifically considered as compute-nodes.

Figure 3.2 illustrates a typical topology of the network, underlying resources and location of the DynaSched components on these resources.

In the following sections we describe the functionality of the above mentioned components in detail and explain the interactions between them. We follow a bottom-up approach in describing the layers of the framework.

3.2 Supporting Components Layer

As described in Section 3.1 the supporting components layer consists of a number of components which provide functionality that is required by the framework's main component (i.e., the WS-Scheduler). In this section, we start by describing the WS-Factory which is a key component to enabling dynamic Web Service deployment. Then, we continue by describing the WS-CodeR, a repository to store undeployed Web Services. We conclude

Figure 3.2: Typical topology of the network, underlying resources and location of the framework components

this section with a description of the RIS which is essential to the WS-Scheduler decision making mechanism. GT4's data management components (e.g., GridFTP or RFT) which are used by the framework to transfer files between nodes are previously described in Section 2.2.1. The current implementation of the framework relies on GridFTP to transfer files between nodes.

3.2.1 Dynamic Web Service Factory (WS-Factory)

A WS-Factory is a Web Service itself and is deployed on each compute-node's WS container (it is assumed that there is only one WS container running on every node). WS-Factory deploys Web Services and undeploys them when they are not needed anymore. Qi et al. [8] proposed, (partially) implemented and evaluated the Highly Available dyNamic Deployment infrastructure (HAND) based on the Java Web Services core of GT4. This service (which is called `DeployService`) is shipped with the Globus Toolkit Version 4.2. We re-implemented the `DeployService` to add further features to it¹. WS-Factory is this reimplementation. First we look into details of `DeployService` and then we identify the features which are added to it by WS-Factory.

Qi et al. [8] explored two different approaches for dynamic Web Service deployment:

- **Container-level deployment** in which dynamic deployment of new services requires the reloading of the whole WS container. During the dynamic deployment and the WS container reloading process, all other services hosted by the WS container are unavailable.
- **Service-level deployment** in which no existing services are deactivated before deploying new services, assuming that the existing services are not updated. In the case of update, the corresponding services are deactivated and re-activated after installation of the new version of services. This approach does not require the reloading of the whole WS container. All other services are unaffected and available in this approach.

In the container-level dynamic deployment the following sequence of actions are performed upon receiving a deployment request [8]:

1. The WS container is put into reloading mode. The WS container remains in this

¹One of the additional features is described on page 50 under the description of the `deploy()` operation, and the other is described in Section 4.1.2

- state until all currently running services terminate. While the WS container is in this state it returns “Service Unavailable” to any service invocation request.
2. Stop and deactivate all services, resource homes and so forth.
 3. Perform cleanup.
 4. Execute the deployment or undeployment scripts.
 5. Reload the WS container.
 6. Return the WS container to the normal operating mode and start accepting new requests.

Figure 3.3 illustrates the container-level deployment sequence.

Figure 3.3: Container-level deployment sequence (inspired from [8])

Unlike container-level deployment, when using service-level dynamic deployment, the WS container is unaffected. The following actions are performed to deploy a service dynamically [8]:

1. Wait for termination of the services that are to be updated if they are already running. During this period the WS container returns “Service Unavailable” only to invocation requests for those services.

3.2 Supporting Components Layer

- 2. Stop the services that are to be updated and deactivate any related resources.
- 3. Perform cleanup.
- 4. Execute the deployment or undeployment scripts.
- 5. Create or update the working space context for the new service.
- 6. Initialize, activate and start the new services.

Figure 3.4 illustrates the service-level deployment sequence in the case of deploying a new service (i.e., it is not an update).

Figure 3.4: Service-level deployment sequence (inspired from [8])

The study by Qi et al. showed that container-level deployment works well when a global upgrade or configuration is needed, while service-level deployment is more flexible, capable and available. In addition the capability of dynamic deployment at the container-level is unpredictable in a complicated dynamic Grid environment [8]. The rest of this section describes the resource properties and operations provided by this component.

Resource Properties

The WS-ResourceProperties published by WS-Factory are as follows:

- **Undeployable:** List of Web Service identifiers that can be undeployed.
- **Deployable:** List of GAR file identifiers that have been transferred to the WS-Factory but not yet deployed.

Operations

WS-Factory provides five operations. One can invoke either of the `download()` or `upload()` operations to transfer an undeployed Web Service in GAR² format into the WS container. The other operations provided are for deploying or undeploying Web Services or reloading the WS container. Where not mentioned, the request or response message should be considered as Null. The operations provided, and their corresponding request and response messages are as follows:

- **upload():** By using this operation the GAR file is transferred to the WS-Factory as an attachment to the SOAP request message.
 - Request message: A String which specifies the name of the GAR file.
 - Response message: A String which identifies the transferred GAR file. This GAR file identifier is used to deploy the GAR file later by invoking the `deploy()` operation.
- **download():** WS-Factory downloads the file from a given URI, using GridFTP.
 - Request message: A String pointing to the GAR file location (GAR file URI).
 - Response message: A String which identifies the transferred GAR file. This GAR file identifier is used to deploy the GAR file later by invoking the `deploy()` operation.
- **deploy():** This operation deploys a GAR file onto the WS container and returns the URI of the dynamically deployed Web Service(s). The `deploy()` method of GT4's

²A GAR file consists of a Service Implementation, Service Interface (WSDL file) and Deployment Descriptor (WSDD file).

`DeployService` does not return the URI of dynamically deployed Web Service(s), but the method in `WS-Factory` does. This functionality is required by the framework because the URI of the dynamically deployed Web Service is used later to invoke its operations.

- Request message: A String which contains the transferred GAR file identifier together with deploy options.
- Response message: An array of Strings with the URI of dynamically deployed Web Service(s).
- `undeploy()`: This operation undeploys a deployed Web Service from the WS container.
 - Request message: A String which identifies the Web Service to undeploy.
- `reload()`: A client can reload the WS container by invoking this operation.

3.2.2 Web Service Code Repository (WS-CodeR)

WS-CodeR stores the undeployed Web Services in GAR format. Undeployed Web Services are transferred to and registered in the code repository by invoking corresponding WS-CodeR operations. The WS-CodeR supports access control mechanisms, i.e., every registered GAR file has a specific owner, group and access attributes (the access control mechanism of the WS-CodeR can be improved to use the GT4's GSI Java API). The WS-CodeR publishes the list of registered GAR files as a resource property. Therefore, in addition to the `search()` operation (described on page 53), any other service or Web application can search in it using the *QueryResourceProperties* interface (denoted as `QueryRP()` in figures).

Resource Properties

The WS-ResourceProperties published by WS-CodeR are as follows:

- **GARList**: List of registered GAR file identifiers which are stored in the WS-CodeR.

Operations

The operations provided, and their corresponding request and/or response messages are described in this section. The GAR files can be transferred to the code repository using any of the following options:

- **upload()**: Transfers the GAR file to the WS-CodeR as an attachment to the SOAP request message.
 - Request message: A String which specifies the name of the GAR file.
 - Response message: A String which identifies the transferred GAR file. This GAR file identifier is used to register the GAR file later by invoking the **register()** operation.
- **download()**: The WS-CodeR downloads the file from a given URI, using GridFTP.
 - Request message: A String pointing to the location of the GAR file (GAR file's URI).
 - Response message: A String which identifies the transferred GAR file. This GAR file identifier is used to register the GAR file later by invoking the **register()** operation.

The following operations are also provided to add or remove a GAR file to/from the WS-CodeR:

- **register()**: To register a transferred GAR file in the WS-CodeR and define its access attributes.
 - Request message: An identifier for the transferred GAR file to be registered in the WS-CodeR plus register options data structure. The file's owner, group and access attributes are specified in the register options data structure.
- **deregister()**: To deregister and delete a GAR file from the WS-CodeR.

- Request message: A String which identifies the GAR file to be deregistered.
- **search()**: to search for a Web Service by providing the GAR file identifier.
 - Request message: A string containing the GAR file identifier.
 - Response message: The URI by which the GAR file can be transferred to a WS-Factory.

3.2.3 Resource Information System (RIS)

It is essential for the WS-Scheduler to have up-to-date information about the available resources. This information is provided by the RIS components. Information about the underlying Grid resources is collected by the MDS information services. We describe the GT4's information services and the information providers which are used in RIS here.

GT4 Information Services

WS-MDS is a WSRF-compliant suite of Web Services to monitor and discover resources and services on Grids. The services which are used by RIS are:

- **Index service**: Collects monitoring and discovery information from Grid resources, and publishes it in a single location.
- **Aggregator framework**: Collects data from an aggregator source and sends that data to an aggregator sink for processing. Aggregator sources include Execution source. Aggregator sinks include modules that implement the Index interfaces (e.g., GT4's `DefaultIndexService`).
- **Execution source**: Any program which generates an XML file can be configured as an execution source.

To summarize, the execution sources generate XML files which contain the information about the hosts on which they are located. These XML files are parsed by the Aggregator

framework and published into the local Index service. The information published on the local Index services of each host are aggregated and published on the head-node's Index service. The WS-Scheduler searches the information published by the Index service using the *QueryResourceProperties* interface.

Ganglia

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It is based on a hierarchical design targeted at federations of clusters [46]. It includes two components:

- **Ganglia Monitoring Daemon (gmond):** Gmond is a multi-threaded daemon which runs on each compute-node which monitors changes in host state. It provides a number of static metrics (e.g., number of CPUs, CPU speed, total memory) and variable ones (e.g., workload, memory usage).
- **Ganglia Meta Daemon (gmetad):** Gmetad periodically polls a collection of child data sources, parses the collected XMLs, saves all the included information and exports the aggregated XML to clients.

The information provided by Ganglia is aggregated in the head-node and published on its Index service periodically.

SysMon

SysMon (derived from System Monitor) is implemented as a scalable distributed monitoring system for Globus Grids. SysMon provides detailed information about CPU and memory load for every compute-node. The information provided by SysMon is published into the local Index service periodically and includes:

- Number of CPU cores, user/system/idle percentage for every CPU core.
- Used and available memory size.

This component is a simplified alternative for Ganglia.

Speedo

Speedo is implemented to provide a network connection bandwidth map in the form of a complete directed graph. It measures upstream and downstream bandwidth by using enhanced TTCP [47]. The information is generated in every compute-node and published into the local Index service periodically. The information includes:

- Host name and IP address of local host.
- List of the remote hosts together with available bandwidths to and from them.

To perform experiments in an emulated WAN environment, we developed a WAN model which replaces enhanced TTCP. We describe this model in Section 4.1.3.

3.3 Scheduling Layer

The scheduling layer contains two components: the WS-Scheduler and the framework database. The WS-Scheduler is the most important component of the framework. It uses the information provided by the RIS to schedule a dynamic Web Service deployment on an available resource. It manages the WS components in the supporting components layer to perform the required corresponding actions. This section describes components of the scheduling layer in detail. The framework database is explained in Section 3.3.2.

3.3.1 Web Service Scheduler (WS-Scheduler)

A scheduler decides where and when to run a job. Dynamic Web Service deployment allows distribution, replication and relocation of Web Services to enhance productivity by increasing resource utilization and decreasing execution time. This goal can be realized

by (i) deploying compute-intensive Web Services on lesser loaded compute-nodes; (ii) deploying data-intensive Web Services on or close to compute-nodes where the input/output data is located. The former provides more computing power while the latter reduces communication time.

The WS-Scheduler decides which resource is a better option to use for a Web Service job. It uses a scheduling approach based on the characteristics of the Web Service to select an available resource on which the Web Service can be deployed and invoked. Compute-intensive Web Services need much computing power while data-intensive Web Services have large file(s) as their input. After selecting the resource, if the requested Web Service is already deployed on that resource, the URI of the Web Service on that resource is returned by the WS-Scheduler. Otherwise if the requested Web Service is not deployed on the selected resource, the undeployed Web Service is transferred from the WS-CodeR to the WS-Factory and deployed dynamically on the selected resource, then the URI of the dynamically deployed Web Service is returned by the WS-Scheduler. We have implemented two scheduling schemes to select an available resource:

- **Workload balancing scheme:** This scheme is suitable for compute-intensive Web Services. The WS-Scheduler tries to balance the workload of compute-nodes by choosing the resource with the least workload. By balancing the workload, we try to increase resource utilization, which should normally result in a decrease in Web Service execution time. However as dynamic Web Service deployment is a time consuming process (especially container-level dynamic deployment, because this approach requires all running services to terminate) there is a trade-off between choosing a resource with the least workload and choosing a resource with slightly higher workload on which the Web Service is already deployed. Therefore we can categorize this trade-off as a workload to waiting time trade-off.
- **Input data location aware scheme:** This approach is suitable for data-intensive

Web Services. In this scheme, the WS-Scheduler selects a compute-node which is the same as or close to the resource where the input data is located. For data-intensive Web Services the execution time is trivial in comparison to the communication time (i.e., the time required to transfer the input file to the local host). Therefore the WS-Scheduler estimates the communication time and the time required for dynamic deployment and selects the resource with the least estimated time cost.

Currently it is assumed that the user knows whether a service is compute or data intensive and specifies its type in the abstract WS-Workflow description. The WS-Scheduler is a modular and customizable component. New scheduling approaches can be implemented and plugged into the WS-Scheduler easily. The remainder of this subsection describes the scheduling algorithms used by these schemes in more detail.

Scheduling Algorithms

We have implemented two scheduling algorithms for the decision making mechanism of the WS-Scheduler:

Load Balancing Algorithm: In the load balancing algorithm, first we create a list of all available WS-Factories together with the workload of the host on which the WS-Factory is located. In Algorithm 1, l_i denotes the workload of the host on which the WS-Factory _{i} is located. Then we sort the list in an ascending order and store the top $p\%$ of the list in a new list further referred to as L'' . The value for the parameter p is determined by performing experiments. We describe the process of identifying this value later in Section 4.2.1. In the next step we process L'' in order and examine if the requested Web Service is already deployed on any of the WS-Factories in the list. If we find the deployed Web Service, the URI of that Web Service is returned and the algorithm terminates. Otherwise a new list of q_i values is generated where $l_i \in L''$ and $q_i = r_i + d_i$. Here r_i denotes the number of services

currently running in the WS-Factory_{*i*}'s WS container and d_i denotes the deploy queue size for WS-Factory_{*i*}. Finally we select the WS-Factory with the minimum value of q , transfer the undeployed Web Service to that resource, invoke the `deploy()` operation and return the resulting URI. Algorithm 1 shows a simplified pseudocode of this algorithm.

Algorithm 1: Load balancing algorithm

Input: requested WS identifier
Output: requested WS URI

```

1  $L = [l_1, l_2, \dots, l_i, \dots, l_k]$  ;          /* where  $l_i$  is the workload of WS-Factoryi */
2  $L' = \text{sort}(L, \text{asc})$ ;
3  $L'' = \text{top}(L', p)$  ;                          /* takes first  $p\%$  of the list */
4 initialize  $Q = []$ ;
5 for ( $i = 0$ ;  $i < |L''|$ ;  $i++$ ) do
6   if WS is already deployed on WS-Factoryi then
7     return  $URI_i$  ;                          /* where  $URI_i$  is the URI of the */
                                              /* requested WS on WS-Factoryi */
8   else
9      $q_i = r_i + d_i$  ;          /* where  $r_i$  is the number of running services */
                                /* and  $d_i$  is the deploy queue size for WS-Factoryi */
10    add  $q_i$  to  $Q$ ;
/* if algorithm reaches here it means that the WS has not been
   deployed on any of the WS-Factories processed in the for loop */
11  $q_j = \min\{q \in Q\}$  ;          /* which identifies the WS-Factoryj */
12  $URI_j = \text{deploy}(\text{WS}, \text{WS-Factory}_j)$ ;
13 return  $URI_j$ ;
```

Input Data Location Aware Algorithm: The input data location aware algorithm estimates the required time to execute a data-intensive Web Service dynamically for all available WS-Factories and selects the resource with the least estimated time cost. The estimated time cost for executing a dynamic Web Service on WS-Factory_{*i*} is denoted as t_i . Parameter t_i consists of a dynamic deployment time cost t_{id} , input data transmission time cost t_{it} , and Web Service execution time t_{ie} . Because of the characteristic of data-intensive Web Services, $t_{ie} \ll t_{it}$. Moreover, estimating t_{ie} is not easy. Therefore, t_{ie} is

not considered in the time cost estimation formula. Typically for a given environment, t_{id} is roughly a constant value which depends on the WS container configuration, underlying Grid resource specifications and undeployed Web Service size. In the algorithm we show this constant value by parameter C . The most dominating factor in the time cost estimation formula for data-intensive Web Services is t_{it} which is calculated by dividing the input data size, denoted as S_{in} , by the measured available bandwidth between the host on which the input data is located and the WS-Factory $_i$, denoted as BW_{in-i} . Note that if the input file is located on the same host as where the WS-Factory $_i$ is deployed then $t_{it} = 0$. Once the time cost is estimated for all available WS-Factories the one with the least time cost is selected. If the requested Web Service is already deployed on the selected resource the URI of that service is returned otherwise the undeployed Web Service is transferred to that resource, the `deploy()` operation is invoked and the resulted URI is returned. Algorithm 2 shows pseudocode of this algorithm.

Resource Properties

The WS-ResourceProperties published by WS-Scheduler are as follows:

- **WSCRList**: List of WS-CodeRs registered in the WS-Scheduler.
- **DWSFList**: List of WS-Factories registered in the WS-Scheduler.
- **DeployedServices**: List of dynamically deployed Web Services in the framework.
- **RequestQueues**: Status of the WS-Scheduler's deploy request queues.

Operations

The operations provided by the WS-Scheduler are as follows (operation's request or response message should be considered as Null where its description is not mentioned):

- **schedule()**: Selects an available resource for a requested Web Service based on the scheduling options and the information from RIS. If the selected resource has

Algorithm 2: Input data location aware algorithm**Input:** requested WS identifier, S_{in} =input data size, input data location**Output:** requested WS URI

```

1   $T = []$ ;
2  forall the  $WS\text{-}Factory_i$  do
3      if  $WS$  is not already deployed on  $WS\text{-}Factory_i$  then
4           $t_{id} = C$ 
5      else
6           $t_{id} = 0$ 
7           $t_{it} = \frac{S_{in}}{BW_{in-i}}$ ;
8           $t_i = t_{id} + t_{it}$ ;
9          add  $t_i$  to  $T$ ;
10  $t_j = \min\{t \in T\}$  ;                               /* which identifies the  $WS\text{-}Factory_j$  */
11 if  $WS$  is already deployed on  $WS\text{-}Factory_j$  then
12     return  $URI_j$  ;                                   /* where  $URI_j$  is the URI of the */
                                                         /* requested WS on  $WS\text{-}Factory_j$  */
13 else
14      $URI_j = \text{deploy}(WS, WS\text{-}Factory_j)$ ;
15     return  $URI_j$ ;

```

the requested Web Service already deployed, the URI of that resource is returned immediately. Otherwise it manages the dynamic Web Service deployment procedure and returns the URI of the dynamically deployed requested Web Service.

- Request message: A String which contains the requested Web Service identifier together with scheduling options data structure. In the scheduling options data structure, the preferred scheduling mode (i.e., load balancing or data location aware) is specified and required information (e.g., input data location and size if using data location aware scheduling mode) is provided. The requester credentials are also attached to this data structure to support security issues.
- Response message: A String which contains the URI of the requested Web Service.
- **cleanup()**: Performs a cleanup by sending undeploy requests for all of the dynami-

cally deployed Web Services to their corresponding WS-Factory.

- Response message: An array of Strings which contains the identifier of the undeployed Web Services.
- **registerWSCR()**: Registers a WS-CodeR's URI in the frameworks's database.
 - Request message: A String which contains the WS-CodeR's URI.
- **deregisterWSCR()**: Deregisters a WS-CodeR's URI from the framework's database.
 - Request message: A String which contains the WS-CodeR's URI.
- **registerDWSF()**: Registers a WS-Factory's URI in the framework's database.
 - Request message: A String which contains the WS-Factory's URI.
- **deregisterDWSF()**: Deregisters a WS-Factory's URI from the framework's database.
 - Request message: A String which contains the WS-Factory's URI.
- **registerSrvcExec()**: Registers a Web Service execution. This operation is used by the WS-Orc before invoking any Web Service job, to allow the WS-Scheduler to know the number of services running on every registered WS container at any given time. Having information about the number of services running on every container is essential for the load balancing scheduling algorithm. It is assumed that the WS-Scheduler has full control of all containers and is aware of every running service on every container.
 - Request message: A String which contains the URI of the Web Service which is going to be invoked.
- **deregisterSrvcExec()**: Deregisters a Web Service execution. This operation is used by the WS-Orc after receiving the result of a Web Service invocation. Using this operation allows the WS-Scheduler to know the number of services running on every registered WS container at any given time.
 - Request message: A String which contains the URI of the Web Service which

has just finished its execution.

3.3.2 Framework Database

The framework database stores the following information:

- **Deployed Web Services:** Any Web Service which is deployed dynamically by the framework is registered in the database.
- **WS-CodeRs:** The framework has one or multiple WS-CodeRs. Every WS-CodeR is identified by an ID which is stored together with its URI in the database. The WS-CodeR's ID is a combination of the IP address of the host and the port number of the WS container on which the WS-CodeR is deployed.
- **WS-Factories:** The framework has one or multiple WS-Factories. Every WS-Factory is identified by an ID which is stored together with its URI in the database. The WS-Factory's ID is a combination of the IP address of the host and the port number of the WS container on which the WS-Factory is deployed. The framework supports multiple WS-Factories on multiple WS containers on a single compute-node.

The WS-Scheduler implements an interface to the database and the information stored in it is published as resource properties. Therefore any other service or Web application can query this information using the *QueryResourceProperties* interface.

3.4 WS-Workflow Orchestration Layer

This layer consists of one component which is described in the following section. Unlike other components of the framework (except WS-CodeR), the user interacts with this component directly.

3.4.1 WS-Workflow Orchestration Engine (WS-Orc)

To use the framework, a WS-Workflow orchestration engine is required which can convert an abstract WS-Workflow to a concrete one. The engine needs to interact with the scheduler during the conversion process. Currently there is no commonly agreed standard for describing abstract WS-Workflows. In order to test the underlying framework which is the main focus of this thesis, we have developed a customized workflow engine with only essential features (i.e., converting abstract Web Services of the WS-Workflow to concrete Web Services using the WS-Scheduler, and orchestrating the Web Services). Currently it provides a command-line based interface to the user.

The input of the WS-Orc is an abstract WS-Workflow where each Web Service is described by a Web Service identifier. WS-Orc manages the control flow and data flow for workflow execution. When it converts the abstract WS-Workflow to an executable one, it binds an abstract Web Service to a new or existing deployed Web Service by sending a scheduling request to the WS-Scheduler's `schedule()` operation. If a `schedule()` request times out, the WS-Orc retries by sending a new request. The `schedule()` operation returns a Web Service URI. The URI points to the location where the requested Web Service is deployed. WS-Orc can now invoke the requested Web Service after registering service execution with the WS-Scheduler. It deregisters the service execution after the Web Service terminates.

3.5 Summary

We described the DynaSched's design in this chapter. We divided the framework into three layers and specified the responsibilities of each layer together with the operations provided by the WS components. We followed a bottom-up approach in describing the framework to illustrate the components' interactions with each other as we moved up the layers in the

system architecture.

A sequence diagram which illustrates how the DynaSched’s components operate with one another concludes this chapter. Figure 3.5 illustrates a typical dynamic Web Service scheduling and deployment process in DynaSched. In this figure WS-ID is the Web Service identifier and GAR-ID is the GAR file identifier. For simplicity the WS-ID and the GAR-ID are the same in our implementation.

The framework is based on OASIS and GGF standards and specifications. It conforms with the SOA, OGSA and WSRF specifications and relies upon standard Grid middleware: the Globus Toolkit. Table 3.1 shows the mapping of DynaSched features to the taxonomy.

Table 3.1: Mapping DynaSched features to the taxonomy

Quality/Feature	DynaSched Type
Workflow structure	non-DAG-based
Workflow specification	Abstract workflow
Workflow execution control management	Centralized
Intermediate data movement	Mediated
Sub-job code acquisition	Dynamic service
Mapping	Dynamic (just in-time)

Figure 3.5: Dynamic Web Service scheduling and deployment sequence diagram

Chapter 4

Empirical Studies

We implemented the dynamic Web Service scheduling and deployment framework designed in Chapter 3. In the current chapter we present empirical studies which show that DynaSched can increase resource utilization and decrease WS-Workflow execution time. As described in Section 3.3.1, WS-Scheduler provides two scheduling algorithms: (i) a load balancing algorithm which is suitable for compute-intensive services and (ii) an input data location aware algorithm which is suitable for data-intensive services. We performed two series of experiments for: (i) compute-intensive WS-Workflows and (ii) data-intensive WS-Workflows. In each series the time required for the WS-Workflow execution is measured when using static or dynamic services. We explain the details of the experiments performed for compute-intensive and data-intensive workflows in Sections 4.2 and 4.3, respectively.

This chapter starts with an introduction section which explains the experimental environment, the service-level dynamic deployment emulator, and the WAN model. Then we explain the compute-intensive WS-Workflow structure and present the experiments performed to identify the optimal value for the parameter p introduced in Algorithm 1 for both service-level and container-level dynamic deployment. After identifying the parameter p , we present the WS-Workflow execution time measurements for static services,

container-level dynamic services, and service-level dynamic services. Finally we explain the data-intensive WS-Workflow structure and present the WS-Workflow execution time measurements for static services, container-level dynamic services, and service-level dynamic services. We discuss the experimental results after presenting them in each section. A summary concludes this chapter.

4.1 Introduction

A comprehensive evaluation of the effectiveness of the framework for WS-Workflows is challenging because of the difficulty of capturing the complexities of a realistic Grid environment and a variety of WS-Workflows. Therefore, we focus on specific experiments designed to show the effectiveness of the WS-Scheduler (and therefore the framework) for two types of WS-Workflows. We measure and compare the time required to execute WS-Workflows when using static services, container-level dynamic services, and service-level dynamic services. In the case of using static services (i.e., executing a concrete WS-Workflow), the WS-Orc invokes each Web Service directly. On the other hand, in the case of using dynamic services (i.e., executing an abstract WS-Workflow), the WS-Orc first converts each abstract Web Service to a concrete one by sending a `schedule()` request to the WS-Scheduler and then uses the corresponding URI returned by the WS-Scheduler to invoke the Web Service. There are two approaches of dynamic Web Service deployment: container-level and service-level. The services which are dynamically deployed using the container-level approach are referred to as container-level dynamic services and the services which are dynamically deployed using the service-level approach are referred to as service-level dynamic services.

4.1.1 Experimental Environment

Our testbed consists of eleven nodes connected by Myrinet. Each server is powered by 3 GHz Intel Xeon dual-core processors and 1 GB memory running Linux 2.4.21 and JDK 1.5.0. We use the Globus Toolkit version 4.1.2. The GT4's WS container is executed on each node and hosts all of the default services shipped with the toolkit. In addition to the default services the DynaSched's WS components, WS-Scheduler and WS-CodeR are hosted on the head-node's WS container and WS-Factories are hosted on the compute-nodes' WS containers. Figure 3.2 illustrates the network topology and location of various components of the framework. Throughout the experiments we have blocked access to these nodes by other users in order to have a dedicated environment.

4.1.2 Service-Level Dynamic Deployment Emulator

At the time of writing, service-level dynamic deployment is not supported by GT4. Implementing this approach is complicated and requires low-level changes to the WS container. In order to evaluate the performance of our scheduler with various underlying dynamic deployment mechanisms, which is the focus of this thesis, we emulate this approach.

The total dynamic deployment time consists of four components:

$$t_{total} = t_{transfer} + t_{pending} + t_{deploy} + t_{reload}$$

Where t_{deploy} is the time it takes to execute the scripts that implement deployment actions and $t_{transfer}$ is the time it takes to transfer the undeployed Web Services. Both of these components are independent of the deployment procedure (i.e., service-level or container-level) [8]. According to Section 3.2.1, the $t_{pending}$ and t_{reload} components are zero for service-level dynamic deployment in the case of deploying a new service (i.e., it is not an

update)¹.

In order to determine t_{deploy} , we took measurements from 100 dynamic deployments. As illustrated in Figure 4.1, measured time cost has a normal distribution. We calculated mean and standard deviation of the measured time cost to derive a normal distribution function in order to generate synthetic values for t_{deploy} in the emulator. In the service-level dynamic deployment emulation process, the synthetic Web Services used for experiments are deployed on all nodes but they are hidden. When the WS-Scheduler orchestrates a service-level dynamic deployment on a compute-node, the undeployed Web Service is transferred to the target compute-node and a `deploy()` request is sent to its corresponding WS-Factory. The WS-Factory, instead of actually deploying the requested Web Service, generates a random value for the t_{deploy} component according to a normal distribution with the above mentioned mean and standard deviation. The WS-Factory emulates the service-level dynamic deployment by putting a delay period equal to the generated random value for the t_{deploy} component. It changes the status of corresponding Web Service to visible and returns its URI after the delay period.

4.1.3 WAN Model

Data-intensive WS-Workflows are composed of data-intensive Web Services. The input data of data-intensive Web Services are typically distributed geographically on the Grid. In order to provide a realistic Grid environment for data-intensive WS-Workflow execution time measurements, we need a WAN emulator. We collected the data provided by S^3 [48], a scalable sensing service for large networked systems, to build a realistic WAN model based on real data. S^3 provides a snapshot of all-pair capacity and available bandwidth metrics updated about every 4 hours on PlanetLab. PlanetLab is a global research network

¹ $t_{reload} = 0$ because WS container reload is not required in the service-level dynamic deployment approach (see page 48). $t_{pending} = 0$ because there is no pending time for service-level dynamic deployment (compare Figures 3.3 and 3.4).

Figure 4.1: Normal distribution of measured dynamic deployment time cost

that supports the development of new network services [49].

We use two terms to describe our WAN model: *connection* and *path*. Every connection is identified by its source node, destination node, and timestamp. Every path is identified by its source node and destination node. We collected available bandwidth for 6,090,278 connections during 14 days. Mean and standard deviation for 208,242 unique paths were calculated. After removing outliers, from the 151,921 remaining paths, 52,669 bidirectional paths were identified from which 37,672 of them were paths between 461 nodes located in different countries. A fully connected path graph with 10 nodes as the vertices and bidirectional paths as edges is extracted from the bidirectional paths model. Every bidirectional path in the model has two pairs of information: mean and standard deviation of the available bandwidth between its source to destination and vice versa. Figure 4.2 depicts the generated WAN model on a map².

²Available online: <http://pdcc.ntu.edu.sg/~shahand/wanmodelmap>

Figure 4.2: Generated WAN model

The WAN model is used throughout the experiments performed to measure data-intensive WS-Workflow execution time. The WAN model replaces the enhanced TTCP component and provides Speedo with the emulated available bandwidth between nodes. The WAN model is also used by the synthetic data-intensive Web Service to emulate the time required for input data transmission. Every time a component (i.e., Speedo or synthetic data-intensive Web Services) inquires the WAN model about the available bandwidth between two specific nodes, the WAN model generates a normally distributed random number according to the corresponding mean and standard deviation for that specific path.

4.2 Compute Intensive Workflow

A compute-intensive WS-Workflow is a workflow composed of compute-intensive services. We implemented 10 synthetic compute-intensive Web Services. Table 4.1 shows the name

of these services and the amount of time required by each service to execute on a dedicated WS container.

Table 4.1: Synthetic compute-intensive services used in our experiments

Service ID	Execution time (s)	Static service location
CIS-1	90	Compute-node 0
CIS-2	180	Compute-node 1
CIS-3	270	Compute-node 2
CIS-4	360	Compute-node 3
CIS-5	450	Compute-node 4
CIS-6	540	Compute-node 5
CIS-7	630	Compute-node 6
CIS-8	720	Compute-node 7
CIS-9	810	Compute-node 8
CIS-10	900	Compute-node 9

As illustrated in Figure 4.3, each compute-intensive WS-Workflow is composed of n batches and each batch is composed of m (here $m = 10$) compute-intensive services. In the current section, wherever we refer to a compute-intensive WS-Workflow by the size x , we mean $m \times n = x$. There are no dependencies assumed between services of the WS-Workflow. That means the workflow is actually a batch workflow with x number of compute-intensive services located at the same level. The WS-Orc creates threads to execute services periodically ($t = 60$ seconds). For static services, the WS-Orc invokes the corresponding compute-intensive service directly. The location of static services is shown in Table 4.1. For dynamic services, no compute-intensive service is initially deployed. The WS-Orc sends a `schedule()` request to the WS-Scheduler, and the WS-Scheduler uses the load balancing algorithm to schedule dynamic deployments of compute-intensive services.

4.2.1 Determination of Parameters

In this section we identify the optimal value for the parameter p that we introduced in Algorithm 1 for service-level and container-level dynamic deployments. In order to identify

this parameter we measured the required time to execute a compute-intensive WS-Workflow with the size 100. We repeated our experiments 10 times to reach an acceptable percentage of standard error (which is shown in the figures by error-lines). We compare and discuss the experiment results in the following sections.

Service-level Dynamic Deployment

The average WS-Workflow execution time is measured for various values of p when using service-level dynamic deployment and illustrated in Figure 4.4. The load balancing scheduling algorithm performed generally better with smaller p values ($20 \leq p \leq 40$). In order to identify the best value for p , we also logged the number of running services on each compute-node at specific time steps (i.e., when a scheduling request is received by the WS-Scheduler). Figures 4.5 and 4.6 illustrate changes in the mean, minimum and maximum values and standard deviation of the number of running services for various values of p in a single set of experiments.

A lower value of the standard deviation of the number of running services implies a better balancing of workload among compute-nodes. Figure 4.7 illustrates the changes in the standard deviation for various values of p in the same set of experiments in one plot to give a better comparison.

In order to compare changes of standard deviations, we calculated maximum and mean of standard deviations for various values of p for all sets of experiments and took the average of those values over time. Figure 4.8 illustrates the average of maximum values, and average of mean values of standard deviations of the number of running services on compute-nodes for various values of p .

Container-level Dynamic Deployment

The average WS-Workflow execution time is measured for various values of p when using container-level dynamic deployment and illustrated in Figure 4.9. The load balancing scheduling algorithm performed generally better with larger p values ($70 \leq p \leq 90$). In order to identify the best value for p , we also logged the number of running services on each compute-node at specific time steps (i.e., when a scheduling request is received by the WS-Scheduler). Figures 4.10 and 4.11 illustrate changes in the mean, minimum and maximum values and standard deviation of the number of running services for various values of p in a single set of experiments.

Figure 4.12 illustrates the changes in the standard deviation for various values of p in the same set of experiments in one plot to give a better comparison.

In order to compare changes of standard deviations, we calculated maximum and mean of standard deviations for various values of p for all sets of experiments and took the average of those values over time. Figure 4.13 illustrates the average of maximum values, and average of mean values of standard deviations of the number of running services on compute-nodes for various values of p .

Discussion

Based on the empirical studies, we concluded that $p = 40$ and $p = 90$ are the optimal values for the parameter p in Algorithm 1 for service-level and container-level dynamic deployments, respectively. We introduced parameter p to apply a trade-off between choosing a node with the least workload or choosing a node with slightly higher workload which has the requested Web Service already deployed on it.

When using service-level dynamic deployment, the performance of the algorithm was not good for extreme values of p . Where $p = 10$, the WS-Scheduler always selected the node with the least workload for dynamic Web Service deployment, and because of the

dynamic deployment time cost, it caused a higher WS-Workflow execution time. On the other hand, where $p = 100$, the WS-Scheduler always used a previously deployed Web Service which overrode the workload balancing. The average WS-Workflow execution time was more stable for other values of p and performed better for smaller p values. We also logged the number of running services on each node to gain a better perspective of the performance of the algorithm. Measurements showed that both maximum and average of standard deviations of number of running services are minimum for $p = 40$. Therefore we concluded that $p = 40$ is the optimal value when using service-level dynamic deployment.

When using container-level dynamic deployment, the smaller values of p lead to decreased performance. In this deployment approach, the WS-Factory is unable to dynamically deploy a Web Service as long as a Web Service is running in the WS container. Therefore the time cost of dynamic deployment is higher than the alternative approach. Because of the greater time cost, larger values of p showed better performance when using container-level dynamic deployment. Experiments showed that the average WS-Workflow execution time was lower and the algorithm performed better for larger values of p . By comparing the average WS-Workflow execution time and maximum and average of standard deviations of number of running services for these values, we concluded that $p = 90$ is the optimal value when using container-level dynamic deployment.

4.2.2 Comparison of Static and Dynamic Deployment

We measured the time required to execute concrete and abstract compute-intensive WS-Workflows to show the effectiveness of the WS-Scheduler in executing compute-intensive WS-Workflows. The WS-Scheduler uses the load balancing scheduling algorithm for scheduling dynamic deployments of compute-intensive services. The load balancing algorithm uses the values identified for the parameter p in the previous section for the corresponding dynamic deployment approach. We executed WS-Workflows with various sizes from 10 to

100 and measured their execution time when using static services, container-level dynamic services, and service-level dynamic services. We repeated our experiments 5 times to reach an acceptable percentage of standard error. Figure 4.14 illustrates the experiment results with error lines.

4.2.3 Discussion

Empirical studies showed that for smaller compute-intensive WS-Workflows, dynamic services was as good as or slightly worse than static services. But for larger compute-intensive WS-Workflows, dynamic services performed better than static services and the WS-Scheduler (and therefore the framework) decreased the WS-Workflow execution time. It implies that the workload was distributed better among compute-nodes and the framework increased the resource utilization. In addition, studies showed that service-level dynamic deployment performed better than container-level dynamic deployment because the time cost of the former is less than the time cost of the latter.

Container-level dynamic deployment is less efficient because a WS-Factory is unable to perform dynamic Web Service deployments as long as there is a running service on that container. In addition during the pending period, the container rejects all service invocation requests. These restrictions cause unstable trends in the average WS-Workflow execution time. For example in Figure 4.14 for WS-Workflow size 60 the WS-Workflow execution time for container-level dynamic deployment is slightly higher than the execution time for static services.

Figure 4.3: Compute-intensive workflow

Figure 4.4: Average WS-Workflow execution time for various values of p using service-level dynamic deployment

Figure 4.5: Number of running services on every compute-node for various values of p in one set of experiments when using service-level dynamic deployment

Figure 4.6: Number of running services on every compute-node for various values of p in one set of experiments when using service-level dynamic deployment (Continued)

Figure 4.7: Standard deviations of number of running services for various values of p in one set of experiments when using service-level dynamic deployment

Figure 4.8: Average of maximum and average of mean of standard deviations of number of running services for various values of p for all sets of experiments when using service-level dynamic deployment

Figure 4.9: Average WS-Workflow execution time for various values of p using container-level dynamic deployment

Figure 4.10: Number of running services on every compute-node for various values of p in one set of experiments when using container-level dynamic deployment

Figure 4.11: Number of running services on every compute-node for various values of p in one set of experiments when using container-level dynamic deployment (Continued)

Figure 4.12: Standard deviations of number of running services for various values of p in one set of experiments when using container-level dynamic deployment

Figure 4.13: Average of maximum and average of mean of standard deviations of number of running services for various values of p for all sets of experiments when using container-level dynamic deployment

Figure 4.14: Average WS-Workflow execution time for various compute-intensive WS-Workflow sizes

4.3 Data Intensive Workflow

A data-intensive WS-Workflow is a workflow composed of data-intensive services. We implemented a synthetic data-intensive Web Service which accepts the URI of an input file, downloads the file and calculates its MD5 hash code. Table 4.2 shows sizes of the input files used in our experiments.

Table 4.2: Input files used in our experiments

File ID	File Size
0	8 KB
1	100 KB
2	500 KB
3	1 MB
4	5 MB
5	10 MB
6	50 MB
7	100 MB
8	500 MB
9	1 GB

As illustrated in Figure 4.15, each data-intensive WS-Workflow is composed of n batches and each batch has m invocations of the synthetic data-intensive service inside it (here $m = 10$). In the current section, wherever we refer to a data-intensive WS-Workflow by the size x , we mean $m \times n = x$. The WS-Orc creates a thread every $t = 60$ seconds to process a batch. Each batch is a sequence of Web Service invocations and configured with a randomly generated *configuration string* which specifies the location and size of input files (e.g., $\{(Location, FileId)\} = \{(8, 6), (0, 1), \dots, (7, 9)\}$ specifies that the first service should be invoked with the URI of the 50 MB input file which is located on compute-node 8 and the last service should be invoked with the URI of the 1 GB input file which is located on compute-node 7). For static services, the data-intensive service is deployed on all compute-nodes and the WS-Orc chooses a random service and invokes it. For dynamic services, no data-intensive service is initially deployed. The WS-Orc sends a `schedule()`

request to the WS-Scheduler, and the WS-Scheduler uses the input data location aware algorithm to schedule dynamic deployment of data-intensive Web Service. We repeated our experiments 5 times to reach an acceptable percentage of standard error.

4.3.1 Comparison of Static and Dynamic Deployment

We measured the time required to execute concrete and abstract data-intensive WS-Workflows, and the total amount of data transferred during the WS-Workflow execution, to show the effectiveness of the WS-Scheduler in executing data-intensive WS-Workflows. The WS-Scheduler uses the input location aware algorithm for scheduling dynamic deployments of the data-intensive service. We executed WS-Workflows with various sizes from 10 to 100 and measured their execution time when using static services, container-level dynamic services, and service-level dynamic services. The configuration string of batches are maintained the same for every size of WS-Workflow (e.g., the WS-Workflow with the size 50 has 5 batches; we generated 5 random configuration strings and run the experiments with the same configurations for the three types of services). Throughout these experiments the WAN Model introduced in Section 4.1.3 is used to generate available bandwidth between nodes and emulate a WAN environment. Figure 4.16 and Figure 4.17 illustrate the average WS-Workflow execution time and the average amount of data transferred for various data-intensive WS-Workflow sizes respectively (using a logarithmic scale for the y-axis).

4.3.2 Discussion

Empirical studies showed that the data-intensive WS-Workflow execution time was decreased when using dynamic services. The framework allowed moving the service close to or on the same site as where the input data is located instead of moving the large input

data to the service. Empirical studies also showed that the amount of data which has been transferred is reduced when using dynamic services.

Figure 4.15: Data-intensive workflow

Figure 4.16: Average WS-Workflow execution time for various data-intensive WS-Workflow sizes

Figure 4.17: Average amount of data transferred for various data-intensive WS-Workflow sizes

4.4 Summary

In the current chapter we explained the experimental environment and the experiments performed to evaluate the performance of the WS-Scheduler and the framework. Empirical studies showed that the framework increased resource utilization and decreased WS-Workflow execution time. Having showed that the framework is able to dynamically schedule and deploy Web Services on Grid resources, we argue that it also ensures more flexible and fault-tolerant WS-Workflows. A concrete WS-Workflow is not as flexible and fault-tolerant as an abstract WS-Workflow, because if one or more of its services are not available, the user should intervene to choose another available static service. On the other hand, in abstract WS-Workflows, the framework deploys services dynamically which helps having more flexible and fault-tolerant WS-Workflows. In addition, if there is a popular service with a high workload, the framework can reduce the likelihood and frequency of bottlenecks by replicating that service on other available resources.

Chapter 5

Conclusions

5.1 Summary of Contributions

In this thesis, first we described some fundamental specifications and technologies and the relations between them for a better understanding of this project. Then we gave taxonomies for workflows and workflow execution management systems, and discussed the advantages and disadvantages of each approach. After that, we studied a number of common workflow execution management systems, each of which is distinguished by different Grid workflow management system layers and we discussed their capabilities, pros and cons. We also mapped these systems to the taxonomies when applicable.

Based on these discussions and the gap analysis we made, we concluded that we need a framework to enable dynamic Web Service scheduling and deployment on the Grid resources at runtime. This enables us to conform with the converging specifications of the Grid community and Web community, support the dynamic nature of large-scale Grids, and reduce WS-Workflow execution time. Dynamic Web Service deployment on the Grid allows us to deploy Web Services at the same site as where the input data is located and therefore decreases the data-intensive WS-Workflow execution time by removing the overhead time

needed for data transfer. Moreover, by using the dynamic Web Service scheduling and deployment feature we are able to distribute workload on the Grid resources and avoid bottlenecks.

We proposed, designed and implemented a dynamic Web Service scheduling and deployment framework called DynaSched that uses some of the GT4 services and provides a number of additional services to enable dynamic Web Service scheduling and deployment on the Grid. In brief, DynaSched includes the WS-Orc which uses the WS-Scheduler to execute abstract WS-Workflows on the Grid. The WS-Scheduler decides where to deploy Web Services based on a scheduling algorithm. There are two scheduling algorithms available: (i) a load balancing algorithm which is suitable for compute-intensive services and (ii) an input data location aware algorithm which is suitable for data-intensive services. The WS-Scheduler contacts other components of the framework including the WS-CodeR and the WS-Factories along with the information and data management services provided by GT4 to deploy Web Services dynamically on selected compute-nodes.

We performed two series of experiments for: (i) compute-intensive WS-Workflows and (ii) data-intensive WS-Workflows to evaluate the performance of the WS-Scheduler and the framework. Empirical studies showed that the framework increased resource utilization and decreased WS-Workflow execution time. We argued that the framework ensures more flexible, fault-tolerant workflows and also reduces the likelihood and frequency of bottlenecks.

5.2 Recommendations for Future Work

DynaSched is not yet a complete framework. Our research to date focuses on the development of a dynamic Web Service scheduling and deployment framework for Grid WS-Workflows and the algorithms under the framework to support: (i) load balancing and

(ii) input data location aware scheduling of dynamic Web Services. However, further research and implementation work along the following directions could also be carried out:

- **Hierarchical WS-Workflow Schedulers:** Building a hierarchy of WS-Schedulers can help utilizing more resources on the Grid and boost the performance of the framework. A meta-scheduler which is able to partition a WS-Workflow into several sub WS-Workflows is a key component to realize a hierarchy of WS-Schedulers.
- **Scheduling Algorithms:** Further scheduling algorithms can lead to a better framework performance for a wider range of WS-Workflows. A number of possible scheduling algorithms are: (i) a hybrid scheduling algorithm for hybrid WS-Workflows which are composed of both compute-intensive and data-intensive Web Services, (ii) a partitioning algorithm which takes the whole WS-Workflow into consideration and makes scheduling decisions according to the dependencies and logical paths between its Web Services, (iii) customized algorithms which are optimized for real-world WS-Workflows.
- **Standard WS-Workflow Orchestration Engine:** Using a standard WS-Workflow orchestration engine accelerates WS-Workflow composition and facilitates collaboration between users. Research issues to propose and standardize the support of abstract WS-Workflows based on standard workflow specifications such as WS-BPEL should be addressed to achieve this goal.
- **Fault Handling:** The Grid is a dynamic environment in which resources join and leave the resource pool unexpectedly. Hence, in such a dynamic environment harnessing newly joined resources and recovering when they leave unexpectedly is an essential research issue that must be taken into consideration. This feature can only be realized by the cooperation of all of the framework components. Handling faults in the framework can be challenging because of the inherent loosely coupled characteristic of the framework's service oriented architecture.

- **Security Issues:** Security is an important issue in every framework which deals with shared resources. It is necessary to keep Web Service implementation details confidential and make Web Service functionalities accessible only for users with the appropriate credentials. In addition the framework must guarantee that shared resources stay safe by rejecting malicious codes so that they are unable to execute on the compute-nodes.

Bibliography

- [1] The Globus Alliance, "An ecosystem of grid components." [Online]. Available: <http://www.globus.org/grid/software/ecology.php>
- [2] Channabasavaiah, Holley, and Tuggle, "Migrating to a service-oriented architecture," IBM DeveloperWorks, Tech. Rep., 2003.
- [3] Jia Yu and Rajkumar Buyya, "A taxonomy of scientific workflow systems for grid computing," *SIGMOD Rec.*, vol. 34, no. 3, pp. 44–49, September 2005.
- [4] Yuhong Feng, "A mobile code collaboration framework for grid computing," Ph.D. dissertation, Nanyang Technological University, 2007.
- [5] Markus Keidl, Stefan Seltzsam, Konrad Stocker, and Alfons Kemper, "ServiceGlobe: Distributing E-Services Across the Internet," in *International Conference on Very Large Data Bases*, 2002, pp. 1047–1050.
- [6] Markus Keidl, Stefan Seltzsam, and Alfons Kemper, "Reliable web service execution and deployment in dynamic environments," in *Technologies for E-Services*, 2003, pp. 104–118.
- [7] Markus Keidl, Stefan Seltzsam, and Alfons Kemper, "ServiceGlobe: Flexible and Reliable Web Services on the Internet," in *World Wide Web Conference (Posters)*, 2003.

-
- [8] Li Qi, Hai Jin, Ian Foster, and Jarek Gawor, "Hand: Highly available dynamic deployment infrastructure for globus toolkit 4," *Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on*, pp. 155–162, Feb. 2007.
 - [9] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," 2002. [Online]. Available: citeseer.ist.psu.edu/foster02physiology.html
 - [10] OASIS SOA Reference Model Technical Committee, "Reference Model for Service Oriented Architecture 1.0," OASIS, Tech. Rep., 2006.
 - [11] W3C Committee, "Web services architecture," W3C Working Group Note, February 2004. [Online]. Available: <http://www.w3.org/TR/ws-arch/>
 - [12] W3C Committee, "SOAP Version 1.2 Part 1: Messaging Framework & Part 2: Adjuncts," W3C Recommendation, June 2003. [Online]. Available: <http://www.w3.org/TR/soap12-part1/http://www.w3.org/TR/soap12-part2/>
 - [13] W3C Committee, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," W3C Recommendation, June 2007. [Online]. Available: <http://www.w3.org/TR/wsdl20>
 - [14] OASIS UDDI Technical Committee, "Universal Description, Discovery and Integration (UDDI): Advanced Web Service Discovery Standard." [Online]. Available: <http://www.uddi.org>
 - [15] Borja Sotomayor and Lisa Childers, *Globus Toolkit 4: Programming Java Services*. Morgan Kaufmann, 2005.

- [16] OASIS Web Services Resource Framework (WSRF) Technical Committee, "Web Service Resource Framework." [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf
- [17] OASIS Web Services Resource Framework (WSRF) Technical Committee, "Web Services Resource Framework (WSRF) Primer v1.2," Committee Draft, May 2006. [Online]. Available: <http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf>
- [18] Karl Czajkowski, Donald F Ferguson, Ian Foster, Jeffrey Frey, Steve Graham, Igor Sedukhin, David Snelling, Steve Tuecke, and William Vambenepe, "The WS-Resource Framework Version 1.0," May 2004. [Online]. Available: <http://www.globus.org/wsrf/specs/ws-wsrf.pdf>
- [19] OASIS Web Services Resource Framework (WSRF) Technical Committee, "Web Services Resource 1.2 (WS-Resource)," April 2006. [Online]. Available: http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf
- [20] OASIS Web Services Resource Framework (WSRF) Technical Committee, "Web Services Resource Properties 1.2 (WS-ResourceProperties)," April 2006. [Online]. Available: http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf
- [21] OASIS Web Services Resource Framework (WSRF) Technical Committee, "Web Services Resource Lifetime 1.2 (WS-ResourceLifetime)," April 2006. [Online]. Available: http://docs.oasis-open.org/wsrf/wsrf-ws_resource_lifetime-1.2-spec-os.pdf
- [22] OASIS Web Services Resource Framework (WSRF) Technical Committee, "Web Services Service Group 1.2 (WS-ServiceGroup)," April 2006. [Online]. Available: http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-os.pdf

- [23] OASIS Web Services Resource Framework (WSRF) Technical Committee, "Web Services Base Faults 1.2 (WS-BaseFaults)," April 2006. [Online]. Available: http://docs.oasis-open.org/wsrp/wsrp-ws_base_faults-1.2-spec-os.pdf
- [24] OASIS Web Services Notification (WSN) Technical Committee, "Web Services Notification." [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn
- [25] W3C WS-Addressing Working Group, "Web service addressing (ws-addressing)," August 2004. [Online]. Available: <http://www.w3.org/Submission/ws-addressing/>
- [26] OASIS WSBPEL Technical Committee, "Web Services Business Process Execution Language Version 2.0," OASIS, Committee Draft, 2006.
- [27] Jia Yu and Rajkumar Buyya, "A taxonomy of workflow management systems for grid computing," *Journal of Grid Computing*, vol. 3, pp. 171–200, January 2006.
- [28] Volker Hamscher, Uwe Schwiegelshohn, Achim Streit, and Ramin Yahyapour, "Evaluation of Job-Scheduling Strategies for Grid Computing," in *GRID 2000: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*. London, UK: Springer-Verlag, 2000, pp. 191–202.
- [29] University of Wisconsin-Madison Condor Team, "Condor project." [Online]. Available: <http://www.cs.wisc.edu/condor/>
- [30] University of Wisconsin-Madison Condor Team, "Condor version 6.8.6 manual," May 2007. [Online]. Available: <http://www.cs.wisc.edu/condor/manual/v6.8/>
- [31] "VDS - The GriPhyN Virtual Data System." [Online]. Available: <http://www.ci.uchicago.edu/wiki/bin/view/VDS/VDSWeb/WebMain>
- [32] "Pegasus." [Online]. Available: <http://pegasus.isi.edu/>

-
- [33] The Globus Alliance, "The Globus Toolkit." [Online]. Available: <http://www.globus.org/toolkit/>
 - [34] Ian Foster, Carl Kesselman, and Steven Tuecke, "The anatomy of the grid - enabling scalable virtual organizations," 2001. [Online]. Available: <http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0103025>
 - [35] Ian Foster, Hiro Kishimoto, Andreas Savva, Dave Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, Jem Treadwell, and J. Von Reich, "The Open Grid Services Architecture, Version 1.5," Published by Global Grid Forum Open Grid Service Architecture Working Group, July 2006. [Online]. Available: <http://www.ggf.org/documents/GFD.80.pdf>
 - [36] Ian Foster, Tom Maguire, and D. Snelling, "OGSA WSRF Basic Profile 1.0," Published by Global Grid Forum Open Grid Service Architecture Working Group, May 2006. [Online]. Available: <http://www.ggf.org/documents/GFD.72.pdf>
 - [37] The Globus Alliance, "Globus Toolkit 4.0 Release Manuals." [Online]. Available: <http://globus.org/toolkit/docs/latest-stable/>
 - [38] Borja Sotomayor, "The globus toolkit 4 programmer's tutorial." [Online]. Available: <http://gdp.globus.org/gt4-tutorial/>
 - [39] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny, "Condor - a distributed job scheduler," in *Beowulf Cluster Computing with Linux*, Thomas Sterling, Ed. MIT Press, October 2001. [Online]. Available: <http://www.cs.wisc.edu/condor/doc/beowulf-chapter-rev1.pdf>
 - [40] Douglas Thain and Miron Livny, "Building reliable clients and servers," in *The Grid: Blueprint for a New Computing Infrastructure*, Ian Foster

- and Carl Kesselman, Eds. Morgan Kaufmann, 2003. [Online]. Available: <http://www.cs.wisc.edu/condor/doc/grid2-ch19.pdf>
- [41] Douglas Thain, Todd Tannenbaum, and Miron Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality*, Fran Berman, Geoffrey Fox, and Tony Hey, Eds. John Wiley & Sons Inc., December 2002. [Online]. Available: http://media.wiley.com/product_data/excerpt/90/04708531/0470853190.pdf
- [42] "VDS Users Guide." [Online]. Available: http://griodine.uchicago.edu/~voeckler/userguide/VDSUG_Master.pdf
- [43] Ewa Deelman, Gurmeet Singh, Mei-Hui su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems." *Scientific Programming*, vol. 13, no. 3, pp. 219 – 237, 2005.
- [44] Gurmeet Singh, Ewa Deelman, Gaurang Mehta, Karan Vahi, Mei-Hui Su, G. Bruce Berriman, John Good, Joseph C. Jacob, Daniel S. Katz, Albert Lazzarini, Kent Blackburn, and Scott Koranda, "The pegasus portal: web based grid computing," in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA: ACM Press, 2005, pp. 680–686.
- [45] Yuhong Feng and Wentong Cai, "A comparison study between distributed and centralized job workflow execution models," in *Proceedings of the 3rd International Workshop on Grid Computing and Applications (GCA 2007)*, 2007, pp. 76–88.
- [46] University of California, Berkeley Millennium Project, "Ganglia." [Online]. Available: <http://ganglia.info/>

BIBLIOGRAPHY

107

-
- [47] “Enhanced TTCP.” [Online]. Available: <http://sourceforge.net/projects/ettcp>
 - [48] Praveen Yalagandula, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Sung-Ju Lee, “S3: a scalable sensing service for monitoring large networked systems,” in *INM '06: Proceedings of the 2006 SIGCOMM workshop on Internet network management*. New York, NY, USA: ACM, 2006, pp. 71–76.
 - [49] “Planetlab.” [Online]. Available: <http://planet-lab.org/>

Appendices

Appendix A

Publications

A.1 Conference Papers

- Shayan Shahand, Stephen John Turner, Wentong Cai, Maryam Khademi Hedayat, “DynaSched: a dynamic Web service scheduling and deployment framework for data-intensive Grid workflows”, in *the 10th International Conference on Computational Science (ICCS 2010)*, Jun 2010.

Appendix B

Glossary

AC	Agent Core
CoD	Code on Demand
DAG	Directed Acyclic Graph
DAGMan	Directed Acyclic Graph Manager
DynaSched	Dynamic Web Service Scheduling and Deployment Framework
EPR	Endpoint Reference
FTP	File Transfer Protocol
GASS	Global Access to Secondary Storage
GGF	Global Grid Forum
GRAM	Grid Resource Allocation Manager

GriPhyN	GRId PHYsics Network
GSI	Grid Security Infrastructure
GT4	Globus Toolkit Ver.4
HAND	Highly Available dyNamic Deployment infras- tructure
LFN	Logical File Name
LMA	Lightweight Mobile Agent
LRC	Local Replica Catalog
MA	Mobile Agent
MCCF	Mobile Code Collaboration Framework
MDS	Monitoring and Discovering System
MPI	Message Passing Interface
OASIS	Organization for the Advancement of Struc- tured Information Standards
OGSA	Open Grid Services Architecture
P2P	Peer to Peer
Pegasus	Planning for Execution in Grids
PFN	Physical File Name
PVM	Parallel Virtual Machine
RFT	Reliable File Transfer

RIS	Resource Information System
RLS	Replica Location Service
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration
URI	Uniform Resource Identifier
VDC	Virtual Data Catalog
VDL	Virtual Data Language
VDS	Virtual Data System
W3C	World Wide Web Consortium
WS	Web Service
WS-BPEL	Web Services Business Process Execution Language
WS-CodeR	Web Service Code Repository
WS-Factory	Dynamic Web Service Factory
WS-Orc	WS-Workflow Orchestration Engine
WS-Scheduler	Web Service Scheduler
WS-Workflow	Web Service Workflow
WSDD	Web Service Deployment Descriptor
WSDL	Web Service Description Language
WSRF	Web Services Resource Framework