Exploiting Dynamic Deployment in a Distributed Query Processor for the Grid

Thesis by

Arijit Mukherjee

School of Computing Science

In Partial Fulfilment of the Requirements
for the Degree of
Doctor of Philosophy

NEWCASTLE UNIVERSITY LIBRARY
206 53499 9
Thesis L8736

Newcastle University
Newcastle upon Tyne, UK

2008
(Submitted 27th February 2008)

To my family

without whose constant encouragement

I'd have remained

just another "software guy"...

Acknowledgements

I remember the day I first came to Newcastle to attend the interview for an RA position in the ^{my}*Grid* project. Newcastle was in stark contrast with the town of Gaithersburg, Maryland, where I used to live and work for Verizon Communications Inc. at Silver Spring. It was the middle of January - cold and cloudy and dark, and the wind from the North Sea almost blew me away. It was hard to make a decision about leaving the job in the US to join academia. Today, when I look back, I know, I was right. I have now spent more than five years in Newcastle, the longest period at one place since I left home in 1997 to join the software industry. And I can say that Newcastle has been nothing less than a home to me. It is like my hometown Calcutta - a city, which slowly grows around a person - Newcastle has grown all around me. And I owe it to Professor Paul Watson, Professor Pete Lee and Dr. Anil Wipat - who extended the first welcome to me. I owe my gratitude to the City of Newcastle, the university, all members of staff in the School of Computing Science, my friends within and outside the university, for the warmest five years I have stayed here, despite the chilling weather.

Professor Paul Watson was my supervisor and I could not have come this far without his constant guidance and support. Apart from always managing to get some funds to keep me employed as an RA without which I couldn't have continued with my PhD, he has provided me with valuable guidance and insights throughout the course of research. On numerous occasions when I was struggling to find the right approach, regardless of his busy schedule as the Director of the North East Regional e-Science Centre, Paul has been tireless in his attempts to make me focus on the problem from the correct angle. No word is sufficient to express my gratitude to Paul.

I would like to thank Professor Pete Lee and Dr. Aad van Moorsel, who were the two other members of my thesis committee for their valuable suggestions during and after the thesis committee meetings which acted as inputs to my work. Dr. Jim Smith is another person who has been a close friend

in these five years and have always helped me when I faced any problem, be it regarding any architectural aspect of my work, or any silly question about LaTeX. I am indebted to Jim for his help and support whenever I asked for. Dr. Savas Parastatidis, who is now at Microsoft Research, was a source of inspiration during the years I was able to work with him. All the long discussions I had with him regarding the architecture of Web Services have contributed a lot towards my knowledge and the research. I must not miss mentioning about the support I received from the Computing Officers, especially Jim Wight and Gerry Tomlinson, who have always listened to my requests about new softwares on the cluster and helped me in configuring my experimental setup, which sometimes required Jim to bypass security rules of the Computing Cluster for the external computers I used during my experiments.

A large section of the work presented here was the result of collaborative research between Newcastle and Manchester Universities. I wish to thank my colleagues from Manchester, especially Professor Norman W. Paton, Dr. Alvaro A. A. Fernandes, Dr. Tasos Gounaris, Steven Lynden and Dr. M. Nedim Alpdemir, who unfortunately left for his country a couple of years ago, for all the active collaboration and support I received from all of them. Another part of the research, the development of the dynamic service oriented framework, was based on collaborative research as well, and I wish to thank Dr. Chris Fowler, Charles Kubicek, John Colquhoun for their valuable contributions.

I can not forget the amount of support I received from my family during this entire journey. My parents, Mrs. Binata Mukherjee and Mr. Prabhat Mukherjee have inspired me to dream since I was a child. And I am extremely indebted to them, and I hope that these three letters, if I am able to achieve them, will fulfil a part of their dreams. I can never express enough gratitude for the support I received from my sister, Dr. Nandini Mukhopadhyay, who has constantly encouraged me, at times pushed me - when I used to get frustrated. One person needs a special mention here, and that is my wife, Sumana, who never fell short in supporting me in every step, and was not shy in sacrificing her perfectly good job in the US, when I decided to join academia in the UK to pursue my dreams. Our little boy, Rik, has been my source of joy at home and our newborn daughter, Riti, has been another source of inspiration during the last few months of my work.

Finally, I would like to thank EPSRC, who have funded the major projects I have worked in, and my colleagues at OGSA-DAI for their valuable support during the course of research.

Abstract

The adoption of the "Web Services" model for building a Grid framework created a considerable shift from the original concept of Grid which was based on "distributed job scheduling". The requirement for the access and integration of heterogeneous data resources over the Grid, and the advances in service-oriented data access standards led to the development of a service-oriented distributed query processor, which forms the basis of this thesis.

The adoption of service-orientation raised the need for a framework which would allow demand-driven deployment of Web Services on available resources. Research into such concepts led to the development of DynaSOAr, a framework which proposed an alternative approach to distributed job scheduling by focussing entirely on the concept of services, rather than the more traditional jobs. DynaSOAr allows services to be deployed on demand to meet changing performance requirements and exploits the advances made in virtualization technologies to support the deployment of services and databases.

The thesis describes a system designed to exploit dynamic deployment features within the context of distributed query processing on the Grid, and argues that such features benefit query evaluation by creating a loose coupling between the services and the available resources. The extended distributed query processing system is able to collocate various entities, such as the evaluation and analysis services with the data, to reduce data traffic over the network, and is also able to reconfigure itself to improve performance by dynamically deploying database snapshots. The thesis evaluates the dynamic distributed query processing framework through several experiments which explore its behaviour in a variety of scenarios.

Contents

Acknowledgements iii

Abstract v

List of Figures x

Table of Listings xi

1 Introduction 1

1.1 Contribution 6

1.2 Structure of the Thesis 7

2 Background and Related Work 10

2.1 The Grid Problem 10

2.2 Service Orientation and The Grid 12

2.3 Databases and The Grid 17

2.3.1 Data Access and Integration Services 19

2.3.1.1 OGSA-DAI 20

2.3.2 Distributed Query Processing, The Grid and SOA 22

2.3.3 Requirements for Dynamism 26

2.4 Towards a Dynamic Service-Oriented Architecture 27

2.4.1 Use of Virtualization Technologies 34

2.5 Exploiting Dynamism in Distributed Query Processing 37

2.6 Discussion 38

3 Service Oriented Distributed Query Processing 40

3.1 OGSA-DQP as a Service Orchestration Mechanism 41

3.2 The Architecture . . . 41
3.3 Setting Up a Distributed Query Service . . . 44
3.4 Distributed Query Plan Generation . . . 47
3.5 The Query Evaluation Service . . . 50
3.5.1 The Overview . . . 50
3.5.2 Evaluation Model . . . 53
3.5.3 Data and Control Tuples . . . 54
3.5.4 Encapsulation of parallelism by Exchange operators . . . 54
3.5.4.1 Horizontal and Vertical Parallelism . . . 55
3.5.4.2 Exchange operators in OGSA-DQP . . . 56
3.5.5 Encapsulating Service State . . . 60
3.6 Discussion . . . 63

4 Dynamic Service Deployment 65

4.1 Distributed Job Scheduling . . . 65
4.2 The Evolution of DynaSOAr . . . 66
4.2.1 Jobs and Service Orientation . . . 66
4.2.2 The Active Information Repository . . . 68
4.2.3 The Consumer View of a Service . . . 68
4.2.4 Formation of Dynamic Virtual Organisations . . . 70
4.2.5 Principles of Dynamic Deployment . . . 71
4.2.6 Requirements for DynaSOAr . . . 72
4.2.7 Design Considerations . . . 74
4.2.7.1 Using SOAP Message Headers . . . 74
4.2.7.2 Using Message Orientation . . . 75
4.3 The design of DynaSOAr . . . 76
4.3.1 Service Provider . . . 76
4.3.1.1 Supporting conventional tools . . . 78
4.3.2 Host Provider . . . 79
4.3.3 Service Repository . . . 82
4.3.4 Registry Service . . . 83
4.3.5 The Software Hypermarket . . . 84
4.4 Using Virtualization . . . 86
4.4.1 Virtualization Technology - An Overview . . . 87
4.4.2 Case for Virtualization in DynaSOAr . . . 89
4.4.3 Using Virtualization in DynaSOAr . . . 90

4.5 Discussion 93

5 Exploiting Dynamic Service Provisioning in DQP 96

5.1 Usage Scenarios 97

5.1.1 Collocating the Query Evaluation Engine with the data 97

5.1.2 Collocating the Analysis Service with the data 97

5.1.3 Increased degree of parallelism 98

5.1.4 Availability of the third-party maintained Analysis Services 98

5.1.5 Data caching by dynamic deployment of databases 100

5.1.6 Services requiring special environments 100

5.2 Towards a Dynamic Distributed Query Processor 101

5.2.1 Overview 101

5.2.2 Architecture 104

5.2.3 Setting up the Distributed Query Processor 105

5.2.4 Proactive deployment of the Analysis Service 106

5.2.5 Distributed Query Plan Generation 107

5.2.6 Using Network-aware Cost Models 111

5.2.7 Virtualization in DQP 112

5.2.7.1 The Model 112

5.2.7.2 The Feedback Methodology 113

5.2.7.3 Reconfiguration of the DQP data resource 117

5.3 Discussion 119

6 Evaluation of the Dynamic DQP Framework 121

6.1 Implementation 121

6.1.1 OGSA-DQP 122

6.1.2 DynaSOAr 122

6.2 Evaluation 123

6.2.1 Evaluation Platform 123

6.2.2 Collocating the Data and Analysis Code 126

6.2.3 Parallelization of OperationCall using Proactive Deployment 129

6.2.4 Collocating the Evaluation Service with Data 131

6.2.5 Experiments on Virtualization 134

6.2.6 Deploying a Database Snapshot Locally 135

6.2.6.1 A simple select query 138

6.2.6.2 A select-project-join query 140

6.2.6.3 A select-project-join-operation_call query 140

6.2.7 Evaluating Availability Issues 145
6.2.8 Services Requiring Special Environments 145
6.3 Discussion 146

7 Conclusions **147**
7.1 Summary and Discussion 147
7.1.1 Service Oriented Distributed Query Processing - Chapter 3 147
7.1.2 Dynamic Service Deployment - Chapter 4 149
7.1.3 Exploiting Dynamic Service Provisioning in DQP - Chapter 5 151
7.1.4 Summary of Contributions 154
7.2 Further Work 155
7.2.1 Efficient Data Movement Between DQP Services 156
7.2.2 Support for Non-relational Data Formats 156
7.2.3 Effective Brokering in DynaSOAr 156
7.2.4 Robust and Efficient Transport for DynaSOAr Deployment 157
7.2.5 Remodelling the Query Compiler 157

Bibliography **159**

List of Figures

2.1 Component roles in Service interaction . . . 15
2.2 The potential data explosion . . . 19
2.3 Basic Architecture of OGSA-DAI . . . 21
2.4 Basic Architecture of OGSA-DQP . . . 25
2.5 Architecture of the dynamic version of OGSA-DQP . . . 38

3.1 Basic DQP Architecture . . . 42
3.2 DQP Initialisation . . . 46
3.3 DQP Interaction . . . 47
3.4 DQP query plan . . . 49
3.5 Query Execution on Component Services . . . 52
3.6 Communication between distributed partitions . . . 57
3.7 Architecture of the Query Evaluation Service . . . 62

4.1 The Active Information Repository Architecture . . . 69
4.2 Invocation of a Web Service . . . 69
4.3 Formation of a Virtual Organization between the Service Provider and the Host Provider 71
4.4 Routing of requests in DynaSOAr . . . 73
4.5 Consumer interaction with DynaSOAr Service Provider . . . 77
4.6 Request types supported by the Service Provider . . . 78
4.7 Interaction between DynaSOAr Service Provider and Host Provider . . . 81
4.8 Request types supported by the Host Provider . . . 81
4.9 Request types supported by the Service Repository . . . 82
4.10 The Software Hypermarket . . . 85
4.11 Generic Architecture of DynaSOAr . . . 86

4.12 Before and after virtualization . . . 87
4.13 XML Schema for describing a Virtual Machine . . . 92

5.1 Various configurations of a dynamic DQP framework . . . 99
5.2 Overview of the static DQP system . . . 102
5.3 Overview of the Dynamic DQP system . . . 103
5.4 Basic DQP Architecture . . . 105
5.5 Query Execution on Component Services in a Dynamic OGSA-DQP framework . . . 111
5.6 XML Schema for collecting performance data . . . 114
5.7 Reconfiguration in dynamic OGSA-DQP . . . 118

6.1 The complete experimental setup . . . 125
6.2 Two configurations used for experimenting with service collocation . . . 127
6.3 Comparing query execution using a local and a remote service . . . 128
6.4 Experimental setup for the proactive analysis service deployment . . . 130
6.5 Comparing parallelised and non-parallelised operation calls . . . 131
6.6 Experimental setup for the collocation of evaluation with data . . . 132
6.7 Experimental results for the collocation of evaluation with data . . . 133
6.8 Comparing the performance of a VM with other setups for a distributed query . . . 134
6.9 Experimental setup for reconfiguration of DQP data resource . . . 136
6.10 Experimental setup for reconfiguration of DQP data resource (contd.) . . . 137
6.11 Comparing data transport cost and query execution cost using a remote setup and a setup which allows switching of a data node . . . 139
6.12 Comparing data transport cost for two data nodes for remote and switching setup . . 141
6.13 Comparing execution cost of a select-project-join query for a remote and switching setup 142
6.14 Comparing data transport cost for two data nodes for a remote and switching setup . 143
6.15 Comparing execution cost of a select-project-join-operation call query for a remote and switching setup . . . 144

Table of Listings

2.1 An example perform document submitted to the OGSA-DAI framework 23
3.1 An Example Query 43
3.2 Configuration Document 46
3.3 XML Partition Document 51
3.4 Tuple Structures in XML 55
3.5 Exchange Operator - Open() method 59
3.6 Exchange Operator - Next() method 59
3.7 Exchange Operator - Close() method 60
3.8 Description of the Query Evaluation Service interface 62
4.1 An example SOAP message requesting the EntropyAnalysis service 75
4.2 An example SOAP message with a modified header element at the Service Provider 77
4.3 An example SOAP message with a modified header element at the Host Provider . . 80
4.4 Description of entities registered within the DynaSOAr registry 84
4.5 Entries in the registry describing services packaged in a virtual machine 91
5.1 Configuration Document for Dynamic OGSA-DQP 106
5.2 Assigning degree of parallelism to operators 108
5.3 Assigning evaluators to partitions 110
5.4 Example of performance data from two remote nodes 116
6.1 Structure of the databases used for the experiments 124
6.2 Configuration Document for a Virtual Machine 126
6.3 A select-opcall query 127
6.4 A select query 131
6.5 A select-project-join query 140
6.6 A select-project-join-opcall query 142

Chapter 1

Introduction

"e-Science aims to provide support for large-scale science by enabling distributed global collaborations through the formation of virtual co-laboratories that allow scientists to work together irrespective of location and permit universal access to scientific resources."[1]

e-Science, as defined above by the European Bio-Informatics Institute (EBI) is considered as one of the primary approaches to inter-disciplinary research where scientists from various streams collaborate in order to achieve collective goals. It was felt from the inception of the e-Science Programme while the use of the evolving Grid frameworks would benefit collective research, science would suffer if the scientists such as bio-informaticians, biologists, astronomers were forced to adopt new languages, platforms and tools to perform the scientific work in such collaborative environments. It would be unreasonable to assume that these scientists were experts in Grid computing or resource sharing in collaborative environments. To make e-Science a reality in scientific researches, the e-Science framework should provide scientists with a higher level abstraction layer with the freedom of using their own standard languages, toolkits and databases leaving the complexity of collaborative resource sharing to the underlying middleware.

Fortunately, this is the approach that was taken by the e-Science community in the UK and has been supported around the globe. Adopting Web Service standards and technologies was a step towards achieving the goal of creating the middleware which would serve the scientific community.

The adoption of *service oriented* technologies was a significant step for e-Science. There was a considerable shift from the traditional Grid computing infrastructures based on distributed job-scheduling systems, such as Condor [2, 3] and Globus [4] where any computation is packaged by

the consumer as a *job* consisting of the *computation code* and in most cases the *input data* on which the code should operate, and submitted to the framework for execution. The scheduling system at the core of the framework decides on which resource the job should be executed based on algorithms such as *matchmaking*, where the requirement of the computational job is matched against the available hosts to select the best suited host for execution of the job. Once execution is complete, the system discards the job. Compared to this, *service orientation* provided an alternative paradigm for distributed computing where a clear separation is created between the interface and the internals of the underlying system and a *loose coupling* between autonomous systems by exchanging SOAP [5] messages is advocated. The emergence of this alternative approach was widely publicised in the computing domain, and the *Web Service* implementation resulted in widespread adoption of the technology as a means to create the Grid platform. A complex system or application may now be modelled as a collection of finer autonomous systems, which are independent of each other and may be distributed across the network, but are highly inter-operable using messages, compared favourably to the earlier tightly coupled object-oriented monolithic systems.

Apart from the new architectural aspect, the service-based approach provided viable alternatives in certain aspects related to the deployment scenarios. A *job*, in job-based systems, is deployed and once the execution is complete, the job is discarded. In scientific domains where e-Science wanted to contribute, frequent execution of similar analysis or queries is a common procedure, which, on a job-based system, would mean submission of the job with the input data if any for each experiment instance. Services, on the other hand, remain deployed unless explicitly removed, and can serve multiple requests throughout their lifetime.

The concepts and ideas presented in this thesis gained momentum in one of the first e-Science pilot projects, *myGrid* [6], which aimed to extend the Grid framework in order to provide a *virtual laboratory* for the biologists and bio-informaticians to perform *in-silico* experiments. It was observed during the initial stages of *myGrid* and from contemporary developments in other streams, such as *particle physics* and *astronomy*, that a strong focus was growing on how to the use and analyse the astronomical volume of raw data that would be generated by scientific experiments in the coming years. Related to the volume of experimental data is the issue about the cost of moving this data for analysis. Such issues regarding locality of the data and the analysis code were raised in the Active Information Repository (AIR) proposal by Watson and Lee [7] and it became more and more relevant as the size of the experimental dataset in various scientific domains started growing with a remarkable speed. The AIR architecture proposed the collocation of analysis code with the data by using mobile agent technologies [8] in an attempt to reduce the overhead of transferring large amounts to data over the network for analysis.

A common experiment performed by biologists in *^{my}Grid* was accessing protein sequence data from existing databases and performing analysis on them using well known sequence analysis services, such as Blast [9]. Scientists in ^{my}Grid adopted the use of workflows such as Taverna [10] to replace the traditional way of doing this analysis manually. At the same time, an approach based on *distributed query processing* [11] was conceived as complimentary to such workflow-based systems, as it was likely that the biologists may have a requirement of accessing data from different databases that are distributed across the globe and perform the analysis on them. It was also noted that this alternative *distributed query processing* system could use parallel database techniques from earlier research [12, 13] for better use of available resources by evaluating relevant sections of a query in parallel. Almost at the same time, there were emerging standards for accessing data from heterogenous databases in a homogeneous way, and the OGSA-DAI [14] project resulted in a set of services which allowed consumers to access data from heterogeneous sources irrespective of the platform, the underlying DBMS and the format in which data is stored. It was decided that the new *distributed query processing* system should reap the benefits of such services.

Thus the challenge was to develop a distributed query processing system based on service-orientation, which would use the emerging data access services, such as OGSA-DAI to access data from heterogeneous sources, and evaluate queries over them, with an option for including analysis services within the query itself. The conceived system would be exposed as a service to consumers and encapsulate the complexities of query compilation and execution within autonomous and inter-operating services. With such a system in mind, the first objective presented in the thesis is:

> *to create a Distributed Query Processing framework which allows homogeneous access to heterogeneous data resources by using existing infrastructures (such as OGSA-DAI) and evaluate distributed queries by parallel evaluation of query fragments using techniques from parallel databases on a Web Service based query processing engine created at run-time.*

A *service oriented distributed query processing system* was thus developed, which came to be known as OGSA-DQP [11]. It allowed scientists such as bio-informaticians to submit queries over a set of distributed bio-informatics databases to find useful data, such as protein sequences and invoke analysis services on them. OGSA-DQP compiled and optimised the query and partitioned it into several fragments or partitions which were then evaluated in parallel on a distributed set of computational nodes. A requirement for the dynamic deployment of services was realised during the development of OGSA-DQP. The system was tightly coupled to the resources that were available in the sense that the component services of DQP were required on all participating nodes. It was bound to existing

instances of analysis service, and there was no notion of collocation of the analysis code with the data as proposed in the Active Information Repository architecture [7]. Further, third-party maintained analysis services were likely to have availability issues due to sudden failure of hosts providing the service which would result in unsuccessful experiments. At the same time, *code mobility* was an integral part of the traditional job-based Grid frameworks, where jobs are queued up and scheduled to run once resources become available. Systems such as Condor performed *matchmaking* to select the best suited node for execution of a particular job. Although the architecture of OGSA-DQP allowed the use of monitoring services to discover lightly loaded resources, the tight coupling of the DQP services with the resources limited the scope for exploiting the dynamism within Grid systems, and did not consider the volatile nature of the resources. A strategy based on *demand-driven deployment* would provide better performance by selecting lightly loaded resources for computation, better reliability by using multiple copies of the service and late allocation of services to nodes allowing loose coupling. The requirement of some form of demand-driven deployment within the context of OGSA-DQP led to the research into *dynamic deployment* of services.

Investigation started regarding the possibilities of "dynamic service deployment" in a Grid environment where services can be deployed on remote nodes on demand rather than having them pre-configured on the system. Within the OGSA-DQP context, this "on-demand" service deployment allowed the movement of query execution and(or) analysis code towards the data thereby reducing the amount of data traffic over the network. While exploring the possibilities of deploying services on demand, the interest into virtualization technologies grew. Proposals for using virtualization technologies such as VMWare [15] and Xen [16] for Grid frameworks were discussed by Keahey et. al. [17]. Live migration of virtual machines was shown to be possible by Ruth et. al. [18]. It was envisaged during the course of this research that the use of virtual machines would allow on-the-fly demand-driven deployment of not only query execution service code or data analysis code, but also on-the-fly replication and deployment of databases. Consider a situation, where a certain database belonging to a remote organisation on another network, is accessed through several queries over a considerably long period of time. It may be beneficial to deploy snapshots of the database on hosts within the local organisation - a step which will reduce the amount of data being transferred over the network.

Dynamic Service Oriented Architecture, or DynaSOAr [19], was developed as a Web Service based framework that allowed dynamic demand-driven deployment of Web Services. It provided a logical separation between service provisioning and resource provisioning by creating a distinct set of entities with a clear division of responsibilities. DynaSOAr maintained the loose-coupling and execution transparency of Web Service platforms by relying on the WS-I [20] recommended model based on the Web Services Basic Profile [21] and advocated a message-oriented model of interaction. Support for

deploying virtual machines as means of deploying special environments or databases was provided in DynaSOAr. Investigations into possible dynamic deployment options provided the second objective of this thesis -

to explore the current status of dynamic deployment and code mobility in a service-oriented setting, and to create a framework that will allow on-demand deployment of services and database snapshots packaged within virtual machine images on available resources.

With the framework for dynamic deployment in place, the next task was to exploit the dynamic deployment concepts within the OGSA-DQP system. Several usage scenarios were identified featuring different aspects in DQP, each exploiting the features of on-demand deployment in order to gain some benefit. Based on the approach proposed in the Active Information Repository architecture and contemporary ideas about moving the computation closer to the data, a potential use case for dynamic deployment within DQP was the collocation of the query evaluation engine and the analysis service with the data. A greater degree of parallelism within the *operation_call* operator responsible for invoking analysis services within a query was another possible scenario for dynamic deployment which could provide potential benefit to DQP. Moving the computation closer to the data is considered as the common approach in most data-oriented Grid systems [22]. In DQP, an alternative approach is explored which allows the deployment of a snapshot of the database on a local node to eliminate the cost involved in data movement over the network. In effect, this approach is similar to database replication techniques, but is more dynamic and demand-driven in nature. Traditional data replication techniques depend on a lengthy offline administrative process for creating the initial copy which is later synchronised with the master copy. In DQP, the approach taken is dependent on the performance of the queries that are being executed, and need not be an administrative process. Snapshots of the databases are packaged within virtual machines with all necessary services. These virtual machines are stored in the software repository and deployed when required resulting in the database snapshot being available within the local network. It is assumed that the queries do not depend on the latest data or that a background process is used to keep the snapshot synchronised with the original copy. Several e-Science experiments rely on databases such as the SkyServer [23] where regular updates are not essential for the experiments. Exploitation of possible dynamic deployment features within the context of OGSA-DQP created the third objective presented in the thesis -

to investigate how to add on-demand deployment features within OGSA-DQP for the deployment of databases, analysis services and query processing operations.

1.1 Contribution

The contribution of this thesis lies in the overall design and evaluation of a service-oriented distributed query processing system that exploits dynamic deployment. The work builds on two key frameworks described in the thesis - OGSA-DQP and the dynamic service provisioning architecture, DynaSOAr. These were the results of collaborative research, and hence the thesis does not make any attempt to claim the sole credit for these two frameworks. Contribution to key aspects of both these frameworks are considered as major contributions in this thesis, along with the design of the DQP system with dynamic deployment capabilities that build on them.

This thesis investigates various architectural aspects of a service-oriented distributed query processing framework, its design, construction, and analysis. This thesis argues that distributed query processing (DQP) can provide effective declarative support for *service orchestration*, and builds a framework that:

- supports queries over a standardised "Grid Data Service" (GDS) and other analysis services made available over the grid thereby combining data access with data analysis;
- allows the framework to use the facility provided by Open Grid Services Architecture (OGSA) to dynamically obtain resources required for efficient evaluation of a distributed query;
- adapts techniques from parallel databases to provide implicit parallelism for complex data-intensive requests;
- uses emerging standards of GDSs to provide consistent access to database metadata and to interact with the databases on the Grid.

The resulting OGSA-DQP enables distributed query processing within a service-oriented setting by exposing itself as an extension of the popular OGSA-DAI service and creating an orchestration of multiple instances of the evaluation service. This evaluation service encapsulates the complexities of the physical algebra operators implemented following the iterator model of query evaluation and the routing of data tuples between the services enabling pipelined and partitioned parallelism [24, 25, 26]. The evaluation service follows the philosophy of using the Web Services Basic Profile [21] standards and toolkits and the proposals made by WS-GAF [27] regarding interaction between stateful services. Such implementation of the evaluation service also contributes towards the adoption of dynamic deployment features.

The thesis also investigates into the possibilities of "dynamic service provisioning" in a Grid envi-

ronment where services can be deployed on available nodes on demand rather than having them pre-configured on the system. This research into dynamic service deployment was again collaborative in nature and was carried out by several researchers, each pursuing a different aspect within the broader concept. The thesis contributes towards the overall design and architecture of the dynamic service oriented architecture, which came to be known as DynaSOAr, and more specifically to certain aspects within DynaSOAr, such as the use of software registries, the message oriented model of communication between the entities, the support for standard toolkits for service invocation and particularly, the use of virtualization technologies to complement the deployment of Web Services with the demand driven deployment of special environments and databases. The thesis also investigates the costs associated with the dynamic deployment of services and virtual machines. Dynamic service deployment will have an associated "deployment cost" signifying the cost incurred while initialising the service within the container and a "routing cost" signifying the cost of downloading the service code to the target host. Deployment of virtual machines incur much higher costs, although, the thesis shows that the costs will eventually be outweighed by the performance benefits.

In summary, the thesis investigates the design of a service-oriented distributed query processing system which supports dynamic service deployment. It describes several usage scenarios where on-demand deployment will benefit distributed query processing and evaluates the extended system to establish the claims. It analyses the results of various experiments performed to evaluate the system and suggests that the usage scenarios and proposals made during the investigation should result into the remodelling of the query compiler/optimizer and the development of new cost models which will take into account dynamic deployment.

1.2 Structure of the Thesis

In Chapter 2 a review of contemporary work on aspects related to the thesis is presented with a discussion. The review takes into account the emergence of various technologies in order to tackle the "Grid" problem including the proposals such as OGSI [28], WS-I Basic Profile [21], WS-GAF [27] and WS-I+ [29] which are related to the work done during the course of research. There are discussions about the requirement for data access and integration standards and the emergence of OGSA-DAI before focusing on the requirement of *distributed query processing* and related works. The need for *dynamic service deployment* is explained and contemporary work in this area is reviewed. Contemporary work on the use of virtualization technologies are also reviewed within the context of dynamic service deployment.

Chapter 3 describes in depth the design and architecture of a *service oriented distributed query processing system* which forms the basis of the entire thesis. The different architectural issues are discussed along with the constituent components. The various phases in executing distributed queries, such as the initialisation, query compilation etc. are discussed in detail. Particular stress is given to the description of the query execution process and the encapsulation of the iterator model in the evaluation services, specially on the functioning of the *exchange* operator which encapsulates the entire communication and distribution mechanism. A discussion is provided regarding the encapsulation of service state within the evaluation service using standardised Web Services technologies.

The requirements for a dynamic service deployment framework and the concepts behind the evolution of DynaSOAr are described in Chapter 4 which forms another pillar for this thesis. This chapter analyses the existing methods of job-based distributed computing such as Condor and Globus and proposes a service-based approach that allows on-demand deployment of services on available resources. The architecture makes a clear separation between service provisioning and resource provisioning, and makes use of standard service registries to enable discovery. Each of the components which build the DynaSOAr framework are described in the chapter leading to the vision of a *software hypermarket*. The chapter also introduces the concept of virtualization and discusses on how such technologies can be used in the context of dynamic deployment within the DynaSOAr framework.

Chapter 5 discusses about the requirements for dynamic service provisioning within OGSA-DQP and puts forward a set of usage scenarios where on-demand deployment of analysis and evaluation services and database snapshots may benefit query processing by enabling the collocation of various entities. The chapter introduces the changes made within the DQP architecture to allow it to take the advantage of DynaSOAr framework by using several DynaSOAr components, such as the registry, the repository and the host provider. Although the thesis does not make any attempt to remodel the existing query compiler, it makes certain extensions to the compiler and the DQP data resource which allows DQP to take into consideration the dynamic deployment concepts during query compilation and processing activities. The chapter also proposes a performance feedback model for triggering the deployment of a database snapshot within the local network in certain scenarios, thereby proposing a "moving the data closer to the computation" paradigm as opposed to the commonly used "moving computation closer to the data."

The evaluation of the extended OGSA-DQP system which exploits the dynamic deployment features is performed in Chapter 6. The chapter provides a brief discussion about the implementation and the experimental setup. Several experiments were performed in varying circumstances and the results show that the distributed query processing system can benefit from the use of the dynamic

deployment features for the usage scenarios mentioned in the thesis.

Chapter 7 provides a summary of the research carried out in the thesis together with a discussion about the benefits of adopting dynamic service deployment features within the context of distributed query processing using OGSA-DQP. The chapter also proposes some areas which can be explored further in future research.

Chapter 2

Background and Related Work

This chapter discusses the background behind the thesis concentrating on the work relevant to the development of the *Service Oriented Distributed Query Processing framework* and the *Dynamic Service Oriented Architecture framework*. It looks into the *Grid Problem*, service-orientation and available service-oriented technologies, databases and the grid, data access mechanisms using service-oriented technologies, parallel database techniques and mobility within the Grid context.

2.1 The Grid Problem

> *"Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed 'autonomous' resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements."* [30]

In the mid-1990s, the concept of Grid was first laid out in front of the scientific community. It was explained using the famous *electric power grid metaphor* where similarities were drawn between the standard way in which electrical power is delivered to a consumer's premises such that any device with the requirement of electrical power can be plugged into a distribution socket and the requirement of a similar computing infrastructure needing standard interfaces that will be capable of providing access to computational resources distributed over the network. In [31] this is explained as "the current status of computation is analogous in some respects to that of electricity around 1910. At that time, electric power generation was possible, and new devices were being devised that

depended on electric power, but the need for each user to build and operate a new generator hindered use. The truly revolutionary development was not, in fact, electricity, but the electric power grid and the associated transmission and distribution technologies." The essence of this metaphor is that the consumer uses electricity by simply plugging the device into a compatible socket with a completely agnostic view about the place where it was produced or how it was delivered - in a similar way, consumers who wish to utilise remote computational and storage resources that are distributed over the network should remain agnostic about the physical location of those resources, or how the actual request is processed, and this should be enabled by a set of standard interfaces similar to the concept of a standard electrical power socket.

Within the decade, this concept gave rise to a new and important field of computing, which although considered to be within the field of distributed computing, is distinguished from the conventional methods due to its focus on collaborative resource sharing. It might be suitable to define the "Grid problem" as flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and organisations which are referred to as *virtual organisations* [32]. The challenges offered by such a setting are related to issues such as authentication, authorisation, resource access, resource discovery, etc. Grid technologies are suitable for addressing such challenges. In [31], Foster and Kesselman defined the computational Grid as follows:

> *"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities."*

The concept of the Grid became well-known, and there were an increasing number of scientific research projects where large scale data sets were offered as resources for sharing with the community. The high energy physics experiments at European Organisation for Particle Physics (CERN) can be cited as an example which started producing an astronomical magnitude of data (to the order of petabytes, i.e. 10^{15} bytes). It became evident that the researchers in such scientific communities from across the world will need to access data from these databases that are geographically distributed and analyse them on computational resources that are spread across the world as well [33]. Chervenak and Foster et. al recognised this need and proposed an architecture for the "Data Grid" in [34] which had storage systems, data access and metadata services at its core.

This requirement for a "data grid" demanded the existence of an efficient, robust, and distributed middleware, as the backbone of the system. This middleware should allow seamless access to all the entities distributed throughout the network, implemented on different platforms and in different languages. This middleware should offer an efficient resource management and scheduling mechanism to deal with the shared usage of the resources. The implementation of such a grid middleware faces

several challenges. In short, the principal requirements to be met by this grid infrastructure are:

- *Information services*: Information about the resources available on the Grid should be accessible through information services. This information should be automatically maintained and continuously updated over time.
- *Resource Brokering*: Grid users should submit their requests to a resource broker specifying their high level requirements. The resource broker should be able to find and allocate suitable resources by querying the information services. This leads to a dynamic environment where resources can be acquired on demand and released after use.
- *Uniform access to resources*: All the resources of the same kind (computational entities, storage elements, etc.) should be accessed in a uniform way, irrespective of the underlying standards and technologies. This should be done through software modules installed on each single system that hide heterogeneity and provide uniform access interfaces.
- *Security*: Grid technologies should be able to provide security mechanisms that enable system administrators to enforce access rules for all the resources made available on the Grid. The use of X.509 certificates and proxy delegation allow systems to verify the identity of the user without exposing their credentials on the Internet. The use of encryption ensures that confidentiality is preserved.
- *Job scheduling*: Jobs submitted by the users should be effectively scheduled on available resources based on well-defined policies.
- *Data Access*: Grid users should be able to access distributed data in a uniform fashion from databases spread all over the world irrespective of the underlying database technology and environment.

2.2 Service Orientation and The Grid

Traditionally, "grid systems" have been synonymous with "distributed job scheduling systems", where systems responsible for the management of the resources were based on the *job* abstraction. Grid infrastructures supported by Condor [35], GRAM [36], ICENI [37] all are examples of such *job-based* systems. Almost at the same time, the concept of *service orientation* was embraced by the e-business community to overcome the challenges offered by *internet-scale distributed business applications*. With the advent of *service oriented technologies*, a contemporary view of the Grid

based on *service orientation* started to evolve - a Grid framework which is based on existing service oriented technologies.

Conceptually, the "Grid Problem" can be considered as a perfect playground for the concepts of Service Orientation. The dependencies between various components of a software system can be minimised resulting in a loosely-coupled architecture which is the main essence of a service oriented architecture. A service can be defined as a unit of work done by a service provider to achieve desired end results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners. In the context of *virtual organisations*, these loosely coupled software components are some of the basic building blocks.

The concept of a *service* perhaps existed in the real world as long as time can go back. Almost every interaction in a consumer market may be treated as buying or selling a service, such as making a travel reservation through a travel agent. But it is a relatively new and still evolving concept in the world of information technology. The idea of service-orientation complements object-orientation and applies the lessons learnt from *component software*, *message-oriented middleware*, and *distributed object computing*. The primary difference between service-orientation and object-orientation is how the term "application" is defined by these two architectural models. In object-oriented development, applications are built from interdependent class libraries, where objects are tightly coupled to each other. In a service-oriented framework, applications are built from *individual autonomous services*.

> *"A service is simply a program that one interacts with via message exchanges. A set of deployed services is a system. Individual services are built to last the availability and stability of a given service is critical. The aggregate system of services is built to allow for change the system must adapt to the presence of new services that appear a long time after the original services and clients have been deployed, and these must not break functionality." - Don Box on Indigo* [38]

Service-orientation is based on the following fundamental theories [38]:

- *Explicit Boundaries*: A complex service-oriented system may often be composed of several atomic services that may actually be spread over large geographical distances, can belong to multiple organisations which may or may not have the similar level of trust amongst themselves, and can also have different execution environments. Crossing these various boundaries may potentially be costly in terms of complexity and performance. Thus, service-oriented systems rely on explicit message passing between cross-border entities rather than implicit method

invocation. The details of a method call is hidden behind the implementation of the service and is not visible outside the service boundary.

- *Service Autonomy*: Each service in a service-oriented system can be autonomous. There is no single entity which assumes control over all parts of a running system. Departing from the standard process of object-oriented system where an application is deployed in totality, the component services in a service-oriented system may be atomically deployed; and in practice, the atomic services may even be deployed much before they are consumed by a composite service-oriented system.

- *Sharing of contract and schema*: Unlike object-oriented systems, which always interact between themselves in terms of classes or objects, composite services in a service-oriented system should almost always communicate by messages. These components do not share the classes or objects between themselves. They interact based on the schema (for structures) and a contract (for behaviour). The Web Service Description Language (WSDL) document describing a Web Service is an example of such a contract/schema exposed by the service for others to interact with it. This "almost legal" contract is machine-readable and verifiable, and hence allows each incoming request to be verified at the receiving end. The machine-readability allows different types of environments to host the service.

- *Semantic compatibility based on policy*: Structural and semantic compatibility in service-oriented systems are two orthogonal issues as opposed to object-oriented systems. The structural compatibility is validated (and enforced) by the schema and contract; whereas the semantic compatibility is based of the set of policies defined by the service. Each service should publish its capabilities and requirements in machine-readable policy statements which express the conditions and assertions that must hold true for normal operation of the service.

The concept of service-orientation is illustrated in Figure 2.1.

With the advent of *Service Orientation*, several different technologies started evolving. All of these were intended to achieve the common goal - architecting complex systems using autonomous, inter-operable, loosely-coupled distributed services. The architecture involving the services came to be known as the *Service Oriented Architecture* or *SOA*. It departs from the traditional distributed systems like *CORBA, RMI* etc in the sense that the architecture comprises of services that are *loosely coupled and highly inter-operable.* They interact by sending and receiving messages on the basis of a shared formalised contract which is independent of the platform and the development environment on which the service is built, for example, the operating system, programming language, web application server etc. It became possible to create complex services by combining more than

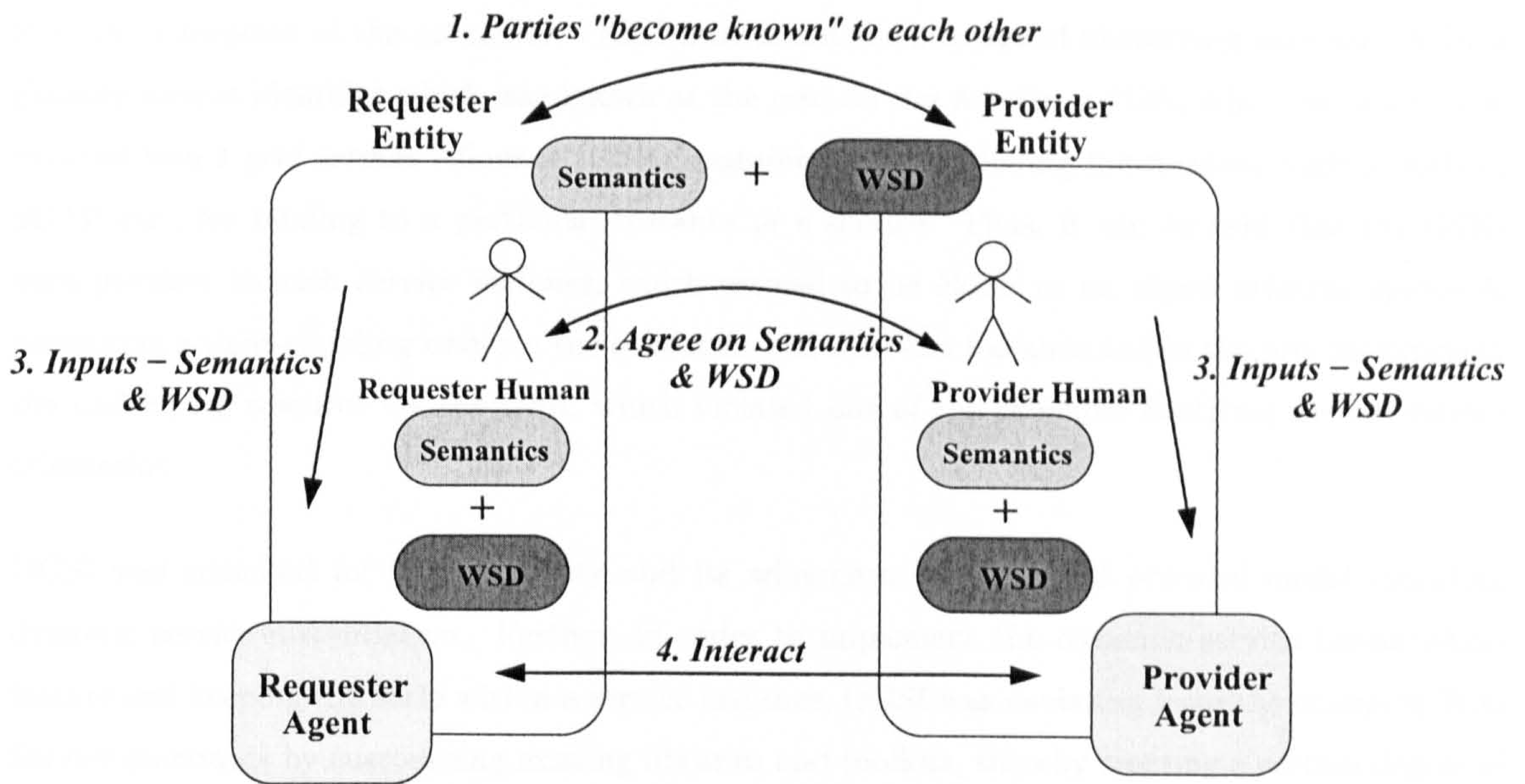


Figure 2.1: Component roles in Service interaction

one autonomous services, where high-level languages such as *BPEL* [39] are used for orchestrating the independent services.

The standardisation body *OASIS* [40] gives the following definitions for SOA:

> *A* ***service oriented architecture*** *is a paradigm for organizing and utilizing* ***distributed capabilities*** *that may be under the control of different ownership domains. It provides a uniform means to* ***offer, discover, interact with*** *and* ***use*** *capabilities to produce desired effects consistent with measurable preconditions and expectations.*

The idea of Grid converged with the developments in *service oriented architectures* and gave birth to the service-oriented view of the Grid middleware in form of the Open Grid Services Architecture (OGSA) [41] which attempted to address the issues involving collaborative resource sharing amongst virtual organisations [32]. A prototype design, known as the Open Grid Services Infrastructure (OGSI) [28] was produced which was built on top of the original job-based Globus [4] and other systems, but also incorporated the evolving service oriented technologies. OGSI introduced *Grid Services* as the building block of this grid middleware which were built upon and extended the service oriented technologies proposed for *Web Services.* Grid Services were Web Services with a set of well-defined interfaces which addressed issues such as *service discovery*, *dynamic service creation*, *service lifecycle management* and *notification*. The argument produced in favour of this extension to web services was that the web services did not support dynamic creation and were stateless

from the viewpoint of the consumer. OGSI introduced the concept of identifying each service by a globally unique identifier which was known as the *grid service handle* or GSH, which required to be resolved into a *grid service reference* (GSR) containing all the binding information, such as SMTP, SOAP etc., for binding to a particular instance of a service. Thus, it can be said that the GSRs were *pointers* to each *service instance*, which seemed to be closer to an *object oriented approach*, promoting a tight-coupling between the consumer and a service instance and in the process exposing the underlying resource via the GSR, which violated one of the principal doctrines behind service orientation.

OGSI was criticised for its complexity and its adherence to the *object oriented model* including dynamic service instantiation. Further, in order to implement the dynamic service instantiation feature and keeping the state within a service instance, OGSI was deviating from the emerging Web Service standards by customising existing libraries and toolkits, thereby creating a certain degree of incompatibility between the systems which defeated the very idea behind the emergence of service orientation. It was argued by contemporary researchers that the requirements of a Grid service can be successfully implemented by existing Web Service standards, technologies and toolkits. One such proposal put forward as an alternative approach to OGSI was the **Web Services - Grid Application Framework** (WS-GAF) [27] which challenged the concept of using references in the form of GSHs and GSRs to identify service instances, and encapsulating service state in an object-oriented fashion. WS-GAF proposed a framework which was built upon existing Web Service standards and toolkits, and advocated *loose-coupling* between the service and the resource. It also proposed that the service state, if any, is internal to the service, and an *interaction state* could be associated to each message exchange to correlate the messages with its *execution state.* WS-GAF ruled out the requirement for binding the consumer with a particular instance of a Web Service and proposed that the only interaction from the consumer point of view with a Web Service should happen by sending a message to the network address (endpoint) exposed by the Web Service and any binding, if at all required, should happen behind the interface to which the consumer should remain agnostic.

As a result of several such criticisms, OGSI was refactored [42] into a set of proposals, collectively known as the WS-Resource Framework (WS-RF) [43, 44]. According to Foster et. al. in [45], in the WS-Resource approach, the goal is to "model state as stateful resources and codify the relationship between Web services and stateful resources in terms of the implied resource pattern, a set of conventions on Web services technologies, particularly XML, WSDL, and WS-Addressing [46]." In this approach, a *stateful resource* can (i) have a specific set of state data which can be expressed in an XML document, (ii) have a well-defined lifecycle and (iii) may have a one-to-many mapping with Web Services. *WS-Resource* is a collective description of a *stateful resource* and a Web Service

together. There are provisions for accessing the resource properties and managing the resource lifetime via the Web Service interface.

The WS-Addressing [46] specification was proposed to standardise the way of describing the network endpoint by the *endpoint reference construct*. It allows the *endpoint reference* to contain apart from the endpoint address of the Web Service, additional information such as metadata associated with the Web Service and reference properties which may be used to qualify the Web Service address. Such an *endpoint reference* is used to identify a WS-Resource and send an invocation to it. WS-Resource Framework uses a *WS-Resource Factory Pattern* which is *stateful* in nature to instantiate the correct resource for the consumer, and associates this new instance with an *endpoint reference* which is returned to the consumer for further conversations. As a resource can be associated with one or more Web Services, it is capable of processing multiple simultaneous messages.

Although the WS-Resource Framework invited similar type of criticisms because of its complexity and resemblance with the earlier OGSI model in terms of *instantiation* of resources, exposing a resource with an identifier and the handling of state, it was nevertheless accepted as a Web Service standard by OASIS in April, 2006. At the same time, there was another set of specifications forwarded by large industrial corporations such as IBM, Hewlett Packard, Intel and Microsoft [47] which combined and built upon WS-Eventing [48], WS-MetadataExchange [49], WS-ResourceTransfer [50] etc. It was thought that WS-ResourceFramework may be superceded by this new set of specifications. The alternative approach was the use of the WS-I standards and procedures. The Web Service Interoperability Organization (WS-I) [20] released the WS-I Basic Profile [21] specifications and based on the proposal of building Grid applications with Web Service standards in order to support the e-Science projects in the UK, the Open Middleware Infrastructure Institute started to build on WS-I to create WS-I+ [29].

2.3 Databases and The Grid

Research into Distributed Database technology started more than three decades ago with a main focus of Distributed Data Management for organisations who had several sites for organisational data. But due to instability in communication technologies, and lack of strong demand, the initial systems did not have much success [51]. The situation has dramatically changed today, and the advances in technology has made Distributed Query Processing feasible and growth in storable and analysable data has made it a necessity. As was discussed earlier in Section 2.1, the Grid problem opens up a plethora of situations where databases will be spread all over The Grid. Applications

will need access to this distributed data from remote locations, and will need to analyse them. The storage service provided by Amazon (Amazon S3 Storage [52]) is an example of the view of the scientific community and service providers about distributed storage systems over the Internet. The need for data access and integration over an internet-scale grid raises some interesting issues [53].

To address the requirements for databases over the grid, the basic issues evolve around:

- *Different types of database*: Different vendors provide different types of database access which are specific to the database provider. Even accessing these databases over JDBC (Java Database Connectivity) would require a different set of JDBC plug-ins for different databases. Further, there can be semi-structural databases, such as XML; unstructured databases, such as files; All these would have different means of access.
- *Data Integration*: The applications accessing the data from different types of database may require integration of data, for example, some data from a relational database may need to be integrated with some data from an XML database for a complete result. This requires some mechanisms to integrate different types of data.
- *Data Transport*: Some applications may request for huge chunks of data and then perform some computation on this dataset to retrieve a smaller result. In such cases, it might be beneficial to do the computation at source, i.e. at the site of the database, rather than transporting the large chunk of data over the wire.
- *Resource Acquisition*: The grid environment allows dynamic acquisition of computational resources. This can be an important feature in Distributed Query Processing over the Grid where availability of resources can govern the scheduling of the query execution plan.

All these make *Distributed Query Processing* a challenging issue within the "Grid Problem". Initially, the support for databases within the Grid middleware was limited. During early 2000s, Globus [4] and the Storage Request Broker (SRB) [54] were considered as most important amongst the emerging Grid infrastructures. Globus primarily focussed on file-based data [31], with a view to access data using GridFTP [55], a high performance file transfer tool, which was designed with the support for security using the Grid Security Infrastructure(GSI). The Storage Request Broker (SRB) concentrated on file-based data too, with some additional features such as catalogues and metadata server (MCAT), logical naming convention for datasets, support for GSI etc. SRB was also capable of supporting BLOBs (Binary large Objects) in a traditional DBMS. But neither of the two available middlewares had complete support for standard database features which underlined the requirement of a new breed of middleware services that could provide seamless access to distributed

data sources irrespective of the database structure, format, vendor and platform.

2.3.1 Data Access and Integration Services

The Grid needed the support for databases at the middleware level in order to be able to allow users access data from Grid databases. Many research projects, such as the high energy physics experiments at European Organization for Particle Physics (CERN) [56] or the SLOAN Digital Sky Survey [23] were inherently data-centric. New research fields such as Bio-Informatics and Neuro-Informatics were heavily dependent on data stored in various different formats within data storages of different types. The following table in Figure 2.2 [57] shows the amount of data that are being or will be generated by recent experiments and real life scenarios.

Experiments/scenarios	Data generated
Computational Fluid Dynamics turbulence simulations	100TB
BaBar particle physics experiment	1TB/day
CERN Large Hedron Collider	1GB/second or 10PB/year
VLBA radio telescope	1GB/second
NCBI/EMBL database	0.5TB, doubles each year
Brain Imaging	4TB/brain (full colour, 10mm resolution)
Disney/Pixar	100TB/movie

Figure 2.2: The potential data explosion

The complexity of the problem was two fold - firstly, there was a need for reconciling implementation difference between various database server products from different vendors, such as Microsoft, IBM, Oracle into a single database paradigm, and secondly, there was the need for reconciling the differences between several different database paradigms, such as object, relational and XML. No out-of-the-box database products had features that would allow it to be integrated with the Grid directly. Further, each database product was the result of a huge amount of effort over the years, and consisted of several important features like security, scalability, performance etc. - and it would have been extremely costly in terms of effort to develop a Grid-enabled database from the scratch. Thus, the requirement for a data access and integration support for accessing databases distributed over the grid grew and the required functionalities became obvious. These can be listed as follows:

- *Type Independence* - The data access mechanism must support querying over different type of data resources, such as relational, XML, files which are exposed over the Grid.
- *Uniform Approach* - It must provide a uniform way of querying, updating, transforming and delivering data in a consistent way independent of the underlying resource.
- *Access to metadata* - It must allow access to the metadata about the data and the resources

where the data is stored.

Almost at the same time the concept of *service orientation* was being applied to the already available grid middlewares, such as Globus, giving birth to the OGSA approach. Proposals for a standardised approach to data access from Grid-enabled databases were created by the Database Access and Integration Services (DAIS) [58] consortium which is a *working group* of the Open Grid Forum (OGF) [59]. This gave rise to a set of standard *Web Services* for accessing data from distributed databases in a uniform way, independent of the structure of the data and type of the storage, known as *OGSA-DAI* [14]. This is described in the following section.

2.3.1.1 OGSA-DAI

The growing requirement for a middleware level support for accessing various types of data storage in a manner that will be agnostic about the storage type, format, vendor and platform along with the rise in Web Service standards and toolkits coupled with the development of OGSA were the vital push for the development of OGSA-DAI. Initially built on top of OGSI, OGSA-DAI is a data access component for the Grid middleware which allows access to distributed data sources irrespective of their location, platform, format and type. In [60] OGSA-DAI has been described as complimentary to other approaches adopted by database vendors which generally support invocation of Web Services from SQL queries, or creation of Web Services from stored procedures, as discussed in [61]. OGSA-DAI allows the organisations exposing the data to the Grid to reuse code in an efficient way, and the consumer accessing data exposed in such manner to implement their client application in a manner which is mostly independent of the specifics of the database, such as the database driver technology, data formatting and delivery techniques. The aim of OGSA-DAI is to enable seamless access to disparate, heterogeneous data sources which are factored out as services which build on and integrate with the OGSA technologies for features like data transport and security. Other higher level services such as distributed querying or federation offering more functionality are able to use OGSA-DAI. The basic components of the OGSA-DAI framework are shown in Figure 2.3.

OGSA-DAI provides several architectural features, such as standardised interface, metadata, security etc., which are considered essential for a Grid infrastructure. These features are outlined below.

- *Standard Interfaces* - One of the major benefits of OGSA-DAI is that it provides a uniform interface over different types of databases irrespective of the underlying technology. The database connectivity issues are hidden behind this consistent interface which allows the consumers to access data from a relational or an XML database in the same way regardless of the specific

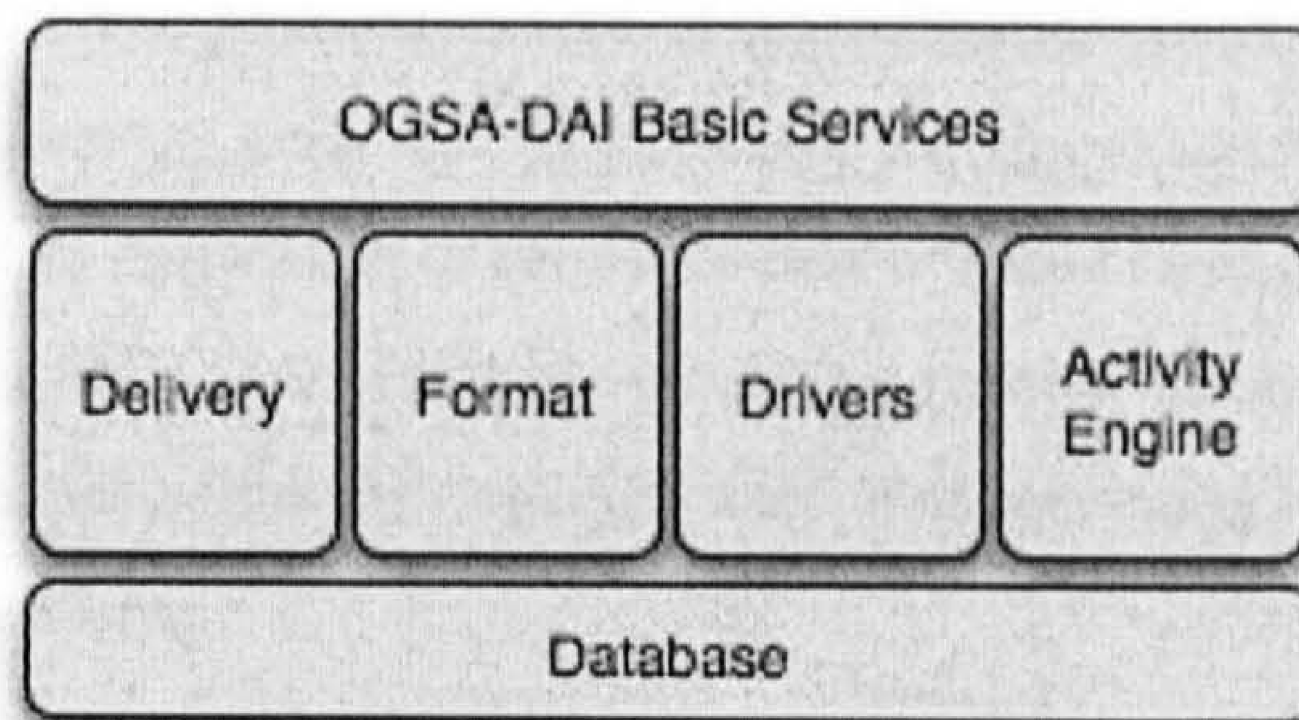


Figure 2.3: Basic Architecture of OGSA-DAI

database paradigm.

- *Access to metadata* - OGSA-DAI provides metadata about the DBMS system that is exposed to the Grid as well as their capabilities through the service interfaces. Metadata about inherent capabilities of the services themselves, such as different delivery options or available transformation mechanisms which can be applied over the data, can also be accessed. Some higher level services, such as a *Distributed Query Processing Service* may require access to the database schema for relational database systems, which can be extracted from the OGSA-DAI service interfaces. Further, OGSA-DAI provides an extensible metadata framework which can be extended to satisfy the requirements of the applications that use OGSA-DAI.

- *Sessions* - OGSA-DAI has a notion of *session* which allows the framework to relate queries submitted by a consumer to a particular transactional context. DBMSs supporting concurrent accesses from multiple consumers provide an interface through which a new session is created every time a client establishes a connection to the database. This allows multiple concurrent transactions to be executed on the same DBMS. OGSA-DAI exposes this mechanism through its service interface.

- *Security* - Security is an important aspect in Grid systems, but various security models are still in the process of development. The current approach taken in OGSA-DAI is based on the X.509 certificates with a view that the universally agreed method will be adopted when there is a consensus within the Grid community.

- *Collective requests* - OGSA-DAI allows the consumers to specify multiple related activities within a single request document which is sent to the service. The activities may include an update to the database, followed by a query and the delivery of the results to a third-party with some transformation. The granularity of the interactions is thus increased, reducing the number of messages to be exchanged between the parties to achieve the same result otherwise.

- *Delivery options* - The synchronous model of a traditional client-server database interaction, such as a JDBC connection, may not be sufficient for the extreme performance (as shown in the table in Figure 2.2) and capacity requirements of some Grid applications. Other delivery options, such as using FTP to transfer the results to another file system, or to store the results of a query at the location of the service itself until it is requested for, are potentially useful for data intensive applications.

The *Grid Data Service* (GDS) from OGSA-DAI provides a document-oriented interface for submitting requests to a database with the option of including multiple related database activities within a single request document. The *perform* operation provided by a GDS carries out all database access and updates based on the request document passed as an input to it. The request document in OGSA-DAI can include a collection of related activities where each activity represents an operation on the service, for example, an updation of a database table, followed by a query and then transformation of the results based on some XSLT stylesheet and finally delivery to a remote file system via FTP. At its core, the GDS has an enactment engine which supports various *activities* such as querying, updates, delivery, transformation etc. OGSA-DAI also allows users to develop new activities and incorporate them into the basic OGSA-DAI framework allowing it to be extended. It is mentioned in [60] that the GDS activity model is not viewed as a complete *Workflow Enactment Engine* but is designed to support "limited expressiveness" for common data access and transformation tasks. Listing 2.1 shows an example of a *perform document* as submitted to the OGSA-DAI framework.

The emergence of OGSA-DAI as a standard interface to databases on the Grid prompted various research projects to adopt this framework as a tool to access databases in a uniform way irrespective of the database structural and functional differences. This gave rise to the requirement of a *Distributed Query Processing* system which would be able to process distributed queries across databases that are exposed as OGSA-DAI services.

2.3.2 Distributed Query Processing, The Grid and SOA

The argument in favour of a *Distributed Query Processing system over the Grid* was put forward by Smith et. al. in [62]. Before OGSA-DAI, most work on data storage, access and transfer within a Grid setting primarily focussed on files, which is one of the central requirements for many applications [33]. But, as outlined in Section 2.3 and [53], facilities for the management of the Grid metadata and support for storage and analysis of application data are provided by database management systems, and these are considered to be important for a range of Grid applications

Listing 2.1 An example perform document submitted to the OGSA-DAI framework

```
<perform xmlns="http://ogsadai.org.uk/namespaces/2005/10/types">
    <documentation>
    This example demonstrates how to parameterise an SQL query statement using a deliverFrom
    activity. A deliverFromGDT activity is used, but any other deliverFrom activity could be
    used in the same way. The values from the delivery are inserted into the query, the query
    performed and the results delivered in the response document.
    </documentation>

                                   describing the delivery activity
      <deliverFromGDT name="parameters">
          <fromGDT streamId="someOutputStream" mode="block">
          http://path/URL
          </fromGDT>
          <toLocal name="parametersOutput"/>
      </deliverFromGDT>

                              describing the parameterized SQL query activity
      <sqlQueryStatement name="statement">
          <sqlParameter position="1" from="parametersOutput"/>
          <sqlParameter position="2" from="parametersOutput"/>
          <expression>
          select * from littleblackbook where id &gt; ? and id &lt;= ?
          </expression>
          <resultStream name="statementOutputRS"/>
      </sqlQueryStatement>

                                  describing the transformation activity
      <sqlResultsToXML name="statementRSToXML">
          <resultSet from="statementOutputRS"/>
          <webRowSet name="statementOutput"/>
      </sqlResultsToXML>

</perform>
```

in streams such as Bio-informatics. Further, it is inevitable that such a distributed environment will be composed of multiple data sources which are related, and applications will be accessing data from several such data sources and performing analysis over them. In bio-informatics, for example, different types of data, such as gene and protein sequences, gene ontologies are stored in different specialist data repositories, and often these repositories need to be inter-related for common analytical work. Smith et. al. in [62] discussed the role that can be played by DQP in a Grid environment and proposed a prototype DQP framework, Polar*, running over the Globus toolkit. Polar* was based on earlier work on parallel databases, Polar, an ODMG-compliant parallel object database server [12, 63], although the requirements of a Grid environment introduced several key changes in Polar*. The key aspects of Polar* can be summarised as follows:

- It provided integrated access to multiple data sources thus satisfying an important requirement of Grid applications.
- By allowing operation calls within the data access and combination operations, it provided a mechanism for integrating data and computational resources.
- It provided a generic, declarative, high-level language interface for the Grid data resources.

- It inherited technologies from parallel databases, and thus provided implicit parallelism within DQP in a Grid setting.

The prototype proposed in Polar* was based on the Globus Toolkit (version 2), which was prior to the advent of service orientation. In Polar*, the grid middleware was accessed using a Grid-enabled version of MPI [64], and further, the absence of the service orientation context led to a relatively less seamless access to data and compute resources distributed over the Grid. The growth of service-orientation and the wide adoption of this paradigm in the e-Science community coupled with the availability of generic interfaces for data access from disparate data sources (OGSA-DAI) led to the research into a service-oriented distributed query processing system, which forms the base of this thesis and is discussed in depth in Chapter 3. The *myGrid* project [6], one of the first e-Science pilot projects in the UK, concentrated on bio-informatics research, with a requirement for analysis of bio-informatics data spread over disparate data sources. Research into a service-oriented DQP system started as a collaboration between *myGrid* and OGSA-DAI, and the result was the first public release of OGSA-DQP [11] in September 2003 which enabled *Distributed Query Processing* over databases spread over the "Grid". It provided effective declarative support for *service orchestration*, and was built on the framework that:

- supported queries over standardised "Grid Data Service" (GDS) and other analysis services made available over the grid thereby combining data access with data analysis;
- used the facility provided by Open Grid Services Architecture (OGSA) to dynamically obtain resources required for efficient evaluation of a distributed query;
- adapted techniques from parallel databases to provide implicit parallelism for complex data-intensive requests;
- used emerging standards of GDSs to provide consistent access to database metadata and to interact with the databases on the Grid.

This framework was built on the concept of *Service Orientation* where the query processing procedures were architected as *services* available over the Grid infrastructure. Within this framework, several "grid-enabled" databases were accessed using standardised *Data Access Services* and well-understood query processing procedures made available as services were applied on the data collected from distributed data sources, as shown in Figure 2.4. Effectively, DQP was service-based in two orthogonal senses: firstly, it supported querying over data storage and analysis resources that were made available as *services* and, secondly, its internal components related to the construction of a distributed query plan and their execution on nodes available on the Grid were architected as *services*.

As a result, the framework provided a declarative approach to *service orchestration* over the Grid and also demonstrated that query processing can benefit from the dynamic access to computational and data resources on the Grid. Apart from incorporating the concept of *Service Orientation*, this framework also adapted the techniques from parallel databases in order to parallelise the execution of the distributed query on several computational nodes available on the Grid, and as a result, it established significant improvements in the performance of queries containing complex data-intensive operations.

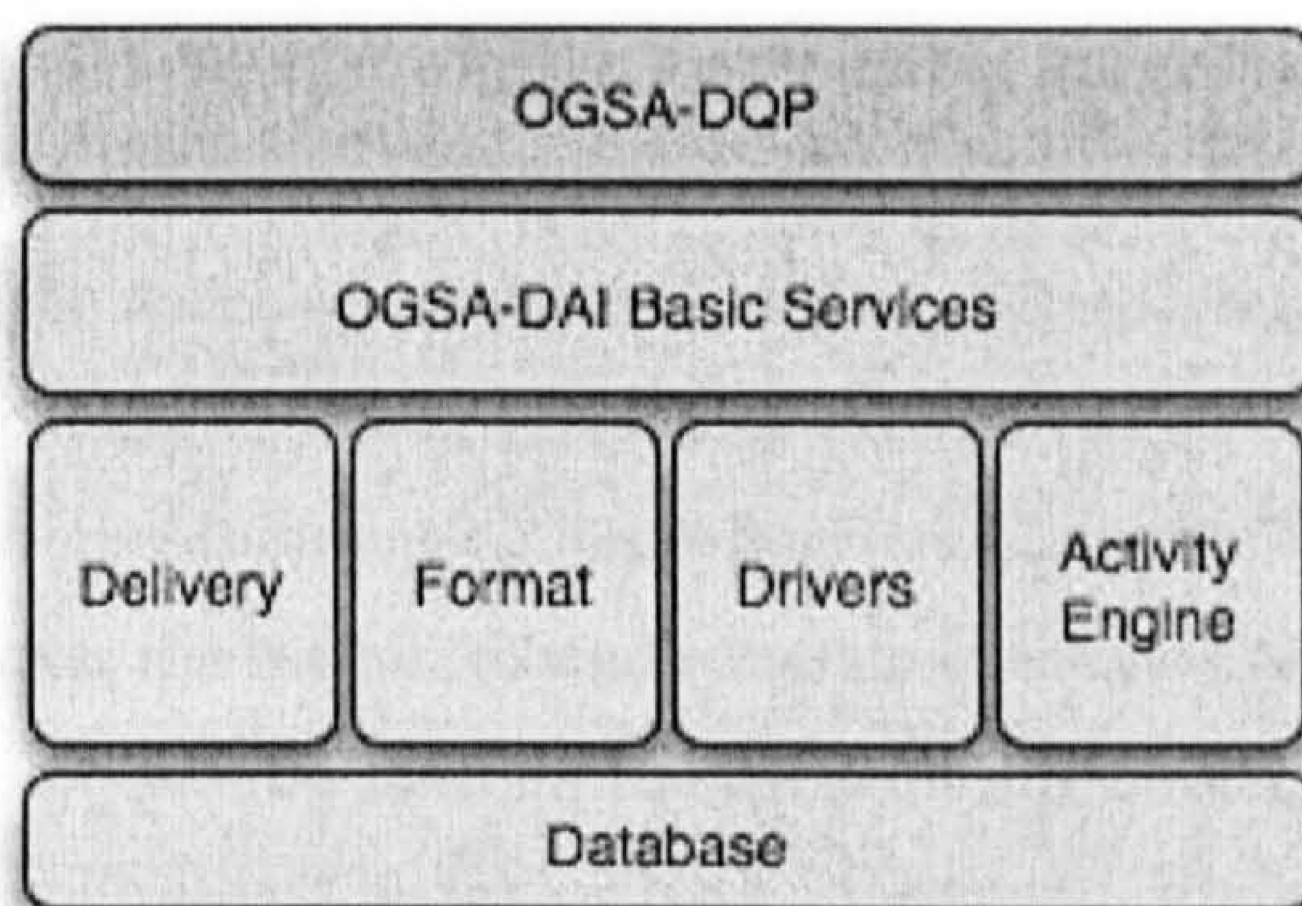


Figure 2.4: Basic Architecture of OGSA-DQP

The popularity of the Web Services led to the development of commercial products that allow integration of Web Services and data management systems [61]. However, the classical *wrapper-mediator* approach proposed in OGSA-DQP where the DQP system acts as a mediator over data sources wrapped by OGSA-DAI is unique in the service-oriented Grid context. One previous proposal, namely *SkyQuery* [65], applies the *wrapper-mediator* approach in a service-based setting, but, it differs from OGSA-DQP in a number of key areas, such as, (i) the data sources are the only services that can contribute in the query evaluation process thus disregarding the option of dynamically allocating the evaluators, (ii) the execution plan in SkyQuery does not incorporate any pipelined parallelism which is provided by DQP by encapsulating the classic *iterator* model [66] and (iii) the query language supported by SkyQuery is a specialised language adopted for astronomical queries. SkyQuery, however, is considered as one of the important early proposals which established the useability of Web Services for supporting distributed queries. Requirements for data-intensive scientific applications motivated other projects such as GRIDDB-Lite [67] which may be mentioned in relation to the work done in OGSA-DQP. GRIDDB-Lite is based on DataCutter [68] in which the users could express the data retrieval tasks as SQL-like queries, although the evaluation procedure was not based on database techniques. It benefited from the declarative manner of expressing complex tasks, but the internal execution mechanism did not exploit the full potential of a DQP

framework.

2.3.3 Requirements for Dynamism

The publicly available version of OGSA-DQP is however not free from issues which are important in the Grid setting, especially in case of data-intensive applications such as the ones presented in the table shown in Figure 2.2. In OGSA-DQP, a consumer can submit queries which may access data sources that are geographically distributed. For long-running queries which retrieve large amounts of data from the network, a relatively high transmission cost may be incurred as the data has to travel over the network to the nodes where the evaluations and analyses take place. An approach to minimise this cost of transporting the data is to deploy the analysis services closer to the data, which has been proposed by the Active Information Repository architecture [7]. Even though OGSA-DQP is able to dynamically discover the evaluators and schedule query evaluation on them, it requires the query evaluation service to be pre-deployed on the participating nodes, either on the nodes hosting the data source, or the computational nodes, or both. A demand-driven deployment feature, which would allow the DQP system to *dynamically deploy* the evaluators on available nodes will greatly enhance the options for selecting the best possible nodes to evaluate a query. In this case, the evaluation services are not tightly coupled to the participating nodes, rather, the service can be deployed at run-time on the nodes which are deemed best suited for evaluating a query, based on their characteristics. Further, it is a common practice in various research streams, such as bio-informatics, to use third party analysis services to analyse the data retrieve from a data source. These services may be hosted on a remote node, which may again incur a heavy transport cost to move the data to the analysis service - a problem which may be resolved if there are means to deploy the analysis service closer to the data. The third party analysis services may be unavailable at the time the query is submitted, resulting in a failure of the query, which can be avoided by deploying a copy of the service locally, again underlining the requirement for a dynamic deployment mechanism.

These issues highlighted the requirement for a flexible and fluid architecture where the available computation and data nodes are "prepared" for query evaluation process as and when required. The Active Information Repository proposed a solution to avoid the network overhead incurred while moving the data between the nodes. Development of a "Dynamic Service Oriented Framework" which would allow demand-driven deployment of services thus emerged as a necessity.

2.4 Towards a Dynamic Service-Oriented Architecture

OGSA-DQP was one of the many scenarios where a dynamic deployment framework could provide substantial benefits. In general, all Grid applications which are data intensive in nature, and where large amounts of data are normally retrieved from a remote data resource to the consumer site for analysis are potential benefactors of the dynamic deployment, which in reality refers to demand-driven installation of the analysis code at run-time in such a way that the host computational node need not be restarted. This is in contrast with the *code on demand* paradigm, an example of which is a *Java applet*, where the execution code is downloaded to the consumer's computer at runtime. In essence, the dynamic deployment referred here is equivalent to *remote evaluation* available in *mobile agent* frameworks or *job scheduling systems*, where the execution code from a consumer is sent to a remote resource for execution. The dynamic deployment methodology adopted in the work discussed in this thesis is otherwise given the name *hot deployment* in many contemporary literature[69]. In this thesis, *DynaSOAr* or *Dynamic Service Oriented Architecture* [19] is proposed as a framework for deploying Web Services on demand over computational resources available over a Grid or the Internet within the context of *Distributed Query Processing* by OGSA-DQP. A consumer request for a service is processed by a host most suited for the requirements specified by the consumer. If there are no existing deployments, an automatic deployment of the service will be triggered within the framework in a way that is transparent to the consumer. In essence, this is analogous to remote job scheduling, with an offer for improved efficiency in the long run as the cost of moving and deploying the service can be shared across the processing of many messages over the time. In this section, the contemporary works in the related area are discussed and assessed against DynaSOAr.

For a certain time, *mobile agents* [8] were considered as one of the primary approaches to enable the dynamism in a Grid environment. But as the Grid community started to move closer to a service-oriented approach, the interest in agent-based systems for the Grid setting diminished as there were concerns regarding the security aspects of such a system[70]. However, some works using the mobile agent technology are noteworthy in this respect. One of them is JASE, a Java-based agent-oriented and service-oriented environment [71]. JASE utilises service abstraction and the remote programming paradigm offered by mobile agent systems. Services, in JASE, are modelled as *agents*. A *Service Interface Agent* is used to encapsulate a local resource, and is composed of two parts - *service agent* and *service interface*. The other agents in the system communicates with a service agent through the encapsulating service interface. The access to the resources on the hosting server is restricted to this service interface agent. The JASE server, which provides the agent environment, and with which the service interface agents register to allow searching, is the core component in the JASE architecture and is responsible for managing the security, mobility, persistence of the agents.

This server is started as an individual process on each of the participating nodes. There are two ways of communication between agents - (i) a Java-based asynchronous messaging system [72], and (ii) communication with the help of shared objects such as a tuple space [73]. But the service hosting environment in JASE is essentially static. Mobile agents are used to locate the services within the network, and then moved to the node where the service is hosted. DynaSOAr on the other hand, allows a demand-driven deployment of the services themselves on available nodes thereby allowing performance enhancements (such as in the case of moving a data access service closer to the data) and load-balancing for the services provided by the system.

Another approach is proposed by Liu and Lewis in [74] where an attempt is made to enable the *web service containers* to accept new mobile code on the fly, so that the new execution code is incorporated into the container dynamically, allowing it to execute within the same address space as the server itself. Liu and Lewis claim three distinct advantages: (i) moving the code to data, where a mobile code component which accesses a remote data source, can be moved to the server where the data is stored, (ii) callbacks, where the consumers can send mobile code which accepts notification messages from the new container allowing service developers to develop more fine-grained notifications by using filters within the mobile code, and (iii) dynamic deployment of new components in a web service. The proposal introduces new languages, such as `C--` for the consumers to write client-side codes, which can then be converted into an XML-based mobile code language, *X#*. It is claimed that this approach will remove any platform dependence as within a "mobile code enabled container", the X# mobile code will be translated into the native language supported by the container. Clearly, the approach is different from what DynaSOAr wants to achieve by dynamically deploying web services with a logical separation between service provider and resource provider. Further, DynaSOAr is completely based on existing standards and toolkits for service orientation, and does not require additional languages to be defined.

WSPeer [75] and FlexiNET [76] are two other proposals for hosting and invoking Web Services in a dynamic way. Unlike other Web Service frameworks, WSPeer [75] is not built around the container model, and its goal was to merge the strengths of Web Services, in particular that of the XML technologies, such as WSDL and SOAP, with the strengths of Peer-to-Peer systems, which allow decentralised resource sharing and discovery mechanisms. The WSPeer architecture allows applications to expose themselves or part of themselves as Web Services by acting as an interface to remote service providers and consumers, and this is referred to as *dynamic deployment*. During a service deployment, a Java class which is a front-end to a process, is passed to the *ServiceDeployer* component of the server interface which deploys this class as a service using Apache Axis [77] and establishes an endpoint through which any interaction with the service will take place. A second deployment method, called *delegated deployment* is also available, where a delegate or proxy compo-

nent is generated and deployed at runtime. This delegated component passes all the invocations back to an object in memory, which can be the application itself, or a component that allows initialisation and driving of sub-processes within the application. Evidently, the *dynamic deployment* in WSPeer is considerably different from the dynamic deployment paradigm proposed by DynaSOAr which allows demand-driven deployment of complete Web Services within a container. Dynamic Service Deployment in FlexiNET [76] aims to define and implement a scalable modular network architecture that will incorporate adequate network elements offering various network functions such as roaming connection control, switching/routing control and advanced service management. The proposal depicts an architecture for dynamic service deployment where a *Web Services Server* is responsible for registering a service with a UDDI repository, and is also capable for discovering other service interfaces. A DSD Controller is responsible for receiving the service requests, download the service code and requirements, check the availability of the resources and select a suitable resource using a matchmaking algorithm, and finally deploying the service on the designated resource, although it is not clear from the proposal how these objects are achieved. FlexiNET is based on the Globus Toolkit 4 [43], and requires specialised network components such as distributed routers, thus ruling itself out of the generic and inter-operable service oriented platforms.

Keidl et. al. examined the dynamic deployment scenarios in the ServiceGlobe system [78, 79, 80]. Services can be stored, published, discovered and deployed on the ServiceGlobe platform. ServiceGlobe distinguish between *external* and *internal* services. *External services* are those that are available on the Internet, provided by third party providers, and may have arbitrary interfaces for invocation. ServiceGlobe uses a concept of *adapters* to transpose internal requests to external interfaces and conversely, in order to integrate these services irrespective of their arbitrary interfaces. *Internal services* are the services provided by ServiceGlobe and are native to the system. These are further classified into *static* services that are location-dependent and *dynamic* services that may be executed on arbitrary ServiceGlobe servers. The native services offered by ServiceGlobe, i.e. the internal services, are mobile code, which are stored in a repository and can be transmitted over the network to the servers running the ServiceGlobe runtime engine where the code can be deployed using a *runtime service loading* service. ServiceGlobe uses UDDI [81] to search for the repository which stores the mobile code for a particular service. The runtime service loading feature allows a service to be distributed to arbitrary hosts within the ServiceGlobe domain, which provides a platform for load-balancing and parallelization. ServiceGlobe uses a method called *dynamic service selection* for invoking a particular implementation of a service. Each service is assigned to a tModel, which in UDDI, is a means for creating any reference items. In ServiceGlobe, the tModel is used to provide a template defining the semantics and service interfaces which implement the template. In ServiceGlobe, therefore, "a service is an *implementation* or *instance* of its tModel" [78]. Service-

Globe offers the creation of *composite services* where it is not necessary to invoke a concrete service endpoint, but is sufficient to invoke a tModel, the implementation of which is decided in a lazy fashion, during compilation or execution. In terms of functionality and scope ServiceGlobe perhaps comes closest to DynaSOAr in that it uses UDDI, and allows *hot deployment* of services which can also be stored in a *repository*. Similarities end at this point, as unlike DynaSOAr, ServiceGlobe does not provide any logical separation between service provisioning and resource provisioning. Further, DynaSOAr does not distinguish between *internal* and *external* services, and is based on WS-I compliant Web Services paradigm, and message oriented approach of invocation, which simplifies the development and deployment process considerably, along with making it easier to use for the consumer. DynaSOAr also allows quality of service specifications from the consumer point of view, which can be taken into considerations while performing dynamic deployment.

The Highly Available Dynamic Deployment Infrastructure (HAND) proposed by Qi et. al. in [82] is another proposal which encompasses the requirements of dynamic deployment of services for Grid applications. HAND acknowledges the need and importance of dynamic service deployment and management in order to enable dynamic and extensible *virtual organisations* which are crucial for the Grid. The proposal considers two approaches for dynamic deployment, (i) *Service-level deployment (HAND-S)*, where one or more services can be activated or deactivated and new services deployed without the requirement of restarting the container, and (ii) *Container-level deployment (HAND-C)*, where deployment of a new service requires the reloading of the entire container. HAND is based on the Globus Toolkit 4 [43], and refactors the kernel structure of the Java WS Core container of Globus (as opposed to other approaches proposed in DynaGrid [83] or by Weissman et. al. in [84, 85] which are discussed later in this section). The approach requires low-level modification of the container but it is claimed that a lightweight dynamic deployment implementation and a simpler management mechanism can be achieved in this fashion. The service-level deployment in HAND allows finer grained deployment of single services as *Grid archive (GAR)* onto the GT-4 container and uses the *ClassLoaders* to activate or deactivate individual services. The experimental results are encouraging showing a better performance for HAND compared to the Apache Tomcat and Axis approach adopted elsewhere, including DynaSOAr. But the cost of deployment is outweighed in DynaSOAr due to the *one-to-many* interaction semantics which allow multiple consumers to interact with the same service endpoint, which is not addressed in HAND. Further, HAND is tightly coupled to a particular container and lacks the generic nature of DynaSOAr. Unlike DynaSOAr, HAND does not address the possibility of creating a marketplace by separating host provisioning and service provisioning.

DynaGrid [83] is another platform based on the Globus Toolkit 4 [43] which allows dynamic deployment of WS-RF services and migration of resources. *ServiceDoor* and *dynamic service launcher*

(DSL) are the two basic components of DynaGrid. A ServiceDoor is a service-specific front-end and exists for each service supported in the framework, keeping track of a list of containers on which the service is deployed and forwards the consumer request to an appropriate DSL on one of the available containers. The ServiceDoor is responsible for the decisions about dynamic service deployment and resource migration. The DSL is a passive service controlled by the ServiceDoor and is responsible for the actual deployment of the service, creation of resources and the invocation of the service. In a WS-RF-based system, a consumer first creates a service resource and gets an EndPointReference (EPR) for this newly created resource. Future invocations use this EPR and are sent directly to the resource. In DynaGrid, the request for creation of a service resource is sent to the ServiceDoor, which uses its scheduling module to select an appropriate container from the list of available containers who have the service already deployed. If no such container can be found, the deployment module initiates the deployment of the service on one of the idle containers. An EPR is created within the scheduling module which refers to the *meta service resource* (a new type of service resource defined in the work which stores the information about the service, such as the service identifier, the interface class, service options, and the class loader object) of the corresponding service within the selected container and the *createResource* method on the target DSL is invoked. A new service resource is created by the DSL using the information contained within the meta service resource, and the local EPR is returned to the ServiceDoor. As DynaGrid allows migration of resources, this EPR is not returned to the consumer, instead, an abstract key is returned and a mapping between the actual EPR and the abstract key is stored within the ServiceDoor. The actual execution or invocation request identified by the abstract key returned to the consumer is also sent to the ServiceDoor which retrieves the target EPR from the mapping information and the target DSL is invoked resulting in the execution of the service leading to transparency in execution. Byun and Kim in [83] claim that this approach does not require any changes in the standard containers as opposed to similar work by Qi et. al. in [82]. DynaGrid also allows migration of service resources from one container to another when certain constraints set by the service developers are met which as per [83] distinguishes it from other WS-RF based platforms. DynaGrid is a relevant work in its own right and indeed presents a view which has similarities with the DynaSOAr concept, but (i) it is restricted to a particular category of services, viz. WS-ResourceFramework, thereby neglecting a large set of services based on the standardised WS-I platform that are used in various e-Science and most commercial projects, (ii) it does not seem to contain a repository allowing developers to upload newly developed services for potential consumers to use and (iii) the separation of service provisioning and computational resource provisioning as in DynaSOAr is not present and nor is the concept of a marketplace.

The idea of dynamic deployment of services is considered a vital step towards forming dynamic virtual organisations by Weissman et. al. in [84, 85]. Weissman in [84] acknowledges that the

static infrastructures currently prevalent will not be adequate enough as the Grid applications are increasingly becoming "multidisciplinary, collaborative, distributed and most importantly, dynamic" and such applications may be "assembled on-the-fly to exist only for a transient period of time." The views expressed in this proposal coincides with the view and motivation behind DynaSOAr, which focusses on using on-demand service deployment as a step for creating dynamic virtual organisations and to deal with unpredictable demand for services. Weissman et. al. present an architecture for dynamic grid services that is based on OGSA and implemented on OGSI [28] for supporting the dynamic virtual organisation concepts. The proposal evolves around a concept of *Adaptive Grid Service (AGS)* which is a fundamental abstraction for a Grid service, that is adaptable to changes in demand and availability of resources. The *AGS* is composed of a front-end, a deployer and a back-end. The front-end is responsible for handling the requests from consumers and making the decisions about where to process the request. The AGS deployer takes the decision about the site(s) which should host and deploy a service, and once a service is deployed, this information is stored within the front-end. An AGS factory which contains the actual code for the service and serves a request by creating an instance known as *Adaptive Grid Service Instance (AGSI)* resides in the back-end. An *Adaptive Resource Provider Service (ARP)* provides a leased pool of resources on which the back-end can be dynamically deployed, and the leasing model conforms to the concept of *service lifecycle* of OGSI. Weissman et. al. have used the Apache Tomcat container to enable dynamic deployment as the stand-alone container available in Globus Toolkit 3 is able to handle statically deployed services only. A mechanism similar to the one used in DynaSOAr is used for packaging the service as a WAR (web archive) file which contains the application code, the deployment descriptor for the web application and other required libraries. Although the concept is similar to DynaSOAr, this proposal is architecturally tied to a particular implementation of the Globus Toolkit and suffers from the limitations pointed out earlier and in [27]. DynaSOAr, on the other hand adopted a more loosely coupled architecture using widely accepted standards and toolkits for Web Services. An approach similar to Weissman et. al. may be seen in Smith et. al. [86] which speaks about a *Service-oriented ad-hoc grid.* In this proposal custom class loaders are used to enable *hot deployment*, and no distinction is made between service provisioning and resource provisioning. Further, the proposal is again tied to a particular platform, that is, Globus Toolkit 3.

An architecture for a *next-generation Internet* based on Web Services and Utility Computing is presented in [87] by Darlington et. al. Utility Computing allows the provision of execution environments by third parties, such as the Amazon Elastic Compute Cloud [88], which provide computational resources to consumers on a use-on-demand, pay-per-use basis. In addition to this, the architecture proposed by the authors create the possibility of a *services market* where Web Services are equipped with the ability to negotiate a price for their usage which has been evaluated within [89]. This

proposal is relevant to DynaSOAr as the DynaSOAr architecture allows the possibility of a similar market for resources and services.

In this section, several contemporary proposals about dynamic service deployment and formation of dynamic virtual organisations have been assessed. The differences between DynaSOAr and all these proposals can be summarised as:

1. DynaSOAr hides the actual deployment behind the service provider interface, so that the transparency of execution is maintained. This is in some ways, analogous to the Amazon S3 Storage Service, where the consumer is allocated some storage space as per the requirements, but the consumer is not required to know the exact location where the data is stored or the format in which it is stored. The only interface the consumer is aware of is the interface to the service provider.

2. DynaSOAr is responsible for making the decisions as to when a dynamic deployment will take place. The decision depends on a number of factors, such as demand, usage statistics, host availability etc.

3. DynaSOAr maintains a loose coupling between the service provider and the consumer. Because the consumer is only aware of the service provider interface, there is no dependency on the consumer's part on the platform on which the service is actually executed or the language in which the service is implemented.

4. DynaSOAr makes a distinction between service provisioning and host provisioning by creating a logical partition between the provider of a service and the provider of a computational resource on which the service is hosted. In effect, a service provider may be different from a resource provider, which creates a possibility of new virtual organisational structures, where several distinct organisations collaborate for sharing the resources in order to achieve a common goal.

5. DynaSOAr creates the possibility of brokering between different available services all of which may perform the same task thereby giving the consumers options to select from all the available services based on the consumer preferences. There is also a possibility for the service provider to choose from different available providers for computational resources. These two features taken together, form a marketplace for all the participants, which in this thesis is termed *Software Hypermarket.*

2.4.1 Use of Virtualization Technologies

Virtualization technologies are considered as one of the key approaches in the dynamic deployment framework covered in this thesis. VMWare [15] defines *virtualization* as "an abstraction layer that decouples the physical hardware from the operating system to deliver greater IT resource utilisation and flexibility". This leads to the opportunity to run multiple virtual machines (VMs) with different operating systems, simultaneously on one physical machine where each of these VMs are completely isolated from each other and from the host environment, removing the tight coupling between the hardware and the software of computational resources which existed in the pre-virtualization era. The traditional operating systems on modern computational resources allow sharing of the resources such as CPU, memory, disk space via multiprocessing capabilities, file-systems and virtual memory. The system resources are accessed by each individual processes indirectly through the abstraction layer provided by the operating system itself. A parallel approach to resource sharing existed since the IBM System/370 [90] where *virtual machines* present a duplicate view of the underlying hardware to the software running within the machines allowing the co-existence of multiple operating systems on the same physical hardware sharing the physical resources through multiplexing. An overview of the virtual machine architectures can be found in [91] by Smith and Nair where the authors categorise virtual machines in two broad categories - (i) those which virtualize a complete instruction set architecture, including user and system instructions, and are known as ISA-VMs, and (ii) those which support an application binary interface with virtualization of system calls, known as ABI-VMs. Smith and Nair also speak about the *classic VMs* as an important class of virtual machines consisting of ISA-VMs which support same-ISA execution of entire operating systems, such as the IBM S/390 series and VMWare, which is used within DynaSOAr as a means for virtualization. On the other hand, UML (User Mode Linux) [92] belongs to the category of ABI-VMs. In the past few years, virtualization has become an attractive option for Grid applications. Figueiredo et. al. in [93] outlines the advantages of using VMs for Grid computing because of the functionalities such as *security and isolation*, *customisation*, *legacy support*, *resource control* etc. offered by the virtualization technology. In this section, some contemporary work about how VMs are used in Grid computing are assessed.

VMPlants [94] is a proposal by Krsul et. al. which attempts to incorporate the functionality to manage dynamic creation and destruction of virtual machines within the Grid middleware. VM-Plants provides support for flexible and automatic customisation of virtual machine environments from a higher-level user perspective. It has the ability to clone and instantiate a VM environment efficiently and monitor the states at execution time based on a service oriented architecture. The goals as claimed by the authors are flexibility of configuration, support for multiple virtualization

technologies and fast instantiation of the VMs, scalability, interoperability and fault-tolerance. Direct Acyclic Graphs (DAG) are used for describing the actions to configure and customise a VM, which together with a VM cloning process allows a flexible configuration process. The management of the virtual machines is not tied to any particular middleware solution and follows a service oriented model. The framework is composed of services such as the *VMShop* which is a front-end to which consumers interact and *VMPlant*, which is responsible for the actual creation of a VM. VMShop uses standard Web Service technologies such as UDDI and WSDL for discovery and service binding. From the point of view of a user, the VMShop performs the tasks of a system administrator to accommodate requests for additional computational resources within the network. The consumer request to the VMShop will contain the desired configuration of the new resource to be created in form of a DAG which are sent to the VMShop as a SOAP message. The VMShop selects a VMPlant for the creation of the VM, and sends a service request to the designated VMPlant which implements a *Production Process Planner (PPP)* to plan the process of VM instantiation based on the specified configuration. It searches a *VM Warehouse* for a VM which matches the desired configuration and clones it to instantiate a VM for the consumer. If the instantiation process is successful, a *classad* containing the information for identifying the VM is returned to the consumer who can communicate with the VM using this identifier.

Sundararaj and Dinda proposed a virtual network tool VNET [95] to deal with the networking issues of virtual machines. In order to successfully use the virtual machines for large scale Grid applications, the VMs should be reachable through the network. The virtualization technologies typically create a virtual ethernet card for the guest virtual machine which is emulated by using the physical network card of the host system. The virtualization layer typically "bridges" this virtual ethernet card with the same network as the physical host, and the VM appears as a normal physical machine on the network. The authors claim that this bridging process works seamlessly within a single site, and new VMs are indistinguishable from real machines within the same network. But the process is not as seamless across different sites, such as when there is a requirement to run a VM on a remote site, in which case, the network presence will largely depend on the policies that are imposed on the remote network. In effect, this may be equivalent to "visiting the site and connecting a new machine" [95]. The issue becomes even more complex as the number of such remote sites increase and if there are any possibilities of migrating the VMs between the sites. VNET is proposed by the authors as a simple layer 2 network tool, using which the VMs will not have any direct network presence at the remote site, rather it provides a mechanism to "project their virtual network cards onto another network" such that all the VMs belonging to a particular user may appear to be within the user's own network. As VNET sits on layer 2 of a network, a particular machine can be migrated between different network without any change in its network presence, i.e. the IP address,

routes etc. VNET uses the concept of *Virtual Private Network* which implements a virtual local area network (VLAN) spread over a wide area network using layer 2 tunnelling. The work is unique and is particularly relevant to the dynamic virtual organisation scenarios which DynaSOAr wants to support. The current virtual networking method used in DynaSOAr is either the default "bridged" or "host-only", but DynaSOAr could utilise tools such as VNET to allow migration of VMs and thus dynamic VOs over a wide geographical area.

The proposal on *Virtual Workspaces* from Keahey et. al. [17] is particularly relevant for the concept of dynamic virtual organisations. The authors here recognise a potential problem in the conventional approach of mapping jobs to resources, where often an assumption is made that the execution environment will be provided independent of the available infrastructure. Situations are possible where applications from different users with widely different requirements try to use the same available resources. The authors propose the concept of *virtual workspace* which can be automatically deployed on resources to provide the required execution environment, resulting in the mapping of jobs to workspaces, which in turn are mapped to actual resources in the Grid. A virtual workspace as characterised by the authors in [17] is "a definition of an execution environment in terms of its hardware requirements, software configuration, isolation properties, and other salient characteristics", which is described by an XML schema and can capture the requirements for the intended execution environment and use automated tools to make it available for use. The prototype is implemented on Globus Toolkit 4 [43] and can use both Xen [16] and the VMWare Workstation [96] for virtualization. The consumers interact with a *VW Factory* with a description of a desired workspace. Negotiation processes may apply predefined policies while creating the workspace, and the resulting workspace is registered in a *VW Repository*, which provides a Grid service interface for management and accounting of the workspaces. An EndpointReference is returned to the client identifying the newly created workspace, which is presented to a *VW Manager* during deployment of a workspace on a resource after which the consumer can perform all the required operations. The Globus Resource Allocation Manager (GRAM) [36] can be used to execute jobs or applications on the workspace once it is started. The proposal is relevant for dynamic virtual organisations as it enables dynamic instantiation of virtual workspaces in form of VMs, but it is limited to the job paradigm, and the fact that it is based on GT4 possibly ties it to a particular implementation.

All these works mentioned in this section underline the importance of virtual machines and the virtualization technology as a whole within the premises of Grid computing. The same outlook is adopted in DynaSOAr and an attempt is made to secure the advantages identified by Figueiredo et. al. in [93]. However, DynaSOAr views virtual machines as a special packaging for services. In the scientific domain, there are certain services, such as BLAST [9], which require a special environment to execute, which may not be present on the available resources. Further, some applications, although

they can be deployed dynamically on any available node, may not function to the best of their ability as they require finer tuning with the underlying hardware. It is viewed in DynaSOAr that such applications can be packaged in a VM and made available as services, which can be deployed on-demand using the same DynaSOAr infrastructure. DynaSOAr adopts a principle of moving the computation closer to the data as outlined in the AIR architecture [7], but, also recognises certain situations where deployment of a snapshot of the data locally may prove to be beneficial. DynaSOAr achieves this by deploying a VM containing a snapshot of the database within the local network. Although deployment of a VM is costly in terms of the deployment time, for frequent long-running requests, the cost is outweighed by the benefits obtained, which are discussed in later chapters of the thesis.

2.5 Exploiting Dynamism in Distributed Query Processing

It was argued in Section 2.3.3 that a dynamic service deployment framework may benefit OGSA-DQP. This has been the final goal of the thesis - to exploit how the dynamic deployment concepts developed in DynaSOAr can be utilised in the service-oriented OGSA-DQP resulting in a dynamic version of DQP which will be more efficient in coping with the inherent dynamism of the Grid. The dynamic version of DQP developed by adapting to the dynamic deployment concepts is explained in greater detail in Chapter 5. This version of DQP incorporates the components developed in DynaSOAr (as shown in Figure 2.5) and is able to collocate the data access and analysis code closer to the actual data source by deploying such services on the nodes which host the data or are closest to it; the evaluation engine is not tightly coupled with the available resources and is deployed as and when required on the nodes that are deemed best suited for evaluating the individual query partitions. This version of DQP also allows a form of data-caching by dynamically deploying snapshots of databases which are actually located on remote nodes on the local network which reduces the cost of transmitting the data over the network to a great extent. This is done for frequent and long running queries, for which a general trend of increasing transmission cost can be observed by analysing the performance feedback from the participating nodes at execution time. This is in contrast with other data replication techniques, such as *replica management techniques* from Oracle [97] which require the existence of a database management system on the target nodes and the intervention of a database administrator to copy the data as an off-line process. In this thesis it is assumed that a database snapshot is available within a VM, and acknowledges that there must be some means to keep the snapshot synchronised with the actual dataset, which is outside the scope of the thesis.

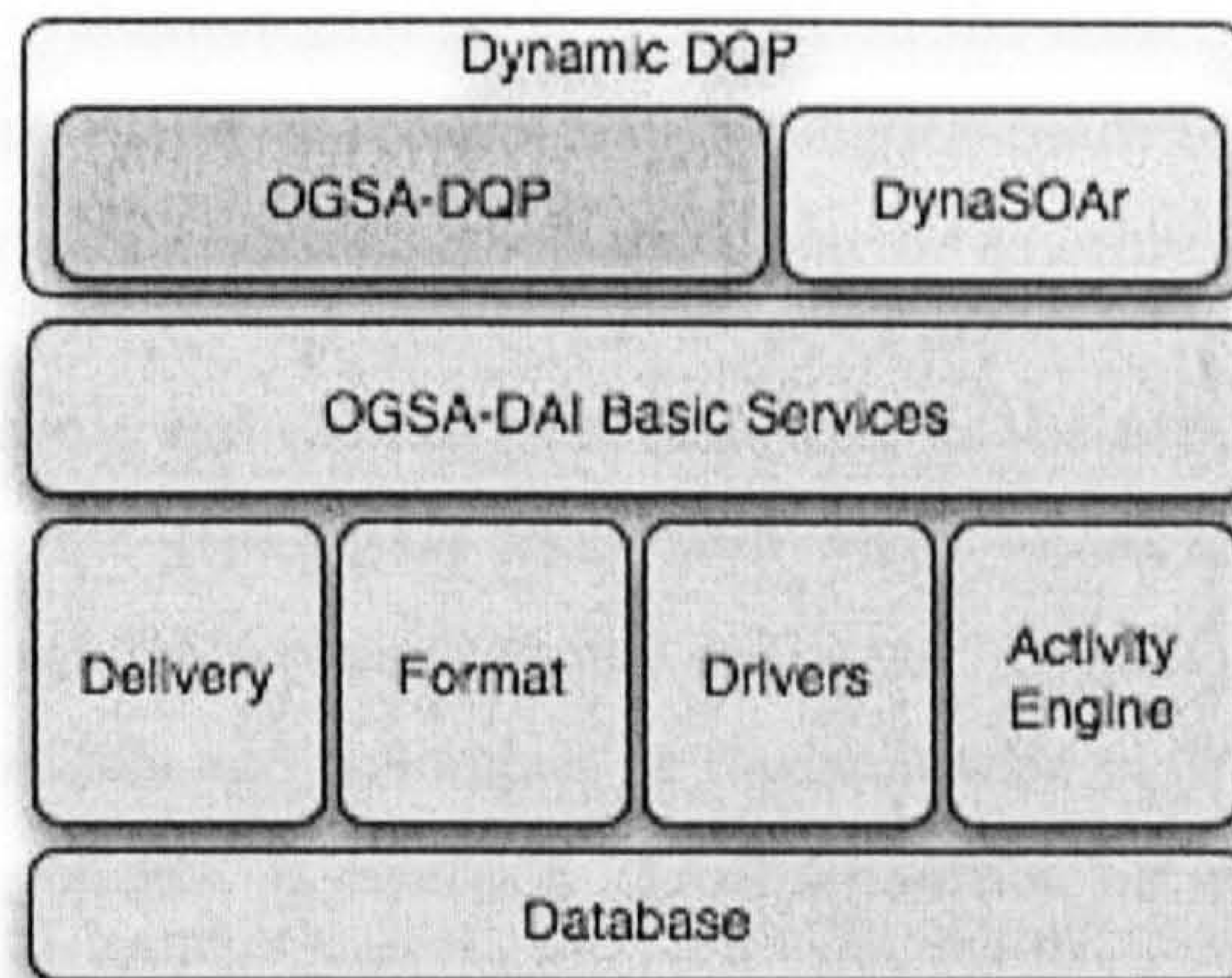


Figure 2.5: Architecture of the dynamic version of OGSA-DQP

2.6 Discussion

The work presented in this thesis attempts to combine the concepts of a service-oriented distributed query processor with that of a dynamic service deployment framework. The service-oriented DQP, or OGSA-DQP is a major development as a framework which is able to execute distributed queries over distributed data resources that are factored out as services. DQP uses existing standards such as OGSA-DAI [14] to access the data from these repositories, and in the process, the database complexities are hidden from the consumer. The query processing engine itself is a collection of services and can be viewed as a declarative service orchestration mechanism. It applies techniques from parallel databases [12, 62] to achieve better results while processing a query and is also able to incorporate analysis services within the query which is deemed as an important requirement in the scientific community.

During the development of OGSA-DQP, a need for a dynamic deployment framework was felt. It was seen that for remote data resources, a high cost of data access was being incurred. Additionally there was a cost for transmitting the data over the network when participating nodes were distributed over a large geographical area. Further, the query processing engine was tightly coupled with the data resources or the pre-configured nodes restricting the option of selecting the best possible nodes for evaluation of a partition during the optimisation process. It was also seen that while invoking a remote analysis service for each row of data retrieved, the cost of invocation was high, which included the cost of transporting the data to the node where the analysis service resided, and this cost was higher for larger row sizes. There were situations where even moving the evaluation engine onto the data nodes were not the best option, for example in situations where all the data nodes

were remote - even if the evaluation process is executed on the data nodes themselves, for queries producing large data sets as results, the cost of transferring the result to the DQP system was costly. Thus it was regarded that a data-caching mechanism may be suitable for such situations.

Motivated by the requirements and existing proposals such as the AIR architecture [7], DynaSOAr started evolving as a dynamic deployment framework which would allow on-demand deployment of services at runtime on available computational resources. DynaSOAr was developed based on standardised Web Services tools and techniques as recommended in [27, 29] and maintains a loose coupling between the components. It creates a logical separation between service provisioning and host provisioning, and the entire deployment mechanism happens in a way that was transparent to the consumer. DynaSOAr also uses virtualization techniques as a means of creating dynamic virtual organisations.

In order to address the requirements of dynamic deployment facilities in OGSA-DQP, the framework developed in DynaSOAr was adopted resulting in a dynamic version of OGSA-DQP. This version of DQP is able to collocate the data access and evaluation services closer to the actual data node based on available network data. It is also able to create a dynamic query processing engine which is fluid in nature as the evaluation services are not tightly coupled with the participating nodes, thus creating the possibility for the optimiser to discover available nodes and accommodate them during the query optimisation phase with a view to schedule a dynamic deployment of the evaluation service on those nodes on which the service does not exist. The dynamic version of DQP also allows on-demand deployment of the analysis service on the best suited nodes in order to reduce the cost of invocation. A form of data caching using virtual machines is also adopted in the dynamic version of DQP. The incorporation of the DynaSOAr framework within DQP also brought in some DynaSOAr features, such as (i) scalability, which is achieved by deploying multiple copies of a service, for example, the analysis service, so that the invocation to this service can be parallelised for better performance, (ii) adaptability, which is provided by the loosely coupled architecture of the framework, where changes in the resource availability and performance trigger reconfiguration of the run-time system, (iii) dependability, which is provided by deploying a new copy of the service if one of the services involved fails. The security of the DynaSOAr system has been researched by Fowler in his thesis [98] and is outside the scope of the work presented in this thesis.

All the aspects of the work presented in the thesis will be discussed in greater depth and an evaluation of the dynamic DQP system will be presented in the later chapters.

Chapter 3

Service Oriented Distributed Query Processing

One of the objectives in this thesis was to create a *Distributed Query Processing* framework which will allow homogeneous access to heterogeneous data resources by using existing infrastructures (such as OGSA-DAI) and evaluate distributed queries by parallel evaluation of query fragments using techniques from parallel databases on a Web Service based query processing engine created at runtime. This chapter describes the architecture and design of a such a framework capable of querying distributed data sources over the grid, publicly available as OGSA-DQP [99, 11], in which query compilation, optimisation and query evaluation are viewed as *services*. Both the query compilation/optimization and the execution take place by exchanging SOAP [5] messages between the component services. The data access mechanism from the data sources is also based on commonly used service for data access and integration, **OGSA-DAI** [14]. The benefits of this architecture include the following:

- Monitoring Services can be used to identify lightly loaded resources that are suitable for query evaluation and to allocate query evaluators on these nodes;
- Grid security supports single sign-on for remote resources, simplifying authentication for distributed execution;
- Consistent resource discovery and allocation mechanisms can be used for both data sources and analysis tools accessed from a query.

3.1 OGSA-DQP as a Service Orchestration Mechanism

OGSA-DQP supports the evaluation of queries expressed in a declarative fashion over one or more services, including data access services and external analysis services. It can be seen as complementary to other service orchestration mechanisms, such as workflow languages. In the field of bio-informatics research, an extremely common workflow is where the bioinformatician uses a data access service to retrieve protein or gene sequence data from a database, and for each retrieved sequence, which may satisfy a certain criteria, invokes a sequence analysis service, such as Blast [9]. OGSA-DQP allows accessing data from distributed data sources using standard data access mechanisms, performs query processing operations such as *project* and *join*, and also allows invocation of external services using a special *operation_call* operator. The same results that were achieved by using the earlier workflow could be obtained by using a query in the OGSA-DQP framework, which will access data from the same database, perform query processing operations on the retrieved data, and for each row that matched the criteria, invoke the same external service, and return the results. This supports the claim that OGSA-DQP can be seen as as an approach alternative to workflow systems as it effectively performs the same task as the workflow mentioned above in a different manner. The creation of the query plan, submission of the partitions to evaluation services who communicate between themselves and the invocation of the analysis service, that is, the entire orchestration of the participating services is performed by DQP, and is transparent to the consumer.

3.2 The Architecture

The distributed query processing framework, OGSA-DQP [99, 11] is a publicly available framework for querying distributed data sources over the grid using a service oriented interface. OGSA-DQP uses and extends the commonly used service for data access and integration, OGSA-DAI [14] and is composed of two major services -

- Grid Distributed Query Service - The Grid Distributed Query Service (GDQS) is an extension of the standard OGSA-DAI service, and is deployed as an OGSA-DAI data service with an exposed data service resource[1]. This is the service which is exposed to the consumer and all interactions from the consumer is directed to this service. When the GDQS is initialised, it interacts with the data services exposing the actual data sources to obtain the metadata

[1]A data service resource implements the core OGSA-DAI functionality. It accepts perform documents from data services, parses and validates them, executes the data-related activities specified within them and constructs response documents. It can also cache data for retrieval by third-parties (if the data service resource is configured to support asynchronous data delivery). Data service resources are accessed via data services [100]

needed for compiling, optimising, partitioning and scheduling the distributed query execution plans over a set of query evaluation nodes (hosting the Query Evaluation Service - the other component service of OGSA-DQP). The GDQS internally uses the Polar* distributed query processor for the Grid [13, 63] by encapsulating its compilation and optimisation capabilities.

- Query Evaluation Service. The Query Evaluation Service (QES) on the other hand is a WS-I compliant Web Service [21] which processes partitions of a query execution plan wrapped inside SOAP messages and communicates with other QESs and third- party web services. Each QES evaluates the partition assigned to it by the GDQS. The QESs implement a physical algebra over the data access services encapsulating the actual data sources whose schemas were imported during the GDQS initialisation phase.

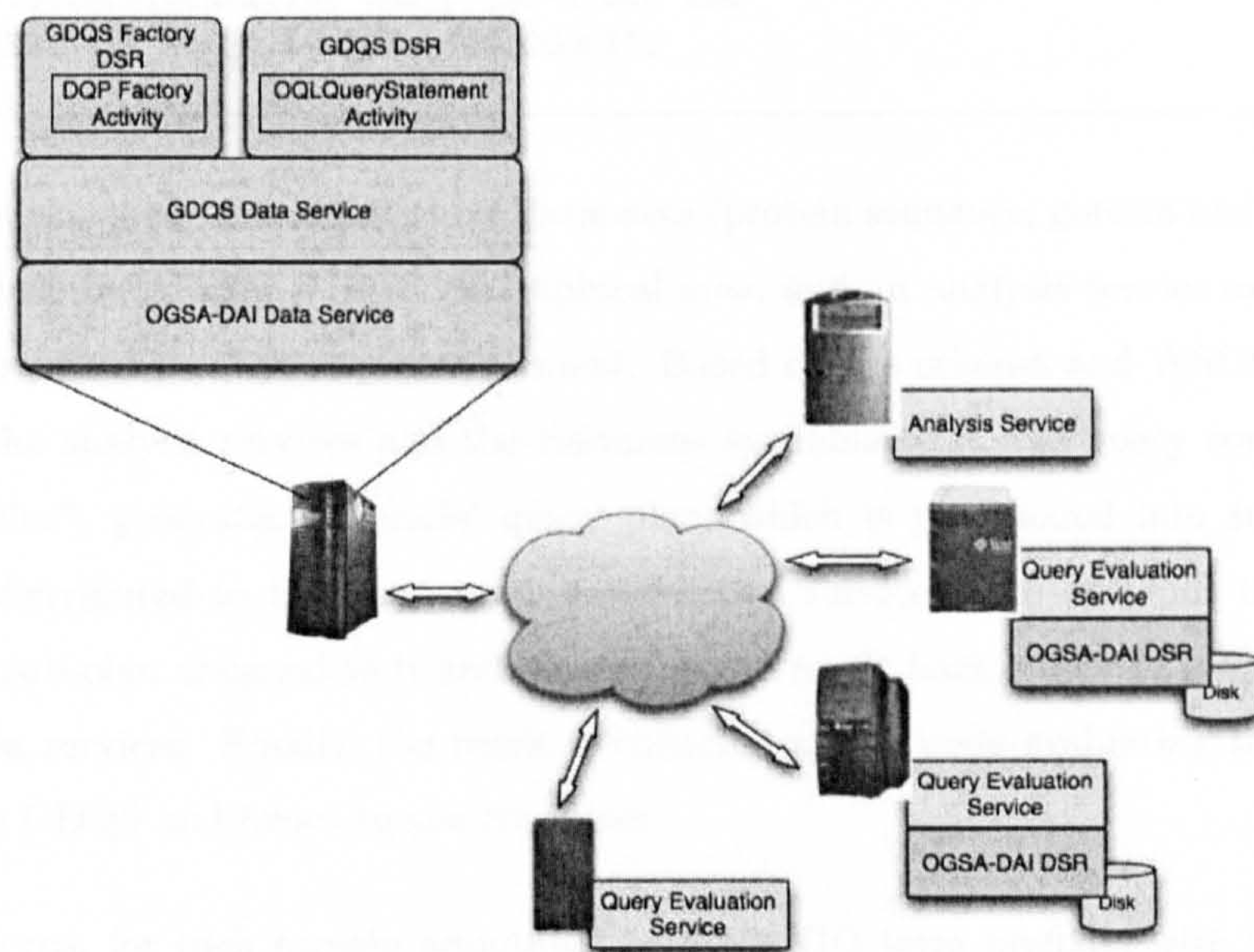


Figure 3.1: Basic DQP Architecture

Figure 3.1 shows the basic architecture of the OGSA-DQP framework. In general, the GDQS must reside on one node on top of the basic OGSA-DAI framework while the actual data resources exposing the data, evaluation and analysis services may reside on other nodes, from where the schemas and metadata are obtained by the GDQS. Once the schemas from the data sources and the WSDL [101] of the analysis service are imported, the OQL query along with the metadata (schema and computational resource) is passed to the compiler/optimiser. The query is compiled, optimised and a set of partitions are generated, each containing a section of the parallel query plan. Each partition, scheduled to execute on individual evaluation nodes, is sent to the corresponding Query Evaluation Service on that node as a SOAP message, and the evaluation starts in each node.

This allows an implicit, parallel evaluation of the individual partitions for a single query with the complexity of the process hidden behind a simplified service interface. The evaluation services and the actual data sources can be located anywhere in the Grid, thus creating an ad-hoc virtual query execution engine based on communicating services. Some QESs interact with the data services to obtain the data after which results start to propagate across the QE services to the GDQS, and eventually to the client.

Listing 3.1 shows a characteristic query that is supported in the DQP framework. This query is written in OQL and based on an ODMG [102] data model setting.

Listing 3.1 An Example Query

```
%print select p.ORF, g.id, calculateEntropy(p.sequence)
from p in protein_sequences, g in goterms, t in protein_goterms
where g.id=t.GOTermIdentifier and p.ORF=t.ORF and
p.ORF like "YBL06%" and g.id like "GO:0000%";
```

In this example, the query spans over three databases (protein sequence, goterm and protein goterm) which can be distributed over a large geographical area, and an analysis service exposed as a Web Service is also invoked on each sequence element. Based on the schema and WSDL imported from the data and the analysis services and the resources available to it, the query compiler/optimizer component, Polar*, generates a parallel query plan, which is partitioned into sub-plans. These sub-plans are distributed to the participating evaluation services each of which is responsible for evaluating the sub-plan assigned to it and conveying the result back either to the root partition or other evaluation services. Finally, the result is collected at the node evaluating the root partition and sent to the GDQS and hence to the consumer.

This query returns, for each protein annotated with the GO term prefixed with 'GO:0000', those proteins that are similar to it along with their GO identifiers. The protein_sequence, goterm and protein_goterm extents are retrieved from three databases, each running under separate MySQL relational database management systems on different hosts, possibly distributed over a large geographical area. The query also invokes the Entropy Analysis Service, which calculates the entropy of a protein sequence. It is to be noted that the query is essentially a select-project-join query but the data is retrieved from three relational databases, and an external application (a web service) is invoked on the results of the join operation. A service-oriented approach to processing this query over a distributed environment allows the optimiser to choose from multiple providers, the concept of service-orientation guaranteeing that most heterogeneities are encapsulated behind uniform interfaces. It also allows the optimiser to initialise multiple copies of an operator to exploit parallelism, for example, multiple copies of the join operation in separate partitions that are to be evaluated in parallel. In the example query, for instance, the optimiser can choose between different source

databases, different EntropyAnalysis services, and different nodes for evaluating the partitions.

Apart from the distributed setting, a service-based DQP engine can offer better assurances of efficiency because dynamic service discovery and dynamic service creation and configuration allow it to take advantage of a constantly changing resource pool which would be troublesome for other approaches. Service registries based on UDDI [81] store details about the services and are also capable of storing service metadata, as done in Grimoires [103]. The service-based architecture of the DQP framework allows it to be extended to utilise the registry in order to discover instances of similar services, and also match the consumer requirements with the available metadata as will be described in Chapter 5. The Grid is composed of a dynamic pool of nodes which may be volatile in nature. Service-orientation can be used as a mechanism to deal with this inherent dynamism of the Grid by allowing loose-coupling with the resources, such as data and analysis services. Service orientation also opens up the possibility of exploiting dynamic deployment features (as discussed in Chapter 5) for better availability and performance by deploying services as and when required, on the best suited resources.

3.3 Setting Up a Distributed Query Service

The Grid Distributed Query Service (the GDQS or the *coordinator*) is the entity exposed by the OGSA-DQP framework to which the consumers interact. The coordinator encapsulates the query compilation/optimisation functionalities, creates the partitioned query plans required for evaluating a query and sends them to the evaluation nodes which are activated at run-time. This section describes the activities within the coordinator which lead to the successful evaluation of a query submitted by a consumer.

OGSA-DQP is implemented as an extension to the basic OGSA-DAI framework. The GDQS exposes a GDQS Data Service, which is an extension of the OGSA-DAI Data Service but encapsulates the DQP functionality. It exposes a GDQS Factory Data Resource which is accessed during the initialisation phase to create a GDQS Data Resource. Both these data resources are extensions of the OGSA-DAI data resource and in addition to the basic activities in OGSA-DAI, the GDQS Factory Data Resource contains a DQP Factory Activity, and the GDQS Data Resource contains an OQL Query Statement Activity, which are extensions of the core Activity[2] framework of OGSA-DAI. The DQP Factory Activity instantiates a GDQS Data Resource which then performs the initial

[2]An activity is a component within the OGSA-DAI software which provides a particular piece of functionality. For example, an activity is provided to perform an SQL query. A data service resource supports a particular set of activities.[100]

configuration of the distributed query processor engine, by obtaining the metadata from the OGSA-DAI data resources and the external web services. The resulting GDQS Data Resource is configured with the schema and computational metadata attached to it, and the user can then submit queries based on the database schemas.

The GDQS initialisation process is outlined in Figure 3.2. The client first interacts with the GDQS Data Service with a set of data and analysis resources to be used in the query (step 1 in the figure). This configuration document is passed to the GDQS Factory Data Resource implementing the DQP Factory Activity which then creates the GDQS Data Resource implementing the OQLQueryStatement Activity (step 2). In step 3, The newly created GDQS Data Resource interacts with the OGSA-DAI data resources and the analysis services, if any (specified in the initial configuration document) and obtains the database schema (from the OGSA-DAI data resources) and the WSDL documents (from the analysis services) (step 4). These are stored as the metadata for the GDQS Data Resource and the GDQS Data Resource identifier is returned to the client, for future reference during the actual querying stage (step 5). The GDQS Data Service supports creation of multiple GDQS Data Resources with unique identifiers to enable multiple query sessions by multiple users at the same time. These data resources can even be shared amongst multiple users and are capable of processing multiple queries, although any changes to the configuration (like addition or deletion of OGSA-DAI data resources or analysis services) would create a new GDQS Data Resource, ensuring that the queries submitted by other users to that GDQS Data Resource are not lost, and it can still be used with the original configuration.

The configuration document submitted by the client specifying the data and analysis resources to be used for the distributed query is based on an XML schema [104, 105] provided by the GDQS Data Service. The XML fragment in Listing 3.2 shows the canonical form in which the data and analysis resources are specified by the client and maintained by the GDQS.

After the related metadata are imported by the GDQS, the GDQS Data Resource is fully configured and the data resource identifier is returned to the client, the client can then submit queries (in OQL) over the set of databases and analysis services specified during the configuration stage. Figure 3.3 describes the steps involved in the query execution process. When a query is submitted to the GDQS Data Resource (step 1), the internal OQLQueryStatement Activity starts processing the query. The OQL query, along with the previously obtained metadata is passed to Polar* (step 2), the encapsulated query compiler/optimizer, which in the first pass, creates a logical query plan and then based on the computational resource metadata and the database metadata, creates an optimised partitioned query plan, scheduled to be executed in parallel on different nodes, which is returned to the GDQS Data Resource (step 3). Each partition is the query plan is then sent to

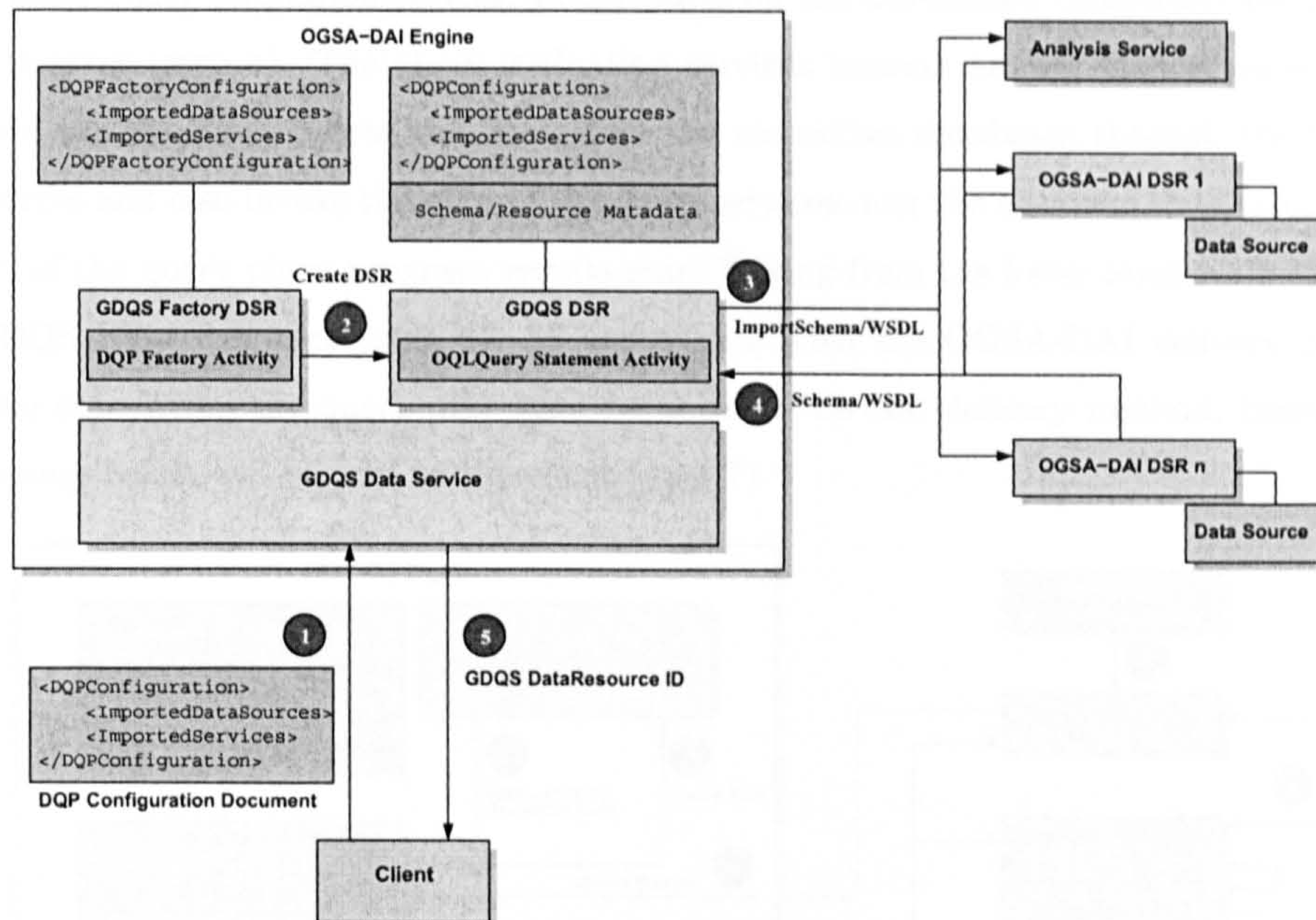


Figure 3.2: DQP Initialisation

Listing 3.2 Configuration Document

```
<DQPConfiguration xmlns="http://uk.org.ogsadai/dqp/configuration">
    <DQPEvaluatorList>
        <EvaluatorURI>http://giga01:8090/evaluator/services/QueryEvaluationService</EvaluatorURI>
        <EvaluatorURI>http://giga02:8090/evaluator/services/QueryEvaluationService</EvaluatorURI>
        <EvaluatorURI>http://giga03:8090/evaluator/services/QueryEvaluationService</EvaluatorURI>
        <EvaluatorURI>http://giga04:8090/evaluator/services/QueryEvaluationService</EvaluatorURI>
        <EvaluatorURI>http://giga05:8090/evaluator/services/QueryEvaluationService</EvaluatorURI>
        <EvaluatorURI>http://giga06:8090/evaluator/services/QueryEvaluationService</EvaluatorURI>
        <EvaluatorURI>http://giga07:8090/evaluator/services/QueryEvaluationService</EvaluatorURI>
        <EvaluatorURI>http://giga08:8090/evaluator/services/QueryEvaluationService</EvaluatorURI>
        <EvaluatorURI>http://giga09:8090/evaluator/services/QueryEvaluationService</EvaluatorURI>
    </DQPEvaluatorList>
    <DataResourceList>
        <ImportedDataSource>
            <URI>http://giga01:8090/wsrf/services/ogsadai/GoDataService</URI>
            <ResourceID>GoTermMySQLResource</ResourceID>
        </ImportedDataSource>
        <ImportedDataSource>
            <URI>http://giga02:8090/wsrf/services/ogsadai/InteractionDataService</URI>
            <ResourceID>InteractionMySQLResource</ResourceID>
        </ImportedDataSource>
        <ImportedDataSource>
            <URI>http://giga03:8090/wsrf/services/ogsadai/ProteinTermDataService</URI>
            <ResourceID>ProteinTermMySQLResource</ResourceID>
        </ImportedDataSource>
        <ImportedDataSource>
            <URI>http://giga04:8090/wsrf/services/ogsadai/ProteinPropertyDataService</URI>
            <ResourceID>ProteinPropertyMySQLResource</ResourceID>
        </ImportedDataSource>
        <ImportedDataSource>
            <URI>http://giga05:8090/wsrf/services/ogsadai/ProteinSequenceDataService</URI>
            <ResourceID>ProteinSequenceMySQLResource</ResourceID>
        </ImportedDataSource>
        <ImportedService name="EntropyAnalyserService"
        wsdlURL="http://giga09:8090/entropy-analyser/services/EntropyAnalyserService?wsdl"/>
    </DataResourceList>
</DQPConfiguration>
```

the designated query evaluation service, which exists on the designated computational resource, as a SOAP message (step 4). The query evaluation services interact among themselves sending data and control tuples. They access the data from the respective databases through the OGSA-DAI data resources and also invoke the external analysis services over the obtained data (step 5). As the evaluation of the query plans progress, results start flowing from the lower level evaluation services to the GDQS Data Resource (step 6). In accordance with the OGSA-DAI delivery options, the client while submitting the query may have requested a certain delivery method, based on which the final query result will be sent to the client (step 7).

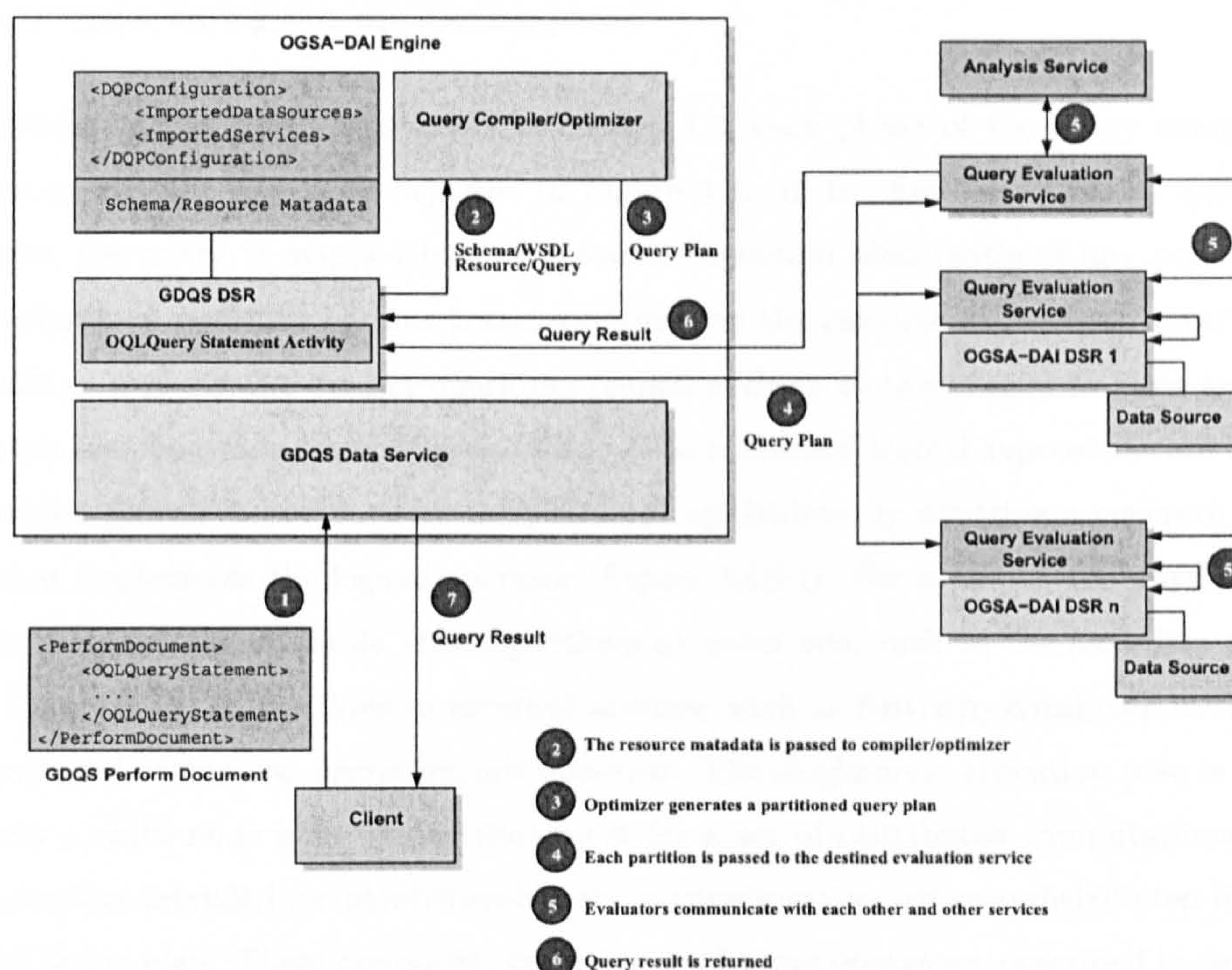


Figure 3.3: DQP Interaction

3.4 Distributed Query Plan Generation

The *query compiler* component inside the GDQS is responsible for generating efficient query execution plans for a declarative OQL query over a distributed set of services (both data and computational, as OQL supports invocation of external functions within a query). Assuming that the consumers are exempt of any charges on the usage of resources, it can be said that the most efficient execution plan is the one which is also the fastest in producing results. The compiler, Polar* [63, 13], developed prior to the OGSA-DQP related work, follows a popular two-phase approach

for optimisation referred in [106] for parallel and distributed databases. Using the Fegaras-Maier approach [107] based on a monoid calculus and algebra implemented in the OPTGEN optimiser generator [108], the compiler in the first phase produces a single-node execution plan by parsing the query and mapping it onto a logical and physical query algebra regardless of the number of execution nodes available to the system. In the second phase of the query compilation process, a partitioner subdivides the single-node execution plan into partitions which are assigned to the respective nodes by a scheduler. The computational resource metadata and the service metadata obtained by the GDQS Data Resource during the DQP initialisation process (described in Section 3.3) is used by the query compiler during this partitioning phase.

With reference to the query mentioned in Listing 3.1, each phase of the query compilation and query plan generation process is depicted in Figure 3.4. In the first phase of the query compilation process, the query is mapped to a single-node execution plan (without any partitions). The **logical optimiser** performs various transformations on the calculus expressions generated by the parser, such as pushing the project operators (called **reduce** in this thesis) as close as possible to the database scan operator (as in Figure 3.4(a)). The optimised logical expressions are transformed into physical algebraic expressions by the **physical optimiser** by selecting a concrete physical algorithm that implements the logical operator (Figure 3.4(b)). For example, the physical optimiser will choose between the available *join* algorithms to select one, such as the *hash-join* operator selected in Figure 3.4b. Invocations to external services, such as **EntropyAnalyser** in the example, are encapsulated within the *operation_call* operator. The single-node execution plan is then transformed into a multi-node plan by partitioning it for a set of distributed computational resources. Special operators intended for parallelization and communication between distributed instances are introduced in the plan. These operators, known as **exchange** operators, described in details in Section 3.5.4, encapsulate the control flow, data distribution and communication methods between the participating services. The **partitioner** at first tries to identify the *attribute-sensitive* and *location-sensitive* operators. The operators which require the input data to be partitioned by a specific attribute during execution on multiple nodes are the *attribute-sensitive* operators, for example, a *join* operator. Certain operators, known as the *location-sensitive* operators may only be executed on specific nodes, for example an *operation_call* operator which encapsulates a function requiring a special environment. As a rule of the thumb, *exchange* operators are placed directly above and below the *attribute-sensitive* and *location-sensitive* operators as they require transmission of data across the network. The multi-node execution plan in Figure 3.4(c) contains six partitions (shown in dotted-line boxes). The *exchange* operator signifies the intersection of partitions and is often immediately preceded by a *reduce* operator inserted in order to ensure that only the section of data required by the consumers of the parent *exchange* are included in the data-packets going across the

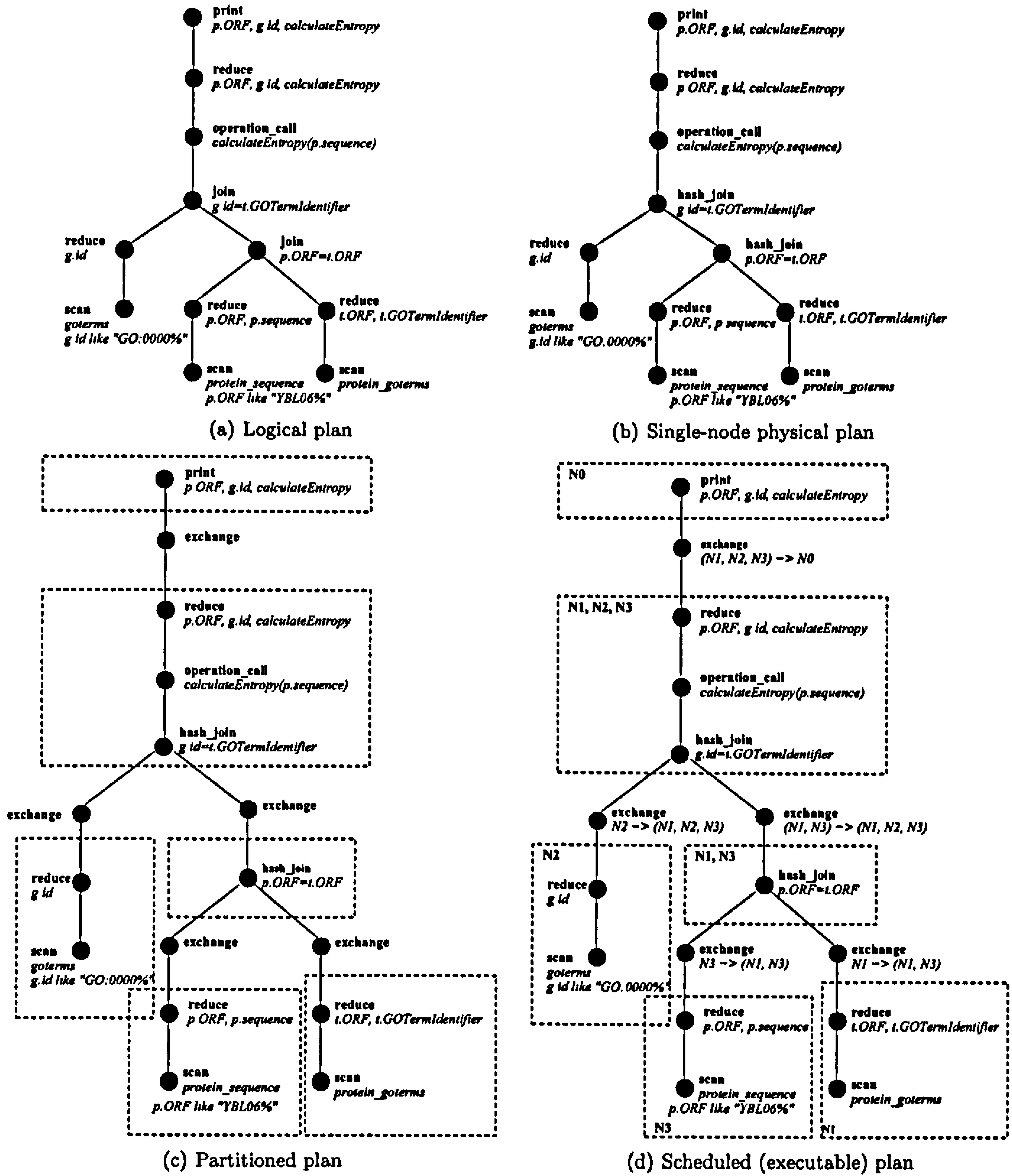


Figure 3.4: DQP query plan

network so that unnecessary network overheads can be avoided. In the final stage of query optimisation, the **scheduler** assigns execution nodes to each of the partitions created by the partitioner. The compiler supports parallelism between operators and hence a physical operator may be assigned to more than one execution nodes. The **scheduler** aims to assign a scan operator to an execution node hosting the relevant database extent (a database table is mapped to an extent), which reduces the communication overheads by avoiding large chunks of data moving between distributed nodes (for example nodes **N1**, **N2** and **N3** in Figure 3.4(d)). Invocations to external Web Services which are encapsulated within operators like *operation_call* tend to be expensive in nature. Such operators are assigned to multiple execution nodes to distribute the invocation cost. The system memory is taken into consideration for memory-intensive operators like *hash_join*. Considering all such scheduling policies, the final execution plan is generated where all the partitions are assigned to specific execution nodes (as in Figure 3.4(d)). It is to be noted that in this case, each execution node (N1, N2 and N3) are assigned with multiple partitions, and the actual executable partition for each node is created by combining together all the respective partitions assigned to that node.

The XML fragment in Listing 3.3 illustrates a partition containing a scan operator assigned to a particular node during a distributed query evaluation.

3.5 The Query Evaluation Service

The Query Evaluation Service (QES or the *evaluator*) forms the basis of the query processing engine which is created at run-time for the successful evaluation of a query submitted to OGSA-DQP. The evaluators implement the physical operators required to process a query and encapsulates all the complexities of query processing and distribution of tuples between nodes during the evaluation, and is considered as a major contribution within this thesis. This section describes the design and architecture of the evaluators.

3.5.1 The Overview

The **Query Evaluation Service (QES)** is implemented as a WS-I compliant *Web Service* [21], and is deployed on each execution node, the endpoint of which is known to the GDQS. Once the query is compiled and the partitions created, each partition is sent to the corresponding QES on the execution node as a SOAP message. On receiving the partition, the query evaluation process starts in each of the QES's. The QES supports multiple queries to be executed at the same time, and in order to

Listing 3.3 XML Partition Document

```
<Partitions xmlns="http://uk.org.ogsadai/dqp/partition">
  ...
  <Partition>
    <EvaluatorURI>http://giga10:8090/evaluator/services/QueryEvaluationService</EvaluatorURI>
    <GDQSResource>
      <CoordinatorURI>
      http://lovelace:8090/axis/services/ogsadai/DynamicDQPService
      </CoordinatorURI>
      <ResourceID>ogsadai-10f533dfa89</ResourceID>
      <InputStreamID>session-ogsadai-10f533dfa8a</InputStreamID>
    </GDQSResource>
    <QueryId>DynamicDQPService.ogsadai-10f533dfa89</QueryId>
    <Operator operatorFlagType="TABLE_SCAN" operatorID="0">
      <TABLE_SCAN>
        <predicateExpr>
          <simplePredicate>
            <comparativeOperator>LIKE</comparativeOperator>
            <leftOperand name=" goterms_goterm.id" type="13"/>
            <rightOperand name=" GO:0000%" type="16"/>
          </simplePredicate>
        </predicateExpr>
        <dataResourceName>goterms_goterm</dataResourceName>
        <dataResourceID>GoTermMySQLResource</dataResourceID>
        <GDSHandle>http://giga10:8090/axis/services/ogsadai/GoDataService</GDSHandle>
        <tableName>goterm</tableName>
      </TABLE_SCAN>
      <tupleType>
        <name>goterms_goterm.OID</name>
        <type>goterms_goterm</type>
        <name>goterms_goterm.id</name>
        <type>string</type>
        <name>goterms_goterm.type</name>
        <type>string</type>
        <name>goterms_goterm.name</name>
        <type>string</type>
      </tupleType>
    </Operator>
    <Operator operatorFlagType="APPLY" operatorID="1">
      <APPLY>
        <inputOperator>
          <OperatorID>0</OperatorID>
        </inputOperator>
        <applyOperationType>PROJECT</applyOperationType>
        <parameters>
          <attributeName>goterms_goterm.id</attributeName>
        </parameters>
      </APPLY>
      <tupleType>
        <name>goterms_goterm.id</name>
        <type>string</type>
      </tupleType>
    </Operator>
    ...
  </Partition>
  ...
</Partitions>
```

do this, on receiving each partition, the service spawns a new *query execution engine* as a separate process which parses the query plan, instantiates the physical algebra operators that implement the corresponding algorithms, and starts processing the partition, as illustrated in Figure 3.5.

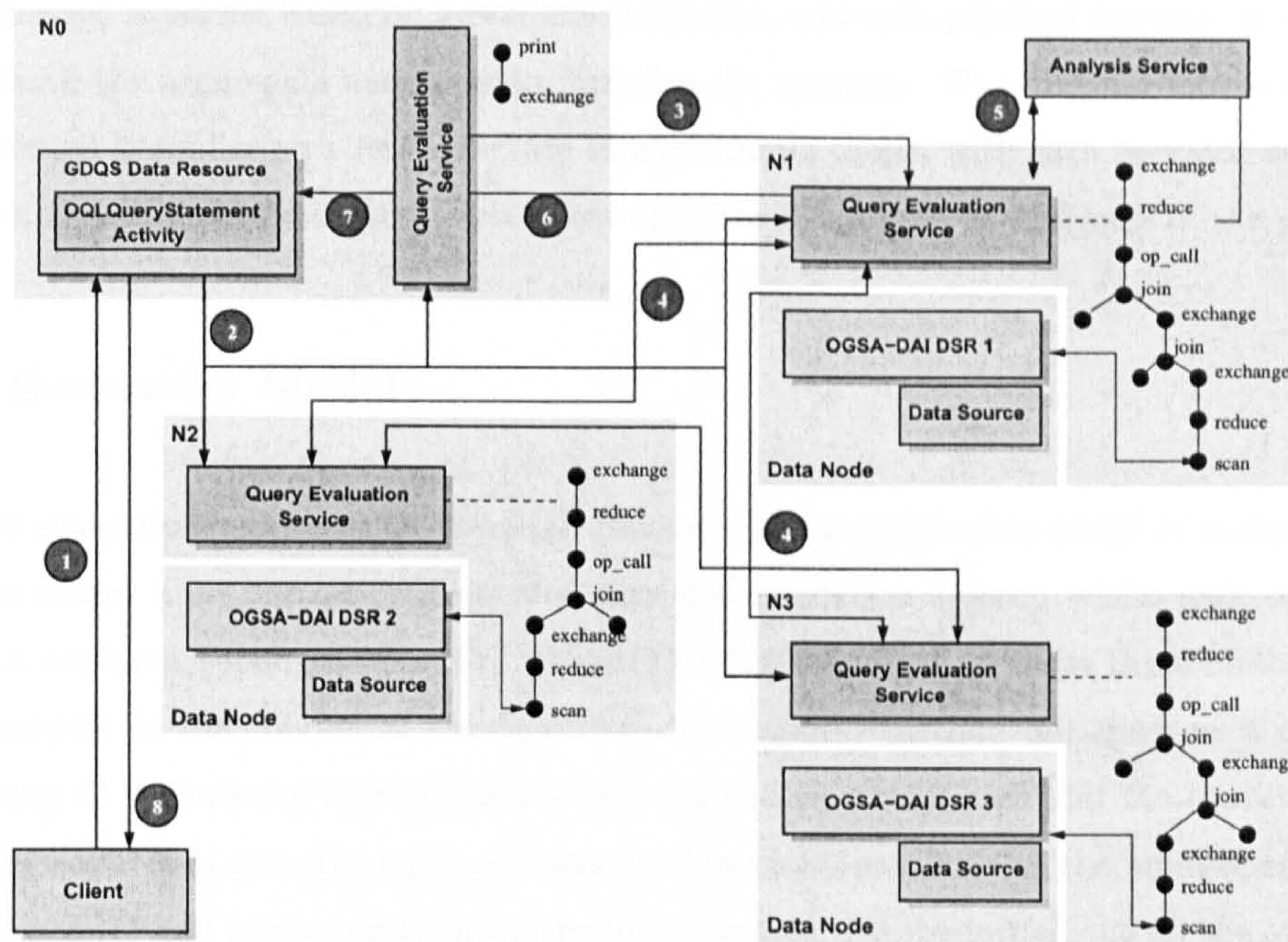


Figure 3.5: Query Execution on Component Services

The figure depicts the way in which the query (from Listing 3.1) is evaluated. In step 1, the consumer sends the query to the GDQS Data Resource (initialised earlier during the setup phase). The query is then compiled and the partitions are sent to each participating nodes (step 2). Each execution node (i.e. each QES) is shown in the figure with the partition it is supposed to evaluate. Based on the compilation rules outlined in Section 3.4, each scan operator which accesses a database is assigned to the nodes where the data exists, while the *operation_call* and *hash_join* operators are parallelised over multiple nodes. The execution starts at the root partition and propagates down through the child execution nodes (step 3). Each execution node evaluating the partitions communicate among themselves by sending control and data tuples (step 4); the control tuples responsible for sending/receiving signals and the data tuples contain the actual data. The analysis service is invoked when needed by the corresponding operator on the designated execution node (step 5), and results start flowing back to the root evaluator evaluating the root partition (step 6), finally back to the GDQS (step 7) and to the consumer (step 8).

The *Query Evaluation Service* on an execution node receives the query partition it is supposed to evaluate as an XML document embedded inside a SOAP message. Each partition document also

identifies itself using a unique identifier corresponding to the original query to allow concurrent evaluation of multiple queries. On receiving a partition document, the service creates a separate *Query Execution Engine* on a new execution thread, and starts it by passing the partition document. This XML document is parsed based on a predefined schema, and each physical operator is instantiated along with all the arguments necessary to initialise the operator. The internal representation of a query partition is similar to a *tree* structure in algorithmic terms, with each operator connected to its input operator, unless the operator is a leaf operator like a scan or exchange of the partition.

3.5.2 Evaluation Model

The query evaluation service in OGSA-DQP follows the classic *iterative model* of query evaluation [66]. This model is an implementation of a data flow execution system, where each operator implements a common {`open()`, `next()`, `close()`} interface. Each of these three methods serve a definite purpose in the context of the operator. The `open()` method initialises each operator by instantiating all the internal storage, data structures and variables, such that the operator becomes ready to consume data from the input operator, and invokes `open()` on all the input operators. This allows the `open()` call to propagate down the operator tree thereby initialising all the operators involved in the partition. The `next()` method in each operator collects single tuples from the input operators and processes them, and the `close()` method closes all connections, releases all memory allocations and clears all temporary variables. The general sequence of invocations in the iterative model is an `open()`, followed by a series of `next()` calls till the end of data is reached and then a `close()` call to complete the operation.

The query evaluation process starts with an `open()` call on the topmost (root) operator in the root partition, which propagates down the operator tree from parent to children until it reaches the leaf operators. The leaf operator can either be an exchange or a scan operator. In case of an exchange operator, the `open()` call is transmitted over the network to the remote producer which can be in another partition being evaluated on another execution node. In case of scan, this call initiates the database access via the OGSA-DAI Data Resource encapsulating the data source on that node. Once all operators are initialised, a series of `next()` calls again propagate down the operator tree from the topmost operator to the leaf operators. When the `next()` call reaches the scan operators, the already initialised OGSA-DAI Data Resource starts returning results from the database. The scan operators in OGSA-DQP use the `getNBlocks()` functionality of OGSA-DAI to return N blocks of data to minimise the service invocation overhead. These blocks of data are buffered inside the scan operator and one tuple at a time is returned to the parent of this scan operator during each invocation of `next()` thereby creating the upward flow of data. Considering the example query (in

Figure 3.1) and the corresponding multi-node execution plan (in Figure 3.4(d)), the evaluation starts at the root operator of the partition assigned to node N0 (*print*, in this example) with an `open()` call on it. This call propagates down to the leaf operator of the partition and then to the partitions at the next lower level until it reaches the final leaf operators at N1, N2 and N3, which are the scan operators, from where the upward data flow starts.

3.5.3 Data and Control Tuples

The data accessed from the database by the OGSA-DAI service interface are formatted as XML. This XML formatted data is translated into an internal tuple structure inside the *scan* operator. The end-of-data is also represented by a special EOF tuple. This processing is done once during the scan operation to avoid expensive XML parsing at each operator level. Throughout the entire query evaluation phase, data is transmitted between the evaluation services on various nodes in this intermediate tuple format. At each operator level, the structure of the outgoing tuple (that is, the data type of each attribute in the tuple) is known as it is passed within the partition document. The structure of the incoming tuple is also known from the input operator. Based on these structures, each individual attribute in the tuple can be accessed and processed. In order to signal the execution nodes about the processing, for example, signalling a child node to invoke `open()` on its root operator, or to signal the end of data, special tuples like control tuples and EOF tuples are also introduced. The tuple, described in Listing 3.4 is structured in such a way that the serialisation and de-serialisation facilities provided by the standard Apache Axis [77] web services framework can be utilised while transmitting tuples between distributed nodes over the network. While transmitting over the network, tuples are grouped together in a serializable structure with a query identifier added to the structure in order to preserve the context of the message at the destination.

3.5.4 Encapsulation of parallelism by Exchange operators

In the multi-node execution plans (in Figure 3.4(c) and (d)) and the overview diagram of the execution process (Figure 3.5), special *exchange* operators are introduced at the intersection of the executable partitions and above and below the attribute and location sensitive operators. The exchange operators, although implementing the same iterator interface, have different functionality in that they encapsulate the communication and data transfer between distributed nodes and introduce horizontal and vertical parallelism in the query execution plan.

Listing 3.4 Tuple Structures in XML

```
    <xs:complexType name="TransportTuple">
        <xs:sequence>
            <xs:choice>
                <xs:element name="DATA" type="msg:TransportDataTuple" minOccurs="0"/>
                <xs:element name="EOF" type="msg:TransportEOFTuple" minOccurs="0"/>
                <xs:element name="STATUS" type="msg:TransportStatusTuple" minOccurs="0"/>
                <xs:element name="CONTROL" type="msg:TransportControlTuple" minOccurs="0"/>
            </xs:choice>
        </xs:sequence>
        <xs:attribute name="size" type="xs:int" use="required"/>
        <xs:attribute name="transportTupleType" type="msg:TransportTupleType"/>
    </xs:complexType>
    <xs:complexType name="TransportDataTuple">
        <xs:sequence>
            <xs:element name="data" type="apachesoap:Vector" nillable="true"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="TransportEOFTuple">
        <xs:sequence>
            <xs:element name="EOF" type="xs:boolean" nillable="true"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="TransportStatusTuple">
        <xs:sequence>
            <xs:element name="OPENED" type="xs:boolean" nillable="true"/>
            <xs:element name="CLOSED" type="xs:boolean" nillable="true"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="TransportControlTuple">
<xs:sequence>
<xs:element name="OPEN" type="xs:boolean" nillable="true"/>
<xs:element name="CLOSE" type="xs:boolean" nillable="true"/>
</xs:sequence>
</xs:complexType>
    <xs:complexType name="ArrayOfTransportTuples">
        <xs:sequence>
            <xs:element name="context" type="xs:string"/>
            <xs:element name="destinationOperator" type="xs:string"/>
            <xs:element name="fields" type="msg:TransportTuple" nillable="true" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="TransportTupleType">
        <xs:restriction base="xs:NMTOKEN">
            <xs:enumeration value="DATA"/>
            <xs:enumeration value="EOF"/>
            <xs:enumeration value="STATUS"/>
        </xs:restriction>
    </xs:simpleType>
```

3.5.4.1 Horizontal and Vertical Parallelism

The Operator Model of Parallelism as explained in [24] are implemented by encapsulating both *vertical* and *horizontal* parallelism in the exchange operator. Horizontal parallelism requires redistribution of data between operators. Each exchange operator is parameterized with a distribution policy (also known as arbitration policy), such as round-robin, hash-distribution based on the result of a hash function applied on a specific attribute. The exchange operator distributes the data tuples based on this arbitration policy. Thus, multiple instances of the same operator can be executed in parallel on different subtrees on different hosts. This is typically done for expensive operators such as *join* and *operation-call* which are assigned to multiple partitions, and *exchange* operators use arbitration policies to distribute data to these instances.

Vertical parallelism is supported in an exchange operator by encapsulating a context-switching between the execution threads and using a shared data-structure between the two processes to synchronise and exchange data. The exchange operator implements the *producer-consumer* scenario by spawning a new thread in its `open()` method, after which, the parent thread operates as a consumer and the child thread operates as a producer. The producer thread then drives the subtree rooted in it, and based on the list of consumers and the arbitration policy, it decides the destination of each data tuple fetched by the `next()` call. The exchange operator in the consumer thread acts as a normal iterative operator, the only difference being in its `next()` implementation, where it receives the tuple via inter-process communication mechanisms from the shared data-structure, such as message queues, instead of the standard `next()` invocation on its input operator. The exchange operator on the producer thread on the other hand invokes the `next()` method on its input operator to fetch the next tuple. Figure 3.6(a) and Figure 3.6(b) (in the context of the example query) illustrate the functionality of the *exchange* operator.

3.5.4.2 Exchange operators in OGSA-DQP

In OGSA-DQP, a slightly different implementation of the *exchange* operator is used to cater to service-orientation and the possibility of a geographically distributed environment where partitions may be evaluated by services on physically different hosts. Thus, the initialisation process of these operators requires a different methodology. They can broadly be classified into three variants -

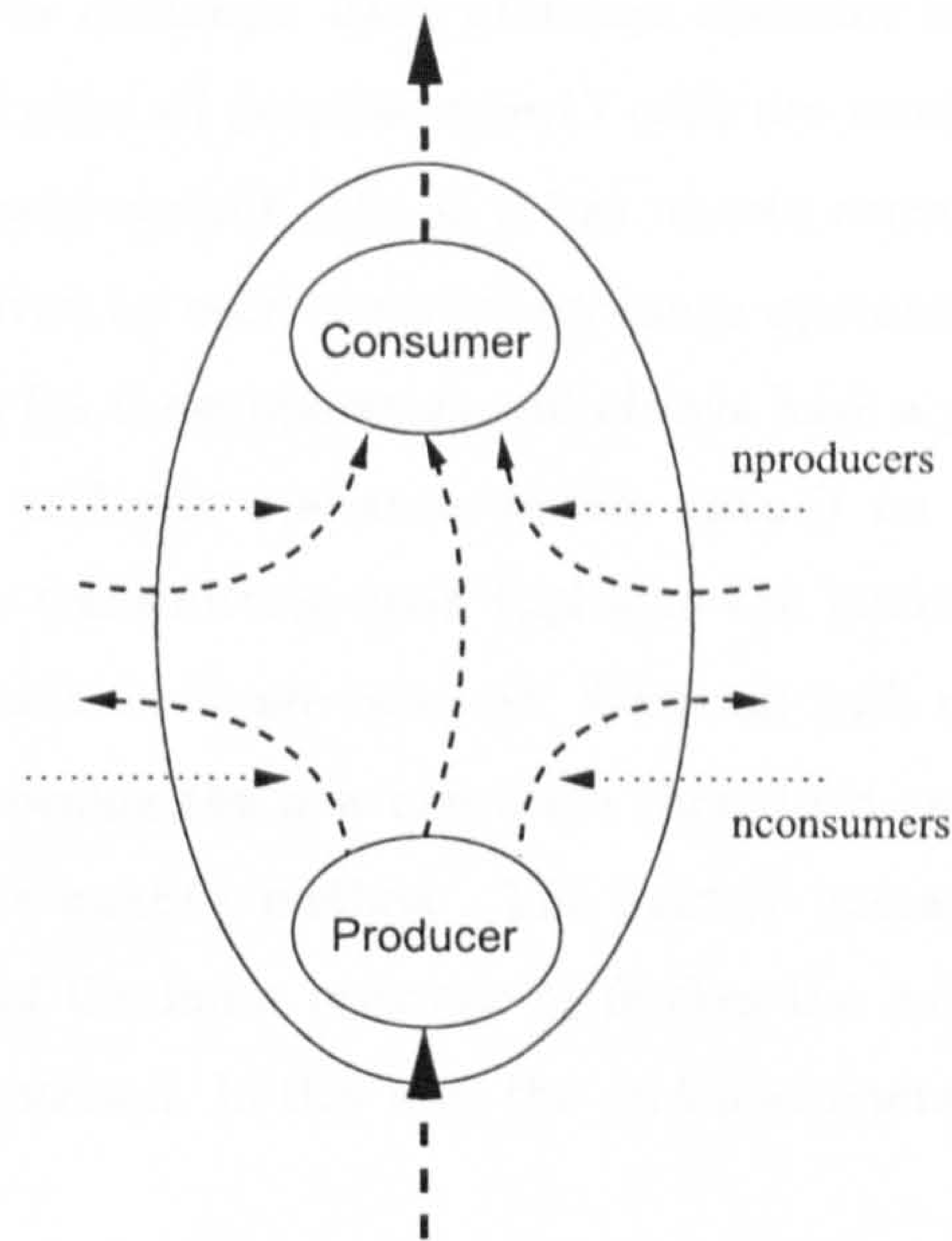
1. Leaf Exchange - *exchange* operators that are the leaf operators in a partition, for example **x0** in partition **N0**. These do not have any input operator, but may have remote producers assigned to other partitions.

2. Root Exchange - *exchange* operators that are the root operators in a partition, for example **x10** in partition **N1**. These have one input operator and possibly one or more remote consumers assigned to other partitions.

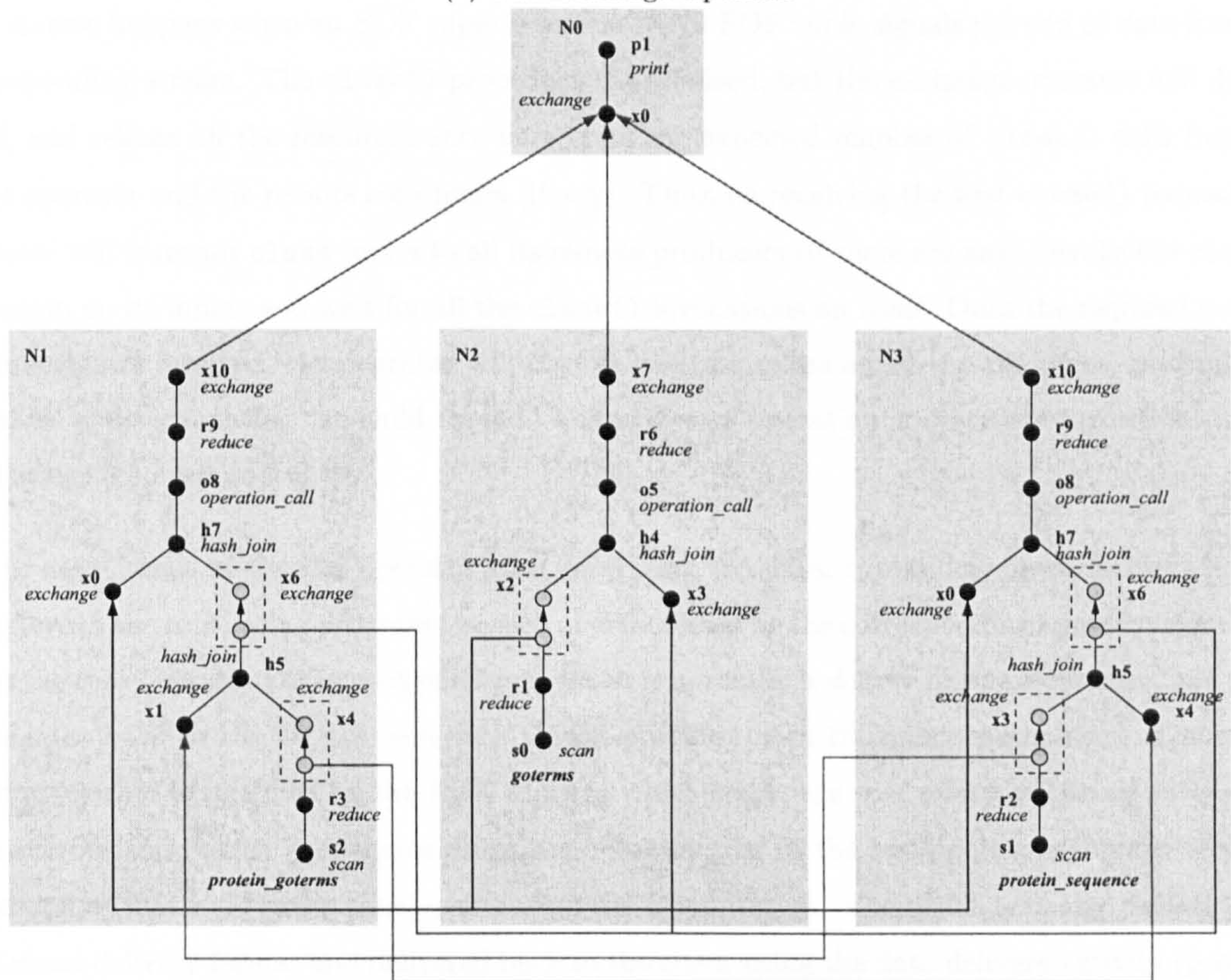
3. Intermediate Exchange - *exchange* operators that exist between other operators, for example **x2** in partition **N2**. These operators have one input operator and can have multiple remote and local producer and consumers[3].

The *leaf* and *root exchange* operators behave with minor variations from the *intermediate exchange* operators as far as the initialisation and the invocation of `next()` operation are concerned. When

[3]The producers and consumers mentioned in this context are essentially other *exchange* operators which can receive data from these exchange operators or send data to them.



(a) The Exchange Operator



(b) Communication between exchange operators on distributed partitions

Figure 3.6: Communication between distributed partitions

the `open()` call propagates down to the *leaf exchange* in a partition, an `open tuple` is sent to each of the remote producers of this *exchange.* Each *exchange* operator initialises itself (and spawns the consumer thread if required) once all possible `open()` calls are received by it. Each *root exchange* operator should receive as many `open()` calls as it has remote consumers. On the other hand, the number of `open()` calls received by each *non-root exchange* operator should be one more than the number of remote consumers (as these operators will always have a parent operator). Thus, during the initialisation phase, the *exchange* operators invoke `open()` on its input operator and all the remote producers (if any) by transmitting `open` tuples to the remote evaluation service, and wait until all the `open()` calls meant for it are received. Once all such calls are received, the operator progresses itself either by spawning the new execution thread (in case of `intermediate exchange` operators) or by invoking the `next()` method. The `next()` operation in turn either continually invokes the `next()` method of the input operator or fetches the next tuple from a message queue where incoming tuples are deposited. In this way, the *exchange* operator drives the operator subtree rooted at itself.

The reverse happens when an EOF tuple is received. An EOF tuple signals the end of data from the corresponding stream. The `close()` procedure is initialised, but the exchange operator will disable itself, and release all the resources once it receives the expected number of `close()` calls from the input operator and the remote consumers (if any). Thus, on receiving the first `close()` request, the operator will transmit `close` tuples to all its remote producers (if there are any), invoke the `close()` operation on its input, and wait for all the `close()` invocations on itself. Once the required number of requests are received, the operator will disable itself by releasing all the resources, resetting the variables and terminating the child thread. The *exchange* operation is described algorithmically in the listings 3.5, 3.6 and 3.7.

As the next() calls at the leaf operator start producing the data, tuples flow upwards from the leaf operators to the root, being processed at each operator level by the corresponding operator algorithm. From the root operator at the root partition, which is normally a *deliver* or *print* operator, the result propagates back to the GDQS using the GDQS-provided data transport mechanism. The end of the data stream is signalled by the EOF tuple at which stage, the root operator (*print*) invokes the `close()` method, which propagates down the operator tree to the leaf operators thereby releasing all resources that were being consumed during the computation. The result is finally packed in the predefined delivery format and delivered back to the client using the data delivery activity (provided by OGSA-DAI) specified by the client.

Listing 3.5 Exchange Operator - Open() method

```
1  public void open() {
2      open calls received = 1;
3      if (exchange has remote producers) {
4          for (i=0; i < number of remote producers; i++) {
5              invoke open on remote producer;
6          }
7      }
8      if (inputOperator != null) {
9          /* invoke open on the input operator */
10         inputOperator.open();
11     }
12     /* wait for all open invocations on this operator */
13     waitForOpen();
14 }
15
16 public synchronized void waitForOpen() {
17     while (open calls received < expected open calls) {
18         /* wait at the message queue */
19         Tuple tuple = tupleQueue.get();
20         if (tuple == OPEN) {
21             increment number of open calls received;
22         }
23     }
24     set status to opened;
25     enableExachange();
26 }
27
28 private synchronized void enableExchange() {
29     /* in case of intermediate exchange operators */
30     if (input operator != null) {
31         producerExchange = new Thread();
32         producerExchange.start();
33     } else {
34         Tuple result = inputOperator.next();
35         if (tuple == EOF) {
36             close();
37         }
38     }
39 }
```

Listing 3.6 Exchange Operator - Next() method

```
1  public Tuple next() {
2      if (producerExchange or root operator) {
3          Tuple tuple = inputOperator.next();
4          if (tuple == EOF) {
5              send EOF to all consumers;
6          }
7          while (tuple != EOF) {
8              send tuple to consumer identified by arbitration;
9              tuple = inputOperator.next();
10             if (tuple == EOF) {
11                 send EOF to all consumers;
12             }
13         }
14     } else if (consumer thread or leaf exchange) {
15         /* wait for tuple in the queue */
16         Tuple tuple = exchangeQueue.get();
17         if (tuple == EOF) {
18             wait for all EOF;
19             number of EOF = number of producers;
20         }
21         return tuple;
22     }
23 }
```

Listing 3.7 Exchange Operator - Close() method

```
1  public void close() {
2      close calls received = 1;
3      if (exchange has remote producers) {
4          for (i=0; i < number of remote producers; i++) {
5              invoke close on remote producer;
6          }
7      }
8      if (inputOperator != null) {
9          /* invoke close on the input operator */
10         inputOperator.close();
11     }
12     /* wait for all close invocations on this operator */
13     waitForClose();
14 }
15
16 public synchronized void waitForClose() {
17     while (close calls received < expected close calls) {
18         /* wait at the message queue */
19         Tuple tuple = tupleQueue.get();
20         if (tuple == CLOSE) {
21             increment number of close calls received;
22         }
23     }
24     set status to closed;
25     disableExachange();
26 }
27
28 private synchronized void disableExchange() {
29     /* in case of intermediate exchange operators */
30     if (producerExchange != null) {
31         terminate thread;
32     }
33     reset variables;
34     release resources;
35 }
```

3.5.5 Encapsulating Service State

Web Services are autonomous entities with explicit boundaries and the execution state of a service (or its child processes) should be encapsulated within the service. The services communicate by exchanging messages and often the requests and responses must be correlated with each other for meaningful operation. While processing a request, a service may receive a series of messages, each related to the same request, in which case, each message received by the service must be correlated to the original request and processed accordingly. Further, multiple services may be participating in processing a particular request, in which case, messages exchanged between all these services must be correlated in the same manner, which can otherwise be described as "associating a message with a specific conversational context" [109]. The Web Service Architecture document [109] describes this form of *message correlation* as:

> *"Message correlation allows a message to be associated with a particular purpose or context. In a conversation, it is important to be able to determine that an actual message*

that has been received is the expected message. Often this is implicit when conversations are relayed over stream-oriented message transports; but not all transports allow correlation to be established so implicitly."

This has been the design choice for the Query Evaluation Service.

Each Query Evaluation Service is capable of processing several queries simultaneously by evaluating multiple partitions in separate instances of the evaluation engines running in separate threads. Each evaluation process is dependent on receiving data from other participating services. As each instance of the evaluation service might be processing several queries at the same time, it may receive data for each of the evaluation processes from other services. Thus, the evaluation service is stateful in the sense that there is a "state" internal to the service, and there is an "interaction state" associated to each message exchange, where the service is responsible for identifying the destination thread for each packet of data received, and each recipient thread has to correlate the packets received with its execution state. In OGSA-DQP, the state is encapsulated within the service itself and a context is embedded within the messages that are transmitted. The service itself is implemented on the basis of the WS-I Basic Profile [21] as opposed to other *stateful web service architectures* such as the use of WS-Resource Framework [43, 110]. The service state (state of each evaluation process) is maintained using various data structures[4], and the context of each communication is passed within the message in form of an identifier. Each service receiving a set of data uses this identifier to locate the destination process for the data received. This implementation is similar to the proposals on using existing Web Services standards and technologies for Grid applications, such as in WS-GAF [27]. In this section, the architecture of the query evaluation service is discussed in details with particular focus on issues regarding the query execution states in order to underline the innovations and novelties. Figure 3.7 shows the components of the query evaluation service.

The Query Evaluation Service adheres to the principles of document-oriented services, and exposes a rich interface which is capable of receiving XML-structured query partitions from the GDQS and partial results in XML format from other evaluation services. The XML fragment in Listing 3.8 is a section from the WSDL description of the evaluation service which describes the evaluation service interface with the operations that are exposed and the data types that are expected for these operations. Three operations, namely *evaluate*, *sendData* and *sendMessage* form the evaluator service interface, each responsible for different activities. The coordinator sends a query partition to the evaluation service using the *evaluate* operation, which triggers the entire evaluate operation. The other two operations, such as *sendData* and *sendMessage* are used by the evaluators to communicate between themselves by exchanging tuples containing data and control messages.

[4]Note that the state can also be stored in databases, which is the primary approach in Transactional Grids.

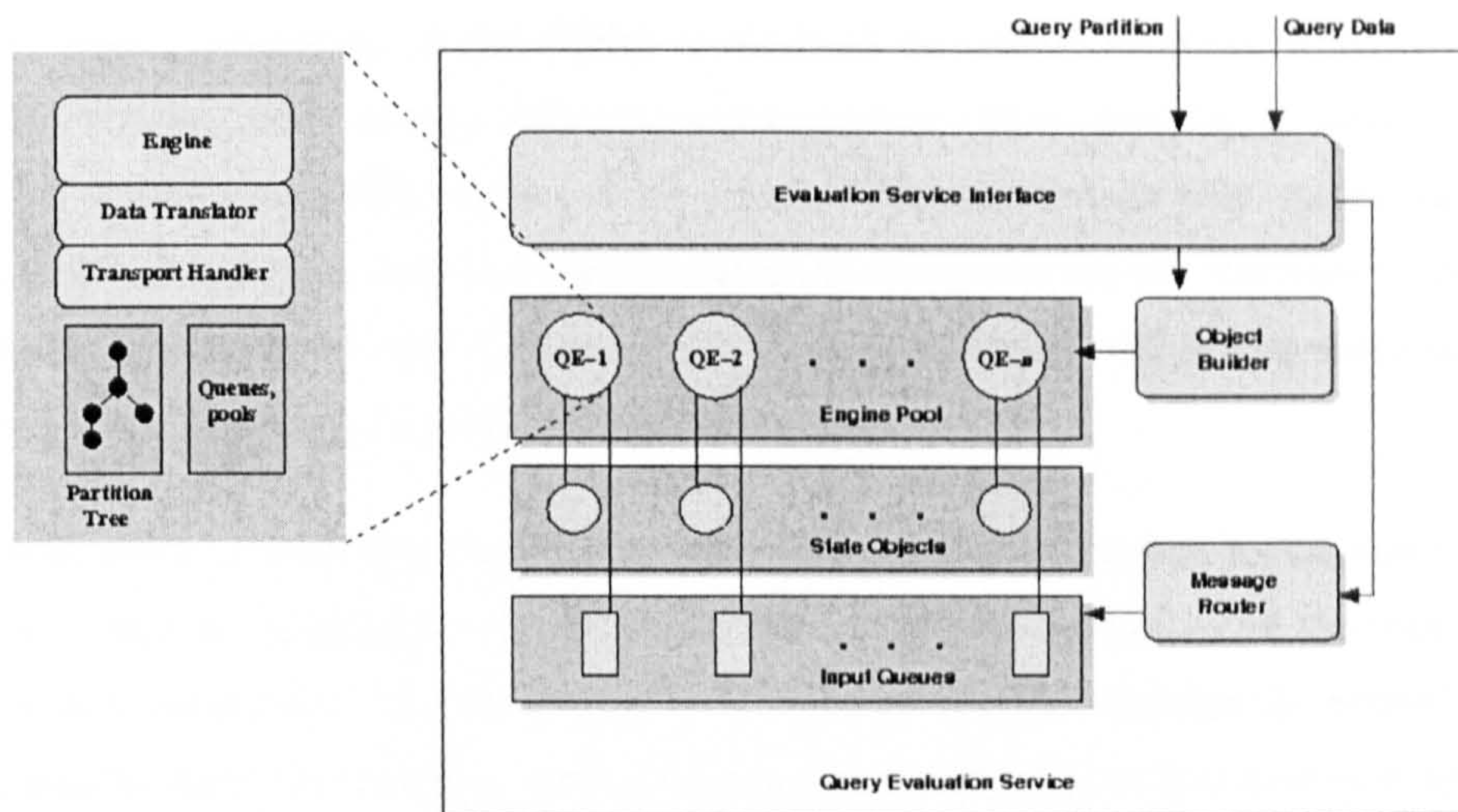


Figure 3.7: Architecture of the Query Evaluation Service

Listing 3.8 Description of the Query Evaluation Service interface

```
<portType name="QueryEvaluationPortType">
    <operation name="evaluate">
        <input message="tns:evaluateRequestMessage"/>
        <output message="tns:evaluateResponseMessage"/>
        <fault name="evaluationException" message="tns:evaluationExceptionMessage"/>
    </operation>
    <operation name="sendData">
        <input message="tns:sendDataRequestMessage"/>
        <output message="tns:sendDataResponseMessage"/>
        <fault name="sendDataException" message="tns:sendDataExceptionMessage"/>
    </operation>
    <operation name="sendMessage">
        <input message="tns:sendMessageRequestMessage"/>
        <output message="tns:sendMessageResponseMessage"/>
        <fault name="sendMessageException" message="tns:sendMessageExceptionMessage"/>
    </operation>
</portType>
```

The service is capable of handling multiple query partitions at the same time. Each time an XML-formatted query partition is sent to the evaluation service, a new *evaluation engine* is initialised with various components such as *Data Translators* which are responsible for translating semi-structured partial results received from other evaluation services participating in the same query into internal tuple format and *Transport Handlers* which are used by the *exchange operators* while sending data to remote evaluation services and the *print* operator while returning the results back to the GDQS. A shared *Object Builder* component internally translates the XML-formatted partition into a tree of physical operators that are responsible for processing each tuple. Each of these operators are objects implementing the iterator interface, and encapsulate the physical algebra within their `open()`, `next()` and `close()` operations. Each *operator tree* is associated with the evaluation engine responsible for the actual processing of the partition. Each engine maintains a collection of data structures for internal operation such as storing the process states and receiving messages from remote services.

Each query that is submitted at the GDQS is assigned an unique identifier which is propagated within the query partition sent to each evaluation service. This identifier is used to keep track of the states of each processing engine. Every data or control message that are exchanged in the system contains this unique identifier which enables the recipient service interface to correlate the request being processed with the messages received and to identify the actual destination process and forward the message to the proper destination queue.

Once the evaluation of a partition is finished, the corresponding evaluation engine would wait for its termination, which is triggered by the invocation of the `close()` operation on the root operator of the associated operator tree. The *print* operator in the root partition invokes the `close()` operation which propagates down the operator tree closing each operator. At the *leaf exchange operator* on a partition boundary, a control message is sent to the evaluation service upstream, where the `close()` method on the root operator is triggered. This chain of events continues until all operators are closed and all resources consumed by the operators are released. Finally, each query execution engine is terminated along with the associated objects such as the *Data Translator* and *Transport Handler.*

3.6 Discussion

This chapter introduced the OGSA-DQP framework which is based on the concepts of service oriented architectures and is capable of processing queries over distributed databases. The research was carried out as a collaboration between Manchester and Newcastle Universities and thus this thesis does not claim the sole credit for the research. The work done in defining the overall architecture of the system is considered as one contribution towards this thesis. The major contribution lies in the design, implementation and evaluation of the run time query evaluation engine, which implements the physical query algebra and encapsulates all the complexities of routing and distribution of messages containing tuples for effective processing of a query.

The OGSA-DQP framework described in this chapter is service-based in two orthogonal senses -

- OGSA-DQP allows virtualization of resources in the sense that it supports queries over distributed data storage and analysis services which are factored out as services.
- The process of generating the distributed query plan, as well as evaluating the query over resources available on the Grid, are factored out as services.

OGSA-DQP uses the existing OGSA-DAI services for accessing data thereby allowing distributed

querying in a homogeneous way over heterogeneous data sources. In essence, OGSA-DQP follows a *wrapper-mediator* approach where the DQP framework acts as a mediator over OGSA-DAI wrappers over the data sources. Previous work on Polar* has been successfully incorporated within the GDQS component of DQP which allows the parallel execution of the query partitions on multiple hosts. The GDQS has been implemented as an extension to the existing OGSA-DAI services.

However, the evaluation service has been designed and implemented with a different approach to the OGSA-DAI based architecture of the coordinator. It is designed as a WS-I Web Service using already existing standards and toolkits thereby making it completely inter-operable. The service state is encapsulated within the service, and messages are correlated using a context embedded within each message, which follows the approach recommended by the Web Services Architecture document [109]. The evaluation service creates the infrastructure for evaluating a query partition by implementing the physical query algebra based on the classic iterator model of query evaluation. It also encapsulates the data distribution between operators evaluating different partitions of the same query within a special *exchange* operator. The query evaluation service is also capable of evaluating multiple partitions simultaneously within separate processes that are isolated from one another, and special attention is given to message correlation within such a setting where a service is likely to receive multiple messages from several remote sites for each query being processed. The evaluation services are thus scalable in terms of the number of queries that can be processed simultaneously.

One concern, however, exists regarding the transmission of data packets between participating nodes. The data tuples are serialised using the Apache Axis libraries and are transmitted across the network as SOAP messages. This is not an efficient way of transmitting packets of large size because the cost of serialisation and de-serialisation tends to increase for larger sized packets and such packets also incur a higher transmission cost. Recently, there has been advancements in sending binary data within a SOAP message using new specifications and standards, and it is possible to adopt these mechanisms within OGSA-DQP, which should reduce the transmission overhead dramatically.

Chapter 4

Dynamic Service Deployment

This chapter will introduce a framework which allows dynamic demand-driven deployment of services on available computational resources. The traditional grid computing concept has a distributed job scheduling system at its core where jobs can be scheduled dynamically, and conventionally, a grid has been synonymous to a computing infrastructure supporting systems such as Condor [3, 2], Globus [4] etc. The DynaSOAr (Dynamic Service Oriented Architecture) framework introduced in this chapter proposes an alternative approach to Grid computing where the distributed applications are built around services.

4.1 Distributed Job Scheduling

Most Grid computing infrastructures like Condor [3, 2], Globus [4] or Sun Grid Engine [111] utilise some form of Distributed Job Scheduling for routing consumer jobs to remote computational resources. A job, which is a combination of the executable code and in most cases the data on which the code will operate, is created by the consumer, and submitted to the job scheduling system. The scheduler routes the job to an available host suitable for executing it, and once the job is completed on that host, the consumer is notified of the result. Condor uses a matchmaking approach to match the requirements of the consumer request with the characteristics of the available resources in order to find out a suitable target for executing the job. In Condor, the consumer submits a job to an *agent*. The agent stores the job in persistent storage and searches for resources that are suitable for executing the job. Agents and the resources advertise their characteristics and policies to a *matchmaker*, which introduces potentially compatible agents and resources. After this matchmak-

ing phase, the agent establishes contact with the resource and verifies the compatibility. Separate processes are started on both sites (the agent and the matching resource) for executing the job. A *shadow* process on the agent side provides all the relevant details required to execute the job, and a *sandbox* process on the resource creates a safe environment for execution for the job protecting the resource from any malicious interference. The job with the execution code and input data is passed to the sandbox at the resource for execution. Although an agent and a resource are logically separate entities, they can reside on a single physical host.

It is to be noted that the common approach for executing a job in Condor is to provide the execution code and the inputs (if required) as well. A consumer request may involve retrieving large amounts of data from a database. Unless the host which stores the data is explicitly specified as the target machine within the job, it might require the data to travel over the network, and as Condor does not have the knowledge about this data access, it will not make any attempt to schedule the job at a host closer to the data. Further, this approach of using jobs, is an *one-time* affair, as the execution code for the job is not stored at the execution site. For each request for the same job, with different data as input, the execution code and data would have to be transferred to the execution site for redeployment and execution. The redeployment of the execution code for each execution request may become costlier when the execution code itself is large, such as virtual machine images. Hence, for requests which process large amounts of data, and are frequent in nature, or for large analysis code, an alternative approach with a "deploy once, use many times" characteristics may be more suited. It may be possible to retain the execution code at the execution site by extending Condor. But, for meaningful use, additional capabilities such as the ability to discover an execution code for future use, or the possibility of multiple deployments to share the invocations, or the standard interface for invocation etc. will be required.

4.2 The Evolution of DynaSOAr

This section discusses the motivation behind the development of DynaSOAr, the conceptual background, the requirements to be satisfied by DynaSOAr and the design issues encountered.

4.2.1 Jobs and Service Orientation

In recent years, there has been a considerable shift towards the use of the *Service Oriented Architecture* and technologies for building Grid and other distributed applications. In a service-oriented

framework, an application can be a combined set of autonomous services which communicate between themselves by exchanging SOAP [5] messages. The service interfaces are described by a standardised language, WSDL [101]. When the computational requirement of these services can not be satisfied by the environments in which these are hosted, the current strategy is to create a job and send it to a distributed scheduling system (like Condor) for execution on a suitable host. This forces the application developers to deal with two different types of computational entities - services and jobs.

This chapter describes an alternative approach which builds on the concept of services only. Dynamic Service Oriented Architecture (DynaSOAr) [19] is a framework for deploying Web Services on demand over computational resources available over a Grid or the Internet. DynaSOAr advocates an approach to Grid Computing where distributed applications revolve around the concept of services rather than jobs. When a consumer makes a request for a service to the Service Provider, the request is serviced by a host most suited for the requirements specified by the consumer. If no existing deployment exists or if performance requirements cannot be met by existing deployments, this framework automatically deploys the service on an available host. In essence, this is analogous to remote job scheduling, but offers the opportunity for improved efficiency in the long run as the cost of moving and deploying the service can be shared across the processing of many messages over the time. Further, the philosophy behind DynaSOAr is "deploy once, use many times", which is contrary to the conventional job-oriented systems, where the execution of a job is an one-time affair.

The key architectural feature in DynaSOAr is the logical separation between *service provisioning* and *resource provisioning* and clear distinction of the responsibilities of the components. The *Web Service Provider* makes services available to the consumers by exposing an endpoint to the service, and *Host Providers* offer computational resources on which the services can be deployed and messages processed. These components are supported by others such as the *Service Repository* which stores the deployable versions of the services, *Brokers*, who decide to which set of Host Providers a message should be routed, and *Registry*, which stores description of the services and the associated metadata. All these components taken together create a framework built over loosely-coupled interactive Web Services. The approach provides three potential benefits over existing approaches that utilise both jobs and services -

1. The development process is simplified as it is confined to the service-oriented architecture alone;
2. There is a possibility of improved performance as the deployed service is retained on the host. This allows the service to be used for as long as required, thereby spreading the cost of

deployment over many invocations of the service;

3. The clear distinction between Service Provider and Host Providers allow new organizational/business models.

4.2.2 The Active Information Repository

The *Active Information Repository* architecture proposed in [7] is aimed at collocating the data processing and analysis code with the data by providing a cluster of computational resources closer to the data. The proposal envisages that information repositories will be made available for the Grid in order to cope with the astronomical volumes of data produced by current research projects such as the high energy physics experiments at European Organization for Particle Physics (CERN) [56] or the SLOAN Digital Sky Survey [23] project. The repositories will be even more valuable if along with the data storage, some computational power is provided for the users to perform analysis on the data retrieved from these data repositories on the same site. This is to avoid transmitting large amounts of data over long distances, potentially reducing the cost of data transmission and thereby increasing the efficiency. The major components of this Active Information Repository are a scalable object database server with a scalable agent-execution server connected to it via a high-bandwidth network. The object database server will hold persistent data on a cluster of nodes, each of which will contribute to the storage and computational capacity of the complete system. Computations to be performed on the data will be sent, in the form of mobile agents for execution on the agent-server, which again can use a set of nodes to increase scalability.

The work on DynaSOAr, although not using agent systems, was inspired by the concept of moving the computation closer to the data to increase performance as proposed in the Active Information Repository architecture. A service-oriented version of the Active Information Repository is shown in Figure 4.1 where the system (a cluster of one or more computational nodes) that hosts the data and metadata also hosts services operating on that dataset, and also allows the consumers to deploy and share their own services for analysing the data [112].

4.2.3 The Consumer View of a Service

From the point of view of the consumer, depicted in Figure 4.2, a Web Service invocation is necessarily an interaction with the service instigated by sending a SOAP message. Various libraries (such as Apache Axis [77]) convert an application level invocation into a correct SOAP message format which is then delivered to the service via a transport mechanism (such as HTTP) specified within

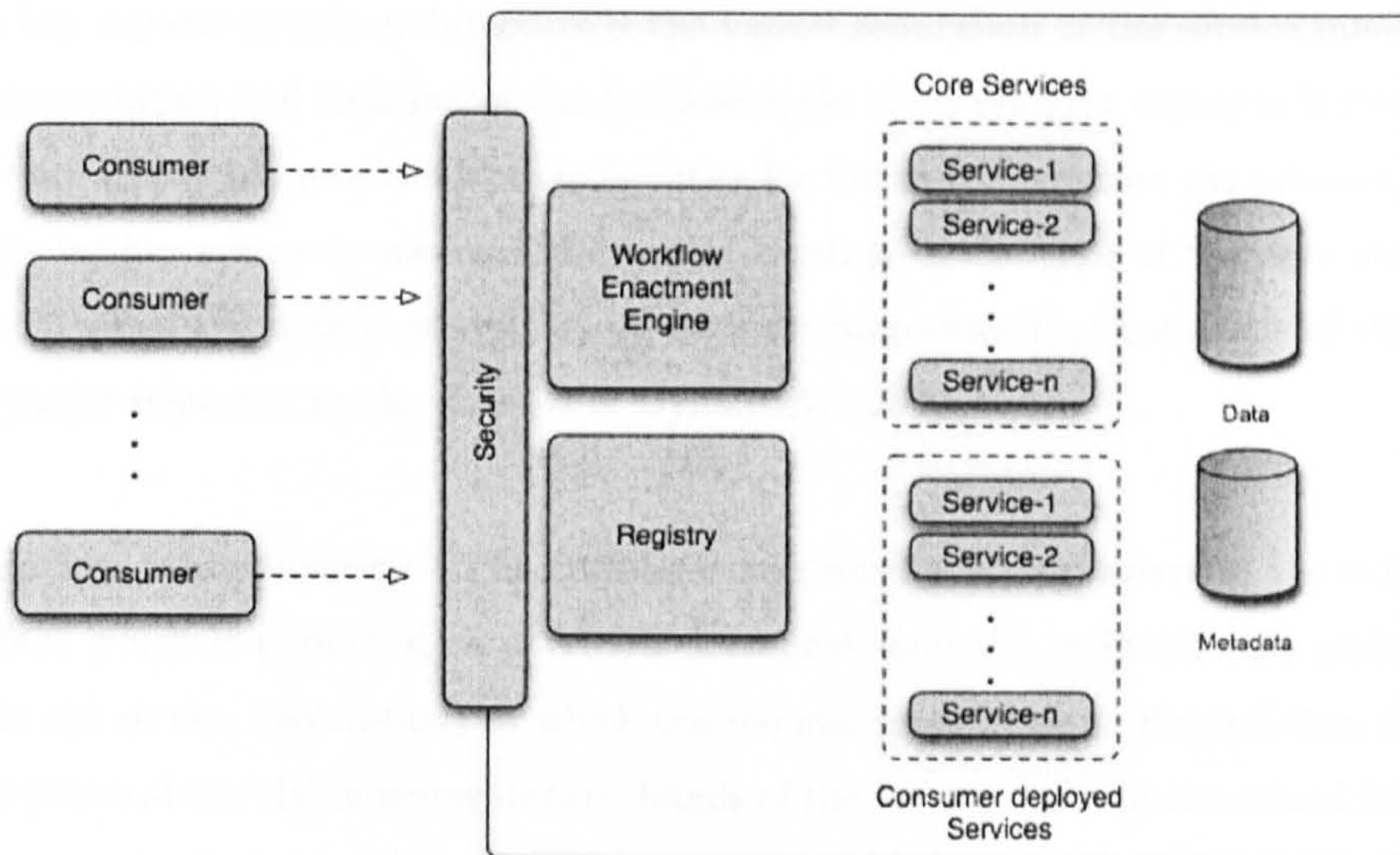


Figure 4.1: The Active Information Repository Architecture

the service interface. For successful processing, the message must conform to the types defined in the service interface description (WSDL [101]), which is checked at the service provider endpoint. The message may then be transformed into objects and structures that are internal to the service and processed by the encapsulated service logic. Any generated response is sent back using the same transport mechanism. The business logic and the implementation of the service remains transparent to the consumer who is never aware of the internal processing that is performed behind the service interface.

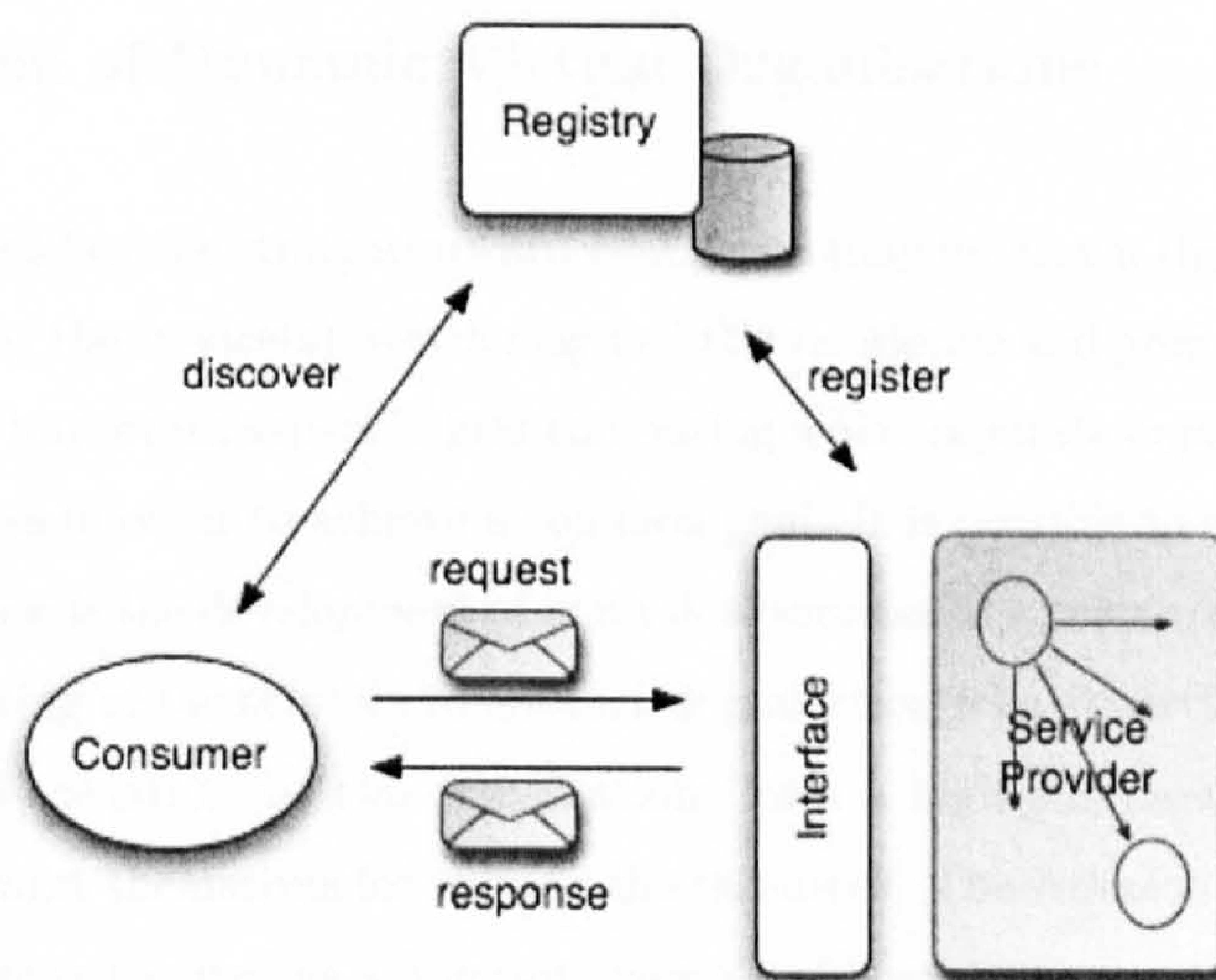


Figure 4.2: Invocation of a Web Service

One of the key aspects in this architecture is the logical separation of the service interface and the actual implementation and location of the business logic. One relevant example is the Amazon S3 Storage Service [52] which provides storage facilities for the consumers over the internet. Consumers who use this facility are never aware of the actual location or the type of the data storage device, or the way in which the data is stored. Thus, two key properties that come out of this separation and are of major relevance to the concept of DynaSOAr are -

- **Execution transparency** - The consumer of a service is only aware of the exposed service interface which is published as a WSDL document, and the endpoint (the address specified within the service description) to which the request must be sent. Beyond this, the consumer is not aware about the implementation details of the service logic, or the actual host where the logic is executed, or any other associated entities, which leads to the "execution transparency".
- **Loose Coupling** - As the consumer is only aware of the interface exposed by the service there is minimal coupling between the consumer and the service. There is no dependency at the consumer side on the platform on which the service is executed or the language in which the service has been implemented. The current toolkits also allow the implementation of a service to be completely replaced without changing the interface.

The concept of dynamic service deployment in DynaSOAr is based on these two properties of service oriented architectures.

4.2.4 Formation of Dynamic Virtual Organisations

From the discussion so far, the straightforward conclusion may be drawn that the service provider is responsible for hosting the service(s), which may be little restrictive and premature in nature. Virtual Organisations are an important aspect in grid computing where separate organisations collaborate for sharing their resources in order to achieve a common goal. It is possible to envisage an organisation whose area of expertise is the development of analytical services in a certain domain who might want to outsource the hosting of the services to another organisation who expertise in providing compute resources. In such a scenario, the two organisations form a highly dynamic Virtual Organization and collaborate amongst themselves for sharing the resources. The Amazon Elastic Compute Cloud (Amazon EC2 [88]) may be sited as a relevant example of hosting services.

DynaSOAr creates the possibility of forming such dynamic Virtual Organisations by differentiating between service provisioning and resource provisioning. To support such on-demand resource pro-

visioning DynaSOAr introduces the concept of a Host Provider which is responsible for providing the computational resources, such as the Amazon Elastic Compute Cloud mentioned before. DynaSOAr does not impose any coupling between the service provider and the host provider other than a mutually agreeable contract in form of message patterns which cause a "handshake" between both the parties. The interface to the service (as seen by the consumer) does not change, neither does the assumption of the consumer about the existence of the service at the service provider's site. The existence of the Host Provider is hidden behind the Service Provider maintaining the two key aspects discussed above, viz., *execution transparency* and *loose coupling*. The introduction of this new level is shown in Figure 4.3.

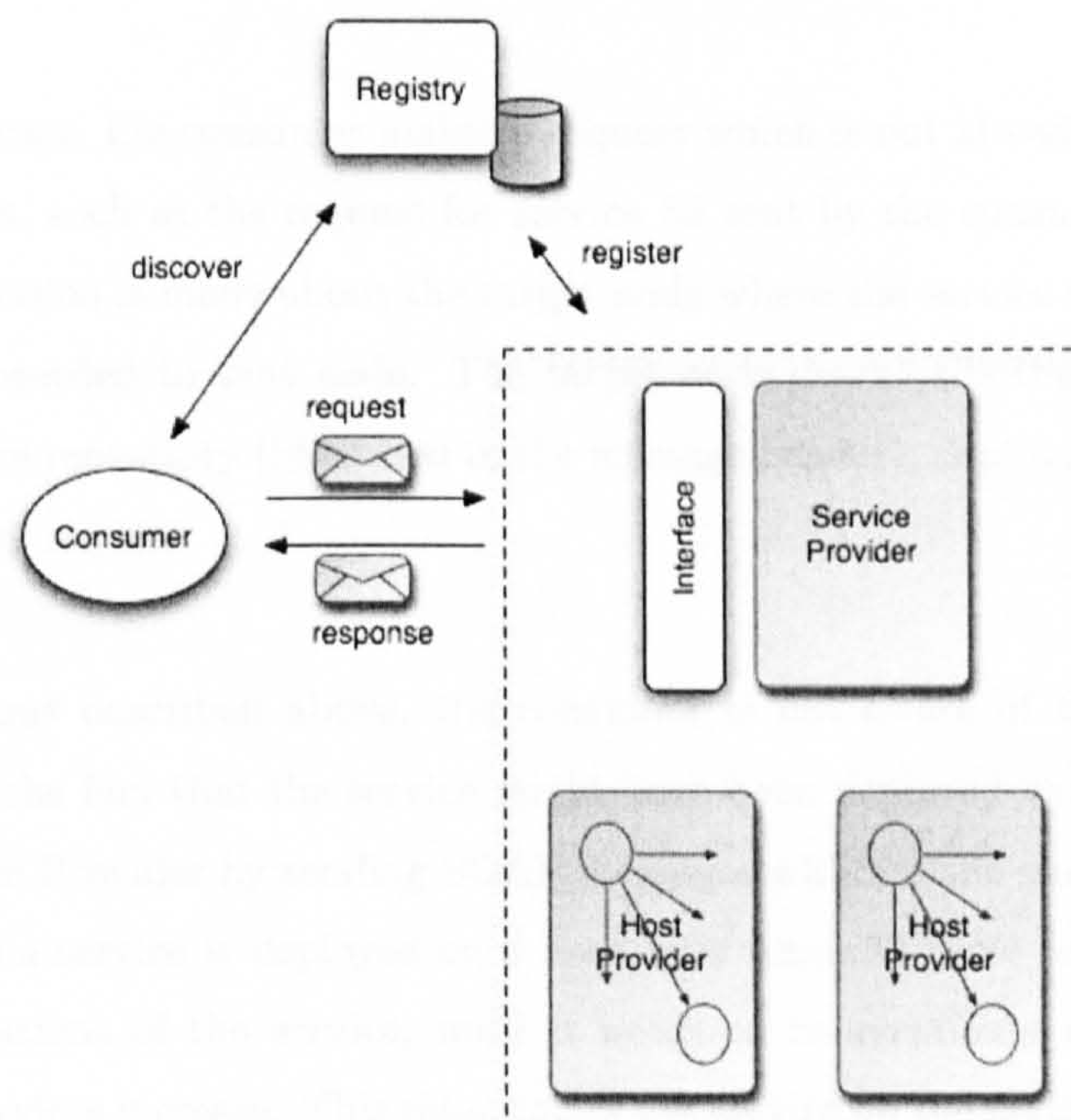


Figure 4.3: Formation of a Virtual Organization between the Service Provider and the Host Provider

4.2.5 Principles of Dynamic Deployment

DynaSOAr conceives the idea of deploying a service when there is a demand for it. All services registered in a registry are advertised by the service provider as services that can be provided, whether or not they are deployed. A consumer is able to request for any of those services, and the decision for deployment is made when a request is received for a particular service. This policy allows the framework to host only the services that are being used, which effectively is a cleaner approach, and secondly, once a service is deployed, it remains available for all the consumers to use,

unless explicitly undeployed. On receipt of a request for a service, the service provider forwards the request to a suitable host provider, and there may be two different interaction patters depending upon whether or not the requested service is already deployed on the node -

1. If the service is already deployed on the computational node where the request has been sent for processing, then the SOAP message is routed to the destination service by the Host Provider. In Figure 4.4(a), the consumer makes a request for service S2, which is already deployed on nodes N1 and N2. Based on the current information about the system load, the Host Provider routes the requested to the lightly-loaded N2 where the request is processed and the response sent back.

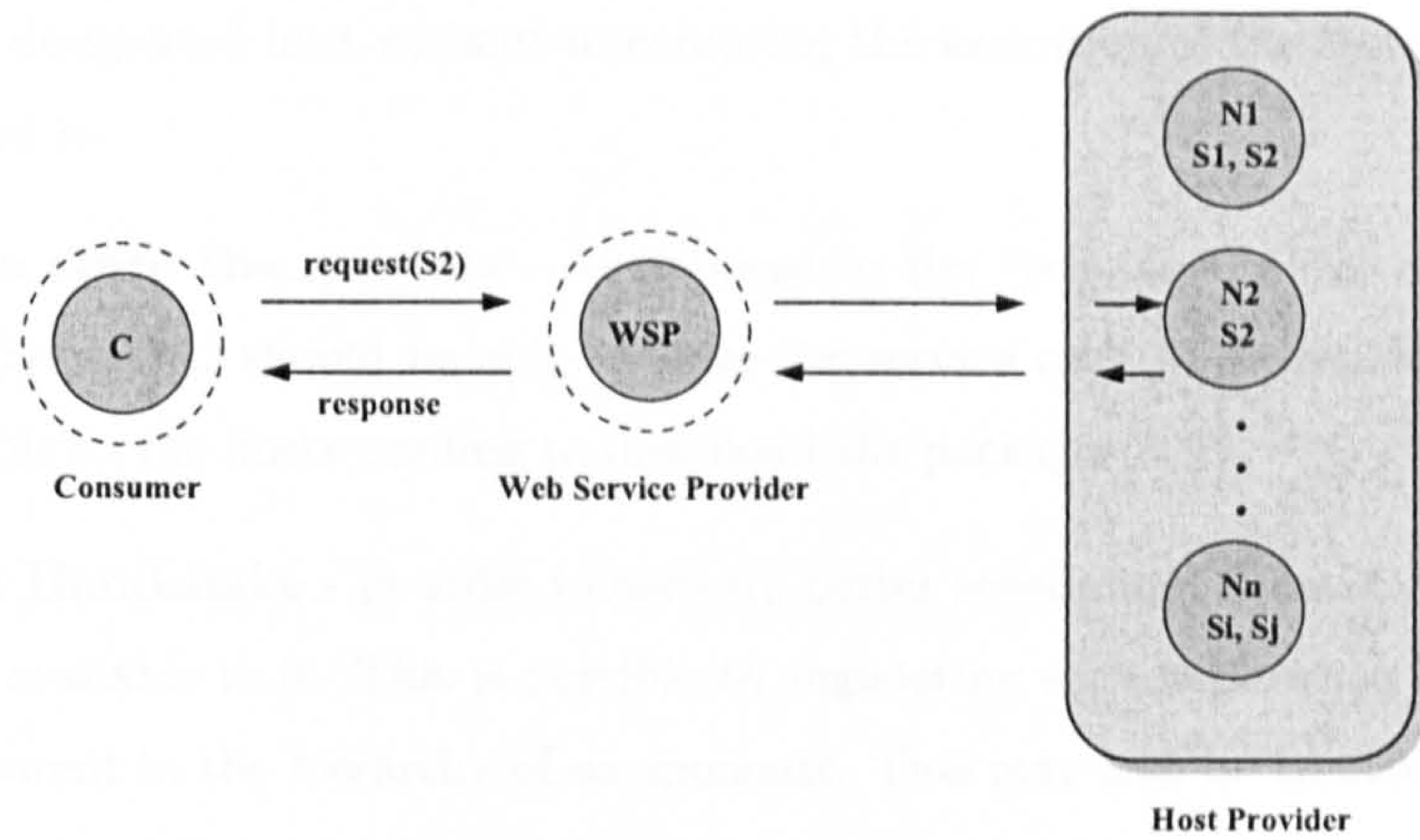
2. In the second case, the consumer makes a request which is not already deployed on any of the available nodes, such as the request for service S8 sent by the consumer in Figure 4.4(b). In this case, a decision is made about the target node where the service is to be deployed and the message is forwarded to that node. The target node downloads the deployable service code from the service repository (identified in the message header), deploys the service dynamically, and process the request.

In both the situations described above, the consumer is not aware of the resources behind the Service Provider or the fact that the service might have been deployed dynamically. They interact only with the Service Provider by sending SOAP messages which is the standard way of interacting with services. Once a service is deployed on a host, it is retained there ready to process messages for all future invocations of the service, until it needs to be reclaimed which is likely when the demand for other services increase. This retention of the service on the node can potentially generate large efficiency gains because the one-time cost of deployment is spread over many invocations of the service. GridSHED [113, 114], a related project has developed heuristics-based algorithms for determining the optimal policies which decide when an existing deployment should be used compared to a new deployment on a new node.

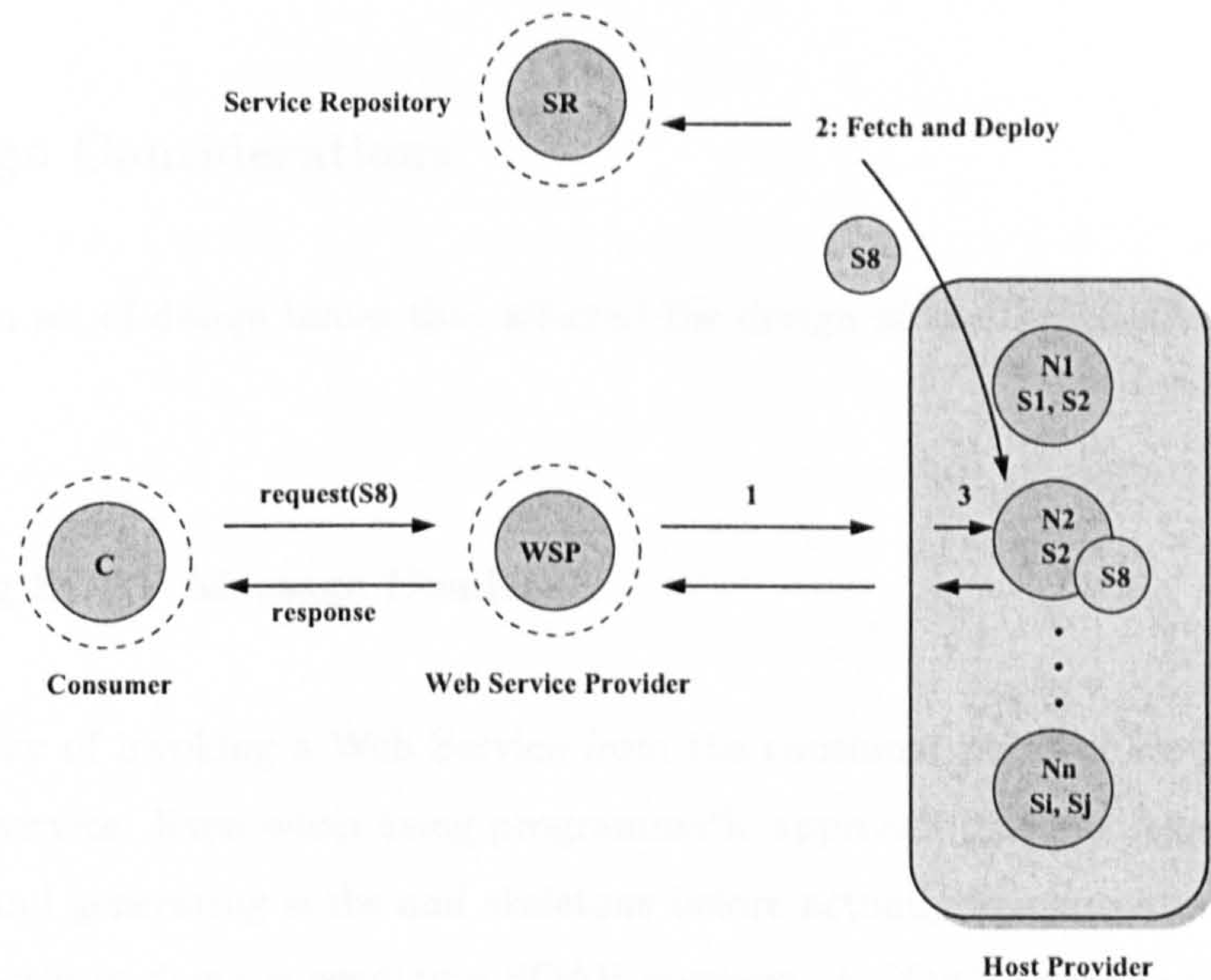
4.2.6 Requirements for DynaSOAr

To satisfy the scenarios presented in Section 4.2.5, certain requirements must be supported within the DynaSOAr framework. The requirements that are satisfied by the prototype system are listed below:

- **Ability to route service requests** - The DynaSOAr Service Provider should be able to



(a) Routing request to existing deployment



(b) Request for a service not already deployed

Figure 4.4: Routing of requests in DynaSOAr

route the requests sent by a consumer for a certain service to an appropriate Host Provider instance in a manner which will be transparent to the consumer.

- **Mobile service implementations** - The services must be implemented and packaged in a way which will allow them to be downloaded from the repository and deployed on any host with a web service container.
- **Resource Allocation** - DynaSOAr should be able to allocate resources on-demand from a pool of available resources for the deployment of a service. It should also be capable of dealing with a volatile environment such as the Grid, where resources may come and go.
- **Un-interrupting deployment of a service** - DynaSOAr must be able to deploy new ser-

vices on a designated host without interrupting the execution of the services that are already deployed on it.

- **Ability to store the services** - As opposed to the "one-time" affair of the job-scheduling systems, DynaSOAr should be able to store the service code in a persistent storage that will be accessible to the hosts needing to download the package.

- **Resource Handshake** - In order to perform better scheduling, DynaSOAr must know about the nodes available to it. This is possible by registering each node when it becomes available with the parent in the hierarchy of components. This may also be treated as an approach for establishing trust between the components.

4.2.7 Design Considerations

In this section, a set of design issues that affected the design of the DynaSOAr component services are discussed.

4.2.7.1 Using SOAP Message Headers

The standard way of invoking a Web Service from the consumer point of view is to send a SOAP message to the service. Even when using programmatic approach (such as Apache Axis) of binding to the service, and generating stubs and skeletons before actually invoking the service, all that the consumer is effectively doing is sending a SOAP message. A SOAP message contained in a SOAP envelope is composed of two distinct components with different purposes. The optional SOAP header element may be used to attach special instructions or transmit authentication or session management information. The actual processing instructions compose the SOAP body section. An example SOAP message sent by the consumer requesting for the EntropyAnalyserService is shown in Listing 4.1 below. It is to be noted that the address to which the message is sent does not appear within the message header or body, but is attached to the envelope when the message is sent across the network. The WS-Addressing [46] standard however specifies the use of explicit addressing fields in the message header. The use of message header to convey special information in DynaSOAr are discussed in the later sections.

Listing 4.1 An example SOAP message requesting the EntropyAnalysis service

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Header>
        <!-- Header information -->
    </soapenv:Header>
    <soapenv:Body>
        <!-- request to the service -->
        <m:calculateEntropy xmlns:m="http://entropy.neresc.ncl.ac.uk">
            <stSequence>AGTCMMMMTGCATMGTCATMMGGCCTACCTT</stSequence>
        </m:calculateEntropy>
    </soapenv:Body>
</soapenv:Envelope>
```

4.2.7.2 Using Message Orientation

Apache Axis [77] provides four styles for designing Web services, namely, (i) RPC, (ii) Document-oriented, (iii) wrapped and (iv) Message-oriented. These styles offer different options while designing the services, the RPC (Remote Procedure Call) being the default style. There have been several debates about the style of a Web Service and the document-oriented style is regarded as the most widely accepted and used format because of its richness and ability to describe a service and its operations. In DynaSOAr, each component makes use of the SOAP message header, and hence the services should be able to process the XML-formatted SOAP messages directly. Further, there are good arguments behind this message-oriented style, because this allows the consumers to be completely unaffected by the implementation at the server. This style proposes the use of a single interface at the service level, and the service implementation performs different operations based on the type of the incoming message. The primary requirement of this style is a detailed description of all the messages that can be processed by the service. In DynaSOAr, the participating components exchange messages which are defined in an XML schema. Consumers invoke the intended service using the standard procedures which result in a SOAP message being sent to the service. These messages sent by the consumers conform to the messages expected by the target service and can be generated using existing toolkits. The DynaSOAr service provider is able to distinguish these messages from any other message from another DynaSOAr component, and can process them accordingly, that is, forward them to the appropriate host provider for processing.

4.3 The design of DynaSOAr

The two major components of the DynaSOAr framework which allow the dynamic provisioning of services by processing the incoming consumer requests at different levels are namely a *Web Service Provider* and a *Host Provider*[1]. Two other components which perform roles that are not directly related to a consumer request, but are of utmost importance for on-demand provisioning of services are the *Service Repository* and the *Registry*. Each component in DynaSOAr is itself implemented as a service which opens up a wide range of deployment scenarios. For example, the *Service Repository* may be owned by the *Service Provider*, and access to it can be restricted to the Host Providers it trusts. The repository can also be a public repository, independent of any one Service Provider. It is even possible for the framework to be domain-specific, so that researchers collaborating in a specific area can share the repository. This section provides a description of each of these components.

4.3.1 Service Provider

The *Service Provider* is the entity with which consumers interact. It advertises the services it can provide, and accepts SOAP messages from consumers requesting a service from a particular endpoint associated with the message. The Service Provider is responsible for arranging the processing of the request. To achieve this, the Service Provider chooses an appropriate Host Provider and forwards the request to it with any associated Quality of Service (QoS) parameters and an added element in the message header identifying one or more *software repositories* from where a deployable version of the service can be acquired in case dynamic deployment is necessary. The Service Provider itself is designed as a Web Service, and conforms to the message-oriented style of Web Services. From a consumer point of view, a service is always exposed at the Service Provider end, and the consumer sends a message to the Service Provider, requesting the intended service. The DynaSOAr Service Provider extends the handler-chain concept used in Apache Axis [77], and the message is intercepted before it reaches the Service Provider. This is essential as the service endpoint within the message sent by the consumer will be that of the requested service, which may be either deployed on a different resource or may not even be deployed at that point of time. The intercepted message is modified in order to route it to the actual Service Provider web service, with a newly added element in the header denoting the abstract name of the service which is requested - an example of which is shown in Listing 4.2.

Once this message reaches the Service Provider, it can process the message accordingly, by either

[1] From this point onwards, the terms DynaSOAr Web Service Provider and Service Provider will be used interchangeably. Similarly, the terms Service and Web Service will be interchangeable as well.

Listing 4.2 An example SOAP message with a modified header element at the Service Provider

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Header>
        <!-- Header information -->
        <soapenv:dynasoar soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next">
            <soapenv:targetService value="EntropyAnalyserService"/>
        </soapenv:dynasoar>
    </soapenv:Header>
    <soapenv:Body>
        <!-- request to the service -->
        <m:calculateEntropy xmlns:m="http://entropy.neresc.ncl.ac.uk">
            <stSequence>AGTCMMMMTGCATMGTCATMMGGCCTACCTT</stSequence>
        </m:calculateEntropy>
    </soapenv:Body>
</soapenv:Envelope>
```

forwarding it to a deployed instance of the service or by requesting a new deployment on a new node and forwarding the message to this newly deployed instance for processing. While forwarding the message to a designated Host Provider, the Service Provider adds two new elements in the message header - one is a message identifier to establish a context between the request the response which will arrive at a later stage, and the other is the location of the software repository from where the code for the requested service may be obtained. The interaction of the consumer with the Service Provider, and the addition of new elements in the message header is depicted in Figure 4.5.

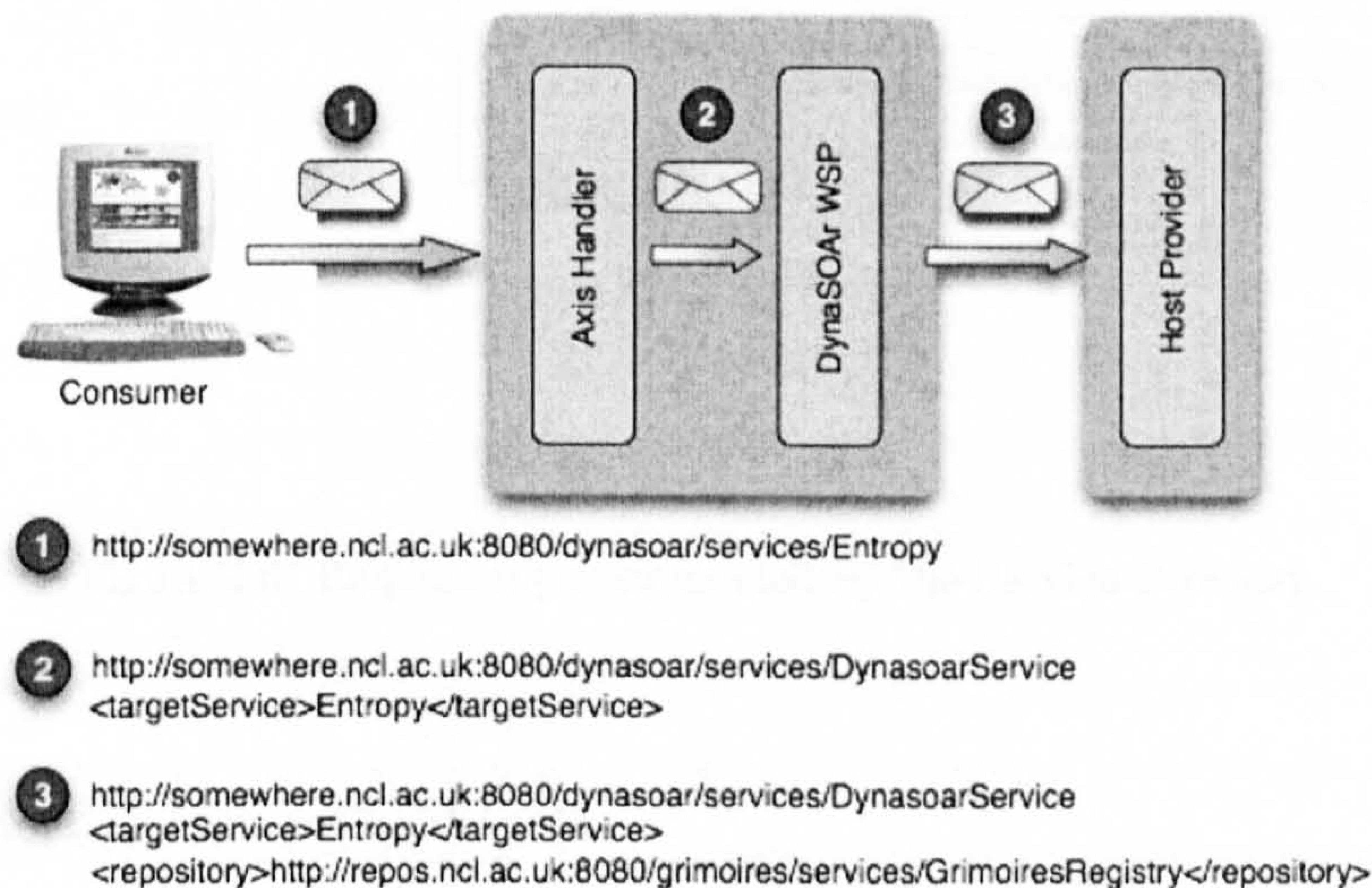


Figure 4.5: Consumer interaction with DynaSOAr Service Provider

Based on the requirements for DynaSOAr outlined in Section 4.2.6, the Service Provider must be able to fulfil the following criteria:

- It should advertise a set of services it can provide irrespective of whether the service is deployed or not.
- It must be able to receive service requests from consumers and arrange for their processing at a designated Host Provider.
- It must maintain a list of Host Providers to whom the request from the consumer can be forwarded for processing. For which the Host Providers should be able to register or deregister with the Service Provider.

As all the DynaSOAr components are designed based on the message-oriented approach, the service interface consists of one operation, which performs different functions based on the type of message it has received. The requests supported by the Service Provider are shown in Figure 4.6.

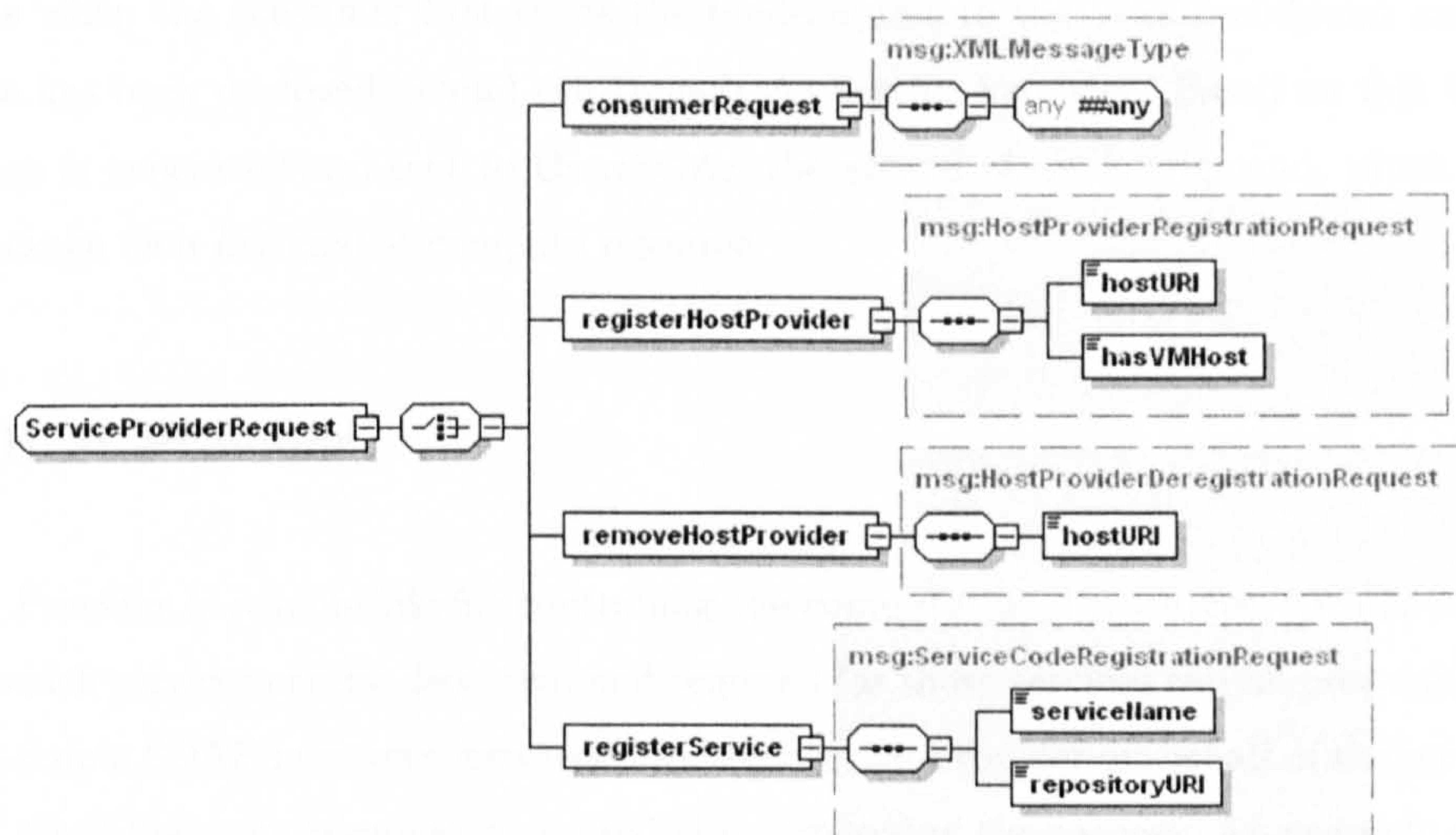


Figure 4.6: Request types supported by the Service Provider

4.3.1.1 Supporting conventional tools

Most common tools, such as XMLSpy (which although being an XML editor has advanced features like sending SOAP messages to a service) [115] or workflow enactment engines such as Taverna [10], used by a large number of consumers, require the endpoint of the service description (WSDL) before binding to the service. Apache Axis [77] provides a standard way of returning the WSDL by appending a "?wsdl" construct to the service endpoint, which is a common approach adopted by the

Web Service community. During the bind process, these tools are able to generate message formats conforming to the service interface which are then sent to the actual service. In the DynaSOAr framework, when a request from a consumer arrives at the service provider, the service may not have been deployed yet, and thus, the standard way of generating a WSDL (that is by adding a "?wsdl" construct after the service endpoint) using tools like Axis will not work. Thus, the consumers would have to know the format of the message that the service expects before invoking the service, which might have been a restrictive feature because many tools which are used to build applications assume that the WSDL can be accessed in the way mentioned above. This is handled in DynaSOAr using another extension of the Axis handler-chain feature, where the extended handler on receipt of any such request for the WSDL description will retrieve the WSDL file that was uploaded during the service upload phase, and return it back to the consumer from which the message formats can be generated by the standard tools. Thus, DynaSOAr, in effect deploys the WSDL description of the service when the consumer first sends the bind-request to the WSDL-endpoint assuming that the service has been deployed behind the DynaSOAr Service Provider. Based on this WSDL, once the message is generated and sent to DynaSOAr, the second phase is triggered, which deploys the service package on a designated compute resource.

4.3.2 Host Provider

The *Host Provider* is responsible for controlling the computational resources, such as a cluster or a Grid, on which services can be deployed, and requests for those services can be processed. The Host Provider accepts SOAP messages forwarded by the Service Provider on behalf of the services hosted by it, and sends back any response generated after processing the request. An example of a message received by the Host Provider is shown in Listing 4.3. The Host Providers are implemented as Web Services and conform to the message-oriented interface. Further, they are classified as manager and managed nodes, where the manager node possesses the knowledge about the managed nodes under its domain and is capable of scheduling the processing of a request on any of them. The manager or ROOT node periodically monitors each of the child nodes and uses this monitoring information to make decisions about the best suited node to process a request. It is also capable of routing a request to a specific instance of a service if multiple copies of the same service exist on different nodes in order to perform some load-balancing. Various scheduling algorithms such as *least recently used*, *least recently allocated*, *best average response time* are utilised within the host provider to make decisions about routing a request to a specific instance of the service. The Host Provider is implemented in such a way that new algorithms can be plugged into the framework. Further, the Host Provider uses resource allocation algorithms proposed in related works such as GridSHED [113]

which are based on heuristics and knowledge of network bandwidth data.

Listing 4.3 An example SOAP message with a modified header element at the Host Provider

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Header>
        <!-- Header information -->
        <soapenv:dynasoar soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next">
            <soapenv:messageId value="8580358022-012-0129090"/>
            <soapenv:targetService value="EntropyAnalyserService"/>
            <soapenv:repositoryURI value="http://repos.ncl.ac.uk/RepositoryService"/>
        </soapenv:dynasoar>
    </soapenv:Header>
    <soapenv:Body>
        <!-- request to the service -->
        <m:calculateEntropy xmlns:m="http://entropy.neresc.ncl.ac.uk">
            <stSequence>AGTCMMMMTGCATMGTCATMMGGCCTACCTT</stSequence>
        </m:calculateEntropy>
    </soapenv:Body>
</soapenv:Envelope>
```

The Host Provider, on receiving a message from the Service Provider, checks whether the requested service has already been deployed as a result of a previous request. If the service has been deployed previously, which can be discovered from the *registry*, the best suited instance is selected and the request is forwarded to that instance for processing. If the service has not been deployed before, or the existing instances are heavily loaded, the request is forwarded to a host which is lightly loaded at that point of time. The destination host, which is a Host Provider as well, extracts the location of the repository where the deployable code for the requested service can be obtained from the message header, downloads the package and deploys it on itself. The request is then processed and a response is sent back. The interactions between the Service Provider and the Host Provider are depicted in Figure 4.7.

The Host Provider is required to support the following requirements -

- It must be able to receive service requests forwarded by the Service Provider and process them accordingly.
- It must maintain a list of child Host Providers under its domain to whom the service request can be routed to. In order to achieve this it must allow other hosts to register and deregister with it.
- It must have the knowledge of the services supported by it.

As explained in Section 4.2.7.2, the Host Provider implements the message-oriented interface and

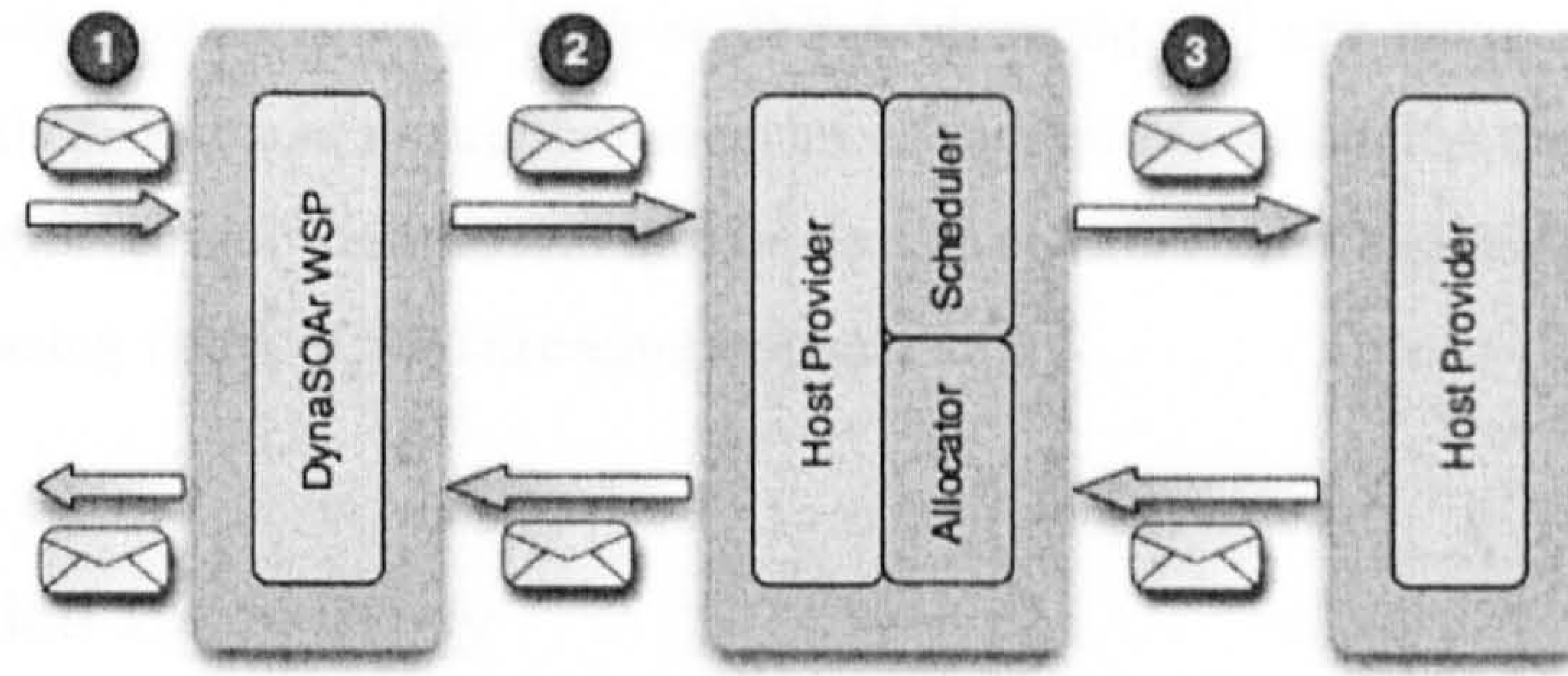


Figure 4.7: Interaction between DynaSOAr Service Provider and Host Provider

exposes one operation which is capable of supporting the requests depicted in Figure 4.8.

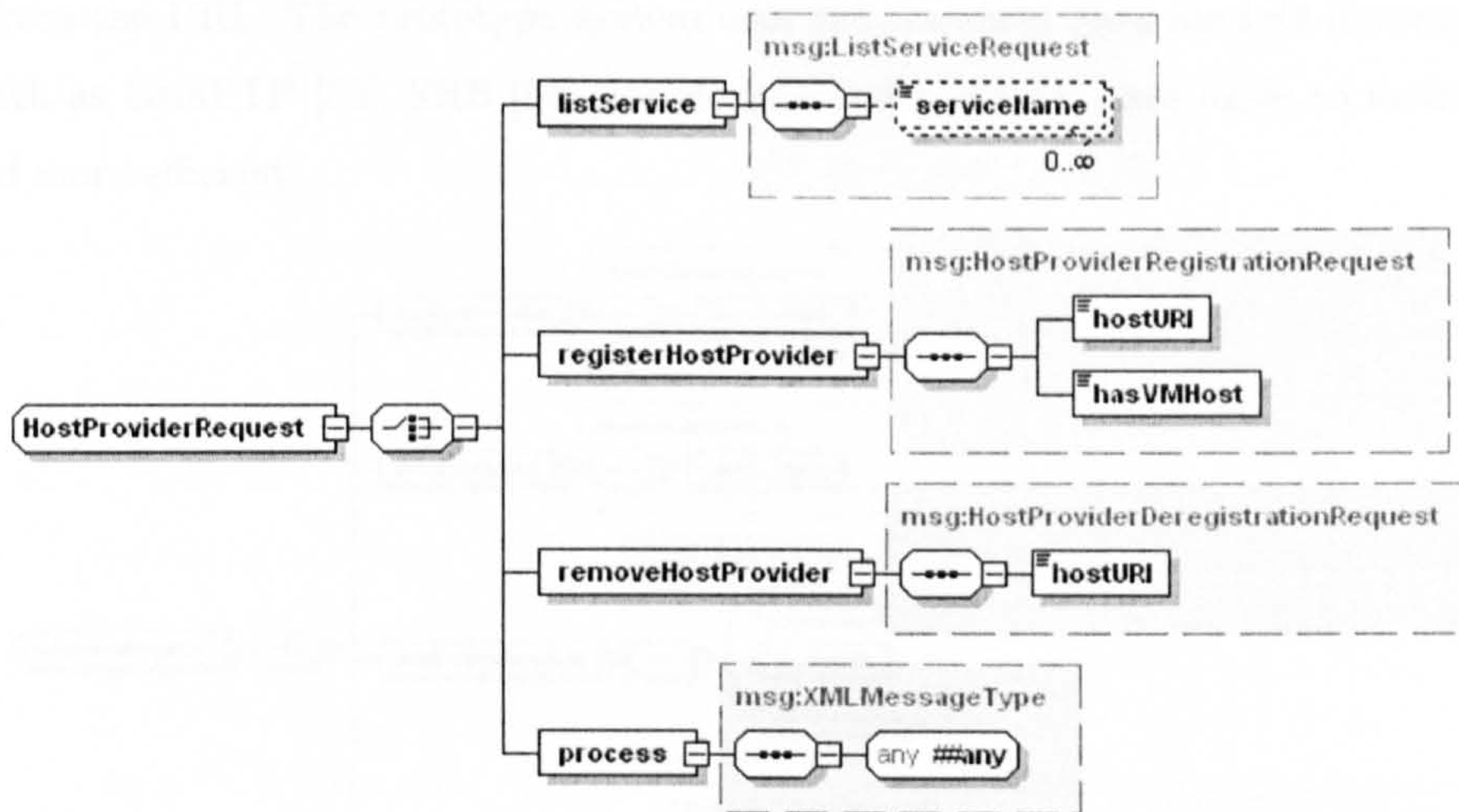


Figure 4.8: Request types supported by the Host Provider

The first prototype is based on the assumption that the requests are synchronous in nature. Hence, the response from the Host Provider retraces the arrival route of the request. Specifications for

addressing asynchronous requests, such as WS-Addressing [46] are being standardised, and it is possible to use these features to support asynchronous requests where the consumer will not have to block for the response, and the response can be sent to the consumer directly from the Host Provider using the addressing fields of the incoming request.

4.3.3 Service Repository

The *Service Repository* is not directly accessible by the consumers, but it plays an important role within the DynaSOAr framework by managing the upload and download of the deployable service code and retaining them for future use. It is implemented as a service based on the message-oriented model supporting the request types outlined in Figure 4.9. A service developer "uploads" the service package along with the description of the service as a WSDL document. Following a successful upload, the service is registered in the registry and stored in a file store, and a unique URL is used to refer to it. The Host Providers communicate with this service by sending and receiving SOAP messages requesting the download of the actual service code for a service to be deployed. When such a download request is received, the repository service allows the host provider to download the software from the URL. The prototype system uses the standard Java file I/O libraries, but other options such as GridFTP [55], SRB [54] (for storage and transport) are likely to make the system robust and more efficient.

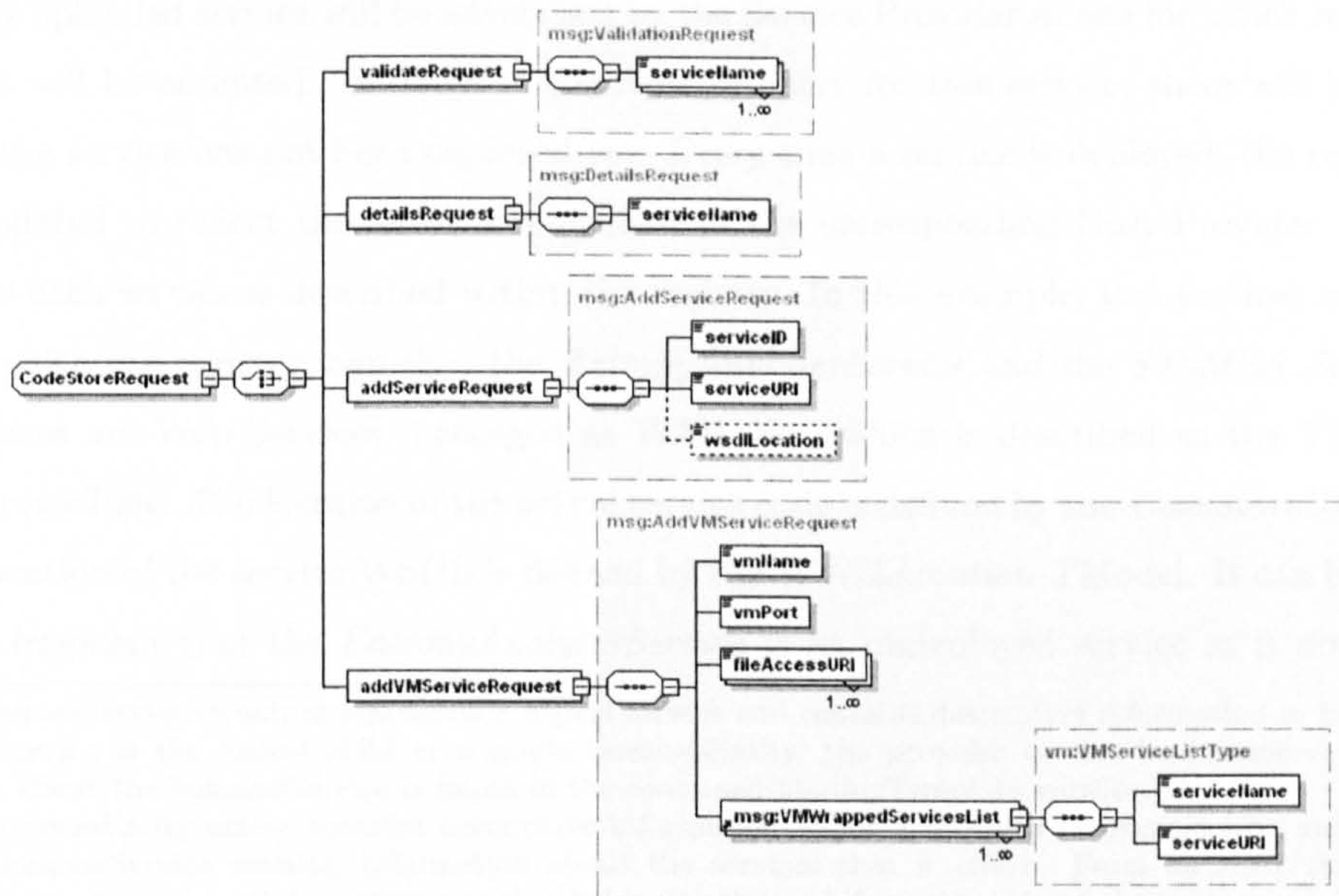


Figure 4.9: Request types supported by the Service Repository

In order to satisfy the requirements in DynaSOAr, the Service Repository must support the following criteria -

- It must allow service developers to "upload" a service package to the repository.
- When a service is uploaded, the repository must register the service with the *service registry* whereby the availability of the service is made known to the consumers.
- It must be able to respond to requests from the Host Providers to download the code.

4.3.4 Registry Service

The *DynaSOAr Registry Service* is provided by GRIMOIRES [103], which is an UDDI-based registry, with added support for metadata as RDF [116] triplets. When a service developer decides to make a service available via the DynaSOAr Web Service Provider, the deployable service code is uploaded to the *service repository* as a result of which, the registry is updated with this information. A *businessService*[2] entity is created within the registry corresponding to this new service as a child of the single "DynaSOAr" businessEntity[3]. Several *TModel*[4] entries are also attached to the new service entry, each describing certain aspects of the service, such as its type, location of the code and the WSDL description, each having a definite use during the service download and deployment process. This newly uploaded service will be advertised by the Service Provider as one for which requests from consumers will be accepted. Initially, in the registry entry for this service, there will be no *access points* as the service has not been deployed yet. Every time a service is deployed, the registry entry will be updated to reflect the recent deployment at the corresponding Host Provider. Listing 4.4 shows how each service is described within the registry. In this example, two services are shown as registered with the registry, namely - the *EntropyAnalyserService* and the *SJEMEA_Service*. Both these services are Web Services, packaged as WAR files, which is described as the TModel entry named *ServiceType*. The location of the actual service code is defined by the *CodeStoreURL* TModel, and the location of the service WSDL is defined by the *WSDLLocation* TModel. It can be seen from the XML fragment that the *EntropyAnalyserService* is an undeployed service as it does not have

[2]The businessService structure represents a logical service and contains descriptive information in business terms. A businessService is the logical child of a single businessEntity, the provider of this businessService. Technical information about the businessService is found in the contained bindingTemplate entities [81].

[3]Each businessEntity entity contains descriptive information about a business or organisation and, through its contained businessService entities, information about the services that it offers. From an XML standpoint, the businessEntity is the top-level data structure that holds descriptive information about the business or organisation it describes [81].

[4]Technical Models(TModels) are used in UDDI to represent unique concepts or constructs. They provide a structure that allows re-use and, thus, standardisation within a software framework. The UDDI information model is based on this notion of shared specifications and uses tModels to engender this behaviour. For this reason, tModels exist outside the parent-child containment relationships between the businessEntity, businessService and bindingTemplate structures [81].

any *AccessPoints* attached to it, whereas, the *SJEMA_Service* is deployed on one of the available nodes, and the endpoint is attached to the service entity as an *AccessPoint* entry.

Listing 4.4 Description of entities registered within the DynaSOAr registry

```
<businessDetailExt xmlns="urn:uddi-org:api_v2">
    <businessEntityExt>
        <businessEntity businessKey="87a248e1-38a8">
            <name>Dynasoar</name>
            <businessServices>
                <businessService businessKey="87a248e1-38a8" serviceKey="13f3afbc-e614">
                    <name>EntropyAnalyserService</name>
                    <bindingTemplates/>
                    <categoryBag>
                        <keyedReference keyName="CodeStoreURL"
                        keyValue="http://giga01:8199/codestore/services/RepositoryService"
                        tModelKey="880ce26e-5abe"/>
                        <keyedReference keyName="WSDLLocation"
                        keyValue="http://giga01:8199/ServiceCode/EntropyAnalyserService.wsdl"
                        tModelKey="659050a3-642d"/>
                        <keyedReference keyName="ServiceType"
                        keyValue="WAR" tModelKey="5bae7c06-d9f1"/>
                    </categoryBag>
                </businessService>
                <businessService businessKey="87a248e1-38a8" serviceKey="e4aa15a8-a383">
                    <name>SJEMEA_Service</name>
                    <bindingTemplates>
                        <bindingTemplate bindingKey="bdf2ca51-8e85" serviceKey="e4aa15a8-a383">
                            <accessPoint URLType="http">
                            http://giga02 :8199/hostprovider/services/HostProviderService
                            </accessPoint>
                            <tModelInstanceDetails>
                                <tModelInstanceInfo tModelKey="2fe4f58f-7562"/>
                            </tModelInstanceDetails>
                        </bindingTemplate>
                    </bindingTemplates>
                    <categoryBag>
                        <keyedReference keyName="CodeStoreURL"
                        keyValue="http://giga01:8199/codestore/services/RepositoryService"
                        tModelKey="880ce26e-5abe"/>
                        <keyedReference keyName="WSDLLocation"
                        keyValue="http://giga01:8199/ServiceCode/SJEMEA_Service.wsdl"
                        tModelKey="659050a3-642d"/>
                        <keyedReference keyName="ServiceType"
                        keyValue="WAR" tModelKey="5bae7c06-d9f1"/>
                    </categoryBag>
                </businessService>
                ...
            </businessServices>
        </businessEntity>
    </businessEntityExt>
</businessDetailExt>
```

4.3.5 The Software Hypermarket

The description so far about DynaSOAr took into consideration the existence of a single Host Provider. In reality, though, one Web Service Provider may have several Host Providers at its disposal. At certain times, it may be advantageous to consider the characteristics of the available Host Providers, such as cost, dependability, QoS, security, etc. to select the most suited candidate. As an example, it may be logical to refer to the Amazon Elastic Compute Cloud [88], where consumers are allowed access to computational resources at a certain cost. If there are other providers offering resources to consumers, the consumers will have the option of choosing between the available resources based on certain criteria, cost being one of them. A similar scenario can be considered for

computational services, for example, tourism reservation systems, where the consumers may want to make selections based on the quality of service.

These scenarios converge to the possibility of a *Software Marketplace* or a *Software Hypermarket*[5] (as shown in Figure 4.10) where similar services are provided by different providers and the consumers have the option of selecting one or more of them, just like a consumer does in a normal shopping mall. The option of choosing between multiple Host Providers also exists from the point of view of the Service Provider.

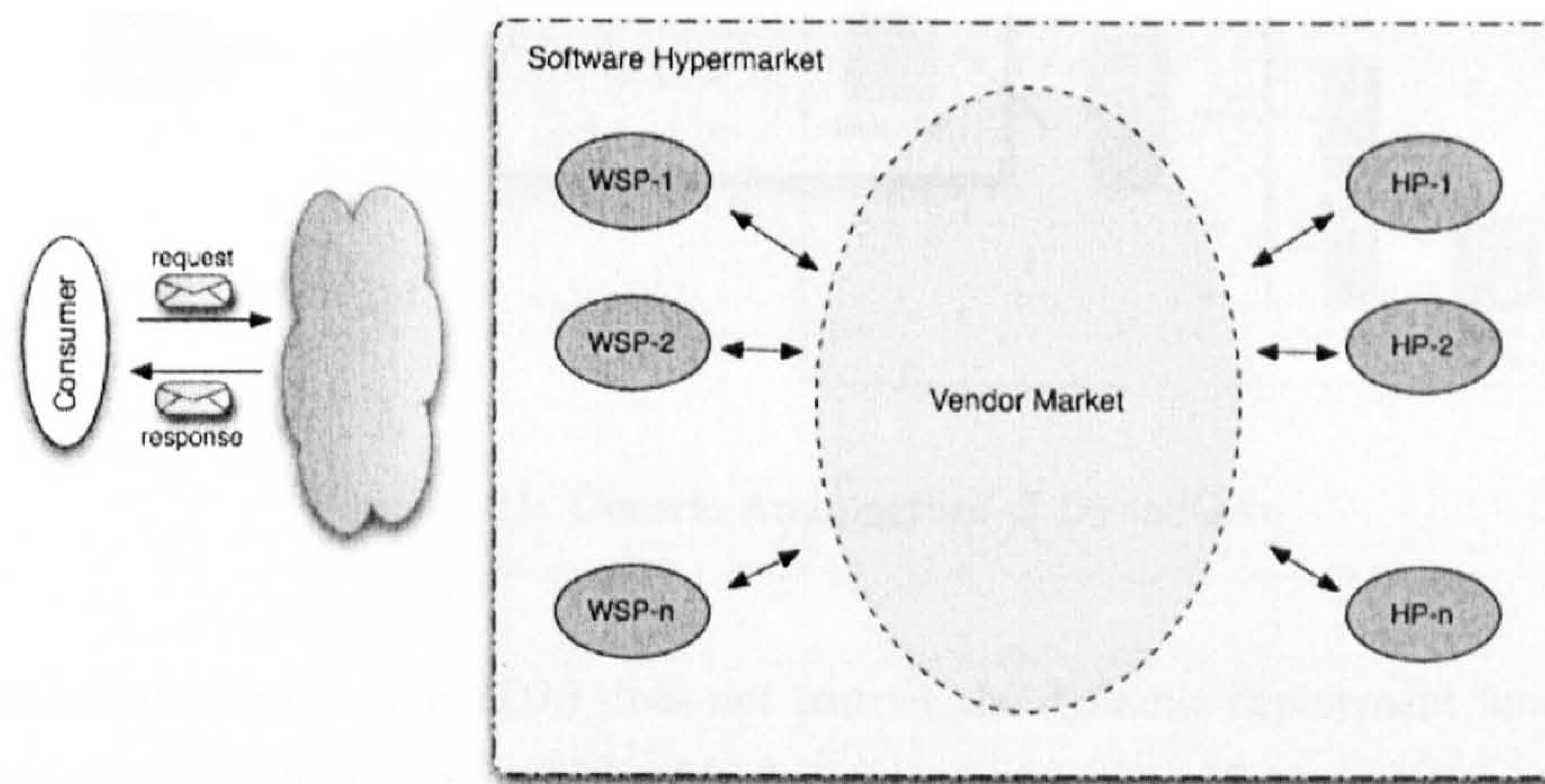


Figure 4.10: The Software Hypermarket

To facilitate this requirements matching (similar to the matchmaking process of Condor), another component, the Broker, with the same interface as the Host Provider, has been introduced in the architecture, which is responsible for making such decisions. The broker has the knowledge about one or more Host Providers, and is thus able to make the decisions based on the characteristics of the available Host Providers and the QoS and security requirements requested by the consumer.

The generic architecture of DynaSOAr follows from the concept of this *Software Hypermarket* which can allow interactions between multiple service providers, brokers and host providers. DynaSOAr allows the hierarchy of Service Provider, Broker and Host Provider to grow dynamically to any level or depth, as illustrated in Figure 4.11. There is no limitation in the number of brokers or Host Providers that can take part in the formation of this ad-hoc Virtual Organization, creating the *Software Hypermarket* where the brokers can choose between available Host Providers meeting the consumer requirements to process the requests.

[5]In commerce, a hypermarket or multi-department store is a superstore which combines a supermarket and a department store. The result is a very large retail facility which carries an enormous range of products under one roof, including full lines of groceries and general merchandise. When they are planned, constructed, and executed correctly, a consumer can ideally satisfy all of his or her routine weekly shopping needs in one trip [117].

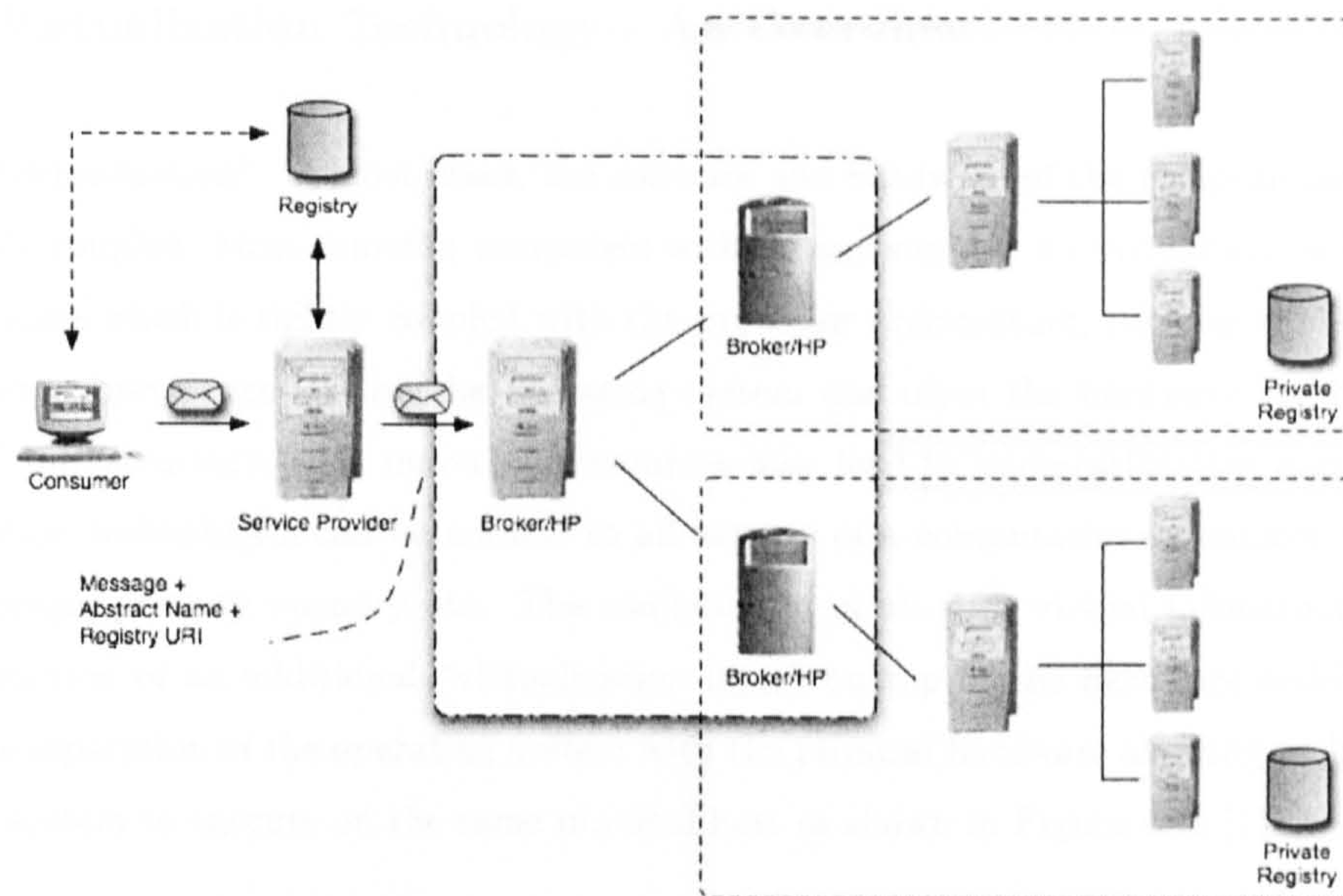


Figure 4.11: Generic Architecture of DynaSOAr

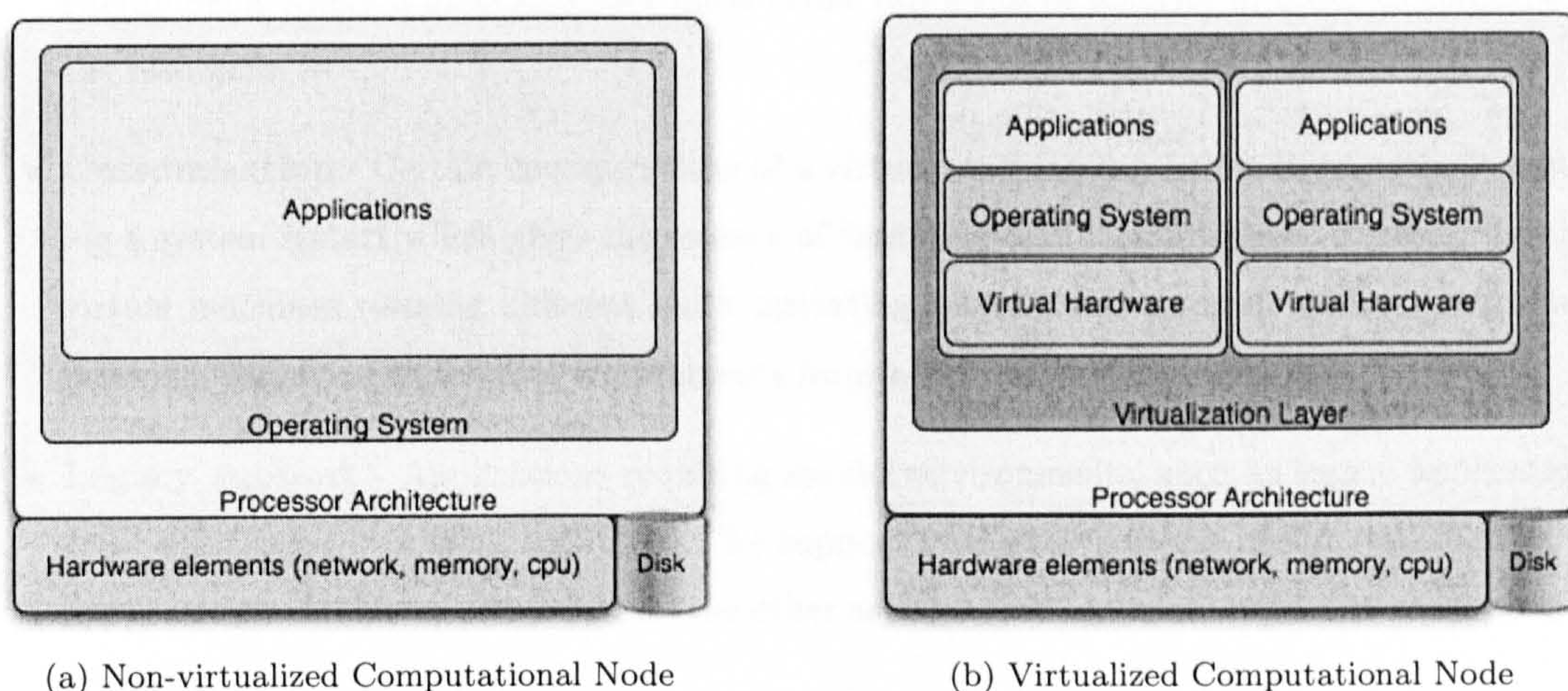
The generic architecture of DynaSOAr does not restrict the dynamic deployment functionality to web services alone. It is capable of allowing dynamic deployment of *Virtual Machines* such as VMWare [15] as will be described in the following sections.

4.4 Using Virtualization

In recent times, virtualization technologies have become a popular choice in the Grid world. Virtualization, as defined by VMWare is: "an abstraction layer that decouples the physical hardware from the operating system to deliver greater IT resource utilisation and flexibility" [15]. One specific virtualization technique is *virtual machines*, which goes back to the IBM System/370 [90]. Using such techniques, it is possible to run several virtual machines (VM) simultaneously, with different operating systems, on the same physical host by isolating each one from the physical environment. The advantages provided by VMs regarding partitioning, isolation and encapsulation make them useful in the Grid infrastructure as proposed in [17]. DynaSOAr utilises this concept and provides an extensive *service-oriented* framework which allows on-demand deployment of virtual machines which can encapsulate databases, services, any special environment that may be required for the services in a flexible way creating an ad-hoc virtual grid.

4.4.1 Virtualization Technology - An Overview

Prior to "virtualization", in most cases, the software and hardware of the computational resources were tightly coupled. Most common computers without any support for virtualization have one operating system which is tightly coupled with the processor architecture, running applications most of which are again dependent on the operating system and often the hardware architecture. For a large IT infrastructure, such individual resources may lead to under-utilisation and inflexibility. Virtualization technologies can be applied to all aspects of a computational resource, such as networks, storage, primary memory etc. The combination of all such virtual infrastructure leads to the introduction of an additional "virtualization layer" on top of the hardware architecture with creates the separation of the operating system with the physical hardware allowing multiple "guest" operating system to execute on the same physical host as shown in Figure 4.12 [118].



(a) Non-virtualized Computational Node

(b) Virtualized Computational Node

Figure 4.12: Before and after virtualization

The possibility of using *Virtual Machines*(VM) for Grid computing is explored in [93, 17, 119]. The traditional grid middleware solutions implement the abstraction layer at the level of the individual nodes which makes it difficult to provide the adequate level of security necessary to protect the resources against untrusted legacy codes submitted as *jobs* by untrusted users. By raising the level of abstraction to the *operating system virtual machine*, three fundamental advantages are obtained, namely, support for legacy applications, security against untrusted code, and a computation process independent of site administration which are described in more details later in this section. Each virtual machine appears to be an individual self-sufficient machine for a user, which de-couples it from the underlying physical resources, and other virtual machines running on the same physical host. From the administrative point of view, the entire operating system of the VM becomes independent of the computational resources, and the VM, including its state, can be described in

a set of conventional files. Further, migration of live virtual machines along with its state from one host to another, as described in [18], increases the their useability manifolds within the Grid application frameworks. The advantages of using VMs in Grid computing are outlined in [93] as follows:

- **Security and isolation** - Resource sharing is a primary requirement for the Grid environment thereby creating the requirement of the integrity and security of the shared resources. A security model based on trust between the user and the provider may still give rise to situations where the integrity and security of the shared resources are compromised by a piece of malicious code, and conversely, the integrity of a computation may be compromised by a malicious resource. Virtual Machines allow completely isolated environments for each user, each sharing the same physical resource but independent of each other, thus creating a more secured environment where a malicious user must break two levels of security in order to compromise the resources.

- **Customisation** - Certain configurations of a virtual machine can be modified without requiring a system restart which gives the essence of flexibility and customisation. Further, multiple virtual machines running different guest operating systems can co-exist within a single host machine, satisfying individual requirements from a pool of physical machines.

- **Legacy support** - Applications requiring special environments, such as legacy applications, can be packaged in virtual machines. The support for legacy systems in not restricted at the application level, but encompasses all the other aspects such as virtual hardware and operating systems.

- **Administrator privileges** - Access restrictions are imposed on traditional multi-programmed systems where most sensitive operations can only be performed by the privileged user, such as the system administrator, which may be limiting in certain cases. In case of virtual machines such regulations can be relaxed because each VM is isolated from the others and the physical resources.

- **Resource control** - Resources can be controlled at a coarser granularity in case of virtual machines as opposed to traditional multi-programmed where resource allocation is normally done for each application or process. In case of virtual machines, resources, such as memory, virtual disk size etc can be allocated during the initialisation phase, and is done per virtual machine from the virtual machine management layer, or the *Virtual Machine Monitor*. Dynamic allocation of resources is also possible for virtual machines which may be considered a key aspect for Grid applications.

- **Site independence** - The virtual machines themselves are independent of the host operating system on the physical resource. This allows cross-domain migration of an entire computation by moving the VM from one host to another irrespective of the physical resources, such as CPU architecture, memory etc. Live migration of running virtual machines is also possible (as shown in [18]), which is another feature that can be exploited within the Grid framework.

The approach to virtualization can broadly be categorised into two different categories, namely *hosted* and *hypervisor*. In the *hosted* approach, several virtual machines with different operating systems can co-exist sharing the underlying physical resources, which is also known as *partitioning*. The VMWare Server [120] and Microsoft Virtual PC [121] offer virtualization based on this approach where the virtualization layer, often known as the *Virtual Machine Monitor* (VMM), relies on the host operating system for device support and management of the physical resources, such as memory, processor, network etc. On the other hand, in case of the *hypervisor* approach, the hypervisor itself is the first layer of software installed on a clean processor hardware layer, because of which it is often known as the *bare metal approach*. In this case, the virtualization layer has direct access to the hardware resources and may provide more efficiency compared to the *hosted* architecture in terms of robustness, scalability and performance. Recent investigations in virtualization technologies have given rise to several enhancements, one of them being *para-virtualization*, supported in the Xen hypervisor [16]. Paravirtualization aims at improving the performance and scalability of the virtual machines and proposes a new virtual machine interface where the guest VMs are 'aware' of their virtualized state as the guest operating systems are modified to exploit this feature.

A comprehensive survey of the existing virtualization technologies and approaches is available in [122].

4.4.2 Case for Virtualization in DynaSOAr

Apart from the advantages of virtualization such as security, isolation, resource control etc. mentioned in Section 4.4.1, there are more usage scenarios which can benefit from using virtualization. DynaSOAr can exploit these scenarios as explained below:

- **Data Caching** - One of the major motivations behind DynaSOAr was the *Active Information Repository* architecture [7] which proposes to deploy the data analysis services closer to the data. An alternative approach may be suitable in some cases where it is not important that the latest version of the data is used for analysis. A bio-informatician may be executing a particular workflow which accesses data from a database located at a large geographical

distance, where it may not be possible to deploy the analysis application closer to the data, or even transferring the results to the consumer may be costly. In such cases a partial copy or a snapshot of the database deployed on a node closer to the analysis code may prove to be beneficial. This, in effect is analogous to data caching. In DynaSOAr, such a snapshot of the database, wrapped with data access services, such as OGSA-DAI can be packaged in a virtual machine and deployed on demand.

- **Special Environments** - Many workflows in bio-informatics use specialised services, such as ***Blast*** [9] which analyses a given protein or gene sequence and returns similarity scores with already existing sequences in a database. This service requires a special set of libraries and a database to perform the similarity analysis, all of which can be packaged in a virtual machine. Many scientific analysis services require considerable amount of tuning to the underlying host for optimal performance, which varies between the type of host on which it is deployed. Such services, if deployed on a host using the approach taken so far in DynaSOAr, will not provide any benefits, as the tuning is generally an offline process. The virtualization approach can provide an alternative solution where such services can be deployed on virtual hosts and tuned before the entire package is stored in the repository, which when deployed will make the optimally configured service available to the consumers.

4.4.3 Using Virtualization in DynaSOAr

In DynaSOAr, the VMWare Server [120] has been used as the primary virtualization infrastructure. DynaSOAr itself is agnostic about the type of virtualization, hence, it should be able to incorporate any other virtualization approach. Virtual Machines are used in DynaSOAr to provide support for services requiring special environments, and a means to allow a form of data caching for certain situations. Instances of VMs are pre-built, and services are deployed on them on an Apache Tomcat (or any other) Web service container. Special environments, such as third-party libraries, databases, required by the service are included in the VM. These instances of virtual machines are uploaded to the *Software Repository* like any other DynaSOAr service, and relevant entries are stored in the *registry*. For the benefit of registering the services, each virtual machine is described as an XML document based on the schema shown in Figure 4.13. This document is uploaded along with the files relevant to the VM (such as the virtual hard disk and the VM configuration file), and each service, including the data access services, is registered in the registry, with several TModel objects used to describe each entry as shown in Listing 4.5.

As opposed to [17], the DynaSOAr method of deployment is transparent to the consumer, and is

Listing 4.5 Entries in the registry describing services packaged in a virtual machine

```
<businessDetailExt xmlns="urn:uddi-org:api_v2">
    <businessEntityExt>
        <businessEntity businessKey="df5ee878-8b12">
            <name>Dynasoar</name>
                                    [A service deployed on a VM and a normal host]
            <businessServices>
                <businessService businessKey="df5ee878-8b12" serviceKey="d6efa37a-4506">
                    <name>HostProviderService</name>
                    <bindingTemplates>
                        <bindingTemplate bindingKey="0b0c4747-689b"
                        serviceKey="d6efa37a-4506">
                            <accessPoint URLType="http">
                            http://giga10:8090/hostprovider/services/HostProviderService
                            </accessPoint>
                            <tModelInstanceDetails>
                                <tModelInstanceInfo tModelKey="6e512ad6-f2ed"/>
                            </tModelInstanceDetails>
                        </bindingTemplate>
                        <bindingTemplate bindingKey="a3763c91-9dab"
                        serviceKey="d6efa37a-4506">
                            <accessPoint URLType="http">
                            http://vm-1:8090/hostprovider/services/HostProviderService
                            </accessPoint>
                            <tModelInstanceDetails>
                                <tModelInstanceInfo tModelKey="6e512ad6-f2ed"/>
                            </tModelInstanceDetails>
                        </bindingTemplate>
                    </bindingTemplates>
                </businessService>
                                    [Description of a service packaged in a VM]
                <businessService businessKey="df5ee878-8b12" serviceKey="4f84545e-d70a">
                    <name>QueryEvaluationService</name>
                    <categoryBag>
                        <keyedReference keyName="CodeStoreURL"
                        keyValue="http://giga01:8090/codestore/services/RepositoryService"
                        tModelKey="c67e2622-e5b1"/>
                        <keyedReference keyName="ServiceType" keyValue="VMWARE"
                        tModelKey="58e48324-764b"/>
                        <keyedReference keyName="VM-NAME" keyValue="GenericLinuxVM"
                        tModelKey="516acd60-fa8b"/>
                        <keyedReference keyName="ServiceURI"
                        keyValue="dqp-evaluator/services/QueryEvaluationService"
                        tModelKey="41f5e363-0a1b"/>
                        <keyedReference keyName="VM-PORT"
                        keyValue="8090" tModelKey="ad35cb4d-3a0a"/>
                        <keyedReference keyName="ServiceTag"
                        keyValue="WEB-SERVICE" tModelKey="ed9e3a02-6232"/>
                    </categoryBag>
                </businessService>
                                    [Description of a data service packaged in a VM]
                <businessService businessKey="df5ee878-8b12" serviceKey="5a081adc-6004">
                    <name>ProteinSequenceMySQLResource</name>
                    <categoryBag>
                        <keyedReference keyName="CodeStoreURL"
                        keyValue="http://giga01:8090/codestore/services/RepositoryService"
                        tModelKey="c67e2622-e5b1"/>
                        <keyedReference keyName="ServiceType" keyValue="VMWARE"
                        tModelKey="58e48324-764b"/>
                        <keyedReference keyName="VM-NAME" keyValue="GenericLinuxVM"
                        tModelKey="516acd60-fa8b"/>
                        <keyedReference keyName="ServiceURI"
                        keyValue="axis/services/ogsadai/ProteinSequenceDataService"
                        tModelKey="41f5e363-0a1b"/>
                        <keyedReference keyName="VM-PORT"
                        keyValue="8090" tModelKey="ad35cb4d-3a0a"/>
                        <keyedReference keyName="ServiceTag"
                        keyValue="DATA-SERVICE" tModelKey="ed9e3a02-6232"/>
                    </categoryBag>
                </businessService>

            </businessServices>
        </businessEntity>
    </businessEntityExt>
</businessDetailExt>
```

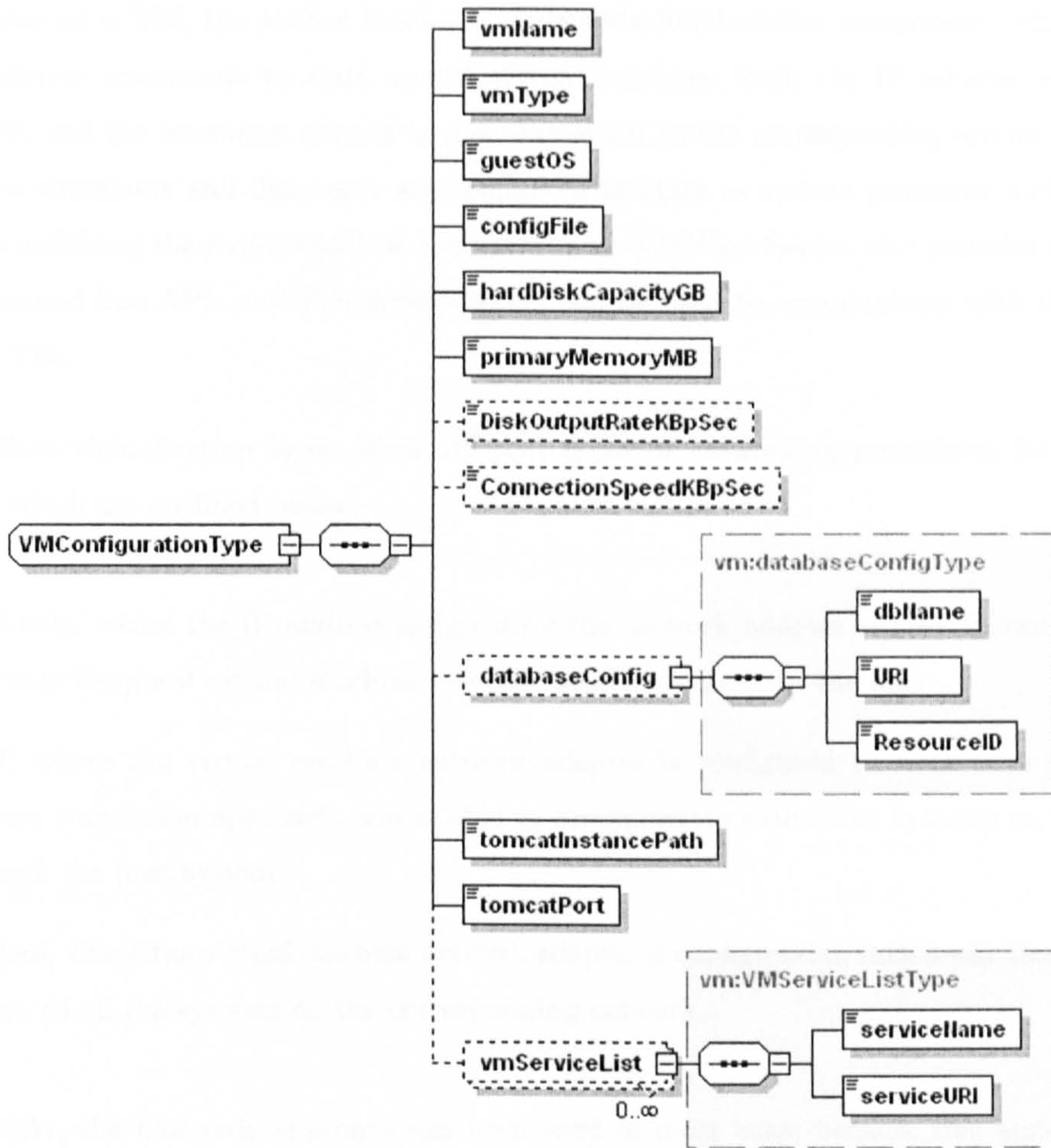


Figure 4.13: XML Schema for describing a Virtual Machine

uniform for all the supported components. When a request is received from a consumer for a service that is either not yet deployed, or for which the existing deployments are overloaded or unavailable, the VM image is downloaded to an appropriate host and started. The request is then forwarded to the hosted service. Starting up a VM also ensures that all the other hosted services in it are available, and any requests for these services can be forwarded to the service without the need for redeployment. It should be noted however, that downloading and starting up a virtual machine is costly, and DynaSOAr ensures that this is done only when necessary to reduce the overhead.

Each available node in DynaSOAr hosts a HostProvider service, exposing the underlying resource. This service is responsible for downloading and deploying the service or VM image from the repository when a dynamic deployment is called for. Otherwise, it forwards the request to the already deployed service and sends the response back to the consumer via the DynaSOAr Service Provider.

When deploying a VM, the service invokes a VirtualMachineInstaller component, which uses the VMWare Server commands to start up the virtual machine, fetch the IP address of the newly started VM, and the consumer request is then forwarded to the corresponding service on the VM. The service containers and databases are configured to start as system processes during the VM boot, thus nullifying the requirement of any user input. VMWare Server also provides an extensive set of command line APIs and Perl scripts which can be used to communicate with the server to control to VM.

The VMWare virtualization layer offers different types of networking procedures for the virtual machines, which are outlined below:

1. *Host-only*, where the IP address assigned by the network adapter is only known to the host and thus the guest virtual machine can only communicate with the host.
2. *NAT*, where the virtual machine network adapter is configured to work with the *network address translation* approach, and is able to communicate with other systems on the network through the host system.
3. *Bridged*, where the virtual machine network adapter is configured in such a way that it becomes known to all the systems on the corresponding network.

In DynaSOAr, the *host-only* approach has been used in most cases because this approach allows the isolation of the VMs from the external world, thus increasing the security of the system, and secondly, this approach rules out the possibility of any IP address collision issue that can arise when a suspended VM becomes active but its IP address has been re-issued by the DHCP server of the network. This approach is also suitable for organisations who do not want to expose the details of their computational resources. However, in some cases, such as the Distributed Query Processors discussed in Chapter 3 and 5, the "bridged" configuration is necessary, because the participating services communicate directly by sending messages to each other. In such cases the IP addresses have been selected from a pre-reserved pool.

4.5 Discussion

This chapter introduced an alternative approach to Grid computing which revolves round *services* rather than *jobs*. Traditionally most Grid middlewares use a distributed job scheduling system which submits the job execution code from the consumer along with all inputs to a target host where the

job is executed. This execution is however an "one-off" execution because the execution code is not retained at the host where it is executed. If the consumer wants to run several experiments using different input data, the execution code must be submitted for each execution. In *service-oriented* systems, a service once deployed is available for a longer period until it is explicitly undeployed. In DynaSOAr, this feature is exploited to spread the cost of deployment amongst multiple possible invocations from the consumers.

Further, DynaSOAr proposes a demand-driven deployment approach. Services are only deployed when there is a requirement for it. All services registered in the DynaSOAr registry are advertised by the Service Provider as available services, whether or not they are deployed at any particular point of time. DynaSOAr also makes a logical separation between the provider of a service and the provider of a resource on which the service is deployed and requests processed. To a consumer, the services appear to be provided by the DynaSOAr Service Provider, although, the services may be deployed on a host different from the actual Service Provider, or may not be deployed at all. Consumers can send a request to any such service, and based on the state of the current deployed instances of the requested service (if any), the request is forwarded to a designated host, which either processes the request or deploys the service in order to process the request. The logical separation of the service provider and resource/host provider by DynaSOAr creates a possibility of different organisational models, and also provides the consumers freedom to choose between available service providers who provide a similar service. This is viewed as the *Software Hypermarket* in DynaSOAr.

In DynaSOAr, a *message-oriented* approach is taken while designing the service interfaces. This allows the consumers to be completely agnostic about the internal data structures and objects that are used within DynaSOAr, and even the Service Providers and Host Providers are not tightly coupled except for a mutually agreeable contract in a form of allowed messages that can be exchanged between them. DynaSOAr extensively uses the UDDI specifications to store the description of the services it can provide along with other characteristics of the services, such as the type of the service and the location of the actual service code, in a the UDDI-compatible GRIMOIRES registry.

Virtualization is an important aspect in DynaSOAr and is viewed as an approach to create ad-hoc virtual organisations on demand. An alternative approach to the Active Information Repository proposal of moving the analysis code to the data is advocated in DynaSOAr by the use of virtual machines where in certain cases a snapshot of the database is moved closer to the computation, which can be used as a way of data caching. Virtualization in DynaSOAr is completely transparent to the consumer, and the services appear to be normal Web Services hosted on real hosts.

The prototype uses standard Java File I/O to manage the upload and download of the service

code, including virtual machines, which is one of the concerns in DynaSOAr. This approach works without any problems for small packages, but is costly for larger files, such as virtual hard-disks, which makes the deployment of a virtual machine extremely costly. For such files, the SFTP utility was used with more success. But the use of a more robust mechanism such as GridFTP and/or SRB for managing the storage and transport should reduce the cost and improve the performance. Other approaches, such as Peer-To-Peer (P2P) systems may also be efficient, especially, when there is the need of distributing the same executable code (services or virtual machines) to multiple hosts at the same time.

DynaSOAr evolved from the research of several researchers pursuing similar interests. The involvement in the architectural aspects of the framework is considered as one contribution towards this thesis. Conceptual design, implementation and evaluation of key aspects of the dynamic deployment framework such as the extensive use of *registries* to describe services and resources, adoption of the message oriented model and the handshaking between resources to announce their availability, introduction of the virtualization approach and the use of virtualization technologies, and several approaches to scheduling within the host provider are the other the major contributions towards the thesis.

Chapter 5

Exploiting Dynamic Service Provisioning in DQP

The concepts of Dynamic Service Provisioning as discussed in Chapter 4 could be exploited within the context of a Service-oriented Distributed Query Processor and in this section, the attempt to combine the concepts of DynaSOAr with OGSA-DQP is discussed. In the publicly available version of OGSA-DQP, a consumer can submit queries that access data from data sources which may be geographically distributed and also invoke a remote analysis service on the data within the queries. Experiments, which will be analysed in later sections, show that frequent long running queries suffer from a heavy data access cost over the network when the data sources are remote. During the invocation of a remote analysis service, the invocation cost increases rapidly as the number of tuples retrieved from the database increases or the size of each tuple increases, because of the additional cost of data transportation over the network. It is also a common practice in certain scientific domains to use external services managed by third parties within the distributed query processing framework, which does not guarantee the availability of the service at the time of the query submission and as a result of which the queries may fail. The concept proposed in the Active Information Repository architecture [7] can provide a potential solution to these issues by allowing the evaluation and analysis services to be provisioned closer to the data sources. The combined Distributed Query Processing system builds on the DynaSOAr architecture [19] and allows the on-demand deployment of various services needed for the DQP system.

5.1 Usage Scenarios

Because OGSA-DQP is exposed as a service, and the query evaluation engine created at run-time is a composition of several services, it seemed worthwhile to exploit dynamic deployment features within the context of query evaluation. There can be several scenarios where the concepts of dynamic service provisioning as seen in DynaSOAr could be used to benefit the OGSA-DQP framework. These scenarios are described in the following sections.

5.1.1 Collocating the Query Evaluation Engine with the data

In the original OGSA-DQP, the databases can be located on nodes that are remote from the evaluation nodes where the actual query evaluation processes take place. The *scan* operators within each partition being evaluated on the evaluator nodes access the data from the databases using OGSA-DAI. Accessing data from remote data sources over the network incurs a data access cost which increases as the number of retrieved rows increases or the size of the tuples increases. Provisioning the query evaluation service on the fly on suitable nodes may be used as a solution to this issue. The DQP system would first try to schedule the *scan* operator on the same node as the data set, if possible, which would completely eliminate the cost of transporting the data over the network. Alternatively, scheduling the scan operator on a node closest to the data source may reduce the data access cost. In order to do this, the DQP system must be able to deploy the query evaluation service on the node which is deemed suitable for this based on the network latency with the node on which the data resides.

5.1.2 Collocating the Analysis Service with the data

Many researchers in fields such as Bio-Informatics and Neuro-Informatics execute workflows that retrieve data from a publicly available database, such as the Gene Ontology (GO) database [123] or SwissProt [124], and perform analysis on them. These workflows may involve invocation of a data retrieval service returning a set of tuples, after which the analysis service is invoked on each resulting tuple. In most cases, these analysis services are remote from the actual data sources, and each invocation incurs an additional cost in transporting the data over the network to the resource where the analysis service is deployed. As in case of service-oriented systems, the data propagates over the network as SOAP messages and the overhead increases for larger amounts of data resulting in larger service invocation cost as have been established in [125], [126] and [127]. The data elements

within the message are serialised by the underlying infrastructure which leads to a certain degree of expansion, and in [127], it is claimed that the SOAP data representation is about 10 times the size of the equivalent binary representation. This substantially large size leads to an additional cost during the data transmission, which is also evident from the experiments analysed in [128]. For frequent long running queries that invoke such a remote analysis service, the cost of transporting the data over the network can be eliminated or reduced if a copy of the analysis service can be deployed closer to the data source.

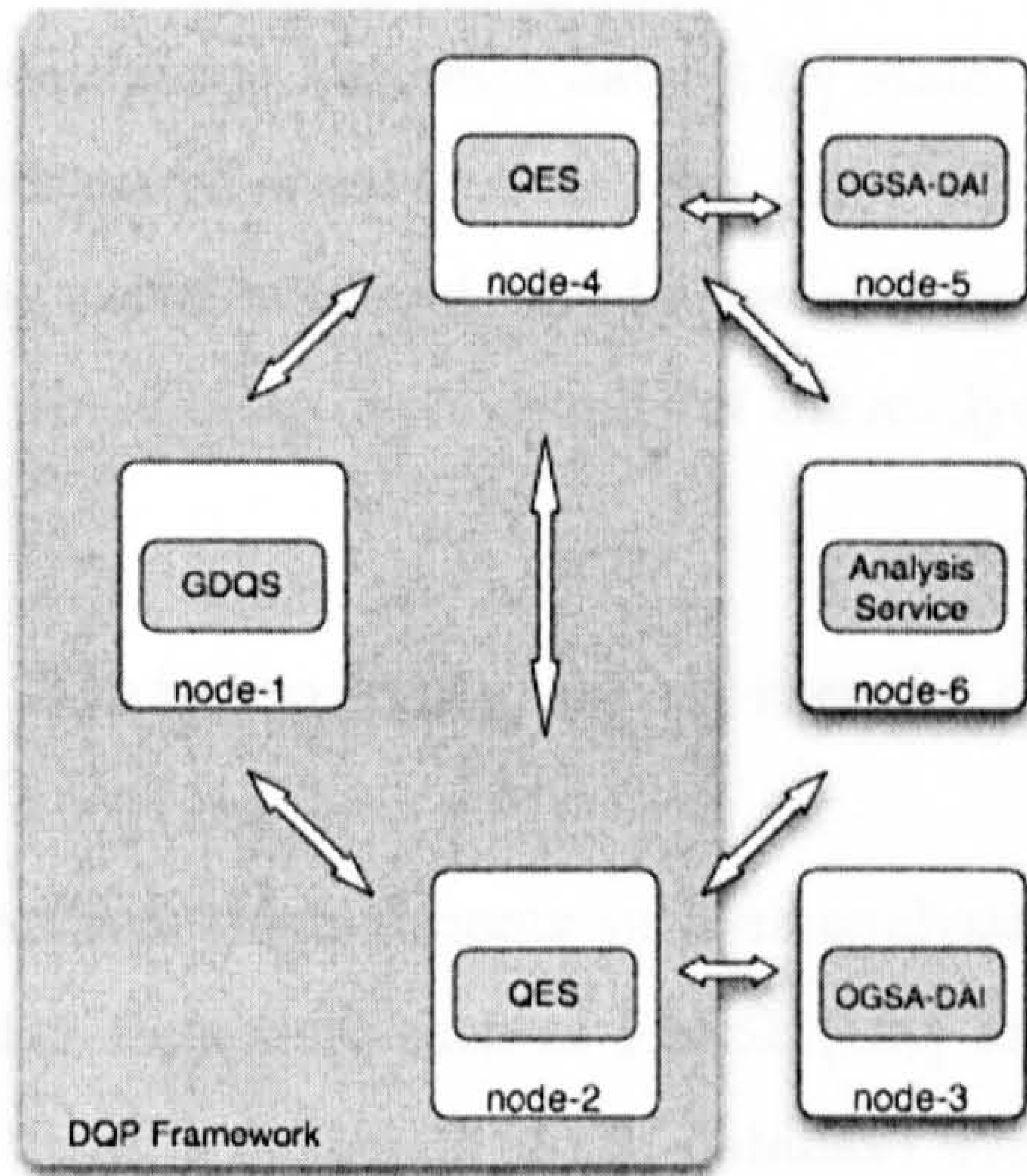
5.1.3 Increased degree of parallelism

It is also possible to deploy multiple copies of the analysis service and parallelise the *operation_call* operator (the physical algebra operator which encapsulates the invocation of the analysis service) by scheduling it on multiple partitions to share the load between several instances. After this, tuples can be routed to different instances of the service by the parallelised *operation_call* operator creating an execution framework similar to the one shown in Figure 5.1(c). In this figure, the analysis service is deployed on each of the available nodes, and thus it is possible to parallelise the *operation_call* operator on partitions running on each of these nodes, and make them invoke different instances of the same service. The redistribution of tuples in such a fashion for queries involving large number of tuples from a database will share the load on each instance thereby increasing the performance. It is to be noted that such a step can be taken for Web Service invocations which are not stateful in nature.

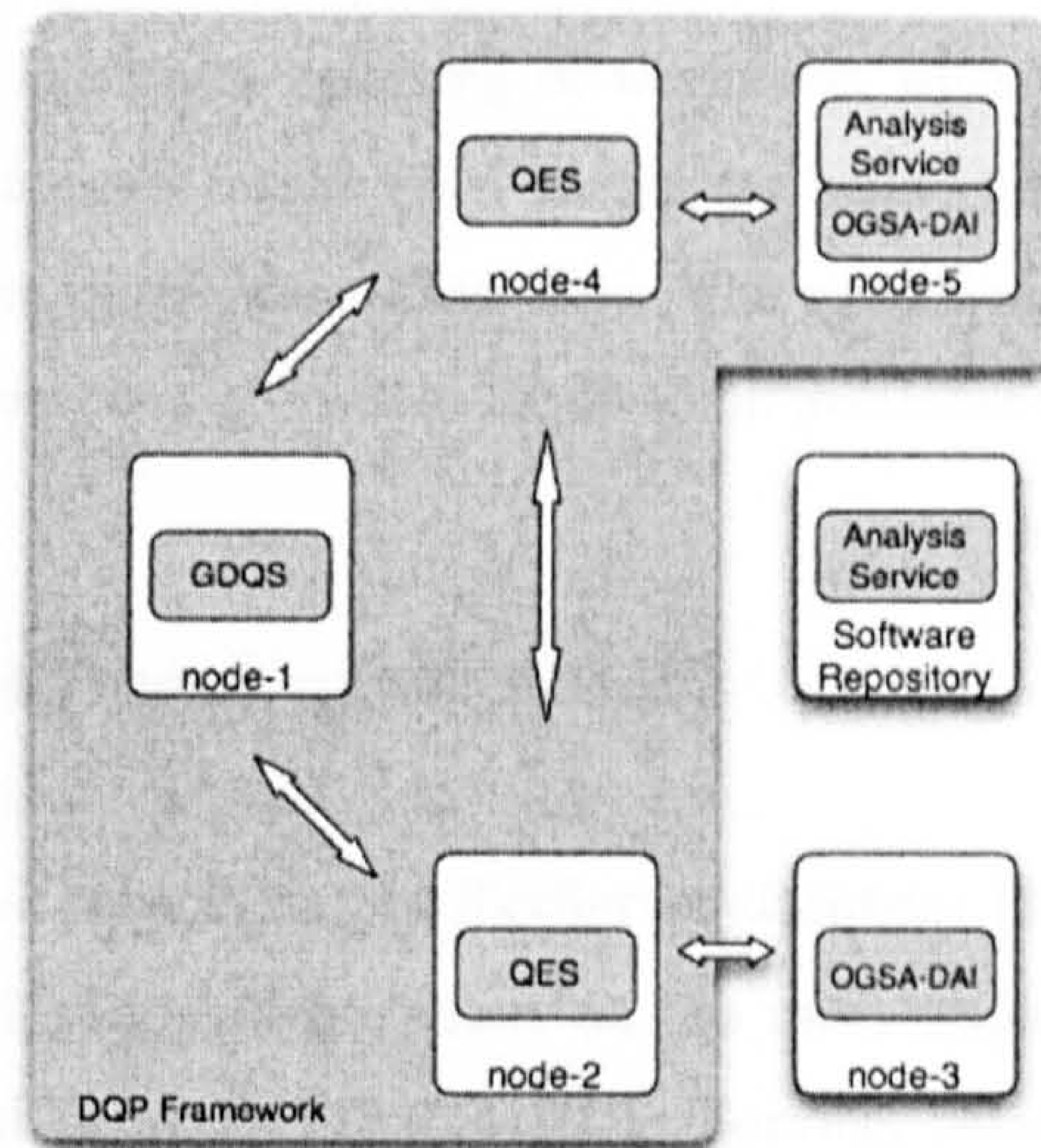
The concepts proposed in Section 5.1.1, 5.1.2 and 5.1.3 are shown in Figure 5.1. Figure 5.1(a) shows a DQP framework where all the data access, analysis and evaluation services are distributed on separate nodes, as in the case of the DQP system described in Chapter 3. An extended framework with the analysis service collocated with the data is shown in Figure 5.1(b). A further extension of the framework with multiple copies of the service deployed on all available nodes is shown in Figure 5.1(c). Finally, Figure 5.1(d) shows the complete dynamic DQP framework where the entire query processing engine is dynamically deployed by collocating both the analysis and evaluation services with the data.

5.1.4 Availability of the third-party maintained Analysis Services

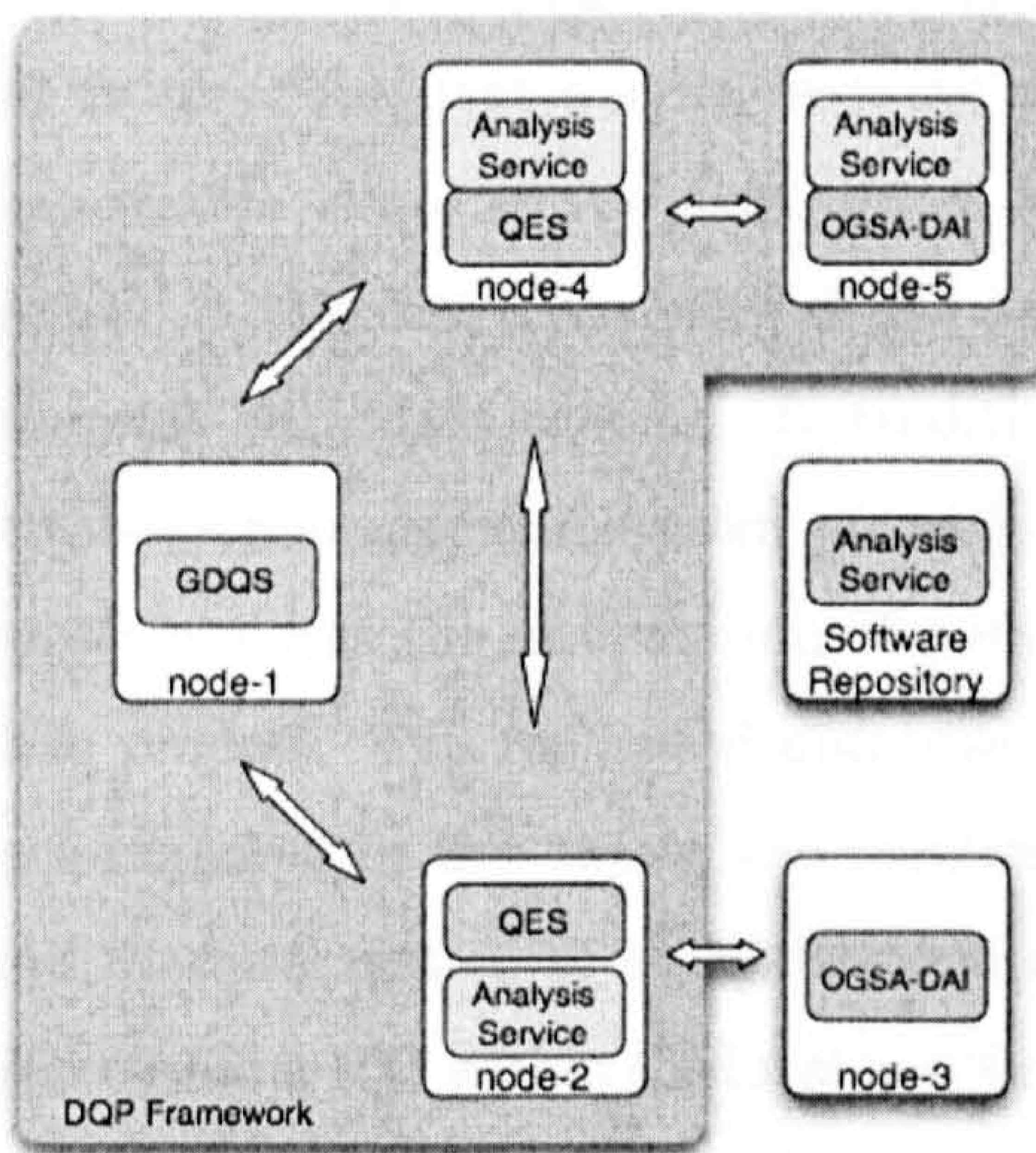
Often, the analysis services invoked during the data analysis are deployed at remote sites and maintained by third parties. This does not guarantee the availability of the service when the query



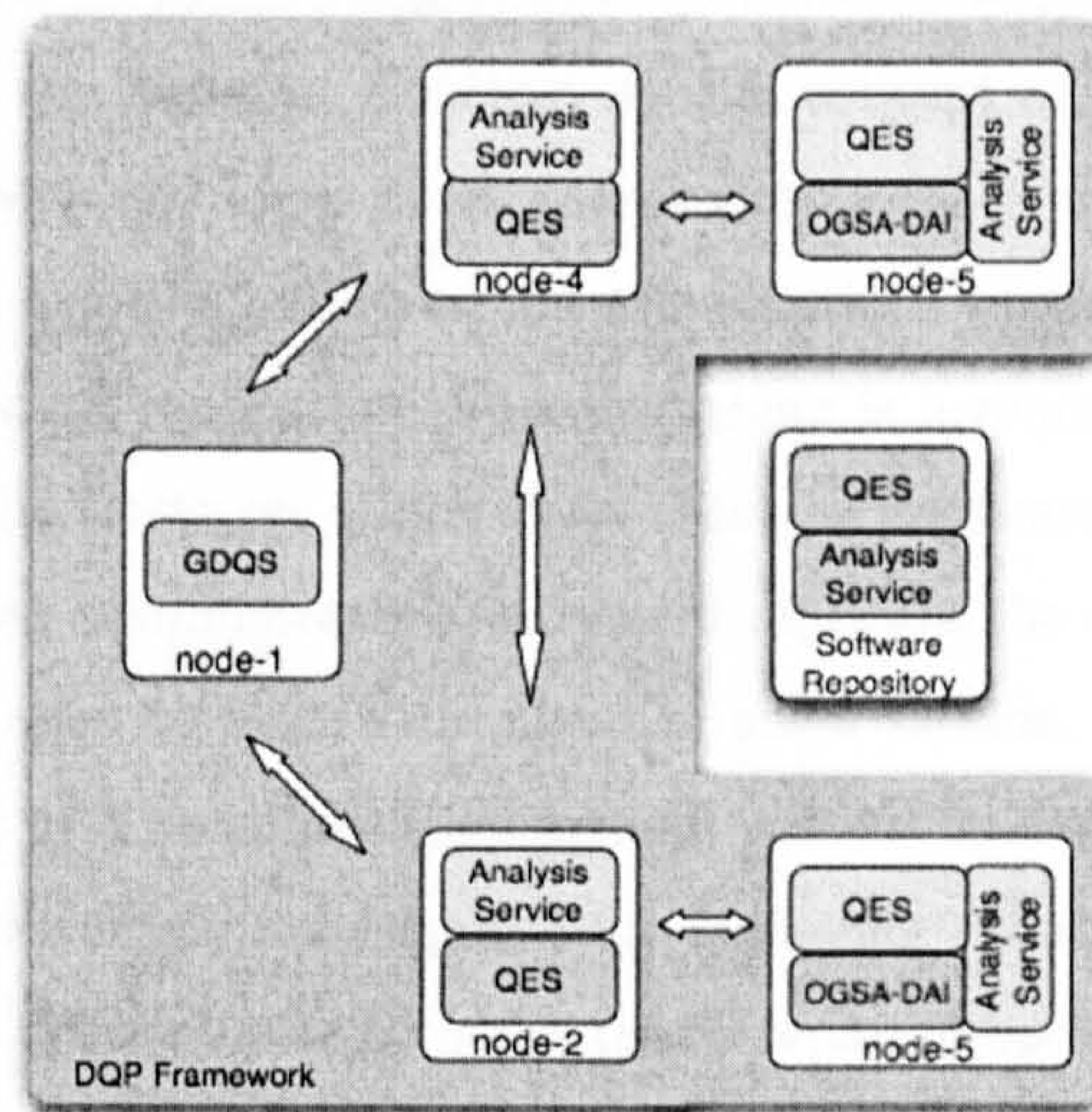
(a) A DQP system with distributed services



(b) A DQP system with analysis service collocated with data



(c) A DQP system with multiple dynamically-deployed analysis services



(d) A DQP system with the query evaluation service and analysis service collocated with data

Figure 5.1: Various configurations of a dynamic DQP framework

is submitted, and as a result of which queries may fail. Apart from the cost of invoking a remote service, the possibility that the service may be unavailable during the actual query execution may often be a cause for concern. It may be possible to avoid such a situation where a query suddenly fails because of a sudden unavailability of the analysis service by deploying a copy of the service on a suitable computation entity available at that point. Investigations into automating *in-silico* experiments using semantic data [129] state that this tight coupling of workflows with particular service instances should be avoided using *service classes* rather than particular instances. In such cases, the workflow would bind to available instances during execution time. The equivalent scenario in DQP would be to deploy copies of the analysis service as and when required on available resources.

5.1.5 Data caching by dynamic deployment of databases

Often, e-Science experiments such as analysing the data from the SkyServer database [23] or bio-informatics databases such as EMBL [130] are not particularly reliant on the most updated data. In such cases, a snapshot of the database wrapped by an OGSA-DAI Data Service, packaged in a virtual machine can be deployed to enable the DQP system to process queries involving the new database. This also provides a functionality similar to the caching of data where this snapshot can be used to serve frequent queries over the same set of data. Contrary to the concept of moving the computation closer to the data, which has been the focus in all recent proposals on dynamic deployment, in this case a snapshot of the data is deployed closer to the computation, which makes it comparable with data caching. In traditional data replication processes, there is a requirement of a database management system on each of the hosts on which data must be replicated, and the process requires the intervention of a database administrator who would replicate the data as an offline process. The replicated database will need to be synchronized in some fashion. In the approach explored in this thesis, the requirement of a database management system on the target host and the intervention of a database administrator are not required for deploying the database packaged within a VM within the local network. It is however to be noted that a separate background process would be required to periodically synchronise the snapshot with the actual data set that was important for the application.

5.1.6 Services requiring special environments

A lot of workflows in the field of Bio-informatics involve specialised services such as Blast (Basic Local Alignment Search Tool)[9], which is a very common gene and protein sequence analysis service. Given a protein or a gene sequence, this service can identify similarities of the sequence with those

stored in a database. This service requires a special environment such as a set of libraries and a database, all of which together can be encapsulated within a virtual machine, and can be deployed as and when required. Further, scientific applications often are tuned with the host on which they are installed. The tuning process is manual and the performance depends heavily on a proper tuning, which in turn depends on the processor architecture, available memory, disk space etc. It may take a considerable amount of time to perform this tuning on the hosts on which the application are installed, and it is impossible to automate this process. In such cases, a flexible alternative approach may be to install the application or service on a virtual machine and tune it beforehand. This virtual machine can then be stored within the repository, and when it is deployed and started, the application, which has already been tuned will perform normally.

The current query compiler/optimiser in OGSA-DQP performs some basic optimisation based on the information available to it. But this optimisation can be enhanced by considering the dynamic deployment scenario, as outlined in the use-cases mentioned above, where the scheduler would be able to schedule the deployment of new evaluation and/or analysis services on new computational resources thereby allowing the query processing framework to capitalise on advantages offered by a dynamic deployment framework.

5.2 Towards a Dynamic Distributed Query Processor

In this section, the functional architecture of the dynamic distributed query processor is discussed.

5.2.1 Overview

The extended version of the Distributed Query Processing system incorporates the dynamic service provisioning concepts from DynaSOAr, and in the process, uses certain components from the DynaSOAr framework, although there are certain deviations from the concepts proposed in DynaSOAr. For example, in DynaSOAr, the request from the consumer for invocation of a service is forwarded without any modification to the message body to the service via the HostProvider once the service is deployed on an available host, and the endpoint of the actual service is only known at the corresponding HostProvider. The forwarding of the message normally happens in a way that is transparent from the user, although, there are options where the consumer is able to provide preferences about the provider, which is a semi-transparent approach. In DQP, the evaluation services need to interact among themselves for evaluating the corresponding partitions and hence the end-

points of the dynamically deployed evaluators are known at the GDQS or the coordinator which is comparable to the DynaSOAr Web Service Provider, although for a consumer, this stays completely transparent. Other components, such as the Software Repository, the Registry Service and the Host Provider service are incorporated into the extended DQP architecture.

The resulting architecture described in detail later in this section, adds a completely new dimension to the OGSA-DQP system. In the earlier DQP system, the data, evaluation and analysis services were tightly coupled with the available computational nodes. It required a pre-configured set of resources with all the necessary components, such as the data services, evaluation and analysis services deployed on those resources (as shown in Figure 5.2), thereby limiting the scope for exploiting the inherent dynamism of a Grid-like environment.

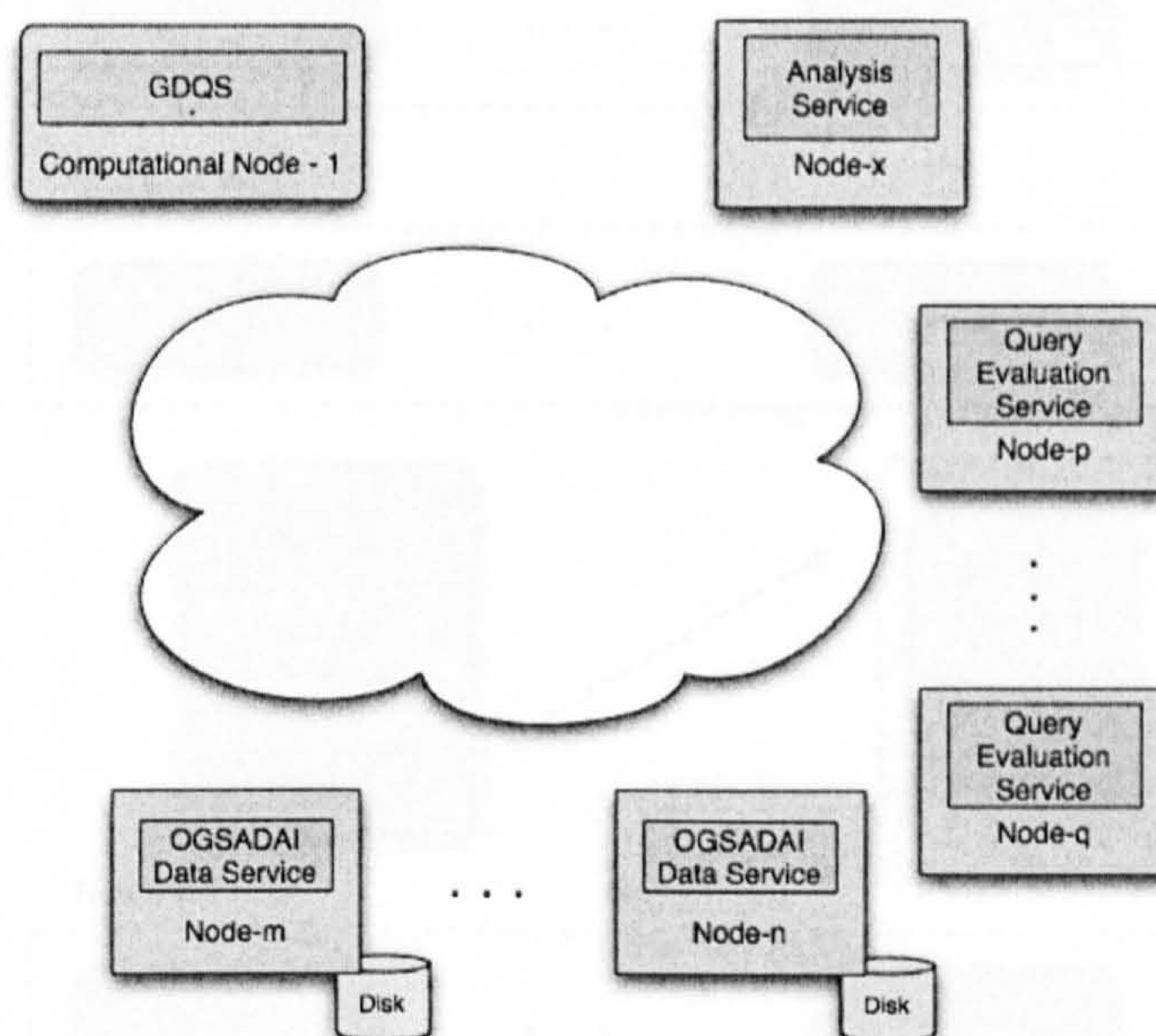


Figure 5.2: Overview of the static DQP system

The modified version of DQP decouples the software components from the available computational and data resources. The system may have a set of pre-defined data resources, but apart from that, all that it requires is a set of hosts which support the DynaSOAr framework. It can be assumed that as the dynamic deployment features are incorporated within the standard containers, the requirement of this additional framework will be eliminated, thereby allowing DQP to take the full advantage of these features embedded inside the containers. In this extended version of DQP, there are pools of services, virtual machines and resources (Figure 5.3). The software pool consists of services such as the evaluation and analysis services, the virtual machine pool may contain virtual machine instances containing a database snapshot and/or specialised analysis services which

require a special environment, and the pool of resources contain a set of computational nodes which can be utilised during query processing. Conceptually, the DQP system in this case "creates" each node by combining a number of components from the pools. This feature also opens up the possibility of using the *Software Marketplace* as mentioned in Section 4.3.5 and [87] where, based on the preferences (such as preference to a particular provider, for example the Blast Service from the European Bioinformatics Institute (EBI), instead of the one from the National Center for Biotechnology Information (NCBI)) or quality of service specifications (such as cost, throughput, reliability etc) from the consumer, the DQP system would select a certain version of the analysis or evaluation service from a list of all the available services.

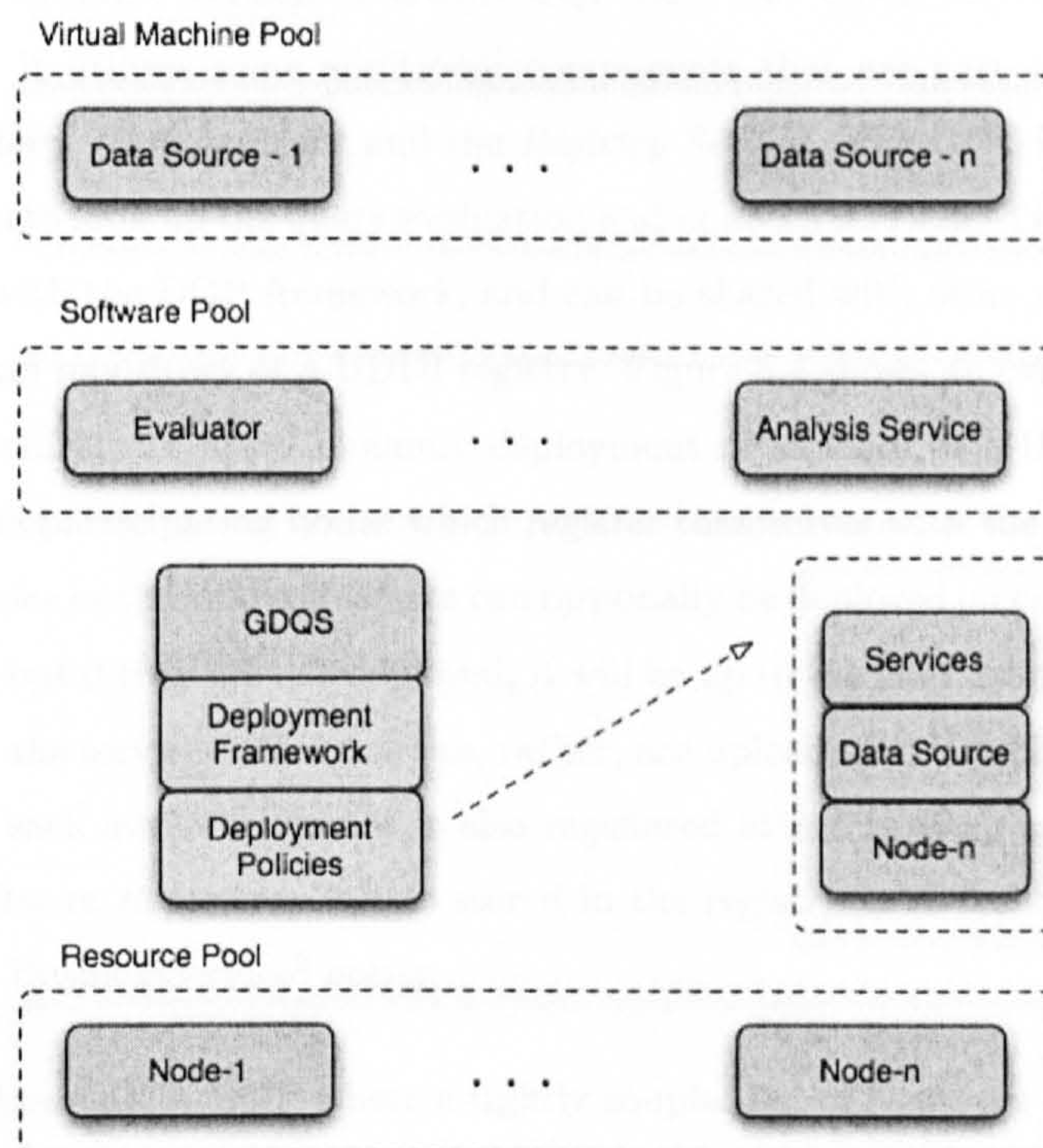


Figure 5.3: Overview of the Dynamic DQP system

In Section 3.1, OGSA-DQP has been considered as an approach that is complimentary to other service orchestration mechanisms, such as workflow execution systems. Most traditional workflow engines, for example Taverna [10], work with a centralised coordinator system which is better known as "centralised enactment" and can be compared with the hub-and-spoke concept where the coordinator is responsible for all communication between the services. OGSA-DQP, in the original form, is in some ways comparable to this, where the query evaluation processes are initiated only on the available evaluation nodes. The dynamic version of OGSA-DQP is more closely related to the distributed workflow concept, such as DECS [131]. DECS allows "decentralised enactment" where the

coordination is distributed amongst multiple enactors who communicate as peers during the enactment process, thereby creating a fluid workflow enactment system which is not tightly coupled with a centralised enactment engine. In the dynamic version of OGSA-DQP, the query evaluation engine is not tightly coupled with the already available evaluation nodes, but is fluid in nature, where new nodes can be allocated to the process of query evaluation based on the run-time situations.

5.2.2 Architecture

Architecturally, the dynamic version of OGSA-DQP does not differ very much from the earlier version, except that it utilises some additional components that are part of DynaSOAr, such as the *Software Repository*, *Host Provider* and the *Registry Service*. The GDQS has knowledge of the registry where it tries to look up the query evaluation and analysis services. These new services again are loosely coupled with the DQP framework, and can be shared with other services or frameworks that require a software repository or a UDDI registry. Figure 5.4 shows an overview of the extended OGSA-DQP architecture. To allow dynamic deployment of services, the HostProvider service is made available on the participating nodes which register themselves with the Registry Service. The query evaluation service or the analysis service can optionally be deployed on computational resources and the data nodes - but if they are not deployed, it will be up to the DQP framework to decide when and where to deploy the services. The services, rather, are uploaded to the *Software Repository* and during this process, each available service is also registered at the registry as a deployable service. The URL of the *Software Repository* is also stored in the registry as a TechnicalModel or tModel[1] reference within the BusinessService[2] entity.

Thus unlike the original OGSA-DQP where a tightly coupled set of resources are used with the data access, evaluation and analysis services pre-deployed on them, the extended version tries to exploit the dynamic deployment features by using only a collection of data hosts, and host providers which allow services to be deployed on them dynamically. In the latter case, the distributed query processing system will have the option of making a choice between the services to deploy, either based on the user preferences, or the quality of service parameters, or even predefined service level agreements. What this dynamic version of DQP provides is an additional framework within the existing DQP system which is capable of exploiting the dynamic deployment features thereby incorporating

[1] Technical Models, or tModels for short, are used in UDDI to represent unique concepts or constructs. They provide a structure that allows re-use and, thus, standardisation within a software framework. The UDDI information model is based on this notion of shared specifications and uses tModels to engender this behaviour. For this reason, tModels exist outside the parent-child containment relationships between the businessEntity, businessService and bindingTemplate structures. [81]

[2] Each businessService is the logical child of a single businessEntity. Each businessService contains descriptive information again, names, descriptions and classification information – outlining the purpose of the individual Web services found within it. [81]

a flexibility to the system which was absent in the earlier versions.

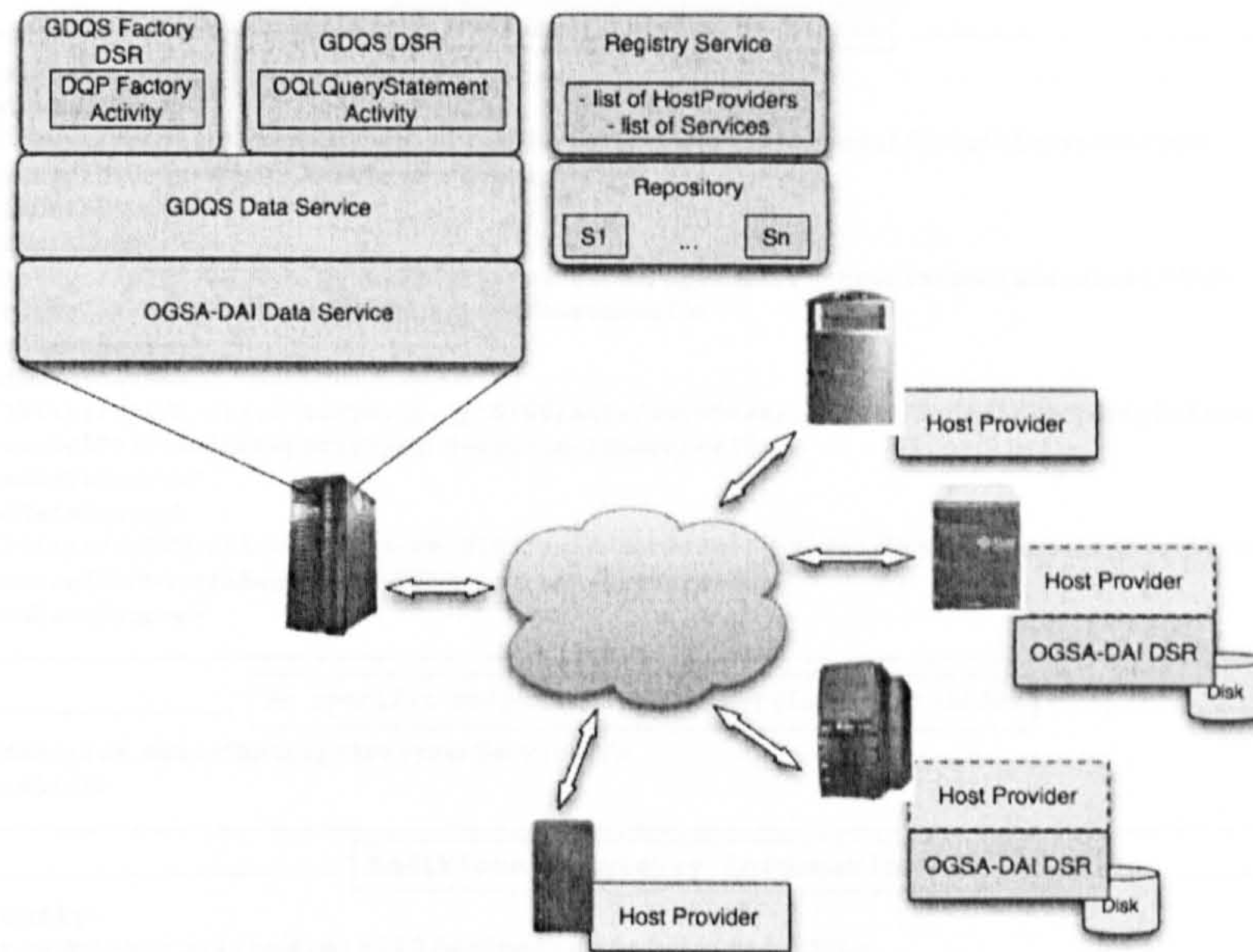


Figure 5.4: Basic DQP Architecture

5.2.3 Setting up the Distributed Query Processor

In this dynamic version of OGSA-DQP, the configuration document submitted by the client specifying the data and analysis resources to be used for the distributed query does not mandate the list of evaluation services to be used. The only requirement is a list of data sources on which the query will be executed and optional registry information identifying the registry which will be looked up for the availability of computational resources and services. The XML fragment in Listing 5.1 is similar to the one shown in Listing 3.2 and shows the canonical form in which the data and analysis resources are specified by the client, the difference being in the additional registry information, and the fact that the list of evaluation services are not explicitly provided, and only the name of the analysis service is provided, giving the GDQS the freedom to deploy these services as and when necessary on available computational resources, thereby creating an ad-hoc grid-like environment. If there are available resources with the query evaluation service already deployed on them, then those can also be included in the configuration file as before, in which case the compiler/optimiser will consider these nodes along with the other available resources which are looked up from the registry.

Once the configuration document is received by the GDQS Factory Data Service, the usual steps of importing the schema and metadata from the data services (as outlined in Section 3.3) are performed.

Listing 5.1 Configuration Document for Dynamic OGSA-DQP

```
<DQPConfiguration xmlns="http://uk.org.ogsadai/dqp/configuration">

                          [No evaluator service is listed]
    <DataResourceList>
        <ImportedDataSource>
            <URI>http://pl01.cs.washington.edu:8199/axis/services/ogsadai/GoDataService</URI>
            <ResourceID>GoTermMySQLResource</ResourceID>
        </ImportedDataSource>
        <ImportedDataSource>
            <URI>http://pl02.cs.tcd.ie:8199/axis/services/ogsadai/ProteinTermDataService</URI>
            <ResourceID>ProteinTermMySQLResource</ResourceID>
        </ImportedDataSource>
        <ImportedDataSource>
            <URI>http://pl01.iii.u-tokyo.ac.jp:8199/axis/services/ogsadai/ProteinPropertyDataService</URI>
            <ResourceID>ProteinPropertyMySQLResource</ResourceID>
        </ImportedDataSource>
        <ImportedDataSource>
            <URI>http://pl02.csl.utoronto.ca:8199/axis/services/ogsadai/ProteinSequenceDataService</URI>
            <ResourceID>ProteinSequenceMySQLResource</ResourceID>
        </ImportedDataSource>

                  [No specific endpoint for the service is provided]
        <ImportedService name="EntropyAnalyserService"/>
    </DataResourceList>

                       [Additional registry information]
    <RegistryConfig>
        <registryURL>http://budle:8090/grimoires/</registryURL>
        <publishURL>http://budle:8090/grimoires/services/publish</publishURL>
        <inquiryURL>http://budle:8090/grimoires/services/inquire</inquiryURL>
        <transportClassName>ApacheAxisTransport</transportClassName>
    </RegistryConfig>

</DQPConfiguration>
```

Additionally, the GDQS also collects an estimate of the network latency time between the available resources (which it looks up from the registry) and the actual data sources by sending and receiving a pre-calculated packet. It should be noted that this process of gathering the network data is not the most perfect way, but it can nevertheless produce an estimate. It would definitely be more accurate if other standard network monitoring tools are used, and the DQP system does not disallow such tools to be used. This estimation of the network latency can later be used during the query optimisation phase to schedule data access and operation call operators to specific hosts which are closest to the data source.

5.2.4 Proactive deployment of the Analysis Service

The configuration document provided by the consumer may optionally include the name of an analysis service that will be used in the queries. The GDQS takes a proactive decision to deploy that service on all available resources during the schema import phase if that service is registered in the registry and can be found in the *Software Repository*. This allows the GDQS to consider multiple instances of the analysis service while optimising the query thereby parallelising the operation call

operator with a view to maximise the performance. GDQS uses multiple threads to process the deployment on each available host, and hence the cost of deploying the analysis service on multiple nodes is almost equivalent to that of the maximum cost amongst all these. Once the analysis service has been deployed on multiple nodes, the GDQS keeps a record of all the endpoints as *replicated operations* and considers them together with the network latency data to determine which of the deployed instances should be used during the query processing. Thus a costly operation call invoking the analysis service may be included in multiple partitions, each with the analysis service hosted on the corresponding node, so that the tuples on which the service will be invoked can be distributed to all these partitions thereby executing it in parallel. DQP uses a heuristical approach to decide on the *degree of parallelism* of expensive operators. For an *operation_call* operator this is equivalent to the number of deployments of the service. Thus, DQP is able to incorporate all the endpoints of the analysis service that has been dynamically deployed to generate query plans where the *operation_call* operator is fully parallelised.

5.2.5 Distributed Query Plan Generation

Unlike the previous OGSA-DQP where evaluation services were required to be already deployed on the participating nodes, there is no such requirement except that the participating nodes must allow the deployment of new services, which is achieved via the HostProvider. After the schema is imported, a query is submitted by the consumer. During the compilation phase of the query, the GDQS collects the list of available computational resources from the configuration file (if it contains such a list) and the registry, where all the HostProviders are registered, as well as all previously deployed evaluation services. It is also possible to add certain characteristics of the participating nodes, such as the CPU speed, amount of available memory etc. within the configuration document as *Computational Metadata*. These can also be stored as the metadata for each HostProvider service within the registry. The GDQS keeps a record of the available computational metadata with the list of available resources. After the query compilation and optimisation phase, a physical query plan is produced which is then partitioned by the *scheduler* component. This component is responsible for parallelising the physical plan into several partitions and assigning operators to the available computational resources which can perform the query evaluation. The scheduler uses certain heuristics for introducing intra-operator parallelism to some physical operators such as *join* and *operation_call*. As a first step, *exchange* operators are introduced in the query plan before any attribute-sensitive (such as *join*) and location-sensitive (such as *operation_call*) operators which is described in Section 3.4. Thereafter, the first phase of the scheduling process follows the algorithm shown in Listing 5.2 for assigning the degree of parallelism to each operator [132].

Listing 5.2 Assigning degree of parallelism to operators

```
1 repeat
2     get costliest parallelisable operator;
3     if more parallelism is beneficial
4     repeat
5         increase degree of parallelism for operator
6         check if more parallelism is beneficial
7     until no changes in parallelism OR no more available resources
8 until no changes in costliest operator
```

The second phase of the algorithm assigns the operators to specific computational resources. The complete physical plan is subdivided into partitions bounded by the *exchange* operators. If a partition does not contain any parallelised operators, the scheduler will attempt to assign it to a specific computational node. On the other hand, if any operator within a partition is parallelised, the other operators in that partition (such as the *reduce* operator) are also parallelised to the same degree and a number of partitions equivalent to the degree of parallelism will be created, each containing the identical query plan fragment. Assignment of partitions to computational resources consider the computational metadata that describes the characteristics of each node, and the most recent network latency information for each available node. The scheduler uses some heuristics while assigning operators to resources, which are as follows:

- If a partition contains a data access operator, such as a *scan* operator, the scheduler assigns them first, and makes an attempt to place it on the same node as the OGSA-DAI data source being accessed. If that node has an evaluation service already deployed, then the endpoint of that service is used, otherwise, the node is assigned only if it allows dynamic deployment of an evaluation service. If it is not possible to assign that node to a partition, because no evaluation service has been deployed on it and dynamic deployment is not possible, the scheduler assigns the partition to the node closest to the data source and hosting a HostProvider service, irrespective of whether an evaluation service is already deployed or not.[3].
- The partitions which contain a *join* operator that is not parallelised, are placed on the same node as the larger input (the *scan* operator), the assignment of which would have already marked the node as requiring dynamic deployment of an evaluation service.
- The partitions which contain parallelised *hash join* operators are placed as the first option on the node which has the larger input, followed by the node containing the second input, followed by other nodes with a preference given to powerful machines (higher CPU speed) with larger memory, in each case checking whether an evaluation service has already been deployed on

[3]The term "node closest to the data source" denotes the node which has the least network latency with the one on which data is located

the nodes, and if not, using a node which hosts the HostProvider service thereby designating them for a dynamic deployment.

- For the partitions which contain parallelised *hash loop join* operators, first preference is given to the node which contain the larger input, followed by the node containing the OGSA-DAI data source being accessed or the node closest to it, followed by other nodes with a preference given to machines with large memory. The CPU speed is not considered as the primary factor as the join operator spends a large proportion of its time waiting for responses from the data service. During each assignment, the availability of the evaluation service or feasibility of a dynamic deployment of the evaluator is verified.
- The partitions containing parallelised *operation call* operators are assigned on the basis of least network latency followed fastest machine first. Thus, if there are nodes on which the analysis service was dynamically deployed, the scheduler will choose to assign the operators on them to reduce the network latency, and thus the invocation cost.

Based on these criteria and the algorithm in Listing 5.3, the scheduler assigns the operators to specific computational resources and generates a set of sub-plans, each designated for a computational resource. As all available resources are considered in this dynamic version of OGSA-DQP during the query compilation/optimisation phase, the selected resources may not have the services required to evaluate the partition scheduled for the node. In such cases, the dynamic deployment framework will send a deployment request to the corresponding node, which will download and deploy the required service, and also update the registry about this new deployment, so that the GDQS can consider these new service instances for subsequent queries without any need for a dynamic deployment. Once the deployment is successful, the query partitions are submitted to each of the evaluation nodes, and query evaluation proceeds as explained in Section 3.5. Figure 5.5 shows the query processing activity in a DQP framework where the concepts proposed in Sections 5.1.1, 5.1.2 and 5.1.3 are brought together.

The figure follows the same sequence as the earlier Figure 3.5. In step 1, the client submits a query to the DQP coordinator service, which compiles and optimises the query and generates a set of query partitions which can be evaluated in parallel. During the compilation/optmisation phase, the coordinator service takes into account all available hosts rather than the hosts which already have the evaluation service deployed on them. The query plan thus generated may contain a number of hosts which are best suited for the evaluation, but do not have the evaluation service pre-deployed, in which case, the GDQS dynamically deploys the evaluation service on these nodes (step 2). Once the deployment phase completes, the query partitions are sent to each participating node (step 3), following which the query execution process starts on each of the nodes as described before in

Listing 5.3 Assigning evaluators to partitions

```
1  discover the set of available evaluators E = {e1, e2, ...en}
2  discover the set of available host providers H = {h1, h2, ...hn}
3  Set deployment_required D = {}
4  for each operator
5  do
6      if (operator == scan) {
7          ip = get data source IP(operator)
8          ComputeNode node = find nearest node(ip, E, H)
9          if (node is in H) {
10             D = D U {node}
11         }
12     } else if (operator == hashjoin) {
13         l = get left input(operator)
14         r = get right input(operator)
15         ComputeNode node = find join node(l, r, E, H)
16         if (node is in H) {
17             D = D U {node}
18         }
19     } else if (operator == hashloop) {
20         l = get left input(operator)
21         ip = get data source IP(operator)
22         ComputeNode node = find join node(l, ip, E, H)
23         if (node is in H) {
24             D = D U {node}
25         }
26     } else if (operator == opcall) {
27         d = get degree of parallelism(operator)
28         Collection nodes = find opcall nodes(d, E, H)
29         for each n in nodes {
30             if (n is in H) {
31                 D = D U {n}
32             }
33         }
34     }
35 end
36 if (D not empty) {
37     for each d in D
38     do
39         deploy evaluation service on d
40         E = E U {d}
41         update registry with evaluator instance d
42     end
43 }
```

Section 3.5. Each node communicate with each other by sending and receiving partial results in form of tuples and once the execution is completed, the final result is returned in step 8 to the coordinator service and hence the client.

The services that are dynamically deployed remain in place unless specifically undeployed. Thus, queries submitted to the same DQP data resource will be able to use all the available service instances as the configuration of the data resource itself is updated as and when dynamic deployments take place. On the other hand, as the newly deployed service instances are reflected in the service registry, new resources created from the DQP factory data resource for the same set of data sources will also be able to utilise all the available services, and may not need to deploy new services.

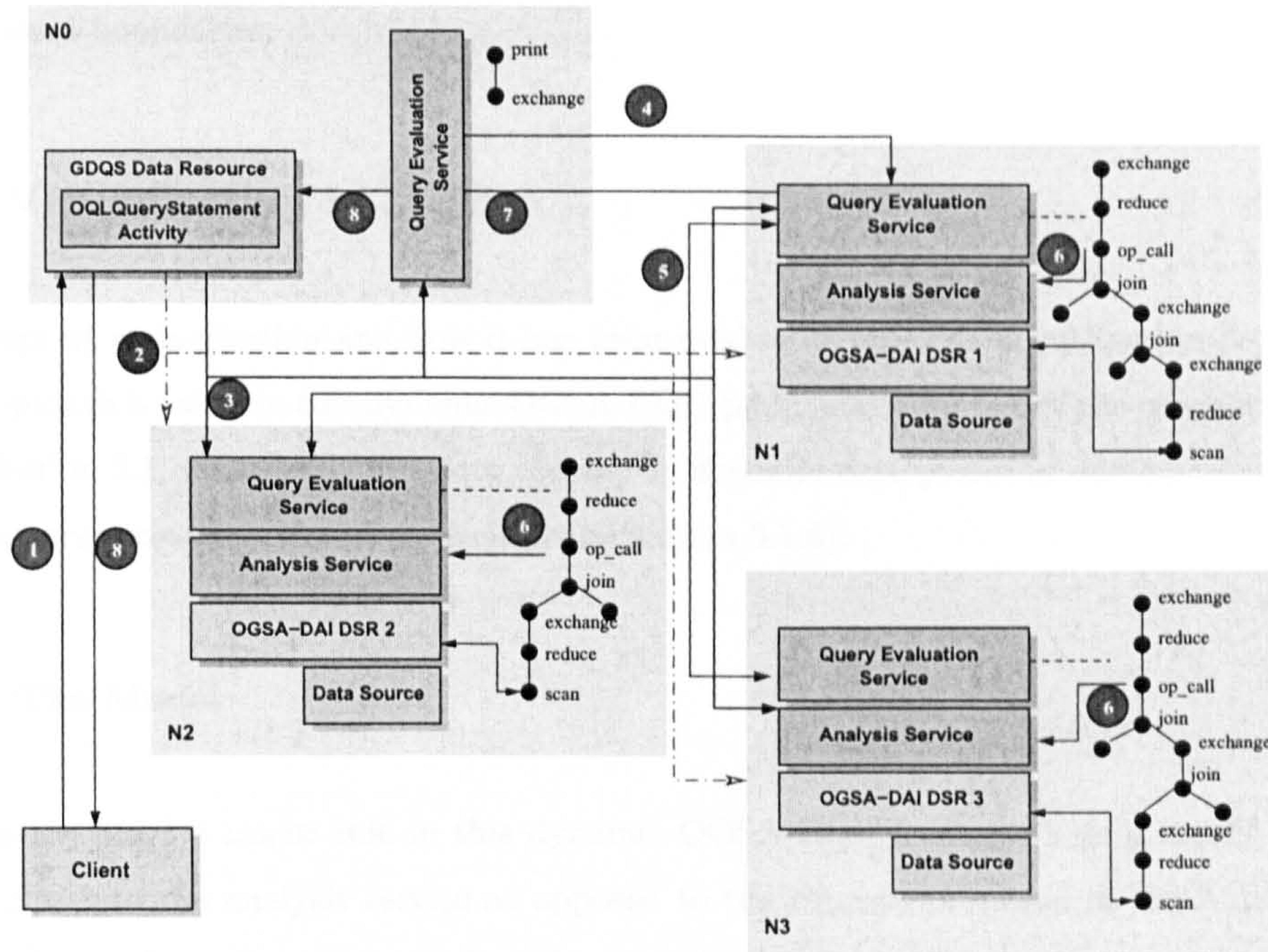


Figure 5.5: Query Execution on Component Services in a Dynamic OGSA-DQP framework

5.2.6 Using Network-aware Cost Models

There is some existing work on cost models for a distributed query processing system considering the network characteristics, such as [133] and [134]. In [133], the authors discuss a concept which utilises a "network graph" for distributed query processing, where the transmission cost between the participating nodes on which data are hosted are used to create the network graph from which least costly routes can be derived using well-known algorithms such as the shortest path algorithm. In [134] new algorithms such as Edge and Edge+ are discussed which can be used for placement of the operators on arbitrary network locations given the availability of the framework which allow such placement. The concepts of DynaSOAr allows the query evaluation engine to be flexible in nature. The network aware cost models are capable of taking better decisions as to where the operators should be placed based on the network data. It can also be envisaged that the real network data would be collected using network monitoring tools, such as CoMon [135] which is a scalable monitoring tool for the PlanetLab system [136]. Bringing all these together, we can envisage a distributed query processing framework which will be fully dynamic, network-aware and flexible, where the query compiler/optimiser would use adaptations of network-aware cost models for making decisions about the assignment of the operators to computational resources after which the dynamic deployment framework will allow it to deploy necessary components on the selected nodes thereby creating an ad-hoc runtime evaluation engine able to exploit computational and data resources across

organisational boundaries.

5.2.7 Virtualization in DQP

The concept of *virtualization* and how it has been used in DynaSOAr is outlined in Section 4.4. A similar approach is taken in this dynamic OGSA-DQP, particularly for two of the usage scenarios outlined in Section 5.1, viz. to perform *data caching by dynamic deployment of databases*(Section 5.1.5) and for *services requiring special environments*(Section 5.1.6).

5.2.7.1 The Model

Virtualization plays a major role in this dynamic OGSA-DQP framework as a means of bringing the data closer to the analysis service as opposed to the concept proposed in the Active Information Repository of moving the analysis code closer to the data. The primary scenario where this alternative approach may be useful is where a consumer, for example a bio-informatician, submits frequent queries over the same dataset, where each query extracts a large number of tuples from one or more databases, performs certain operations, such as *join* on them, and for each matching tuple, invokes an analysis service. The DQP query compiler initially attempts to assign the *join* operator on the node which has the largest input as the first preference, followed by the node with the second input, and then other available nodes. But this approach may still give rise to a situation where a large number of tuples have to be transferred from a distant node to the root evaluator (which is collocated with the co-ordinator). As DQP relies on SOAP messages for transferring the tuples, the transport cost incurred in such situations may become extremely high. If it can be established that the consumer is submitting the queries over the same data set, and if the data set does not need to be the most up to date copy, it may be possible to take the alternative approach of deploying a snapshot of the database closer to the computation to avoid the large transport costs.

Database Replication is often considered as a solution to manage distributed databases and bringing the data closer to the computation. But the technology requires a long offline administrative process of setting up the master and slave databases after which the slave databases can synchronise themselves with the master copy. Thus it is evident that *database replication* alone can not be used to deal with the problems as mentioned above. In the dynamic version of OGSA-DQP, *virtualization* is viewed as one of the potential solutions. As it has been assumed that the requirement does not mandate the use of the most recent data set, in this extended DQP, a snapshot of the database is deployed in a pre-built *virtual machine*, along with the necessary data access services to access the

data. This pre-built VM is stored in the *software repository* and the DQP system is able to deploy it when required.

Virtual Servers from VMWare [120] have been used in DynaSOAr, and hence in this dynamic version of DQP. As in the case of DynaSOAr, the available HostProviders are registered so that the service provider, in this case the DQP coordinator can discover them from one or more registries. In order to be able to use the virtualization features, some hosts with an installed version of the VMWare Server are registered as HostProviders which are stored in the registry as individual entities. The registry contains the information about the VMWare Server, such as the location of the executables and the perl scripts which wrap the VMWare Server commands. When required, the service provider would be able to search the registry for hosts with the VMWare Server, and request the HostProvider service on that host to download a particular virtual machine and start it, which, in effect will make all the services deployed within that VM available. A "bridged" networking model is followed within DQP for such cases, as the services deployed on the VM, such as the evaluation service must communicate with external services, such as remote evaluators on other participating hosts, thus highlighting the requirement of the "network presence" of the VM to be known. For this reason, a pool of IP addresses is used to assign each new VM a unique IP address when the VM is booted, so that they can communicate with external systems. Each service deployed on the VM will be accessible using this assigned IP address within the service endpoint.

5.2.7.2 The Feedback Methodology

In DQP, all services are deployed when required, and the same philosophy is followed for the deployment of a database snapshot. The other dynamic deployment options, such as moving the analysis code closer to the data and collocating the query evaluation engine with the data are exploited during the query compilation/optimisation phase, but for caching the data or deploying the data closer to the computation, a different methodology of feedback is used. This is because deploying a database wrapped in a VM is costly, and is done only when it is a necessity. Several performance measurements, such as number of tuples transferred, total cost of transferring the data, the total cost of the actual evaluation process etc. are collected from each of the participating evaluation services and are sent back to the central DQP coordinator (GDQS) after the completion of each query evaluation process. An XML schema as shown in Figure 5.6 is used for collecting the performance data from each node.

The performance monitoring schema is used to collect three distinct types of measurements - (i) measurements related to the data access process, such as the total cost of accessing the data, the

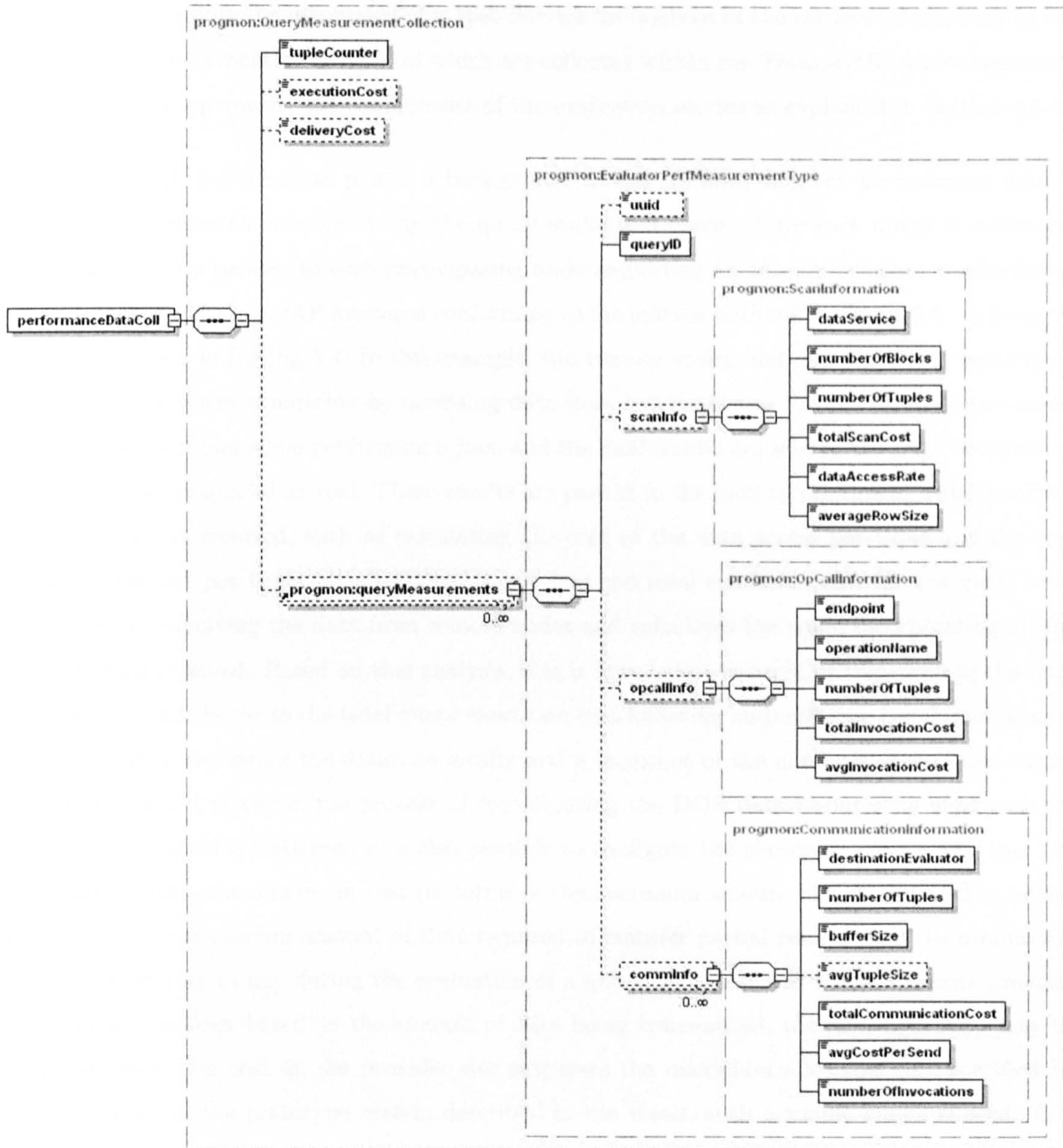


Figure 5.6: XML Schema for collecting performance data

data access rate, number of tuples, and average row size for each tuple, all of which are collected within the scan operator, (ii) measurements related to the data transfer process, such as the average row size of the tuples which are sent across the network, the total number of tuples that are sent, the cost of each transport operation and the total cost of transferring all the tuples, and (iii) similar information regarding the invocation of a Web Service for analysis of the retrieved data, such as the total and average invocation cost, all of which are collected within the *TransportHandler* component and the relevant operators, the components of the evaluation service as explained in Section 3.5.5.

During the DQP initialisation phase, a background thread for analysing the performance data is started, which remains inactive during the query evaluation phase. After each query is evaluated, the GDQS sends a request to each participating node requesting for the performance results which are sent back in form of SOAP messages conforming to the schema outlined in Figure 5.6, an example of which is shown in Listing 5.4. In this example, two remote nodes, *salt* and *planetlab4* participate in a distributed query evaluation by accessing data from two databases located on these two nodes and exchanging tuples while performing a *join*, and the final results are sent to the DQP coordinator node, which is designated as *root*. These results are passed to the monitoring thread which analyses the results it has received, such as calculating the cost of the data access per tuple and the cost of data transport per tuple. It also tries to correlate the total execution cost for the query with the cost of transporting the data from remote nodes and calculates the trend incorporating all the previous data received. Based on this analysis, if it is found that the cost of transporting the data is the major contributor to the total query execution cost following an increasing trend, and is more than the cost of deploying the database locally and a snapshot of the database is available within a pre-built virtual machine, the process of reconfiguring the DQP data resource by deploying the VM on a local node is initiated. It is also possible to configure the process in such a way that the consumer specifies the maximum cost (in terms of the maximum amount of time required to invoke a service, or the maximum amount of time required to transfer partial results from the evaluators) he or she is willing to pay during the evaluation of a query. In a scenario where a remote provider charges the consumer based on the amount of data being transmitted, the reconfiguration may be triggered when this cost at the provider site surpasses the maximum allowable cost specified by the consumer. In the prototype system described in the thesis, such a simple model is used. It is possible to use more complex analysis model based on data access costs, network monitoring tools for available bandwidth and real transport cost to decide the stage when the reconfiguration process should be triggered.

Listing 5.4 Example of performance data from two remote nodes

```
<progmon:QueryMeasurementCollection queryID="dqpogsadai-11492de1f50.Thu-Aug-23-14:21:44-BST-2007">
```

data reflecting the total execution cost

```
    <progmon:tupleCounter>4000</progmon:tupleCounter>
    <progmon:executionCost>51249</progmon:executionCost>
    <progmon:deliveryCost>812</progmon:deliveryCost>
```

Cost from an individual evaluator on SALT

```
<progmon:queryMeasurements sourceID="http://salt:8199/EvaluationService">
    <progmon:queryID>dqpogsadai-11492de1f50.Thu-Aug-23-14:21:44-BST-2007</progmon:queryID>
    <progmon:scanInfo>
        <progmon:dataService>ProteinSequenceMySQLResource</progmon:dataService>
        <progmon:numberOfBlocks>9</progmon:numberOfBlocks>
        <progmon:numberOfTuples>8000</progmon:numberOfTuples>
        <progmon:totalScanCost>17893</progmon:totalScanCost>
        <progmon:dataAccessRate>121.0</progmon:dataAccessRate>
        <progmon:averageRowSize>271</progmon:averageRowSize>
    </progmon:scanInfo>
    <progmon:commInfo>
        <progmon:destinationEvaluator>root</progmon:destinationEvaluator>
        <progmon:numberOfTuples>7911</progmon:numberOfTuples>
        <progmon:bufferSize>30000</progmon:bufferSize>
        <progmon:totalCommunicationCost>76839</progmon:totalCommunicationCost>
        <progmon:avgCostPerSend>544.0</progmon:avgCostPerSend>
        <progmon:numberOfInvocations>141</progmon:numberOfInvocations>
    </progmon:commInfo>
    <progmon:commInfo>
        <progmon:destinationEvaluator>http://planetlab4:8199/EvaluationService</progmon:destinationEvaluator>
        <progmon:numberOfTuples>7953</progmon:numberOfTuples>
        <progmon:bufferSize>30000</progmon:bufferSize>
        <progmon:totalCommunicationCost>77164</progmon:totalCommunicationCost>
        <progmon:avgCostPerSend>535.0</progmon:avgCostPerSend>
        <progmon:numberOfInvocations>144</progmon:numberOfInvocations>
    </progmon:commInfo>
</progmon:queryMeasurements>
```

Cost from an individual evaluator on PLANETLAB4

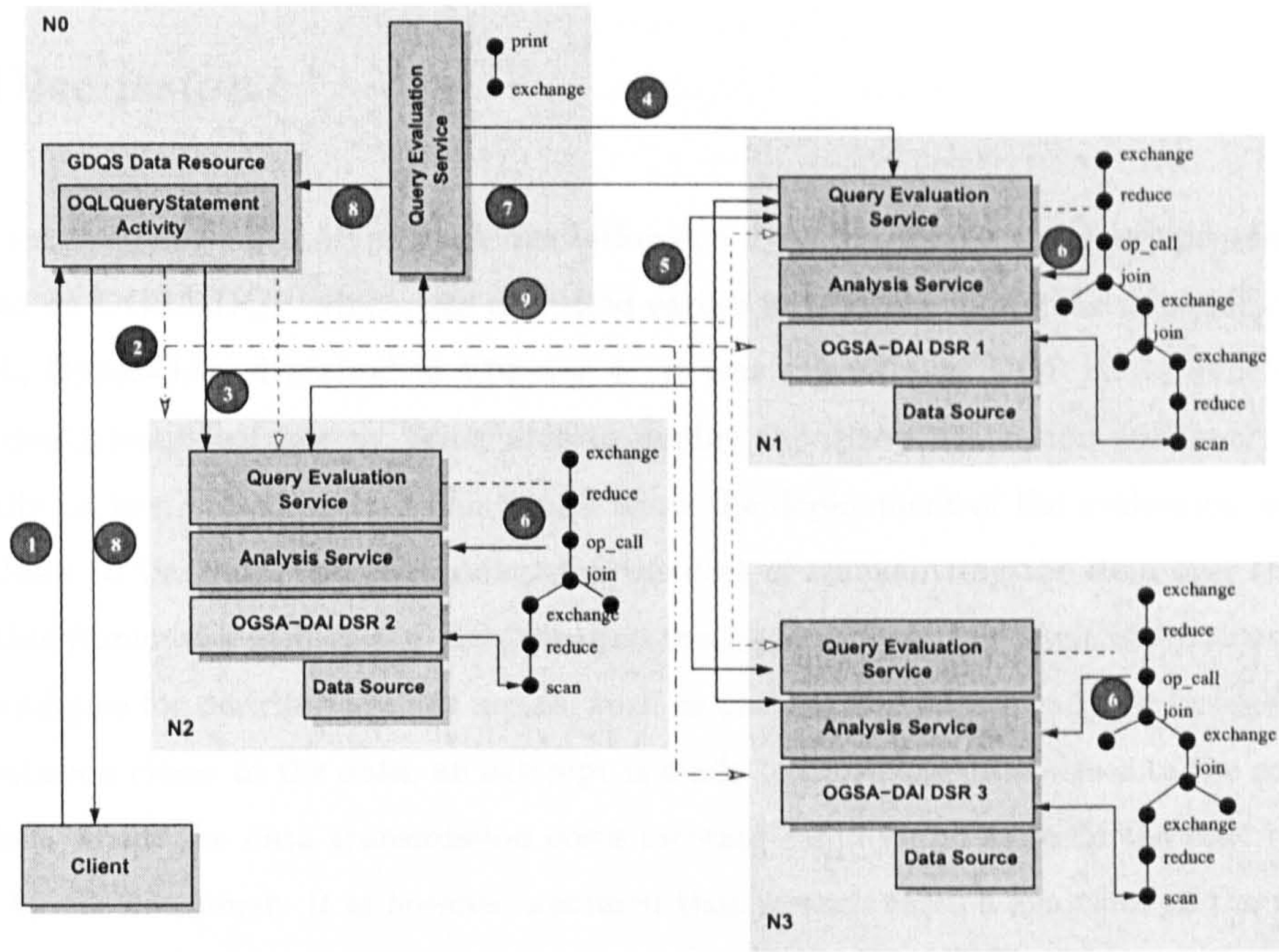
```
<progmon:queryMeasurements sourceID="http://planetlab4:8199/EvaluationService">
    <progmon:queryID>dqpogsadai-11492de1f50.Thu-Aug-23-14:21:44-BST-2007</progmon:queryID>
    <progmon:scanInfo>
        <progmon:dataService>ProteinPropertyMySQLResource</progmon:dataService>
        <progmon:numberOfBlocks>9</progmon:numberOfBlocks>
        <progmon:numberOfTuples>8000</progmon:numberOfTuples>
        <progmon:totalScanCost>4911</progmon:totalScanCost>
        <progmon:dataAccessRate>50.0</progmon:dataAccessRate>
        <progmon:averageRowSize>31</progmon:averageRowSize>
    </progmon:scanInfo>
    <progmon:commInfo>
        <progmon:destinationEvaluator>root</progmon:destinationEvaluator>
        <progmon:numberOfTuples>8006</progmon:numberOfTuples>
        <progmon:bufferSize>30000</progmon:bufferSize>
        <progmon:totalCommunicationCost>55893</progmon:totalCommunicationCost>
        <progmon:avgCostPerSend>1054.0</progmon:avgCostPerSend>
        <progmon:numberOfInvocations>53</progmon:numberOfInvocations>
    </progmon:commInfo>
    <progmon:commInfo>
        <progmon:destinationEvaluator>http://salt:8199/EvaluationService</progmon:destinationEvaluator>
        <progmon:numberOfTuples>8009</progmon:numberOfTuples>
        <progmon:bufferSize>30000</progmon:bufferSize>
        <progmon:totalCommunicationCost>57214</progmon:totalCommunicationCost>
        <progmon:avgCostPerSend>1021.0</progmon:avgCostPerSend>
        <progmon:numberOfInvocations>56</progmon:numberOfInvocations>
    </progmon:commInfo>
</progmon:queryMeasurements>
```

```
</progmon:QueryMeasurementCollection>
```

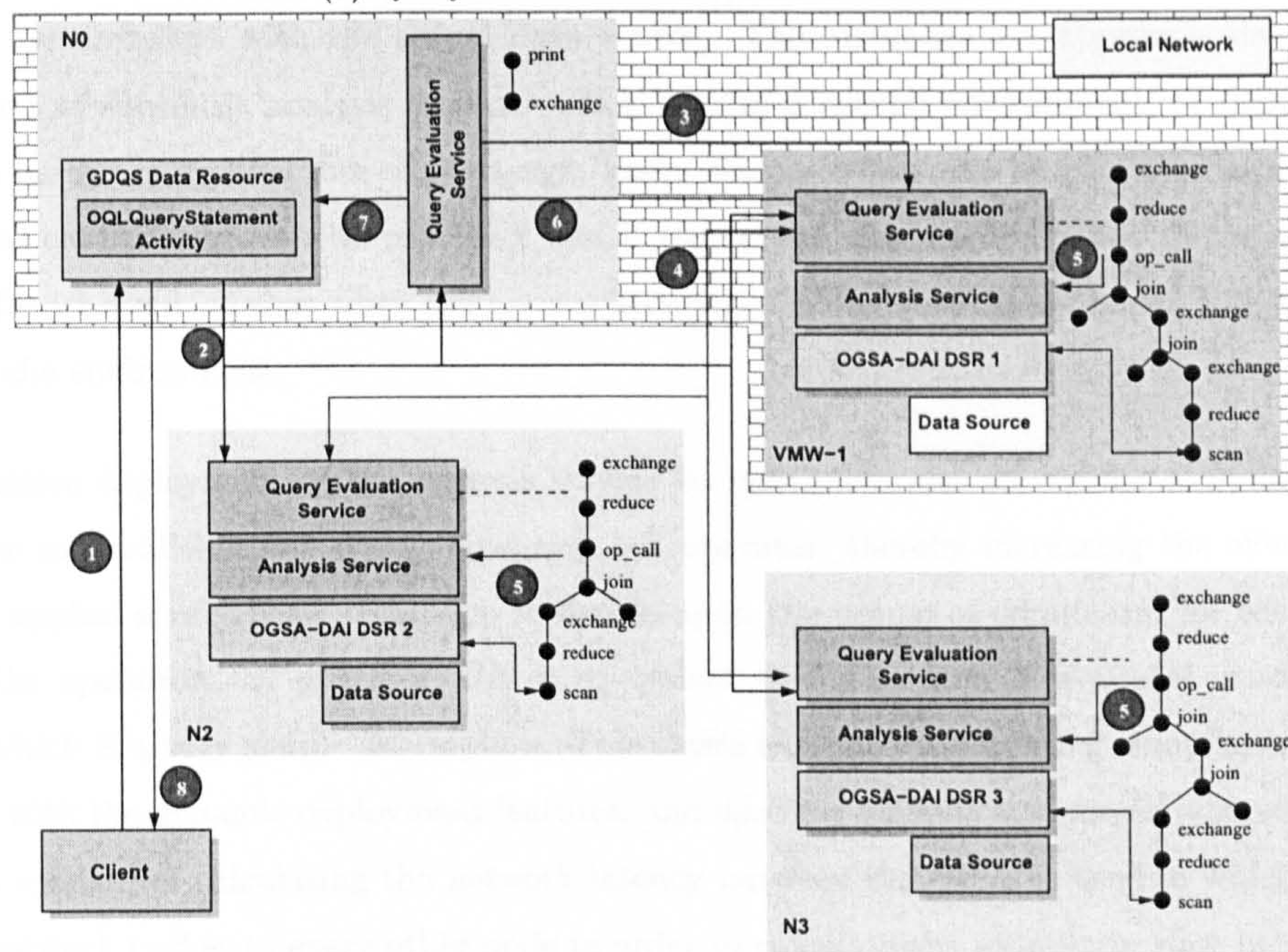
5.2.7.3 Reconfiguration of the DQP data resource

The reconfiguration of the DQP data resource happens as a background process. During this phase, the existing resource which was processing the earlier queries remains alive, and a new data resource is created, which after the configuration is completed, replaces the former. Once the VM is instantiated properly, the schema from the newly deployed data resource and other services must be imported by the DQP data resource. The endpoint of the data service which was pointing to the actual remote database is overwritten with the endpoint of the new data service on the VM. Any instance of the evaluation service or the HostProvider service on that remote node are excluded as well, as all these services are locally available after the VM initialisation. Once this is done successfully, the former DQP data resource is replaced with the new one, and all further queries submitted to this particular instance or session of DQP are processed within this new DQP data resource. A new consumer can however initiate a new DQP session, with the same configuration as the original, in which case the original resources will be used, unless a reconfiguration is called for. The cost of the second reconfiguration may be minimal as it may be possible to reuse the previous deployment of the virtual machine. The process of collecting performance feedback and reconfiguring the corresponding data resource is explained in Figure 5.7.

Figure 5.7(a) shows the process of query evaluation and feedback in the extended version of DQP which collocates various participating services, and collects the performance data from the participating evaluation services once the query is completed. As described before in the context of Figure 5.5, the coordinator service after receiving the query from the client (in step 1) performs the compilation/optimisation of the query and dynamically deploys necessary services on participating nodes. The query execution process follows a similar sequence of sending the query partitions to respective evaluation services (step 3), evaluating each partition by communicating with other evaluation and analysis services (step 4, 5, 6 and 7) and finally sending the complete result to the coordinator and hence the client (step 8). Once the query execution is complete, the coordinator service sends a request to all participating nodes requesting for the performance data as described in Figure 5.6(step 9). Once the feedback from all the nodes are collected, the data is analysed and based on the analysis as explained in Section 5.2.7.2, a virtual machine which encapsulates the data source, data access and analysis services, is deployed within the local network (Figure 5.7(b)) which creates a new configuration for the DQP data resource. For all subsequent queries directed to this particular instance of the DQP data resource, this new configuration is used, thereby utilising the newly deployed resources. However a new consumer may create a new instances of DQP data resource with the original configuration, and submit queries to it until a reconfiguration is required, in which case, it may be possible to avoid the redeployment of services or virtual machines, as that



(a) Query evaluation and feedback collection



(b) Reconfiguring the data resource by deployment of a VM

Figure 5.7: Reconfiguration in dynamic OGSA-DQP

has already been done during the earlier scenario.

5.3 Discussion

This chapter discussed a prototype implementation of a dynamic distributed query processing framework based on OGSA-DQP which was discussed earlier in Chapter 3 and the dynamic deployment framework, DynaSOAr, discussed in Chapter 4. It was argued that DQP could benefit from the dynamic deployment features by being able to deploy the query evaluation and analysis services dynamically on best suited nodes. This would allow the deployment of the evaluation and analysis services closer to the data, thereby minimising the cost of transmitting the data over the network. Further, this framework utilises the virtualization methods proposed in DynaSOAr to exploit alternative paradigms for distributed data access, such as caching the data locally, or instead of moving the computation closer to the data, an attempt is made to move the data closer to the computation in situations where the data transmission costs increase rapidly and exceeds the cost that can be tolerated by the consumer. It is however assumed that in such cases a snapshot of the actual data would cause no hindrance to the analysis and there should be some offline method for keeping the snapshot synchronized with the actual data source. The virtualization approach also allows the deployment of specialist analysis services which require a special environment, or depend heavily upon the hardware architecture of the target node. Such services can be packaged in a VM where the special environment can be provided, and if the service needs to be tuned to the system, this can also be achieved, so that when the service is deployed along with its container VM, it is already tuned to the environment.

The proactive deployment of the analysis service on multiple nodes allows the compiler/optimiser to be able to parallelise any costly *operation_call* operator, thereby increasing the efficiency. The compiler applies a very basic technique to decide upon the *degree of parallelism* for each operator, and for the *operation_call* operator, this is equivalent to the number of available instances of the service, which is a very simple assumption. This thesis extended the existing compiler/optimiser to enable it with the dynamic deployment features, and used the existing cost model wherever possible. A simple method of calculating the network latency between the nodes is used in which each node sends a network packet to every other node in order to calculate the round-trip time between them. A more sophisticated cost model such as the network-aware optimisation methods proposed in [134] and a sophisticated network monitoring tool or framework such as [135] would develop the framework further.

As mentioned before, the compiler/optimiser uses a very simple cost model, which has been extended to incorporate the dynamic deployment features. An adaptive DQP, which adapts to the changing dynamics of a Grid system is still a long way away. The effects of changes in resources at runtime have been considered in the investigations into *adaptive distributed query processing* [137, 138]. It would be an effective solution to combine the findings of GridSHED [113], DynaSOAr and the *adaptive DQP* investigation. There have been some work on *fault-tolerance in distributed query processing* [139, 140]. The concepts of the dynamic DQP are also relevant to fault-tolerant query processing systems where a failure of an evaluation node can be handled through the deployment of the same service on another node or a virtual machine as a replacement of the failed node, and by replaying certain sections of the query evaluation to regain the state where the processing stopped due to the failure.

In summary, the dynamic version of DQP creates a loose coupling between the services and the computational resources, and adopts the dynamic deployment techniques to "create" each node as and when required by deploying services on the available resources. A dynamic and fluid query processing engine is created at runtime which is able to reconfigure itself based on the changes in the environment and the feedback from the query processing activities. The evaluation of the prototype, the experimental set-up and the results are discussed in later chapters.

Chapter 6

Evaluation of the Dynamic DQP Framework

The evaluation of the research work described in Chapters 3, 4 and 5 is presented in this chapter. The goal of this research has been to define a service oriented distributed query processing (DQP) framework capable of evaluating distributed queries over disparate data sources using emerging standards for data access and integration, and exploit the advantages of dynamic service deployment to enhance the performance. The purpose of this chapter is to verify the validity of the claims made while presenting the research work by executing several experiments within an experimental set-up. Several scenarios involving the static DQP (presented in Chapter 3) and the dynamic extension (presented in Chapter 5) are presented and both frameworks are evaluated. The results are analysed thereafter to establish the earlier claims.

6.1 Implementation

Both OGSA-DQP and DynaSOAr frameworks have evolved through a series of implementations since they were first conceived. This section takes a brief look at these different implementations.

6.1.1 OGSA-DQP

OGSA-DQP started as a part of ^{my}*Grid* and OGSA-DAI and was a collaboration between Newcastle and Manchester universities. The first release of OGSA-DQP in September 2003 was based on the Globus Toolkit 3.0 (OGSI). Since the refactoring of OGSI into WS-ResourceFramework, the implementation of OGSA-DQP has changed considerably. The central coordinator component (GDQS) was developed to support both WS-RF and WS-I, and the query evaluation component, which is considered as a major contribution towards this thesis was based on WS-I in order to benefit from the findings from ongoing research on dynamic deployment. The earlier releases of OGSA-DQP treated the Polar* compiler/optimiser component as black box, thus making it extremely difficult to incorporate the dynamic deployment features within the compiler/optimiser component. Later on, a new version of OGSA-DQP was created which contained a new Java-based compiler/optimiser component which supported SQL queries instead of the earlier OQL, but the concepts behind the compilation process, such as using parallel database techniques remained the same. This is the version of OGSA-DQP that has been used to extend the framework and incorporate the dynamic deployment features.

6.1.2 DynaSOAr

Research on DynaSOAr progressed in different strands and received contributions from many researchers since its inception presented in [19]. The first version of DynaSOAr benefited from the work done in GridSHED [113] and used the Condor scheduling system and Class Ads in order to schedule jobs on a set of nodes each offering the HostProvider service. The jobs invoked the HostProvider Web Service for retrieving the code for the service requested by the consumer, deployed and processed the request. The basic components of DynaSOAr, such as the *Service Provider*, *Host Provider* and *Software Repository* were developed during this phase. The initial version of DynaSOAr was restructured to incorporate the concepts of *message orientation* and each of the components were redesigned to support this model. The new architecture introduced the *Service Registry* component based on GRIMOIRES, a basic *broker* component for brokering service requests, refactored the HostProvider to support a highly dynamic structure such as clusters at different institutions coming together to form a dynamic virtual organisation. More developments into DynaSOAr resulted in a framework which allowed dynamic deployment of databases, specialised services wrapped in a virtual machine allowing the creation of an on-demand ad-hoc Grid, and this is the version of DynaSOAr which has been presented in this thesis, and these are the concepts that are incorporated within the DQP framework leading to a dynamic version of OGSA-DQP.

6.2 Evaluation

This section describes in detail all the experiments carried out in order to validate the claims made in earlier chapters.

6.2.1 Evaluation Platform

An extensive experimental setup was created for evaluating the dynamic DQP framework. This consisted of local computational resources within the same university network and remote computational resources distributed across the world to simulate a real-world like situation when consumers request services that are distributed globally. PlanetLab [136] was used to acquire remote nodes for this purpose.

A dynamic DQP framework consisting of five test data resources, the structure of which are shown in Listing 6.1, and the HostProvider service was setup on a set of Linux machines within the Newcastle University GIGA cluster - each of them being a four-processor Intel® Xeon^{TM} CPU 2.80GHz system, with 2GB memory and hosting the VMWare Server software. The GDQS was deployed on a desktop running on Windows XP (Service Pack 2) machine - a four-processor Intel® Pentium(R)^{TM} CPU 3.0GHz with 2GB memory. The same machine which hosted the GDQS also hosted the *Software Repository* and the *Service Registry*. A copy of the analysis service was deployed on a Linux (one-processor Intel® Xeon^{TM} CPU 2.40GHz system with 1GB memory) system at the Edinburgh Parallel Computing Centre (EPCC). To compare the results with a real-world situation, an exactly similar DQP framework with databases, data access, and HostProvider services was set up on a set of Planetlab[136] nodes, with compute resources located at geographically remote locations, such as in Toronto, Tokyo, Berkeley, and several European cities. Each database was replicated on five nodes to avoid experimental problems due to node failures. The local network was a high speed 100Mbps ethernet, and the connection with the PlanetLab network being through the JANET [141], a high speed gigabit ethernet between the universities in the UK and GREN [142], the Global Research and Educational Network. Apache Tomcat version 5.0.28 and MySQL version 5.24 were used as the web service container and DBMS respectively. The complete experimental setup is shown in Figure 6.1.

One of the databases used for the test queries was loaded with several tables, each with 100,000 records, and fixed record sizes of 128 bytes, 256 bytes, 512 bytes, 1 Kbytes, 2 Kbytes, 4 Kbytes, 8 Kbytes and 10 Kbytes. Experiments were designed to fetch data out of each table with varying cardinalities, optionally perform a join with data from another remote database and perform the analysis on each tuple using the analysis service. Results were collected in order to compare the

Listing 6.1 Structure of the databases used for the experiments

```
1   /* Description of the goterm database */
2   table goterm {
3       'id' varchar(32) NOT NULL default '',
4       'type' varchar(55) NOT NULL default '',
5       'name' varchar(255) NOT NULL default '',
6       PRIMARY KEY ('id')
7   }
8
9   /* Description of the interaction database */
10  table gene_sequence {
11      'Sequence_ID' varchar(50) NOT NULL default '',
12      'Sequence_Type' varchar(100) default '',
13      'Sequence_Source' varchar(100) default '',
14      'Sequence_Description' varchar(255) default '',
15      'Sequence' text NOT NULL,
16      PRIMARY KEY ('Sequence_ID')
17  }
18
19  /* Description of the proteinproperty database */
20  table protein_property {
21      'ORF' varchar(55) NOT NULL default '',
22      'molecularWeight' float NOT NULL default '0',
23      'hydrophobicity' float NOT NULL default '0',
24      PRIMARY KEY ('ORF')
25  }
26  table random_property {
27      'id' bigint(20) NOT NULL,
28      'ORF' varchar(55) NOT NULL default '0',
29      'molecularWeight' float NOT NULL default '0',
30      'hydrophobicity' float NOT NULL default '0',
31      PRIMARY KEY ('id')
32  }
33
34  /* Description of the proteinsequence database */
35  table protein_sequence {
36      'ORF' varchar(50) NOT NULL default '',
37      'sequence' text NOT NULL,
38      PRIMARY KEY ('ORF')
39  }
40  /* average row size 128 bytes */
41  table random_sequence_128 {
42      'id' bigint(20) NOT NULL,
43      'sequence' text NOT NULL,
44      PRIMARY KEY ('id')
45  }
46  ...
47  /* average row size 10 Kbytes */
48  table random_sequence_10K {
49      'id' bigint(20) NOT NULL,
50      'sequence' text NOT NULL,
51      PRIMARY KEY ('id')
52  }
53
54  /* Description of the proteinterm database */
55  table protein_goterm {
56      'ORF' varchar(55) NOT NULL default '',
57      'GOTermIdentifier' varchar(32) NOT NULL default '',
58      PRIMARY KEY ('ORF', 'GOTermIdentifier')
59  }
```

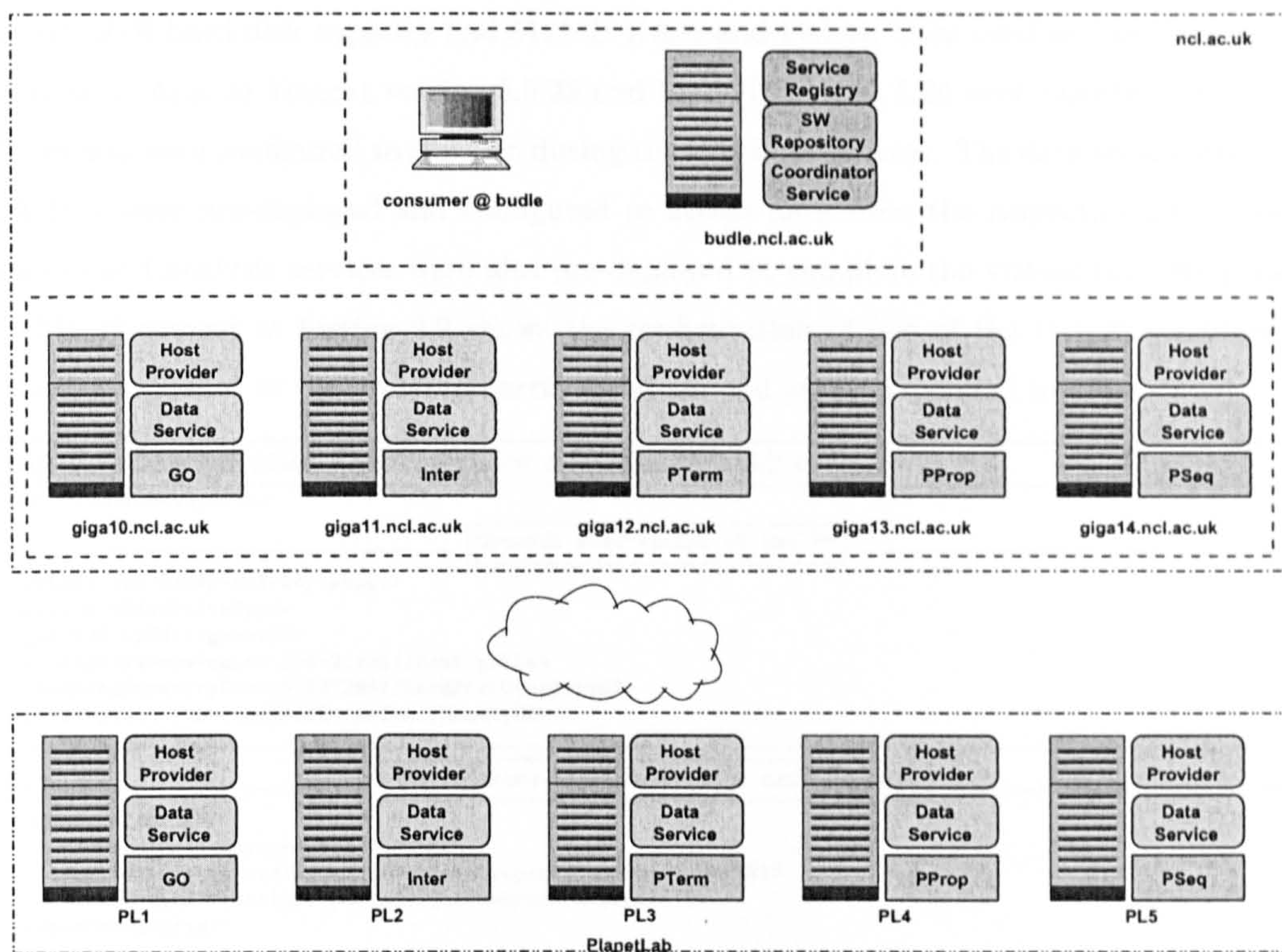


Figure 6.1: The complete experimental setup

performance of the various setups, such as (i) a PlanetLab setup with remote analysis service, (ii) a PlanetLab setup where the evaluation and analysis services are collocated on a PlanetLab node using the dynamic deployment framework, (iii) a setup with multiple copies of the analysis services dynamically deployed on multiple nodes and (iv) a setup where a reconfiguration of the DQP data resource takes place by deploying one or more data sources (wrapped in VMs) within the local network.

There are issues regarding the usage of PlanetLab infrastructure for experiments, but it was chosen as one of the primary components within the evaluation platform as it is "designed to subject network services to real-world conditions" [143]. The concerns about the reliability of PlanetLab nodes due to their heavy load have been addressed by selecting lightly loaded nodes with higher virtual memory and CPU speed that are known not to have any problems. Such a selection can be made through the PlanetLab monitoring platform, CoMon [135]. PlanetLab also offers a "reservation" mechanism by which the nodes can be reserved for a certain duration guaranteeing a majority share of the CPU during that period.

Two virtual machines were used to investigate the deployment of a database snapshot. The *protein-sequence* and *proteinproperty* databases were wrapped in two separate VMWare virtual machines,

each with 4GB hard disk capacity and 512MB RAM and Fedora Core version 4 as the guest operating system. Apache Tomcat version 5.0.28 and MySQL version 5.24 were installed on the virtual machines and were configured to start up during the VM boot process. The data access services from OGSA-DAI were pre-deployed and configured to access data from the respective databases. The evaluation and analysis services were also pre-deployed to complete the virtual machine packaging. The XML document in Listing 6.2 shows the configuration of one of the virtual machines which contained a snapshot of the *proteinproperty* database and other associated services.

Listing 6.2 Configuration Document for a Virtual Machine

```
<VirtualMachineDescription>
                                   General description of the VM
    <vmName>vmw-nam48-fc4-2</vmName>
    <vmType>VMWARE</vmType>
    <guestOS>LINUX</guestOS>
    <configFile>vmw-nam48-fc4-2.vmx</configFile>
    <hardDiskCapacityGB>4294967296</hardDiskCapacityGB>
    <primaryMemoryMB>536870912</primaryMemoryMB>

                              The database and data service configuration
    <databaseConfig>
        <dbName>proteinproperty</dbName>
        <URI>axis/services/ogsadai/ProteinPropertyDataService</URI>
        <ResourceID>ProteinPropertyMySQLResource</ResourceID>
    </databaseConfig>

                              Information about web service containers
    <tomcatInstancePath>/root/addon/tomcat/5.0.28</tomcatInstancePath>
    <tomcatPort>8090</tomcatPort>

                                 Information about deployed services
    <vmServiceList>
        <serviceName>QueryEvaluationService</serviceName>
        <serviceURI>dqp-evaluator/services/QueryEvaluationService</serviceURI>
    </vmServiceList>
    <vmServiceList>
        <serviceName>EntropyAnalyserService</serviceName>
        <serviceURI>entropy-analyser/services/EntropyAnalyserService</serviceURI>
    </vmServiceList>

</VirtualMachineDescription>
```

The virtual machines and all other deployable services, such as the evaluation service and the analysis services were uploaded to the *Software Repository* which in turn created entries of these in the *Service Registry* so that the DQP system is able to search for required entities within the registry.

6.2.2 Collocating the Data and Analysis Code

The purpose of this experiment was to establish the effect of the cost of invoking an analysis service on the query execution cost, and to investigate the benefits of collocating the analysis service with the data. The usage scenario is explained in Section 5.1.2 where a query which accesses a large volume of data from a database and performs an analysis on each of the retrieved tuples is submitted by an

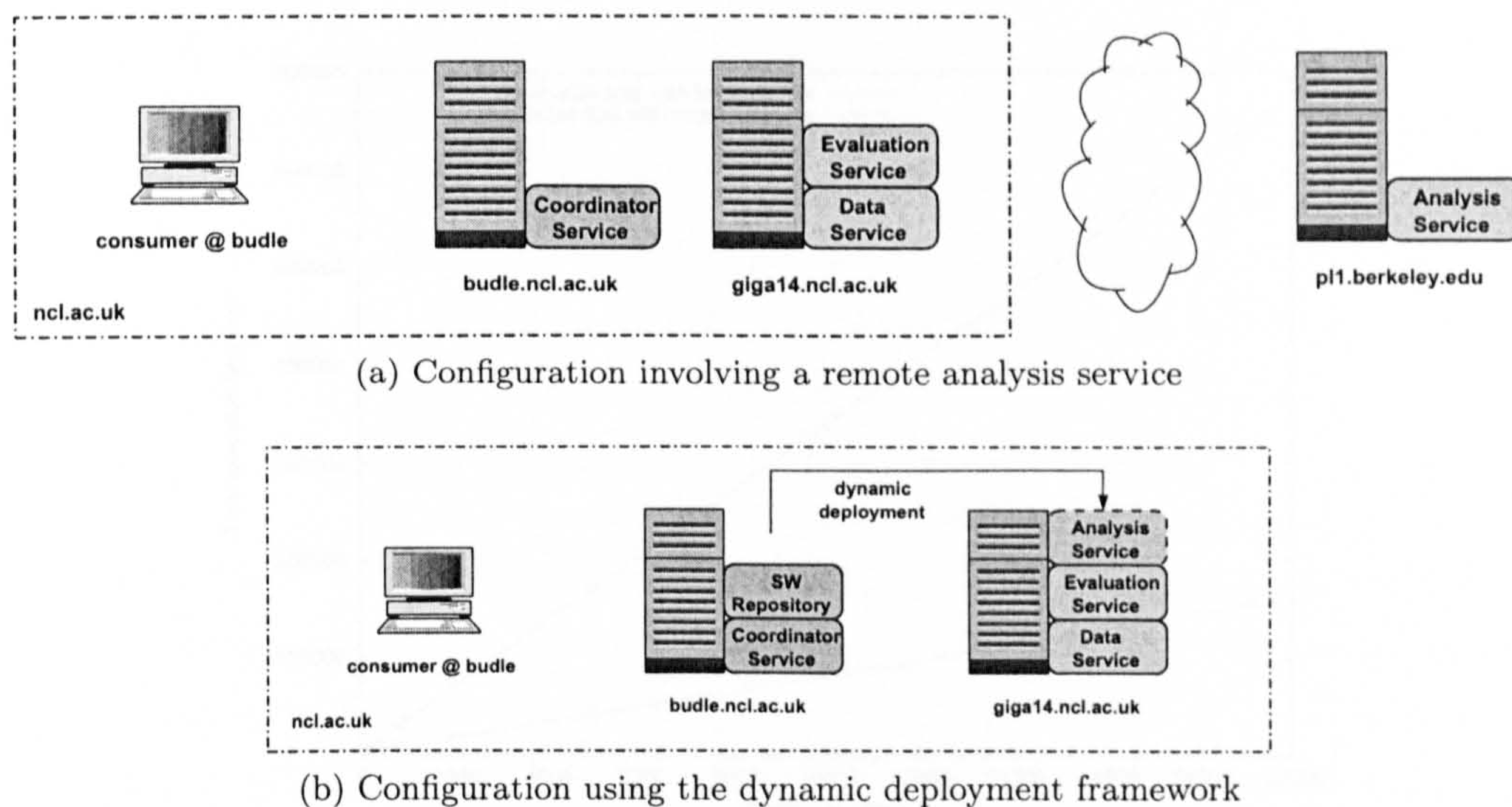
e-scientist. The rationale behind the scenario is to minimise the volume of data being transferred over the network, and thus reduce the cost of overall query execution by deploying a copy of the service closer to the data source to avoid the cost of a remote Web Service invocation which increases with the volume of data.

The *select-operation_call* query as shown in Listing 6.3 was used to investigate the effects of collocating the analysis service with the data.

Listing 6.3 A select-opcall query

```
select proteinsequence_random_sequence_256.id,
calculateEntropy(proteinsequence_random_sequence_256.sequence)
from proteinsequence_random_sequence_256 where
proteinsequence_random_sequence_256.id <= n;
```

The experiment was conducted with two different configurations, (i) where the analysis service is deployed on a remote PlanetLab node at the University of California, Berkeley, United States and the data is located on one of the nodes within the giga cluster of Newcastle University, and (ii) where the DQP system is not bound to any particular instance of the analysis service and the dynamic deployment framework selects a node closest to the node hosting the database and the data access service. The two configurations are described in Figure 6.2.



(a) Configuration involving a remote analysis service

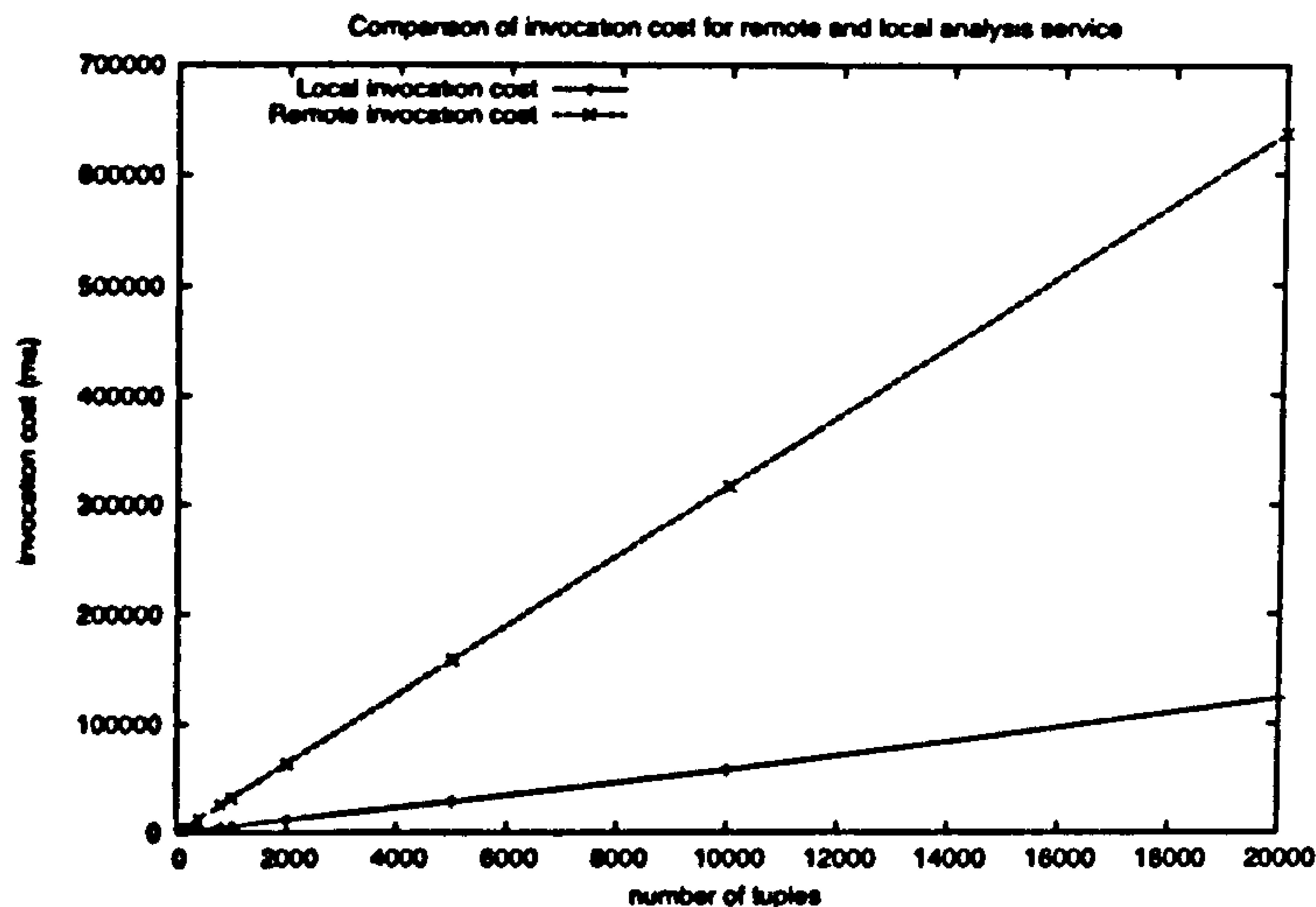
(b) Configuration using the dynamic deployment framework

Figure 6.2: Two configurations used for experimenting with service collocation

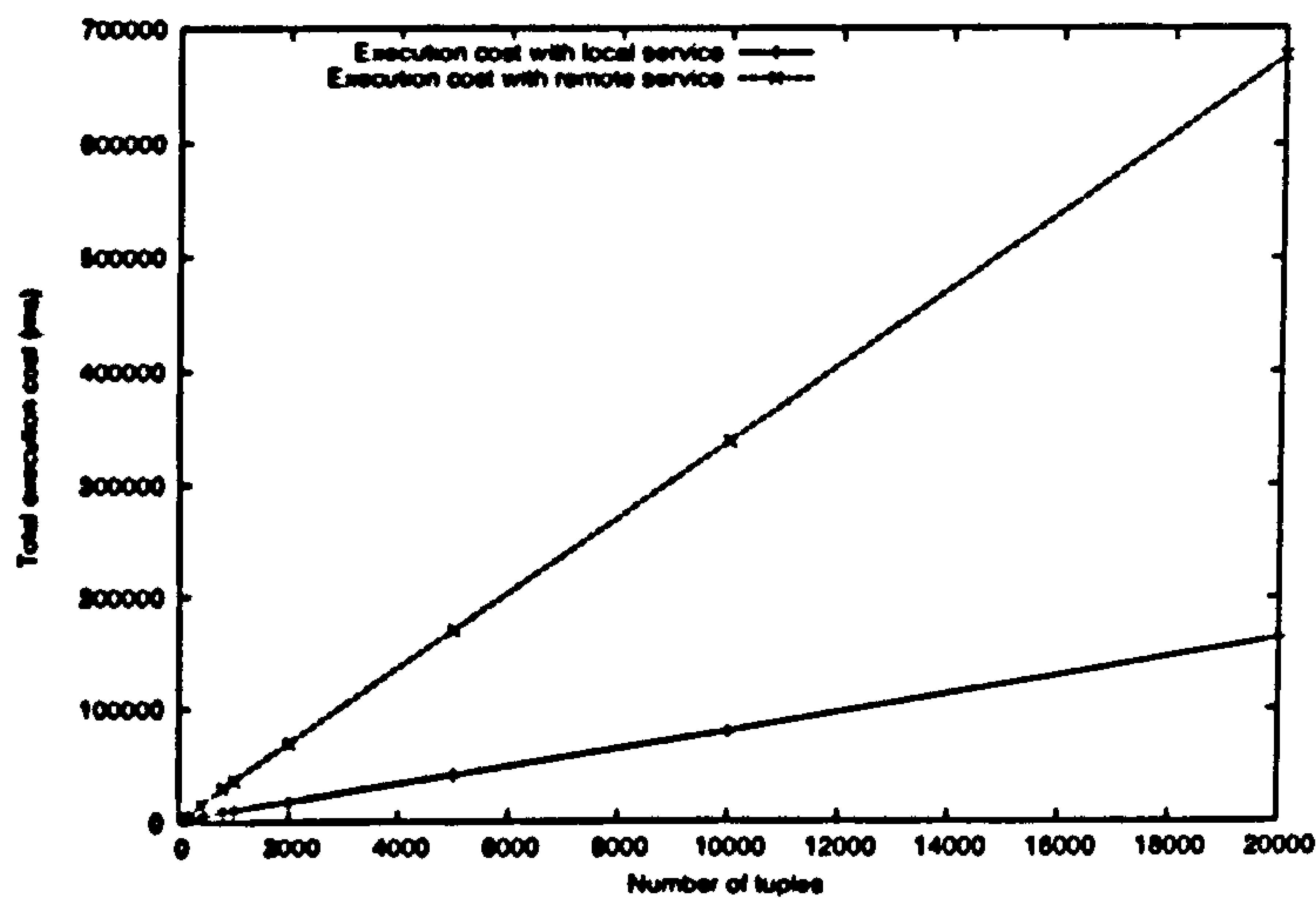
The dynamic deployment takes place during the schema import phase of DQP initialisation and does not affect the query execution costs. The query retrieves data from the "random_sequence" table in the "proteinsequence" database with various granularity from 100 to 20000 tuples depending on the value of *"n"* and for each retrieved tuple the analysis service is invoked and the result is returned to

the consumer. In the first scenario, a remote Web Service is invoked for each tuple, whereas in the second scenario, the analysis service is local to the data source.

It was *expected* that queries for which the configuration with a remote analysis service is used would have higher execution cost because of the greater cost in invoking the remote service compared to the configuration where the analysis service is deployed locally which can be seen in Figure 6.3.



(a) Comparison of the service invocation cost



(b) Comparison of the query execution cost

Figure 6.3: Comparing query execution using a local and a remote service

It can be seen in Figure 6.3(a), that, as the number of tuples retrieved by the query increased, the cost of invoking the remote Web Service increased and had a direct effect on the overall execution cost of the query which is plotted in the graph in Figure 6.3(b). The graphs also show the amount of

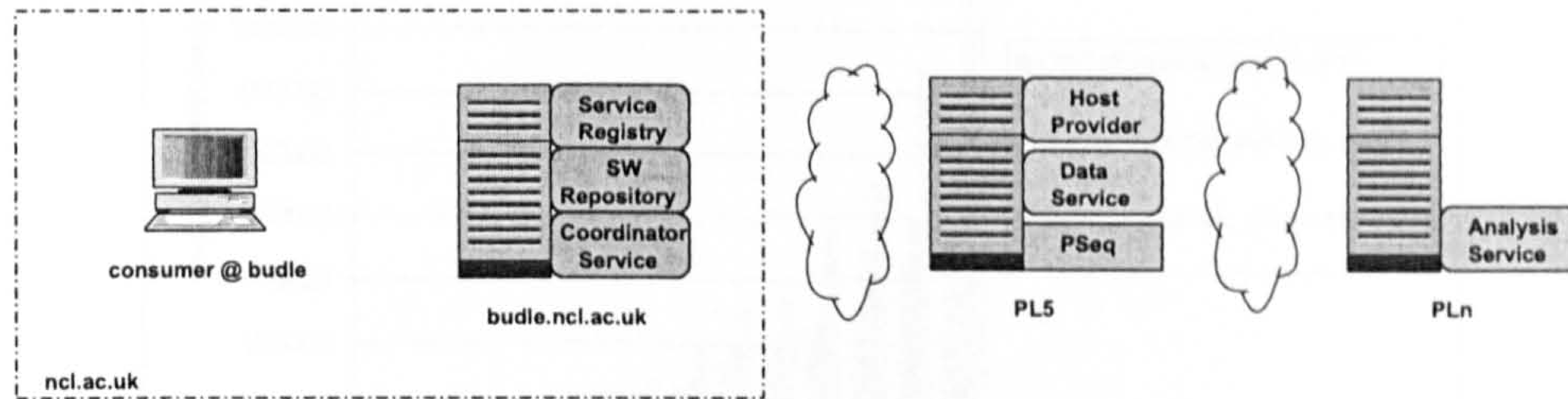
performance gain that can be achieved by deploying a copy of the service closer to the data source which results in lower invocation cost, and as a result lower overall execution cost.

6.2.3 Parallelization of OperationCall using Proactive Deployment

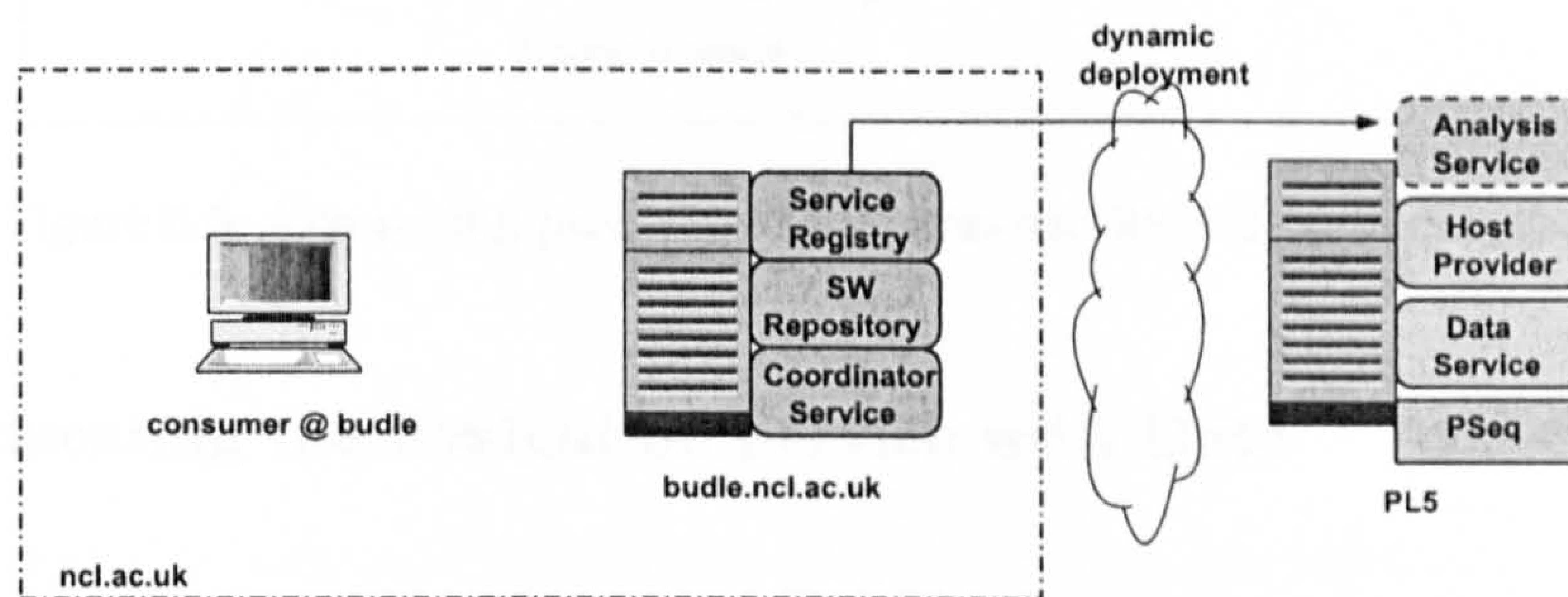
The effect of an increased degree of parallelism was investigated in this experiment. The dynamic version of OGSA-DQP performs a proactive deployment of the analysis service on the available hosts during the initial schema import phase, which was described in Section 5.1.3 and 5.2.4. The approach is based on experimental results observed in [144]. Using this approach, OGSA-DQP is able to distribute the invocations to the Web Service to several service instances and parallelise the *operation_call* operator. It is envisaged that the distribution of the service invocation among multiple instances by parallelising the operator should improve the overall performance of query execution.

The query as mentioned in Listing 6.3 was used to investigate the effects of the proactive deployment of the analysis service. Two DQP configurations with a similar set of data sources and Host Provider services deployed on a group of PlanetLab nodes were used with slight variations - (i) where the endpoint of the analysis service is mentioned in the DQP configuration script and (ii) where no specific endpoint is mentioned and DQP is allowed to deploy the service pro-actively on suitable nodes. The different configurations are explained diagrammatically in Figure 6.4. As in the experiment described in Section 6.2.2, the query retrieves data of various cardinality from the *proteinsequence* database and invokes the analysis service on each "sequence" attribute of the tuples. In the first configuration, one remote instance of the analysis service is invoked for each tuple, and in the second configuration, the analysis service is dynamically deployed over five PlanetLab nodes and the *operation_call* operator is parallelised over five evaluation services.

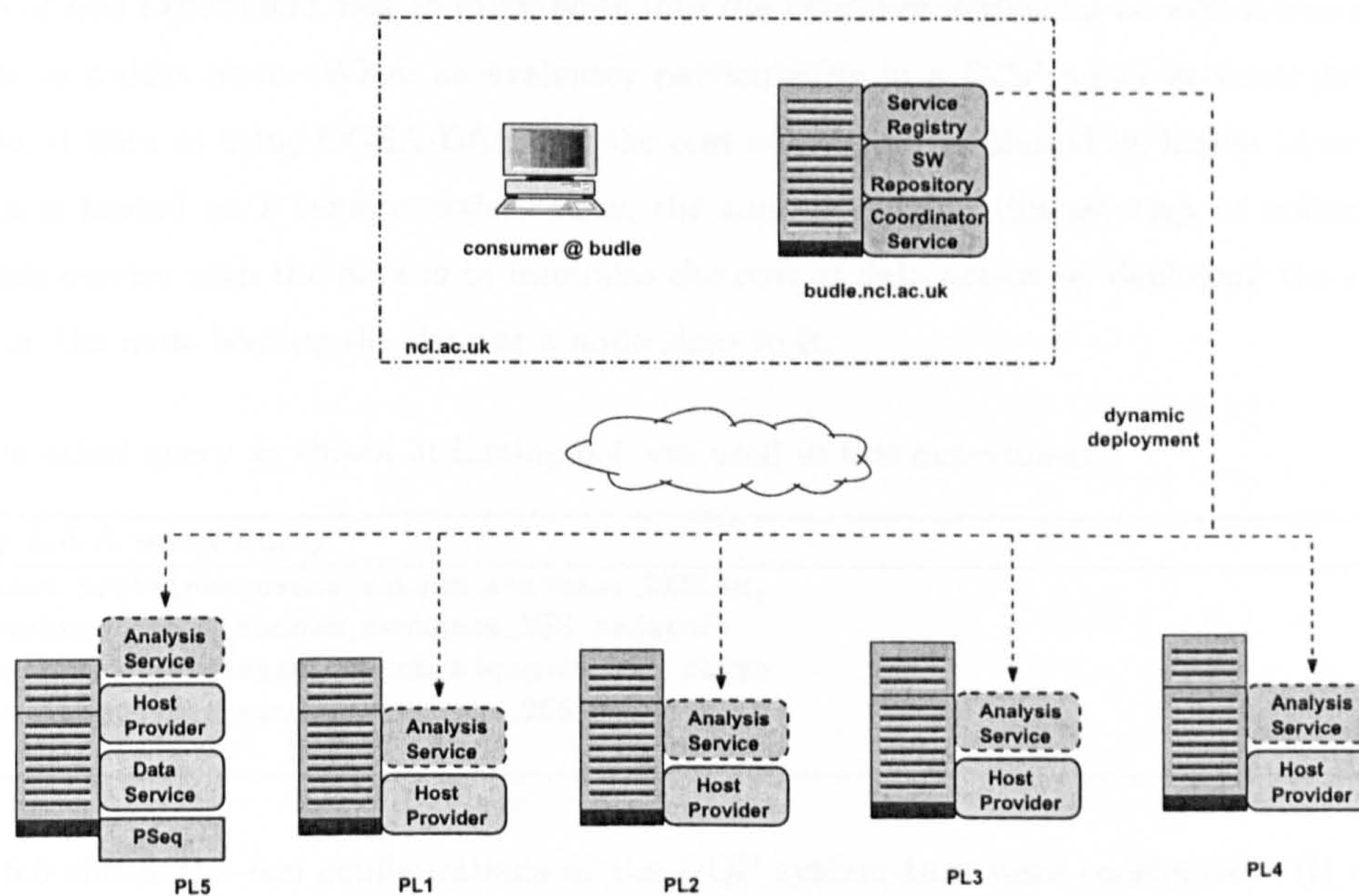
The *expected* result in this experiment was an improved performance of the overall query execution by reducing the query execution time for the configuration where the *operation_call* operator is parallelised. This can be seen in Figure 6.5 which compares the total query execution cost for (i) a configuration with one remote analysis service (Figure 6.4(a)), (ii) a configuration with one instance of the analysis service close to the data source (Figure 6.4(b)) and (iii) a setup where the *operation_call* operator is parallelised over five nodes (Figure 6.4(c)). The cost of query execution in a parallelised setup is lower than the setup which uses one copy of the analysis service close to the data, and it should prove even more beneficial for a analysis service which is expensive in nature because of increased partitioned parallelism.



(a) Using a remote analysis service



(b) Single instance of analysis service collocated with data



(c) OperationCall parallelised over 5 instances of the analysis service

Figure 6.4: Experimental setup for the proactive analysis service deployment

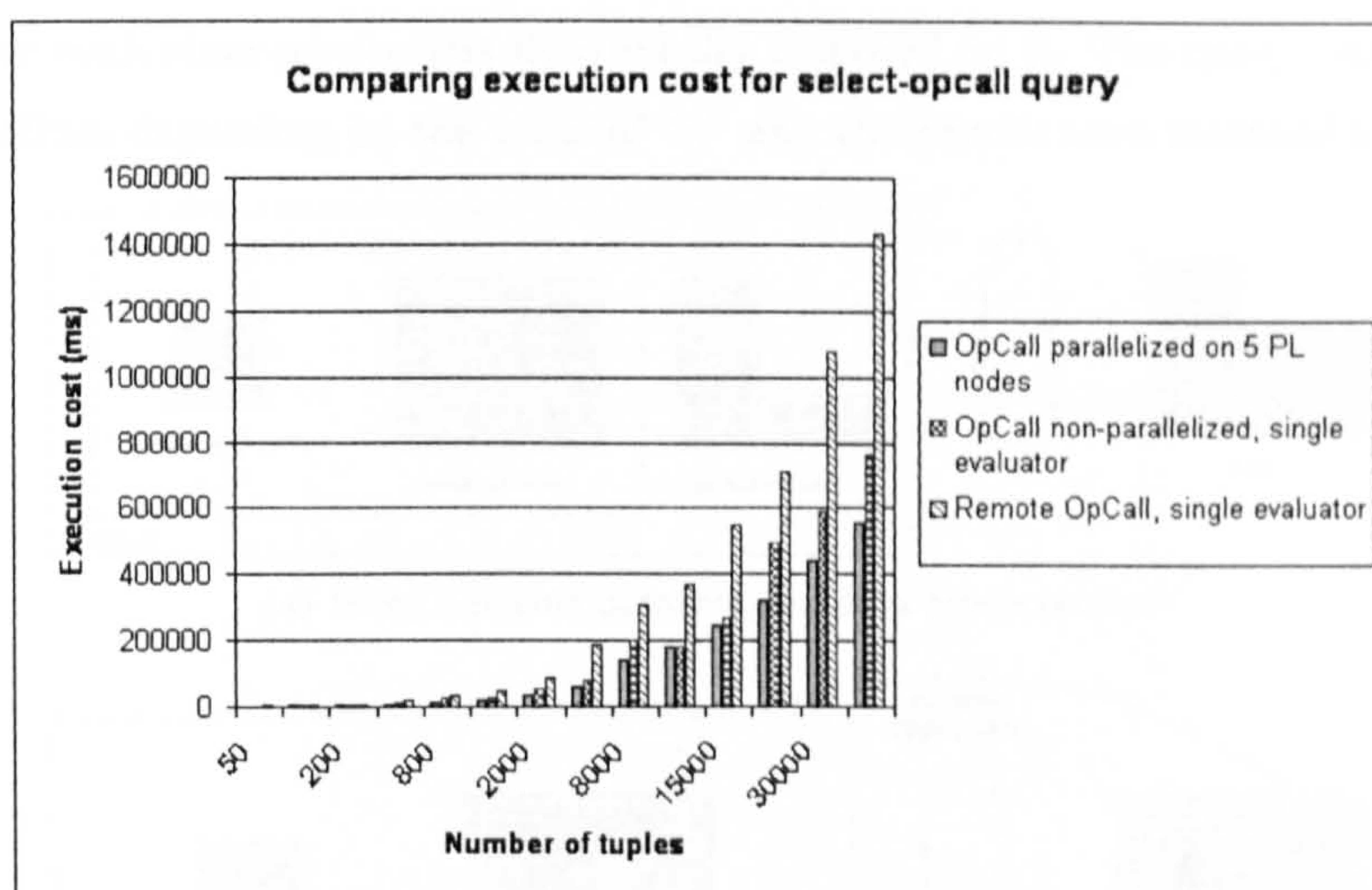


Figure 6.5: Comparing parallelised and non-parallelised operation calls

6.2.4 Collocating the Evaluation Service with Data

The rationale behind collocation of the evaluation service with the data was discussed in 5.1.1. The purpose of this experiment was to investigate into the effects of deploying an evaluation service on or closer to a data node. When an evaluator participating in a DQP query accesses data from a database, it does so using OGSA-DAI, and the cost of data access should be higher in cases where the data is hosted on a remote node. Thus, the concept behind this strategy of collocating the evaluation service with the data is to minimise the cost of data access by deploying the evaluation service on the node hosting the data or a node close to it.

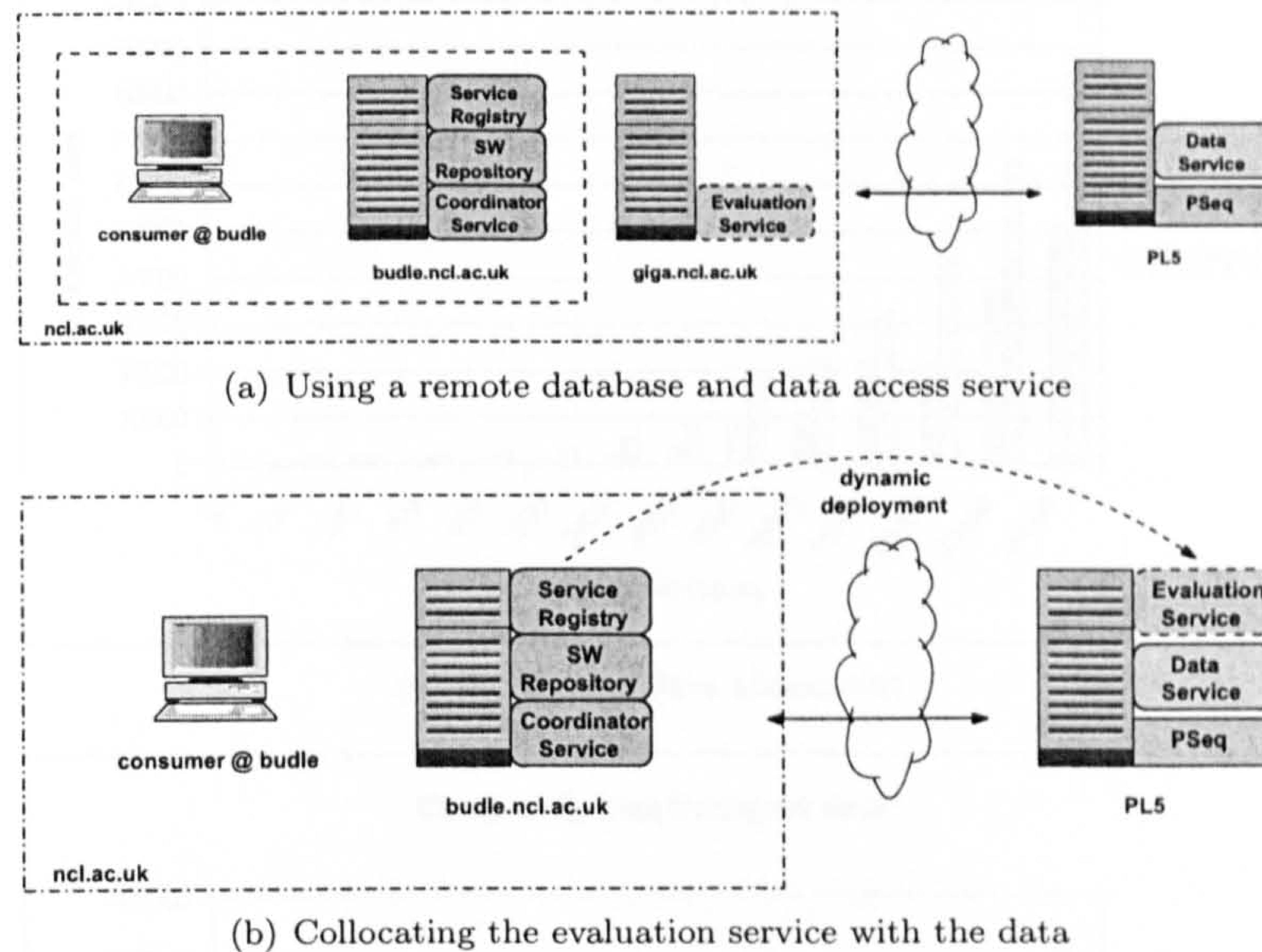
A simple *select* query as shown in Listing 6.4 was used in this experiment.

Listing 6.4 A select query

```
select proteinsequence_random_sequence_256.id,
proteinsequence_random_sequence_256.sequence
from proteinsequence_random_sequence_256 where
proteinsequence_random_sequence_256.id <= n;
```

Figure 6.6 shows the two configurations of the DQP system that were considered - (i) where the data was hosted on one of the PlanetLab nodes and the evaluation services on the local Giga cluster within the Newcastle University campus, and (ii) where the data was hosted on the same PlanetLab node and the DQP system was allowed to assign the best possible node for the *scan* operator irrespective of whether the node hosted an evaluator or not. In the second configuration, the compiler/optimiser selected the node which hosted the data and the data service for the *scan*

operator and an evaluation service was dynamically deployed on it. The query retrieved data with various cardinalities depending on the value of "n" and the results were returned to the consumer.

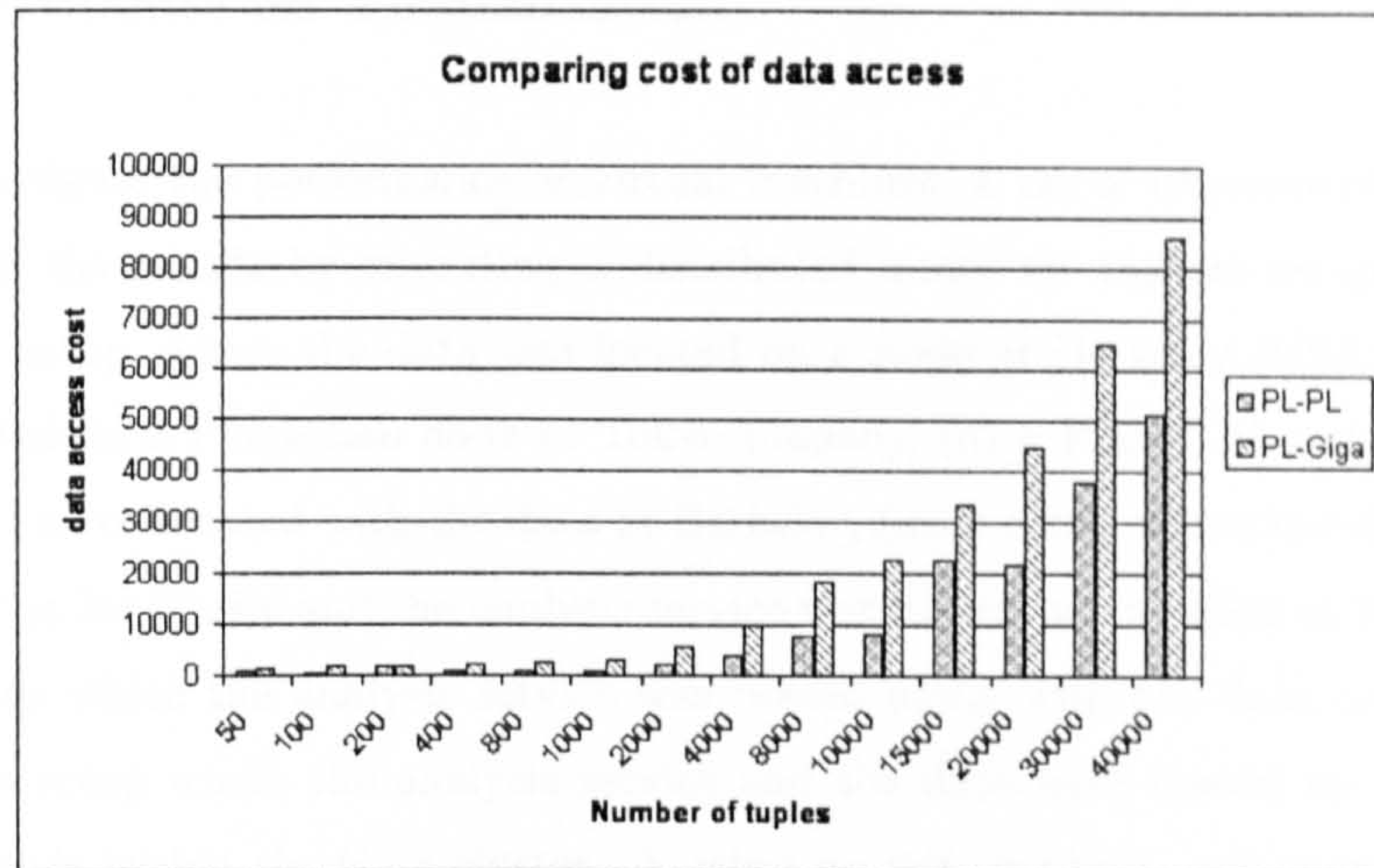


(a) Using a remote database and data access service

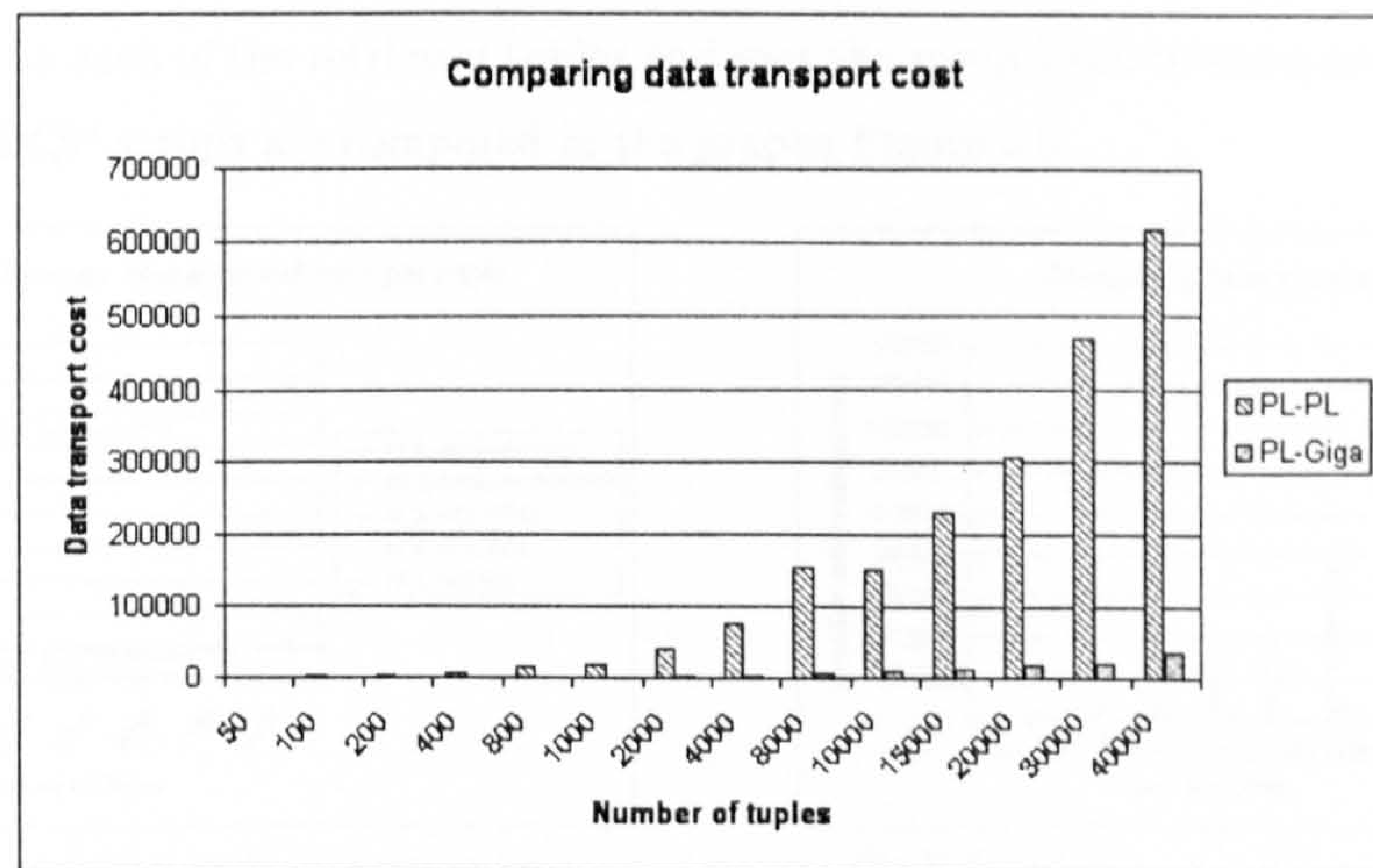
(b) Collocating the evaluation service with the data

Figure 6.6: Experimental setup for the collocation of evaluation with data

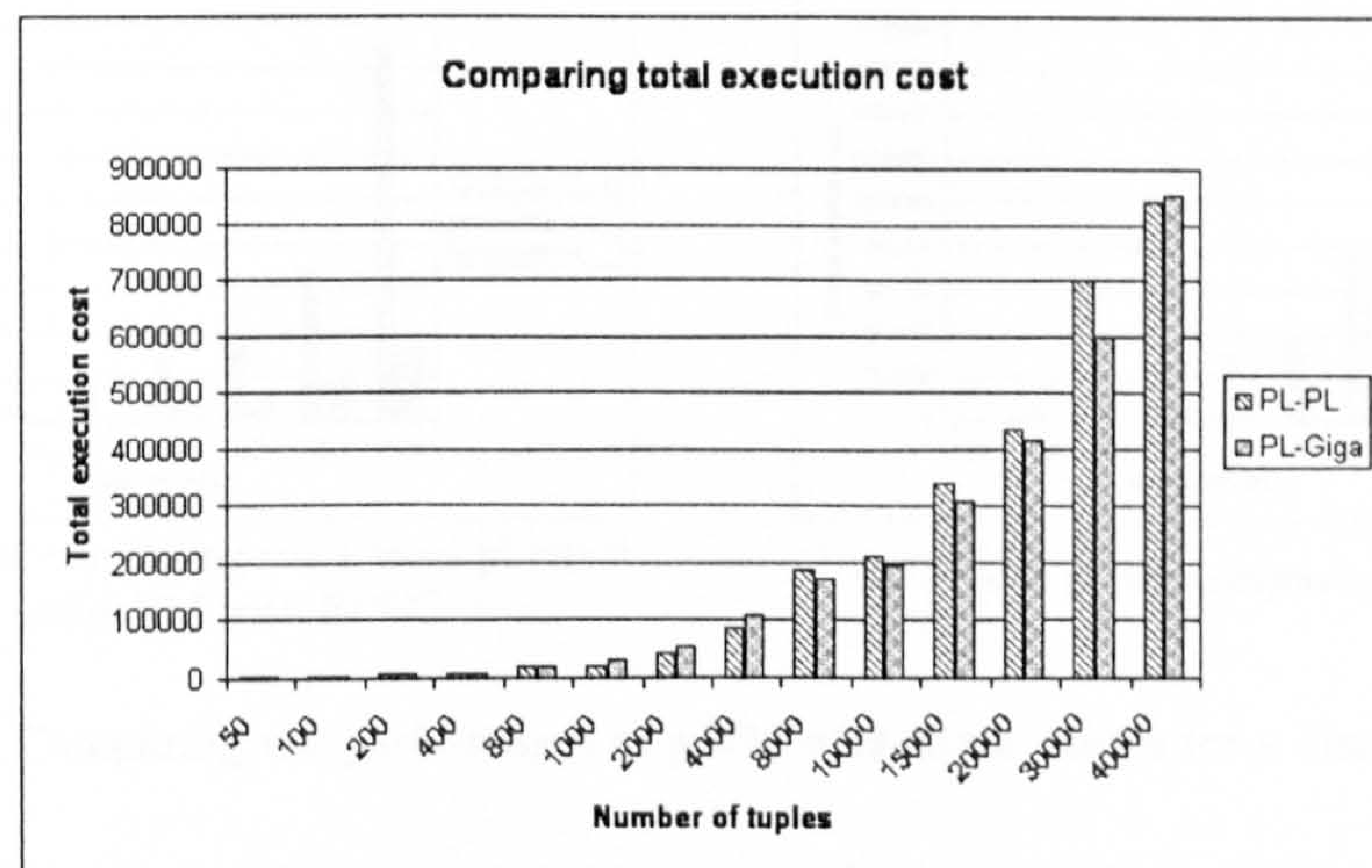
The results of the experiment are shown in Figure 6.7. In the charts, the results for the setup where the data is hosted on PlanetLab and the evaluation service on a giga cluster node are labelled as "PL-Giga" and the setup where the evaluation service is collocated with the data on PlanetLab is denoted with "PL-PL". The *expected* result was an improved overall query execution cost when the evaluation service was moved on the node which contained the data because of the lower data access cost. This was true up to a certain number of tuples beyond which, the overall execution cost was almost similar for both the configurations. The reason behind this was the increased cost in transporting the results as the number of tuples, i.e. the amount of data travelling over the network increased. It can be seen from Figure 6.7(a) that the cost of accessing the data is lower when the evaluation service is collocated with the data on the PlanetLab node than when the evaluation service is hosted on a local node within the giga cluster, but the cost of transporting the results from PlanetLab to the consumer increases rapidly (Figure 6.7(b), as a result of which the improvement achieved by lowering the data access cost is nullified. This highlights the requirement of a better transport mechanism for DQP, which currently relies on the serialisation and de-serialisation mechanisms of Apache Axis [77] for transporting the data as SOAP. OGSA-DAI, on the other hand uses a streaming mechanism, which still relying on SOAP produces a better performance because of the streaming functionality.



(a) Comparing data access cost



(b) Comparing data transport cost

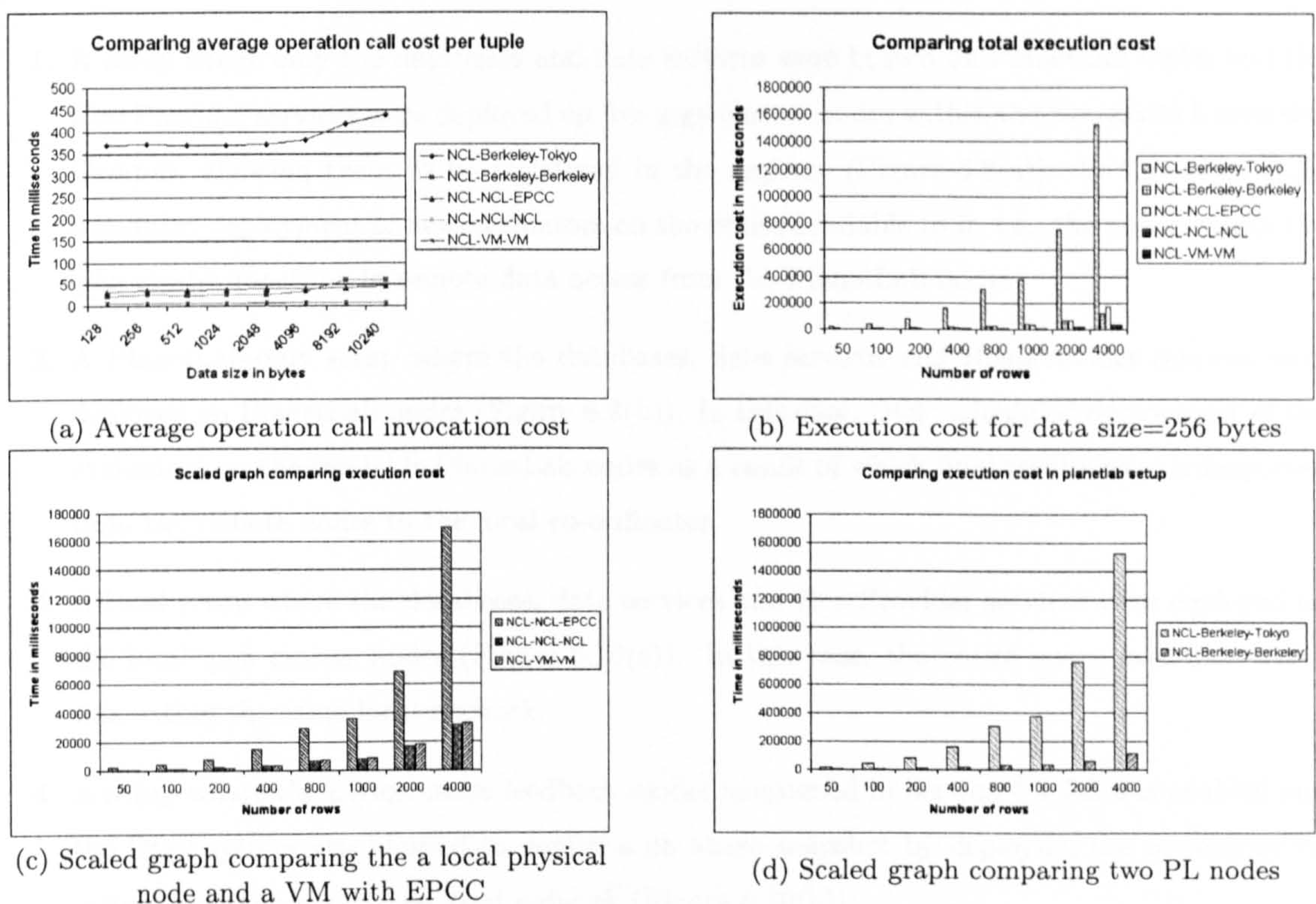


(c) Comparing total execution cost

Figure 6.7: Experimental results for the collocation of evaluation with data

6.2.5 Experiments on Virtualization

In order to investigate the performance of virtual machines, a set of experiments were performed which compared the results by executing a distributed query on various setups, such as - (i) a PlanetLab only setup, where the data was located on a node at Berkeley (USA) and the analysis service was hosted on a PlanetLab node at Tokyo (Japan), (ii) a PlanetLab only setup, where the analysis service was collocated with the data at Berkeley, (iii) a setup where the data was hosted on the Giga cluster at Newcastle and the analysis service was hosted on the node at EPCC, Edinburgh, (iv) a local setup where the analysis service was hosted along with the data on the Giga cluster nodes and (v) a setup where the analysis service and the data were hosted on a virtual machine deployed on a node within the Giga cluster. A *select-project-operation_call* query as in Listing 6.3 was used which retrieved the data from the database with different cardinalities and invoked the analysis service on each of the retrieved tuples and sent the result back to the consumer. The results for each of the DQP setups are compared in the graphs Figure 6.8.



(a) Average operation call invocation cost

(b) Execution cost for data size=256 bytes

(c) Scaled graph comparing the a local physical node and a VM with EPCC

(d) Scaled graph comparing two PL nodes

Figure 6.8: Comparing the performance of a VM with other setups for a distributed query

It can be seen from the results that the performance of the query when executed within a virtual machine is comparable to the performance of the same query when executed on nodes within the local network. The cost of invoking the analysis service, even when the service was hosted at EPCC,

Edinburgh, was extremely high compared to the cost when the service was local (either on a VM or a real host). The costs were even higher when the participating nodes were remote, such as in the case of the PlanetLab nodes.

6.2.6 Deploying a Database Snapshot Locally

All the experiments described in this section use the concept of deploying a snapshot of the data locally described in Section 5.1.5 which was enabled by allowing DQP to reconfigure itself based on the performance feedback from all the participating nodes (as described in Sections 5.2.7.1 and 5.2.7.2). The virtual machines mentioned in Section 6.2.1 containing snapshots of the *ProteinSequence* and *ProteinProperty* databases were uploaded to the *Software Registry.* For each experiment mentioned later in this section, four separate configurations as shown in Figures 6.9 and 6.10 were used for comparing the results, namely:

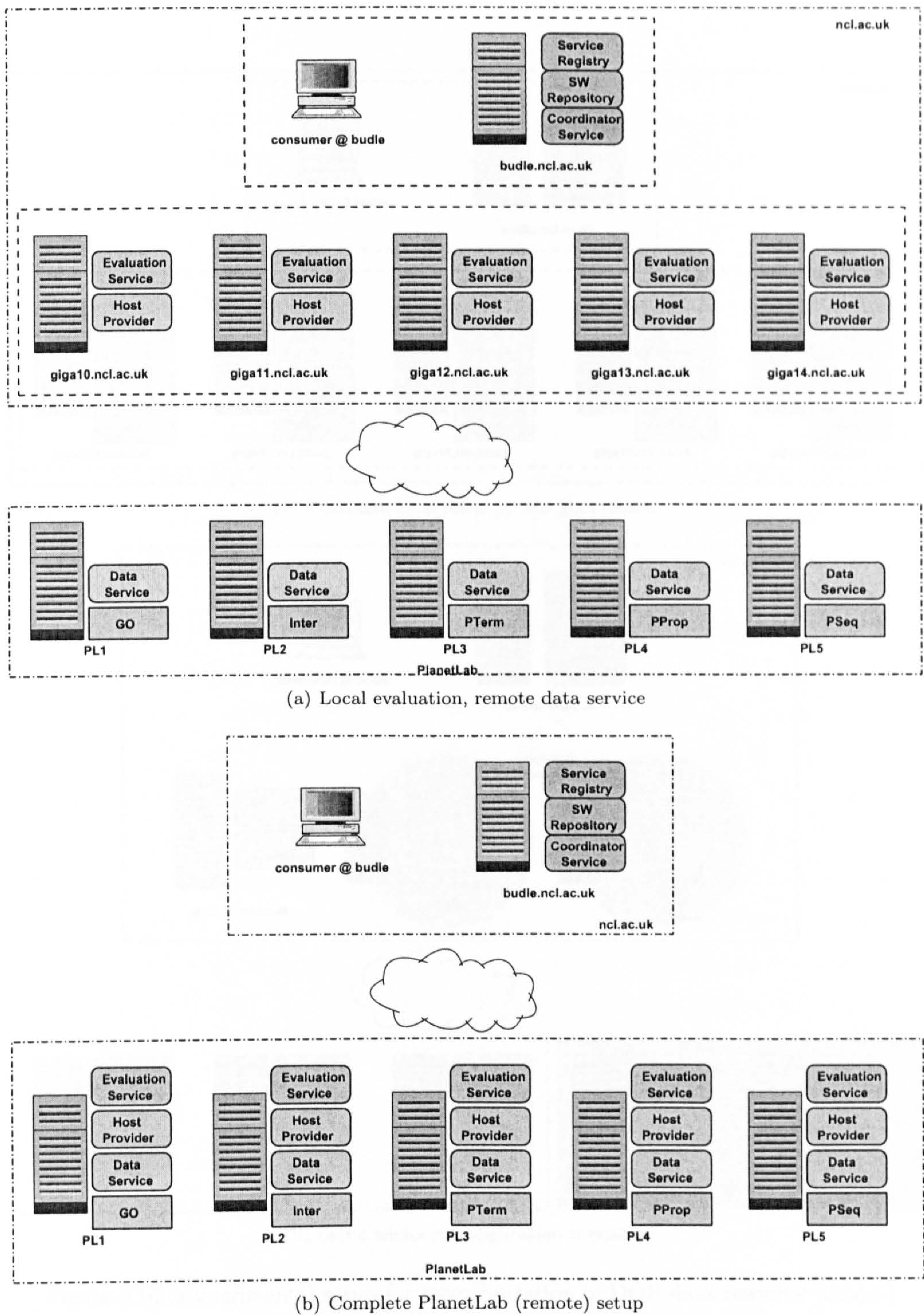
1. A setup where only the databases and data services were hosted on PlanetLab nodes and the HostProvider services were deployed on five giga cluster nodes within the Newcastle University Campus, allowing them to be registered in the registry (Figure 6.9(a)). In this case, DQP scheduled deployment of new evaluators on the nodes available to it, i.e., the nodes within the giga cluster resulting in remote data access from the PlanetLab nodes.

2. A PlanetLab-only setup where the databases, data services and HostProvider services were deployed on PlanetLab nodes (Figure 6.9(b)). In this case, DQP scheduled deployment of the evaluators on the available PlanetLab nodes as a result of which final results were transported from the remote nodes to the local co-ordinator.

3. A local setup where the databases, data services and HostProvider services were deployed on the local giga cluster nodes (Figure 6.10(a)). In this case, the entire query execution setup was within the same local network.

4. A setup where the performance feedback model mentioned in Section 5.2.7.2 was enabled and the DQP system was allowed to deploy a database snapshot by deploying the corresponding virtual machine within the local network (Figure 6.10(b)).

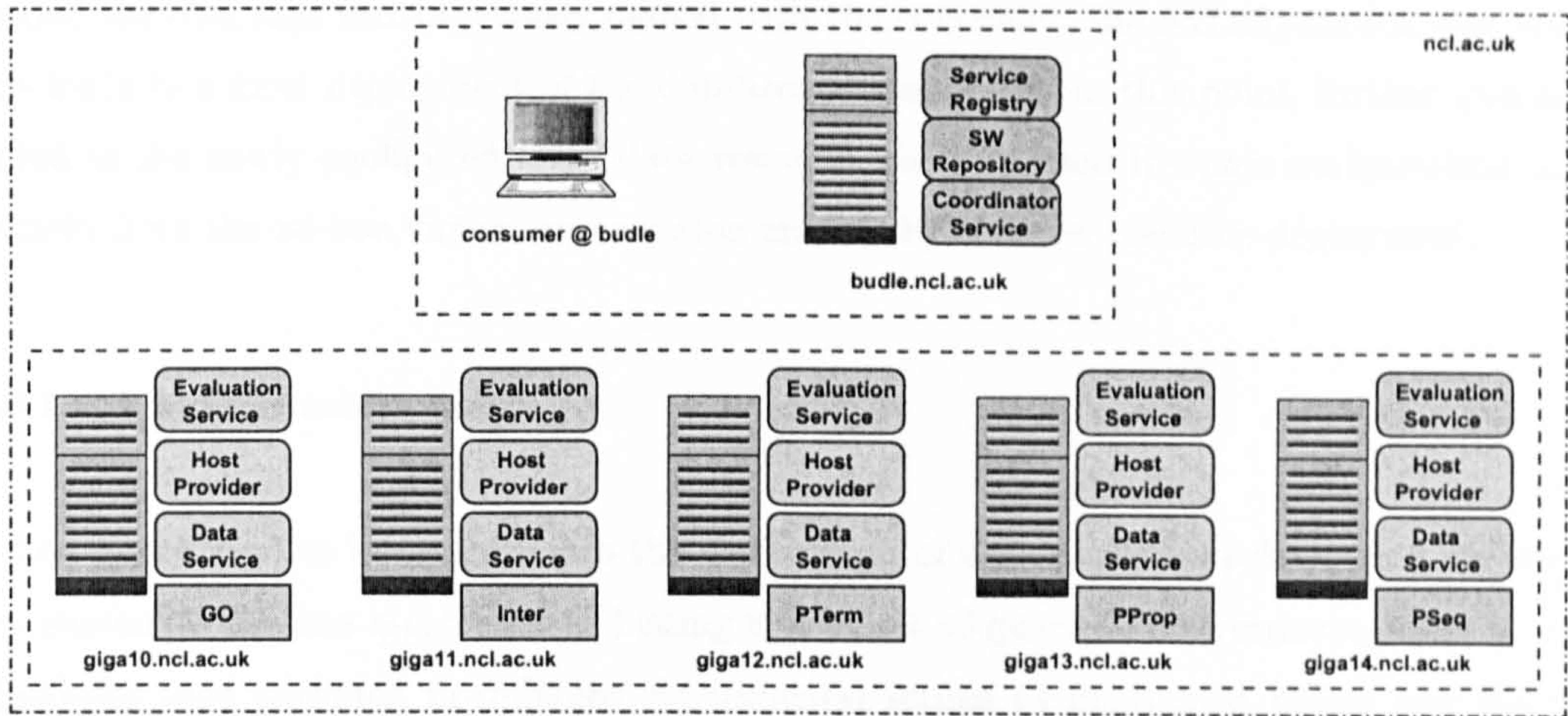
In each of the above mentioned configurations, a series of queries are submitted, each retrieving data of different cardinalities, and the results were collected. After the completion of each query, a request for performance data was sent to all the participating nodes by the DQP system, which were returned as XML documents. In the setup where the reconfiguration of the DQP data resource



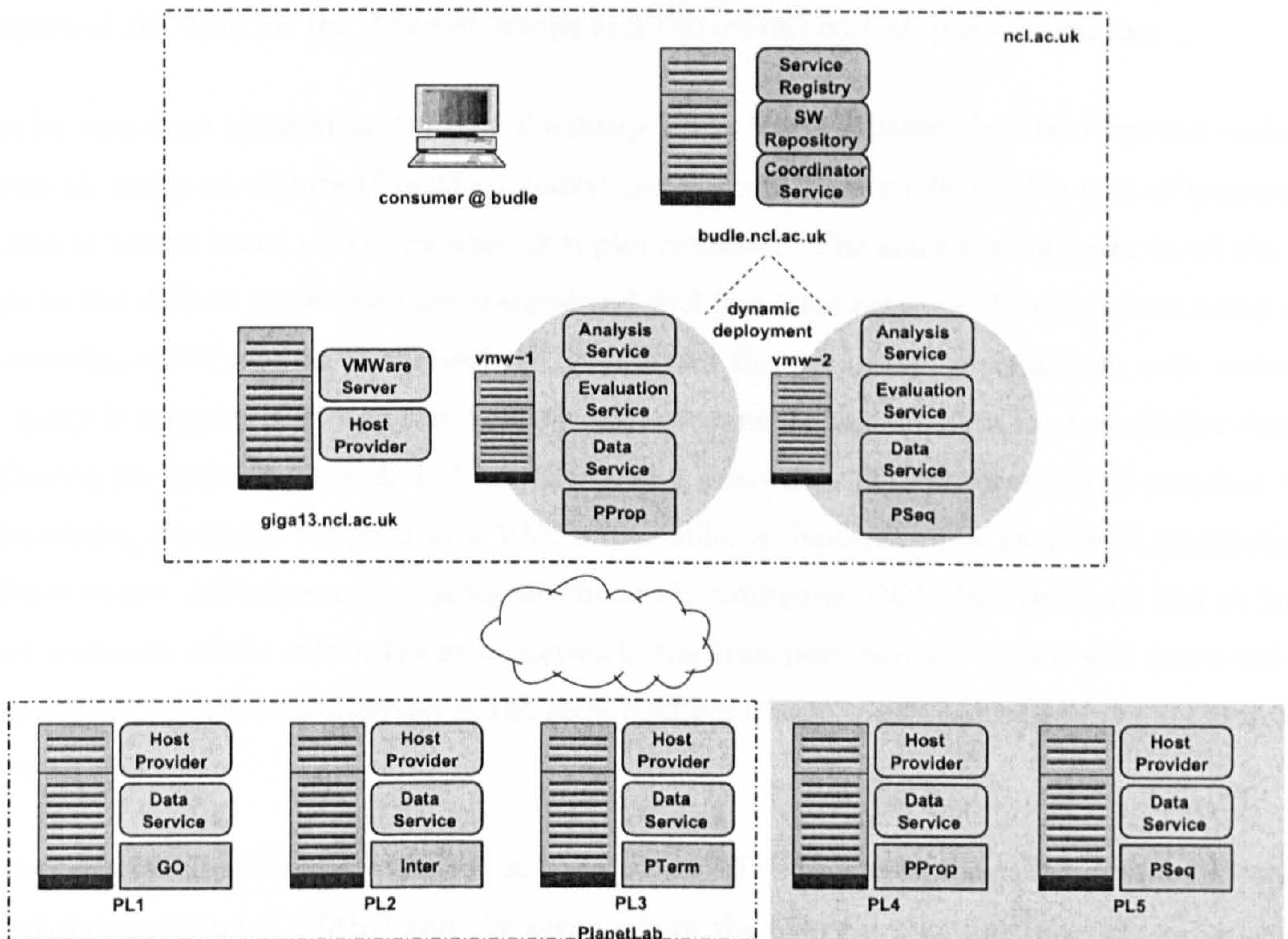
(a) Local evaluation, remote data service

(b) Complete PlanetLab (remote) setup

Figure 6.9: Experimental setup for reconfiguration of DQP data resource



(a) Complete local setup on the giga cluster



(b) Setup where reconfiguration is enabled

Figure 6.10: Experimental setup for reconfiguration of DQP data resource (contd.)

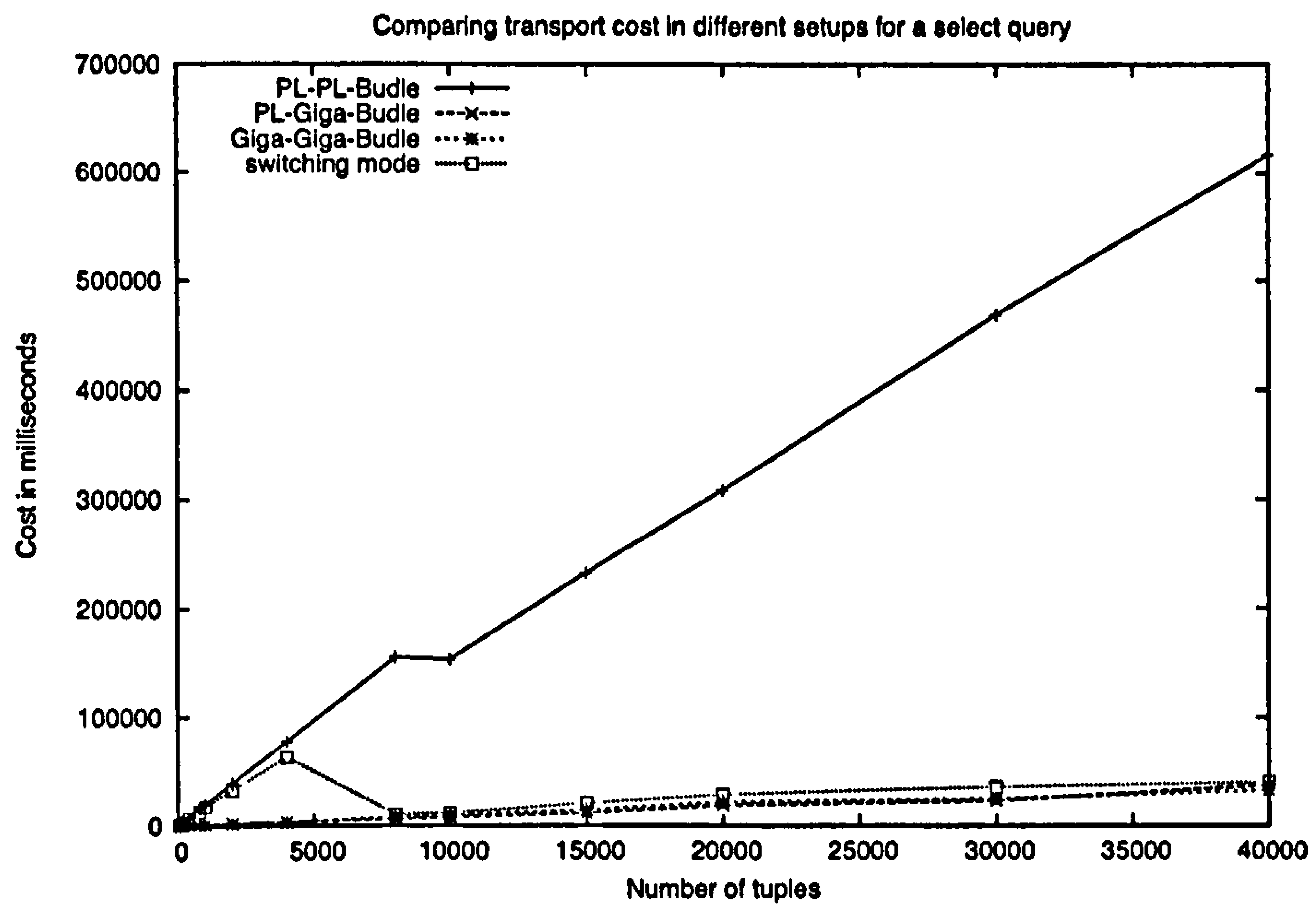
is enabled, the performance results were analysed and based on the analysis, if it was found that the cost of transporting the data from a particular database and evaluation service was following an increasing trend, and the cost was higher than the cost of deploying a snapshot of the database or exceeded the cost that was previously agreed with the consumer, the reconfiguration was triggered which leads to a local deployment of the database snapshot. From this point, further queries were directed to the newly configured DQP data resource which utilised the new configuration in order to benefit from the ad-hoc virtual organisation created by the new resource deployment.

6.2.6.1 A simple select query

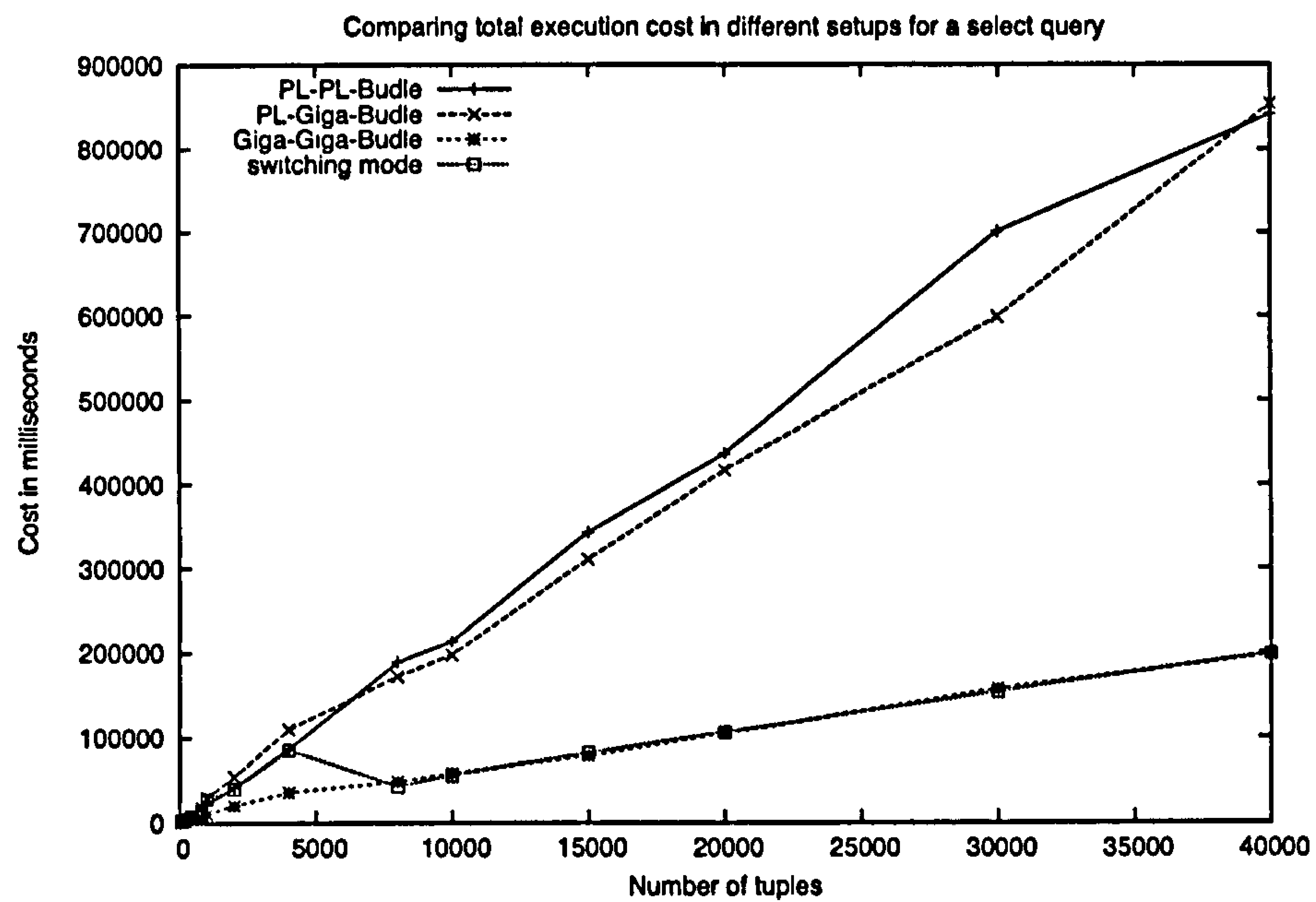
The first query used to investigate into the deployment of a database snapshot was a simple select query similar to the one mentioned in Listing 6.4. A set of queries each retrieving data of varying cardinalities were executed in different experimental setups as mentioned before and the results were collected. The graphs in Figure 6.11 show the results of this evaluation by plotting the cost of transport of the data for the different setups and the overall cost of query execution.

It can be seen from the graphs, that for the setup where the databases, data services and evaluation services all reside on remote PlanetLab nodes (as shown in Figure 6.9(b)), the cost of transporting the data increases based on the number of tuples retrieved. The same cost is lower in all the other setups as the data in these cases are transported within a local network. In case of the setup where the reconfiguration module is enabled, DQP analyses the performance data from each node after each query is completed. If it is seen that the cost of transporting the data for a particular database is following an increasing trend, and has exceeded a previously agreed threshold, a snapshot of the corresponding database wrapped in a VM, if available, is deployed on a local node containing the VMWare server. All subsequent queries use the newly configured DQP data resource, and as the new service instances reside within the same network, the transport cost is reduced and starts following the normal trend which is observed in the local configurations, which can be seen from the graph in Figure 6.11(a).

The overall execution costs are plotted in Figure 6.11(b). The overall execution cost for a complete PlanetLab setup(Figure 6.9(b)) and the setup where the query evaluation takes place on the local giga cluster, but the data is accessed from the remote data services on PlanetLab (Figure 6.9(a)) follow the similar increasing trend. This is because in the latter setup, the cost of data access nullifies the reduced cost of data transport. It is also seen from the plot that after the reconfiguration of the DQP data resource is complete, the overall execution cost is almost identical to the execution cost which is seen for a completely local setup.



(a) Comparison of the data transport cost from a data node



(b) Comparison of the total query execution cost

Figure 6.11: Comparing data transport cost and query execution cost using a remote setup and a setup which allows switching of a data node

6.2.6.2 A select-project-join query

In order to validate the rationale behind the concept of reconfiguring the DQP data resource, more complex queries were used which involved *joining* data from two databases. An example of such a query is shown in Listing 6.5.

Listing 6.5 A select-project-join query

```
select proteinsequence_random_sequence_256.id,
proteinsequence_random_sequence_256.sequence,
proteinproperty_random_property.hydrophobicity from
proteinsequence_random_sequence_256, proteinproperty_random_property where
proteinsequence_random_sequence_256.id = proteinproperty_random_property.id and
proteinsequence_random_sequence_256.id <= n and proteinproperty_random_property.id <= n;
```

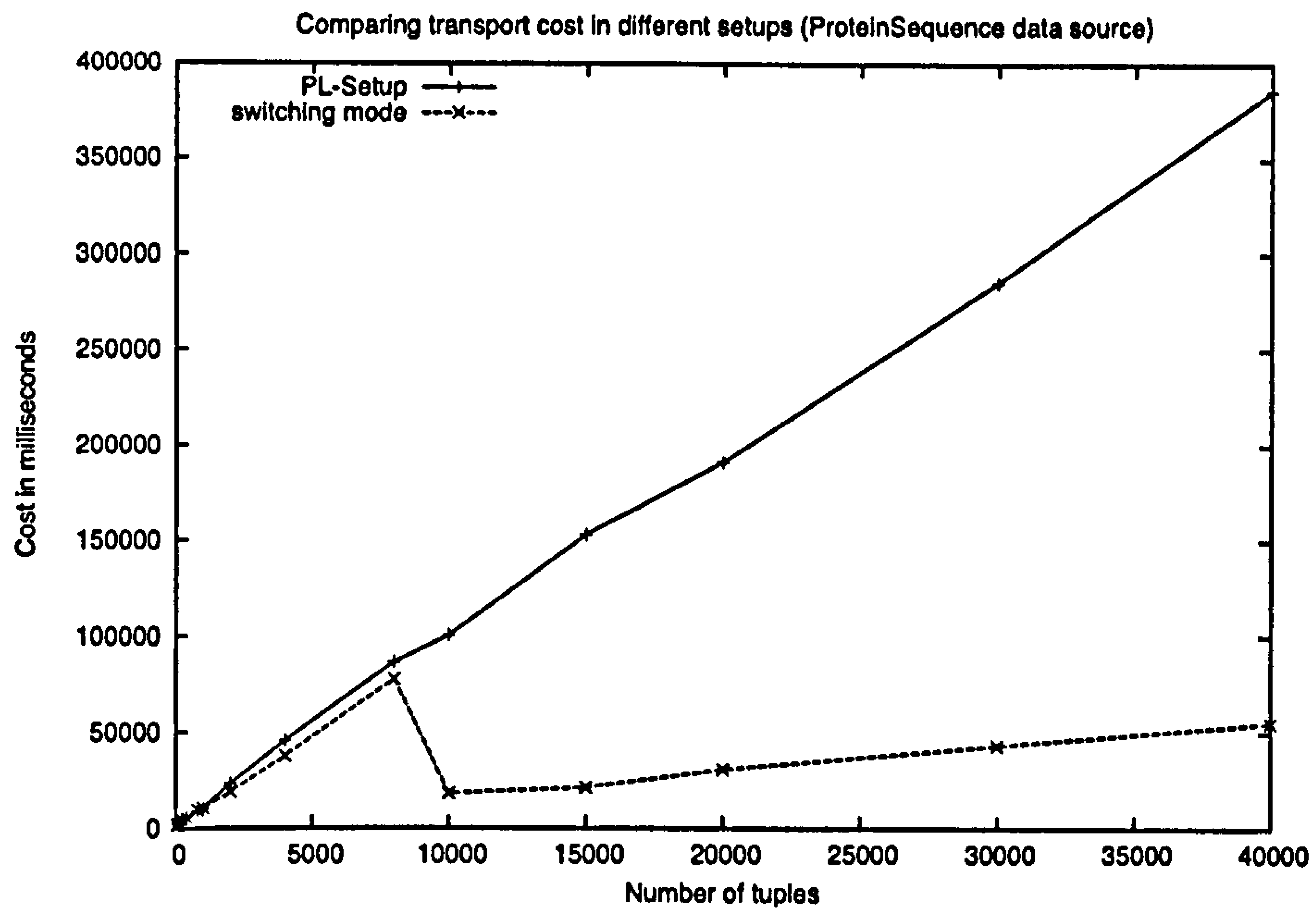
This query returns the sequence identifier, the sequence string and the hydrophobicity attribute after performing an *equijoin* on the identifier attribute from the two different databases. Results are of varying cardinalities depending on the value of "n". As in the case of the previous experiment, DQP performs a reconfiguration of the data resource when the cost of transporting data from a particular database exceeds the threshold, thereby creating a new instance of the data service which is used in the subsequent queries.

The results of this experiment are shown in the graphs in Figures 6.12 and 6.13. Figures 6.12(a) and 6.12(b) plot the cost of transporting the data from the *ProteinSequence* and *ProteinProperty* databases respectively and the values for a PlanetLab setup is compared with the setup where the reconfiguration module is enabled. The plot of the overall execution cost is shown in Figure 6.13.

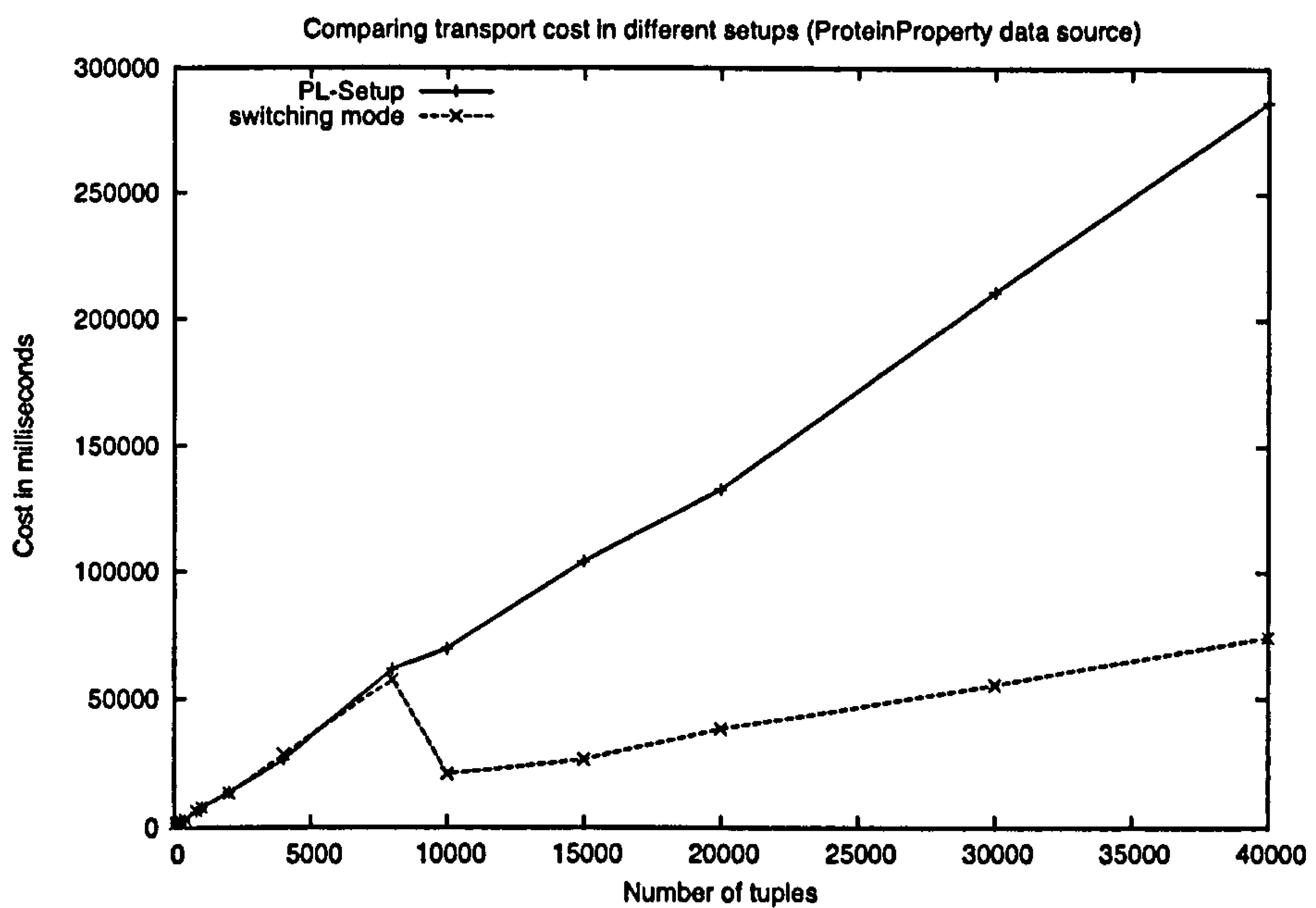
The *expected* result in the experiment was to observe a lower data transport cost and overall execution cost after the reconfiguration is completed by the DQP co-ordinator. This is observed in Figures 6.12(a) and 6.12(b) where, once the increasing trend in the cost of data transport is detected for both nodes hosting the *ProteinSequence* and *ProteinProperty* databases respectively, DQP deploys a snapshot for both of them on the local network. After the reconfiguration, the transport cost for both nodes reduces dramatically, as a result of which the overall execution cost is also reduced (shown in Figure 6.13).

6.2.6.3 A select-project-join-operation_call query

The final experiment to establish the rationale behind the deployment of a database snapshot involved a *select-project-join-operation_call* query as shown in Listing 6.6 using two databases, namely *ProteinSequence* and *ProteinProperty*. The query retrieved data of varying cardinalities depending



(a) Comparison of the data transport cost from the node hosting ProteinSequence DB



(b) Comparison of the data transport cost from the node hosting ProteinProperty DB

Figure 6.12: Comparing data transport cost for two data nodes for remote and switching setup

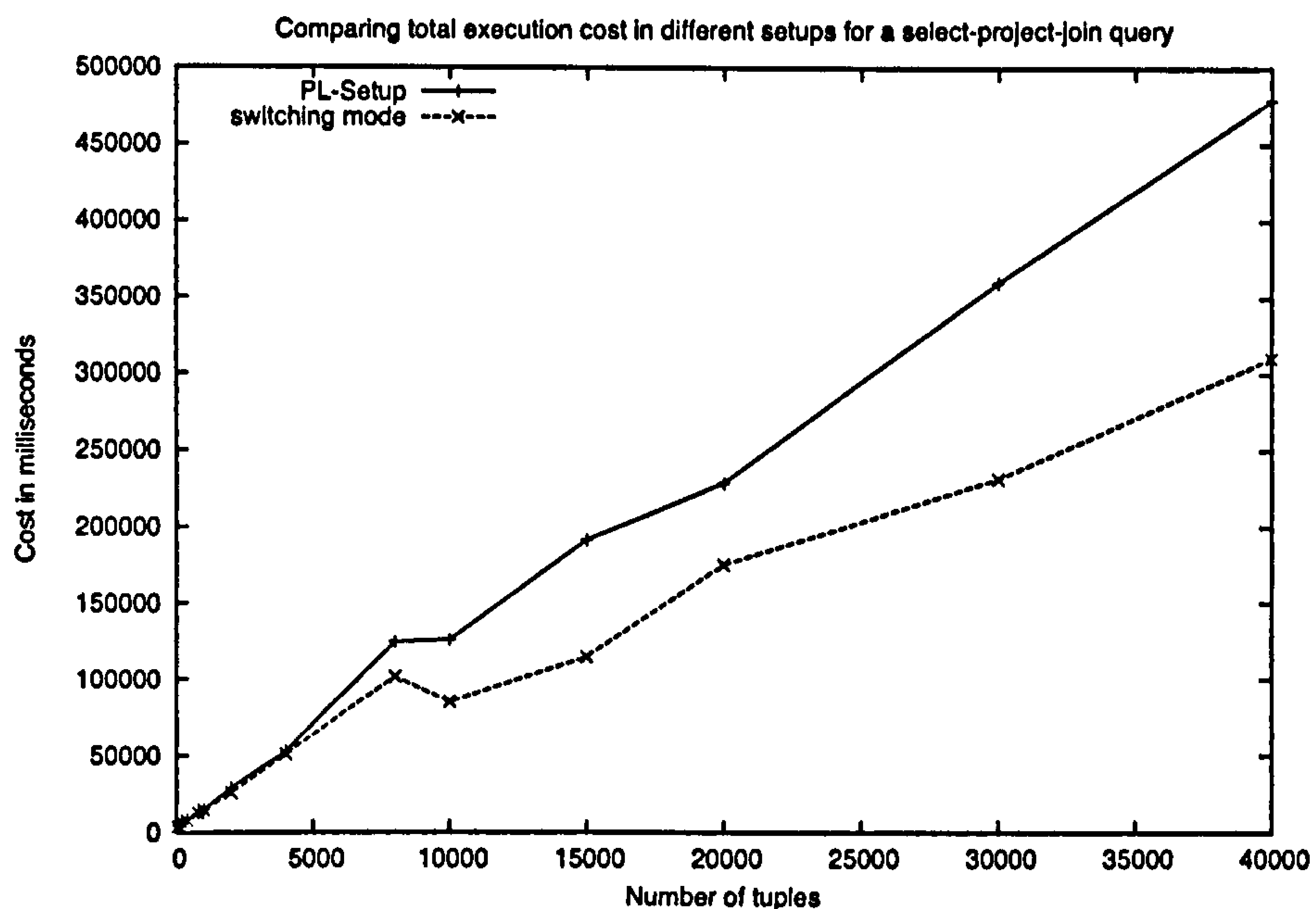


Figure 6.13: Comparing execution cost of a select-project-join query for a remote and switching setup

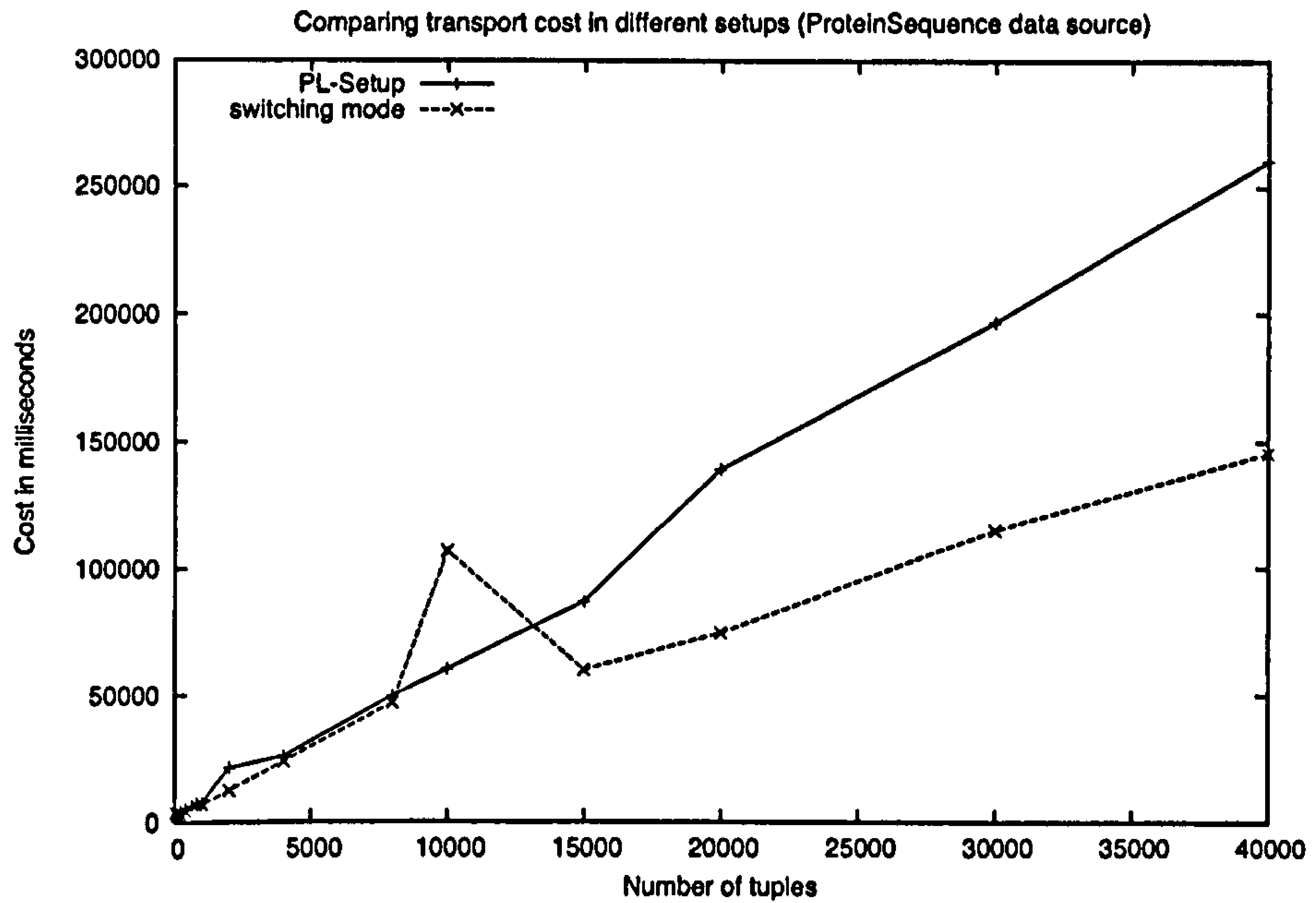
on the predicate of the select clause. The retrieved data consisted of the identifier, the *sequence* attribute and the *hydrophobicity* attribute. An *equijoin* was performed on the tuples based on the *identifier* attribute and an analysis service was invoked with the *sequence* attribute as input.

Listing 6.6 A select-project-join-opcall query

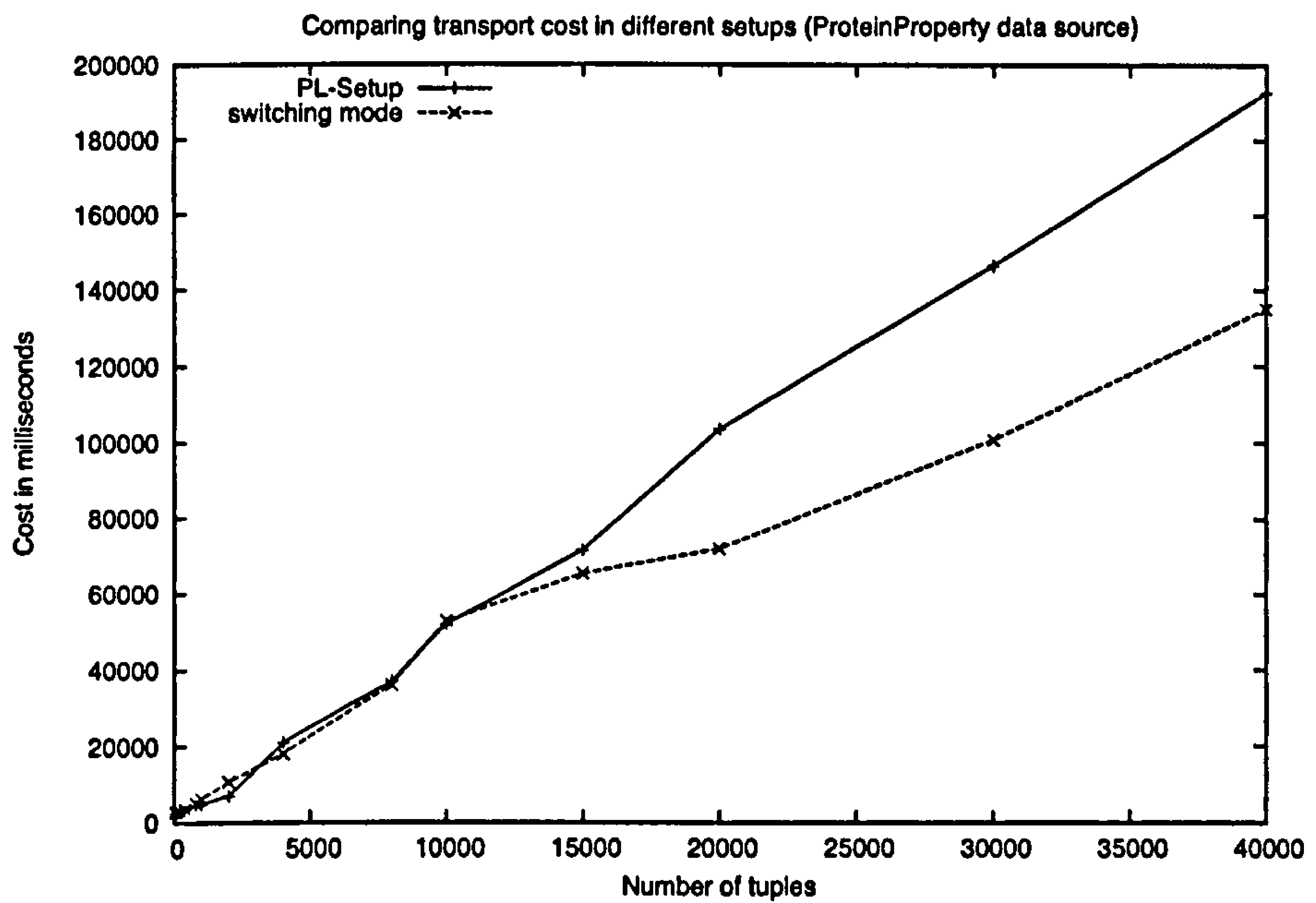
```
select proteinsequence_random_sequence_256.id,
calculateEntropy(proteinsequence_random_sequence_256.sequence),
proteinproperty_random_property.hydrophobicity from
proteinsequence_random_sequence_256, proteinproperty_random_property where
proteinsequence_random_sequence_256.id = proteinproperty_random_property.id and
proteinsequence_random_sequence_256.id <= n and proteinproperty_random_property.id <= n;
```

As in the case of all previous experiments, DQP collected the performance data from each of the participating nodes and analysed them after the completion of each query. When the analysis showed an increasing trend for the data transport cost from a particular database, a snapshot of that database was deployed locally. The results of this experiment are plotted in the graphs shown in Figures 6.14 and 6.15. Figure 6.14(a) plots the cost of data transport for the *ProteinSequence* database and Figure 6.14(b) plots the same cost for the *ProteinProperty* database. The plot for the overall execution cost is shown in Figure 6.15.

The *expected* result was a reduced transport cost after the data resource reconfiguration takes place. However, Figure 6.14(a) has an interesting feature. The cost of data transport for the node hosting



(a) Comparison of the data transport cost from the node hosting ProteinSequence DB



(b) Comparison of the data transport cost from the node hosting ProteinProperty DB

Figure 6.14: Comparing data transport cost for two data nodes for a remote and switching setup

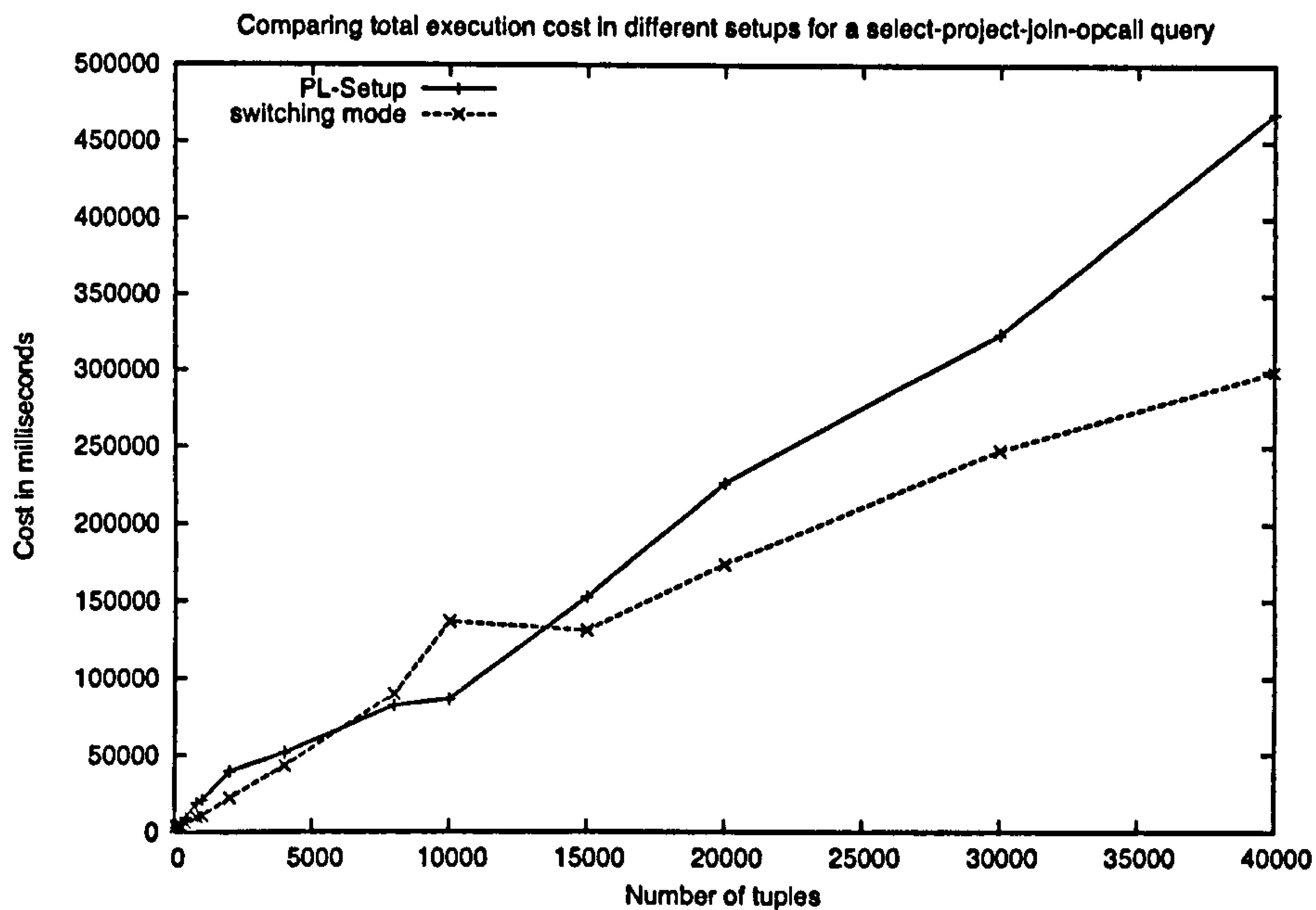


Figure 6.15: Comparing execution cost of a select-project-join-operation call query for a remote and switching setup

the *ProteinSequence* database momentarily increases after the *ProteinProperty* database is deployed locally. The reason behind this can be explained by considering the location of the participating nodes and the evaluation process in DQP. All the participating nodes in DQP exchange data between themselves during evaluation. The *join* operation in this query is parallelised on two nodes hosting the two relevant databases. Initially, both the nodes are from the PlanetLab domain, and both are located in the United States of America, which means they were closer in terms of network latency than when the *ProteinProperty* database was deployed locally. When the first reconfiguration takes place, the network latency between the new node on the local network and the PlanetLab node hosting the *ProteinSequence* database increases as they are farther apart in terms of network connectivity than before. Hence, the cost of transport as seen from the PlanetLab node hosting the *ProteinSequence* database showed an increase, thereby affecting the overall query execution cost, shown in Figure 6.15. DQP subsequently reconfigures the *ProteinSequence* database by deploying the snapshot on a local node, and the transport cost as well as the overall query execution cost reduces to what is normally observed within the local network.

6.2.7 Evaluating Availability Issues

The analysis services used in DQP, or for that matter, in any service oriented framework, are often maintained by third parties. It has been noted in Section 5.1.4 that such services may be unavailable during the query execution which may result in execution failure. The dynamic deployment features of DynaSOAr have been used within the dynamic version to DQP to partially deal with such situations. If during DQP initialisation phase, a particular instance of an analysis service is provided within the configuration script, the GDQS imports the WSDL from that endpoint. If the endpoint is not valid during the schema import phase, i.e., the WSDL can not be obtained using the standard WSDL import process, the GDQS will search the registry for the availability of the deployment package of that particular service. If the service is available in the repository, the GDQS will be able to find an entry within the registry which would point to the endpoint of the service code, which will then be dynamically deployed on the nodes selected by DQP.

A service failure may also occur when a query is being processed which will result in the execution failure. In such situations, even if the service is deployed dynamically, there might be loss of valuable data which flows into each evaluator from other evaluation services. This is a challenging issue which was not a focus for this thesis. However, it may be resolved by using the dynamic deployment features along with the checkpointing approach proposed in [139].

6.2.8 Services Requiring Special Environments

Another major rationale behind the use of virtualization technologies within DQP is the possibility of using service that require special environment, such as a special set of libraries or databases or even platform, which was noted in Section 5.1.6. Such services can be packaged with all the dependencies within a virtual machine, which can then be uploaded to the software registry. This results in the packaged services being entered in the service registry to allow discovery. If such a service is named within the initial configuration script submitted to the GDQS, the virtual machine is downloaded to an appropriate node which contains the VMWare server and started, which results in the service being made available. Subsequent queries containing an invocation to the service will be using this newly deployed endpoint. This has been validated by wrapping a version of the analysis service that has been used in the experiments within a virtual machine and deploying it dynamically on one of the local nodes within the giga cluster and by submitting queries which resulted in this service being invoked. It has been observed in all the experiments that the response time of a service deployed on a virtual machine is almost equivalent to the response time of the service when deployed on a local physical node.

6.3 Discussion

In this chapter, a brief discussion on the implementation of OGSA-DQP and DynaSOAr was provided preceding the explanation about the experimental setup used for the evaluation. The evaluation of DynaSOAr itself has been presented in other works, such as [19] and [98] and thus has been excluded in this thesis. This chapter presented the evaluation of the concepts proposed regarding dynamic OGSA-DQP in Chapter 5.

Several experimental scenarios have been considered for the evaluation purpose, each making an attempt to validate the conceptual proposals made in Section 5.1, such as the collocation of an analysis service closer to the data, or the proactive deployment of multiple copies of the analysis service to balance to load between the instances, or the deployment of a database snapshot within the local network. For each experiment, different configurations were used to prove the efficiency of the dynamic deployment features incorporated in DQP. The observed results validate the claims made in Chapter 5 and show an overall improvement in the query execution process. An interesting observation was made in Section 6.2.6.3 where the local deployment of the snapshot of one database temporarily increased the data transport cost as seen at the node hosting the other database involved in the query.

Few experiments were performed in order to investigate into the issues regarding the availability of third party maintained web services and for services requiring special environments. The dynamic version of OGSA-DQP resolves the issues regarding the failure of a third party web service partially by deploying a copy of the service, if available, during the schema import phase. Failure of a service during the query execution phase however requires a more sophisticated approach such as the checkpointing approach proposed in [139] apart from the dynamic deployment framework. The dynamic deployment features along with the virtualization model allows the deployment of services requiring special environments by packaging them within a virtual machine and making the entire virtual machine available via the *Software Repository*.

In the final chapter, the thesis will discuss the overall findings of the research and possible further work.

Chapter 7

Conclusions

In this concluding chapter of the thesis a summary of the research and its contributions is presented. A discussion about the opportunities for further work and possible improvements to the concept of *dynamic service provisioning in distributed query processing* is also presented.

7.1 Summary and Discussion

Arthur D. Little, the founder of the world's first management consulting firm, once said

> *"Research serves to make building stones out of stumbling blocks."* [145]

The foundation that is built with these building stones allow further research to continue. But this progression requires a knowledge about the foundation just as in case of buildings. This section summarises and discusses the foundation, that is, the main work and contributions presented in Chapters 3, 4 and 5. The claims made in the introductory chapters are scrutinised with respect to the results obtained during the evaluation to verify their validity.

7.1.1 Service Oriented Distributed Query Processing - Chapter 3

This chapter introduces the concept of *service orientation* to *distributed query processing* and provides an architectural description of a *service oriented distributed query processor.* This new service-

oriented DQP system, commonly known as *OGSA-DQP*, was the result of collaborative research between Manchester and Newcastle Universities, and has made a positive impact on the e-Science community, amongst researchers who want to perform distributed query processing over disjoint databases and perform analysis on the results. Since its first release in September 2003, it has been downloaded over 800 times from all over the world.

The architecture of OGSA-DQP (Section 3.2) is based on service-orientation where each component is factored out as a service. The query compilation/optimisation process takes place within a Grid Distributed Query Service (GDQS), commonly known as the coordinator. The coordinator creates an optimised query plan which is then partitioned and submitted to one or more Query Evaluation Services (QES), commonly known as the evaluators. The evaluators access data from databases exposed using OGSA-DAI services. Thus it will be reasonable to say that OGSA-DQP is service-based in two orthogonal senses - (i) it allows resource virtualization by supporting queries over distributed data sources and analysis services that are factored out as services, and (ii) the process of query compilation, optimisation and evaluation takes place within a set of interacting services. It was also claimed in Section 3.1 that OGSA-DQP can be considered as an approach complimentary to other *service orchestration* mechanisms, such as workflows as typical workflows executed by researchers in bio-informatics have similarities with a section of queries possible in OGSA-DQP.

OGSA-DQP requires initialisation before it is able to process queries. During the initialisation phase, the schemas from the databases and the analysis service(s) are imported which are used during the query compilation process. The query compilation/optmisation process applies parallel database techniques as proposed in [13], [63] and [62]. Initially, the GDQS re-used the Polar* component implemented earlier ([62]) as a black-box component, and later on, a Java-based compiler was introduced which was able to compile SQL queries. The approach to query compilation remained similar as in case of Polar* regarding the use of parallel database techniques.

Apart from the contribution in defining the broad architecture of OGSA-DQP and the underlying concepts, the other major contribution towards this thesis has been the work on the runtime query evaluation system, or the evaluators. The QESs or the evaluators have been exposed as services based on standard WS-I [20] guidelines and principles. Each evaluator receives a query partition structured as an XML document and the evaluation process follows the classical iterator model, originally proposed by Graefe et. al. in [24] and [66]. It was a challenge encapsulating the iterator model within a service, as in such a model, the state of the query evaluation process is extremely important. Each of the participating evaluators exchange messages between them while evaluating a query, and the process of distributing the messages is encapsulated within a special *exchange* operator. Thus, it is very important that the evaluators are able to correlate the messages they

receive with the queries that are being evaluated as each evaluation service may be processing multiple queries simultaneously. In essence, the principles followed in the design and implementation of the evaluators are similar to the proposals made in WS-GAF [27] about using the standard Web Service technologies to achieve the stateful interactions. The motivation behind the use of the standard WS-I technologies was the requirement for mobility so that the evaluation services can be deployed at run time on nodes that are deemed best suited for the evaluation of a particular query which would create a loosely coupled and fluid run time architecture.

One primary objective of the thesis was to propose the architecture of a DQP framework which allows homogeneous access to heterogeneous data resources by using existing infrastructures, such as OGSA-DAI and standard Web Service technologies. This objective has been fulfilled considering the involvement in the broad architectural design of OGSA-DQP and the work on the run-time evaluation system which follows the WS-I standards.

7.1.2 Dynamic Service Deployment - Chapter 4

The need for a more dynamic framework where services could be deployed on demand on available hosts was felt when the core OGSA-DQP was being developed. The original OGSA-DQP was tightly coupled with a set of resources, both computational and data in a sense that only the hosts which already had the evaluation or analysis service deployed were used. There was no notion of on-demand deployment which would allow new potential resources to be considered for query evaluation or make an attempt to optimise the evaluation process by collocating different entities involved in the evaluation process.

At the same time, a considerable shift in the focus was observed in the e-Science domain, where more and more research projects started adopting the emerging service oriented technologies with a view to benefit from the Grid. Traditionally, Grid computing was based on distributed job-schedulers which form the core of Grid frameworks like Condor [3, 2], Globus [4] or Sun Grid Engine [111]. But with the advances in the service-oriented technologies and emergence of new standards and toolkits supporting Web Services, a need was felt for a Grid computing infrastructure supporting on-demand resource allocation such as the traditional frameworks, but based on service-orientation. Further, the execution of a job in a job-based environment is an "one-time" affair, whereas, in many e-Science researches, there are requirements for executing the same workflow or service multiple times with different inputs. In a job-oriented paradigm, this would mean submitting the executable code along with the input data each time, whereas, a service, once deployed, would remain so, unless specifically undeployed by an administrator, thus allowing a "deploy once, use multiple times" philosophy, which

fits with the requirement for e-Science research.

Thus with a similar motivation, both for advancing the research into OGSA-DQP and the other e-Science projects adopting the service-orientation paradigm, a concept of a framework allowing the dynamic deployment of Web Services was conceived, which came to be known as DynaSOAr, or, Dynamic Service Oriented Architecture, which was introduced in Chapter 4. The arguments in favour of DynaSOAr were - (i) a simplified development process concentrated only on services, (ii) a possibility of improved performance as the service is retained on the host unless specifically undeployed thereby distributing the deployment cost over multiple invocations to the service and (iii) the logical separation of service provisioning and host provisioning which would allow new organizational/business models. The evolution of the concepts behind DynaSOAr, such as the motivations from the Active Information Repository architecture [7], the requirement for "loose-coupling" and "execution transparency" and formation dynamic virtual organisations were outlined in Section 4.2 - all of which contributed to the requirements for the DynaSOAr architecture. Two deployment patterns were described in Section 4.2.5 and the DynaSOAr architecture ensures that the service invocation procedure from the consumer point of view does not change and the consumer is able to use conventional tools and procedures for invoking a Web Service. The design of DynaSOAr including each component was described in Section 4.3. DynaSOAr was developed as a collaborative work by several researchers and apart from the contribution towards the overall architecture of DynaSOAr, the development of a message-oriented framework and the incorporation of a registry service are particularly important for this thesis. These, along with the introduction of a broker, the use of one or more registries and the handshaking between available resources were the contributions towards knowledge during the course of the thesis. Collectively, they contribute towards the formation of a *Software Hypermarket* where consumers will have the option of choosing between multiple *service providers* and *host providers* based on certain parameters such as trust, cost and quality of service.

The version of DynaSOAr described in this thesis relies on "virtualization" technologies such as VMWare in order to create ad-hoc "virtual organisations" which enables new organisational models for collaboration in the scientific domains. An overview of the available virtualization technologies is provided in Section 4.4.1 which outlines the advantages that can be obtained from their use. Section 4.4.2 provides a couple of scenarios, such as *data caching* and *support for services requiring special environments* which are considered to be important in the context of this thesis. The question about how these technologies are used within DynaSOAr is answered in Section 4.4.3. The most important aspect in DynaSOAr is that, it considers the virtual environments with an approach similar to the approach taken in case of Web Services, and from a consumer point of view, this happens in a completely transparent manner, thus keeping the perception of *execution transparency* intact.

DynaSOAr has been evaluated in previous publications, such as [19] and by Fowler in his thesis [98]. Initial experiments with DynaSOAr showed promising results regarding the performance of the system when services were deployed dynamically. This laid the foundation for the use of these concepts within other frameworks where a requirement of such dynamic deployment was felt. Preliminary work started in order to exploit the concepts within the OGSA-DQP framework [128] where the effects of collocating an analysis service with the data was observed and the initial results were encouraging. The possibility of adopting the virtualization technologies within the OGSA-DQP context was explored in [146], and showed encouraging results during the evaluation (Section 6.2.5). The exploitation of the DynaSOAr concepts within the DQP framework thus gained considerable momentum and led to the research into a dynamic version of OGSA-DQP. The concepts of DynaSOAr, including the virtualization approach have been adopted in the OGSA-DQP framework in order to exploit the possibilities of dynamic deployment within the context of *distributed query processing*, which is considered as the main contribution towards the thesis.

7.1.3 Exploiting Dynamic Service Provisioning in DQP - Chapter 5

The aim of the entire research was

> *"to create a dynamic distributed query processing framework for a Grid environment allowing co-ordinated resource sharing and on-demand deployment of data-sources, evaluation and analysis services on available computational resources."*

This was achieved in Chapter 5 where the thesis attempted to investigate the possibilities of exploiting the features of DynaSOAr into the distributed query processing system, OGSA-DQP, which was discussed earlier. Certain scenarios are outlined in Section 5.1 where the on-demand deployment features of DynaSOAr can be useful. These include the collocation of the analysis and evaluation service with the data, increased degree of parallelism by a pro-active deployment of the analysis service and the deployment of a database snapshot in case of frequent similar queries over the same dataset. The DQP system can benefit from the use of virtualization technologies in case of such deployment of database snapshots or for services which require a special environment or considerable tuning with the underlying system. Figure 5.1 shows the ad-hoc virtual organisation that may be created by DQP with the dynamic deployment features during the query evaluation process.

The overview of the dynamic OGSA-DQP system which incorporates the on-demand deployment features of DynaSOAr is given in Section 5.2.1. A comparison is made with the static OGSA-DQP system where the evaluation and analysis services are bound to certain hosts resulting in a

tightly coupled system. This limits the scope of exploiting the dynamism of a Grid environment where resources are often volatile in nature. On the other hand, the dynamic DQP system offers a collection of resources grouped as pools, such as a *Software pool* or a *Virtual Machine pool*, and nodes are effectively *created* on-demand. It is however assumed that the DynaSOAr framework exists on the available nodes which would allow dynamic deployment to take place, and that certain features may be incorporated within the standard Web Service containers to allow such deployment, at which point, the requirement for DynaSOAr components such as the HostProvider may not be necessary.

The detailed architecture of the dynamic DQP system is described in Section 5.2.2 where the use of the DynaSOAr components, such as the *Software Repository*, *Service Registry* and *Host Provider* are outlined. The interactions in DQP differ from DynaSOAr in that the participating services must interact with each other directly. Hence the DQP coordinator, which is the entity equivalent to the DynaSOAr *Service Provider*, needs to know the actual endpoints of the newly deployed services which are used within the query plan produced by the compiler. Later sections elaborate on the initialisation and query compilation activities of the DQP system, where the new features for dynamic deployment are considered. The DQP data resource is able to collect the set of available resources from the registry and all resources are considered during the generation of the query plan irrespective of whether the evaluation service exists on the resource or not. A plan is generated which uses the resources that are best suited for evaluating the particular query, and if required, services are deployed on these nodes. Once a service is deployed on a resource, it stays there, and can be used by the DQP system without any further need to redeploy the service on it. DQP uses a simple method of sending a fixed sized network packet to the computing nodes for collecting the network latency between the available nodes, and it is possible to replace this with a sophisticated network monitoring tool which will lead to more accurate results. As shown in Figure 5.5, DQP makes an attempt to collocate the data, analysis and evaluation service in order to gain an improvement in performance. The use of virtualization technologies in DQP is described in Section 5.2.7 where a performance feedback model was developed in order to collect performance data from all participating nodes. Based on the performance, and a previously agreed threshold limit on the cost of transporting data from a particular node, a reconfiguration may take place within the DQP system. If it is observed that for a similar set of queries on a particular DQP resource, the cost of transporting the data from a particular node hosting a data is following an increasing trend, and the cost exceeds the previously agreed cost that was acceptable by the consumer, a snapshot of that database if available, is deployed as a virtual machine on a local node, and subsequent queries use this newly deployed snapshot. It is assumed that some offline process will be used to keep the snapshot synchronized with the actual dataset.

The dynamic deployment features are evaluated in Chapter 6 to challenge the claims made in the thesis about the benefits of dynamic deployment. Several different DQP setups were used in order to show the benefits of collocation of various services, the advantages of virtualization technologies etc in comparison with the earlier static OGSA-DQP system. It is possible to break down the objective of *exploiting dynamic service provisioning within distributed query processing* into finer objectives, which, along with whether they were validated during the evaluation are discussed:

- *Collocation of the analysis service with the data* - One of the claims made in Chapter 5 was about a possible improvement in the performance of DQP if the analysis service could be collocated with the data. The rationale behind the claim was that this collocation would reduce the cost of invoking the analysis service, which is normally high when the service is remote, which in turn would reduce the overall cost of query execution. The experiments described in Section 6.2.2 clearly show that such collocation of the analysis service with the data was able to reduce the overall cost of query execution quite dramatically.

- *Collocation of the query evaluation engine with the data* - It was thought that collocating the query evaluation engine with the data would reduce the cost of accessing the data from the database thereby reducing the overall cost of query execution. The evaluation of this claim in Section 6.2.4 however had a mixed outcome. The cost of accessing the data was reduced, but when the evaluation engine itself is remote, the cost of transferring the processed data becomes higher for queries returning a large resultset. This highlighted the requirement of using a more robust method of transferring tuples between the DQP services.

- *Increased degree of parallelism for analysis service invocation* - It was envisaged that the pro-active deployment of the analysis service on multiple hosts would allow the query optimiser in DQP to parallelise the *operation_call* operator, which would actually distribute the tuples across multiple endpoints of the analysis service. This distribution would result in a better performance because of the inherent partitioned parallelism within the DQP evaluation process. This claim was validated by the experimental results analysed in Section 6.2.3.

- *Deployment of a database snapshot* - The performance of DQP using virtualization technologies was analysed in Section 6.2.5 which showed that the performance of virtual machines within the local network are equivalent to real physical hosts and does not affect the performance of the query evaluation system in any adverse way. The objective of bringing the data closer to the analysis code by deploying a snapshot of the database locally in situations where frequent queries are submitted against the same dataset, and the use of a snapshot does not affect the queries adversely, is realised by the deployment of virtual machine instances. The experiments in Section 6.2.6 successfully defend the claim made in the thesis.

- *Services requiring special environments* - One of the objectives was to enable deployment of services which require special environments, such as a special database or a special set of libraries. This was realised by encapsulating such services within a virtual machine and deploying the VM on demand. This procedure also created the possibility of deploying services which require considerable tuning with the underlying system. Such services can be deployed on a VM and tuned properly before storing the VM in the repository. Deployment of the VM would deploy the service, which would already be tuned with the underlying system. These claims have been validated by the experiments described in Section 6.2.8.

The incorporation of dynamic service provisioning features within OGSA-DQP enables the possibility of the *software marketplace*. A consumer may want to use a certain analysis service within the queries, and the service may be hosted by different host providers, or may be provided by different service providers. The consumer may have a set of quality of service requirements or provider preference, using which it may be possible for the DQP system to select the best suited host or service provider. It may also be possible to establish *Service Level Agreements (SLAs)* during the initialisation phase where the consumer submits the set of databases and services required for the queries, based on which the DQP data resource is created.

7.1.4 Summary of Contributions

Some of the research mentioned in the thesis was the result of collaboration between several researchers in different projects. For example, OGSA-DQP was the result of collaborative research between Manchester and Newcastle Universities. DynaSOAr was the result of the research into dynamic deployment by several researchers at Newcastle University. Thus, this thesis does not and should not claim the complete credit for such collaborative research. The following list summarises the contribution towards knowledge made by this thesis.

1. **Service-oriented Distributed Query Processing System** was a result of a collaborative work between Manchester and Newcastle Universities, supported by other researchers in OGSA-DAI [14] and the *myGrid* [6] project. The contribution made by this thesis lies in the overall architecture and design of the DQP system and the philosophies behind it and the creation of the run-time query evaluation engine which encapsulates the iterator model of query evaluation within a service which evaluates a query partition by processing the plan submitted to it as an XML document.

2. **Identifying the case for dynamic deployment** where the scenarios within OGSA-DQP

where the dynamic deployment features would benefit both the consumers and the query execution process.

3. **Dynamic Service Oriented Architecture framework** was another collaborative work where several researchers made their contribution. The contribution towards the overall design and architecture of the system, and more specifically the incorporation of the concepts behind the use of *registries*, *brokers*, the *message-oriented* model and the generic hierarchy leading to the convergence of multiple organisations into logical virtual organisation account for the contribution towards this thesis.

4. **Evolution of a Software Hypermarket** is a conceptual feature within the DynaSOAr framework which allows consumers to choose between available service and resource providers based on parameters such as trust, security, quality of service, cost etc.

5. **Virtualization in DynaSOAr** is a major contribution in which virtualization technologies are used to deploy services that require special environments or considerable tuning to the underlying system, and also to deploy database snapshots for situations where similar queries are executed over the same remote dataset.

6. **Improved service performance** is a result obtained by dynamically deploying analysis services closer to the data or by pro-active multiple deployment of computationally expensive services over a set of hosts in order to distribute the number of invocations between all the instances.

7. **Dynamic Service-Oriented Distributed Query Processing framework** is the final contribution where the original OGSA-DQP system was extended in terms of architecture, design and implementation to incorporate the features from DynaSOAr to enable dynamic deployment. The extended system was evaluated to establish the claims of improved performance while processing distributed queries.

7.2 Further Work

Referring back to Arthur D. Little's statement about research, the stumbling blocks of research into OGSA-DQP, DynaSOAr and the dynamic version of OGSA-DQP, resulted into building stones for future research and development. In this section, the opportunities of further work that were identified during the course of the research are presented. There is scope for making considerable improvements to the existing work which are listed along with the new ideas that can be explored.

7.2.1 Efficient Data Movement Between DQP Services

The OGSA-DQP system was the pioneer in distributed query processing based on service-orientation which adopted technologies from parallel databases and supported queries over a heterogeneous set of databases using existing services such as OGSA-DAI. There is, however, once concern related to the communication between the component services of OGSA-DQP. The data packets or tuples are sent as SOAP messages, and are serialised and de-serialised using Apache Axis [77] libraries, which is not the most efficient way of communication. It was observed by Alpdemir et. al. in [144] that the cost of communication between DQP components contributes largely to the overall execution of a query, and within this communication cost, the serialisation/de-serialisation is the most expensive operation. Further, the concerns expressed in [125], [126] and [127] and the very reason why the need for dynamism was felt, are present in OGSA-DQP itself, and affects the performance. Adopting the emerging standards and techniques regarding binary data communication over SOAP, such as MTOM [147], would provide significant improvement over the current Axis-based transfer used within OGSA-DQP.

7.2.2 Support for Non-relational Data Formats

The existing OGSA-DQP caters to relational database systems that are wrapped as OGSA-DAI services. OGSA-DAI however supports other data formats such as CSV, XML etc. This is a void within OGSA-DQP which needs to be filled so that the ultimate goal of a middleware that is able to seamlessly integrate data from various data sources irrespective of the format and platform, can be achieved.

7.2.3 Effective Brokering in DynaSOAr

The current version of DynaSOAr incorporates a very simple brokering approach. But the complete vision of a *Software Hypermarket* is the one where on one hand the consumers will be able to submit their requirements, such as cost, reliability, provider preference etc., using which one or more broker entities would be able to provide the consumer with the optimal service, and on the other hand, the service providers should be able to choose between available host providers based on parameters such as cost and reliability. To achieve this, the characteristics of each service and host, in terms of the cost of using it, the reliability must be considered while making scheduling or routing decisions regarding hosting a service on a certain resource or forwarding the consumer request to a certain host. The consumers also should be allowed to specify their preferences while

submitting the requests, possibly using predefined *Service Level Agreements (SLAs)*. Some research is going on in this respect in the CRISP project [148] and GRIA [149] where an identifier for the SLA is added to the header of the SOAP message sent by the consumer. Policies can be defined at the service provider end and/or the host provider end, against which the SLA of the consumer can be validated in order to provide the desired quality of service. The GridSHED project [113] looked into heuristical algorithms for resource allocation based on usage characteristics, which can also contribute towards the development of the vision of a *Software Hypermarket.*

7.2.4 Robust and Efficient Transport for DynaSOAr Deployment

DynaSOAr normally uses Java-based network IO libraries for transferring files required for deploying a service over the network. This performance of this system declines rapidly as the size of the files being transported increases. The system is largely ineffective for the movement of large virtual machine images which are normally few gigabytes in size. To deal with this, the SFTP utility has been used for transferring the virtual machine images, which performs in a considerably robust and efficient way when compared to the previous system. But, the system still requires a more efficient method of transporting large files. GridFTP [55] is one possible option which has been used in DEBUT [150], but this would create a tight coupling of DynaSOAr with the Globus system [43]. Alternative methods such as SRB [54], which provides an efficient way of storing and transferring large binary files may be explored. Another attractive option is the peer-to-peer systems, which may be particularly suitable for situations where the same software image must be downloaded to multiple hosts.

7.2.5 Remodelling the Query Compiler

All the points mentioned in the previous sections are likely to cause a positive effect to the dynamic OGSA-DQP system by improving the performance. The research carried out in the thesis is not about rewriting the query compiler or the cost model that is used within the compiler, but is to exploit the possibilities of using dynamic service provisioning inside OGSA-DQP. The claims made in the thesis are established by the experimental results, which may form the basis for re-engineering the query compiler and the cost model in the lights of dynamic service provisioning. The approach taken in this thesis was to extend the existing compiler with dynamic service provisioning features, but, it may lead to a better and more robust system if the query compiler is remodelled considering the ideas developed during this thesis.

One such requirement is a network-aware cost model. Dynamic service provisioning must consider the connectivity of the available resources. The original cost model used in the DQP query compiler had no notion of considering the network bandwidth or latency between two nodes while optimising the query plan. In the extended version presented in this thesis, a simple method of estimating the network latency is used and some intelligence is added to the compiler for deciding on the "closeness" between nodes based on the network latency. There are existing work on network-aware cost models such as the ones proposed in [133] and [134]. Adopting such an algorithm or re-engineering the current cost model to consider network latency for DQP is likely to provide a huge boost towards the realisation of a Grid middleware system that is able to process queries distributed over remote heterogeneous databases.

A casual look at the prospects may seem to suggest that Arthur D. Little's comment about research may be too far-fetched in this case. The convergence of DynaSOAr and OGSA-DQP may have resulted in some performance benefits, but will it really contribute to the middleware that was envisaged during the start of the e-Science programme? These prototypes should really be considered as small steps towards the greater goal. Already considerable developments into various related areas such as Utility Computing has resulted into remarkable systems such as 3Tera [151] which provides intuitive interfaces to provision complex networked applications. Virtualization technologies are developing and have proved to be remarkably efficient within Grid application domains. OGSA-DQP transformed into using the more popular *SQL* from the earlier *OQL*. It may not be unreasonable to imagine sophisticated human-computer interfaces allowing scientists the freedom of using their own languages and procedures to build the queries or workflows in an intuitive way. A simple *submit* button will enable the DQP system to perform an optimisation of the query based on the available resources and *create* for the scientist the most optimal *virtual laboratory* for executing the experiments, storing the results in an electronic form of *labbook* and sharing it with research colleagues pursuing similar interests. Imagination is the first step in research, the stumbling blocks of which may result into building stones paving the path for future research.

> *"Imagination is the beginning of creation. You imagine what you desire, you will what you imagine and at last you create what you will." - George Bernard Shaw*

Bibliography

[1] eScience @ EBI. http://www.ebi.ac.uk/escience/.

[2] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[3] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – A Distributed Job Scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[4] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11 (2):115–128, Summer 1997.

[5] SOAP Version 1.2 Part 1: Messaging Framework. http://www.w3.org/TR/soap12-part1/.

[6] myGrid. http://www.mygrid.org.uk/.

[7] Paul Watson and Pete Lee. The NU-Grid Persistent Object Computation Server. In *1st European Grid Workshop, Poznan, Poland*, 2000.

[8] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[9] Basic Local Alignment Search Tool. http://www.ncbi.nlm.nih.gov/BLAST.

[10] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R. Pocock, Peter Li, and Tom Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34:729–732, 2006.

[11] What is OGSA-DQP? http://www.ogsadai.org.uk/about/ogsa-dqp/.

[12] Jim Smith, Sandra Sampaio, Paul Watson, and Norman W. Paton. The Design, Implementation and Evaluation of an ODMG Compliant, Parallel Object Database Server. *Distributed and Parallel Databases*, 16(3):275–319, 2004.

[13] Sandra de F. Mendes Sampaio, Norman W. Paton, Paul Watson, and Jim Smith. A Parallel Algebra for Object Databases. In *Workshop on Parallel and Distributed Databases, in conjunction with DEXA'99*, Florence, Italy, August 1999.

[14] OGSA-DAI. `http://www.ogsadai.org.uk/`.

[15] VMWare - Virtualization Overview. `http://www.vmware.com/virtualization/`.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-757-5.

[17] Katarzyna Keahey, Ian T. Foster, Timothy Freeman, Xuehai Zhang, and Daniel Galron. Virtual Workspaces in the Grid. In *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference*, volume 3648 of *Lecture Notes in Computer Science*, pages 421–431. Springer, 2005.

[18] Paul Ruth, Junghwan Rhee, Dongyan Xu, Rick Kennell, and Sebastien Goasguen. Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure. In *Autonomic Computing, 2006. ICAC 2006. IEEE International Conference on*, pages 5–14, 2006.

[19] Paul Watson, Chris Fowler, Charles Kubicek, Arijit Mukherjee, John Colquhoun, Mark Hewitt, and Savas Parastatidis. Dynamically deploying Web services on a grid using Dynasoar. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC*. IEEE, 2006.

[20] WS-I - Web Services Interoperability Organization. `http://www.ws-i.org/`.

[21] WS-I Basic Profile Version 1.0. `http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html`.

[22] Jim Gray. Distributed computing economics. `http://research.microsoft.com/research/pubs/view.aspx?tr_id=655`.

[23] Alexander S. Szalay, Peter Z. Kunszt, Ani Thakar, Jim Gray, Don Slutz, and Robert J. Brunner. Designing and mining multi-terabyte astronomy archives: the Sloan Digital Sky Survey. *SIGMOD Rec.*, 29(2):451–462, 2000.

[24] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 102–111, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-365-5.

[25] Goetz Graefe. Iterators, Schedulers, and Distributed Memory Parallelism. *Software practice and Experience*, 26(4), April 1996.

[26] Goetz Graefe, Richard L. Cole, Diane L. Davison, William McKenna, and Richard H. Wolniewicz. Extensible Query Optimization and Parallel Execution in Volcano. In Johann Christiph Freytag, David Maier, and Gottfried Vossen, editors, *Query Processing for Advanced Database Applications*, pages 336–335. Morgan Kaufmann, San Francisco, CA, 1994.

[27] Savas Parastatidis, Jim Webber, Paul Watson, and Thomas Rischbeck. WS-GAF: a framework for building Grid applications using Web Services: Research Articles. *Concurrency and Computation : Practice and Experience*, 17(2-4):391–417, 2005.

[28] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. Open Grid Services Infrastructure (OGSI) Version 1.0., 2003. Open Grid Services Infrastructure(OGSI)Version 1. 0.

[29] Malcolm Atkinson, David DeRoure, Alistair Dunlop, Geoffrey Fox, Peter Henderson, Tony Hey, Norman Paton, Steven Newhouse, Savas Parastatidis, Anne Trefethen, Paul Watson, and Jim Webber. Web Service Grids: an evolutionary approach: Research Articles. *Concurrency and Computation: Practice & Experience*, 17(2-4):377–389, 2005.

[30] Grid Computing Information Centre. `http://www.gridcomputing.com/gridfaq.html`.

[31] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* San Francisco, Morgan Kaufmann Publishers Inc, 1999. ISBN 1558604758.

[32] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001.

[33] Reagan W. Moore, Chaitanya Baru, Richard Marciano, Arcot Rajasekar, and Michael Wan. Data-intensive computing. In Ian Foster and Carl Kesselman, editors, *The grid: blueprint for a new computing infrastructure*, pages 105–129. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-475-8.

[34] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.

[35] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the Grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[36] GT Execution Management: GRAM. `http://www.globus.org/toolkit/gram/`.

[37] Nathalie Furmento, Anthony Mayer, Stephen McGough, Steven Newhouse, A. J. Field, and John Darlington. ICENI: Optimisation of component applications within a Grid environment. *Parallel Computing*, 28(12):1753–1772, December 2002.

[38] Don Box. Code Name Indigo - A Guide to Developing and Running Connected Systems with Indigo. *MSDN Magazine - The Microsoft Journal for Developers*, 19(1), January 2004.

[39] OASIS Web Services Business Process Execution Language (WSBPEL) TC. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.

[40] OASIS - Organization for the Advancement of Structured Information Standards. `http://www.oasis-open.org`.

[41] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, 2002.

[42] Karl Czajkowski, Don Ferguson, Ian Foster, Jeffrey Frey, Steven Graham, Tim Maguire, David Snelling, and Steve Tuecke. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution. 2004.

[43] The WS-Resource Framework. `http://www.globus.org/wsrf/`.

[44] Karl Czajkowski, Don Ferguson, Ian Foster, Jeffrey Frey, Steven Graham, Igor Sedukhin, David Snelling, Steve Tuecke, and William Vambenepe. The WS-Resource Framework. 2004.

[45] Ian Foster, Jeffrey Frey, Steve Graham, Steve Tuecke, Karl Czajkowski, Don Ferguson, Frank Leymann, Martin Nally, Igor Sedukhin, David Snelling, Tony Storey, William Vambenepe, and Sanjiva Weerawarana. Modeling Stateful Resources with Web Services v. 1.1. 2004.

[46] W3C Web Services Addressing (WS-Addressing). `http://www.w3.org/Submission/ws-addressing/`.

[47] Kevin Cline, Josh Cohen, Doug Davis, Donald F Ferguson, Heather Kreger, Raymond McCollum, Bryan Murray, Ian Robinson, Jeffrey Schlimmer, John Shewchuk, Vijay Tewari, and William Vambenepe. Toward converging web service standards for resources, events, and management. version 1.0. Technical report, A joint white paper from Hewlett Packard Corporation, IBM Corporation, Intel Corporation, and Microsoft Corporation, 2006.

[48] Web Services Eventing (WS-Eventing). http://www.w3.org/Submission/WS-Eventing/.

[49] Web Services Metadata Exchange (WS-MetadataExchange). http://schemas.xmlsoap.org/ws/2004/09/mex/.

[50] Web Services Resource Transfer (WS-ResourceTransfer). http://schemas.xmlsoap.org/ws/2006/08/resourcetransfer/.

[51] Michael Stonebraker. *Readings in Database Systems (2nd ed.)*. San Mateo, Morgan Kaufmann Publishers Inc, 1994.

[52] Amazon Simple Storage Service (Amazon S3). http://www.amazon.com/gp/browse.html?node=16427261.

[53] Paul Watson. Databases and the Grid. *Computing Science Technical Report Series*, (CS-TR-755).

[54] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The SDSC storage resource broker. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1998.

[55] Bill Allcock, Joe Bester, John Bresnahan, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, 2002.

[56] Martin Schaller. Reclustering of high energy physics data. In *Statistical and Scientific Database Management*, pages 194–203, 1999.

[57] Mark Hayes. Grids: A Reality Check. www.escience.cam.ac.uk/mark/GEFD-2.ppt.

[58] Database Access and Integration Services WG (DAIS-WG). http://forge.gridforum.org/projects/dais-wg.

[59] Open Grid Forum. http://www.ogf.org/.

[60] Mario Antonioletti, Malcolm Atkinson, Rob Baxter, Andrew Borley, Neil P. Chue Hong, Brian Collins, Neil Hardman, Alastair C. Hume, Alan Knox, Mike Jackson, Amy Krause, Simon Laws, James Magowan, Norman W. Paton, Dave Pearson, Tom Sugden, Paul Watson, and Martin Westhead. The design and implementation of Grid database services in OGSA-DAI: Research Articles. *Concurrency and Computation: Practice & Experience*, 17(2-4):357–376, 2005.

[61] S. Malaika, C. J. Nelin, R. Qu, B. Reinwald, and D. C. Wolfson. DB2 and Web services. *IBM Systems Journal*, 41(4):666–685, 2002.

[62] Jim Smith, Anastasios Gounaris, Paul Watson, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. Distributed Query Processing on the Grid. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, pages 279–290. Springer-Verlag, 2002. ISBN 3-540-00133-6.

[63] Paul Watson. The Design of an ODMG Compatible Parallel Object Database Server. In *International Meeting on Vector and Parallel Processing (VECPAR), LNCS 1573*. Springer, June 1998.

[64] Ian Foster and Nicholas T. Karonis. A grid-enabled MPI: message passing in heterogeneous distributed computing systems. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–11, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X.

[65] Tanu Malik, Alexander S. Szalay, Tamas Budavari, and Ani Thakar. SkyQuery: A Web Service Approach to Federate Databases. In *CIDR*, 2003.

[66] Goetz Graefe and Diane L. Davison. Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Execution. *IEEE Transactions of Software Engineering*, 19(8):749–780, August 1993.

[67] Sivaramakrishnan Narayanan, Umit Catalyurek, Tahsin Kurc, Xi Zhang, and Joel Saltz. Applying database support for large scale data driven science in distributed environments. In *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing*, page 141, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2026-X.

[68] Michael Beynon, Renato Ferreira, Tahsin M. Kurc, Alan Sussman, and Joel H. Saltz. DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems. In *IEEE Symposium on Mass Storage Systems*, pages 119–134, 2000.

[69] Thomas Friese, Matthew Smith, and Bernd Freisleben. Hot service deployment in an ad hoc grid environment. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 75–83, New York, NY, USA, 2004. ACM. ISBN 1-58113-871-7.

[70] Giovanni Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer, 1998. ISBN 3-540-64792-9.

[71] Li Chunlin and Li Layuan. An agent-oriented and service-oriented environment for deploying dynamic distributed systems. *Computer Standards & Interfaces*, 24(4):323–336, 2002.

[72] Daniel Wu, Divyakant Agrawal, and Amr El Abbadi. StratOSphere: mobile processing of distributed objects in Java. In *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 121–132, New York, NY, USA, 1998. ACM Press. ISBN 1-58113-035-X.

[73] Andrea Omicini and Franco Zambonelli. Coordination for Internet Application Development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999.

[74] Pu Liu and Michael J. Lewis. Mobile Code Enabled Web Services. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 167–174, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2409-5.

[75] Andrew Harrison and Ian J. Taylor. Dynamic Web Service Deployment Using WSPeer. In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 11–16. Louisiana State University, February 2005.

[76] Christos Chrysoulas, Evangelos Haleplidis, Robert Haas, Spyros Denazis, and Odysseas Koufopavlou. A Web-Services Based Architecture for Dynamic Service Deployment, 2005.

[77] Apache Axis Project. `http://ws.apache.org/axis/`.

[78] Markus Keidl, Stefan Seltzsam, and Alfons Kemper. Flexible and Reliable Web Service Execution. In *1st Workshop on Entwicklung von Anwendungen auf der Basis der XML Web-Service Technologie*, pages pages 17–30, 2002.

[79] Markus Keidl and Alfons Kemper. Towards context-aware adaptable web services. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 55–65, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-912-8.

[80] Markus Keidl, Stefan Seltzsam, Konrad Stocker, and Alfons Kemper. Serviceglobe: distributing e-services across the internet. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 1047–1050. VLDB Endowment, 2002.

[81] UDDI Version 3.0.2. `http://uddi.org/pubs/uddi_v3.htm#_Toc85907988`.

[82] Li Qi, Hai Jin, Ian T. Foster, and Jarek Gawor. HAND: Highly Available Dynamic Deployment Infrastructure for Globus Toolkit 4. In *15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2007)*, pages 155–162. IEEE Computer Society, 2007.

[83] Eun-Kyu Byun and Jin-Soo Kim. DynaGrid: A dynamic service deployment and resource migration framework for WSRF-compliant applications. *Parallel Computing*, 33(4-5):328–338, 2007.

[84] Jon B. Weissman, Seonho Kim, and Darin England. Supporting the dynamic grid service lifecycle. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 808–815, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-9074-1.

[85] Jon B. Weissman, Seonho Kim, and Darin England. A Framework for Dynamic Service Adaptation in the Grid: Next Generation Software Program Progress Report. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 10*, page 221.2, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2312-9.

[86] Matthew Smith, Thomas Friese, and Bernd Freisleben. Towards a Service-Oriented Ad Hoc Grid. In *ISPDC '04: Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04)*, pages 201–208, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2210-6.

[87] John Darlington, Jeremy Cohen, and William Lee. An Architecture for a Next-Generation Internet Based on Web Services and Utility Computing. In *WETICE '06: Proceedings of the 15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 169–174, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2623-3.

[88] Amazon Elastic Compute Cloud (Amazon EC2). `http://www.amazon.com/gp/browse.html?node=201590011`.

[89] A Market for Computational Services Project. `http://www.lesc.ic.ac.uk/markets`.

[90] Richard A. Meyer and Love H. Seawright. A Virtual Machine Time-Sharing System. *IBM Systems Journal*, 9(3):199–218, 1970.

[91] J. E. Smith and Ravi Nair. An overview of virtual machine architectures. In *Virtual Machines: Architectures, Implementations and Applications*. Morgan-Kaufmann, 2004.

[92] Jeff Dike. A user-mode port of the linux kernel. In *ALS'00: Proceedings of the 4th conference on 4th Annual Linux Showcase & Conference, Atlanta*, pages 7–7, Berkeley, CA, USA, 2000. USENIX Association.

[93] Renato J. Figueiredo, Peter A. Dinda, and José A. B. Fortes. A Case For Grid Computing On Virtual Machines. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 550, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1920-2.

[94] Ivan Krsul, Arijit Ganguly, Jian Zhang, Jose A. B. Fortes, and Renato J. Figueiredo. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 7, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2153-3.

[95] Ananth I. Sundararaj and Peter A. Dinda. Towards virtual networks for virtual machine grid computing. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.

[96] VMWare - VMWare Workstation. `http://www.vmware.com/products/ws/`.

[97] Oracle9i Advanced Replication, Release 2 (9.2). `http://www.lc.leidenuniv.nl/awcourse/oracle/server.920/a96567/toc.htm`.

[98] Chris Fowler. *Dynamic Deployment of Web Services on the Internet or Grid*. PhD thesis, Newcastle University, UK, 2007.

[99] M. Nedim Alpdemir, Arijit Mukherjee, Norman W. Paton, Paul Watson, Alvaro A. A. Fernandes, Anastasios Gounaris, and Jim Smith. Service-based Distributed Querying on the Grid. In *First International Conference on Service Oriented Computing (ICSOC 2003), LNCS 2910*, pages 467–482. Springer-Verlag, 2003.

[100] OGSA-DAI Glossary of Terms. `http://www.ogsadai.org.uk/documentation/ogsadai-wsrf-2.2/doc/reference/glossary.html`.

[101] W3C Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`.

[102] R. G. G. Cattell and Douglas K. Barry. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[103] GRIMOIRES. `http://www.grimoires.org`.

[104] W3C Extensible Markup Language (XML). `http://www.w3.org/XML/`.

[105] W3C XML Schema. `http://www.w3.org/XML/Schema`.

[106] Donald Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[107] Leonidas Fegaras and David Maier. Optimizing Object Queries Using an Effective Calculus. *ACM TODS*, 24(4):457–516, 2000.

[108] Leonidas Fegaras. Query Unnesting in Object-Oriented Databases. In *SIGMOD*, pages 49–60, 1998.

[109] W3C Web Services Architecture. `http://www.w3.org/TR/ws-arch/`.

[110] Laura Sebu and Horia Ciocarlie. The design of stateful web services based on web service resource framework implemented in globus toolkit 4. In *SYNASC '06: Proceedings of the Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 309–316, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2740-X.

[111] Sun Grid Engine. `http://www.sun.com/software/gridware/`.

[112] Paul Watson, Tom Jackson, Georgios Pitsilis, Frank Gibson, Jim Austin, Martyn Fletcher, Bojian Liang, and Phillip Lord. The CARMEN Neuroscience Server. In *UK eScience All Hands Meeting 2007*, 2007.

[113] Jenny Palmer and Isi Mitrani. Optimal Server Allocation in Reconfigurable Clusters with Multiple Job Types. In *Computational Science and its Applications (ICCSA 2004), Assisi, Italy*, 2004.

[114] Charles Kubicek, Mike Fisher, Paul McKee, and Rob Smith. Dynamic Allocation of Servers to Jobs in a Grid Hosting Environment. *BT Technology Journal*, 22:251–260, 2004.

[115] XMLSpy: XML editor for modeling, editing, transforming, and debugging XML technologies. `http://www.altova.com/products/xmlspy/xml_editor.html`.

[116] Resource Description Framework. `http://www.w3.org/RDF`.

[117] Hypermarket From Wikipedia. `http://en.wikipedia.org/wiki/Hypermarket`.

[118] VMWare. `www.vmware.com/pdf/virtualization.pdf`.

[119] Sriya Santhanam, Pradheep Elango, Andrea Arpaci-Dusseau, and Miron Livny. Deploying virtual machines as sandboxes for the grid. In *WORLDS'05: Proceedings of the 2nd conference on Real, Large Distributed Systems*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[120] VMWare - VMWare Server. `http://www.vmware.com/products/server/`.

[121] Microsoft Virtual PC. `http://www.microsoft.com/windows/virtualpc/default.mspx`.

[122] Susanta Nanda and Tzi-Cker Chiueh. A Survey on Virtualization Technologies. Technical Report ECSL-TR-179, Department of Computer Science, SUNY at Stony Brook, February 2005.

[123] The Gene Ontology Database. `http://www.geneontology.org/index.shtml`.

[124] SwissProt Protein Knowledgebase. `http://expasy.org/sprot/`.

[125] Madhusudhan Govindaraju, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, and Dennis Gannon. Requirements for and evaluation of RMI protocols for scientific computing. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 61, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7803-9802-5.

[126] Fabian E. Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Efficient wire formats for high performance computing. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7803-9802-5.

[127] Kenneth Chiu, Madhusudhan Govindaraju, and Randall Bramley. Investigating the Limits of SOAP Performance for Scientific Computing. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 246, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1686-6.

[128] Arijit Mukherjee and Paul Watson. Adding Dynamism To OGSA-DQP: Incorporating The DynaSOAr Framework In Distributed Query Processing. In *Euro-Par 2006 Workshops: Parallel Processing, EuroPar 2006*, volume LNCS 4375, pages 22–33. Sringer Verlag, 2006.

[129] Chris Wroe, Carole Goble, Mark Greenwood, Phillip Lord, Simon Miles, Juri Papay, Terry Payne, and Luc Moreau. Automating experiments using semantic data in a bioinformatics grid. *Intelligent Systems*, 19(1):48–55, 2004.

[130] EMBL Nucleotide Sequence Database. `http://www.ebi.ac.uk/embl/`.

[131] Simon J. Woodman, Douglas J. Palmer, Santosh Shrivastava, and Stuart Wheater. DECS: A System for Decentralised Coordination of Web Services. In V. Tosic, A. Van Moorsel, and R. Wong, editors, *Middleware for Web Services (MWS) 2005 Workshop, EDOC 2005*, pages 24–31, 2005.

[132] Anastasios Gounaris, Rizos Sakellariou, Norman W. Paton, and Alvaro A. A. Fernandes. A novel approach to resource scheduling for parallel query processing on computational grids. *Distributed and Parallel Databases*, 19(2-3):87–106, 2006.

[133] Chang-Hung Lee and Ming-Syan Chen. Distributed Query Processing in the Internet: Exploring Relation Replication and Network Characteristics. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 439, Washington, DC, USA, 2001. IEEE Computer Society.

[134] Yanif Ahmad and Uğur Çetintemel. Network-Aware Query Processing for Distributed Stream-Based Applications. In *VLDB*, pages 456–467, 2004.

[135] KyoungSoo Park and Vivek S. Pai. CoMon: a mostly-scalable monitoring system for PlanetLab. *SIGOPS Operating Systems Review*, 40(1):65–74, 2006.

[136] PlanetLab - An Open Platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org`.

[137] Anastasios Gounaris, Jim Smith, Norman W. Paton, Rizos Sakellariou, Alvaro A. A. Fernandes, and Paul Watson. Adapting to Changing Resource Performance in Grid Query Processing. In *Data Management in Grids, First VLDB Workshop, DMG 2005, Revised Selected Papers*, volume 3836 of *Lecture Notes in Computer Science*, pages 30–44. Springer, 2005.

[138] Anastasios Gounaris, Norman W. Paton, Rizos Sakellariou, Alvaro A. A. Fernandes, Jim Smith, and Paul Watson. Practical Adaptation to Changing Resources in Grid Query Processing. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 165, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2570-9.

[139] Jim Smith and Paul Watson. Fault-Tolerance in Distributed Query Processing. In *IDEAS '05: Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS'05)*, pages 329–338, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2404-4.

[140] Jim Smith and Paul Watson. Failure Recovery Alternatives In Grid Based Distributed Query Processing: A Case Study. In Domenico Talia, Angelos Bilas, and Marios D. Dikaiakos, editors, *Knowledge and Data Management in Grids*, pages 51–63. Springer US, 2007.

[141] I. L. Smith. Joint academic network (JANET). *Computer Networks and ISDN Systems*, 16 (1-2):101–105, 1988.

[142] Suman Banerjee, Timothy Griffin, and Marcelo Pias. The interdomain connectivity of planetlab nodes. In *Passive and Active Network Measurement, 5th International Workshop, PAM 2004*, volume 3015 of *Lecture Notes in Computer Science*, pages 73–82. Springer, 2004.

[143] Neil Spring, Larry Peterson, Andy Bavier, and Vivek Pai. Using planetlab for network research: myths, realities, and best practices. *ACM SIGOPS Operating Systems Review*, 40(1):17–24, 2006.

[144] M. Nedim Alpdemir, Anastasios Gounaris, Arijit Mukherjee, Desmond Fitzgerald, Norman W. Paton, Paul Watson, Rizos Sakellariou, Alvaro A. A. Fernandes, and Jim Smith. Experience on performance evaluation with OGSA-DQP. In *Fourth UK e-Science All Hands Meeting*, 2005.

[145] "Arthur D. Little." Quotations. Quotations Book, 2005. Answers.com 20 Nov. 2007. http://www.answers.com/topic/arthur-d-little.

[146] Arijit Mukherjee and Paul Watson. Virtual Machines in DynaSOAr: Creating an on-demand ad-hoc Virtual Grid. Technical report, School of Computing Science, Newcastle University, CS-TR 1002.

[147] SOAP Message Transmission Optimization Mechanism. http://www.w3.org/TR/soap12-mtom/.

[148] CRISP: Commercial R3 IEC Service Provision. http://crisp-project.org/index.html.

[149] GRIA - Service Oriented Collaborations for Industry and Commerce. http://www.gria.org/.

[150] Sachin Wasnik, Terence J. Harmer, Paul Donachy, Andrew Carson, Peter Wright, John Hawkins, Christina Cunningham, and Ronald H. Perrott. Self Managing Middleware for Dynamic Grids. In *Advances in Grid and Pervasive Computing, Second International Conference, GPC 2007*, volume 4459 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 2007.

[151] Utility Computing - 3Tera - Utility Computing for Web Applications. http://www.3tera.com/.