A mobile code collaboration framework for grid computing

Feng, Yuhong

2008

Feng, Y. H. (2008). A mobile code collaboration framework for grid computing. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/13596>

<https://doi.org/10.32657/10356/13596>

Nanyang Technological University

A Mobile Code Collaboration Framework for Grid Computing

Ph.D Thesis

by

Feng Yuhong

Supervisor: Dr. Cai Wentong

Division of Computer Science
School of Computer Engineering
Nanyang Technological University
Singapore 639798

May 2008

Statement of Originality

I hereby certify that the content of this thesis is the result of work done by myself and has not been submitted for a higher degree to any other University or Institution.

.....

Date

.....

Signature

To my family,
especially to my beloved son, Ningyuan,
who is born during the writing of this thesis.

Abstract

Data intensive scientific applications often involve diverse, high volume, and distributed data sets. They can generally be viewed as job workflows in which subjobs (i.e., nodes or activities) represent application components and dependencies represent the interactions between the components.

To reduce the communication overhead caused by data movement and to provide decentralized control of execution during the workflow enactment, the Mobile Code Collaboration Framework (MCCF) is developed in this thesis to map the execution of subjobs to the distributed resources and to coordinate the subjobs' execution at runtime according to the abstract workflow provided by users. Light-weight Mobile Agent (LMA) and Code-on-Demand (CoD) techniques are adopted in the development of the MCCF, so that an analysis module in data intensive scientific applications can be executed at a computational resource close to where the required data set is located.

The MCCF, which does not have a centralized engine, is different from the existing scientific workflow engines (e.g., Condor's DAGMan, SCIRun, Triana, and Taverna). When multiple data independent subjobs can be executed concurrently, replicas of an existing LMA will be generated so that there is one LMA for each subjob. The LMAs will then be migrated to the different computational resources for the execution of these data independent subjobs in parallel. Because of the data dependencies in a job workflow, before an LMA executes a subjob, it needs to locate the execution results of its predecessors. When multiple concurrently executing subjobs have a common immediate successor, only one of the corresponding LMAs should be selected for the latter's execution. Others should be discarded if they are not migrated to any successors' execution. Due to the lack of a centralized engine, execution coordination is therefore required in the MCCF. In addition, the distributed nature of the MCCF also gives rise to the requirement of a distributed algorithm for various LMAs working collaboratively to collect complete, ag-

gregated provenance information. In the context of the MCCF, provenance information includes execution provenance and data provenance, where execution provenance refers to the computational resources and executable code used for executing subjobs, and data provenance refers to the identities of the data sets used in the execution of subjobs and the locations of intermediate results.

A simple prototype of MCCF has been developed using Globus Toolkits, and its associated algorithms for execution coordination and provenance recording and collection have also been implemented on top of the MCCF prototype. Quantitative analysis and an experimental comparison study have also been carried out to demonstrate their effectiveness.

Keywords: Grid Computing, Distributed Job Workflow Execution, Code Mobility, Mobile Agent, Dynamic Service, Execution Coordination, Mobile Agent Communication, Communication Partner Identification, Provenance.

Acknowledgement

First, I would like to express my deep gratefulness to my supervisor, Dr. Cai Wentong, for his patient and systematic guidance, heuristic advice and encouragement in my research work and the writing up of this thesis.

Next, I would give my thankfulness to Dr. Stephen John Turner, Dr. Cao Jiannong, Dr. Low Yoke Hean Malcolm, Dr. Lee Bu Sung and Dr. Bertil Schmidt for their patience of listening and giving me valuable advice.

Third, I also wish to say thanks to Ms. Liu Li, Dr. Wu Jigang, Dr. Zeng Yi, Dr. Tang Ming, Mr. Wang Lizhe, Dr. Tan Choong Leong, Dr. Liu Weiguo, Mr. Ho Quoc Thuan, who have rendered their help to my research work in one way or another.

Last but not least, I am deeply grateful to my husband, Haoming, for his heuristic advice and encouragement for my research work, and consistent support throughout my Ph.D. study.

Contents

Abstract	i
Acknowledgement	iii
Contents	iv
List of Figures	ix
List of Tables	xii
List of Notations	xiv
1 Introduction	1
1.1 The Background	1
1.1.1 Data Intensive Scientific Computations	1
1.1.2 Job Workflow Execution	2
1.2 Scope and Objectives	5
1.3 Thesis Organization	9
2 A Taxonomy of Job Workflow Execution Models	12

Ph.D Thesis

CONTENTS

2.1	Mobile Code Design Paradigms	13
2.2	Job Workflow	16
2.2.1	Job Workflow Representation	16
2.2.2	A Taxonomy of Job Workflow Execution Models	17
2.3	Simulation Study	26
2.3.1	Grid Resources	27
2.3.2	Assumptions and Parameters	28
2.3.3	Simulation of Networks	29
2.3.4	Simulation of Job Workflow Models	31
2.3.5	Simulation Result & Analysis	34
2.4	Summary	36
3	Mobile Code Collaboration Framework (MCCF)	38
3.1	System Overview	38
3.2	System Design	41
3.2.1	Build Time Functions	41
3.2.2	Runtime Functions	47
3.3	Job Workflow Execution on the MCCF	49
3.3.1	Dynamic Execution of Job Workflows	50
3.3.2	Distributed Job Workflow Execution	50
3.4	Prototype	53
3.4.1	Multi-agent Platform	53

3.4.2	Prototype Implementation	55
3.5	Summary	56
4	Dynamic Partner Identification in Mobile Agent based Distributed Job Workflow Execution	58
4.1	Classification of Execution Coordination Techniques	59
4.2	Agent Communication	63
4.3	Partner Identification	66
4.3.1	Preprocessing – Subjob Grouping Algorithm	66
4.3.2	Static Partner Identification Algorithm	68
4.3.3	Dynamic Partner Identification Algorithm	72
4.3.4	Correctness and Optimality	75
4.4	Experimental Study	78
4.4.1	Testbed Configuration	79
4.4.2	Experiments Using d-h-TGs	80
4.4.3	Experiments Using Real Application TG and Randomly Generated TGs	87
4.5	Summary	92
5	A Delegator based Execution Coordination in Mobile Agent based Distributed Job Workflow Execution	95
5.1	Existing Execution Coordination Techniques Applied in the MCCF . . .	96
5.1.1	Mailbox Based Coordination	97

5.1.2	Contact List Based Coordination	102
5.2	Delegator based Coordination	104
5.3	Performance Evaluation	108
5.3.1	Experiments Using d-h-TGs	109
5.3.2	Experiments Using Real Application TG and Randomly Generated TGs	112
5.4	Summary	115
6	Provenance Recording and Collection in Mobile Agent-Based Distributed Job Workflow Execution	117
6.1	Related Work	118
6.2	Provenance Recording and Collection	120
6.2.1	Properties of Preprocessing Algorithm	120
6.2.2	Provenance Recording and Propagation	123
6.3	Performance Evaluation	125
6.4	Summary	128
7	A Comparison Study between Centralized and Distributed Job Work- flow Execution Models	129
7.1	Introduction to Condor DAGMan	129
7.2	Job Workflow Execution on Condor DAGMan and the MCCF	131
7.3	Performance Evaluation	133
7.4	Summary	135

8	Conclusions	138
8.1	Summary of Contributions	138
8.2	Recommendations for Future Work	142
8.3	Summary	144
	Appendices	145
A	Condor DAGMan Scripts	145
B	Publication List for this Research	149
B.1	Journal	149
B.2	Conference	149

List of Figures

2.1	An Example of Indirect Successors	17
2.2	Intermediate Result Movement	18
2.3	Subjob Code Acquisition	21
2.4	Control Management	23
2.5	A Taxonomy of Job Workflow Execution Models	25
2.6	Job Workflow Execution over the Grid	28
2.7	Makespans of Different Job Workflow Execution Models	35
3.1	The MCCF System Model	39
3.2	The MCCF Architecture	40
3.3	Schema for Job Workflow Specification	42
3.4	Schema for Agent Core	46
3.5	AC Migration	49
3.6	Distributed Execution of Job Workflow $\mathcal{W}_a$	51
4.1	Execution Coordination Techniques	63
4.2	Subjob Grouping for Job Workflow $\mathcal{W}_a$	66

4.3	Reduced Task Graph for Static Partner Identification	69
4.4	Preprocessing-based Static Partner Identification	70
4.5	Update of Partner Sets during the Execution of Job Workflow $\mathcal{W}_a$	73
4.6	Job Workflow Execution Scenarios	76
4.7	Different Types of Regular TGs	81
4.8	A “3-2-TG” Job Workflow	82
4.9	Total Number of Partner Update Messages for the Simulated “d-2-TG” Job Workflows	84
4.10	Simulated “d-2-TG” Job Workflows’ Makespan on Emulated WAN . . .	85
4.11	Simulated “d-2-TG” Job Workflows’ Makespan on a LAN Cluster	87
4.12	Real Application TG	88
4.13	Random Task Graphs	89
4.14	Preprocessed Random Task Graphs	90
4.15	Total Number of Partner Update Messages for the Simulated SDSS and Random Job Workflows	90
4.16	Simulated SDSS and Random Job Workflow Makespans on Emulated WAN	91
5.1	Mailbox based Coordination	98
5.2	Dynamic Partnership	99
5.3	The Retrieval of the Location of Predecessors’ Execution Results	101
5.4	Contact List based Coordination	103
5.5	Delegator based Coordination	106

5.6	Experiment Results for Simulated “d-2-TG” Job Workflows (I)	109
5.7	Experiment Results for Simulated “d-2-TG” Job Workflows (II)	111
5.8	Experiment Results for Simulated Small SDSS and Random Job Workflows (I)	113
5.9	Experiment Results for Simulated Small SDSS and Random Job Workflows (II)	114
7.1	Workflow Execution over Condor DAGMan and the MCCF	132
A.1	Random Task Graph TG4	145

List of Tables

2.1	Mobile Code Design Paradigms	14
2.2	Evaluation for Codes Acquired by Different Approaches	22
2.3	Taxonomy Mapping to Existing Job Workflow Management Systems . . .	25
2.4	Distributed Control Management	26
2.5	Overhead Cost Assumptions	28
3.1	Data Location Types	43
3.2	Code Location Types	44
3.3	An Example Job Workflow Specification	45
3.4	Agent Platform Evaluation & Comparison	55
4.1	Partner List Obtained Using Preprocessing-based Static Partner Identifi- cation Algorithm for the Reduced Graph Shown in Figure 4.3(b)	71
4.2	Configuration of the ModelNet Testbed	79
4.3	Link Characteristics in the Emulated WAN	80
4.4	Average Size of Maximum Partner Set for the Simulated “d-2-TG” Job Workflows	84

4.5	Comparison between Preprocessing Based Static and Dynamic Algorithms (Number of Partner Update Messages)	91
6.1	Experiment Results	125
7.1	Performance Evaluation on Condor DAGMan and the MCCF.	134
A.1	General DAGMan Submit File for TG4	146
A.2	Submit File for Subjob 4 in TG4	147
A.3	“PRE” Script for Subjob 4 in TG4	147
A.4	Script for LoadDataSet_ntu.sh	148

LIST OF NOTATIONS

In this table, “X_[m₁.m₂.m₃]” means that notation “X” first appears in Section m₁.m₂.m₃. For example “ $\mathcal{G} = (\mathcal{J}, \mathcal{E})$ _[2.2.1]” means that the notation $\mathcal{G} = (\mathcal{J}, \mathcal{E})$ first appears in Section 2.2.1.

Notation	Description
$\mathcal{G} = (\mathcal{J}, \mathcal{E})$ _[2.2.1]	This denotes the directed acyclic graph (DAG) (i.e., $\mathcal{G}$) that specifies the subjobs (i.e., $\mathcal{J} = \{J_0, J_1, \dots, J_{n-1}\}$, which is the set of vertices representing subjobs) in the job workflow and the data dependency among subjobs (i.e., $\mathcal{E}$, which is the set of directed edges between subjobs). A DAG is also referred to as a task graph (TG) in the literature.
(J_i, J_j) _[2.2.1]	This denotes the directed edge between subjob J_i and J_j , and the edge belongs to $\mathcal{E}$.
$J_i < J_j$	This denotes that there is data dependency between subjob J_i and J_j , i.e., there exists a path between these two subjobs.
$\mathcal{A}(J_i)$ _[2.2.1]	This denotes the set of ancestors of J_i .
$\mathcal{O}(J_i)$ _[2.2.1]	This denotes the set of offsprings of J_i .
$\mathcal{S}(J_i)$ _[2.2.1]	This denotes the successor set of subjob J_i .
U_DS _[2.2.2]	This denotes that the subjob code is acquired by dynamic remote code staging in.
XX-YY-ZZ _[2.2.2]	This is used to express a job workflow execution model, where XX can be centralized or distributed, YY can be centralized, mediated, or P2P (Peer-to-Peer), and ZZ can be SS (static service), DS (dynamic service) or U_DS.
$\mathcal{D}$ _[2.3.1]	This denotes the set of data repositories having the required data set.
$\mathcal{M}$ _[2.3.1]	This denotes the set of computational resources satisfying the subjob’s computation requirements.
$\mathcal{C}$ _[2.3.1]	This denotes the set of code repositories having the required executable code.

Notation	Description
$\mathcal{N}$ [2.3.1]	This denotes the matrix capturing network characteristics among nodes in $\mathcal{D}$, $\mathcal{M}$, and $\mathcal{C}$.
$(\mathcal{D}, \mathcal{M}, \mathcal{C}, \mathcal{N})$ [2.3.1]	This denotes the Grid, which consists of resources $\mathcal{D}$, $\mathcal{M}$, $\mathcal{C}$, and $\mathcal{N}$.
$\mathcal{W}_x$ [3.3.2]	This denotes the job workflow with id x .
$\mathcal{C}_{x,k}$ [3.3.2]	This denotes the k th AC replica created for the execution of job workflow $\mathcal{W}_x$. The AC replica is uniquely identified using tuple (x, k) . Particularly, the first AC created for executing a job workflow $\mathcal{W}_x$ is denoted as $\mathcal{C}_{x,0}$.
(x, k) [3.3.2] AC_id [5.1.1]	The id of an AC replica.
$\mathcal{A}_{J_i}^{x,k}$ [3.3.2]	This denotes the group of agents created from AC replica $\mathcal{C}_{x,k}$ to execute subjob J_i .
$((x, k), J_i)$ [4.2]	This denotes a partner in the partner set of an AC replica.
g_i [4.3.1]	This denotes a subjob group generated by the preprocessing process.
$\mathcal{G}' = (\mathcal{J}', \mathcal{E}')$ [4.3.2]	This denotes the reduced task graph for task graph $\mathcal{G} = (\mathcal{J}, \mathcal{E})$.
$\mathcal{E}'_c$ [4.3.2]	This denotes the set of “c” edges in the reduced graph.
$\mathcal{E}'_d$ [4.3.2]	This denotes the set of “d” edges in the reduced graph.
$\mathcal{E}_d$ [4.3.3]	This denotes the set of “d” edges in the original task graph $\mathcal{G} = (\mathcal{J}, \mathcal{E})$.
$\mathcal{P}$ [4.3.2]	This denotes the communication partner set. $(g_i, g_j) \in \mathcal{P}$ means that g_i and g_j are partner groups.
$\mathcal{A}_c(g_i)$ [4.3.2]	This denotes a set of vertices so that for each vertex (e.g., g_t) in the set, there is a path from g_t to g_i and for every edge e in the path, $e \in \mathcal{E}'_c$.
$\mathcal{T}_{J_i}$ [4.3.3]	This denotes the sub-tree rooted at J_i in the spanning tree generated by the subjob grouping algorithm described in the Subsection 4.3.1.
$\mathcal{R}(\mathcal{C}_{x,k}, T)$ [4.3.4]	This denotes the set of AC replicas to be created from $\mathcal{C}_{x,k}$ at a specific time T , and the set includes $\mathcal{C}_{x,k}$ itself.
“d-h-TG” [4.4.2]	This denotes a type of task graph, which are generated by mixing an out-tree and an in-tree TGs. For a given “d-h-TG”, h is the height of the out-tree and the in-tree used in constructing the graph and d is the out-degree (in-degree) of a node in the out-tree (in-tree).
$N_J(d, h)$ [4.4.2]	This denotes the number of subjobs for a job workflow with “d-h-TG” shape.
$N_{AC}(d, h)$ [4.4.2]	This denotes the number of AC replicas created during job workflow execution for a job workflow with “d-h-TG” shape.
$\bar{L}(d, h)$ [4.4.2]	This denotes the average size of maximum partner set of all AC replicas for a job workflow with “d-h-TG” shape.
SjbId [5.1.1]	This denotes the id of the current subjob being executed by the AC replica.
MBLoc [5.1.1]	This denotes the location of the AC replica’s mailbox.

Notation	Description
$ACLoc$ [5.1.1]	This denotes the current location of the AC replica, i.e., the location of the computational resource where the current subjob is being executed.
$(AC_id, SjbId, ACLoc, MBLoc)$ [5.1.1]	This denotes the tuple information of each AC replica recorded in MBLocator.
$SjbResultLoc[]$ [5.1.1]	This denotes the received subjob result locations.
$SNewSjbId$ [5.1.1]	This denotes the id of the subjob to be executed next by the sending AC.
$ACNewLoc$ [5.1.1]	This denotes the new location of the AC.
$NewACTuple[]$ [5.1.1]	This denotes an array of the tuple information $(AC_id, ACLoc, SjbId, MBLoc)$.
$SSjbId$ [5.1.1]	This denotes the id of the newly completed subjob of the sending AC.
$PSjbId$ [5.1.1]	This denotes the id of the partner's current subjob.
$SjbIds[]$ [5.1.1]	This denotes the array of the ids of the predecessors of the AC's current subjob.
$(AC_id, SjbId, ACLoc)$ [5.1.2]	This denotes a partner in the contact list which is associated with an AC replica.
$SjbResultLoc$ [5.1.2]	This denotes the location of the execution result of the newly completed subjob.
$DelegatorLoc$ [5.2]	This denotes the current location of the partner delegator.
$(AC_id, SjbId, DelegatorLoc)$ [5.2]	This denotes the tuple information of a delegator.
$ContactList[]$ [5.2]	This denotes the contact list of the delegator's partners. It is an array of delegator tuples.
$NewDelegator-Tuple[]$ [5.2]	This denotes the tuple information of the newly created delegators.
$\mathcal{E}_i$ [6.2.1]	This denotes the set of edges that indicate indirect successors.
$\mathcal{G}'' = (\mathcal{J}'', \mathcal{E}'')$ [6.2.1]	This denotes the graph generated from the original DAG $\mathcal{G} = (\mathcal{J}, \mathcal{E})$ after removing the edges indicating indirect successors, i.e., $\mathcal{J}'' = \mathcal{J}$ and $\mathcal{E}'' = \mathcal{E} - \mathcal{E}_i$.
$P(J_i, J_j)$ [6.2.1]	This denotes the set of paths from J_i to J_j in $\mathcal{G}''$.
$G(J_i)$ [6.2.1]	This denotes the group id of subjob J_i .
$J_i \xrightarrow{m} J_j$ [6.2.1]	This denotes that the edge (J_i, J_j) is annotated with "m".
$J_i \xrightarrow{c} J_j$ [6.2.1]	This denotes that the edge (J_i, J_j) is annotated with "c".
$J_i \xrightarrow{d} J_j$ [6.2.1]	This denotes that the edge (J_i, J_j) is annotated with "d".

Notation	Description
$PP(J_i, J_{n-1})$ [6.2.1]	This denotes the set of propagation paths between J_i and J_{n-1} .
$\mathcal{R}(J_i)$ [6.2.2]	This denotes a subset of J_i 's successors, where for any subjob $J_j \in \mathcal{R}(J_i)$, we have either $(J_i \xrightarrow{m} J_j)$ or $(J_i \xrightarrow{c} J_j)$.
$H_p(J_i)$ [6.3]	This denotes the number of hops required to propagate J_i 's execution provenance information between the agents along a propagation path p of J_i (i.e., $p \in PP(J_i, J_{n-1})$) before it reaches the original AC replica.
$ g_0 $ [6.3]	This denotes the number of subjobs in the first group g_0 .

Chapter 1

Introduction

Grid computing aims at providing a flexible, secure, and coordinated resource sharing environment on distributed systems across administrative domains. It is regarded as one of the key enabling technologies for solving both computation intensive and data intensive scientific computations. This thesis addresses two key issues in data intensive scientific computations in grid computing: execution management and coordination.

1.1 The Background

1.1.1 Data Intensive Scientific Computations

Complex scientific applications can generally be viewed as job workflows in which sub-jobs (i.e., nodes or activities) represent application components and dependencies between subjobs represent the interactions between the components. In a Service Oriented Architecture (SOA), common application components are usually deployed as services. A workflow engine coordinates the data and control flow amongst multiple services and allows the enactment of workflows to take advantage of distributed services and resources.

Data intensive scientific applications, such as bioinformatics [73] and digital imaging survey (e.g., Sloan Digital Sky Survey (SDSS) [22]), often involve diverse, high volume,

and distributed data sets. They can be generally expressed as a workflow of a number of analysis modules, each of which acts on specific sets of data and performs cross multidisciplinary computations.

Grid computing aims at providing scientific communities flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions and resources [63]. Currently the Grid provides the following facilities to support executing data intensive scientific computation applications as job workflows:

- DataGrid [112] provides distributed data management facilities which support data storage, access, replication, selection and filtering.
- Globus Security Infrastructure (GSI), an implementation of the grid security architecture proposed by the Globus project [62], uses a user proxy to address the single sign-on requirement and introduces a resource proxy to enable inter-operability with local security solutions.
- Gridftp [18], Grid implementation of the common data transfer and access protocol, provides secure and efficient data movement over the Grid.
- Metacomputing Directory Service (MDS) [61] provides APIs for resource information query, discovery and comparison. MDS-1, provided in Globus1.1.2 or earlier version, provides a centralized database for information query, while MDS-2 provides distributed querying services.

1.1.2 Job Workflow Execution

Job workflow (also named as scientific workflow in the literature [27]) was first introduced as an amalgamation of scientific problem-solving and traditional workflow techniques in 1996 [101]. A job workflow specification consists of at least three components: (i) a

directed acyclic graph (DAG) that specifies the subjobs in the job workflow and the data dependency among subjobs; (ii) user policies for resource scheduling (e.g., minimum execution time or random); and (iii) execution information (e.g., the information about the execution resources for each subjob, such as the locations of the input data sets, the code to be executed by the subjob, and the computational resources used to execute the subjob). A job workflow that includes only the DAG and the user policies is called an abstract workflow; whereas one that includes also the execution information is called a concrete workflow (or executable workflow in the literature) [52, 53].

An abstract job workflow model hides the low level implementation details from the users. This allows scientists to focus on the high level domain specific aspects of their experiments or computations. The goal of the dynamic job workflow execution is to select resources for each subjob, manage the data dependency between subjobs, and enact the subjob execution automatically for scientists.

A subjob cannot start its execution before it is bound to specific execution resources. There are three binding models: *early binding model*, *intermediate binding model*, and *late binding model* [114]:

- When the early binding model is applied, a subjob's execution resources are selected from user input (i.e., at the job workflow composition time);
- When the intermediate binding model is applied, a subjob's execution resources are selected immediately before the job workflow execution takes place (i.e., at the time when an abstract job workflow is transformed into an executable format); and
- When the late binding model is applied, a subjob's execution resources are selected during the job workflow execution.

Because of Grid resources' dynamism (e.g., Grid resources may join and leave with or

without notice, and the resources' characteristics, such as the load information, can also change over time), the late binding model enables the workflow engine to use the most up-to-date information for resource scheduling. This enhances the adaptability of the job workflow execution over the Grid.

During the course of a job workflow execution, a control thread is the thread that manages the data dependency between subjobs, selects resources for ready subjobs, and enacts the subjobs' executions. According to how many control threads are used and where the control threads are executed, the job workflow execution model can be classified into centralized execution and distributed execution. When the centralized execution model is used, there is only one control thread and the control thread is fixed on a certain computational resource. It is easy to implement. However, since the control thread handles all the control messages during the job workflow execution, it may cause problems such as poor performance, single point of failure, and poor scalability.

When the distributed execution model is used, there can exist multiple control threads and the control threads are deployed on distributed computational resources. Multiple control threads manage the control messages for data independent subjobs, which can improve the system performance. Therefore, research works have been carried out to develop distributed workflow execution models using *collaborative engines* [25, 49, 119] or *mobile agents* [13]. When collaborative engines are used, the subjob executions in a job workflow are not handled by a centralized engine, but by multiple distributed engines collaboratively. These engines are instantiated on distributed computational resources and share the same job workflow specifications. They independently or collaboratively select the resources, and control the executions for the subjobs for which they are responsible.

A mobile agent is an object which is capable of migrating from computational resource to computational resource, performing computations on behalf of the user [1]. Mobile agent based job workflow execution is another paradigm for distributed job workflow ex-

ecution. Data intensive scientific computations often involve large distributed data sets, the size of the computation code can be assumed to be less than that of the required distributed data sets [24]. Therefore, moving code instead of data sets over the network can help overcome network latency in Grid computing. In addition, agents are autonomous active entities that can perceive, act and react in their environment [28], thus, mobile agent based distributed job workflow execution can be more adaptable to the dynamic Grid resources.

1.2 Scope and Objectives

To reduce the communication overhead caused by data movement and to provide decentralized control of execution during the workflow enactment, a Mobile Code Collaboration Framework (MCCF) using mobile agent technology for job workflow execution is developed in this thesis. The MCCF, which does not have a centralized engine, is different from the existing scientific workflow engines (e.g., Condor's DAGMan^{1.1} and SCIRun^{1.2}). It maps subjobs to the distributed Grid resources and coordinates the subjob executions at runtime according to the abstract workflow provided by users.

In addition to dynamism, the Grid resources have the following characteristics: (i) Distributed location: the Grid resources are physically distributed in the world; (ii) Heterogeneity: the Grid may consist of heterogeneous resources; (iii) Long latency: the communication latency over Wide Area Network (WAN) is significantly longer than that of Local Area Network (LAN); and (iv) Limited bandwidth: the network bandwidth on WAN is limited and varies unpredictably with the underlying network traffic. In existing mobile agent based job workflow systems, e.g., Bioagent [13], a mobile agent carries all the implementation codes during the migration. In order to adapt to Grid resources' het-

^{1.1}<http://www.cs.wisc.edu/condor/dagman/>

^{1.2}<http://software.sci.utah.edu/scirun.html>

erogeneity, a mobile agent may need to carry all possible supporting packages for various architectures during its migration, which greatly enlarges the code volume. Therefore, the overhead caused by code transfer over the WAN is increased fast when the scale of the job workflow gets larger. This compromises the benefits of mobile agents.

The major objectives the MCCF aims to achieve are: (i) Develop a framework using light-weight mobile agents to support distributed execution of the job workflow on the Grid; and (ii) Develop mechanisms under the framework to support:

- **Parallel Execution of Data Independent Subjobs:** In the MCCF, a mobile agent may be responsible for the execution of a group of, but not all, subjobs. When multiple data independent subjobs can be executed concurrently, replicas of an existing mobile agent will be generated so that there is one mobile agent for each subjob. The mobile agents will then be migrated to different computational resources for the execution of these data independent subjobs in parallel. When multiple concurrently executing subjobs have a common successor, only one of the corresponding mobile agent replicas should be selected for the latter's execution. Others should be discarded if they are not migrated to any successor's execution .
- **Execution Coordination:** During a job workflow's execution in the MCCF, it is necessary for the mobile agent executing a subjob to locate the input data from its predecessors. If the subjob's predecessors are executed by the same mobile agent, the locations of the input data from its predecessors would be readily available in the job workflow specification carried by the mobile agent. Otherwise, the mobile agent executing the subjob needs to get the locations of its predecessors' execution results through an execution coordination mechanism.

According to how a subjob obtains the locations of its predecessors' execution results, the execution coordination techniques for mobile agents can be classified into

indirect and *direct* coordination. In indirect coordination, the coordination does not take place between two mobile agents directly, instead, it relies on a shared memory for the exchange of the location of a subjob's execution result [42]. That is, a subjob obtains the locations of its predecessors' execution results through accessing the shared memory. There is no direct communication between two mobile agents. Shared memory can be implemented either globally or locally. For global shared memory, a centralized server is used. Agents communicate with the server to access the memory. Local shared memory can only be accessed by the agents residing on the same host. Therefore, indirect coordination either relies on a centralized server or requires that two distributed mobile agents move to the same host for coordination. A centralized server is prone to become the system bottleneck and causes a single point of failure; whereas requiring two distributed agents moving to the same host for coordination introduces much more overheads. Thus, in this thesis, we only consider direct coordination techniques.

When a direct coordination technique is used, mobile agents interact with each other directly or through the resources that belong to the interacting mobile agents (e.g., mailbox) [42]. An explicit communication between agents or their resources is initiated (i.e., the mobile agent must explicitly name its partner before communication takes place). Two mobile agents communicating with each other are called *communication partners* or *partners* in short. For mobile agent communication, the first step is to identify corresponding partners, the second step is to locate (or discover) them, and the third step is to route the message to them. Previous work [48, 69, 75, 92, 98, 104] on mobile agent communication normally assumes that the partner is already known and focuses mainly on dynamic mobile agent location discovery and message transmission. However, with dynamic mobile agent replication and distributed job workflow execution, the partners with which a mobile agent

needs to communicate may change during runtime. Thus, partner identification is an important issue for direct coordination in the MCCF.

Mobile agent coordination is a well researched area, especially in the area of information retrieval [43]. However, in the context of mobile agent based workflow execution, how an execution coordination technique may affect a job workflow execution has not been investigated yet.

- **Distributed Provenance Recording and Collection:** The provenance of some data is defined as the documentation of the process that led to the data [71]. The necessity of provenance for a job workflow execution is apparent since provenance provides a traceable path on how a job workflow was executed and how the resulting data were derived. It is particularly important in SOA since shared services and data sets might be used in the course of the job workflow execution. Job workflow systems automate the execution of scientific applications, however they may hide how the results are achieved.

The provenance information can be generated from the static information available in the abstract workflow (e.g., data dependencies) together with the runtime details obtained by tracing the execution of the workflow execution. The trace can be automatically generated by developing either a special “wrapping service” of the engine [94] or an “engine plugin” [126] to capture and record provenance related data directly from the workflow engine. The workflow trace can also be collected collectively by the services that execute the subjobs [35] or the services together with the workflow engine [100]. But, this puts the responsibility of provenance data recording to the service providers and may also require service modification.

No matter how the traces are collected, in general some special provenance services are used in the current systems to store the provenance data and to provide an interface for users to query the data. Thus, a protocol is needed for various service

providers and the workflow engine to communicate with the provenance services during the provenance collection process [71].

There is no centralized workflow engine in the MCCF. So, given the distributed nature of the MCCF, there is a need to develop a distributed algorithm so that mobile agents can work collaboratively to collect complete, aggregated provenance information.

1.3 Thesis Organization

The rest of the thesis is organized as follows.

- **Chapter 2** develops a taxonomy of job workflow execution models and maps it to the existing job workflow management systems. To characterize the cost/benefit tradeoff between models, a simulation study was carried out to compare the job workflow execution time (also referred to as “job workflow makespan”) of different models. The investigation results based on the simulation in this chapter show that although distributed job workflow execution using light weight mobile agent and dynamic service^{1.3} is promising, currently there does not exist a job workflow management system that supports this execution model.
- **Chapter 3** describes the preliminary design and implementation of the Mobile Code Collaboration Framework (MCCF), which uses light weight mobile agent and dynamic service for distributed job workflow execution.
- **Chapter 4** develops a classification of the existing execution coordination techniques for mobile agent systems and identifies that only direct coordination techniques are suitable for the MCCF. As above mentioned, with dynamic mobile agent

^{1.3}These concepts will be introduced in Chapter 2

replication and distributed job workflow execution, partner identification is an important issue for direct coordination in the MCCF. Based on the study of agent communications during a job workflow execution on the MCCF, unnecessary agent communications that degrade the system performance are identified. Then, a novel subjob grouping algorithm is proposed for preprocessing the job workflow's DAG specification (i.e., subjobs and their data dependencies). The obtained information is used in both static and dynamic algorithms to identify partners for agent communication. The mobile agent dynamic location and communication based on this approach is expected to reduce the agent communication overhead by removing unnecessary communication partners during the dynamic job workflow execution. The proof of the dynamic algorithm's correctness and effectiveness is elaborated in the chapter, and the algorithms are evaluated through a comparison study using simulated job workflows executed on a prototype implementation of the MCCF on a LAN environment and an emulated WAN setup. The results show the scalability and efficiency of the algorithms as well as the advantages of the dynamic algorithm over the static one.

- **Chapter 5** discusses how existing direct coordination techniques can be applied in the MCCF. Taking the combined advantages of the existing techniques, an original decentralized execution coordination technique, delegator based coordination, is proposed. Each mobile agent is associated with a delegator, which is responsible for the execution coordination on behalf of its owner using a decentralized approach. Therefore, a mobile agent can be isolated from execution coordination and achieve better performance in workflow execution. A performance study has been conducted to evaluate our approach against the existing ones using real and simulated job workflows. The results are presented and discussed in this chapter.

- **Chapter 6** describes the development and evaluation of a decentralized recording and collection scheme for job workflow provenance in mobile agent-based distributed job workflow execution. First, the differences between job workflow provenance in our context and those in the existing workflow systems are discussed. By further exploiting the properties of the partner identification algorithm, paths for execution provenance information propagation are identified, so that the provenance information is delivered along the paths together with the communication messages for data dependency coordination. A performance study was conducted to evaluate our approach against the one using a centralized provenance server. The results are discussed in this chapter.
- **Chapter 7** describes a comparison study between the MCCF (using delegator based coordination) and a commonly used job workflow management system (i.e., Condor DAGMan). The experiment results show that the MCCF achieves better job workflow makespan.
- **Chapter 8** summarizes the achievements of our research and gives a brief outline on the unsolved issues that are related to mobile agent based distributed job workflow execution over the Grid.

Chapter 2

A Taxonomy of Job Workflow Execution Models

Code mobility can be defined as the movement of the executable code over the network towards the location of the needed resource for the execution [38]. Data intensive scientific computations often involve large distributed data sets, and the size of the required subjob specific code can be assumed to be less than that of the required distributed data sets [24]. Therefore, moving code instead of data sets over the network can help overcome the long network latency over the Grid. In addition, some petabytes of data, such as data in hospitals and European Bioinformatics Institute, cannot be moved, i.e., they have to stay where they are produced [24]. In this case, moving code instead of moving data is indispensable.

In this chapter, we investigate mobile code design paradigms, develop a taxonomy of job workflow execution models, and map the taxonomy to the existing job workflow management systems. To characterize the cost/benefit tradeoff between models, a simulation study was carried out to compare the job workflow execution time of different models. Finally, a gap in the existing job workflow management systems is identified, which helps to define our research objectives.

2.1 Mobile Code Design Paradigms

According to where the executable code is provided, where its execution actually takes place, and who makes the decision for its movement, the mobile code design paradigms can be classified into *remote evaluation (REV)*, *code on demand (COD)*, and *mobile agent (MA)*. The paradigms are described and illustrated in Table 2.1, where a filled rectangle represents where the code is provided, and a rectangle with a star represents where the execution takes place.

Mobile code technologies can introduce the following benefits for Grid computing:

- They make it possible to move code to where data is located for execution. Data intensive scientific computations often involve large distributed data sets, and the size of the required subjob specific code can be assumed to be less than that of the required distributed data sets [24]. Therefore, moving code instead of data sets over the network can help to reduce communication overhead.
- The code, once written, can be re-deployed on demand. This further improves the code reusability. In order to realize the execution of the mobile code on arbitrary execution environments, there is also effort to enable automatic software dependency analysis and configuration of the execution environment [50].

In particular, new desirable capabilities can be downloaded or uploaded to systems on the fly using CoD or REV technologies. Meanwhile, unused capabilities can be removed. Such on-demand swapping of capabilities allows systems to use memory efficiently. This is especially useful for small memory constrained devices [105]. In addition, REV and MA can facilitate load balancing by moving code to execute on another remote resource when the workload on a machine becomes heavy.

Table 2.1: Mobile Code Design Paradigms

Design Paradigm	Description	Example
CoD	<p>Mobile code applications developed using this paradigm can download and link on-the-fly part of their codes from the remote resources that act as code servers [113]. It is the application who makes the decision for the code movement.</p>	Java applets [14]
REV	<p>Applications developed using this paradigm can send code and get it executed remotely [103]. It is the application who makes the decision for the code movement.</p>	extlet [115]
MA	<p>Mobile agent (MA) is an agent which is capable of migrating from one computational resource to another, performing computations on behalf of the user [1]. An agent is an autonomous active entity that can perceive, act and react in its environment [28]. It is the MA itself who makes the decision for the code movement.</p>	Ajanta [1] Aglets [83]

An MA has agent specific features, such as proactivity, communication, social ability and learning ability. An MA based job workflow execution model enables the self-controlled execution of a job workflow, which is adaptable to the dynamic Grid resources. The benefit of adopting agent technology, and mobile agents in particular, into Grid com-

puting has been detailed in [96]. Meanwhile, mobile agent technology has been widely exploited in Grid computing. Examples are the work on routing and handling of FIPA ACL messages in the Grid environment [110]; and the work on using mobile agents for load balancing [26, 118], service discovery [46], resource monitoring and management [89, 107], exception handling [47], and security [33].

In general, MA implementations can be separated into two parts: one part is the non-functional codes (e.g., codes for handling resource selection, execution coordination, and MA migration), which are common to all MAs during a job workflow execution. The other part is the functional codes, such as codes for subjob specific executions. According to what codes are carried during the MA migration, the MA in the current mobile agent systems can be classified into two categories: *monolithic mobile agent (MMA)* and *light-weight mobile agent (LMA)*. If an MA carries all the codes when migrating from one resource to another, we call it MMA. This makes the MMA include a heavy chunk of codes, and the migration of such a heavy chunk of codes over the network can compromise the benefits of the code mobility. Most of the existing mobile agent systems, e.g., Bioagent [13], Aglets [83], Voyager [12], Grasshopper [74], and NOMAD [10], do not differentiate non-functional codes and functional codes.

The Light-weight MA (LMA) technology is introduced to reduce the codes transferred over the network. When an MA carries only its non-functional codes on migration [29, 38] and functional codes are downloaded and instantiated on demand on the destination resource, the MA is called LMA. A specific functional code can be associated with a functional description. The functional description describes what services or capabilities are actually provided by that executable code [79]. It can be described using natural language, which can be ambiguous and not accessible to machines, or semantic web services technologies, e.g., OWL-S [88]. In other words, a LMA can carry only its blueprint (the functional description of the codes [39]) when it migrates. The actual executable code

will be selected based on the functional description and downloaded and instantiated when the LMA arrives at the destination resource.

2.2 Job Workflow

Job workflow was first introduced as an amalgamation of scientific problem-solving and traditional business workflow techniques in 1996. A job workflow is more data oriented than a business workflow. The goal of a job workflow execution is the derived data.

2.2.1 Job Workflow Representation

As introduced in Section 1.1.2, a job workflow includes a DAG, which can also be used to represent the job workflow. A DAG is also referred to as a task graph (TG) in the literature. It can be denoted as $\mathcal{G} = (\mathcal{J}, \mathcal{E})$, where $\mathcal{J}$ is the set of vertices representing subjobs (i.e., $\mathcal{J} = \{J_0, J_1, \dots, J_{n-1}\}$, where n is the number of subjobs). $\mathcal{E}$ is the set of directed edges between subjobs. There is a directed edge from subjob J_i to J_j (that is, $(J_i, J_j) \in \mathcal{E}$), if J_j requires J_i 's execution results as input. In this case, J_i is called J_j 's predecessor, and J_j is called J_i 's successor. *Data dependency* (denoted as “ $<$ ”) between two subjobs exists if there is a *path* (a concatenation of directed edges) between the subjobs. If $J_i < J_j$, then J_i is called J_j 's ancestor, and J_j is named as J_i 's offspring. The set of ancestors of J_i is denoted as $\mathcal{A}(J_i)$, and the set of offsprings of J_i is denoted as $\mathcal{O}(J_i)$. It is also assumed that a DAG representing a job workflow always has a unique starting node, J_0 , $J_0 \in \mathcal{A}(J_i)$ for any $J_i \in \mathcal{J}$; and a unique end node, J_{n-1} , $J_{n-1} \in \mathcal{O}(J_i)$ for any $J_i \in \mathcal{J}$.

Let $\mathcal{S}(J_i)$ denote the successor set of subjob J_i . $\forall J_{j_1}, J_{j_2} \in \mathcal{S}(J_i)$, J_{j_2} is called J_i 's *indirect successor* if $J_{j_1} < J_{j_2}$, otherwise, it is called J_i 's *immediate successor*. For example, in Figure 2.1, subjob J_2 is identified as an indirect successor of subjob J_0 since

Figure 2.1: An Example of Indirect Successors

subjobs $J_2, J_1 \in \mathcal{S}(J_0)$ and “ $J_1 < J_2$ ”. Meanwhile, J_1 is identified as an immediate successor of J_0 . Similarly, we can define *indirect predecessor* and *immediate predecessor*.

2.2.2 A Taxonomy of Job Workflow Execution Models

During a job workflow execution, the output files of a subjob may be required by its successors. Thus, such output files, also referred to as *intermediate results*, need to be transferred to the selected computational resource for the successor before the successor can start its execution. The subjob intermediate result, the subjob specific code and the control thread (see definition of the control thread in Section 1.1.2) are the three major elements involved in a job workflow execution. In this section, we study job workflow execution approaches by developing a taxonomy for them according to *intermediate result movement*, *subjob code acquisition*, and *control management*, and map the taxonomy to the existing job workflow management systems.

Intermediate Result Movement

Automation is one of objectives of the job workflow execution, therefore, only approaches that automate the movement of the intermediate results are considered. These can be classified as *centralized*, *mediated*, and *peer-to-peer* [123], as illustrated in Figure 2.2.

For the centralized approach, a central server relays the intermediate results between

Figure 2.2: Intermediate Result Movement

subjob codes. After a subjob completes its execution, its result is transferred to the central server, who will transfer the result to the selected computational resources for the subjob's successors. It is obvious that this approach produces data movement around the central server and makes it the bottleneck of the system.

For the mediated approach, the locations of the intermediate results are managed by a distributed data management system. After a subjob completes its execution, its result is registered in the data management system. The subjob's successors can obtain the location of the data via querying the data management system, then the successors can get the result using data transfer protocols (e.g., Gridftp). For both centralized and mediated approaches, the intermediate results can be shared by other job workflow applications and be monitored by users easily. However, the mediated approach is more scalable than the centralized approach.

For the peer-to-peer approach, the location of a subjob execution result or the result itself is transferred directly to its successor's computational resource. Unlike the centralized approach and mediated approach, no third party service is involved. Thus it can significantly reduce the data transmission or reduce the control message transmission. However, this approach requires the selected computational resources provide both data management and movement services.

Peer-to-Peer intermediate result movement can be carried out either concurrently with subjob execution, e.g. using *data streaming* technology [32, 34], or after subjob execution

by *post-processing*.

- **Data streaming:** Transmission of the predecessor's execution results is overlapped with the execution of the predecessor and/or the successor.
- **Post-processing:** Results of the subjob execution are written to local disk and are transferred to the destination resource after the subjob execution.

By overlapping the result transfer with subjob execution, the data streaming technique can shorten the job workflow makespan. However, this technique may not always be more effective than the post-processing technique. A subjob may require intermediate results from multiple predecessors. In a dynamic Grid environment, those predecessors may not be scheduled to execute at the same time and thus the execution of the subjob and the transfer of the results of its predecessors may not be overlapped. In the case where the execution resource of a subjob can only be scheduled after all its predecessors complete their execution, the data streaming technique cannot be used.

Subjob Code Acquisition

The subjob execution can be enacted through invoking the published service (e.g., web services) or through running codes provided by the users. The published service (or service for short) is the encapsulated functionality that hides the details of implementation and provides APIs for use. The service deployment is the process of instantiating the service into a computational resource. The service deployment can be location dependent or location independent. The deployment is location dependent when it is bound to a certain computational resource, assuming that resource has the corresponding executable code. The deployment is location independent when a service can be downloaded and deployed on any computational resource on demand at runtime, assuming that the corresponding executable code is stored at some resources acting as code servers. The services deployed

in the location dependent way are called *static services*, while the services deployed in the location independent manner are named as *dynamic services* [78].

Dynamic service is promising for data intensive scientific computations, since it allows a service to be deployed on the computational resource near to where the the required data set is located, therefore reducing the data transmission over the network. For the purpose of being discovered and integrated at runtime, a dynamic service has at least the following information:

- **Functional description:** This describes what is actually provided by the service. It can be specified using semantic web services technologies (e.g., OWL-S).
- **Function definition:** This provides the information for the service integration or invocation, e.g., the function name, the input data and output data;
- **Function code and its annotation:** This gives information about how a service works. For a given functional description, there can exist multiple function codes, which have different implementation details (e.g., implementation language and estimated execution time) and computing resource dependencies, as indicated in its associated annotation. The annotation can be used for runtime code selection so that the selected code can match the configuration of the selected computational resource.

Subjob codes can also be provided by users. Some workflow management systems, like Condor DAGMan [4], allow users to dynamically stage in codes to remote computational resources for execution. Similar to dynamic services, dynamic remote codes staging in allows codes to be executed near to where the data is located.

To summarize, the classification of the approaches for the subjob code acquisition is illustrated in Figure 2.3, where U_DS represents that the subjob code is provided by the

users.

Figure 2.3: Subjob Code Acquisition

Dynamic and distributed job workflow execution raises the following desirable requirements for subjob codes:

- **Reusability:** Scientists share their analysis modules over the Grid to reduce the duplicated development efforts.
- **Substitutability:** With Grid resources' heterogeneity and various computing resources provisioning, e.g., specific library codes, alternative implementation with equal functionality should be easily achieved.
- **Location independent instantiation:** As stated in the beginning of this Chapter, the size of required codes are assumed to be less than that of the required data sets. Therefore, when codes can be transferred to and instantiated near to where data is located, the data transmission over the network can be reduced. In addition, dynamic location independent code instantiation can also further reduce unnecessary intermediate result movement.
- **Security:** Considering a malicious code can do damage to a computational resource, the subjob code should be secure. First of all, the code itself should be trusted or from a trusted third party.

Based on these requirements, an evaluation of different code acquisition approaches is described in Table 2.2. When U_DS is applied, the executable codes are provided by the user. The executable codes are not exposed as public services to be shared by other users. Meanwhile, it is difficult for the user to provide all possible alternative implementations for a given functionality. Therefore, the “Reusability” and “Substitutability” of U_DS are “N”. The comparison results show that the code acquisition approach based on the dynamic service is more promising for distributed job workflow execution.

Table 2.2: Evaluation for Codes Acquired by Different Approaches

Method	Reusability	Substitutability	Location Independent Instantiation	Security
U_DS	N	N	Y	N
Static Service	Y	Y	N	Y
Dynamic Service	Y	Y	Y	Y ^a

^aAssuming that the codes for dynamic services are provided in trusted code repositories.

However, acquiring codes from dynamic services creates other runtime overhead: additional time for code discovery and transmission over the network and dynamic instantiation. To characterize the benefit/cost between models, an experimental comparison is to be carried out in Section 2.3 and Chapter 7.

Control Management

As presented in Section 1.1.2, according to how many control threads are used and where the control threads are executed, the job workflow execution can be classified into centralized execution or distributed execution. That is, the job workflow control management can be classified into a *centralized approach* or *distributed approach*.

When a centralized approach is applied, the control thread is fixed on a certain computational resource. The control thread handles all the control messages during the job workflow execution. For example, the engine in Business Process Execution Language for Web Services (BPEL4WS) [21] uses a typical centralized model, which requires the engine mediate at each step of the subjob execution [66] (i.e., it requires control messages to be transferred over the network for each subjob execution). This causes problems such as poor performance, single point of failure, and poor scalability.

Figure 2.4: Control Management

As stated in Section 1.1.2, there may exist multiple control threads for a job workflow execution when distributed execution is applied. The distributed execution can optimize a job workflow execution in two ways: (i) Reducing the number of control messages by transferring them directly between corresponding subjob executions (in contrast, in the centralized model, every message is relayed by the centralized engine); and (ii) Concurrently processing control messages for data independent subjobs. According to how the control thread is implemented, the distributed model can be further classified into two categories: *collaborative engine based* and *mobile agent based*.

When the collaborative engine based approach is used [25, 49, 119], the subjobs in a job workflow execution are not controlled by a centralized engine, but by multiple distributed

engines collaboratively. These engines are instantiated on distributed computational resources and share the same job workflow specification. They select the resources and control the executions for the subjobs for which they are responsible independently and collaboratively.

Mobile agent based job workflow execution is another paradigm for distributed job workflow execution, e.g., in BioAgent [13], an MA is used to gather the information about the gene expression. Because of the agent's autonomy, pro-activeness, and social ability, multi-agent based workflow enactment is believed to be adaptive to an unforeseen environment like the Grid [41]. As discussed in Section 2.1, according to what codes are carried during the MA migration, the MAs in the current mobile agent systems can be classified into MMAs and LMAs. The MMA supported job workflow execution does not scale well when the size of subjob codes get larger. By reducing the volume of codes transferred over the network, an LMA supported job workflow execution model is proposed [60], in which only the workflow specification is transferred, agents are instantiated on demand.

To summarize, the classification of the control management is illustrated in Figure 2.4. The distributed approach scales better than the centralized one. But the distributed approach also has additional runtime overhead. For example, when the LMA based approach is applied, there will be additional time for the LMA migration and dynamic agent instantiation .

Taxonomy of Execution Models

The classification of the three elements, i.e., intermediate data movement, subjob code acquisition, and control management, as discussed in previous subsections, can be used to develop a three-dimensional model, as shown in Figure 2.5. Each of the three elements represents one dimension in the model, showing a sub-taxonomy for that dimension.

SS: Static Service DS: Dynamic Service
 U_DS: Users provide code for dynamic staging in

Figure 2.5: A Taxonomy of Job Workflow Execution Models

Table 2.3: Taxonomy Mapping to Existing Job Workflow Management Systems

Project	Control Thread Deployment		Subjob Code Acquisition			Intermediate Data Movement		
	Centralized	Distributed	SS	DS	U_DS	P2P	Mediated	Centralized
DAGMan [4] Chimera [23] Pegasus [54]	✓				✓	✓ ^a		
Triana [106]		✓	✓		✓	✓		
ICENI [90]	✓		✓				✓	
Taverna [93]		✓	✓					✓
GrADS [31]	✓		✓		✓	✓		
GridFlow [45]	✓		✓			✓		
UNICORE [19]	✓		✓				✓	
Gridbus workflow [122]	✓		✓				✓	✓
Askalon [59]		✓	✓		✓			✓
Karajan [84]	✓		✓		✓			✓
Kepler [87]	✓		✓			✓	✓	✓
SwinDeW [119]		✓	✓			✓		
BioAgent [13]		✓			✓	✓		

^aDAGMan requires users to manage the intermediate data movement. P2P data movement can be implemented if data staging operations are specified accordingly in the workflow specification.

The three dimensions are orthogonal. That is, each of these three elements may be discussed independently of each other, and a sub-taxonomy in one dimension can, logically, have various combinations with the sub-taxonomies in the other dimensions. This model

introduces a taxonomy of job workflow execution models according to the different sub-taxonomy combinations. A string of format “XX-YY-ZZ” is used to express a model, where XX can be centralized or distributed; YY can be centralized, mediated, or P2P (Peer-to-Peer); and ZZ can be SS (static service), DS (dynamic service), or U_DS. Any job workflow execution model should have a specific value for each of the three parameters. Most combinations have plausible applications for the existing job workflow management systems, as illustrated in Table 2.3. More particularly, Table 2.4 shows the projects that employ distributed control management. As shown in the table, most of the projects adopt the collaborative engine based model. BioAgent is the only project that employs the mobile agent based model as far as we know.

Table 2.4: Distributed Control Management

Project	CE	MMA	LMA
Triana [106]	✓		
Taverna [93]	✓		
Askalon [59]	✓		
SwinDeW [119]	✓		
BioAgent [13]		✓	

2.3 Simulation Study

As discussed in the previous section, the distributed models avoid the single point of failure problem and reduce the number of control messages transmitted over the network. However, they result in more runtime overhead, especially for an MA based model. Although a dynamic service can be deployed near to the required data, thus reducing the data transmission over the network, it has also additional runtime overhead for code discovery, transmission over the network and computational resource discovery.

To characterize the cost/benefit tradeoff between models, a simulation study was

carried out to compare these execution models.

2.3.1 Grid Resources

Assuming that executable codes are kept in code repositories to provide dynamic services, Grid resources then include data repositories, computational resources, code repositories, and network resources.

For a certain data set required by a subjob, there may exist several data repositories having the data set, which are represented as a set $\mathcal{D} = \{d_0, \dots, d_{d-1}\}$. Similarly, for a certain executable code required by a subjob, multiple code repositories may have the code, which are represented as a set $\mathcal{C} = \{c_0, \dots, c_{c-1}\}$. For a certain subjob to be executed, there could be multiple computational resources satisfying the subjob's computation requirements, which are represented as a set $\mathcal{M} = \{m_0, \dots, m_{m-1}\}$. Grid resources are shared among different organizations, which may involve communications over the Internet. Let $\mathcal{N}$ be the connectivity matrix capturing network characteristics among nodes in $\mathcal{D}$, $\mathcal{M}$, and $\mathcal{C}$. A Grid can be modeled as $(\mathcal{D}, \mathcal{M}, \mathcal{C}, \mathcal{N})$ (see Figure 2.6 (b)).

Job workflow applications are composed of subjobs which require locating data and/or codes at runtime. There may be data dependencies among the subjobs. Job workflow execution is to map the application model $\mathcal{G} = (\mathcal{J}, \mathcal{E})$ to the Grid model $(\mathcal{D}, \mathcal{M}, \mathcal{C}, \mathcal{N})$, as shown in Figure 2.6.

Our objective is to simulate job workflow execution using various job workflow execution models for a specified job workflow application over the Grid, and compare the makespans of the job workflow executions.

Figure 2.6: Job Workflow Execution over the Grid

2.3.2 Assumptions and Parameters

As discussed in Section 1.2, in MA based distributed job workflow execution, it is necessary for the MA executing a subjob to locate the input data from its predecessors. When the subjob has multiple data independent subjobs as its predecessors, who are being executed simultaneously, execution coordination is needed. Different coordination techniques may have a different effect on system performance. In order to focus on the study of cost/benefits between execution models, we consider only pipeline job workflow, where $\mathcal{J} = \{J_0, \dots, J_{n-1}\}$, $\forall J_i \in \mathcal{J}$, J_i will be executed on h_i , $h_i \in H$, $H = \{h_0, \dots, h_{n-1}\}$ ($H \subseteq \mathcal{M}$). The simulation is to compare the makespan of the execution of the job workflow.

Table 2.5: Overhead Cost Assumptions

Parameter	Value
Agent Code Size	20 Kbytes
Dynamic Loading Time	1.085 sec [36]
Agent Serialization Time	0.002 ms/bytes [76]
Agent Installation Time	0.16 ms/ bytes [76]
Agent Creation Time	1.7 ms [77]
Agent Destroy Time	0.4 ms [77]

Overhead cost assumptions are given in Table 2.5. Serialization and deserialization

time increases as the agent size increases (provided that the agent code's call stack size is the same) [58]. We notice that overheads given in Table 2.5 are trivial compared to the time for data transmission and subjob execution time (e.g., subjob execution time ranges from several minutes to tens of CPU hours). Therefore they are not considered in the simulation.

2.3.3 Simulation of Networks

Grid resources are shared among different organizations over the network. The workflow execution over the Grid may involve multiple distributed resources, so the topology of the network may affect the execution performance.

Network Topologies

A network can be viewed as a collection of routing domains. A routing domain is a group of nodes (routers, switches and computational resources) with single administration for routing information and policy sharing. There are two kinds of routing domains: stub domain and transit domain. Stub domains are generally campus network or LANs, while transit domains are always WAN or metropolitan-area networks (MAN). Existing topology models include the random graph model, transit-stub model and tier model [44]. The nodes in the network are placed at random points in the random graph model, which cannot guarantee a connected network. The transit-stub model produces connected subgraphs, and there are two domains in the graph: the transit domain and stub domain. There are three levels of domain in the tiers model: WAN, MAN and LAN. The tier model also produces connected subgraphs. Since Grid resources are constructed hierarchically, the transit-stub model and the tier model are preferred. Here, GT-ITM created by Zegura et al [124] was chosen to construct the transit-stub model topology. In our experimental study, the transit-stub graph generated has 600 vertices and 1228 edges, with average

degree of 4.09.

Network Simulator

Network simulator (ns-2) is a discrete event simulator, which supports the simulation of TCP, network traffic, routing, and multicast protocols over wired and wireless (local and satellite) networks [11]. ns-2 was used to set the bandwidth and delay for the links generated by GT-ITM and to generate traffic on the topology to simulate the real network. Data and codes were transferred over the simulated network and the transmission time was calculated for different workflow execution models.

`sgb2hierns` [11] is used to convert the topology generated by GT-ITM to ns-2 hierarchical format. A modification is made on the file generated by `sgb2hierns` in the simulation study: bandwidth to the links is not set identically. Links on the Internet are classified into three categories: the WAN link, access link and stub link. A link is a WAN link when both of its vertices belong to transit domains. A link is an access link when one of its vertices belongs to a transit domain, and the other belongs to a stub domain. A link is a stub link when both of its vertices belong to the same stub domain. In our experiments, the bandwidth of a WAN link is set to 50 Mbps, the bandwidth of an access link is set to 5 Mbps, and the bandwidth of a stub link is set to 1 Mbps. The delay on the link is set according to the link distance.

Random traffic is generated on the networks described above. The possibility of a node being a constant bit rate (CBR) traffic generator is set to 0.8. When a CBR traffic generator belongs to a transit domain, the package size is set according to a normal distribution with average of 1500 bits and standard deviation of 200 bits. An access node is a stub node that has a single access link [86]. When a CBR traffic generator is an access node, the package size is set according to a normal distribution with average of 1000 bits and standard deviation of 100 bits. When a CBR traffic generator is a stub node

(excluding the access node), the package size is set according to a normal distribution with average of 300 bits and standard deviation of 100 bits. For each traffic generator, the package will be sent every 1 second.

2.3.4 Simulation of Job Workflow Models

A pipeline workflow application with 7 subjobs was assumed in the experiments. Except for subjob 0 and subjob 6, i.e., the starting node and the end node which do not have real computation, the input data of subjobs are from both a data repository and the previous subjob execution. The final result is returned to where the workflow is originally submitted. The data set volume from the data repository is assumed to be 3 times the volume of the output data generated by each subjob execution.

First, for each data set required by a subjob, there will be 5 replicas. So 25 nodes were chosen as the data repositories during the job workflow execution. Second, for each static service, there will also be 5 replicas, so another 25 nodes were selected as static services. Third, for each subjob code, there will be 10 replicas. So 10 nodes were selected as code repositories before execution and each of them contains all subjob codes. The selection was random and all the chosen nodes belong to stub domains and are not access nodes.

In the following experiments, the volume of the input/output data is set following the size of the medical image data in [68], which is in the order of MBytes. Initially it is set to 1 Mbyte. The operations described below for each workflow model are repeated by increasing input/output data volume by 1 Mbyte, until it reaches 7 Mbytes. The size of the agent code is set to be 20 Kbytes, following the range given in [109]. As reported in [85], a subjob specific code size can be assumed in the order of Kbytes. Therefore, the functional code size for subjob execution is initialized to be 20 Kbytes. The operation *get_package_size(sb)* checks whether special software or lib packages are

needed for the specified subjob sb . The size of a possibly required large library code is set according to the size of the library codes required for Aglets [83], a typical mobile agent library. It is set between 1 Mbyte and 5 Mbytes. Meanwhile, we notice that only 2 out of 35 ($2/35 = 0.057$) library codes for Aglets have the size between 1Mbyte to 5Mbytes. Therefore, the probability that a subjob execution needs such big additional supporting software is set to 0.05. The subjob execution time is set to be 3600 seconds, according to the one used in [56].

- Centralized-Centralized-SS model: For each subjob execution, the engine looks up the services and data repositories (TCP protocol is used for sending messages, and this is the same for the following models), selects the service that is closest to the engine, and the nearest data repository for the selected service. The engine then sends input data to the selected service, and the required data set is transmitted from the corresponding data repository to where the service is located (FTP protocol is used for data transmission, and this is the same for the following models). After the subjob completes its execution, the result is returned to the client. At the same time, the engine selects the service and data repository for next subjob execution. The same operations are repeated 5 times with different service computational resources and data repositories.
- Centralized-Centralized-DS model: For each subjob execution, the engine first looks up the data repository and selects the one that is nearest to the engine. For the selected data repository, a node belonging to the same stub as the data repository is then selected as the service computational resource. The selected node should be neither an access node nor a data repository node. During the selection, the traffic state of the link between the data repository and the service computational resource is considered. The code repository nearest to the selected computational resource is selected. The code volume for each subjob execution is set to (subjob service code

size + *get_package_size()*). Before each subjob begins its execution, the required codes are downloaded from the selected code repository. This operation is invoked at the same time as the data set being transferred from the data repository. Other operations are similar to those of the Centralized-Centralized-SS model.

- Distributed-P2P-U_DS model with MMA: An MMA moves from computational resource to computational resource carrying input data (the first subjob) or intermediate subjob execution results and all the executable codes. The code volume for each subjob execution is set to (subjob service code size + *get_package_size()*). The algorithm for computational resource selection is similar to the one used in the Centralized-Centralized-DS model, that is, a data repository that is closest to the current computational resource is first selected and a node belonging to the same stub as the selected data repository is then chosen as the computational resource for the next subjob. After all the subjobs are executed, the MMA returns to the user. Since in MMA subjob code is migrated with the agent and is dynamically staged to the resource, the subjob code acquisition in MMA can be classified as U_DS (i.e., users provide subjob code for dynamic staging in).
- Distributed-P2P-SS model with LMA: An LMA acts similarly to the MMA. The difference is that an LMA does not carry subjob execution codes on its migration. The specific subjob code is assumed to be provided as static services. In this model, LMA moves from computational resource to computational resource carrying input data (the first subjob) or intermediate subjob execution results. The algorithm for destination computational resource and data repository scheduling is the same as the one used for the Centralized-Centralized-SS model. After all the subjobs are executed, the LMA returns to the user.
- Distributed-P2P-DS model with LMA: In this model, the specific subjob code is

downloaded from the code repository and instantiated on demand. The code volume for each subjob execution is set to: $(\text{subjob service code size} + \text{get_package_size}())$. When client input data or intermediate results reach the destination computational resource, input data from the corresponding data repository and the corresponding subjob code is transferred to the destination host simultaneously. The resource scheduling policy is similar to that used in the Centralized-Centralized-DS model. Other operations are similar to the Distributed-P2P-U_DS model with MMA.

To further evaluate overheads of the models that use dynamic services, experiments were repeated for these models by increasing the subjob code size from 1 Kbyte to 3125 Kbytes, with output data volume and data set volume fixed at 3 Mbytes and 9 Mbytes respectively.

2.3.5 Simulation Result & Analysis

The simulation results are shown in Figure 2.7(a) and (b) respectively. Figure 2.7(a) shows that the makespans of the five execution models increase as the data volume increases. From this result, we can see that when the subjob codes are acquired by using static services, the makespan is always higher than that obtained using the models whose subjob codes are acquired by using dynamic services.

When the code size is increased, as shown in Figure 2.7(b), the makespan of Distributed-P2P-U_DS model with MMA increases faster than that of the Distributed-P2P-DS model with LMA. Furthermore, when the code size is larger than 625 Kbytes, the makespan of the Distributed-P2P-U_DS model with MMA is larger than that of the Centralized-Centralized-DS model. In our simulation, the subjob execution code may require additional supporting packages (the possibility is set to be 0.05, and the package size is set from 1 Mbyte to 5 Mbytes randomly.). From the results in Figure 2.7(b), we can see that when the code size increases, the makespan of the Distributed-P2P-U_DS model with

(a)

(b)

Figure 2.7: Makespans of Different Job Workflow Execution Models

MMA increases greatly correspondingly. However, the makespans of the Distributed-P2P-DS model with LMA and the Centralized-Centralized-DS model are less affected.

Based on the simulation results shown in both Figure 2.7(a) and (b), we can draw the conclusion that a distributed workflow execution model based on the Distributed-P2P-DS model with LMA has better performance than that of the other four models that were simulated, for the pipeline job workflow execution.

2.4 Summary

A taxonomy of job workflow execution models based on three elements, i.e., intermediate data movement, subjob code acquisition, and control management, is developed in this chapter. The taxonomy is mapped to the existing job workflow management systems. In addition, to evaluate the benefits and overhead of different execution models, we carried out a simulation study to compare the makespan of five execution models: Centralized-Centralized-SS, Distributed-P2P-SS model with LMA, Centralized-Centralized-DS, Distributed-P2P-U_DS model with MMA, and Distributed-P2P-DS model with LMA. This includes the simulation of resource lookup, resource scheduling, and data/code transferred over the network. Based on the simulation results, a job workflow execution using Distributed-P2P-DS model with LMA gets better performance than the other models simulated.

From Table 2.3, we can see that Triana, Taverna, Askalon and Swindew support distributed control management, meanwhile Triana and Askalon support U_DS subjob code acquisition. We believe that for the makespan of the job workflow execution, Triana and Askalon can have rival results to that achieved by the Distributed-P2P-DS model with LMA. However, both of them use a collaborative engine based distributed control model, as discussed in Section 1.1.2. LMA is more suitable for Grid computing because

of an agent's autonomy, pro-activeness, and social ability. In addition, as discussed in Section 2.2.2, compared to U_DS subjob code acquisition, having subjob code provided as dynamic services means that users do not need to provide the executable codes, thus reducing users' development effort. The Distributed-P2P-DS model with LMA is promising for distributed job workflow execution over the Grid. However, currently there does not exist a job workflow management system that supports this execution model. In the following chapters, a Mobile Code Collaboration Framework (MCCF) and mechanisms under the framework to support the Distributed-P2P-DS model with LMA for job workflow execution will be developed.

Chapter 3

Mobile Code Collaboration Framework (MCCF)

The results of the simulation study in Section 2.3 show that the Distributed-P2P-DS model with LMA gets better performance than the other four models simulated for pipeline job workflow execution. The analysis in Section 2.2.2 shows that currently there does not exist a job workflow management system that supports this execution model. This chapter describes the preliminary design, implementation, and evaluation of the Mobile Code Collaboration Framework (MCCF), which supports the Distributed-P2P-DS model with LMA based job workflow execution.

3.1 System Overview

Initially, the job workflow specification in the MCCF is abstract (i.e., the computational resources to execute subjobs, locations of data sets, and locations of code to be executed for subjobs are not specified). The objective of the MCCF is to map the abstract job workflow provided by users to the Grid resources dynamically for distributed job workflow execution, as shown in Figure 3.1. The resources here include computational resources, data repositories, and code repositories. Figure 3.1 (b) shows how the job workflow shown

Figure 3.1: The MCCF System Model

in Figure 3.1 (a) is executed dynamically over the MCCF.

An LMA approach is used in the MCCF for the purpose of separating the functional description and the executable code. The functional description is described using an *Agent Core* (AC). An AC is essentially an XML file, containing the job workflow specification and other necessary information for agent creation and execution. An AC can be migrated from one resource to another. As for the executable code, to separate subjob specific code and common non-functional code (i.e., the code for handling resource selection, subjob execution, agent communication, and AC migration), CoD is used in the MCCF, that is, subjob specific code is downloaded to the computational resource and executed on demand. The execution of common non-functional codes is carried out by a group of underlying *AC agents* (or agents in short). These agents are temporary and local to the computational resource on which they are running. They are constructed on the computational resource when an AC arrives and destructed when the AC is migrated

or discarded. The AC agents include *schedule agent* (as), *task agent* (at), *partner agent* (ap), and *coordinator agent* (ac).

An AC may be responsible for the execution of a group of, but not all, subjobs. Upon the completion of a subjob's execution, the AC will be updated with the location of the subjob's execution result before it is migrated to execute the next subjob. When multiple data independent subjobs can be executed concurrently, replicas of an existing AC will be generated so that there is one AC for each subjob. The ACs will then be migrated to the different computational resources for the execution of these data independent subjobs in parallel. When multiple concurrently executing subjobs have a common immediate successor, only one of the corresponding AC replicas should be selected for the latter's execution. Others should be discarded if they are not migrated to any successor's execution. How these replicas coordinate to consolidate their immediate results will be addressed in Chapter 5 and which replica is selected to continue will be addressed in Section 4.3.1.

Figure 3.2: The MCCF Architecture

3.2 System Design

The MCCF system design consists of *build time functions* and *runtime functions*. The build time functions include functions for the AC construction. The user uses a *job workflow editor* to generate the job workflow specification, which is processed by the *AC constructor* to generate an AC. The runtime functions consist of two parts: *pre-installed functions* and *dynamically generated functions*. The pre-installed functions are always running in a computational resource, and they are responsible for managing the migration of an AC (which is carried out by the *AC dispatcher*), receiving an AC and instantiating the corresponding AC agents (which are carried out by the *AC receiver*). The dynamically generated functions are carried out by the AC agents. The functions consist of subjob execution, subjob resource selection, execution coordination (which will be detailed in Chapter 4 and Chapter 5), provenance recording and collection (which will be detailed in Chapter 6). Grid services are used for subjob resource selection (e.g., MDS is used for resource information query), data/code movement and AC migration (Gridftp). Code repositories are used to provide subjob specific codes. The components for distributed job workflow execution in the MCCF and their relationship are illustrated in Figure 3.2, where the arrows represent communication interfaces between two components. Every computational resource on the Grid is assumed to be deployed with a job workflow engine which consists of AC Dispatcher, AC Receiver, and Globus Tool Kit.

3.2.1 Build Time Functions

Job Workflow Editor

In the MCCF, the job workflow specification generated from the job workflow editor is an abstract workflow. To simplify the implementation, for subjob code specification, we use a function name, instead of the functional description (refer to the discussion in

Figure 3.3: Schema for Job Workflow Specification

Section 2.2.2). A functional description calls for *high precision* and *high recall* [80], where high precision minimizes the delivery of irrelevant information and high recall ensures the delivery of the relevant information. This is not our research focus. The execution resources for each subjob are filled in by the AC agents at runtime. The format of the job workflow specification is defined in Figure 3.3. Mainly, a job workflow consists of subjobs, and the attributes and elements of each subjob are described as follows:

- **subjobId**: This is the id to identify the subjob.

- **ComputationalResource:** This specifies the computational resource for the sub-job execution.
- **policies:** This is the metadata information describing the user preferences for resource scheduling, e.g., minimum execution time, random, and so on.
- **dataLocations:** This specifies the required input data sets and their locations for each subjob. In addition, the specification here also describes the data dependencies among subjobs. There are three kinds of data location types (DLTs), as outlined in Table 3.1. The DLTs of a subjob can be changed during the course of the job workflow execution, which will be detailed in Sections 3.2.2 and 3.3.1.

Table 3.1: Data Location Types

DLT	Description
<i>static</i>	This is used to keep the data dependency between subjobs after the required data sets are located. Data dependencies between subjobs are part of the provenance information to be recorded and collected (Provenance recording and collection will be further detailed in Chapter 6).
<i>dynamic</i>	This is used to indicate the data dependency between subjobs. For example, if subjob J_i has a DLT specified as “dynamic” and the location is specified as J_j , that means one of J_i ’s input data is the output of subjob J_j , i.e., $J_j < J_i$.
<i>lookup</i>	This is use to indicate that the data set location needs to be discovered during runtime. When this DLT is used, the corresponding “location” field is set empty initially.

- **code:** This specifies the required codes and their locations. Similar to the data location type, there are two kinds of code location types (CLTs), as outlined in Table 3.2. Similar to DLTs, the CLTs of a subjob can also be changed during the course of the job workflow execution, which will be detailed in Section 3.2.2.

Table 3.2: Code Location Types

CLT	Description
<i>static</i>	The code location is specified in URL.
<i>lookup</i>	The code location needs to be discovered during runtime.

As an example, the initial specification of the job workflow $\mathcal{W}_a$ (as illustrated in Figure 2.6) is shown in Table 3.3. To save space, only the detailed specification for subjob 7 is illustrated. The computational resource for subjob 7 is not specified and it will be selected by the AC agents when it is ready to run. The policy for the resource selection for subjob 7 is “random”. There are five input data for subjob 7, the DLT of one input data is specified as “lookup” and its corresponding “location” has not been specified (i.e., its location will be determined by the AC agents at runtime). The other DLTs show that the execution of subjob 7 depends on the outputs of the execution results of subjob 0, 1, 2, and 4. Finally, the CLT for the executable code of subjob 7 is “lookup”, i.e., the location of the code will be determined by the AC agents during runtime.

AC Constructor

After a user submits his/her job workflow specification, the AC constructor is triggered to construct an AC accordingly. The AC created here is called the *original AC* for a job workflow. During a job workflow execution, multiple AC replicas may be created and discarded at runtime for subjob executions, however, only one AC, the original AC, will be returned back to the user after the job workflow completes its execution. As stated in Section 3.1, an AC is an XML file, whose format is shown in Figure 3.4. Its attributes are described as follows:

- “AC_GUID”: a unique id to identify the AC.

Table 3.3: An Example Job Workflow Specification

```

<?xml version="1.0" encoding="UTF-8"?>

<JobWorkflowSpec xmlns:JobWorkflowSpec="JobWorkflowSpecification">
  <JobWorkflow>
    <subjobs>
      ...
      <subjob subjobId="7">
        <policies policyId="1"><policy>random</policy></policies>
        <dataLocations>
          <dataLocation name="dependant0">
            <type>dynamic</type>
            <location>0</location>
          </dataLocation>
          <dataLocation name="dependant1">
            <type>dynamic</type>
            <location>1</location>
          </dataLocation>
          <dataLocation name="dependant2">
            <type>dynamic</type>
            <location>2</location>
          </dataLocation>
          <dataLocation name="dependant3">
            <type>dynamic</type>
            <location>4</location>
          </dataLocation>
          <dataLocation name="remoteDataSet7">
            <type>lookup</type>
          </dataLocation>
        </dataLocations>
        <code name="JobExec7.JobExec">
          <codeLocation>
            <type>lookup</type>
          </codeLocation>
        </code>
      </subjob>
      ...
    </subjobs>
  </JobWorkflow>
</JobWorkflowSpec>

```

Figure 3.4: Schema for Agent Core

- **“AC_createTime”**: the time when the AC is created. The time format is “hh-mm-ss”, where “hh” is the hour, “mm” is the minute, and “ss” is the second.
- **“Date”**: the date when the AC is created. The date format is “yy-mm-dd”, where “yy” is the year, “mm” is the month, and “dd” is the day.
- **“User”**: the name of the user.
- **“Signature”**: the signature of the user.
- **“JWId”**: the job workflow Id.

The elements in an AC are described as follows:

- **“ACOrientalHost”**: the computational resource to which the original AC is returned after the job workflow completes its execution. By default, it is specified

using the computational resource on which the original AC is constructed.

- **“JobWorkflowSpec”**: the job workflow specification.
- **“State”**: the execution state, where “currSubjobId” is the id of the subjob with which an AC is currently associated and which is ready to run. In the original AC, “currSubjobId” = 0.

3.2.2 Runtime Functions

AC Agents

The itinerary of a job workflow execution refers to the execution information of each subjob in the job workflow. Specifically, for each subjob, the itinerary keeps its detailed execution information, i.e., the locations of the input data sets, the code to be executed by the subjob, and the computational resources used to execute the subjob. As stated in Section 3.1, the AC agents include *coordinator agent*, *partner agent*, *task agent*, and *schedule agent*:

- *Coordinator Agent* starts the partner agent and makes sure that all the input data for its corresponding subjob are ready. Then, it starts the execution of the task agent and the schedule agent. After the task agent completes the subjob execution and the schedule agent selects the resources for the ready subjobs, the coordinator agent will perform AC disposal or contact the AC dispatcher agent for AC migration.
- *Partner Agent* is responsible for communicating with other AC agents and updates the AC’s itinerary to include the received subjobs’ execution information. It, together with the coordinator agent, is also responsible for execution coordination during dynamic job workflow execution (this will be detailed in Chapter 4 and Chapter 5).

- *Task Agent* is responsible for subjob execution. It will download input data for the corresponding subjob from the selected data repository and the computational resources where the subjob's predecessors were executed, download the executable code from the selected code repository, instantiate the code for subjob execution, and update the AC itinerary with the location of the subjob's execution result when it completes the execution.
- *Schedule Agent* is in charge of the dynamic resource scheduling for immediate successors according to the policies specified in the AC and updates its itinerary to include the newly selected execution information accordingly. Policies are the meta-data information for user preferences such as resource scheduling strategies (e.g., minimum execution time), security setting requirements and so on.

AC Receiver & AC Dispatcher

After a subjob completes its execution, the coordinator agent updates the state field of the AC and triggers the AC Dispatcher to migrate the AC to the next subjob's execution. The AC Receiver on the selected computational resource for the next subjob will manage the receiving process. Figure 3.5 illustrates the AC migrating and receiving process. When the AC dispatcher on the original computational resource receives a migration request, it will perform the following operations:

- **Serialize AC:** This serializes the AC so that it can be transported over the network.
- **Encode data:** This encodes the serialized AC into XML format streams for transmission.
- **Transfer data:** This establishes a network connection to the destination computational resource to deliver the encoded and serialized AC.

Figure 3.5: AC Migration

When the destination computational resource receives an AC, it will perform the following:

- **Receive data:** This receives the encoded and serialized AC.
- **Decode data:** This decodes the encoded AC.
- **Deserialize AC:** This deserializes the serialized AC and restores it to the original format.
- **Construct AC Agents:** This constructs the AC agents for the arriving AC accordingly.

3.3 Job Workflow Execution on the MCCF

Recall that the objective of the MCCF is to map the abstract job workflow specification to the Grid resources dynamically for distributed job workflow execution. This section describes the dynamic and distributed job workflow execution over the MCCF.

3.3.1 Dynamic Execution of Job Workflows

As stated in Section 3.2.1, the job workflow specification in the initial AC includes only the abstract job workflow specification, which specifies the job composition, dependency between subjobs and subjob execution policies. Because of Grid resources' dynamism, resources for a subjob execution will be selected at runtime when the subjob is ready to run. The resources for a subjob execution include the computational resource to execute the subjob, the locations of the required data sets, and the location of the code to be executed for the subjob. Particularly, when the required data sets for a subjob are located, its DLT is modified accordingly. For example, when a subjob completes its execution, its successors' corresponding DLTs are changed from “dynamic” to “static”, with the corresponding “location” filled with the location of the subjob's execution result.

The selection of the resources is based on the specified user policies, e.g., minimum execution time or random.

3.3.2 Distributed Job Workflow Execution

For the simplicity of description, notations used throughout the rest of the thesis are given below. The first AC created for executing a job workflow $\mathcal{W}_x$ is denoted as $C_{x,0}$. Considering the dynamic AC replica creation, AC replicas created during $\mathcal{W}_x$'s execution can be represented as $C_{x,0}, \dots, C_{x,n'-1}$, where n' is the number of AC replicas. The k th AC replica can then be uniquely identified using tuple (x, k) , $0 \leq k < n'$. Recall that agents are constructed on the computational resource when an AC arrives and destructed when an AC is migrated or discarded. One AC replica corresponds to only one group of agents at a particular instance. The group of agents created from $C_{x,k}$ to execute subjob J_i is denoted as $A_{J_i}^{x,k}$. The group of agents created for J_0 in $\mathcal{W}_a$ shown Figure 2.6 is illustrated in Figure 3.6(a), where rectangles represent computational resources and **as**, **at**, **ap** and

ac are used to represent schedule, task, partner and coordinator agents respectively.

Figure 3.6: Distributed Execution of Job Workflow $\mathcal{W}_a$

One possible scenario of the distributed execution of job workflow $\mathcal{W}_a$ is illustrated in Figure 3.6, where the AC migration is denoted using a dotted arrow line and the retrieval of an execution result from a predecessor is denoted using a solid arrow line. For example, after subjob 0 completes its execution, subjobs 1 and 2 are ready. A replica of $C_{a,0}$ needs to be created, denoted as $C_{a,1}$. These two AC replicas are then migrated to their selected computational resources for subjob execution simultaneously, as illustrated in Figure 3.6(b). The selection of the computational resource could be based on the load situation of resources or the cost of the computation, depending on the user preference specified in the policy.

Similarly, after subjobs 1 and 2 complete their executions, the resources for the execution of subjobs 3, 4, 5, and 7 are scheduled. Two new AC replicas, $C_{a,2}$ and $C_{a,3}$, need to be created again for the parallel execution of these four subjobs, as shown in Figure 3.6(c).

Subjobs 3 and 4 are executed in parallel on distributed computational resources. After their execution, subjob 6 is ready. In this case, only one AC should be migrated for subjob 6's execution, the other should be discarded. In general, when a subjob has multiple data independent subjobs as predecessors, an algorithm needs to be used to determine which AC will be migrated for the subjob's execution. In addition, it also calls for the mechanism for execution coordination. For example, subjob 6 needs to locate the execution results of its predecessors, subjob 3 and subjob 4, before it can be executed. These problems will be further investigated in Chapter 4 and Chapter 5. Here we assume that $C_{a,1}$ is migrated to the computational resource where subjob 6 is executed. Similarly, $C_{a,0}$ will be migrated to where subjob 8 is executed, as illustrated in Figure 3.6(d).

As mentioned above, when an AC replica arrives at a computational resource, a group of agents will be created. The coordinator agent starts the execution of the task agent only after all the input data for the corresponding subjob are ready. Thus, only after subjob 4

completes its execution can subjob 7 begin its execution, as illustrated in Figure 3.6(d). Subsequently, subjob 8 can only be executed after subjob 7 completes its execution (see Figure 3.6(e)).

After subjobs 6 and 8 complete their execution, $C_{a,0}$ will be migrated to where subjob 9 is executed, after which the final result will be returned to the user, as illustrated in Figure 3.6(f).

As illustrated above, during its lifetime, an AC replica may correspond to multiple groups of agents. However, at a specific time, an AC replica only corresponds to one group of agents which is responsible for executing a particular subjob. For example, during $\mathcal{W}_a$'s execution, $C_{a,0}$ corresponds to $A_0^{a,0}$, $A_2^{a,0}$, $A_5^{a,0}$, $A_8^{a,0}$, and $A_9^{a,0}$ at different instances of time.

3.4 Prototype

3.4.1 Multi-agent Platform

Because of the AC agents, the MCCF is first a multi-agent system. There must be an infrastructure for multiple agents to communicate. To facilitate agent communication, standards such as Foundations for Intelligent Physical Agents (FIPA) [5] and OMG Mobile Agent System Interoperability Facility (MASIF) [92] have been proposed. OMG MASIF aims at enabling agents to migrate between agent systems of the same profile via standardised CORBA IDL interfaces. The agent profile includes agent implementation language, agent system type, authentication type and serialisation methods. FIPA works on enabling the agent interoperability via standardized agent communication and content languages, and specifying ontology and negotiation protocols to support interoperability in specific application areas (e.g., travel assistance) [125].

Based on what was discussed in the previous sections, the multi-agent system sup-

porting the MCCF should have the following characteristics:

- **FIPA Compliant:** On the MCCF, only AC replicas are migrated between computational resources, therefore, MASIF compliance is not essential for the MCCF. Besides, MASIF is now out of date, and no active work has been undertaken on it since 2002. However, agents need to communicate for resource selection and execution coordination. Therefore, FIPA compliance is required.
- **Open Source:** With open source, the user will be supplied with the source code as well as the executable version of the software, and will not be restricted from modifying and distributing the source.
- **Document Availability:** Documents about APIs, examples, manuals, and tutorials can help the users get familiar with the software.
- **Support:** An active support community ready to answer users' questions concerning the software's deployment and development, e.g., an active mailing list archive, can further help users get familiar with the software. Meanwhile, this also implies that there is on-going effort to add more enhancement or features to the software.

There exist many environments for modelling and developing multi-agent systems [1, 6, 7, 8, 15, 16, 74, 83]. Several researchers have carried out evaluations and comparisons between the agent platforms [20, 67]. Their opinions are summarized in Table 3.4.

By comparing existing multi-agent systems according to FIPA compliancy, open source, document availability, and support, we can see that FIPA-OS, JADE, and ZEUS can work for the MCCF. We chose JADE.

Table 3.4: Agent Platform Evaluation & Comparison

Agent Platform	FIPA Compliant	Open Source	Document Availability	Support
Aglets [83]	N	Y	Y	Y
Ajanta [1]	N	N	Y	Y
Tryllian [15]	Y	N	Y	Y
FIPA-OS [6]	Y	Y	Y	Y
Grasshopper [74]	Y	N	N	N
JADE [8]	Y	Y	Y	Y
JACK [7]	Y	N	Y	Y
Zeus [16]	Y	Y	Y	Y

3.4.2 Prototype Implementation

We have built a simple prototype of the MCCF to illustrate the distributed job workflow execution over the Grid. Our implementation is built on J2ME (Java 2 MicroEdition) and JADE. Java is chosen as the implementation language for the following two reasons: it is platform-independent and its customized ClassLoader and inspection mechanism make dynamic class loading possible. Java CoG-1.1 [9] is utilized to access Grid services. Globus2.2 needs to be installed on each selected computational resource.

AC agents are implemented by extending the JADE agent class. For simplicity, resources for subjob execution are selected randomly at runtime. A simple code repository service is implemented as proof-of-concept. Java byte codes are provided at the specified directory on the host acting as the code repository. Data sets are stored as files, which are provided at the specified directory on the host acting as the data repository. Gridftp is used to download the codes, fetch the required data sets and transfer the AC replicas between the selected resources. MDS is used to provide information, for example, on workload information, OS of the destination resource, etc. The obtained information can be used for resource selection. GSI is used for authentication and authorization.

3.5 Summary

The explosion of the scientific data and dynamic nature of Grid resources pose great challenges to the existing job workflow execution models. The MCCF utilizes dynamic services and light-weight mobile agent technology to fulfill the new requirements:

- **Automated:** The user provides only the abstract job workflow specification. The subjob data dependency management, resource selection, and execution are taken care of by the workflow engine.
- **Flexibility:** The MCCF achieves flexible job workflow execution via: (i) Dynamic job workflow execution that adapts to the changes of the resources; and (ii) Executable codes that are provided in code repositories for dynamic download and instantiation. Therefore, when the selected computational resource lacks some library codes, they can be downloaded from code repositories and instantiated on demand. This reduces the requirement on the computational resources that can be used to execute the subjobs.
- **Scalability:** Distributed job workflow execution removes the single point of failure problem of the centralized execution model.
- **Easing Application Development:** In concept, the MCCF provides a two-level job workflow program model: component developers develop executable code for dynamic services and application developers just need to specify the input data and code description for subjob execution and the data dependency among the subjobs. This will greatly reduce the efforts for the application developers to “gridify” their applications.

In the following chapters, we develop mechanisms used by the MCCF to support: (i) parallel execution of data independent subjobs (see Chapter 4); (ii) execution coordina-

tion between data dependent subjobs (see Chapter 4 and Chapter 5); and (iii) provenance recording and collection (see Chapter 6).

Chapter 4

Dynamic Partner Identification in Mobile Agent based Distributed Job Workflow Execution

To reduce the communication overhead caused by data movement and to provide decentralized control of execution during the workflow enactment, the Mobile Code Collaboration Framework (MCCF) was developed in the last chapter to map the execution of subjobs to the distributed resources and to coordinate the subjobs' execution at runtime according to the abstract workflow provided by users. Light-weight Mobile Agent (LMA) [37] and Code-on-Demand (CoD) [65] techniques are adopted in the development of the MCCF, so that an analysis module in data intensive scientific applications can be executed at a computational resource close to where the required data set is located.

During a job workflow's execution in the MCCF, it is necessary for the agents executing a subjob to locate the input data from its predecessors. If the subjob's predecessor is executed by the same AC, the location of the input data from its predecessor would be readily available in the AC. Otherwise, the agents executing the subjob need to get the location of its predecessor's execution result through an execution coordination mechanism.

Execution coordination of mobile agents is a well researched area. There are many techniques that have been developed. Section 4.1 gives a classification of these techniques according to how they can be applied in the mobile agent-based distributed job workflow execution. The discussion in Section 4.1 shows that only direct coordination techniques will be suitable for the MCCF. When a direct coordination technique is used, agents need to obtain the location of the subjob's execution result through communication. Two agents communicating with each other are called *communication partners* or *partners* in short. Similarly, two AC replicas are partners if agents created from them are partners. This chapter is concerned with the design, implementation and evaluation of algorithms for communication partner identification in mobile agent-based distributed job workflow execution.

4.1 Classification of Execution Coordination Techniques

According to how a subjob obtains the locations of its predecessors' execution results, the execution coordination techniques for mobile agents (MAs) can be classified into *indirect* and *direct* coordination. In indirect coordination, an agent interacts with a shared memory for subjob result sharing [42]. There is no explicit communication between two mobile agents executing subjobs. That is, a subjob obtains the locations of its predecessors' execution results through accessing the shared memory. Indirect coordination can be further categorized into *co-residing coordination* and *global shared memory based coordination*.

When the co-residing coordination technique is used, two MAs interact only when they co-reside on the same host. MA interaction is performed through a shared local memory associated with the hosting environment (e.g., *blackboard based* and *meeting oriented* techniques described in [43]). This kind of coordination technique is fully decentralized.

However, it may not be suitable for data-intensive computations. During the course of a job workflow execution, an MA may need to communicate with many other MAs for the locations of the required subjob execution results, especially when a subjob has a large set of predecessors. Compared to the direct coordination techniques (to be discussed later), requiring two distributed MAs to move to the same host for coordination may introduce more overhead for the job workflow execution.

Global shared memory based coordination can be classified into *middle agent based coordination* [81] (which is also named as *client/server based coordination*) and *Linda like coordination* [43]. When these coordination techniques are applied, after a subjob completes its execution, the location of its result needs to be published in a global shared memory. The subjob's successors can then obtain the location through accessing the shared memory. For example, when the middle agent based model is used, after a subjob completes its execution, the corresponding MA publishes the location of the subjobs' execution results through a *middle agent*. When a subjob, whose input data depends on other subjobs' execution results, is ready, its corresponding MA sends a request message to the middle agent for the location of these subjobs' execution results.

When a direct coordination technique is used, mobile agents interact with each other directly or through the resources that belong to the interacting mobile agents (e.g., mailbox) [42]. An explicit communication between agents or their resources is initiated (i.e., the mobile agent must explicitly name its partner before communication takes place). After a subjob completes its execution, its corresponding MA will notify partner MAs the location of the subjob's execution result, so that the subjob's successors can locate the result. Depending on the mechanism for locating communication partners, existing direct coordination techniques are classified into *centralized lookup infrastructure based coordination* and *decentralized coordination*.

In centralized lookup infrastructure based coordination, a centralized server is used

for MAs' location update and retrieval. An MA keeps known partners' location information. It contacts the partners using the known locations first, and contacts the centralized server for the up to date partner location only when it finds that its partner's location is out of date (e.g., when the partner is unreachable). According to how an MA locates known partners, this type of coordination technique can be classified into *home-based coordination* [92], *forwarding-based coordination* [48] and *hierarchical-based coordination* [104].

- The home based approach simply adopts a centralized discovery server to look up the agent's current location. The centralized discovery server is named as "home" and maintains a dynamic name and location database of the MAs.
- The forwarding based approach keeps a forwarding pointer to the next host of the MA's itinerary on the current host of the MA. Each sender has to know its communication partner's original host, on which the communication partner is created. A global server is usually used to maintain this information. The frequent updating of location caused by agent migration is eliminated in this method, which reduces traffic around the server.

Mailbox-based scheme [48] is an example of the forwarding based approach which equips each agent with a mailbox that buffers the messages sent to it. The mailbox itself is also a mobile object. But it is a reactive mobile object with less migration frequency than the MA and can be located on a different host to where the MA is located. The sender sends messages to the receiver's mailbox, and the mailbox forwards the message to MA based on the pull or push mechanism. A centralized server can be used to keep track of the location of the mailboxes. In this case, a direct communication is established between two agents through their mailboxes.

- The hierarchical based approach organizes the discovery servers in a hierarchical

fashion, e.g., a tree in which MAs are treated as leaves. Internal nodes of the tree maintain names of agents with their corresponding locations.

The decentralized coordination includes *flooding-based coordination* [69], *distributed hash table (DHT) based coordination* [75], and *contact list based coordination* [98]. When these coordination techniques are adopted, an MA locates its partners using either locally maintained information or a flooding technique.

- Using the flooding based approach, the sender broadcasts an agent discovery request to all of its neighbors, and then the discovery requests will be generated by these neighbors until the destination partner agent is located. This coordination technique generates excessively flooding messages which will degrade the overall system's performance considerably and cause a large amount of traffic over the network.
- Using the DHT based approach, a community is formed for each group of cooperative agents. An MA may keep track and maintain partial states of a global structure which represents the member locations of the community and thus an overlay network (e.g., a "k-ary search tree-like" overlay) can be formed among the MAs. No discovery infrastructure is involved. However, because the overlay network is jointly formed by all MAs, with dynamic MA replication, disposal, and migration, the maintenance of the DHT overlay is costly. In addition, the average number of hops required for sending a message in this approach is of the order of $\log n$ (n is the number of computational resources). This may in turn increase the time for message delivery.
- Using the contact list based approach, each MA maintains a list of all its partners' locations. Before an MA is migrated or discarded, it will notify its partners so that they can update their contact lists accordingly. In order to ensure the message is

received by the partners, the MA cannot be migrated until it gets all the acknowledgements from its partners. This may introduce considerable overhead for MA migration and thus prolong the job workflow execution.

Figure 4.1: Execution Coordination Techniques

To summarize, the classification of the execution coordination techniques is illustrated in Figure 4.1. As discussed above, co-residing coordination introduces considerable overhead for the cost associated with the job workflow execution. The global shared memory based coordination relies on a centralized server for execution coordination, which is prone to become the system bottleneck and cause a single point of failure. Thus, we only consider the direct coordination techniques for the MCCF.

4.2 Agent Communication

When a direct coordination technique is used, agents need to obtain the location of the subjob's execution result through communication. For agent communication, the first step is to identify corresponding partners, the second step is to locate (or discover) them, and the third step is to route the message to them. Previous work [48, 69, 75, 92, 98, 104]

on MA communication normally assumes that the partner is already known and focuses mainly on dynamic MA location/discovery and message routing. However, with the dynamic MA replication during the distributed job workflow execution in the MCCF, this assumption does not hold any more. Particularly, for a given data dependency between two subjobs, the partnership of agents may vary at runtime due to the dynamic characteristics of job workflow execution (e.g., dynamic AC replication and different computation time of subjobs). For example, for the data dependency “*subjob 1 < subjob 7*” in $\mathcal{W}_a$, the AC replication and migration is illustrated in Figure 3.6(c), there are two possible partnerships for $A_1^{a,1}$:

- Assume that subjob 1 finishes its execution before subjob 2. $A_1^{a,1}$ notifies $A_2^{a,0}$ the location of subjob 1’s execution result. This location is included in $C_{a,0}$ ’s itinerary and is in turn replicated in $C_{a,2}$ which is migrated to where subjob 7 is executed. In this case, the partner of $A_1^{a,1}$ is $A_2^{a,0}$.
- Assume that subjob 2 finishes its execution before subjob 1. After subjob 2 completes its execution, computation resources for executing subjobs 5 and 7 can be scheduled. A new AC replica of $C_{a,0}$, $C_{a,2}$, is created. In this case, $A_1^{a,1}$ communicates with $A_7^{a,2}$ directly about the location of subjob 1’s execution result.

So, the dynamism of job workflow execution gives rise to the requirement of partner identification. Each AC replica has a set of partners and each partner is represented by a tuple $((x, k), J_i)$, where J_i is the subjob currently being executed. Note that, $((x, k), J_i)$ can be used to uniquely identify the group of agents, $A_{J_i}^{x,k}$, that is responsible for J_i ’s execution. When an AC replica is created or discarded, its partner information in the partner agents’ partner sets need to be updated.

A simple solution for partner identification is to determine the partnership based on the static job workflow specification, so that only the contact information of the partners

need to be updated dynamically. A naive solution is based on the following observation: $\forall J_i, J_j \in \mathcal{W}_x$, if $\mathcal{O}(J_i) \cap \mathcal{O}(J_j) \neq \emptyset$, then $C_{x,k}$ and $C_{x,t}$ are partners, assuming agents executing J_i and J_j are created from $C_{x,k}$ and $C_{x,t}$ respectively. In this solution, for each AC replica, its partner set is predetermined and is kept unchanged during the job workflow execution. Agents only need to communicate to update contact information dynamically.

However, this solution results in communication partners that are not necessary at run-time:

- As shown in Figure 3.6(d), after subjobs 3 and 4 complete their executions, $C_{a,3}$ will be discarded. Agents created from $C_{a,3}$ and agents created from $C_{a,0}$ do not need to communicate. Hence, they are not partners. For the same reason, $C_{a,3}$ and $C_{a,2}$ are not partners either.
- Assume that subjob 4 finishes its execution before subjob 2 does. $A_1^{a,1}$ and $A_4^{a,1}$ will communicate with $A_2^{a,0}$ about the location of subjobs 2 and 5's execution result respectively. So, in this case, $C_{a,1}$ (which is used to create $A_1^{a,1}$ and $A_4^{a,1}$) and $C_{a,2}$ (which is used to create $A_7^{a,2}$) are not partners.

Obviously, for a direct coordination approach, the larger the size of the partner set, the more communication traffic is generated during the job workflow execution. The communications between redundant partners will result in unnecessary agent communications. Our objective is to design a solution for partner identification, which can dynamically determine the communication partners so as to include only necessary AC replicas in the partner set. Hence, the runtime overhead for contact information updating will be reduced.

4.3 Partner Identification

To include only necessary AC replicas in a partner set, our solution is to identify the number of AC replicas required and subjobs executed by each AC replica based on the analysis of the static job workflow, and use the information to determine the partnership during the job workflow execution dynamically.

Figure 4.2: Subjob Grouping for Job Workflow $\mathcal{W}_a$

4.3.1 Preprocessing – Subjob Grouping Algorithm

In this subsection, an algorithm, *subjob grouping algorithm*, is proposed to preprocess the static job workflow specification in order to: (i) identify subjobs that are to be executed by the agents created from the same AC replica; (ii) generate information for determining when and how to create AC replicas; (iii) generate information for deriving partners; (iv) generate information for the decision of which AC replica's agents will be responsible for the resource selection for a common immediate successor.

The subjob grouping algorithm performs the following steps (job workflow $\mathcal{W}_a$ shown in Figure 2.6(a) is used as an example to illustrate how the algorithm works):

1. **Assign subjob ids:** A topological ordering algorithm is applied on the DAG to generate a ordered sequence of subjobs. Then, ids from 0 to $n - 1$ are assigned to

the subjobs according to the ordered sequence. The subjob ids for job workflow $\mathcal{W}_a$ are shown in Figure 4.2(a).

2. **Remove indirect successors:** If J_j is the indirect successor of J_i , the corresponding edge (J_i, J_j) will be removed from the DAG. For example, subjob 7 is identified as an indirect successor of subjob 1 since subjobs $4, 7 \in \mathcal{S}(1)$ and “*subjob 4 < subjob 7*”. So, the edge $(1, 7)$ is removed from the DAG (see Figure 4.2(b)).
3. **Generate spanning tree:** A spanning tree is generated based on a depth-first search on the DAG. Here, we assume that the depth-first search and pre-order traversal algorithms visit child nodes in the order from right to left. The spanning tree generated for $\mathcal{W}_a$ is shown in Figure 4.2(c).
4. **Group subjobs:** A pre-order traversal is applied to the spanning tree to generate a visit sequence of nodes: $\langle J_{p_0}, J_{p_1}, \dots, J_{p_{n-1}} \rangle$ (i.e., a permutation of subjob ids). The leaves in this sequence will partition it into a number of groups. For example, the following visit sequence is generated for the spanning tree shown in Figure 4.2(c):

$$\langle 0, 2, 5, 8, 9, 7, 1, 4, 6, 3 \rangle$$

where 9, 7, 6 and 3 are leaves. Therefore, the subjob groups generated are $\{0, 2, 5, 8, 9\}$, $\{7\}$, $\{1, 4, 6\}$ and $\{3\}$ (that is, g_0 , g_1 , g_2 , and g_3 as illustrated in Figure 4.2(d)).

5. **Label edges:** Edges are labelled with “*m*”, “*c*” or “*d*”, as illustrated in Figure 4.2(d), where an edge labelled with “*m*” is denoted using a solid arrow line, an edge labelled with “*c*” a dark dashed arrow line, and an edge labelled with “*d*” a light dotted arrow line. The edges between two subjobs in the same group are labelled “*m*” (e.g., edge $(0, 2)$). The edges between subjobs of different groups are labelled “*c*” (e.g., edge $(0, 1)$). Finally, the edges in the original DAG but not in the spanning tree are labelled “*d*” (e.g., edge $(1, 7)$). The DAG with the labelled edges is also referred to as the *annotated DAG*.

All these preprocessed information are included in the AC and will be used during the dynamic job workflow execution. Suppose the current subjob is J_i and its corresponding AC replica is $C_{x,k}$:

- If an outgoing edge (J_i, J_j) is labelled with “ m ”, the AC agent $A_{J_i}^{x,k}$ will select the resources for executing J_j . The AC replica $C_{x,k}$ will be migrated for J_j ’s execution after J_i completes its execution. In this case, $C_{x,k}$ ’s itinerary will be updated with the location of J_i ’s execution result. Its communication partners will also be informed about this.
- If an outgoing edge (J_i, J_j) is labelled with “ c ”, similar to the last case, $A_{J_i}^{x,k}$ will select the resources for executing J_j and update $C_{x,k}$ ’s itinerary after J_i completes its execution, but a new AC replica will be created for J_j ’s execution.
- If an outgoing edge (J_i, J_j) is labelled with “ d ”, $A_{J_i}^{x,k}$ will notify its corresponding partners of the location of J_i ’s execution results. Moreover, if all outgoing edges of J_i are marked with “ d ”, $C_{x,k}$ will be discarded after its communication partners are notified with the location of J_i ’s execution result.

Effectively, all the subjobs in the same group will be executed by different groups of agents created from the same AC replica. Hence, the number of groups determines the number of AC replicas required to execute the corresponding job workflow.

4.3.2 Static Partner Identification Algorithm

Using the information obtained from preprocessing, a static partner identification algorithm is described in this subsection.

The annotated DAG $\mathcal{G} = (\mathcal{J}, \mathcal{E})$, such as the one shown in Figure 4.3(a), can be reduced to a graph $\mathcal{G}' = (\mathcal{J}', \mathcal{E}')$ according to the following rules:

Figure 4.3: Reduced Task Graph for Static Partner Identification

- Subjob groups in the original, annotated DAG (e.g., g_0, g_1 , etc., in Figure 4.3(a)) become the vertices in the reduced graph.
- There is a “c” edge in the reduced graph if there is a “c” edge in the original, annotated DAG between two subjob groups, and
- There is a “d” edge in the reduced graph if there is one or more “d” edges in the original, annotated DAG between two subjob groups.

According to the above rules, the annotated DAG in Figure 4.3(a) is reduced to the graph shown in Figure 4.3(b). The set of “c” edges in the reduced graph is represented by $\mathcal{E}'_c$, and the set of “d” edges in the reduced graph is represented by $\mathcal{E}'_d$.

For simplicity of description, let $\mathcal{P}$ represent the communication partner set. $(g_i, g_j) \in \mathcal{P}$ means that g_i and g_j are partner groups. Meanwhile, communication between partners is bidirectional. So, it is sufficient to have either (g_i, g_j) or (g_j, g_i) in the partner set $\mathcal{P}$. That is, AC replicas (and thus the agents created by these replicas) of the corresponding groups are communication partners. Also let $\mathcal{A}_c(g_i)$ represent a set of vertices so that for each vertex (e.g., g_t) in the set, there is a path from g_t to g_i and for every edge e in

the path, $e \in \mathcal{E}'_c$. The algorithm for preprocessing based static partner identification is presented in Figure 4.4.

```

Input:  Reduced Graph
Output:  $\mathcal{P}$ 
1)  $\mathcal{P} = \emptyset$ ;
2) for (each  $(g_i, g_j) \in \mathcal{E}'_d$ ) do {
3)   if  $((g_i, g_j) \notin \mathcal{P} \wedge (g_j, g_i) \notin \mathcal{P})$  then
4)      $\mathcal{P} = \mathcal{P} \cup \{(g_i, g_j)\}$ ;
5)    $g_k = \text{1stCommonAncestor}(g_i, g_j)$ ;
6)   for (each  $g_t \in ((\mathcal{A}_c(g_j) - \mathcal{A}_c(g_k)) \cup g_j)$ 
7)     and each  $g_{t'} \in ((\mathcal{A}_c(g_i) - \mathcal{A}_c(g_k)) \cup g_i)$  do
8)     if  $((g_t, g_{t'}) \notin \mathcal{P} \wedge (g_{t'}, g_t) \notin \mathcal{P})$  then
9)        $\mathcal{P} = \mathcal{P} \cup \{(g_t, g_{t'})\}$ ;
10) }
```

Figure 4.4: Preprocessing-based Static Partner Identification

Obviously, two subjob groups are partners if there is a “d” edge between them. So, for each $(g_i, g_j) \in \mathcal{E}'_d$ (e.g., (g_4, g_3) in Figure 4.3(b)), if the corresponding partner information is not found in $\mathcal{P}$, (g_i, g_j) is added into $\mathcal{P}$ (lines 3 and 4 in Figure 4.4). However, as described in Subsection 3.3.2, AC replicas are created dynamically. Hence, during the dynamic distributed execution of the job workflow, for instance, agents created by g_i ’s corresponding AC replica may become ready to communicate before the AC replica for group g_j is even created. In this case, an alternative partner for g_i needs to be identified. This task is carried out in lines 5 to 9 in Figure 4.4.

According to the subjob grouping algorithm described in the last subsection, for any two subjob groups, their corresponding AC replicas must be created from a common AC replica. So, in the reduced graph, for any g_i and g_j , there must exist g_k such that there are two non-overlapping paths of “c” edges, one from g_k to g_i and the other from g_k to g_j . g_k is referred to as the first common ancestor of g_i and g_j in the algorithm. For the

example shown in Figure 4.3(b), g_1 is the first common ancestor for g_4 and g_3 .

Assume that there is a “d” edge from g_i to g_j . We now consider the partner pairs, in addition to (g_i, g_j) , that need to be included in the partner set. AC replicas of subjob groups g_i and g_j are created through AC replicas of subjob groups in $\mathcal{A}_c(g_i)$ and $\mathcal{A}_c(g_j)$, respectively, in a particular sequence during job workflow execution. Since g_i and g_j share the common replica creation path till g_k , any subjob group in $\mathcal{A}_c(g_k)$ will not be considered. Intuitively, if there is a “d” edge from g_i to g_j , every subjob group $g_t \in \mathcal{A}_c(g_j) - \mathcal{A}_c(g_k)$ may be a potential partner of g_i or a subjob group in the set $\mathcal{A}_c(g_i) - \mathcal{A}_c(g_k)$ since i) the creation of g_j ’s AC replica can be at any stage when the agents created by g_i ’s corresponding AC replica becomes ready to communicate; and ii) a direct coordination approach is used to keep track of the partner location. g_i ’s actual partner will be known only during the runtime execution of the job workflow.

The partner lists obtained in terms of subjob groups for the reduced graph shown in Figure 4.3(b) are illustrated in Table 4.1. For example, $g_0 \in \mathcal{A}_c(g_1)$ is not a potential partner of g_4 . However, g_1 and g_2 , which are in the set $\mathcal{A}_c(g_3) - \mathcal{A}_c(g_1)$, are potential partners of g_4 .

Table 4.1: Partner List Obtained Using Preprocessing-based Static Partner Identification Algorithm for the Reduced Graph Shown in Figure 4.3(b)

Subjob Group	Partner List
g_0	(g_1, g_2)
g_1	(g_0, g_2, g_3, g_4)
g_2	(g_0, g_1, g_4)
g_3	(g_1, g_4)
g_4	(g_1, g_2, g_3)

Based on the algorithm described in Figure 4.4, some unnecessary communication partners in the “Naive” partner identification algorithm (see Subsection 4.2) can then be removed. For example, g_3 and g_2 in Figure 4.3 are not partners. They are identified

as partners using the “Naive” algorithm since they have a common offspring (that is, g_0). However, they are not included in the partner set if the preprocessing based static partner identification algorithm described above is used.

4.3.3 Dynamic Partner Identification Algorithm

The preprocessing-based static partner identification algorithm is superior to the “Naive” algorithm. However, there still exist unnecessary partners in the communication partner set. For example, in Figure 4.3, g_1 , g_2 , and g_3 have been identified as possible partners of g_4 using the static partner identification algorithm, though agents created by g_4 ’s AC replica only need to communicate with agents of either g_1 , g_2 , or g_3 . Information regarding which group is the actual partner of g_4 will be known only during the runtime. In this section, a runtime partner set updating algorithm is described. Its correctness and the effectiveness, in terms of optimality, will be proven in the next subsection.

A job workflow $\mathcal{W}_x$ can be represented as $\mathcal{G} = (\mathcal{J}, \mathcal{E})$ as described in Section 2.2.1. Assume $\mathcal{E}_d$ is a set of edges that are marked with “ d ” in the annotated DAG corresponding to $\mathcal{G}$. Also assume that in the spanning tree generated by the subjob grouping algorithm described in Subsection 4.3.1, a sub-tree rooted at J_i is denoted as $\mathcal{T}_{J_i}$.

The partner identification is based on the following principle (which will be formally proven in the next subsection): in general, suppose J_i and J_j are two subjobs of $\mathcal{W}_x$ that are currently under execution, and that their corresponding AC replicas are $C_{x,k}$ and $C_{x,t}$ respectively. If $\exists (J_{i'}, J_{j'}) \in \mathcal{E}_d$, and $J_{i'} \in \mathcal{T}_{J_i} \wedge J_{j'} \in \mathcal{T}_{J_j}$, denoted as $\mathcal{T}_{J_i} \xrightarrow{d} \mathcal{T}_{J_j}$, then $C_{x,k}$ and $C_{x,t}$ are partners.

During the job workflow execution, the partner set needs to be dynamically updated due to (i) the creation of new AC replicas, and (ii) the disposal of existing AC replicas. When new replicas are created, new partners may need to be added and existing partners

may need to be deleted. When an AC is discarded, the corresponding entry in its partners' partner sets need to be removed.

Suppose the current subjob is J_i and its corresponding AC replica is $C_{x,k}$, and that there are m outgoing edges in the spanning tree. According to m 's value, we have:

1. $m > 0$: The m outgoing edges are $(J_i, J_{j_0}), \dots, (J_i, J_{j_{m-2}}), (J_i, J_{j_{m-1}})$. Assume $(J_i, J_{j_{m-1}})$ is marked with "m" and the rest with "c". So, $m - 1$ new replicas $C_{x,t_0}, \dots, C_{x,t_{m-2}}$ need to be created for the execution of subjobs $J_{j_0}, \dots, J_{j_{m-2}}$ respectively. $C_{x,k}$ will be used to execute subjob $J_{j_{m-1}}$. The rules for dynamic partner set update are described as follows (part of the update of the partner set during the execution of $\mathcal{W}_a$ is illustrated in Figure 4.5):

Figure 4.5: Update of Partner Sets during the Execution of Job Workflow $\mathcal{W}_a$

- Determine partnership between $C_{x,k}$ and its replicas:** if $\exists \mathcal{T}_{j_{t_l}}, 0 \leq l \leq m - 2$, and $\mathcal{T}_{j_{j_l}} \xrightarrow{d} \mathcal{T}_{j_{j_{m-1}}}$ or $\mathcal{T}_{j_{j_{m-1}}} \xrightarrow{d} \mathcal{T}_{j_{j_l}}$, then $C_{x,k}$ and C_{x,t_l} are partners. Corresponding entries need to be added to the partner set of $C_{x,k}$ and C_{x,t_l} respectively. For example, in Figures 4.5(a) and 4.5(b), after subjob 0 completes

its execution, a replica of $C_{a,0}$, $C_{a,1}$, is created so that subjobs 1 and 2 can be executed concurrently. $C_{a,0}$ and $C_{a,1}$ are partners since $\mathcal{T}_1 \xrightarrow{d} \mathcal{T}_2$. Similarly, after subjobs 1 and 2 complete their execution, new AC replicas need to be created from $C_{a,0}$ and $C_{a,1}$ respectively. In Figure 4.5(c), $C_{a,0}$ and $C_{a,2}$, $C_{a,1}$ and $C_{a,3}$ become partners.

- (b) **Determine partnership between $C_{x,k}$'s partner and its replicas:** For any communication partner of $C_{x,k}$, that is, $\forall((x,p), J_q) \in C_{x,k}$'s partner set, if $\exists \mathcal{T}_{J_{j_l}}, 0 \leq l \leq m-2$, and $\mathcal{T}_{J_q} \xrightarrow{d} \mathcal{T}_{J_{j_l}}$ or $\mathcal{T}_{J_{j_l}} \xrightarrow{d} \mathcal{T}_{J_q}$, then $C_{x,p}$ and C_{x,t_l} are partners. Corresponding entries need to be added to the partner set of $C_{x,p}$ and C_{x,t_l} respectively. For example, refer to Figures 4.5(c), $C_{a,2}$ is replicated from $C_{a,0}$ for the execution of subjob 7 and $((a,1),4) \in C_{a,0}$'s partner set. Since $\mathcal{T}_4 \xrightarrow{d} \mathcal{T}_7$, $C_{a,2}$ and $C_{a,1}$ are partners. However, this is not true for $C_{a,3}$ and $C_{a,0}$, although $C_{a,3}$ is a replica of $C_{a,1}$ and $((a,0),5) \in C_{a,1}$'s partner set. There does not exist an edge marked with “ d ” between $\mathcal{T}_3$ and $\mathcal{T}_5$ or $\mathcal{T}_5$ and $\mathcal{T}_3$.
- (c) **Identify partners between replicas:** $\forall \mathcal{T}_{J_{j_{l_1}}}, \mathcal{T}_{J_{j_{l_2}}}$, where $0 \leq l_1, l_2 \leq m-2$, if $\mathcal{T}_{J_{j_{l_1}}} \xrightarrow{d} \mathcal{T}_{J_{j_{l_2}}}$ or $\mathcal{T}_{J_{j_{l_2}}} \xrightarrow{d} \mathcal{T}_{J_{j_{l_1}}}$, then $C_{x,t_{l_1}}$ and $C_{x,t_{l_2}}$ are partners. Corresponding entries need to be added to the partner set of $C_{x,t_{l_1}}$ and $C_{x,t_{l_2}}$ respectively.
- (d) **Delete obsolete partners for $C_{x,k}$:** $\forall((x,p), J_q) \in C_{x,k}$'s partner set, if neither $\mathcal{T}_{J_q} \xrightarrow{d} \mathcal{T}_{J_{j_{m-1}}}$ nor $\mathcal{T}_{J_{j_{m-1}}} \xrightarrow{d} \mathcal{T}_{J_q}$ holds anymore, then $C_{x,p}$ and $C_{x,k}$ are no longer partners. Corresponding entries in their partner sets need to be removed. For example, after subjob 4 completes its execution, $C_{a,1}$ and $C_{a,2}$ are no longer partners, as illustrated in Figure 4.5(d).

2. $m = 0$: J_i is a leaf node in the spanning tree, that is, all outgoing edges of J_i , if any, are marked with “ d ”. AC replica $C_{x,k}$ will be discarded in this case. Accordingly, the tuple corresponding to $C_{x,k}$ needs to be removed from its partners' partner sets.

Note that the rules described above can be applied to identify partnership between AC replicas in any dynamic situation.

4.3.4 Correctness and Optimality

In this subsection, we prove the correctness and the effectiveness, in terms of optimality, of our runtime partner set updating algorithm. The correctness means for any subjob in the job workflow, it is able to locate results generated by its predecessors as its inputs. Optimality means at any time of the job workflow execution, there is no redundant partner in an AC's partner set.

At a specific time T , let $\mathcal{R}(C_{x,k}, T)$ represent the set of AC replicas to be created from $C_{x,k}$, including $C_{x,k}$ itself. For example, in Figure 4.5, when subjob 0 is being executed, the set of AC replicas to be created from $C_{a,0}$ is: $\{C_{a,0}, C_{a,1}, C_{a,2}, C_{a,3}\}$. When subjob 2 is being executed, the set of AC replicas to be created from $C_{a,0}$, at this time instance, is $\{C_{a,0}, C_{a,2}\}$. We have:

Lemma 4.3.1. *If at time T the corresponding AC replica for executing subjob J_i is $C_{x,k}$, then $\forall J_j \in \mathcal{T}_{J_i}$, J_j will be executed by an AC replica $C_{x,t} \in \mathcal{R}(C_{x,k}, T)$ and J_j will be able to locate J_i 's execution results.*

Proof: Based on the subjob grouping algorithm and the definition of $\mathcal{R}$, it is obvious that if $J_j \in \mathcal{T}_{J_i}$, J_j will be executed by an AC replica $C_{x,t} \in \mathcal{R}(C_{x,k}, T)$. Since $C_{x,t} \in \mathcal{R}(C_{x,k}, T)$, $C_{x,t}$ will inherit $C_{x,k}$'s itinerary when it is created and thus J_j should be able to locate J_i 's execution results. #

Theorem 4.3.1 (Correctness). *If there is a direct edge from subjob J_i to subjob J_j (that is, $\exists(J_i, J_j) \in \mathcal{E}$), the agents that execute subjob J_j will be able to locate results generated by subjob J_i .*

Figure 4.6: Job Workflow Execution Scenarios

Proof: Assume that at time T , J_i is being executed by AC replica $C_{x,k}$. If J_j will be executed by an AC replica $C_{x,t} \in \mathcal{R}(C_{x,k}, T)$, according to Lemma 4.3.1, J_j will be able to locate J_i 's execution results. If this is not the case, according to our subjob grouping algorithm, J_i and J_j must have a common ancestor and the edge (J_i, J_j) must be marked with “ d ”. So, the following situation happens at a certain stage of job workflow execution: an AC has to replicate itself, replicas (e.g., $C_{x,k'}$ and $C_{x,t'}$) are migrated to execute different subjobs (e.g., $J_{i'}$ and $J_{j'}$) respectively, and $J_i \in \mathcal{T}_{J_{i'}}$ and $J_j \in \mathcal{T}_{J_{j'}}$. The situation is depicted in Figure 4.6(a). According to our runtime partner set updating rules 1(a) and 1(c), $C_{x,k'}$ and $C_{x,t'}$ are partners. This situation will eventually evolve to one of the following two cases at a specific time T' ($T' > T$):

- J_i and $J_{j''}$ are being executed, $C_{x,k}$ and $C_{x,t''}$ are the corresponding AC replicas, and $J_j \in \mathcal{T}_{J_{j''}}$ (see Figure 4.6(b)). Our runtime partner set updating rule 1(b) guarantees that $C_{x,k}$ and $C_{x,t''}$ become communication partners during the evolution from Figure 4.6(a) to Figure 4.6(b) at time T' . Thus, $C_{x,t''}$'s itinerary will contain the location of J_i 's execution results. According to Lemma 4.3.1, J_j will be executed by an AC replica $C_{x,t} \in \mathcal{R}(C_{x,t''}, T)$. Since $C_{x,t} \in \mathcal{R}(C_{x,t''}, T)$, $C_{x,t}$ will inherit $C_{x,t''}$'s itinerary and thus J_j will be able to locate J_i 's execution results.
- J_i and J_j are being executed, $C_{x,k}$ and $C_{x,t}$ are the corresponding AC replicas (see Figure 4.6(c)). Similarly, the runtime partner set updating rule 1(b) guarantees

that $C_{x,k}$ and $C_{x,t}$ become communication partners during the evolution from Figure 4.6(a) to Figure 4.6(c) at time T' . Hence, agents executing J_i will communicate directly with the agents executing J_j about the location of J_i 's execution results.

So, no matter how the job workflow is executed, if there is a direct edge from subjob J_i to subjob J_j , the agents that execute subjob J_j will be able to locate results generated by subjob J_i . $\#$

Lemma 4.3.2. *At any specific time T , assume that the corresponding AC replicas for executing subjobs $J_{i'}$ and $J_{j'}$ are $C_{x,k'}$ and $C_{x,t'}$ respectively. Agents corresponding to $C_{x,k'}$ and $C_{x,t'}$ need to communicate with each other iff $\mathcal{T}_{J_{i'}} \xrightarrow{d} \mathcal{T}_{J_{j'}}$ or $\mathcal{T}_{J_{j'}} \xrightarrow{d} \mathcal{T}_{J_{i'}}$.*

Proof: We first prove, by contradiction, the necessity (that is, if agents corresponding to $C_{x,k'}$ and $C_{x,t'}$ need to communicate with each other, then there must be a “ d ” edge between $\mathcal{T}_{J_{i'}}$ and $\mathcal{T}_{J_{j'}}$). According to our subjob grouping algorithm, if there is an edge between $J_i \in \mathcal{T}_{J_{i'}}$ and $J_j \in \mathcal{T}_{J_{j'}}$, it must be marked with “ d ”. Assuming that there is no edge between $\mathcal{T}_{J_{i'}}$ and $\mathcal{T}_{J_{j'}}$, obviously there is no need for the agents executing subjobs in $\mathcal{T}_{J_{i'}}$ to know the execution results of subjobs in $\mathcal{T}_{J_{j'}}$ and vice versa. So, agents corresponding to $C_{x,k'}$ and $C_{x,t'}$ need not to communicate with each other. Thus, there is a contradiction to the necessity condition.

We then prove the sufficiency (that is, if there is a “ d ” edge between $\mathcal{T}_{J_{i'}}$ and $\mathcal{T}_{J_{j'}}$, agents corresponding to $C_{x,k'}$ and $C_{x,t'}$ must communicate with each other). Assume that $J_i \in \mathcal{T}_{J_{i'}}$ and $J_j \in \mathcal{T}_{J_{j'}}$ and there is a “ d ” edge between J_i and J_j . $C_{x,k'}$ and $C_{x,t'}$ are the AC replicas for executing subjobs $J_{i'}$ and $J_{j'}$ respectively. This situation is similar to the one depicted in Figure 4.6(a). As discussed in the proof of Theorem 4.3.1, the situation evolves to one of the situations shown in Figures 4.6(b) and (c). Obviously, if agents corresponding to $C_{x,k}$ and $C_{x,t''}$ (or $C_{x,k}$ and $C_{x,t}$) do not communicate with each other, there is no way for the agents executing subjob J_j to locate the execution results

of subjob J_i . From Lemma 4.3.1, we know that $C_{x,k} \in \mathcal{R}(C_{x,k'}, T)$, $C_{x,t} \in \mathcal{R}(C_{x,t'}, T)$, and $C_{x,t''} \in \mathcal{R}(C_{x,t'}, T)$. So, for agents corresponding to $C_{x,k}$ and $C_{x,t''}$ (or $C_{x,k}$ and $C_{x,t}$) to communicate with each other, agents corresponding to $C_{x,k'}$ and $C_{x,t'}$ must communicate with each other in the first place. $\sharp$

Lemma 4.3.3. *At any specific time T , assume $C_{x,k}$ and $C_{x,t}$ are the AC replicas for executing subjobs J_i and J_j . $C_{x,k}$ has $C_{x,t}$ in its partner set iff $\mathcal{T}_{J_i} \xrightarrow{d} \mathcal{T}_{J_j}$ or $\mathcal{T}_{J_j} \xrightarrow{d} \mathcal{T}_{J_i}$.*

Proof: For necessity, the condition is $C_{x,k}$ has $C_{x,t}$ in its partner set. We prove by contradiction. Assume that neither $\mathcal{T}_{J_i} \xrightarrow{d} \mathcal{T}_{J_j}$ nor $\mathcal{T}_{J_j} \xrightarrow{d} \mathcal{T}_{J_i}$ holds. According to our runtime partner set updating rules 1(d) and 2, $C_{x,t}$ must be deleted from $C_{x,k}$'s partner set and vice versa. This is against the condition of the necessity. So, if $C_{x,k}$ has $C_{x,t}$ in its partner set, either $\mathcal{T}_{J_i} \xrightarrow{d} \mathcal{T}_{J_j}$ or $\mathcal{T}_{J_j} \xrightarrow{d} \mathcal{T}_{J_i}$ must hold.

To prove sufficiency, we need prove that if $\mathcal{T}_{J_i} \xrightarrow{d} \mathcal{T}_{J_j}$ or $\mathcal{T}_{J_j} \xrightarrow{d} \mathcal{T}_{J_i}$, $C_{x,k}$ has $C_{x,t}$ in its partner set. This is obvious based on the runtime partner updating rules 1(a), 1(b) and 1(c). $\sharp$

Theorem 4.3.2 (Optimality). *At any specific time T , assume $C_{x,k}$ and $C_{x,t}$ are the AC replicas for executing subjobs J_i and J_j . $C_{x,k}$ has $C_{x,t}$ in its partner set iff agents corresponding to $C_{x,k}$ and $C_{x,t}$ need to communicate with each other.*

Proof: The proof is obvious based on Lemmas 4.3.2 and 4.3.3. $\sharp$

4.4 Experimental Study

It has been proven in the last section that no redundant partner is included in the partner set of any AC replica during the execution of a job workflow over the MCCF, when the dynamic partner identification algorithm is used. To compare it empirically with the “Naive” partner identification algorithm (mentioned in Subsection 4.2) and the

Table 4.2: Configuration of the ModelNet Testbed

Micro-processor	Main Memory	OS	Number of Machines
2.6 GHz P4	512 MB	FreeBsd 4.11	1
2*2.8 GHz P4	1 GB	Fedora Core 2	8
450 MHz PII	128 MB	Redhat 7.3	3
450 MHz PII	384 MB	Redhat 7.3	2
333 MHz PII	64 MB	Redhat 7.3	2
450 MHz PII	256 MB	Redhat 7.3	1
733 MHz PIII	512 MB	Redhat 7.3	2
866 MHz PIII	512 MB	Redhat 7.3	2

preprocessing-based static partner identification algorithm, we executed the simulated job workflows on a prototype system of the Mobile Code Collaboration Framework (MCCF) using contact list based agent communication to keep track of partner location.

In general, the efficiency of the MA communication approach can be measured by the number of messages sent, the size of the messages, the distance travelled by the messages and possible synchronization cost between MA migration and messages delivery [48]. In this experimental study, we measure the number of messages sent and the overall job workflow execution time in particular.

4.4.1 Testbed Configuration

A Grid based distributed computation is usually carried out on WAN rather than on LAN clusters. To emulate the network characteristics over the Grid, ModelNet [121], which emulates WAN network delivery characteristics on a LAN cluster, was used. The MCCF prototype system was also deployed to the emulated WAN using ModelNet.

Our ModelNet setup consists of 21 machines, one of which runs the emulator and the

others host virtual nodes. Machine configurations are shown in Table 4.2. The network topology used in the WAN emulation experiment was generated by inet [116] using the transit-stub model. It consists of 4000 nodes. 20 clients, which run on the virtual nodes, were created. They were randomly attached to the stub domains in the generated network topology. The link delay is set based on the length of the link and the link bandwidth. There are four types of links in the transit-stub model (i.e., client-stub, stub-stub, stub-transit, and transit-transit links). Their bandwidths are given in Table 4.3.

Table 4.3: Link Characteristics in the Emulated WAN

Type of Links	bandwidth
client-stub	100Mbps
stub-stub	10Mbps
stub-transit	10Mbps
transit-transit	100Mbps

4.4.2 Experiments Using d-h-TGs

As stated in Section 2.2.1, a job workflow can be represented as a DAG, which is also referred to as task graph (TG) in the literature. TGs can be classified into *regular* and *irregular graphs* [102, 120]. The regular TG can be further classified into *pipeline*, *out-tree*, *in-tree*, *lattice* and *chain*, which are depicted in Figure 4.7. Irregular TGs are composed by mixing the regular types of sub-TGs. Wisconsin Center for Education Research (WCER) video processing pipeline is a typical application having a pipeline TG [82]. Master/Slave applications, e.g., the application to tracking space junk [51], are simple examples of out-tree type TGs. The TGs of more complex applications, e.g., Sweep3D [17], could be usually composed by mixing the above types of regular TGs.

As shown in Figure 4.7, TGs are of different shapes. Here, we first evaluate the

Figure 4.7: Different Types of Regular TGs

algorithms using a class of TGs, called “d-h-TG”, which are generated by mixing an out-tree and an in-tree TGs. Then, in the next subsection we evaluate the algorithms using some randomly generated TGs.

For a given “d-h-TG”, h is the height of the out-tree and the in-tree used in constructing the graph and d is the out-degree (in-degree) of a node in the out-tree (in-tree). An example “3-2-TG” is shown in Figure 4.8. Four job workflows with “d-h-TG” shaped DAGs are evaluated in the experiment, with h fixed at 2 and d varied from 2 to 5. Given the structure of “d-h-TG”, it is not difficult to derive that the preprocessing based static and dynamic partner identification algorithms will generate the same partner set for each subjob group.

Scalability Analysis of the Preprocessing Based Algorithms

For a job workflow with “d-h-TG” shape, the number of subjobs, $N_J(d, h)$, can be calculated by:

$$\begin{aligned}
 N_J(d, h) &= 1 + d + \dots + d^h + d^{h-1} + \dots + 1 \\
 &= d^h + \frac{2(d^h - 1)}{d - 1}
 \end{aligned} \tag{4.4.1}$$

Figure 4.8: A “3-2-TG” Job Workflow

The number of AC replicas created during job workflow execution, $N_{AC}(d, h)$, can be calculated by

$$N_{AC}(d, h) = d^h \quad (4.4.2)$$

It is obvious that when larger d or h is used, N_{AC} increases quickly. The size of a partner set may vary during the dynamic execution of the job workflow. To analyze the scalability of the preprocessing based algorithms, we consider the worst case using the maximum possible size of a partner set. Let $\bar{L}(d, h)$ denote the average size of maximum partner set of all AC replicas. If $\bar{L}(d, h)$ is too big, it will be very costly for updating MA contact information during job workflow execution.

From Figure 4.8, we observe that one “d-h-TG” is composed of d “d-(h-1)-TG”, a head node with d out-degree and a tail node with d in-degree. Figure 4.8 also shows subjob groups generated by the subjob grouping algorithm introduced in Subsection 4.3.1. Let $g_{i,k}$ denote the k th group of the i th “d-(h-1)-TG”. The total size of maximum partner sets of all AC replicas of a “d-(h-1)-TG” graph can be estimated by $\bar{L}(d, h-1) * N_{AC}(d, h-1) * d$. Because of the hierarchical construction, the additional partners added to the group $g_{i,0}$,

$0 \leq i \leq (d-1)$, are $2(d-1)$. Thus, we have:

$$\begin{aligned}\bar{L}(d, h) &= \frac{\bar{L}(d, h-1) * N_{AC}(d, h-1) * d + 2(d-1)}{d^h} \\ &= \bar{L}(d, h-1) + 2\frac{d-1}{d^h}\end{aligned}\quad (4.4.3)$$

Specially, for a given workflow with “d-2-TG” shape, the number of subjobs is $N_J(d, 2) = d^2 + 2d + 2$. The number of AC replicas is $N_{AC}(d, 2) = d^2$. An example “3-2-TG” is shown in Figure 4.8.

- For $g_{0,0}$, its maximum partner set will be $\{g_{0,1}, \dots, g_{0,d-1}, g_{1,0}, g_{2,0}, \dots, g_{d-1,0}\}$, whose size is $2(d-1)$.
- For $g_{i,0}, 1 \leq i \leq (d-1)$, its maximum partner set will be $\{g_{0,0}, g_{i,1}, g_{i,2}, \dots, g_{i,d-1}\}$, whose size is $(d-1) + 1 = d$.
- For $g_{i,k}, 0 \leq i \leq (d-1)$ and $1 \leq k \leq (d-1)$, its maximum partner set will be $\{g_{i,0}\}$, whose size is 1.

Hence, the average size of maximum partner set of a “d-2-TG” shaped job workflow is:

$$\begin{aligned}\bar{L}(d, 2) &= \frac{2(d-1) + d(d-1) + d(d-1)}{d^2} \\ &= 2 - \frac{2}{d^2}\end{aligned}\quad (4.4.4)$$

Based on Equation (4.4.3) and Equation (4.4.4), we have:

$$\bar{L}(d, h) = 2 - \frac{2}{d^h} < 2 \quad (4.4.5)$$

When d and h become larger, d^h becomes larger too, i.e., the number of AC replicas gets larger. However, according to Equation (4.4.5), the average size of the maximum partner set will still be less than 2.

Table 4.4: Average Size of Maximum Partner Set for the Simulated “d-2-TG” Job Workflows

d	2	3	4	5
N_J	10	17	26	37
N_{AC}	4	9	16	25
$\bar{L}(d, 2)$	1.5	1.78	1.88	1.92

The number of subjobs, the number of AC replicas and the average size of maximum partner set of all AC replicas for the simulated job workflow used in this experiment are listed in Table 4.4.

Experiment Results

Figure 4.9: Total Number of Partner Update Messages for the Simulated “d-2-TG” Job Workflows

To evaluate the efficiency of the algorithms, the total number of partner update messages are recorded during the job workflow execution using both the “Naive” partner identification algorithm (introduced in Subsection 4.2) and the preprocessing based algorithms (introduced in Subsection 4.3.2 & 4.3.3). To measure the effect of agent commu-

nication on job workflow execution, the time for the entire job workflow execution is also compared. The number of partner update messages against the number of AC replicas required in the job workflow execution is plotted in Figure 4.9. For the “Naive” partner identification algorithm, the results show that during the job workflow execution, about $O(n^2)$ update messages are transmitted, where n is the number of AC replicas. However, for preprocessing based partner identification algorithms, only $O(n)$ update messages are required. This result is also consistent with the scalability analysis in the last subsection (i.e., the average size of the maximum partner set is $O(1)$ (Please refer to Equation (4.4.5))).

Figure 4.10: Simulated “d-2-TG” Job Workflows’ Makespan on Emulated WAN

In Figure 4.10, the execution time of the job workflow on the emulated WAN is plotted against “d-2-TG” job workflows with increasing d . Each data point was obtained using the average over multiple runs. In general, preprocessing based partner identification algorithms give better performance than the “Naive” algorithm.

When d varies from 2 to 5, there will be 10, 17, 26, and 37 subjobs in the task graph respectively (according to Equation (4.4.1)). The number of AC replicas required are 4,

9, 16, and 25 respectively (according to Equation (4.4.2)). Intuitively, as the number of subjobs and the number of partner update messages increase, the execution time of the job workflow should also increase. However, as shown in Figure 4.10, a makespan of “4-2-TG” is shorter than that of “3-2-TG” when preprocessing based partner identification algorithms are used.

Different mappings of the subjobs can lead to different cost for message delivery because of the communication latency and bandwidth. For “d-2-TG”, when d increases, the number of subjobs in the job workflow will also increase. However, this does not change the height of the graph (which remains as 5). So, when there are sufficient computing resources, the communication cost becomes a dominating factor in determining the makespan. The MCCF uses a random scheduling algorithm to map ready subjobs to the available computing resources during runtime. For “3-2-TG”, at any time, there are at most 9 subjobs that can be executed concurrently (one by each AC replica). So, there is more likelihood for these subjobs to be mapped to the clients belonging to different stub-domains during the dynamic execution of the job workflow, compared to the case of “4-2-TG” (which can have up to 16 subjobs executed concurrently). This makes it possible that a makespan of “4-2-TG” is shorter than that of “3-2-TG” when preprocessing based partner identification algorithms are used.

Figure 4.11 shows the makespan of executing the “d-2-TG” job workflows on a LAN cluster with 20 machines using the “Naive” and the dynamic partner identification algorithm. When $d = 2$, since the number of AC replicas required is small, there is a small difference between the two algorithms in terms of the total execution time. When d becomes larger (that is, when more AC replicas are required in the job workflow execution), the difference in terms of the number of partner update messages between the two algorithms becomes more and more obvious (see Figure 4.9). With the decrease in the number of update messages, the results clearly show that the system using the prepro-

Figure 4.11: Simulated “d-2-TG” Job Workflows’ Makespan on a LAN Cluster

cessing based dynamic partner identification algorithm has better performance. It is also observed that the situation shown in Figure 4.10 does not occur since in a LAN cluster environment, the latencies and bandwidths between different machines are exactly the same.

4.4.3 Experiments Using Real Application TG and Randomly Generated TGs

Simulated Job Workflow

To be more general, we apply the “Naive” and the preprocessing based partner identification algorithms on a real application TG and some randomly generated TGs to test their performance. The real application TG is based on the workflow for finding clusters of Galaxies in SDSS [23], which resembles an in-tree task graph. The workflow includes four stages of computation: (i) At the first stage, there are 24 data independent subjobs whose input data come from distributed data sets; (ii) At the second stage, there are 12 data independent subjobs. The input data of each subjob at this stage include the

Figure 4.12: Real Application TG

output data of the 24 subjobs in the first stage and distributed data sets; (iii) At the third stage, there are 6 data independent subjobs. The input data of each subjob at this stage include the output data of the 12 subjobs in the second stage and distributed data sets; and (iv) At the last stage, there is one subjob, whose input data include the output data of each subjob in the third stage. This real application TG is presented in Figure 4.12, where subjob 0 is a hypothetical subjob added to the workflow. It serves to start the job workflow. It does not do any computation, nor generates any data.

We also generated 6 pseudorandom TGs using the Task Graphs for Free (TGFF) system [55] for the comparison study. Since in our discussion, we assume that there is a unique end node for every TG, when a TG has multiple subjobs that have no offspring, a hypothetical subjob, also referred to as the end node, is added. This subjob serves as their common immediate successor to consolidate the final results and has no computation cost. The generated TGs are illustrated in Figure 4.13, where the dotted filled circle denotes the added hypothetical subjob, and the dotted edge denotes the added edges. The random TGs after preprocessing are shown in Figure 4.14. Each job workflow execution

Figure 4.13: Random Task Graphs

is repeated 5 times. The average number of partner update messages and the execution time of the job workflow are shown in Figures 4.15 and 4.16 respectively. Figure 4.15 also shows the percentage improvement of the preprocessing based static and dynamic partner identification algorithms over the “naive” algorithm. From these results, we observe that:

- The “Naive” algorithm generates the highest number of update messages, and the number of partner update messages generated by the static algorithm is greater than or equal to those generated by dynamic algorithm. The results obtained are consistent with those reported in the last subsection. That is, when the number of AC replicas required to execute the job workflow increases, the preprocessing based partner identification algorithms achieve better performance compared with the “Naive” algorithm.
- For makespan, in general, the “Naive” algorithm generates the longest makespan,

Figure 4.14: Preprocessed Random Task Graphs

Figure 4.15: Total Number of Partner Update Messages for the Simulated SDSS and Random Job Workflows

Figure 4.16: Simulated SDSS and Random Job Workflow Makespans on Emulated WAN

the dynamic partner identification algorithm generates the shortest makespan. It is noted that when the number of AC replicas required is small and there is a dominating group in the job workflow (e.g., TG1), although there is considerable percentage improvement in terms of reduction of partner update messages, there may not be an improvement on makespan. This is because, in these cases, the total number of update messages is small even if the “Naive” algorithm is used.

Table 4.5: Comparison between Preprocessing Based Static and Dynamic Algorithms (Number of Partner Update Messages)

TG ID	$ \mathcal{P} $	# of ‘d’ Edges	Static Algorithm	Dynamic Algorithm	% Improvement
0	32	25	48	38	21%
1	5	5	5	5	0%
2	10	10	14	14	0%
3	20	11	56	44	21%
4	4	4	12	12	0%
5	9	5	27	21	22%

As proven in Subsection 4.3.4, the dynamic partner identification algorithm guarantees

the optimality. There is no redundant partner in an AC partner set during the dynamic execution of the job workflow. However, for the static partner identification algorithm, because of the lack of dynamic information, for each “d” edge in the reduced graph, additional partners may need to be identified, as described in the algorithm shown in Figure 4.4. Table 4.5 shows the number of partner update messages generated using either the static or dynamic partner identification algorithm during the dynamic execution of job workflows. It also shows the number of “d” edges in the reduced graph corresponding to each TG and the total number of group pairs (i.e., partners) in the partner set $\mathcal{P}$, denoted as $|\mathcal{P}|$, generated using the static partner identification algorithm. For every “d” edge, optimally there should be only one group pair defined in the partner set. The relative performance of the static algorithm is proportional to the ratio between $|\mathcal{P}|$ and the number of the “d” in the reduced graph. The results presented in Table 4.5 also empirically show the optimality of the dynamic partner identification algorithm. Obviously, the value of $|\mathcal{P}|$ generated by the static partner identification algorithm depends on the structure of the TG and it determines the performance of the algorithm. Particularly, for the small SDSS shown in Figure 4.12, according to the preprocessing algorithm, subjobs 0, 24, 36, 42, and 43 will be in group g_0 . For other subjobs, each will belong to a separate group. So, for a given subjob, no matter whether the static or dynamic algorithm is used, it will have the same partner set. Therefore, preprocessing based static and dynamic partner identification algorithms generate the same makespan in this case.

4.5 Summary

During a job workflow’s execution in the MCCF, it is necessary for the agents executing a subjob to locate the input data from its predecessors. If the subjob’s predecessor is executed by the same AC, the location of the input data from its predecessor is readily available in the AC. Otherwise, the agents executing the subjob need to get the location

of its predecessor’s execution result through an execution coordination mechanism.

When a direct coordination technique is used, agents need to obtain the location of the subjob’s execution result through communication. Existing research on mobile agent communication focuses on dynamic agent location/discovery and communication, assuming that the partners are already known. However, with the dynamic AC replica creation in the MCCF, this assumption does not hold any more. By studying the dynamic characteristics and agent communications during a job workflow execution over the MCCF, we identify the unnecessary communication partners. A novel subjob grouping algorithm for preprocessing the static job workflow specification is developed. Information generated from preprocessing is used for partner identification in both the static and dynamic algorithms. The static partner identification algorithm employs the preprocessed information and determines the partner set before the execution of the job workflow. Since AC replicas are dynamically generated, redundant partners will have to be included; whereas, in the dynamic partner identification algorithm, an agent’s partner set is dynamically updated using the preprocessed information during the distributed job workflow execution. Therefore, the partners in the set are limited to only those necessary ones. The correctness and optimality of the dynamic partner identification algorithm have also been proven formally.

To evaluate our approach, experiments have been carried out on “d-h-TG” shaped job workflows, randomly generated job workflows as well as the workflow of a real application. For the “d-h-TG” type of job workflows, the theoretical analysis of our approach shows that the average size of the maximum partner set will always be less than two regardless how many AC replicas are required in the job workflow execution (see Equation 4.4.5). The experimental results show that the total number of partner update messages increases linearly with the total number of AC replicas required, which is consistent with the theoretical analysis. Similar behaviour is also observed for the randomly generated job

workflows.

The experiments show that the preprocessing based partner identification algorithms have better performance than the “Naive” algorithm in all the cases. The preprocessing based dynamic algorithm gives the optimal result in terms of the number of partner update messages generated. It out-performs the preprocessing based static algorithm when redundant partners have to be included due to the lack of dynamic information. As analyzed in the static partner identification algorithm (see Subsection 4.3.2) and the experiments, whether or not redundant partners need to be added depends on the topology of the workflow.

As described in Section 4.1, there are many existing direct execution coordination techniques. To further analyze their applicability, how they can be implemented in the MCCF needs to be further investigated. The analysis of the implementation of the existing techniques leads to the development of a new coordination technique, *delegator based coordination*, which will be described in the next chapter.

Chapter 5

A Delegator based Execution Coordination in Mobile Agent based Distributed Job Workflow Execution

In the last chapter, we presented a taxonomy of the existing execution coordination techniques, and investigated algorithms for partner identification to support direct execution coordination for mobile agent based distributed job workflow execution. The contact list based direct execution coordination was adopted for the simulation experiments in Chapter 4. However, how an execution coordination technique may affect a job workflow execution has not been investigated yet.

In general, metrics to evaluate the performance of an execution coordination technique include: (i) the cost associated with maintaining the coordination mechanism (e.g., additional operations for partner discovery using direct coordination techniques); and (ii) the cost associated with the job workflow execution (e.g., the cost for a subjob to obtain the locations of the required results from its predecessors). Using a decentralized technique, e.g., contact list based approach, the coordination between agents is fully decentralized. However, to maintain the coordination mechanism (e.g., the contact list), before an MA is migrated, it has to notify all its partners about its new location. In order to ensure the message is received by the partners, the MA cannot be migrated until it

gets all the acknowledgements from its partners. This may introduce considerable overhead during MA migration and thus prolong the job workflow execution. The centralized lookup infrastructure based coordination is a hybrid of the centralized techniques and the decentralized techniques. As pointed out in [75], centralized lookup infrastructure based coordination techniques rely on some static lookup servers for MA discovery, which do not scale well for large-scale MA systems. That is, the existing techniques either rely on a centralized mechanism for execution coordination or introduce substantial overhead when applied in the MCCF. The analysis of the implementation of the existing techniques in Section 5.1 leads to the development of a new coordination technique, delegator based coordination, which will be described in this chapter.

5.1 Existing Execution Coordination Techniques Applied in the MCCF

As discussed in Section 4.1, we only consider the direct coordination techniques for the MCCF. The existing direct coordination techniques include centralized lookup infrastructure based coordination and decentralized coordination. For decentralized coordination, the flooding based technique generates excessive flooding messages which will degrade the overall system's performance considerably and cause a large amount of traffic over the network. The maintenance of the distributed hash table (DHT) in a DHT based coordination overlay is costly. In addition, the average number of hops required for sending a message is of the order of $\log n$ (n is the number of computational resources). Therefore, to further analyze the applicability of various coordination techniques in the MCCF and to evaluate their performance, mailbox based (one of the centralized lookup infrastructure based coordination techniques) and contact list based (one of the decentralized coordination techniques) approaches have been selected and implemented in the MCCF.

5.1.1 Mailbox Based Coordination

Mailbox based coordination [48] is one of the centralized lookup infrastructure based coordination techniques. It relies on a centralized lookup server, denoted as *MBLocator*, for partner identification and location discovery. The *MBLocator* keeps the *annotated DAG* and the spanning tree generated from the preprocessing, and maintains a record in the following form for each AC replica: (AC_id, SjbId, ACLoc, MBLoc).

- AC_id: The id of the AC replica (which has the same meaning as (x, k) , as introduced in Section 3.3.2);
- SjbId: The id of the current subjob being executed by the AC replica (which has the same meaning as J_i , as introduced in Section 4.2);
- ACLoc: The current location of the AC replica (which has the same meaning as $L_{J_i}^{x,k}$, as introduced in Section 4.2, where the AC replica's current subjob id is J_i); and
- MBLoc: The location of the AC replica's mailbox.

It is assumed that each job workflow has a separate *MBLocator*, which keeps track of the execution status of the entire workflow.

Each AC is associated with a mailbox, which is used by the AC agent to retrieve the locations of predecessors' execution results. In addition to the annotated DAG and the spanning tree, a mailbox maintains the following information:

- SjbResultLoc[]: The locations of subjob results that have been received; and
- SjbId: The id of the current subjob being executed by the AC.

When an AC is replicated, the mailbox for the new AC replica is generated by duplicating the existing one.

The basic principle of the mailbox based coordination is as follows: when an AC agent completes the execution of its current subjob (e.g., J_i), it asks MBLocator for the locations of its partners. After that, it will send the location of the execution result to the partners' mailboxes directly. Assume that a sub-tree rooted at J_j is denoted as $\mathcal{T}_{J_j}$ and J_j is the current subjob being executed by the AC agent. Obviously, the subjobs to be executed by this AC may require the execution results of subjob J_i if and only if there is an edge labelled with “ d ” from J_i to $J_{j'}$, $J_{j'} \in \mathcal{T}_{J_j}$ (e.g., as shown in Figure 5.2(a)). The components for mailbox based coordination and their communications are illustrated in Figure 5.1. The main steps of the communication are:

Figure 5.1: Mailbox based Coordination

- The AC agent sends a “Request” message and an “UpdateSjbId” message con-

currently to the MBLocator and its corresponding mailbox (i.e., Sender Mailbox), respectively. The “Request” message is for updating the MBLocator and identification of partners, and the “UpdateSjbId” message is for updating the mailbox with the id of the subjob to be executed next by the AC (i.e., SNewSjbId).

- Upon receiving the “Request” message, the MBLocator will first update the corresponding AC record with the id of the subjob to be executed (i.e., SNewSjbId) and the new location of the AC (i.e., ACNewLoc). New AC records will be also generated using the information contained in NewACTuple[]. It is an array of tuples, (AC_id, ACLoc, SjbId, MBLoc), each of which contains information about a newly created AC replica. It will then identify the partners for the subjob using its id (i.e., SSjbId), the status of the job workflow (i.e., AC records), and the information from the spanning tree and the annotated DAG. An array of PMailboxLoc, one element for each partner, will be returned. PMailboxLoc is a tuple that contains the partner’s mailbox location (i.e., MBLoc) and the partner’s current subjob id (i.e., PSjbId).
- Once receiving the “Reply” message from the MBLocator, the AC agent will send a “ResultNotification” message to each of its partner’s mailbox.

Figure 5.2: Dynamic Partnership

However, due to the dynamic AC replication and concurrent execution of data independent subjobs, the partner information obtained by the AC agent from the MBLocator

may not be up to date. The partnership, therefore, needs to be further checked by the receiving mailbox. This is carried out by the functions shown in Figure 5.1, where “Partner(SSjbId, PSjbId, SjbId)” checks whether the sender is still a partner of the AC that corresponds to the receiving mailbox. SSjbId is the id of the subjob being executed by the sender, PSjbId is the id of the partner’s subjob returned by the MBLocator, and SjbId is the id of the current subjob being executed by the receiver. “NewPartner(SSjbId, PSjbId, SjbId)” checks whether there are new AC replicas created by the AC that corresponds to the receiving mailbox during the message delivery. Figure 5.2 illustrates various scenarios. It is assumed that the AC agent that completes the execution of subjob J_i is updating its partner’s mailbox with the location of J_i ’s execution result and the J_j is its partner’s current subjob id returned by MBLocator. The current subjob being executed by the AC agent is highlighted with a solid circle in the figure.

- If the partnership still holds, the receiving mailbox will create a record for the received location of the subjob’s execution result. If there is no new partnership discovered (as shown in Figure 5.2(a)), it will return a “Reply” message to indicate the success of the operation. Otherwise (i.e., there is new partner discovered as the result of AC replication (as shown in Figure 5.2(b))), the receiving mailbox will return a “Reply” message to indicate that the sending AC agent needs to contact the MBLocator to locate the mailboxes of the new partners.
- If the partnership does not hold anymore because of AC migration (as shown in Figure 5.2(c)), the receiving mailbox will return a “Reply” message to notify the sending AC agent that it needs to contact the MBLocator again to rediscover the partners and locate their mailboxes.

Before a subjob starts execution, its corresponding agent needs to obtain the locations of its predecessors’ execution results from the mailbox and to get the necessary input for

Figure 5.3: The Retrieval of the Location of Predecessors' Execution Results

the subjob. The main communications between the mailbox and the AC agents for the AC agents to retrieve the location of execution result are shown in Figure 5.3.

- AC agent sends a “Request” message to the mailbox for the locations of its predecessors' execution results with “SjbIds[]” (which is an array of subjob ids, one for each predecessor).
- If the locations of all the requested execution results are ready, a “reply” message with an array of the locations of the requested execution results will be returned, as illustrated in Figure 5.3(a). Otherwise, a “reply” message will be returned to indicate the requested information is not ready.
- When the requested info becomes available, the mailbox sends a “Notification” message with an array of the locations of the requested execution results to the waiting agent, as illustrated in Figure 5.3(b).

5.1.2 Contact List Based Coordination

Contact list based coordination [98] is one of the decentralized coordination techniques. Without a centralized server, AC agents take the responsibility for partner identification, partner location discovery, and execution coordination. Each AC maintains a contact list of partners, which is an array of tuples (AC_id, SjbId, ACLoc), where ACLoc is the location of the computational resource where the subjob J_i is currently being executed. Before an AC is migrated or discarded, it will notify its partners so that they can update their contact lists accordingly.

Let $\mathcal{E}_d$ be the set of edges that are marked with “ d ” in the annotated $\mathcal{G}$. As explained in Chapter 4, the partner identification for this approach is based on the following principle: in general, suppose J_i and J_j are two subjobs that are currently under execution. If $\exists (J_{i'}, J_{j'}) \in \mathcal{E}_d$, and $J_{i'} \in \mathcal{T}_{J_i} \wedge J_{j'} \in \mathcal{T}_{J_j}$ (see the definition of $\mathcal{T}_{J_i}$ in the last subsection), then the AC replicas corresponding to J_i and J_j are partners (see Section 4.3.3 for details).

Suppose the current subjob is J_i and its corresponding AC replica is $C_{x,k}$, and that there are m outgoing edges in the spanning tree. After the subjob J_i completes its execution, the AC can be migrated or discarded depending on the annotated DAG.

The components for the contact list based execution coordination and their communications are illustrated in Figure 5.4.

- The AC agent sends a “ResultNotification” message to its partner agents for contact list update.
- Upon receiving all the “Ack” messages from its partner agents (as illustrated in Figure 5.4 (a)), the agent will update the contact lists accordingly using the algorithm described in Section 4.3.3. Then, the AC replicas are ready to be migrated. When J_i is a leaf node in the spanning tree (i.e., all outgoing edges of J_i , if any, are

Figure 5.4: Contact List based Coordination

marked with “ d ”), AC replica $C_{x,k}$ will be discarded.

For a partner, assuming its AC replica is $C_{x,k'}$, upon receiving the “ResultNotification” message, the partner’s AC agent will update $C_{x,k'}$ to include the location of J_i ’s execution result (i.e., `SjbResultLoc`). Then, the partner’s AC agent will determine partnership between $C_{x,k'}$ and the newly created AC replicas in `NewACTuple[]`. If the new AC replica is $C_{x,k'}$ ’s partner, a new entry corresponding to the AC replica needs to be added to $C_{x,k'}$ ’s contact list accordingly. In addition, the partner’s AC agent will check whether $C_{x,k}$ is still its partner since it will be migrated to execute a new subjob. If so, the entry corresponding to $C_{x,k}$ in $C_{x,k'}$ ’s contact list will be updated using `ACNewLoc` and `SNewSjbId`. Otherwise, the entry will be removed.

Due to the concurrent execution of data independent subjobs, it is possible that two

partners complete their subjobs and send the “ResultNotification” messages simultaneously. Assume that for some reason the transmission of one “ResultNotification” message is delayed. As illustrated in Figure 5.4(b), when the message reaches the partner’s original location, the partner has been migrated to a new location. Therefore, the sending agent’s “ResultNotification” message will not be received by the partner. However, since the sending agent has received the partner’s “ResultNotification” message, the corresponding entry in the contact list should have been modified. If the partnership still holds, the sending agent needs to resend the “ResultNotification” message using the partner’s new location information if it does not receive the “Ack()” message within a given time threshold (e.g., the average round-trip-time for a message transmitted over the network).

Only after the agent receives the “Ack” message from all the partners can the corresponding AC replica be migrated. So, when two partner AC agents sent “ResultNotification” messages concurrently, at least one of them will succeed to receive all “Ack()” message and migrate. However, when the above scenario happens, the cost for AC replica migration will be increased significantly.

Different from the mailbox approach, the AC in the contact list based approach also contains the location of the subjob’s execution results. These are updated when “ResultNotification” messages are received. Thus, an AC agent can retrieve the location of a subjob execution result directly from its corresponding AC replica when needed. When the required location of the subjob’s execution result is not yet available, the AC agent then needs to wait for the arrival of the corresponding “ResultNotification” message.

5.2 Delegator based Coordination

There is no centralized server in contact list based coordination. It is easy to implement and the location of the required subjob execution result is also readily available

in the local AC. However, the problem of concurrent result notification, as described in the subsection 5.1.2, may cause considerable delay in AC migration. The mailbox based approach avoids this problem by associating a mailbox with each AC. AC agents do not communicate directly. Instead, they send result notifications to the mailboxes. But, the mailbox based approach requires complex communications between the agent, MBLocator and the mailbox for partner identification. The MBLocator is used for preliminary partner identification and the mailbox is used for the confirmation of the partnership. In the case where the partner identified by the MBLocator is obsolete, the procedure for sending result notification to the partners needs to be repeated. To take advantages of both the contact list based coordination and the mailbox based coordination, an original decentralized execution coordination technique, *delegator based coordination*, is proposed in this section.

The delegator based coordination is a hybrid of the mailbox and contact list based approaches. Each AC is associated with a delegator. Like a mailbox, it is used by the AC agents to retrieve the locations of the predecessors' execution results. Unlike a mailbox, which is a passive object only for receiving messages, a delegator is an active object, which is responsible not only for receiving messages, but also for sending "ResultNotification" messages to partners on behalf of its owner. In order to identify partners and discover partners' locations, a delegator uses a contact list based communication mechanism. AC agents do not communicate with each other directly, instead, their corresponding delegators communicate with each other for partner identification, location discovery, and result notification. Similar to the contact list based approach, suppose J_i and J_j are two subjobs that are currently under execution, if $\exists(J_{i'}, J_{j'}) \in \mathcal{E}_d$, and $J_{i'} \in \mathcal{T}_{J_i} \wedge J_{j'} \in \mathcal{T}_{J_j}$, then the delegators corresponding to J_i and J_j are partners.

In addition to the annotated DAG and the spanning tree, a delegator also maintains the following information:

- SjbId: The id of the current subjob being executed by the AC replica;
- ACLoc: The current location of the AC replica;
- SjbResultLoc[]: The received subjob result locations; and
- ContactList[]: The contact list of the delegator’s partners. It is an array of *delegator tuples* (AC_id, SjbId, DelegatorLoc), where “DelegatorLoc” is the current location of the partner delegator.

Figure 5.5: Delegator based Coordination

The components for the delegator based coordination and their communications are illustrated in Figure 5.5. After a subjob completes its execution, if the AC is replicated, the delegator for the new AC replica is generated. The AC agent sends a “ResultNotificationA” message to its delegator (denoted as “Sender delegator” in Figure 5.5). The parameter “NewDelegatorTuple[]” is an array of delegator tuples, each of which contains information of a newly created delegator. Upon receiving the “Ack()” message from its delegator, the AC replica can then be migrated. Different from the mailbox and the contact list based approach, the AC replica can be migrated once the delegator acknowledge the receipt of the result notification. The result notification to the partners and contact list updating will be carried out by the delegator:

- Upon receiving the “ResultNotificationA” message from the agent, the sender delegator will update the id of the current subjob (i.e., SNewSjbId) and the current location of the AC replica (i.e., ACNewLoc). Then, it will send an “Ack” message to the agent and a “ResultNotificationB” message to its partners concurrently.
- On receiving the “ResultNotificationB” message, the receiving delegator will update its “SjbResultLoc[]” using the received subjob result location (i.e., SjbResultLoc), and update the “ContactList[]” in a way similar to the contact list based approach. After updating, the receiving delegator will reply an “Ack” message to the sender delegator.
- After receiving the “Ack” message from all its partners, the sender delegator will use its “ContactList[]” and “SjbResultLoc[]” and the received “NewDelegatorTuple[]” to initialize the newly created delegators. Then, it will update its contact list.

Similar to the mailbox based approach, agents retrieve the location of a subjob execution result from the corresponding delegator. The communications between AC agents and the delegator are similar to the ones used in the mailbox based approach, as illustrated in Figure 5.3.

As with the contact list based approach, the delegator based approach is a fully decentralized approach. By associating each AC with a delegator, the AC migration overhead is reduced. However, it introduces additional communication messages for the retrieval of the location of subjob execution results, compared with the contact list based approach.

In order to evaluate the performance of the three coordination approaches aforementioned, a comparison experimental study of the real and simulated job workflows executed on a prototype implementation of the MCCF will be given in the next section.

5.3 Performance Evaluation

In general, the efficiency of MA coordination techniques can be measured by the following factors:

- **Number of Messages Communicated:** This is to measure how much communication traffic the coordination mechanism produces. It includes the messages for both the maintenance of the coordination mechanism and data dependency management.
- **Total Waiting Time:** Total waiting time includes *migration waiting time* and *synchronization waiting time*. Migration waiting time is the duration between the time when an AC is ready to be migrated and the time when the AC migration actually takes place. An AC is ready to be migrated once its current subjob completes its execution and the resources for new subjobs, if any, have been selected. When an AC migration can actually take place depends on the coordination mechanism used. For example, in the contact list based approach, an AC migration cannot take place until the agent receives all the acknowledgement messages from the partners (see Section 5.1.2). Synchronization waiting time is the duration between the time when AC agents are instantiated and the time when all input data are ready.
- **Job Workflow Makespan:** The *makespan* of a job workflow is its total execution time. Given the same mapping of subjobs to resources and the same computational resource configuration, the makespan of a job workflow reflects the efficiency of the coordination model.

The MCCF prototype with the implementation of the three direct coordination techniques was deployed to the emulated WAN using the ModelNet. The Modelnet configuration

and network topology are the same as the ones used in Section 4.4.1.

5.3.1 Experiments Using d-h-TGs

(a) Total Number of Messages Communicated

(b) Maximum Number of Messages for a Single Component

Figure 5.6: Experiment Results for Simulated “d-2-TG” Job Workflows (I)

We first evaluate the three coordination techniques using “d-h-TG” task graphs, as defined in Section 4.4.2. In the next subsection we will evaluate the coordination techniques using a real application TG and some randomly generated TGs.

To evaluate the efficiency of the coordination mechanisms empirically, the total number of messages communicated, the maximum number of messages received by a single component, the total waiting time for the first AC replica (any AC replica will do as long as the same AC replicas are used for all mechanisms), and the execution time of the entire job workflow (i.e., job workflow makespan) were recorded during the job workflow execution. Each job workflow execution was repeated 5 times and the average is used in the analysis.

The total number of messages communicated against the number of subjobs is plotted in Figure 5.6(a), from which we can see that the delegator based approach generates the most number of messages. Similar to the contact list based approach, the messages in the delegator based and the mailbox based approaches are not handled by a single component (see Figures 5.1, 5.4, and 5.5). Instead, they are handled by multiple distributed components (e.g., delegators in the delegator based approach). Delegators in the delegator based approach adopt a contact list based coordination mechanism. They are responsible for the partner identification and location. In addition to the communication among the delegators, there are communications between AC agents and delegators. This definitely increases the number of messages that are handled by a single component in the delegator based approach compared to the original contact list based approach. The maximum number of messages that were handled by a single component in the delegator based approach and the mailbox based approach during the job workflow execution is plotted against the number of subjobs for each simulated “d-h-TG” in Figure 5.6(b). The results show that the maximum number of messages handled by a single component in the delegator based approach is much less than that of the mailbox based approach.

(a)) Total Waiting Time

(b) Job Workflows' Makespan

Figure 5.7: Experiment Results for Simulated “d-2-TG” Job Workflows (II)

Figure 5.7(a) shows the total waiting time of the job workflow execution for each “d-2-TG” job workflow used in the experiment. In general, the delegator based approach has similar total waiting time to the mailbox based approach, and the contact list based approach has the longest waiting time amongst all the approaches. When the situation

described in Figure 5.4(b) happens, the contact list based approach will generate a much longer migration waiting time than the delegator based approach (as illustrated in Figure 5.7(a) for “3-2-TG” and “5-2-TG” job workflows). For the mailbox based approach, when the situation depicted in Figure 5.1(b) occurs, the migration waiting time will be increased and therefore the total waiting time will be increased accordingly, compared to the delegator based approach (as illustrated in Figure 5.7(a) for “3-2-TG” job workflow).

In Figure 5.7(b), the job workflow makespan is plotted against the “d-2-TG” job workflows. In general, a large waiting time implies long makespan. The result shows that generally the delegator based approach achieves a shorter makespan than the contact list based approach. In addition, the makespan achieved by the delegator based approach is similar to that achieved by the mailbox based approach.

5.3.2 Experiments Using Real Application TG and Randomly Generated TGs

To be more general, we apply the three coordination approaches on the real application TG and some randomly generated TGs described in Section 4.4.3 to test their efficiency. Each job workflow execution was repeated 5 times and the average is used in the analysis. The total number of messages communicated, the maximum number of messages received by a single component, the total waiting time, and the execution time of the job workflow are shown in Figure 5.8(a) and (b) and Figure 5.9(a) and (b) respectively. From these results, we observe that:

- The delegator based approach always generates the highest number of messages (as shown in Figure 5.8(a)). However, for a single delegator, the maximum number of messages it communicates is usually less than that communicated by the *MBLocator* (as shown in Figure 5.8 (b)). When a delegator is a common partner of all other delegators, the number of message communicated by this delegator will be in a

(a) Total Number of Messages Communicated

(b) Maximum Number of Messages for a Single Component

Figure 5.8: Experiment Results for Simulated Small SDSS and Random Job Workflows (I)

similar range to that of the *MBLocator* (as shown in Figure 5.8(b) for “TG1” and “TG4” job workflows).

- As discussed in the last subsection, both the mailbox and the contact list based

(a) Total Waiting Time

(b) Job Workflows' Makespan

Figure 5.9: Experiment Results for Simulated Small SDSS and Random Job Workflows (II)

approaches may generate a long waiting time and thus affect overall execution performance. As shown in Figure 5.9(a), for “TG2”, the total waiting time for the mailbox based approach is 235 seconds. It is 146 seconds for the delegator based

approach, and it is 176 seconds for the contact list based approach. That is, the mailbox based approach generates an obviously longer waiting time for “TG2” than other approaches. The contact list based approach generates an obviously longer waiting time for “TG0”, “TG3”, “TG4”, and “TG5”. We also notice that the delegator based approach generates a slightly longer waiting time than the mailbox based approach for some task graphs. For example, for “TG0”, the delegator based approach generates 26 seconds waiting time, while the mailbox based approach generates 15 seconds waiting time.

Compared to the mailbox based approach, there are more communication overheads in the delegator based approach. For example, there are more routing steps for a result notification message delivery in the delegator based approach. This explains why the mailbox based approach outperforms the delegator based approach in some cases (e.g., TG0 and TG4 in Figure 5.9). However, as explained in Section 5.1.1, in the mailbox based approach, if the partner identified by the MBLocator is obsolete, the procedure for sending a result notification to the partners needs to be repeated. If this situation happens frequently, the performance of the mailbox based approach will be affected (e.g., TG2 and TG5 in Figure 5.9).

Figure 5.9(a) and (b) show that overall the delegator based approach gives the most consistent performance. Unlike the mailbox and the contact list based approaches, its performance is not affected by the topology of the job workflow and the dynamic execution of the job workflow.

5.4 Summary

In mobile-agent based distributed job workflow execution, execution coordination is required to support data dependencies amongst subjobs. However, as shown in this chapter,

because of the dynamic agent migration, replication and disposal employed in the MCCF, both the mailbox-based and the contact-list based approaches require complex communications for result notifications and thus may introduce considerable migration delay during job workflow execution.

To provide an efficient, decentralized execution coordination mechanism in the MCCF, an original delegator-based coordination approach is introduced in this chapter. It takes advantage of the mailbox approach by assigning each Agent Core (AC) a delegator so that the agents are freed from the tasks of identifying, locating and communicating with the partners. These tasks are performed, instead, by the delegators using a contact-list based mechanism. The results of our experimental study show that our delegator-based approach generates less number of messages per component compared with the mailbox based approach and has the most consistent performance amongst the above-mentioned three execution coordination mechanisms.

Chapter 6

Provenance Recording and Collection in Mobile Agent-Based Distributed Job Workflow Execution

The provenance of some data is defined as the documentation of the process that led to the data [71]. The necessity of provenance for job workflow execution is apparent since provenance provides a traceable path on how a job workflow was executed and how the resulting data were derived. It is particularly important in a Service Oriented Architecture (SOA) since shared services and data sets might be used in the course of the job workflow execution. Provenance information can be processed and used for various purposes, for example, for validation of e-Science experiments [117], credibility analysis of the results of workflow execution [95], fault-tolerance for service-based applications [108], and data sets regeneration for data intensive scientific applications [64]. The objective of this chapter is to develop a provenance recording and collection algorithm so that mobile agents deployed in the execution of a job workflow can collectively collect a complete set of information about the job workflow execution.

6.1 Related Work

Provenance is an annotation that can be used to explain how a particular result has been derived. *Execution provenance* provides the information regarding the execution of subjobs in the workflow (e.g., time and location of the services used for subjob execution, and condition of the execution environment), and *data provenance* provides the information about the derivation history of data sets in the workflow (e.g., time and location of the input data sets of subjobs, and the data set generated by subjobs). Thus, combination of the execution and data provenance will present a complete picture of the process that leads to the generation of the result, that is, the ancestral data sets from which the result has been derived and the services and execution flow used in the transformation process.

The provenance information can be generated from the static information available in the abstract workflow (e.g., data dependencies) together with the runtime details obtained by tracing the execution of the workflow execution. The trace can be automatically generated by developing either a special “wrapping service” of the engine [94] or an “engine plugin” [126] to capture and record provenance related data directly from the workflow engine. The workflow trace can also be collected collectively by the services that execute the subjobs [35] or the services together with the workflow engine [100]. But, this puts the responsibility of provenance data recording to the service providers and may also require service modification.

No matter how the traces are collected, in general some special provenance services are used in the current systems to store the provenance data and to provide an interface for users to query the data. Thus, a protocol is needed for various service providers and the workflow engine to communicate with the provenance services during the provenance collection process [71]. A taxonomy of data provenance techniques can be found in [99], and a comprehensive documentation on provenance architecture can be found in [70].

In the context of the MCCF, a job workflow specifies the subjobs, their data dependencies, required data and code for each subjob, and the policies for resource scheduling, as shown in Figure 3.3. Initially, the job workflow specification is abstract (i.e., the computational resources to execute subjobs, locations of data sets, and locations of the codes to be executed for subjobs are not specified). The objective of the MCCF is to map the abstract job workflow provided by the users to the dynamic Grid resources on the fly for distributed job workflow execution, as shown in Figure 3.1. Therefore, in the MCCF, execution provenance refers to the computational resources and executable code used for executing subjobs, and data provenance refers to the identities of the data sets used in the execution of subjobs and the locations of intermediate results. There is no centralized workflow engine in the MCCF. So, given the distributed nature of the MCCF, our objective is, therefore, to develop a distributed algorithm so that agents of various ACs can work collaboratively to collect complete, aggregated provenance information. In particular,

- The workflow specification contained in the AC should be used to record provenance information.
- Provenance information recording is non-functional (i.e, not subjob specific), so it should be performed by AC agents.
- Various ACs may be used in the provenance recording, but the AC that is finally returned to the user should contain the complete provenance information about the job workflow execution.

Provenance is especially important in the MCCF because of the use of mobile agents. How much trust can be placed on a mobile agent must be based on the provenance of that agent (that is, where it is originated and its history) [72]. The provenance can be used to generate a security audit to detect illegitimate changes made on the agent [111]. In the MCCF, the provenance data contained in the AC can be used by the host (i.e., the

computational resource) to obtain the origin and the history of the AC (i.e., the creator of the AC, subjobs executed by the AC, and code, data sets, and computational resources used in the execution of these subjobs). A security audit can then be performed to determine the trustworthiness of the AC before creating agents to execute the scheduled subjob.

6.2 Provenance Recording and Collection

Based on the direct agent execution coordination described in the last two chapters, it is obvious that the agents of the original AC replica may not be the partner of the agents of all other AC replicas created during the dynamic job workflow execution. Therefore, in order that the original AC replica returned back to user has a complete set of provenance information, the properties of the annotated DAG generated using the preprocessing algorithm introduced in Chapter 4 are further explored. Based on these properties, a provenance recording and collection algorithm will be developed.

6.2.1 Properties of Preprocessing Algorithm

For a DAG $\mathcal{G} = (\mathcal{J}, \mathcal{E})$, let $\mathcal{E}_i$ represent edges that indicate indirect successors. The DAG $\mathcal{G}'' = (\mathcal{J}'', \mathcal{E}'')$, where $\mathcal{J}'' = \mathcal{J}$ and $\mathcal{E}'' = \mathcal{E} - \mathcal{E}_i$, is the graph generated after removing the edges indicating indirect successors (e.g., Figure 4.2(b)). Note that this process does not change the data dependency between subjobs since during the removing process, when an edge (J_i, J_j) is removed, $J_i < J_j$ still holds.

Let $P(J_i, J_j)$ denote the set of paths from J_i to J_j in $\mathcal{G}''$, where a path $p \in P(J_i, J_j)$ represents a sequence of subjobs $(J_{k_0}, \dots, J_{k_l})$, $l = |p|$, $J_{k_0} = J_i$, $J_{k_l} = J_j$. Let $G(J_i)$ be the group id of subjob J_i .

Lemma 6.2.1. *In $\mathcal{G}''$, $\forall J_i \in \mathcal{J}'', i \neq (n-1), P(J_i, J_{n-1}) \neq \emptyset$.*

Proof: As mentioned in Section 2.2.1, $\forall J_i \in \mathcal{J}, 0 < i < (n - 1), J_0 < J_i < J_{n-1}$. Then we have $\forall J_i \in \mathcal{J}, i \neq (n - 1), P(J_i, J_{n-1}) \neq \emptyset$. Thus, the lemma holds. $\#$

Lemma 6.2.2. *The start node J_0 and end node J_{n-1} must be within the same group with the lowest group id, i.e., $G(J_0) = G(J_{n-1}) = 0$.*

Proof: According to Lemma 6.2.1, there exists at least a path from start node J_0 to end node J_{n-1} in $\mathcal{G}''$. Then it is obvious that J_0 and J_{n-1} must be visited before any backtracking takes place in the DFS and the pre-order traversal (steps 3 and 4 in the preprocessing algorithm). From the preprocessing algorithm described in Section 4.3.1, we can see that the group id will be incremented if and only if a backtracking takes place. Thus the lemma holds. $\#$

Lemma 6.2.3. *In $\mathcal{G}''$, $\forall (J_i, J_j) \in \mathcal{J}'', J_i \xrightarrow{d} J_j \Rightarrow G(J_i) > G(J_j)$*

Proof: It is obvious that the edges belong to the generated spanning tree are labeled with either “m” or “c”. An edge (J_i, J_j) annotated with “m”, “c”, or “d” is denoted as $J_i \xrightarrow{m} J_j$, $J_i \xrightarrow{c} J_j$, or $J_i \xrightarrow{d} J_j$ respectively. If an edge $J_i \xrightarrow{d} J_j$, it means that J_i is visited later than J_j in the pre-order traversal during the preprocessing algorithm (steps 3 and 4). Since group ids are assigned incrementally during the pre-order traversal, the lemma holds. $\#$

For example, in Figure 4.2(d), $6 \xrightarrow{d} 9$, $G(6) = 2$ and $G(9) = 0$, $G(6) > G(9)$ is true.

Lemma 6.2.4. *In $\mathcal{G}''$, $\forall (J_i, J_j) \in \mathcal{J}'', J_i \xrightarrow{m} J_j \Rightarrow G(J_i) = G(J_j)$*

Proof: Since only edges between two subjobs in the same group are labelled with “m”, the lemma holds. $\#$

For example, in Figure 4.2(d), $1 \xrightarrow{m} 4$, $G(1) = 2$ and $G(4) = 2$, $G(1) = G(4)$ is true.

Based on the above lemmas, we have the following theorem:

Theorem 6.2.1. *In $\mathcal{G}''$, $\forall J_i \in \mathcal{J}''$ and $G(J_i) > 0$, there exists a path $\mathbf{p}$ from J_i to J_{n-1} , where for any two consecutive subjobs, e.g., J_{k_q} and $J_{k_{q+1}}$, on the path $\mathbf{p}$, we have $G(J_{k_q}) \geq G(J_{k_{q+1}})$.*

Proof (by contradiction)

Since $G(J_i) > 0$, based on Lemma 6.2.2, $G(J_{n-1}) = 0$, we have $i \neq (n-1)$. According to Lemma 6.2.1, $P(J_i, J_{n-1}) \neq \emptyset$. We prove this Theorem by contradiction and assume that $\forall p \in P(J_i, J_{n-1})$, there exists at least two consecutive subjobs such that $G(J_{k_q}) < G(J_{k_{q+1}})$. Suppose J_{k_q} and $J_{k_{q+1}}$ are the first two such subjobs on path p , $p = (J_i, \dots, J_{k_q}, J_{k_{q+1}}, \dots, J_{n-1})$.

According to Lemma 6.2.3, if $J_{k_q} \xrightarrow{d} J_{k_{q+1}}$, then $G(J_{k_q}) > G(J_{k_{q+1}})$. According to Lemma 6.2.4, if $J_{k_q} \xrightarrow{m} J_{k_{q+1}}$, then $G(J_{k_q}) = G(J_{k_{q+1}})$. Therefore, $J_{k_q} \xrightarrow{c} J_{k_{q+1}}$ must be true. Note that once a subjob has an outgoing edge labeled with “c” in the spanning tree, there must exist another subjob, $J_{k_{q'}}$, $J_{k_{q'}} \neq J_{k_{q+1}}$, such that $J_{k_q} \xrightarrow{m} J_{k_{q'}}$. That is, there exists another path p' , whose sequence is $(J_i, \dots, J_{k_q}, J_{k_{q'}}, \dots, J_{n-1})$. For path p' , again assume that there exists at least two consecutive subjobs such that $G(J_{k_r}) < G(J_{k_{r+1}})$. Suppose J_{k_r} and $J_{k_{r+1}}$ are the first two such subjobs between $J_{k_{q'}}$ and J_{n-1} , $p' = (J_i, \dots, J_{k_q}, J_{k_{q'}}, \dots, J_{k_r}, J_{k_{r+1}}, \dots, J_{n-1})$. Applying the same argument as the above, we then can obtain another path p'' , $p'' = (J_i, \dots, J_{k_q}, J_{k_{q'}}, \dots, J_{k_r}, J_{k_{r'}}, \dots, J_{n-1})$, $J_{k_{r'}} \neq J_{k_{r+1}}$. Obviously, p , p' , and p'' are different paths between J_i and J_{n-1} .

The above procedure can be repeated again and again. However, this is impossible because the number of paths between J_i and J_{n-1} is finite. Therefore, the assumption is not true. There must exist a path $\mathbf{p}$ from J_i to J_{n-1} , where for any two consecutive subjobs, e.g., J_{k_q} and $J_{k_{q+1}}$, on the path $\mathbf{p}$, we have $G(J_{k_q}) \geq G(J_{k_{q+1}})$. $\#$

Corollary 6.2.1. *In $\mathcal{G}$, $\forall J_i \in \mathcal{J}$ and $G(J_i) > 0$, there exists a path $\mathbf{p}$ from J_i to J_{n-1} , where for any two consecutive subjobs, e.g., J_{k_q} and $J_{k_{q+1}}$, on the path $\mathbf{p}$, we have*

$$G(J_{k_q}) \geq G(J_{k_{q+1}}).$$

Proof: Since $\mathcal{G}''$ is created by removing the edges indicating indirect successors from $\mathcal{G}$, it is obvious that $\forall$ edges $(J_i, J_j) \in \mathcal{E}''$, we have $(J_i, J_j) \in \mathcal{E}$. Based on Lemma 6.2.1, the correctness of this corollary is obvious. $\sharp$

$\mathbf{p}$ is called a *propagation path*. For a subjob i , there can be multiple propagation paths. For example, for subjob 1 in $\mathcal{W}_a$ in Figure 4.2 (d), there are three propagation paths: $(1, 7, 8, 9)$, $(1, 4, 7, 8, 9)$, and $(1, 4, 6, 9)$. The set of propagation paths between J_i and J_{n-1} is represented as $PP(J_i, J_{n-1})$. From Theorem 6.2.1, it is obvious that the AC that is finally returned to the user (that is, the original AC created by the user) will contain complete provenance information for the job workflow if partners with a higher group id propagate provenance information to partners with lower group ids during the job workflow execution.

6.2.2 Provenance Recording and Propagation

Based on the corollary described in the last subsection, agents of various AC replicas need to work collectively to record and collect provenance information. Since the contact list is also used by the delegator in the delegator based approach, the algorithm for provenance recording and propagation of job workflow execution using the delegator based coordination will be similar.

Let $\mathcal{R}(J_i)$ denote a subset of J_i 's successors, where for any subjob $J_j \in \mathcal{R}(J_i)$, we have either $(J_i \xrightarrow{m} J_j)$ or $(J_i \xrightarrow{c} J_j)$. Assuming that subjob J_i is under execution and contact list based approach is used for execution coordination, the main steps of provenance recording and propagation are:

- On receiving the “ResultNotification” message from its partner, J_i updates its AC to include the provenance information and updates its contact list accordingly.

- On completion of J_i 's execution, J_i 's corresponding AC agents will record the location of J_i 's execution result into J_i 's AC.
- If $\mathcal{R}(J_i) \neq \emptyset$, as stated in Section 4.3.1, for each subjob J_j , $J_j \in \mathcal{R}(J_i)$, AC agents corresponding to J_i will locate resources, that is, the computational resource, the input data sets from the distributed data repository, and the code from the code repository, for the execution of J_j . This information will be recorded in J_i 's AC. Then, if $|\mathcal{R}(J_i)| > 1$, $|\mathcal{R}(J_i)| - 1$ replicas of J_i 's AC will be created, one for each subjob J_j , $J_j \in \mathcal{R}(J_i)$ and $(J_i \xrightarrow{c} J_j)$.
- Before J_i 's corresponding AC is migrated (or discarded if $\mathcal{R}(J_i) = \emptyset$), J_i 's AC agents will send a "ResultNotification" message to all its partners in the contact list for execution coordination and contact list updating. The message contains the location of J_i 's execution result, and the scheduling information for each J_j , $J_j \in \mathcal{R}(J_i)$. The scheduling information for subjob J_j includes: subjob id, the id of the AC replica to be used to execute the subjob, locations of the selected computational resource, input data sets, and code for the execution of the subjob. In addition, if the partner has a smaller group id, provenance information received by J_i from its partners with a larger group id (recorded in J_i 's AC replica) during J_i 's execution is also piggybacked on the message.

Using the above steps, the provenance information will be recorded in the AC replicas and propagated along propagation paths during the distributed execution of job workflow. Eventually, the AC that is finally returned to the user will contain the complete provenance information about the job workflow execution.

6.3 Performance Evaluation

As explained in the last section, the provenance information is transmitted along with the messages for execution coordination and contact list updating. Although there is no additional message required, the size of message will be increased. There is no centralized server used during the provenance information recording and collection.

Execution provenance information can also be collected using a centralized provenance server which maintains a provenance repository (e.g., using a MySQL^{6.1} database). For each subjob executed, the AC agents need to notify the centralized server about the provenance information. After a job workflow completes its execution, users can then get the provenance information from the server. In this centralized approach, additional messages are required for AC agents to communicate with the provenance server.

Table 6.1: Experiment Results

Task Graph	# of Msgs for Data Dependency and Contact List Updating	# of Msgs for Provenance		Improvement %
		Decentralized	Centralized	
TG0	38	8	33	76%
TG1	5	0	14	100%
TG2	14	0	15	100%
TG3	44	11	31	65%
TG4	12	0	13	100%
TG5	21	7	16	56%

Let $H_p(J_i)$ be the number of hops required to propagate J_i 's execution provenance information between the agents along a propagation path p of J_i (i.e., $p \in PP(J_i, J_{n-1})$) before it reaches the original AC replica. That is, $H_p(J_i)$ is the number of agents traversed during provenance information propagation along path p . Also let $|g_0|$ denote the number of subjobs in the first group g_0 . Assuming that each propagated provenance record, one

^{6.1}<http://www.mysql.com/>

per subjob, is sent individually and independently from the messages used for execution coordination, then the total traffic resulting from the provenance propagation during the job workflow execution will be:

$$\overline{msg} \times \sum_{i=0}^{n-1-|g_0|} \sum_{p \in PP(J_i, J_{n-1})} H_p(J_i) \quad (6.3.1)$$

where $\overline{msg}$ denotes the average size of a provenance message carrying a single provenance record. Obviously, the number of subjobs per AC group and the number of ACs required during the job workflow execution are the major factors that may affect the value of $H_p(J_i)$.

For the centralized model, after a subjob completes its execution, its provenance record needs to be sent to the central provenance server. Assuming that there is no need to collect provenance information for the start and end nodes (since they are assumed to have zero computation cost), the traffic generated in the centralized model can be estimated by:

$$(n - 2) * \overline{msg} \quad (6.3.2)$$

In general, the bandwidth consumed by two or more messages sent separately is larger than that of sending them in a single bundle [30]. Since in the MCCF the propagated provenance records are in fact sent together with the messages used for execution coordination, formula (6.3.1) is an over-estimation of the actual traffic generated. In addition, since communication partners are changed dynamically during the job workflow execution (refer to Section 4.2), the actual propagation path a provenance record travels may also vary (and so does the actual value of $H_p(J_i)$).

Therefore, to evaluate the performance of our distributed provenance collection algorithm, randomly generated Task Graphs (TG), that is, job workflows, were executed in the prototype MCCF system on a cluster of computers. The six pseudo-random TGs generated in Section 4.4.3 are used as simulated job workflows.

The number of messages for execution coordination and contact list updating is illustrated in Table 6.1. Those messages are required no matter whether a decentralized or centralized method is used. In the decentralized method, the propagated provenance records are in fact piggybacked in the messages for execution coordination and contact list updating. During a job workflow execution, the messages that carry the propagated provenance records and the number of provenance records carried by such message were tracked. In general the bandwidth consumed by two or more messages sent separately is larger than that of sending them in a single bundle [30]. For a fair comparison with the centralized method, if each propagated provenance record was carried by a separate provenance message, the total number of such messages in the decentralized method would be the summation of the number of propagated provenance records contained in all the tracked messages. Each job workflow was executed 3 times, and the average numbers are shown in Table 6.1. Note that for the centralized method, formula (6.3.2) is used to calculate the number of messages generated for the provenance collection.

Table 6.1 also shows the percentage improvement of the decentralized method over the centralized one. From these results, we observe that the centralized model always generates a higher number of messages for provenance information recording. Particularly, by recording the provenance information in the workflow specification in the AC, a provenance record does not always need to be propagated during a job workflows' execution. For example, no provenance record is propagated during the execution of task graphs TG1, TG2, and TG4 (and thus there is no provenance message required). In the execution of these task graphs, the agents of the AC replicas created during runtime are the partners of the agents of the original AC. Therefore, propagation of provenance information is not required. In this case, the percentage improvement of the decentralized model over the centralized model is 100%.

6.4 Summary

Mobile agent-based distributed job workflow execution hides scientists from the Grid details, but also hides how the result is achieved (that is, the provenance of the job workflow execution). Since data processing in scientific computing may require some level of validation and verification, information on the services and data sets used during the workflow execution is required. The provenance in many existing scientific workflow engines relies on a centralized provenance collection server for provenance recording and collection. However, in mobile agent-based distributed job workflow execution, there is no centralized workflow engine and thus naturally the provenance recording and collection should also be carried out in a distributed manner.

By studying the agent communication in the MCCF and the properties of the pre-processing algorithm for partner identification, a distributed provenance recording and collection mechanism has been developed. The subjob provenance information is transmitted along the provenance propagation paths. Since provenance information is piggy-backed with the messages for execution coordination and contact list updating, there is no additional message required. To evaluate our approach, an experimental study has been carried out on randomly generated job workflows. The results show that our approach has less communication overhead than the one using a centralized provenance server for provenance information recording and collection.

Chapter 7

A Comparison Study between Centralized and Distributed Job Workflow Execution Models

As discussed in Section 2.2.2, job workflow execution can be classified into centralized execution models and distributed execution models. The centralized execution model may cause problems such as a single point of failure and poor scalability; whereas the distributed job workflow execution model may bring additional runtime overhead, e.g., execution coordination, as discussed in Chapters 4 and 5. The MCCF is our framework to implement the distributed job workflow execution model and Condor DAGMan is a commonly used centralized job workflow execution model. This chapter describes a comparison study between these two execution models.

7.1 Introduction to Condor DAGMan

Condor Directed Acyclic Graph Manager (DAGMan) [4] is a commonly used Grid-based workflow execution engine. Condor [2] is a specialized resource management system, which constructs a distributed high-throughput computing facility by collecting distributively owned workstations and dedicated clusters. Condor-G [3] is a grid enabled version

of Condor, which augments the Globus toolkit with a reliable job submission mechanism. Condor DAGMan uses the centralized model to manage the execution of a job workflow and uses Condor and Condor-G to enable the job execution on the cluster and Grid environment respectively. Normally, the submission host schedules and coordinates the execution of subjobs, and stages the executable codes to the scheduled computational resource for execution. In addition, the submit host also manages the data transfer between computational resources. Therefore, Condor DAGMan implements Centralized-P2P-U_DS job workflow execution.

The workflow specifications submitted to Condor DAGMan are concrete workflow specifications (see Section 1.1.1), which require physical locations of input/output data, executable codes, and execution platforms. For a given job workflow, there is one general Condor DAGMan submit file that defines all subjobs and the subjob data dependencies. It contains the following information (see Appendix A.1 for a general Condor DAGMan submit file for a job workflow with task graph TG4 (which is illustrated in Appendix A.1)):

- A list of the subjobs.
- The tasks to be executed before a subjob's execution ("PRE" scripts) or after a subjob's execution ("POST" scripts).
- The subjob data dependencies.

For each subjob there is a Condor submit description file, which specifies the operations to be performed on the given remote machine (see Appendix A.2 for an example for subjob 4 in the job workflow with task graph TG4). Condor DAGMan does not support automatic intermediate data movement, users have to describe the process of the stage-in and stage-out of files which are required by the subjob execution. Thus, "PRE" or "POST" scripts can be used to stage the required data in or out the execution host (see Appendix A.3 for an example, which specifies the script to be executed before the

execution of subjob 4 in the job workflow with task graph TG4). Data are transferred between the execution hosts directly.

7.2 Job Workflow Execution on Condor DAGMan and the MCCF

The simulation study in Section 2.3 was carried out to compare the makespan of the following job workflow execution models: Centralized-Centralized-SS, Distributed-P2P-SS model with LMA, Centralized-Centralized-DS, Distributed-P2P-U_DS model with MMA, and Distributed-P2P-DS model with LMA. The results show that the Distributed-P2P-DS model with LMA has better performance than that of the other four models simulated for pipeline job workflows. The MCCF is our framework to implement the Distributed-P2P-DS model with LMA. As discussed in the previous section, Condor DAGMan implements the Centralized-P2P-U_DS model, which was not included in the simulation study. In order to complement the simulation study, a comparison study between job workflow execution on Condor DAGMan and the MCCF is carried out in this chapter.

Figures 7.1(a) and 7.1(b) illustrate how a simple job workflow, which consists of two subjobs J_1 and J_2 with $J_1 < J_2$, is executed on Condor DAGMan and the MCCF respectively. The submit host acts as the centralized server, which manages the data dependency between subjobs, stages the required executable codes and data sets in the selected execution host, manages the transfer of intermediate results between execution hosts, and submits the subjob execution to the remote execution host. As illustrated in Figure 7.1 (a), for Condor DAGMan, all the control messages are handled by the submit host. In comparison, Figure 7.1 (b) illustrates how the same job workflow is executed on the MCCF. To be fair, the submit host is assumed to act as the code repository. Figure 7.1 (b) shows that the control messages for the MCCF are distributed on all execution hosts during the job workflow execution.

Figure 7.1: Workflow Execution over Condor DAGMan and the MCCF

From Figures 7.1 (a) and (b), it can be seen that there is almost the same number of control messages over the network during the job workflow execution in Condor DAGMan as that in the MCCF. When there are multiple data independent subjobs, AC replications are created in the MCCF, the control messages for these subjobs's execution will be managed by several groups of distributed AC agents simultaneously. However, in the Condor DAGMan, all the control messages for the execution of the data independent subjobs are handled sequentially. This makes the submit host the system bottleneck.

For the MCCF, the light weight mobile agent mechanism reduces the number of

control messages transferred over the network. However, it brings additional overhead:

- The overhead caused by AC migration;
- The delay caused by AC agents constructed on each host on AC arrival;
- The overhead for execution coordination (refer to Chapter 4 and Chapter 5)).

To evaluate the performance of the MCCF and Condor DAGMan, an experimental comparison study is carried out to compare the makespan of the simulated job workflow execution over the MCCF and Condor DAGMan.

7.3 Performance Evaluation

The MCCF prototype system (using dynamic partner identification, delegator based execution coordination, and distributed provenance recording and collection) and Condor DAGMan were deployed to the emulated WAN using the ModelNet. The configuration of the Modelnet testbed is the same as the one used in Section 4.4.1, except that 4 Fedora Core 2 machines are used (instead of 8). The network topology used in the WAN emulation experiment is also the same as the one described in Section 4.4.1, except that 16 machines are used as the client (instead of 20). Except for the emulator, all 16 client machines are installed with Globus2.2 and Personal Condor 6.6.1 or Condor 6.6.5 using Condor's default configuration. Both MCCF and Condor DAGMan here use GridFtp for code/data transmission and GSI for authentication and authorization.

Resource scheduling may have an effect on job workflow execution. However, resource scheduling is not the research focus of this thesis. In order to avoid the undetermined factors that may affect a job workflow makespan and focus on the evaluation of the benefit/cost of distributed job workflow execution in terms of the number of control messages

and control threads generated to handle the job workflow execution, the machines were dedicated for our use during our experiments and the resources (e.g., code repository, data repository, and computational resource) for each subjob execution are predetermined.

We use the 6 pseudorandom TGs generated in Section 4.4.3 for the comparison study. Each subjob in the workflow executes the same executable file and generates a 28000-byte output file, which is the input of its successors (if any). Since the objective is to evaluate the overhead of control messages and communication messages transmission over the network, it is important to vary the complexity of the topologies of the job workflows. What each subjob executes is not really relevant.

The number of control threads (in the MCCF, this is equal to the number of AC replicas), the number of control messages sent over the network, the maximum number of messages handled by an engine^{7.1}, and the makespan of each simulated job workflow are shown in Table 7.1. In addition, Table 7.1 also shows the percentage improvement of the makespans of the job workflows executed on the MCCF over that on Condor DAGMan. From these results, we observe that:

Table 7.1: Performance Evaluation on Condor DAGMan and the MCCF.

TG ID	# of Tasks		# of Control Threads		# of Control Messages			Makespan (s)		
	DAGMan	MCCF	DAGMan	MCCF	DAGMan	MCCF		DAGMan	MCCF	%
				(# of AC)	(on submit host)	max per AC agent	overall			Improvement
TG0	33	35	1	22	246	34/86%	248	2346	1795	23 %
TG1	14	16	1	6	120	14/88%	122	2660	1906	28 %
TG2	15	17	1	11	114	18/84%	116	1259	991	23 %
TG3	31	33	1	11	218	18/92 %	220	2471	1571	36 %
TG4	13	15	1	5	88	10/89 %	90	1988	1174	41 %
TG5	16	18	1	6	114	10/91 %	116	1510	1160	23 %

^{7.1}An engine refers to the AC agents in the MCCF, and Condor DAGMan on the submit host in Condor DAGMan.

- On the MCCF, there are two more tasks for each random task graph because of the added hypothetical starting node and end node. Meanwhile, there can be multiple control threads, i.e., multiple AC replicas, for job workflow execution. However, on Condor DAGMan, there is always only one centralized control thread. When there are multiple data independent subjobs ready for execution, multiple control threads can handle the control messages for these subjobs' execution concurrently, which can improve the job workflow execution time.
- Compared with Condor, the distributed job workflow management in the MCCF requires two additional control messages to return the original AC to the submit host. So, the total number of control messages is more than that handled by the submit host in Condor DAGMan. However, the maximum number of messages handled by an AC agent in the MCCF is much less than that in Condor DAGMan, the improvement ranges from 84% to 92% for the task graphs used in the experiments. All the control messages are handled by the submit host in Condor DAGMan and this makes Condor DAGMan the system bottleneck.

In all, because of the distributed nature of Grid resources, for a large job workflow, there can be a non-negligible overhead involved in sending the control messages over the network for the centralized model. In addition, by decentralizing and paralleling the management of the control messages using light weight mobile agent technology, the MCCF achieves a better job workflow makespan than Condor DAGMan, the improvement is 23 % to 41 %, as shown in Table 7.1.

7.4 Summary

Job workflow execution can be classified into centralized execution models and distributed execution models. A centralized execution model may cause problems such as single point

of failure and poor scalability. There is no centralized server in the MCCF. However, it brings additional runtime overhead, e.g., the overhead caused by AC migration and the delay caused by AC agents constructed on each host on AC arrival. By studying a job workflow execution using the MCCF and Condor DAGMan, we first analyzed the benefits and costs of distributed job workflow execution. Then, we carried out an empirical comparison study on these two workflow engines using the simulated job workflows. The experiment results show that the MCCF achieves a better job workflow makespan.

However, compared to a centralized job workflow execution model, the MCCF has the following limitations:

- **Dynamic workflow:** “Dynamic workflow” was proposed in business process management to handle workflows that are subject to changes in a highly dynamic business environment [91]. It refers to runtime modification of a workflow to dynamically alter the subjob functionality, delete/add a subjob, delete/add a data dependency, or change the user policies for resource scheduling. Particularly, in a dynamic computing environment like the Grid, runtime optimization can be achieved by altering the workflow topology. For example, in order to reduce the workload on a certain machine, a subjob can be decomposed into two smaller subjobs, that can be executed simultaneously on different machines [57]. Such runtime optimization leads to deleting at runtime an original subjob and adding two new subjobs. Such dynamic changes present a great challenge to the MCCF. With dynamic resource selection and lack of a centralized management server, it is hard to arbitrarily locate an AC replica and notify its corresponding AC agents about the changes of the job workflow. In addition, modifying data dependency between subjobs may also change the partnership between AC replicas.
- **Workflow execution monitoring/tracking/recovering:** Similar to the above

discussion, it is difficult to implement job workflow execution monitoring and tracking in the MCCF, because of the lack of a centralized server. Particularly, centralized management like Condor DAGMan, adopts centralized provenance recording and collection. Therefore, upon abnormal termination of a subjob execution, Condor DAGMan knows where to recover the job workflow execution. However, this is very difficult to achieve in the context of mobile agent based distributed job workflow execution using decentralized provenance recording and collection.

Chapter 8

Conclusions

8.1 Summary of Contributions

The main contributions of this thesis are:

- **Mobile Code Collaboration Framework**

A taxonomy of job workflow execution models has been developed based on three elements, i.e., intermediate data movement, subjob code acquisition, and control management and mapped to the existing job workflow management systems. In addition, to evaluate the benefits and overhead of different execution models, we carried out a simulation study to compare the makespan of five execution models using pipeline job workflows: Centralized-Centralized-SS, Distributed-P2P-SS model with LMA, Centralized-Centralized-DS, Distributed-P2P-U_DS model with MMA, and Distributed-P2P-DS model with LMA. The simulation results show that the Distributed-P2P-DS model with LMA has better performance than the other models simulated. However, currently there does not exist a job workflow management system that supports this execution model.

In this thesis, a Mobile Code Collaboration Framework (MCCF) to support Distributed-P2P-DS model with LMA job workflow execution has been developed. To reduce

the communication overhead caused by data movement and to provide decentralized control of execution during workflow enactment, the MCCF is developed to map the execution of subjobs to the distributed resources and to coordinate the subjobs' execution at runtime according to the abstract workflow provided by users. LMA and CoD techniques are adopted in the development of the MCCF, so that an analysis module in data intensive scientific applications can be executed at a computational resource close to where the required data set is located.

In concept, the MCCF provides a two-level job workflow program model: component developers develop executable code for dynamic services and application developers just need to specify the input data and code description for subjob execution, and the data dependency among the subjobs. This will greatly reduce the effort required for application developers to "gridify" their applications. As a proof of concept, a simple prototype of the MCCF, utilizing LMA and dynamic services, has been developed using Globus Toolkit.

- **Execution Coordination in Mobile Agent based Distributed Job Workflow Execution**

When multiple data independent subjobs can be executed concurrently, replicas of an existing AC will be generated so that there is one AC for each subjob. The AC replicas will then be migrated to the different computational resources for the execution of these data independent subjobs in parallel. When multiple concurrently executing subjobs have a common immediate successor, only one of the corresponding AC replicas should be selected for the latter's execution. Others should be discarded.

During a job workflow's execution in the MCCF, it is necessary for the agents executing a subjob to locate the input data from its predecessors. If the subjob's predecessor is executed by the same AC, the location of the input data from its pre-

decessor is readily available in the AC. Otherwise, the agents executing the subjob need to get the location of its predecessor's execution result through an execution coordination mechanism. This thesis makes three important contributions in the field of execution coordination for mobile agent based distributed job workflow execution:

1. **A taxonomy of execution coordination techniques:** Execution coordination of mobile agents is a well researched area. There are many techniques that have been developed. We give a classification of these techniques according to how they can be applied in the mobile agent-based distributed job workflow execution. The discussion about the techniques (see Section 4.1) suggests that we only consider direct coordination techniques for the MCCF. When a direct coordination technique is used, agents need to obtain the location of the subjob's execution result through communication.
2. **Dynamic partner identification for direct execution coordination:** A subjob grouping algorithm for preprocessing the job workflow's static specification is proposed. The obtained information is then used in both the static and dynamic algorithms for communication partners identification. MA dynamic location and communication based on this approach is expected to reduce the agent communication overhead. In addition, the proof of the correctness and effectiveness of the dynamic algorithm are elaborated. A performance comparison study of the simulated job workflows on the actual implementation of the MCCF and the algorithms has been conducted. The results show that the algorithm is scalable and efficient.
3. **A delegator based direct coordination technique:** To further analyze the performance of an execution coordination technique, how existing techniques can be implemented in the MCCF are investigated. The investigation results

show that the existing techniques, such as middle agent based or contact list based approaches, either rely on a centralized mechanism for execution coordination or introduce substantial overhead when applied in the MCCF. Taking the combined advantages of the existing techniques, an original decentralized execution coordination technique, delegator based coordination, is proposed. Each mobile agent is associated with a delegator, which is responsible for the execution coordination on behalf of its owner using a decentralized approach. Therefore, a mobile agent can be isolated from execution coordination and achieve better performance in workflow execution. A performance study has been conducted to evaluate our approach against some existing ones using real and simulated job workflows. The results are presented and discussed in this thesis.

- **Provenance Recording and Collection in Mobile Agent-Based Distributed Job Workflow Execution:** Job workflow systems automate the execution of scientific applications, however, they may hide how the results are achieved. This thesis proposes the development and evaluation of a decentralized recording and collection scheme for job workflow provenance in mobile agent-based distributed job workflow execution. First, the differences between job workflow provenance in our context and those in the existing workflow systems are discussed. The MCCF uses direct agent communication for execution coordination during the job workflow execution, in which a preprocessing based communication partner identification algorithm is used. By further exploiting the properties of the partner identification algorithm, paths for execution provenance information propagation are identified, so that the provenance information is delivered along the paths together with the communication messages for execution coordination and contact list updating. A performance study was conducted to evaluate our approach against the one using

a centralized provenance server. The results show that our approach generates less communication overhead.

- **Performance Study:** In order to evaluate the performance of the MCCF, an experimental study is carried out using simulated job workflows executed on the prototype implementation of the MCCF and the Condor Directed Acyclic Graph Manager (DAGMan), a commonly used Grid-based workflow execution engine. The results show that the MCCF achieves better job workflow execution time.

8.2 Recommendations for Future Work

The MCCF is not yet a complete system. Our research to date focuses on the development of an LMA based workflow management infrastructure and the algorithms under the framework to support: (i) agent replication, migration, and destruction; (ii) direct execution coordination; and (iii) provenance recording and collection. However, there can be further optimization on our current proposed algorithms:

- **Partner Identification:** The current optimization on partner identification is based on the assumption that the number of AC replicas and the subjobs assigned to an AC replica are fixed. The runtime partner set updating algorithm is optimal for the given grouping of subjobs. For a DAG, there may exist other subjob grouping algorithms that are different from the one introduced in Subsection 4.3.1. Hence, further investigation should be carried out on how the number of AC replicas and the subjobs assigned to an AC replica may affect the cost for communication as well as the execution.
- **Execution Coordination Model:** In our experiments, the MBLocator, mailboxes and delegators are assumed to remain at fixed locations. In a dynamic environment,

such as the Grid, this assumption might not hold. Further investigation on how their migration may affect the overall system performance should be carried out.

- **Provenance Recording and Collection:** In the current algorithm, a subjob's provenance information might be propagated along multiple propagation paths. For future work, the unique shortest propagation path for a given subjob will be identified. This will further reduce the communication cost caused by propagation of provenance information.

In addition, further research and implementation work along the following directions could also be carried out:

- **Code & Data Repositories:** Algorithms for replicating data and code need to be investigated. Efficient and secure mechanisms for data transfer and code upstaging (e.g., using GridFTP) need to be developed. Admission control policies for data repositories and security mechanisms for CoD also need to be investigated.
- **Resource Scheduling Algorithms:** The benefits of dynamic mapping of job workflow to resources are twofold: i) it allows dynamic adaptation to the changes of the shared resources; and ii) it enables the subjob to move closer to where the required data set is located so as to reduce data movement. To realize these benefits, dynamic resource scheduling algorithms need to be developed to identify the location of the required data set and the subjob specific code, and the computational resource for subjob execution. This is a challenging task given the dynamic nature of resources, the possible replications of the code and the data set.
- **Security:** The MCCF adopts mobile code for subjob execution. In the current MCCF implementation, only GSI authentication and authorization are used to make sure only authorized users can enact the workflow execution. In addition, we

assume that the executable codes are stored at trusted code repositories. However, there are known open security issues that have limited the use of mobile codes for distributed computing, e.g., denial of service, disclosure of confidential information damage, or modification of data, etc [40]. Some security mechanisms, e.g., code signing and proof carrying (a survey of such mechanisms can be found in [97]), have been proposed to improve the degree of trustworthiness of mobile code. From a user's point of view, CoD is often considered to be more secure and trustworthy than REV since the resource receiving the code is the one that required the code [105]. Usually, before a code is selected and downloaded, there will be some measurement to ensure that the code is as safe as possible, e.g., from a trusted remote site. REV on the other hand allows a system to receive codes from any remote resources for execution. Particularly, in the context of the MCCF, how to ensure the AC's integrity on a malicious computational resource, on which AC agents are created, could be a challenging problem.

8.3 Summary

This thesis presents the design, implementation, and evaluation of a mobile code collaboration framework. This framework facilitates the dynamic mapping of the job workflow to distributed resources, the coordination of subjobs' execution, and the collection of provenance information during the job workflow execution. In particular, a dynamic communication partner identification algorithm, a delegator based execution coordination approach, and a decentralized provenance recording and collection scheme are proposed and evaluated in the thesis.

Appendix A

Condor DAGMan Scripts

Figure A.1: Random Task Graph TG4

Table A.1: General DAGMan Submit File for TG4

```

# General DAGMan submit file Corresponding to TG4

#The list of subjobs:
Job TG4_1 /home/yuhongf/Condor/dagman/randomTG4/tg4_1/tg4_1.submit
Job TG4_2 /home/yuhongf/Condor/dagman/randomTG4/tg4_2/tg4_2.submit
Job TG4_3 /home/yuhongf/Condor/dagman/randomTG4/tg4_3/tg4_3.submit
Job TG4_4 /home/yuhongf/Condor/dagman/randomTG4/tg4_4/tg4_4.submit
Job TG4_5 /home/yuhongf/Condor/dagman/randomTG4/tg4_5/tg4_5.submit
Job TG4_6 /home/yuhongf/Condor/dagman/randomTG4/tg4_6/tg4_6.submit
Job TG4_7 /home/yuhongf/Condor/dagman/randomTG4/tg4_7/tg4_7.submit
Job TG4_8 /home/yuhongf/Condor/dagman/randomTG4/tg4_8/tg4_8.submit
Job TG4_9 /home/yuhongf/Condor/dagman/randomTG4/tg4_9/tg4_9.submit
Job TG4_10 /home/yuhongf/Condor/dagman/randomTG4/tg4_10/tg4_10.submit
Job TG4_11 /home/yuhongf/Condor/dagman/randomTG4/tg4_11/tg4_11.submit
Job TG4_12 /home/yuhongf/Condor/dagman/randomTG4/tg4_12/tg4_12.submit
Job TG4_13 /home/yuhongf/Condor/dagman/randomTG4/tg4_13/tg4_13.submit
Job TG4_14 /home/yuhongf/Condor/dagman/randomTG4/tg4_14/tg4_14.submit

#The tasks to be executed before a subjob execution:
Script PRE TG4_1 /home/yuhongf/Condor/dagman/randomTG4/tg4_1/tg4_1.pre
Script PRE TG4_2 /home/yuhongf/Condor/dagman/randomTG4/tg4_2/tg4_2.pre
Script PRE TG4_3 /home/yuhongf/Condor/dagman/randomTG4/tg4_3/tg4_3.pre
Script PRE TG4_4 /home/yuhongf/Condor/dagman/randomTG4/tg4_4/tg4_4.pre
Script PRE TG4_5 /home/yuhongf/Condor/dagman/randomTG4/tg4_5/tg4_5.pre
Script PRE TG4_6 /home/yuhongf/Condor/dagman/randomTG4/tg4_6/tg4_6.pre
Script PRE TG4_7 /home/yuhongf/Condor/dagman/randomTG4/tg4_7/tg4_7.pre
Script PRE TG4_8 /home/yuhongf/Condor/dagman/randomTG4/tg4_8/tg4_8.pre
Script PRE TG4_9 /home/yuhongf/Condor/dagman/randomTG4/tg4_9/tg4_9.pre
Script PRE TG4_10 /home/yuhongf/Condor/dagman/randomTG4/tg4_10/tg4_10.pre
Script PRE TG4_11 /home/yuhongf/Condor/dagman/randomTG4/tg4_11/tg4_11.pre
Script PRE TG4_12 /home/yuhongf/Condor/dagman/randomTG4/tg4_12/tg4_12.pre
Script PRE TG4_13 /home/yuhongf/Condor/dagman/randomTG4/tg4_13/tg4_13.pre
Script PRE TG4_14 /home/yuhongf/Condor/dagman/randomTG4/tg4_14/tg4_14.pre

#The subjob data dependencies
Parent TG4_1 Child TG4_3
Parent TG4_1 Child TG4_2
Parent TG4_3 Child TG4_4
Parent TG4_4 Child TG4_7
Parent TG4_4 Child TG4_6
Parent TG4_7 Child TG4_9
Parent TG4_2 TG4_6 Child TG4_8
Parent TG4_9 Child TG4_11
Parent TG4_11 Child TG4_13
Parent TG4_11 Child TG4_12
Parent TG4_8 Child TG4_10
Parent TG4_3 Child TG4_5
Parent TG4_5 TG4_10 TG4_13 TG4_12 Child TG4_14

#Retry each subjob 3 times if failure takes place
Retry TG4_1 3
...

```

Table A.2: Submit File for Subjob 4 in TG4

```
#TG4 subjob 4 submit script
Executable = /bin/sh

#Executable codes are dynamically staged in "PRE" scripts
#U_DS subjob code acquirement, See "PRE" scripts
transfer_executable = False
Universe = globus
Globusscheduler = 10.0.0.45/jobmanager-condor
GlobusRSL = (condor_submit=(Universe vanilla))
Arguments = /tmp/run.sh JobExec \"/home/yuhongf/Condor/Data/datasetsI\" \"4\" \"4\"
environment = CLASSPATH=/tmp; PATH=/usr/local/globus-2.2/bin

#Intermediate files are transferred directly between selected computational resources
#For P2P Intermediate data movement, see "PRE" scripts
transfer_files = NEVER

#Provenance Information Recording
error = /home/yuhongf/Condor/dagman/randomTG4/tg4_4/tg4_4.error
log = /home/yuhongf/Condor/dagman/randomTG4/tg4_4/tg4_4.log

Queue 1
```

Table A.3: "PRE" Script for Subjob 4 in TG4

```
#!/bin/sh
#Dynamically stage in subjob 4's executable codes
#The scripts code for "LoadDataSet_ntu.sh" is described in Table9.4
#It uses GridFtp for staging in required data sets or executable codes
/home/yuhongf/Condor/dagman/LoadDataSet_ntu.sh
  #the codes' source host: submit host
  "pdcc18.ntu.edu.sg"
  #the codes destination host: selected computational resource for subjob 4's execution
  "pdcc12.ntu.edu.sg"
  #executable codes' source directory
  "/home/yuhongf/Condor/dagman/JobExec.class"
  #executable codes' destination directory
  "/tmp/JobExec.class"
/home/yuhongf/Condor/dagman/LoadDataSet_ntu.sh
  "pdcc18.ntu.edu.sg"
  "pdcc12.ntu.edu.sg"
  "/home/yuhongf/Condor/dagman/run.sh"
  "/tmp/run.sh"

#Transfer Subjob 4's predecessors' execution result
/home/yuhongf/Condor/dagman/LoadDataSet_ntu.sh
  "pdcc11.ntu.edu.sg"
  "pdcc12.ntu.edu.sg"
  "/tmp/resultTG4_3"
  "/tmp/resultTG4_3_4"
```


Table A.4: Script for LoadDataSet_ntu.sh

```
#!/bin/sh

#Gridftp is used for data or executable codes transfer

#$1 is the source host

#$2 is the destination host

#$3 is the source file (with absolute directory)

#$4 is the destination file (with absolute directory)

globus-url-copy gsiftp://$1$3 gsiftp://$2$4
```

Appendix B

Publication List for this Research

B.1 Journal

1. **Yu-Hong Feng** and Wen-Tong Cai and Jian-Nong Cao, “Dynamic partner identification in mobile agent-based distributed job workflow execution”, *Journal of Parallel and Distributed Computing*, vol. 67, no. 11, November, 2007, pp 1137-1154. doi:10.1016/j.jpdc.2007.06.012
2. **Yu-Hong Feng** and Wen-Tong Cai, “Execution Coordination in Mobile Agent based Distributed Job Workflow Execution”, accepted by *Journal of Systems Architecture*.

B.2 Conference

1. **Yu-Hong Feng** and Wen-Tong Cai and Jian-Nong Cao, “A Simulation Study of Job Workflow Execution Models over the Grid”, In *proceedings of Grid and Cooperative Computing: Second International Workshop (GCC2003)*, Shanghai, China, December 2003. Volume 3033 of Lecture Notes in Computer Science (LNCS), pp. 935–943.

2. **Yu-Hong Feng** and Wen-Tong Cai, “MCCF: A Distributed Grid Job Workflow Execution Framework”, In *proceedings of the 2th International Symposium on Parallel and Distributed Processing and Applications (ISPA’2004)*, Hong Kong, China, December 2004. Volume 3358 of Lecture Notes in Computer Science (LNCS), pp. 274–279.
3. **Yu-Hong Feng** and Wen-Tong Cai and Jian-Nong Cao, “Communication Partner Identification in Distributed Job Workflow Execution Over the Grid”, In *proceedings 3rd International Workshop on Mobile Distributed Computing (in conjunction with the 25th IEEE International Conference on Distributed Computing Systems) (ICDCSW’05)*, Columbus, Ohio, USA, June, 2005. Volume 06 of IEEE Computer Society, pp. 587–593.
4. **Yu-Hong Feng** and Wen-Tong Cai, “Provenance Provisioning in Mobile Agent-Based Distributed Job Workflow Execution”, In *proceedings of International Conference on Computational Science 2007 (ICCS 2007) (1)*, Beijing, China, May, 2007. pp. 398–405.
5. **Yu-Hong Feng** and Wen-Tong 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)*. Biopolis, Singapore, June, 2007. pp. 76–90.

Bibliography

- [1] Ajanta Mobile Agents Research Project. <http://www.cs.umn.edu/Ajanta/>, last visited on 03.11.2007.
- [2] Condor. <http://www.cs.wisc.edu/condor/>, last visited on 03.11.2007.
- [3] Condor-G. <http://www.cs.wisc.edu/condor/condorg/>, last visited on 03.11.2007.
- [4] Directed Acyclic Graph Manager (DAGMan). <http://www.cs.wisc.edu/condor/dagman/>, last visited on 03.11.2007.
- [5] Foundation for Intelligent Physical Agents (FIPA). <http://www.fipa.org>, last visited on 03.11.2007.
- [6] FIPA-OS. <http://fipa-os.sourceforge.net/tutorials.htm>, last visited on 03.11.2007.
- [7] JACK - Agent solutions. <http://weblogs.asp.net/edaniel/archive/2004/10/21/245979.aspx>, last visited on 03.11.2007.
- [8] Java Agent DEvelopment Framework (JADE). <http://jade.tilab.com/>, last visited on 03.11.2007.
- [9] Java CoG Kit. http://wiki.cogkit.org/index.php/Main_Page, last visited on 03.11.2007.

-
- [10] NOMADS: A Java Based Mobile Agent System. <http://www.ihmc.us/research/projects/Nomads/>, last visited on 17.11.2007.
 - [11] Network Simulator - ns-2. <http://www.isi.edu/nsnam/ns/>, last visited on 03.11.2007.
 - [12] ObjectSpace Voyager ORB 3.1.1. <http://www.inf.fu-berlin.de/lehre/WS99/VS/Misc/Voyager/API/>, last visited on 17.11.2007.
 - [13] The BioAgent Project. <http://www.bioagent.net/>, last visited on 04.11.2007.
 - [14] The Java language: An overview, 1995. Sun Microsystems, <http://java.sun.com/docs/overviews/java/java-overview-1.html>, last visited on 03.11.2007.
 - [15] Tryllian. <http://www.tryllian.com/>, last visited on 03.11.2007.
 - [16] Zeus. <http://labs.bt.com/projects/agents/zeus/>, last visited on 03.11.2007.
 - [17] V.S. Adve, R. Bagrodia, J.C. Browne, E. Deelman, A. Dube, E.N. Houstis, J.R. Rice, R. Sakellariou, D.J. Sundaram-Stukel, P.J. Teller, and M.K. Vernon. POEMS: end-to-end performance design of large parallel adaptive computational systems. *IEEE Transactions on Software Engineering*, 26(11):1027–1048, November 2000.
 - [18] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke. GridFTP: Protocol Extensions to FTP for the Grid, March 2001. <http://www-fp.mcs.anl.gov/dsl/GridFTP-Protocol-RFC-Draft.pdf>, last visited on 03.11.2007.
 - [19] J. Almond and D. Snelling. UNICORE: Secure and Uniform Access to Distributed Resources via the World Wide Web. White Paper, October 1998.
 - [20] J. Altmann, F. Gruber, L. Klug, W. Stockner, and E. Weippl. Using Mobile Agents in Real World: A Survey and Evaluation of Agent Platforms. In *proceedings of the*

- 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS at the 5th International Conference on Autonomous Agents*, pages 33–39, Montreal, Canada, May 2001. ACM Press.
- [21] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services Version 1.1*, May 2003. <http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf>, last visisted on 03.11.2007.
- [22] J. Annis, S. Kent, F. Castander, D. Eisenstein, J. Gunn, R. Kim, R. Lupton, R. Nichol, M. Postman, and W. Voges. The maxBCG Technique for Finding Galaxy Clusters in SDSS Data. *Bulletin of the American Astronomical Society*, 31:1391, 1999.
- [23] J. Annis, Y. Zhao, J. Voekler, M. Wilde, S. Kent, and I. Foster. Applying Chimera Virtual Data Concepts to Cluster Finding in the Sloan Sky Survey. In *proceedings of Supercomputing, ACM/IEEE 2002 Conference*, pages 56–56, 2002.
- [24] M. Atkinson. Structured Data and the Grid Access and Integration, June 2003. <http://www.nesc.ac.uk/talks/mpa/HPDC12Seattle23June03V2.pdf>, last visited on 04.11.2007.
- [25] V. Atluri, S.A. Chun, and P. Mazzoleni. A Chinese wall security model for decentralized workflow systems. In *proceedings of ACM Conference on Computer and Communications Security*, pages 48–57, 2001.
- [26] R. Aversa, B.D. Martino, N. Mazzocca, and S. Venticinque. MAGDA: A Mobile Agent based Grid Architecture. *Journal of Grid Computing*, 4(4):395–412, December 2006.

-
- [27] A. Barker and J. van Hemert. Scientific Workflow: A Survey and Research Directions. Proceedings of the 3rd Grid Applications and Middleware Workshop (GAMW'07), 2007.
- [28] E. Bartocci, F. Corradini, and E. Merelli. Enacting Proactive Workflows Engine in e-Science. In *proceedings of International Conference on Computational Science (3)*, pages 1012–1015, 2006.
- [29] N.M. Belaramani, Y. Chow, V.W.M. Kwan, C.L. Wang, and F.C.M. Lau. A Component-based Software Architecture for Pervasive Computing. In *proceedings of Intelligent Virtual World: Technologies and Applications in Distributed Virtual Environments*, pages 201–222, 2002.
- [30] M. Berger. Multipath Packet Switch Using Packet Bundling. In *proceedings of High Performance Switching and Routing (Workshop on Merging Optical and IP Technologies)*, pages 244–248, Kobe, Japan, 2002.
- [31] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Computing Applications(JHPCA)*, 15(4):327–344, 2001.
- [32] V. Bhat, S. Klasky, S. Atchley, M. Beck, D. McCune, and M. Parashar. High performance threaded data streaming for large scale simulations. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 243–250, November 2004.
- [33] W. Binder, G.D.M. Serugendo, and J. Hulaas. Towards a Secure and Efficient Model for Grid Computing using Mobile Code. Proceedings of the 8th ECOOP
-

- Workshop on Mobile Object Systems: Agent Applications and New Frontiers, June 2002. <http://citeseer.ist.psu.edu/cache/papers/cs/26715/http:zSzzSzcuil.unige.chzSz~ecoopwszSzws02zSz..zSzsub02zSzGrid.pdf/binder02towards.pdf>, last visited on 08.11.2007.
- [34] J. Blower, K. Haines, and E. Llewellyn. Data streaming, workflow and firewall-friendly Grid Services with Styx. Proceedings of the UK e-Science All Hands Meeting, September 2005. http://www.resc.rdg.ac.uk/publications/Blower_AHM_2005.pdf, last visited on 08.11.2007.
- [35] R. Bose and J. Frew. Composing Lineage Metadata with XML for Custom Satellite-Derived Data Products. In *proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, pages 275–284, Washington, DC, USA, 2004.
- [36] R. Brandt. Dynamic Adaptation of Mobile Code in Heterogeneous Environments. Technical report, Technische University Munich, February 2001.
- [37] R. Brandt and H. Reiser. Dynamic Adaptation of Mobile Agents in Heterogeneous Environments. In *proceedings of the 5th International Conference on Mobile Agents*, volume 2240, pages 70–87, 2001.
- [38] R. Brandt and H. Reiser. Dynamic Adaptation of Mobile Agents in Heterogenous Environments. In *proceedings of the 5th International Conference on Mobile Agents*, pages 70–87, London, UK, Dec 2001.
- [39] F.M.T. Brazier, B.J. Overeinder, M. van Steen, and N.J.E. Wijngaards. Agent factory: generative migration of mobile agents in heterogeneous environments. In *proceedings of the 2002 ACM symposium on Applied computing*, pages 101–106, 2002.

-
- [40] L. Brown. Mobile Code Security. Technical Report CS07/96, 1996. <http://www.unsw.adfa.edu.au/~lpb/papers/mcode96.html>, last visited on 08.11.2007.
- [41] P.A. Buhler and J. Vidal. Towards Adaptive Workflow Enactment Using Multiagent Systems. *Information Technology and Management*, 6(1):61–87, January 2005.
- [42] G. Cabri, L. Leonardi, and F. Zambonelli. Coordination in Mobile Agent Applications. Technical Report DSI-97-24, Dipartimento di Scienze dell’ Ingegneria, Universit à di Modena, October 1997.
- [43] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile-Agent Coordination Models for Internet Applications. *IEEE Computer*, 33(2):82–89, February 2000.
- [44] K. Calvert, M. Doar, and E.W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.
- [45] J. Cao, S.A. Jarvis, S. Saini, and G.R. Nudd. GridFlow: Workflow Management for Grid Computing. In *proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 198–205, Tokyo, Japan, 2003.
- [46] J. Cao, D.P. Spooner, S.A. Jarvis, and G.R. Nudd. Grid load balancing using intelligent agents. *Future Generation Computer Systems*, 21(1):135–149, 2005.
- [47] J. Cao, J. Yang, W.T. Chan, and C. Xu. Exception handling in distributed workflow systems using mobile agents. In *proceedings of 2005 IEEE International Conference on e-Business Engineering (ICEBE’05)*, pages 48–55, Beijing, China, October 2005.
- [48] J. Cao, L. Zhang, X. Feng, and S.K. Das. Path Pruning in Mailbox-based Mobile Agent Communications. *Journal of Information Science and Engineering*, 20(3):405–424, 2004.
-

-
- [49] Q. Chen and M. Hsu. Inter-Enterprise Collaborative Business Process Management. In *proceedings of the 17th International Conference on Data Engineering*, pages 253–260, Germany, 2001.
 - [50] P.D. Coddington, L. Lu, D. Webb, and A.L. Wendelborn. Extensible job managers for grid computing. In *proceedings of the 26th Australasian computer science conference (ACSC'03)*, volume 16, pages 151–159, 2003.
 - [51] S. Coffey, L. Healy, and H. Neal. Applications of parallel processing to astrodynamics. *Celestial Mechanics and Dynamical Astronomy*, 66(1):61–70, November 2004.
 - [52] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. *Grid resource management: state of the art and future trends*, chapter Workflow management in GriPhyN, pages 99–116. Kluwer Academic Publishers, Norwell, MA, USA, 2004.
 - [53] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.H. Su, K. Vahi, and M. Livny. Pegasus: Mapping Scientific Workflows onto the Grid. In *proceedings of European Across Grids Conference*, pages 11–20, 2004.
 - [54] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping Abstract Complex Workflows onto Grid Environments. *Journal of Grid Computing*, 1(1):25–39, 2003.
 - [55] R.P. Dick, D.L. Rhodes, and W. Wolf. TGFF: Task Graphs for Free. In *proceedings of International Workshop Hardware/Software Codesign*, pages 97–101, Los Alamitos, Calif., 1998.
 - [56] P. Domingues, P. Marques, and L. Silva. DGSchedSim: A Trace-Driven Simulator to Evaluate Scheduling Algorithms for Desktop Grid Environments. In *proceed-*
-

- ings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 83–90, 2006.
- [57] R. Duan, R. Prodan, and T. Fahringer. Run-time Optimisation of Grid Workflow Applications. In *proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pages 33–40, Barcelona, September 2006.
- [58] E. Elliott. Design and implementation of Mojo: A mobile agent precompiler. Master’s thesis, University of California, San Diego, June 2000.
- [59] T. Fahringer, A. Jugravu, S. Pillana, R. Prodan, C.S. Junior, and H.L. Truong. ASKALON: A Tool Set for Cluster and Grid Computing. *Concurrency and Computation: Practice and Experience*, 17(2–4), 2005. <http://dps.uibk.ac.at/askalon/>, last visited on 03.11.2007.
- [60] R.S.S. Filho, J. Wainer, E.R.M. Madeira, and C. Ellis. A Fully Distributed Architecture for Large-scale Workflow Enactment. *International Journal of Cooperative Information Systems (IJCIS)*, 12(4):411–440, December 2003.
- [61] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-performance Distributed Computations. In *proceedings of the 6th IEEE Symposium on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.
- [62] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *proceedings of the 5th ACM conference on Computer and communications security*, pages 83–92, New York, NY, USA, 1998. ACM Press.
- [63] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

-
- [64] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *proceedings of the 14th Conference on Scientific and Statistical Database Management*, pages 37–46, July 2002.
- [65] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.
- [66] D. Gannon, R. Ananthakrishnan, S. Krishnan, M. Govindaraju, L. Ramakrishnan, and A. Slominski. *Grid Web Services and Application Factories*, chapter 9. Wiley, June 2003.
- [67] N.T. Giang and D.T. Tung. Agent platform evaluation and comparison. Technical report, Institute of informatics, Slovak academy of sciences, June 2002.
- [68] T. Glatard, J. Montagnat, and X. Pennec. Efficient services composition for grid-enabled data-intensive applications, 2006. <http://www.i3s.unice.fr/~johan/publis/HPDC06.pdf>, last visited on 05.12.2007.
- [69] L. Gong. Jxta: A Network Programming Environment. *IEEE Internet Computing*, 5(3):88–95, 2001.
- [70] P. Groth, S. Jiang, S. Miles, S. Munrow, V. Tan, S. Tsasakou, and L. Moreau. An Architecture for Provenance Systems. Technical report, Electronics and Computer Science, University of Southampton, October 2006. <http://eprints.ecs.soton.ac.uk/12023/>, page last visited on 01.11.2007.
- [71] P. Groth, M. Luck, and L. Moreau. A Protocol for Recording Provenance in Service-oriented Grids. In *proceedings of the 8th International Conference on Principle of Distributed Systems (PODIS'04)*, pages 124–139, Grenoble, France, December 2004.

-
- [72] R.J. Hayton, M.H. Bursell, D.I. Donaldson, W. Harwood, and A.J. Herbert. Mobile Java Objects. *Distributed System Engineering*, 6(1):51–60, March 1999.
- [73] G. Heffelfinger and A. Geist. Report on the Computational Infrastructure Workshop for the Genomes to Life Program, 2002. http://doegenomestolife.org/compbio/mtg_1_22_02/infrastructure.pdf, page last visited on 01.11.2007.
- [74] M. Hft, T. Magedanz, and J. Quantz. Grasshopper - the Agent Platform. *AgentLink News*, 3:7–9, 1999. <http://www.agentlink.org/newsletter/3/newsletter3.pdf>, last visited on 03.11.2007.
- [75] H.C. Hsiao, P.S. Huang, A. Banerjee, and C.T. King. Taking Advantage of the Overlay Geometrical Structures for Mobile Agent Communications. In *proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 46 – 55, Santa Fe, NM, U.S.A., 2004.
- [76] L. Ismail and D. Hagimont. A performance evaluation of the mobile agent. In *proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 306–313, 1999.
- [77] W. Jie, W. Cai, and S. J. Turner. POEMS: A Parallel Object-oriented Environment for Multi-computer Systems. *The Computer Journal*, 45(5):540–560, 2002.
- [78] M. Keidl, S. Seltzsam, and A. Kemper. Flexible and Reliable Web Service Execution. In *proceedings of the 1st Workshop on Entwicklung von Anwendungen auf der Basis der XML Web-Service Technologies*, pages 17–30, 2002.
- [79] U. Keller and H. Lausen. Functional Description of Web Services, 2004. <http://www.wsmo.org/TR/d28/d28.1/v0.1/>, last visited on 18.11.2007.
-

-
- [80] L. Khan and F. Luo. Ontology construction for information selection. In *proceedings of 14th IEEE International Conference on Tools with Artificial Intelligence*, pages 122–127, November 2002.
 - [81] M. Klusch and K. Sycara. Brokering and matchmaking for coordination of agent societies: a survey. *Coordination of Internet agents: models, technologies, and applications*, pages 197–224, 2001.
 - [82] G. Kola, T. Kosar, and M. Livny. A fully automated fault-tolerant system for distributed video processing and off-site replication. In *proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 122–126, Kinsale, Ireland, 2004.
 - [83] D.B. Lange and O. Mitsuru. *Programming and Deploying Java Mobile Agents Aglets*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
 - [84] G.V. Laszewski. Java CoG Kit Workflow Concepts for Scientific Experiments. Technical report, Argonne National Laboratory, Argonne, IL, USA, 2005.
 - [85] P. Liu. *Mobile Code Enabled Web and Grid Services*. PhD thesis, Computer Science Department, Binghamton University, August 2006.
 - [86] J. Lu and J. Turner. Constraint-driven Virtual Network Design on A Shared Substrate. Poster in 14th IEEE International Conference on Network Protocols, November 2006. <http://www.ieee-icnp.org/2006/posters/icnp06-posters-lu.pdf>, last visited on 04.11.2007.
 - [87] B. Ludaescher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience, Special Issue on Workflow in Grid Systems*, 18(10):1039 – 1065, August 2006.
-

-
- [88] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic Markup for Web Services, 2004. <http://www.w3.org/Submission/OWL-S/>, last visited on 18.11.2007.
 - [89] B.D. Martino and O.F. Rana. Grid performance and resource management using mobile agents. *Performance analysis and grid computing*, pages 251–263, 2004.
 - [90] A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse, and J. Darlington. ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time. In *proceedings of UK e-Science All Hands Meeting*, pages 627–634, Nottingham, UK, September 2003. IOP Publishing Ltd, Bristol, UK.
 - [91] J. Meng, S.Y.W. Su, H. Lam, A. Helal, J. Xian, X. Liu, and S. Yang. DynaFlow: a dynamic inter-organisational workflow management system. *International Journal of Business Process Integration and Management*, 1(2):101–115, June 2006.
 - [92] D. Milojevic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF: The OMG Mobile Agent System Interoperability Facility. *Personal Technologies*, 2(3):117–129, December 1999.
 - [93] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
 - [94] S. Rajbhandari and D.W. Walker. Support for Provenance in a Service-based Computing Grid. Proceedings of the UK e-Science All Hands Meeting 2004, September
-

2004. <http://www.allhands.org.uk/submissions/papers/194.pdf>, last visited on 03.11.2007.
- [95] S. Rajbhandari, I. Wootten, A.S. Ali, and O.F. Rana. Evaluating Provenance-based Trust for Scientific Workflows. In *proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'06)*, pages 365–372, Singapore, May 2006.
- [96] O.F. Rana and L. Moreau. Issues in Building Agent based Computational Grids. Proceedings of the 3rd Workshop of the UK Special Interest Group on Multi-Agent Systems (UKMAS'00), December 2000. <http://www.ecs.soton.ac.uk/~lavm/papers/ukmas.ps.gz>, last visited on 08.11.2007.
- [97] A.D. Rubin and D.E.G. Jr. Mobile Code Security. *IEEE Internet Computing*, 02(6):30–34, November/December 1998.
- [98] O. Shehory. A Scalable Agent Location Mechanism. In *proceedings of Agent Theories, Architectures, and Languages*, pages 162–172, 1999.
- [99] Y.L. Simmhan, B. Plale, and D. Gannon. A Survey of Data Provenance in e-Science. *SIGMOD Record*, 34(3):31–36, September 2005.
- [100] Y.L. Simmhan, B. Plale, and D. Gannon. A Framework for Collecting Provenance in Data-Centric Scientific Workflows. In *proceedings of the IEEE International Conference on Web Services 2006 (ICWS'06)*, pages 427 – 436, Chicago, USA, September 2006.
- [101] M.P. Singh and M.A. Vouk. Scientific workflows: scientific computing meets transactional workflows. In *proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, pages 28–34, Univ. Georgia, Athens, GA, USA, 1996.

-
- [102] S. Srinivas and N.K. Jha. Safety and Reliability Driven Task Allocation in Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(3):238–251, 1999.
 - [103] J.W. Stamos and D.K. Gifford. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–722, July 1990.
 - [104] M.V. Steen, F. Hauck, P. Homburg, and A.S. Tanenbaum. Locating Objects in Wide-area Systems. *IEEE Communications Magazine*, pages 104–109, 1998.
 - [105] N. Suri and H. Hexmoor. Editorial: Mobility in Distributed Systems. *Scalable Computing: Practice and Experience Scientific International Journal for Parallel and Distributed Computing, Special Issue: Software Agent Mobility*, 7(4):i–ii, December 2006. <http://www.scpe.org/vols/vol07/no4/vol07no4editorial.html>, last visited on 08.11.2007.
 - [106] I. Taylor, I. Wang, M. Shields, and S. Majithia. Distributed computing with Triana on the Grid. *Concurrency and Computation: Practice and Experience Journal*, 17(9):1197–1214, 2005.
 - [107] O. Tomarchio and L. Vita. On the use of mobile code technology for monitoring Grid system. In *proceedings of 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 450–455, May 2001.
 - [108] P. Townend, P. Groth, and J. Xu. A Provenance-aware Weighted Fault Tolerance Scheme for Service-based Applications. In *proceedings of the 8th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC’05)*, pages 258–266, Seattle, USA, May 2005.
 - [109] E. Türsem, M.H. Günes, M. Yildiz, and S. Kuru. Performance Analysis of Mobile Agents Using Simulation. *Proceedings of the Advanced Engineering Design Con-*
-

- ference (AED'03), 2003. <http://www.utd.edu/~mhg042000/papers/AED03.pdf>, last visited on 05.12.2007.
- [110] A. Tveit. jfipa:an Architecture for Agent-based Grid Computing. In *Proceedings of AISB'02 Convention, Symposium on AI and Grid Computing*, London, United Kingdom, April 2002. <http://www.idi.ntnu.no/~amundt/publications/2002/jfipaACC.pdf>, last visited on 08.11.2007.
- [111] G.J. van 't Noordende, F.M.T. Brazier, and A.S. Tanenbaum. A Security Framework for a Mobile Agent System. In *proceedings of the 2nd International Workshop on Security in Mobile Multiagent Systems (SEMAS 2002), associated with AAMAS-2002*, pages 43–50, Bologna, Italy, July 2002.
- [112] S. Vazhkudai, S. Tuecke, and I. Foster. Replica Selection in the Globus Data Grid. In *proceedings of the 1st IEEE/ACM International Conference on Cluster Computing and the Grid (CCGRID'01)*, pages 106–113. IEEE Computer Society Press, May 2001.
- [113] J. Vitek and C.F. Tschudin, editors. *Mobile Object Systems - Towards the Programmable Internet, Second International Workshop, MOS'96, Linz, Austria, July 8-9, 1996, Selected Presentations and Invited Papers*, volume 1222 of *Lecture Notes in Computer Science*. Springer, 1997.
- [114] D.W. Walker, O.F. Rana, Y. Huang, and L. Huang. Workflow Optimisation for e-Science Applications. In *proceedings of the 27th International Conference on Information Technology Interfaces (ITI)*, pages 86–91, Dubrovnik, Croatia, June 2005.

-
- [115] T. White. *A Mobile Code Framework and Mobile Agent Facility Specifications*.
<http://www.sce.carleton.ca/netmanage/mctoolkit/lecture/index.html>, last visited on 03.11.2007.
 - [116] J. Winick and S. Jamin. Inet-3.0: Internet topology generator. Technical Report CSE-TR-456-02, Computer Science Department, University of Michigan, Ann Arbor, MI, 2002.
 - [117] S.C. Wong, S. Miles, W. Fang, P. Groth, and L. Moreau. Provenance-based Validation of E-Science Experiments. In *proceedings of the 4th International Semantic Web Conference (ISWC'05, LNCS Vol.3729)*, pages 801–815, Galway, Ireland, November 2005.
 - [118] C.Z. Xu and B. Wims. A mobile agent based push methodology for global parallel computing. *Concurrency - Practice and Experience*, 12(8):705–726, 2000.
 - [119] J. Yan. *A Framework and Coordination Technologies for Peer-to-peer based Decentralised Workflow Systems*. PhD thesis, Swinburne University of Technology, 2004.
 - [120] K. Yelick. Applications of Parallel Computers Load Balancing, 2004. <http://www.cs.berkeley.edu/~yelick/cs267-sp04/lectures/21/lect21-load.pdf>, page last visited on 02.11.2007.
 - [121] K. Yocum, K. Walsh, A. Vahdat, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGCOMM Computer Communication Review*, 32(3):28–28, July 2002.
 - [122] J. Yu and R. Buyya. A Novel Architecture for Realizing Grid Workflow using Tuple Spaces. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID'04)*, volume 0, pages 119–128, 2004.
-

- [123] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3(3-4):171–200, 2005.
- [124] E.W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *proceedings of IEEE Infocom*, volume 2, pages 594–602, March 1996.
- [125] T. Zhang, St. Covaci, and T. Magedanz. OMG and FIPA Standardisation for Agent Technology: competition or convergence? Proceedings of the 4th International Symposium on Interworking '98, February 1998. <http://cordis.europa.eu/infowin/acts/analysys/products/thematic/agents/ch2/ch2.htm>, last visited on 03.11.2007.
- [126] J. Zhao, C. Goble, M. Greenwood, C. Wroe, and R. Stevens. Annotating, linking and browsing provenance logs for e-Science. Proceedings of Workshop on Semantic Web Technologies for Searching and Retrieving Scientific Data (in conjunction with ISWC'03), CEUR Workshop Proceedings, Vol.83, October 2003. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-83/prov_2.pdf, last visited on 02.11.2007.