Dynamic Deployment of Web Services on the Internet or Grid

Thesis by

Christopher Piers Fowler

In partial fulfilment of the requirements

for the Degree of

Doctor of Philosophy

Newcastle University

Newcastle upon Tyne, UK

(submitted)

14th May 2007

(defended)

26th July 2007

"*Come to the edge,*" he said.
They said, "*We are afraid.*"
"*Come to the edge,*" he said.
They came.
He pushed them and they flew.

Guillaume Apollinaire (1880-1918)

Acknowledgements

No journey this long is without encounters that change one's perspective, lighten the load, or provide the opportunity to learn new lessons in life. For me, my PhD years have been a very personal journey, far more than a transition from student to researcher, but an experience both profoundly rewarding and uniquely humbling – in short, a real *catharsis*. I have no regrets, despite the often painful reality. These years have changed my life and I have learnt far more than I could ever have hoped for: about myself, about life, and about what is truly important. Although sometimes disbelieving that I made it, I am immensely proud of what I have achieved. However, I could not have done it alone. The people I have met along the way have captivated, inspired and engaged me. It is these people who have truly opened my mind, and it is they that deserve the most credit.

First, to my supervisor, Professor Paul Watson, who saw something in me and woke it up. From the very start, his inspiring approach, enthusiasm, and lively discussion, drew the best from me, helping to fashion my thoughts into something approaching structured. His belief guided me during some very difficult times. It is hard to express adequately my thanks and appreciation for all he has done, except to say a very sincere: *thank you.*

I acknowledge with thanks the contribution of Arijit Mukherjee and Charles Kubicek, for essential implementation work in realising the ideas presented in this thesis. Also to Panos Periorellis and Jim Smith.

To my friend Sam Aaron goes my most profound respect. I would not be writing this if he had not been there; through highs and lows he has kept me going. Through his friendship he showed me a world I had never known and I have learnt so much from him. His trust, openness, and clarity of thought have challenged both me and my work. I hope for many more years of friendship. To Paul Robinson – an all round *good egg* – who, always eager to advise and offer help, has been great fun to work with. Whether it be hours in the saddle, or in the pub, he has made me laugh, which is a gift I hope we will continue to share. To Susanna Graham who has brightened my way. Simon Gamester; Benedict and Jenna Singleton; my flatmates over the years; cs-squash; cs-pint; and my trusty bike; all of whom kept me out of mischief and provided much needed relief from to much thinking - to you all goes a very warm *thank you.*

I would not be the person I am without the love of my parents. They are the ground from which I have grown. Their constant encouragement and tireless support, not to mention the use of their garden shed in which to sleep and write this thesis, is something that will stay with me. My brother Simon, my sister-in-law Jane, and their three beautiful children, Nick, Ben and Jenny, have all shown me love, brought me joy, and made me calm. These past few years have brought us all closer together, which is the true prize of my PhD years. To my brother Julian, out of sight and out of contact, but never out of my heart.

Lastly to Miss Wang Lifeng for her love and her simplicity; she challenged me to challenge myself and then set me free to fly. Without her this thesis would never have become.

Abstract

This thesis focuses on the area of dynamic Web Service deployment for grid and Internet applications. It presents a new Dynamic Service Oriented Architecture (DynaSOAr) that enables the deployment of Web Services at run-time in response to consumer requests.

The service-oriented approach to grid and Internet computing is centred on two parties: the service provider and the service consumer. This thesis investigates the introduction of mobility into this service-oriented approach allowing for better use of resources and improved quality of service. To this end, it examines the role of the service provider and makes the case for a clear separation of its concerns into two distinct roles: that of a Web Service Provider, whose responsibility is to receive and direct consumer requests and supply service implementations, and a Host Provider, whose role is to deploy services and process consumers' requests on available resources. This separation of concerns breaks the implicit bond between a published Web Service endpoint (network address) and the resource upon which the service is deployed. It also allows the architecture to respond dynamically to changes in service demand and the quality of service requirements. Clearly defined interfaces for each role are presented, which form the infrastructure of DynaSOAr. The approach taken is wholly based on Web Services.

The dynamic deployment of service code between separate roles, potentially running in different administrative domains, raises a number of security issues which are addressed. A DynaSOAr service invocation involves three parties: the requesting Consumer, a Web Service Provider and a Host Provider; this tripartite relationship requires a security model that allows the concerns of each party to be enforced for a given invocation. This thesis, therefore, presents a Tripartite Security Model and an architecture that allows the representation, propagation and enforcement of three separate sets of constraints.

A prototype implementation of DynaSOAr is used to evaluate the claims made, and the results show that a significant benefit in terms of round-trip execution time for data-intensive applications is achieved. Additional benefits in terms of parallel deployments to satisfy multiple concurrent requests are also shown.

Contents

List of Figures

Chapter 1

Introduction

A scientist should not need an '*e*' to use the grid. The grid should appear simple, so scientists can be scientists, thinking, working, and exploring in their own language. With current grid systems and infrastructures, scientists have to work at a very low level, perhaps via a command line interface, juggling jobs, services, containers, instances, handles, references or factories - to name a few. However, a grid user is fundamentally a biologist, an astronomer, or a geographer, not necessarily having any expertise in grid computing, but needing access to computing resources beyond that provided within their own computing infrastructure. If e-science is to become more readily a part of the *tools of science*, then there is a need for higher-level abstractions, that simplify interaction and development.

This is an admirable aim, which may only come about as a consequence of developing grid systems able to satisfy the requirements of the target user, while at the same time adopting higher-level implementation technologies that provide a simpler application view. The recent move towards Web Services in the grid community is a step towards this higher level view.

The motivation for the ideas presented in this thesis lies in two areas: the recent adoption of *service-orientation* (more specifically Web Services), as the architectural platform for grid and Internet applications; and the proliferation of raw data available to scientists intent on grid computing as the means to process these data.

Service-orientation has received wide-spread publicity in the computing media, largely as a result of the Web Services implementation of the service-oriented architecture (SOA). Consequently, the concepts and functionalities are widely understood and accepted. By separating the execution of a system's functionality from the interface, and relying on a common language of message-passing between interfaces to invoke services, the service-oriented architecture has enabled *loosely coupled* systems. These systems remain independent of the underlying implementation and service-execution location. Systems are now viewed as collections of potentially distributed, independent, loosely coupled *services*, rather than previously, where a system was seen as a monolithic singular entity, tightly coupled by the underlying object-oriented architecture.

To date, grid computing infrastructures based on job scheduling systems (for example Condor [1] or Globus [2]), have formed the majority of grid application frameworks. These systems use the *job* as the unit of deployment. A job - a combination of code and in many cases the data on which it is to operate - is created by the client and passed to the scheduling system to be routed to a suitably configured host that has the resources to execute the job [3]. After execution the job is discarded.

Grid systems that result from adopting this model may have certain performance issues when considering the needs of typical data-intensive grid applications, where the choice of both computational

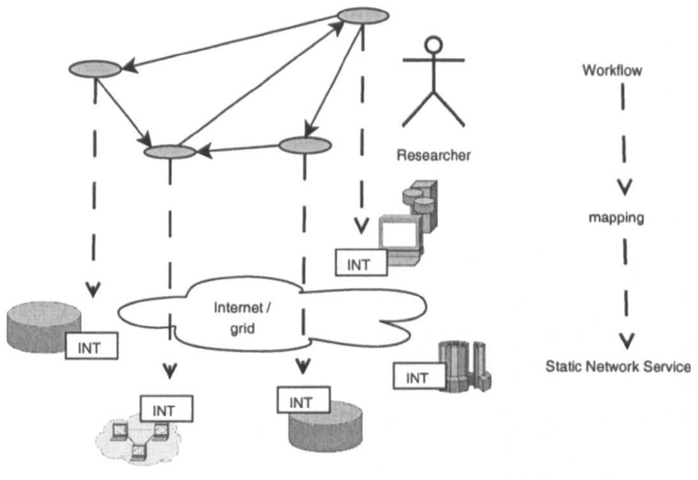


Figure 1.1: Example of static workflow mapping.

resource and data locality begin to bear on the quality of service of a particular job execution [4]. As the data-sets being analyzed increase in size from Giga-bytes to Tera-bytes, the need to locate the processing resources close to the data-source is important - to minimize the movement of data over the network. A job-scheduling system might then consider deploying a particular job based on the location of the data it uses when executing - it generally being the case that moving a small amount of code rather than a larger amount of data is more cost effective. These issues were highlighted by Watson and Lee when proposing an Active Information Repository specifically designed to address the problem of moving potentially large data-sets between data sources and processing resources across limited bandwidth networks [5].

Additional limitations relating to job scheduling systems include the need to repeatedly transport and install the code for every job, including repeat calls to the same job - recall that jobs are discarded after execution is complete. There may also be a need to shut down and reconfigure a host machine in order to process a particular type of job requiring a different operating system, or to install additional supporting libraries.

In grid computing, Web Services are being adopted to provide access to both computing and data resources, allowing researchers to build and orchestrate large, complex *workflows* that represent the required application. Examples of this can be found within the OGSA-DAI project for exposing databases as Web Services [6] and in the Taverna workflow language and enactment engine [7]. Workflows are mapped onto sets of services made available by resource providers - be they compute resources, such as network clusters, or data sources, such as large databases or instrumentation.

A typical Web Service based grid application is shown in figure 1.1, where a researcher encodes an experiment as a workflow and maps this onto suitable network services (published service interfaces). The example might show an *in silico* protein data analysis experiment involving the querying of a number of remote databases, the transfer of data to a network cluster service to perform bespoke analysis, and the result being fed to a data-storage service.

The example given in figure 1.1 highlights some of the advantages of workflow: a Web Service application can be distributed across many providers; the use of Web Services means that the system is loosely coupled; recovery from service failure can be rectified within the workflow enactment system by the user choosing to *bind* to another service that offers the same interface.

A limitation to the Web Service approach, howver, is highlighted by an implicit assumption: that a Web Service execution is bound to the resource hosting the published Web Service, i.e. as far as the Web Service consumer is concerned, the execution of the Web Service takes place at the network address (endpoint) of the published Web Service. This implicit binding means applications, or workflows, are being mapped onto static Web Service topologies and, by implication, static resources.

Systems of greatest flexibility, where service and hosting environment have been completely separated, are those adopting *code mobility* as the method of deployment [8]. These systems still fall under the umbrella of service-orientation, but provide more flexibility in terms of code *migration*. Code migrates between servers to respond to run-time conditions, or consumer instruction, with benefits such as: load-balancing; matching of code requirements to resource capabilities at run-time; fault tolerance and recovery; and the execution of code close to the data for improved performance. For example, in *mobile agent* systems, any agent can run on any agent-server. By making a distinction between the hosting resource (the hardware) and the agent (the software) these fully dynamic systems are able to deploy code on-demand and leverage the benefits of flexibility, resource utility, and improved quality of service, without the need to reconfigure the server for each new agent arrival.

The movement of arbitrary code, in the form of mobile agents, has proved problematic in terms of security when considered in a grid context, and these systems have not been widely exploited in grid computing - although some work is referenced in the background (chapter 2) of this thesis. The mobility-of-code as a wider concept, however, is relevant to the work presented in this thesis. Mobility of code is defined here as the ability to move, or *dynamically deploy*, a piece of code onto an available resource. (This differs from the notion of *migration*, which refers to the movement of running code between nodes on a network [9]).

Job scheduling systems use code mobility by deploying jobs to resources able to support them. Jobs are scheduled to run on resources when they become available. Such systems make a distinction between the code requirements, and the resource upon which the code is executed. Job scheduling based systems and mechanisms exist that match jobs to a supporting resource. Examples of these include:

- Condor's *classAds* [10] which matches resource capabilities to the requirements of a job before scheduling via the Condor system;
- Globus Resource Allocation Manager (GRAM) [11] which enables the deployment and management of jobs to pre-configured resources capable of supporting those jobs;
- ICENI [12], which matches requests for services to already deployed interfaces at run-time to execute requests in such a way as to balance the loading of resources, thereby optimizing resource utility.

A key distinction between the job scheduling of these systems and Web Service invocation, is that a job is a *deploy-once-run-once* execution model, whereas a Web Service is a *deploy-once-invoke-many-times* execution model. This point is key when considering the cost of deployment. In job scheduling this cost is attached to every job execution, and in turn every consumer must pay this cost; in Web Services, the cost of deployment is distributed between all consumers for the service.

In the discussion of Web Services and the service-oriented architecture, the anatomy of Web Service provisioning has been highlighted. A Web Service is presented to a consumer as a single endpoint address, which hides the fact that a Web Service is a combination of service implementation and service hosting (execution). (This is to be expected given the weight placed on loose coupling and the transparent execution described in the Service-Oriented Architecture Reference Model specification [13] - of which Web Services is an instance). This thesis argues that, as a result of the implied binding of service to resource, Web Services do not make the most efficient and flexible use of the resources available in a highly dynamic grid or Internet context. A more efficient use of resources would be to make an explicit separation between the Web Service provision and the resource provision, and exploit code mobility to allow the *dynamic deployment of services* between two providers: Host Providers that specialise in providing hardware resources, and Web Service Providers that specialise in providing Web Service implementation. The ability to dynamically deploy Web Services at run-time means a *late binding* is achieved between service and resource. Late binding is a major contributor to improved performance as service requests can be processed by the best available resources *at the time of the request.* This is in direct contrast to a static execution topology, where the processing of a message is bound to the resource hosting the service at the time the system was deployed. This separation of concern also allows explicit consumer-to-host specification, e.g. a consumer may explicitly specify which Host Provider is required to process the service request. This is seen as a key benefit, allowing data-locality, security, performance or other consumer quality-of-service requirements to be used in dynamic deployment decisions.

Therefore, given the benefits of flexibility and performance of the mobile code systems discussed and the perceived benefits to service-deployment, the first objective of the research presented in this thesis was:

> *To determine if the introduction of mobility into the service-oriented architecture can allow the dynamic deployment of Web Services, leading to a more efficient use of available resources and better performance for Web Service clients.*

This research was evaluated using an experimental prototype by measuring the response time for dynamically deployed Web Services when compared with static deployments. Various scenarios were investigated to evaluate the different aspects and benefits of such a dynamic deployment system.

This thesis proposes an architecture to support dynamic Web Service deployment. A dynamic service deployment system intended for use on a grid or Internet has security concerns that must be addressed. The focus here is on authorisation based on *constraints* (a limitation being placed on a deployment/invocation). These constraints will potentially span multiple domains. Currently, a Web Service interaction is between a calling consumer and a Web Service. If we now consider a separate host provider, then a Web Service interaction in DynaSOAr involves three entities: a consumer, a Web Service Provider and a Host Provider. Each of these entities will have security constraints which need to be enforced. Rather than the current bipartite interaction, the dynamic deployment system proposed is now tripartite. Therefore, the second objective of the research presented in this thesis was:

> *To investigate, design, and evaluate a tripartite security model able to satisfy the security constraints of all three parties in a dynamic Web Service deployment system.*

This aspect of the research was evaluated by modelling and prototyping a system to show correct enforcement.

1.1 Contribution

The contribution of this thesis lies in the design, specification and evaluation of a Dynamic Service-Oriented Architecture (DynaSOAr) that makes a clear and explicit separation between a Web Service Provider, responsible for providing services, and a Host Provider, responsible for providing resources upon which services are deployed and executed. This new architecture is designed to allow the following:

- dynamic deployment of Web Services at run-time between a Web Service Provider and a Host Provider. The Dynamic Service Oriented Architecture invocation model, shown in figure 1.2, breaks the bond between the Service Provider and the resources hosting a service, and introduces a new tripartite relationship between a Consumer, Web Service Provider and a separate Host Provider (this interaction is shown in grey).

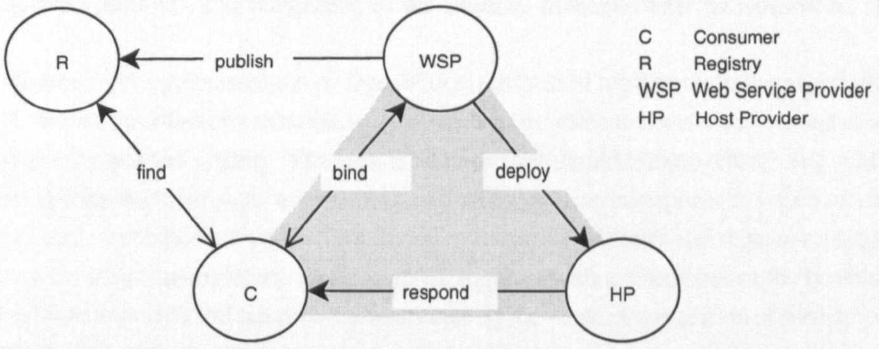


Figure 1.2: Dynamic service-oriented architecture.

- better quality of service for consumer and Web Service Provider in terms of performance, by deploying a Web Service to the best available resources, or close to the data upon which it operates, thereby reducing network latency when processing large data-sets.

- a wholly Web Service-based development abstraction that is simpler than the current mix of jobs and services.

- the opportunity for *brokers* to establish a marketplace for service and host provisioning.

In addition, the design and specification of a tripartite security model and infrastructure is presented that enables:

- the propagation, unification and enforcement of security constraints across all three parties in a DynaSOAr invocation.

Both the dynamic service-oriented architecture and tripartite security model have been evaluated using an experimental prototype.

1.2 Thesis Structure

In chapter 2 a literature survey and discussion is presented. The survey draws from the areas discussed above to show the relevant points from each domain, before focusing on related work in the area of dynamic Web Service deployment. Although, in the early stages of this research no related work in the area of dynamic Web Service deployment could be found, in recent years, as the grid has moved towards the Web Services model, a number of related projects have published work. These are discussed in detail with reference to the research presented in this thesis.

In chapter 3 a Dynamic Service-Oriented Architecture (DynaSOAr) is presented. The architecture makes a clear separation between Web Service provision, in terms of implementation code, and host provision, in terms of the hardware upon which Web Services are deployed for execution. Two deployment scenarios are presented for which the architecture must cater. A detailed description of the surrounding issues and the requirements of the architecture is given. A discussion of the benefits and possible uses of this novel system is also presented.

In chapter 4 tripartite security for DynaSOAr is discussed. The constraints of the three parties involved in an interaction are analyzed and modelled to show how these constraints can be enforced within the system. It is further shown how the eXtensible Access Control Markup Language (XACML) may be used in part to implement an architecture for a Security Manager component. This component is designed and specified. The integration of the Security Manager with the DynaSOAr system is also shown.

In chapter 5 the implementation of DynaSOAr is discussed together with the experimental platform used. A series of evaluation scenarios are presented to examine the performance, and discuss the benefits, of the DynaSOAr system. The results show a considerable improvement in performance when considering data-intensive applications, and parallel deployment in response to issues of scalability. The tripartite security component was also evaluated with example policies and request messages.

In chapter 6 a summary of the thesis is given, together with a discussion of the benefits of adopting the DynaSOAr approach and the contributions made by the work. A suggestion of further work identified during the course of the research is also presented.

Chapter 2

Background and Related Work

This chapter considers the background of this thesis and looks especially at those aspects that led to the specification of a Dynamic Service-Oriented Architecture for Web Service deployment. It looks at mobility, service-orientation, and grid computing, as well as the intersections between these areas. This is illustrated in figure 2.1 below.

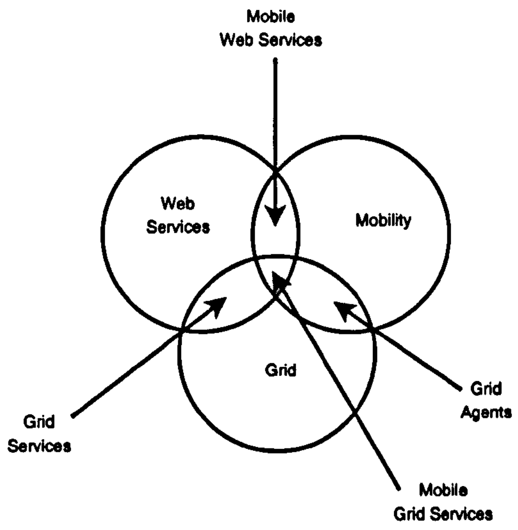


Figure 2.1: Background domains of interest.

Grid and Internet scale distributed computing offers unique challenges; the processing of large scale data-sets is one such challenge. In the early stages of the research presented by this thesis, investigation into mobile agents was gaining some ground as a way of moving computation around the network - rather than moving the data - as a possible solution to processing large data-sets. The need was for a system that could adapt to changes in the environment, communicate between heterogeneous software and hardware systems, and provide a higher-level programming abstraction to develop, and manage, complex distributed systems in an interoperable way.

Despite some effort, agents were not adopted by the grid community. Resource management systems were based on the *job* abstraction, so grid scale architectures that could support distributed job scheduling and resource management were being proposed. At the same time, the e-business community were investing heavily in Web Services and service-orientation to tackle Internet-scale distributed business activity. It would seem inevitable that the grid community would look towards service-orientation, and in particular Web Services, to find answers to the challenges of grid computing.

Section 2.1 presents a sort glossary of terms used for this thesis; section 2.2 discusses code mobility and defines more closely the notion of dynamic deployment; section 2.3 examines related work in the area of dynamic deployment; section 2.4 discusses grid and service-oriented computing, again examining related work; section 2.5 considers security in the context of this thesis and the research presented; section 2.6 presents a brief discussion that position this thesis and explains the key differentials.

2.1 Glossary of Terms

Grid	A virtual network composed of resources hosted on the Internet, potentially from different resource providers.
Resource	Hardware used to host services.
Grid Service	A service composed of an interface and implementation that has a number of mandated functionalities compliant with the Open Grid Services Architecture [14].
Web Service	A network addressable interface and software implementation that supports a set of operations defined by the interface compliant with the Web Services Architecture [15].
Grid Computing	The use of grid resources to deploy and run large-scale distributed applications, usually requiring resources beyond the scope of the user's current resources.
Service-Oriented Computing	The use of services to compose loosely coupled applications based on the Service-Oriented Architecture.
eScience	Large-scale and often collaborative use of grid/service-oriented computing to support data- or computationally-intensive, scientific applications.
Agent	A software module that exists in a particular context to serve a defined functionality.
Mobile Agent	An agent with the ability to migrate through a network of agent servers that host a run-time system that supports mobile agents.

2.2 Code Mobility

Work that examines some key motivations in the mobile agent community can be found in [16, 17, 18]. This early work concentrated on abstracting a methodology for developing interoperable systems of communicating agents that address issues such as: scalability, adaptability, and complex communication semantics. Its aim was to provide an alternative solution to overcome the challenges of Internet-scale distributed systems by providing a dynamic agent-based communicating middleware.

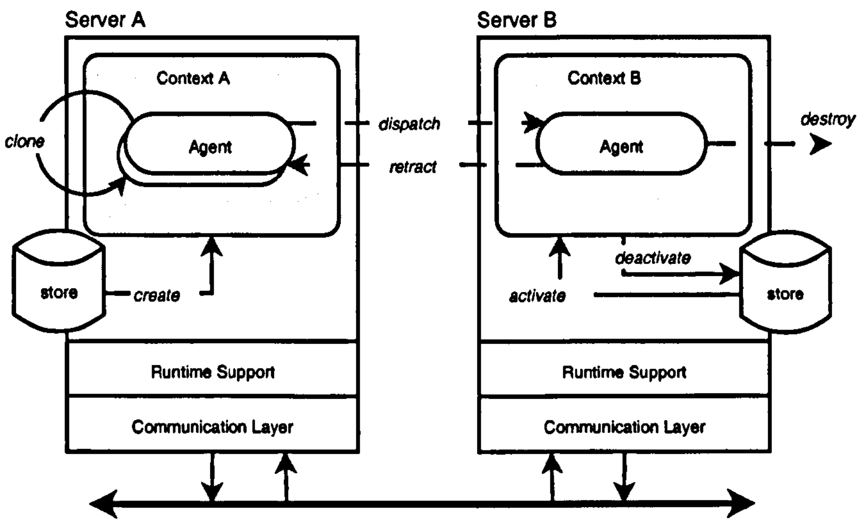


Figure 2.2: Typical Agent-System Architecture and Lifecycle Operations.

A typical mobile-agent-system architecture is illustrated in figure 2.2. This shows two *agent servers* (A and B). An *agent* exists within a *context*, through which agents are able to communicate with the *runtime support* system, and between different agents in different contexts, via the *communication layer*. The runtime mechanism supports a number of *lifecycle operations* which give the agent and context the ability to control various states of existence for each agent.

Mobility of code is not restricted to mobile agents, or the movement of byte-code (as in Java mobile agents). There are many notions to consider. Firstly, there is a distinction to be made between *strong* mobility and *weak* mobility [9]. Strong mobility refers to the migration of execution state, where an execution is stopped on one machine, transferred to another machine with the associated execution stack - containing the running state of the code - and then restarted. Weak mobility refers to code shipping, or fetching, and does not involve the movement of a running execution state [9]. (This thesis is concerned only with the latter). Second, there is a distinction to be made between the meaning of mobility in different programming patterns [8]. *Code on demand* for example, refers to a user requesting code to be downloaded for execution on a local resource; an example of this is Java Applets. *Remote evaluation* refers to a user providing the code and sending it to be executed on a remote resource; job scheduling is an example of this. Mobile agents are an instance of remote evaluation characterised by strong mobility. An in-depth guide, and discussion on the meaning and issues of mobility, can be found in [8, 19].

This thesis adopts the term *dynamic deployment* to mean the movement and installation of service implementation at run-time, without the need to restart the hosting resource. As such, it is an instance of remote evaluation. Another term adopted in the literature, which has the same semantics, is *hot deployment.*

At the beginning of the research presented in this thesis, an extensive review of current work revealed *no research* that exploited the possibilities of dynamic deployment within a Web-Services-based, service-oriented architecture, that permitted dynamic deployment of Web Services at run-time. This could possibly be credited to the e-business domain in which Web Services were conceived. If one considers an e-business interaction, where businesses interact via Web Services hosted from within an organisational boundary, the need to move implementation outside this boundary does not necessarily arise - possibly owing to reasons of security, or business confidentiality - so the fact that mobility in Web Services was not being investigated appears reasonable.

At the same time grid computing was focused on job scheduling as the underlying execution model, and not on service-orientation - although Web Services were beginning to be used in grid computing. As described earlier, although job scheduling is a form of *code mobility* (remote evaluation), the concern of this thesis is in Web Service mobility in the form of dynamic Web Service deployment.

2.3 Dynamic Deployment

There is a body of research that examines the dynamic deployment of Web Services within the Service Globe system [20, 21, 22]. This system was developed for load balancing between hosts inside a LAN, but the work has some similarities in approach with that presented here, despite being concerned only with a closed system.

Service Globe is focused on three aspects: dynamic service selection, load balancing, and service replication. Dynamic service selection is based on Universal Description, Discovery and Integration (UDDI) [23]. Each service being offered is assigned to a tModel, which provides a semantic classification based on the service functionality and a formal description of its interface. Within the dynamic service selection system it is possible to *call* a tModel rather than an explicit service endpoint. The system will then select a service based on a number of criteria; the first of these is *constraints*. The constraints are intended to provide a means of matching the request with the most appropriate service, or set of deployed services, using meta-data or other associated attributes. Load balancing is achieved by monitoring the load carried by each deployment, or associated systems, and then routing requests using a load balancing algorithm. An interesting aspect is the ability to *hot deploy* a requested service if the current loading means an unacceptable delay would result from using current deployments. As described in [20], the dynamic deployment within Service Globe is achieved using *mobile code*. A requested service is deployed into the run-time system of the selected hosting node via a run-time-service loading-service. If a service is not already cached then the system uses a UDDI repository to find the required service code. Hosts are also discovered via a UDDI repository. Services in Service Globe are classified as *internal* or *external*. External services are third-party services running outside the system, which are accessed via *adapters*, which interpret messages between Service Globe and the external provider. Internal services can be static (bound to an execution host), or dynamic (deployable), and are implemented using the Service Globe APIs. Dynamic services can be hot deployed on demand.

No specific separation between service-provision and resource-provision is detailed in the Service Globe system, though it is implicit in the deployment capabilities. The approach taken by DynaSOAr makes a clear separation between service and resource provision, with interfaces defined for each provider. Unlike Service Globe, DynaSOAr makes no distinction between external or internal services; all services in DynaSOAr are Web Services, which simplifies the development and deployment abstraction. Another difference is that the Service Globe system makes no provision for explicit consumer-to-host specification, which in the context of DynaSOAr is a key benefit, allowing data-locality, or other consumer quality-of-service requirements to be used in dynamic deployment decisions.

Service deployment using mobile agents is an approach to distibuted systems which has attracted some research activity in the area of grids. Chunlin and Layuan [24] present JASE, a Java-based Agent-oriented and Service-oriented Environment for deploying dynamic distributed systems. The system uses the service abstraction and the remote programming model offered by mobile agents. Service Interface Agents wrap local service implementations as agents. These are the only agents that have access to the underlying resource of the hosting server. Service Interface Agents register their interface with a JASE Server, which allows for searching. The server resides as a running process on all nodes in the network which offer resources. Mobile agents are used to migrate and interact with registered Service Interface Agents. The JASE server is responsible for co-ordinating communication, mobility, service registry and security. Communication is via proprietary asynchronous messages for agents communicating between remote servers, or via a closed and tightly coupled tuple space mechanism for local agents - this is a proposed mechanism in [25].

The architecture of the JASE system is a common one amongst distributed service-oriented systems; it is characterised by: *static services*, with a run-time binding of the request to a suitable service. The use of mobile agents as the transient link between Service Interface Agents, allows for coordinated working, however, the Service Interface Agents are static. The mobile agents in the system are programmed with a set task, which may involve interaction with a number of Service Interface Agents located at different sites. The execution of the work is *not* mobile.

This work does not exploit the movement of Service Interface Agents. Incorporating mobility into these working agents would allow the work-load to be moved, perhaps to optimise the agents' performance in fulfilling the required task; although this is not a stated aim of the work. This is in contrast to DynaSOAr that is designed to allow the movement of Web Service implementation specifically to optimise performance of the consumer's application.

Many studies have been carried out by the agent community to investigate the issues commonly associated with grid-computing. Yang and Rana et al investigated the use of mobile agents to enable *dynamic configuration of distributed parallel computing resources* to support the processing of remote sensor data archives [26]. They propose an architecture of interconnected agents which are wrappers for the key aspects of the architecture: presentation, interaction, management, resource control, data-retrieval, security and transport. This is a common approach and is similar to the JASE system [24] mentioned above, the difference being that Yang and Rana use mobile agents to transfer (dynamically deploy) specific algorithms between data sources. As agents arrive they communicate with the local site to gain authorized access and are then able to carry out the necessary processing on the local resource. The effect of adopting mobile agents rather than static agents to carry out the work, means that the processing is moved to the data. This is in common with the DynaSOAr approach.

Further evaluation work by Cao, Kerbyson and Nudd [27], put forward an agent-based hierarchy to model and investigate the effectiveness of agents in dealing with the *scalability* and *adaptability* issues associated with grid computing environments. The hierarchy proposed by Cao et al uses *broker*, *coordinator* and *agent* abstractions, which map effectively on to typical grid-/service-oriented infrastructures similar to DynaSOAr. Although no separate host is modelled, the agent abstraction is analogous to a service execution running on a host. The coordinator is responsible for passing request messages to deployed agents, with brokers adopted to aid searching for agent capabilities. This work is discussed to show how the architectural abstractions adopted by DynaSOAr map onto dynamic systems (such as those adopting agents), and how they respond to the demands of scalability and adaptability of grid environments. The model represented only demonstrates the invocation stage of DynaSOAr, not the deployment, as no host provider is modelled, and therefore no dynamic deployment is modelled. Dynamism in Coa's work is achieved through brokers that route messages to deployed agents.

Grids are heterogeneous by nature, with disparate resources connected via public networks. Making use of these highly dynamic environments requires system architectures that can respond to changes in both performance characteristics of resource and the physical attributes of the resources in question. The modelling scenarios shown require service discovery, resource matching, request routing and include different performance metrics typical of heterogeneous environments.

DynaSOAr deals with issues raised by Coa et al in the following way: scalability is dealt with by dynamically deploying a new service to satisfy an increased demand (shown in chapter 5, section 5.2.3);

dependability is handled by routing messages to a different, or new, host provider if one fails or is unavailable; adaptability is handled by deploying a new service to a different host provider as the prevailing conditions of the grid (hosting) environment change.

These works, together with many others [28, 29, 30, 31, 32] highlight some of the key benefits of mobile-agent technology and its possible use for grid and service-based infrastructures: modularity, scalability, adaptability, reduced latency, low bandwidth usage, code mobility and communication.

The grid offers unique challenges in terms of resource sharing, processing requirements, load balancing, and the movement of data. As data sets grow, the need for new infrastructures to process these data becomes apparent. Moving data is no longer always the best, or most cost effective, means of bringing data and processing together. Agents have played a part in understanding and evaluating these challenges. Architectures targeting these specific requirements have been proposed [28, 29, 33] based on agents. A motivating example for this thesis was the Active Information Repository (AIR) proposed by Watson and Lee [5], where the use of mobile agents to dynamically deploy computations on to hosting resources was proposed. The underlying principle of the Active Information Repository described, is: *move the processing to the data.* The architecture of an AIR links a scalable object-oriented database with a parallel agent execution server. Clients interact with the architecture by sending user defined queries in the form of mobile agents. The agents arrive and are distributed in such a way as to balance the load over all nodes of the execution server. Both the server and database exploit parallelism to provide scalability [5].

DynaSOAr differs from an Active Information Repository by supporting both service provision and resource provision through two separate components; on an Active Information Repository the consumer is the service provider, with resources managed via the agent server. An Active Information Repository supports clients pushing the service (code in the form of mobile agents) onto a hosting server, with the focus on close integration between an object-oriented database (as the data source) and on high-performance hosting resources linked via high bandwidth networks. The focus of DynaSOAr, however, is on generic support for dynamic service provisioning that allows multiple system topologies to be designed and exploited. An Active Information Repository can be implemented using the DynaSOAr infrastructure (see work published in [34]). The separation of concerns in DynaSOAr means that co-location of the components in support of the required application gives greater flexibility in distributing a service-oriented application. For example, co-locating the service provider with the consumer, with the host provider located close to the data-source, provides an Active Information Repository. Scalability in DynaSOAr is supported in a similar fashion to the Active Information Repository architeture by allowing multiple services to be deployed in response to consumer requests.

Despite research investment in the use of mobile agents for data-intensive applications, both LAN-based and on the grid, the grid community has moved in recent years towards service-orientation in seeking a way forward.

2.4 Grid and Service-Oriented Computing

Grid computing emerged in the mid-1990s in response to the ever increasing amount of data for processing [35]. The need to share resources to meet the challenges of processing these data meant a new infrastructure, built on top of the Internet, which enabled scientists to collaborate across organisational boundaries, share resources, and interrogate large distributed data-sets. The proposals at the time were for an infrastructure designed to run on top of resource management systems, based on job scheduling, such as Condor [1] and Globus [2].

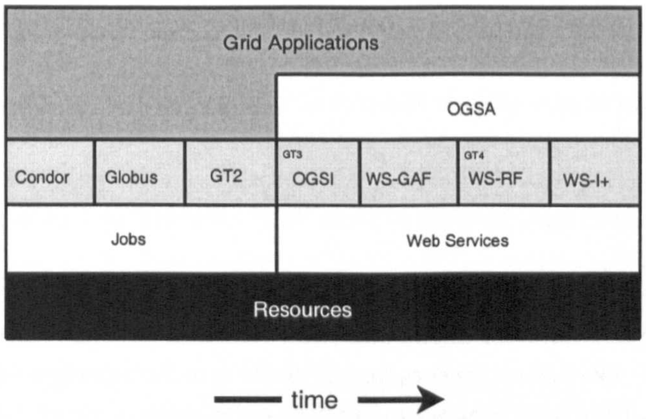


Figure 2.3: Evolution of the Grid technology.

Much work was done, and indeed continues, on job-scheduling-based grids, however, a consensus has emerged in the grid community that led to the adoption of service-orientation as the underlying architectural principle for grid based systems. The needs of the community required an architecture for distributed systems integration across multiple domains to capitalise on the collective resources of virtual organisations. The Open Grid Services Architecture (OGSA) [14] was proposed to address this challenge.

Figure 2.3 shows an evolution of the key technologies in the grid community, both job based, and those adopting the Web Service approach to implementing the OGSA. The time-line encompasses early job based systems, such as Condor [1] and Globus [36], through to the adoption of service-orientation and approaches based on an augmented set of Web Service standards (WS-I$^+$ [37]), promoted by the Web Services Interoperability Organisation [38]

The authors of the OGSA produced the Open Grid Service Infrastructure [39], as an implementation of the architecture, that built on both existing Globus [36] and Web Services technology. The infrastructure defines mechanisms for management, discovery, lifetime and notification in support of Grid Services [40]. A *Grid Service* is defined as:

> *A Web Service that provides a set of well-defined interfaces and that follows specific conventions. The interfaces address discovery, dynamic service creation, lifetime management, notification, and manageability; the conventions address naming and upgradeability.* [40]

The naming convention of the OGSI uses a *grid service handle* (GSH) as a globally unique service identifier. A grid service handle contains enough information to identify a grid service, but not to interact with it. To interact with a grid service, the GSH must be resolved into a *grid service reference* (GSR). The grid service reference holds binding information that enables a grid service consumer to bind to a particular *grid service instance*. Binding may be via a number of different binding mechanisms (IIOP, SOAP, SMTP). A grid service instance is created following a consumer request for a particular grid service, using a grid service factory . The factory returns the grid service handle. The consumer resolves the grid service handle to obtain the grid service reference and binding information. The consumer then binds to a *specific* grid service instance. The grid service reference is, in effect, a *pointer* to a grid service instance. This pointer may be passed by reference within the client application [39]. The lifetime, management, control and notification mechanisms are associated with a grid service reference.

The approach proposed by the OGSI seems complicated. It is fundamentally an object-oriented approach that promotes tight-coupling between a service consumer and a grid service instance, accessed via a pointer. The OGSI was designed in response to the perceived requirements of grid applications, as discussed in the Open Grid Services Architecture [14]. The conventions, naming strategies and approach of the OGSI (the implementation of OGSA) were later challenged by an approach that promoted the exclusive use of *service-oriented* principles and conventions associated with the Web Services Architecture [15]. The Web Service Grid Application Framework (WS-GAF) [41] demonstrated how the same grid requirements implemented in the OGSI could be achieved using standard Web Services and commodity tooling.

WS-GAF challenges the notion of the grid service handle and grid service reference, as these are seen to expose the underlying resource upon which a grid service executes. This breaks a central tenet of the service-oriented architecture, that of execution transparency. Instead, WS-GAF proposes the use of *loosely coupled* Web Services that do not name or expose the underlying resource, or pass pointers to resource. In WS-GAF, services and resource are separate entities. Web Services are invoked by sending messages to a published network address (Web Service endpoint). There is no binding between the consumer and a particular instance of a Web Service. Any instance-level binding is done transparently behind the network address, so as not to expose the underlying resource. The notion of mapping service-requests to resources is left to the service implementers.

The emergence of the WS-GAF resulted in a refactoring [42] of the OGSI into the Web Services Resource Framework (WS-RF) [43]. WS-RF is designed to allow the modelling of stateful resources [44], by codify the relationship between a Web Service and the *stateful resource* upon which it operates [44]. A WS-Resource is a composition of a Web Service and a stateful resource. A stateful resource is described by a stateful resource description. A Web Service may expose one-to-many portTypes (put simply, a portType is a description of a Web Service function; a collection of portTypes therefore constitute the functionality of a Web Service). In WS-RF, a portType may map to only *one* stateful resource description. A stateful resource description may map to many portTypes. This allows a single description to be used by many portTypes.

A WS-Resource is referenced (and ultimately invoked) via an *endpoint reference*, encoded using the WS-Addressing [45] standard. A consumer sends an invocation message that specifies a particular portType. The portType invokes a *stateful resource factory* to create a stateful resource instance. The factory uses the stateful resource description associated with the portType to create the correct resource instance. On creation, the stateful resource instance returns an endpoint reference. The endpoint reference is later used to invoke the WS-Resource (this in turn, via the endpoint reference id passed in the request message, invokes the stateful resource instance). A stateful resource instance may be associated with many Web Services, allowing one instance to process many messages. The interaction of a consumer with a WS-Resource is illustrated in figure 2.4. This too would appear complicated when compared with the loose coupled nature of the pure Web Service approach. The grid service handle and reference has been replaced by the endpoint reference; the interaction semantics stay the same.

Web Services Resource Transfer (WS-RT) specification was recently published [46] to satisfy the core requirement of the WS-Resource Framework. (WS-RT is not reviewed here). This suggests a further evolution to the stateful-resource approach to grids.

The two approaches discussed - resource and service - mark a fork in the development of grid technologies: one specific to resource instances and the associated resource bindings; and one aimed at loosely coupled systems of Web Services. Given the uncertainty in the route taken by the stateful resource approach, the grid community is now more widely adopting the WS-I approach endorsed by [37] and first proposed by Parastatidis et al in [41].

So, how does this relate to DynaSOAr? The DynaSOAr approach promotes two interactions: one exe-

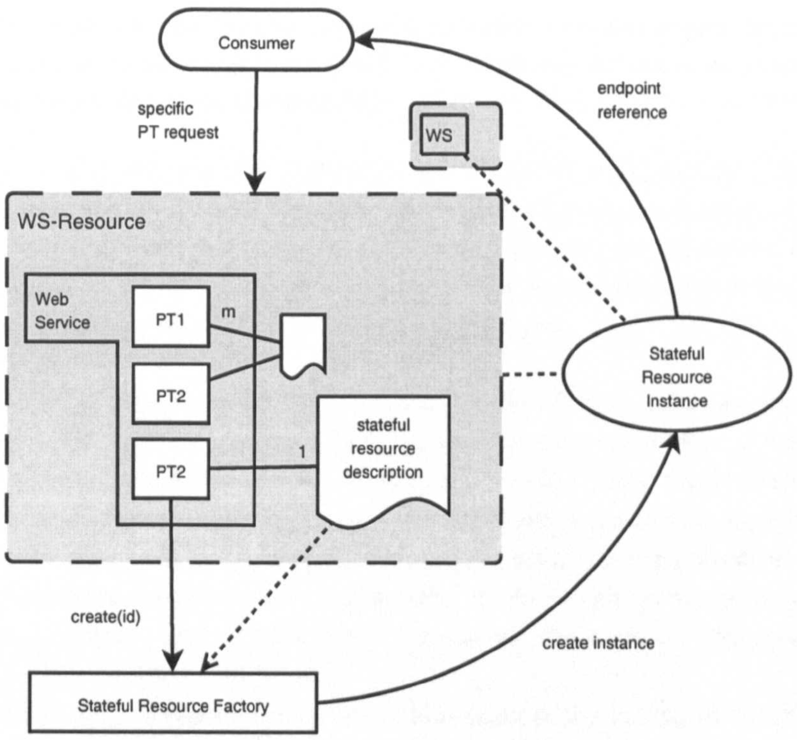


Figure 2.4: WS-Resource interaction.

cution transparent, where a consumer calls a standard Web Service with no notion of the execution of the service, this is analogous with the WS-I approach illustrated by WS-GAF [41]; and another, semi-transparent approach, where an execution is partially visible. In the latter interaction, a consumer may stipulate a specific host provider upon which a service may be executed (this is discussed further in section 3.3.1). The latter approach is analogous to the WS-RF in that a consumer is aware of the resource (host) upon which a service is executed. The difference is that no service instance is created and *no* binding mandated; the consumer makes a stipulation of a host provider (perhaps for security, or quality-of-service reasons) and the host provider is free to deploy the service onto the resources under its control. DynaSOAr does not require factories or reference binding. (This is seen as a heavy weight solution that encourages tight coupling and implies complex interaction control mechanisms [41]). By keeping the service and host-provision separate, and by controlling deployment through agreed policy constraints, a lighter, simpler architecture is possible, without loss to the consumer, or possible system topologies. By adopting the flexibility of the loosely coupled approach, and the consumer-specified, semi-transparent approach of a host-provider specification, DynaSOAr is able to satisfy both sets of consumer/architectural needs. The DynaSOAr approach is centered on processing a consumer's request to the best available service deployment, matching the needs of the service with the capabilities of the host provider. If no deployment exists, a new deployment is made at run-time to satisfy the request.

This section now moves on to discuss other motivating factors and work related to this thesis.

Weissman et al [47] identify the same needs as those outlined in the introduction to this thesis. The idea of mobility, and hence dynamic deployment, is now being recognised as an important and vital step forward in enabling dynamic deployment, and hence, dynamic virtual organisations.

> The next generation of Grids is focusing on how to elevate the level of abstraction to better enable Grid application designers and end-users to solve real problems. It has been persuasively argued that next-generation Grid applications will be increasingly multidisciplinary,

collaborative, distributed, and most importantly, dynamic. The latter implies that static infrastructure will not be adequate since such applications may be assembled on-the-fly to exist only for a transient period of time. [47]

The views expressed by Weissman et al endorse the premise underlying DynaSOAr. Their research is focused on the Open Grid Services Infrastructure view of grids, whereas DynaSOAr is wholly Web Services based. However, Weissman et al in [47, 48] propose a Dynamic Service Architecture similar in approach to DynaSOAr. One of the fundamental issues raised, and one in support of this thesis, is that of separation of service provision and resource (hosting) provision.

The work presented defines two components: an Adaptive Grid Service (AGS) and an Adaptive Resource Provider (ARP). The Adaptive Grid Service consists of three components: a *front-end*, which handles client requests; a *deployer*, which makes decisions on which site(s) should handle the request; and a *back-end*, which consists of an *AGS Factory*. The AGS Factory contains the implementation code for the requested service. The factory serves each request by creating a new instance of the requested service (this is named the Adaptive Grid Service Instance - the use of a grid service instance is consistent with the OGSI architectural principles adopted by the system). The use of factories means that a new factory is required for each new service type.

To dynamically deploy a grid service, the back-end is deployed to a hosting service provided by the Adaptive Resource Provider (ARP). As such, the system responds to a consumer request by deploying the back-end, the adaptive factory then creates an Adaptive Grid Service Instance (AGSI), which returns a Grid Service Handle to the consumer. The handle *binds* the consumer and the deployed service instance. The system provides support for notification between service instance and consumer. The Adaptive Resource Provider supports a *query* interface that allows the deployer to query the resource on a number of deployment-related metrics: lease time, platform features, resource profiles.

An interesting aspect of the work is the use of incremental deployment, whereby, once the OGSI infrastructure has been deployed to an ARP, subsequent grid service deployment is initiated by sending only the grid service implementation code. This gives a factor of four saving in the cost of subsequent deployments to an ARP. The current implementation of DynaSOAr uses a .war file as the unit of deployment. Each deployment requires the AXIS SOAP processing engine to be deployed together with the service implementation. The deployment package is smaller in size (2.5MB compared to 10MB for a 150KB sized service), but the ability to separate the two, and make saving in deployment cost, is a factor to investigate in DynaSOAr.

Architecturally, Wiessman's work is tied to the OGSI implementation and the use of reference bindings to service instances. The requirements of dynamism in DynaSOAr mean a new loosely coupled approach was taken. The two works (DynaSOAr and [47]) share the aspiration of separation of service from resource. However, by maintaining the grid service instance binding of OGSI, Wiessman's work maintains the key architectural limitation challenged in [41]. Although Weissman calls for better security, this is not addressed in the work. DynaSOAr has included security from the start.

Smith et al[49] are working with the Globus Toolkit 3.0 (GT3). They identify four areas of concern in the formation of *ad hoc* grids: node discovery, node property assessment, service deployment and security. They describe an ad hoc grid as one formed on-the-fly, perhaps short lived, and not necessarily within the confines of a virtual organisation, i.e., with implied trust relationships.

Unlike DynaSOAr, Smith et al make no distinction between service provision and resource provision, which is the premise of this thesis, but instead introduce ***hot deployment*** by leveraging a custom class-loading mechanism that uses a *HotFactoryCallbackImpl*, which acquires a service context for the AXIS engine rather than the standard *FactoryCallbaclImpl* used in Globus Toolkit 3.0. The system uses custom class-loaders to enable *hot deployment* without having to restart the GT3 service container (which would be a requirement if any new service, or new version, is installed in an unaltered GT3 container). Once a service is created and deployed the grid service handle is returned to the consumer.

The key architectural difference between the work presented by Smith et al [49] and DynaSOAr is the clean separation of service and host provisioning and the specification of Web Service interfaces for both.

The Highly Available Dynamic Deployment Infrastructure (HAND) [50] is an addition to the Globus Toolkit version 4.0 [51] (GT4). Globus has evolved since the late 1990s to accept the benefits of service-orientation and Web Services. The HAND infrastructure shows an acceptance by Globus of the need for more dynamic deployment systems as opposed to the wholly static earlier version of Globus [50], and in this sense is in agreement with other work presented here, and that of Weissman and Smith [47, 49].

The focus in Qi et al [50] is the dynamic deployment of either *containers*, or *grid services*. To this end they define two deployment approaches: HAND-S, which refers to service-level deployment, and HAND-C, which refers to container-level deployment. HAND-C necessitates the re-initialisation and reconfiguration of the whole container. The container in the HAND-C system is a holder in which a set of one or many grid services may run. Deployment at the container level, therefore, can be seen as analogous to the deployment of a *virtual machine* hosting a number of services. (The deployment of virtual machines is an area currently being investigated with DynaSOAr; see Mukherjee and Watson [52]).

The motivation for HAND is that of a robust implementation to support dynamic deployment of grid services and containers within the Globus GT4 [51] infrastructure. The results presented in [50] are concerned with the costs of deployment (for both container and service-level deployment) and the success rate of the deployment requests. The cost of deployment, in terms of time t, is given by:

$$t^{total} = t^{transfer} + t^{pending} + t^{deploy} + t^{reload} \text{ [50]}$$

The total cost of deployment (t^{total}) is the sum cost of the following: transferring the service/container; a deployment pending period; the time to actually deploy into the running system; and the reload time prior to the service/container becoming active. A comparison is made between the HAND/GT4's custom system and a vanilla Tomcat plus AXIS system. The result of the comparison shows that the HAND system is less expensive in terms of deployment cost in all but one case. Although the cost of deployment is significant in any dynamic deployment system, when compared to the round-trip time of an invocation for data-intensive or computationally expensive applications, it may become less of a consideration. A clear benefit of the DynaSOAr approach is that DynaSOAr is able to exploit the *one-to-many* service interaction semantics of Web Services, and distribute the cost of deployment between potentially many consumers of a particular service deployment. Qi et al [50] do not address this point.

HAND is tied to a specific grid technology and only addressed the deployment of service or container. HAND does not address the more generic benefits of the DynaSOAr approach which allows complete separation of resource and service provisioning with brokers for each. The clean separation of the DynaSOAr architecture, into component Web Services, enables different co-location scenarios to be devised, thereby enabling configurable deployment topologies to best suit the requirements of the application. HAND does not address the security concerns of code deployment, or provide a constraints model and architecture for enforcing the constraints of the actor in a service invocation/deployment.

Other systems have been examined to acquire further insights into the relevant issues being addressed by different grid infrastructures. A useful starting reference for this review included: *A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing* by Krauter et al [53]. In this work a taxonomy was built citing *Resource Dissemination*, using a *push*, or *pull* model (see [53] Figure 10.).

This classification refers to a method of disseminating the *work* (processing) across a grid system. A number of infrastructures were examined that use this classification.

BOND [54] is a system using Java *distributed agents* [55]. It uses the knowledge query and manipulation language (KQML) for inter-agent communication. Agents are then given a task and communicate with other agents in the system to achieve their goal. It uses strong mobility [9] to checkpoint and migrate the processing between resources. This approach is similar to that taken by Yang and Rana [26] and discussed above.

NetSolve [56] and its evolution to GridSolve [57] examines client-agent-server programming models for grid applications (similar to [24] discussed above). These systems are based on mobile agents that migrate across the network searching for a NetSolve server best suited to perform the task in hand. The agent then returns to the client with the answer. This is a typical usage of mobile agents.

Additional searching uncovered a number of projects that, although not looking at service-orientation, introduced the author to some of the commonalities and wider issues in the field. These are not reviewed here, although the knowledge gained will undoubtedly have had an influence. Areas such as: adaptive scheduling; resource discovery based on run-time performance; process/agent telemetry for capturing historical performance; are among the areas studied.

Having reviewed the related work for dynamic deployment of Web Services, this background chapter now moves on to examine briefly the nature of the security dealt with by the work presented in chapter 4 of this thesis and draws on related work in the area.

2.5 Security

Security is a very large subject area with many challenges. In a highly dynamic environment such as the grid, these challenges become even more complex. The creation of virtual organisations [58], means that the physical administrative boundaries of an organisation are in some way being shifted, to become a wider, virtual organisational boundary. These new boundaries may only be in existence for a transient period, sufficient to meet the goals of the interaction. If we consider the need for businesses to collaborate with multiple partners, all with their own security concerns, then the control of multiple transient virtual organisations is needed.

> In many ways, arguably the most significant challenge for Grid computing is to develop a comprehensive set of mechanisms and policies for securing the Grid. Users [participants in the virtual organisations] need to know if they are interacting with the right piece of software, or human, and that their messages will not be modified, or stolen, as the message traverses the virtual organisation. Users will often require the ability to prevent others from reading data that they have stored in the virtual organisation. In short, users must trust the software infrastructure of the Grid to sufficiently prevent malicious activities (of course, it is reasonable to expect that the user must be proactive in this regard and that this does not come without some burden on the part of the individual user). [59]

Humphrey et al [59] clearly draw attention to the scale of the challenge in delivering a *trusted* grid environment. Further discussion of the implications for typical grid usage scenarios can be found in [60]. If we include in this discussion the now widely held belief that dynamic deployment must become a reality for grid systems (see [47, 50, 4]), then the challenges grow still further. This level of dynamism is hard to organise and administer. The risks to each business involved, and to the individual/collective intellectual property, are high.

This thesis proposes a new Dynamic Service Oriented Architecture to leverage the benefits of mobility in a wholly Web Service based system. It makes a separation of concerns that introduces a third party to the usual bipartite Web Service interaction (this is fully detailed and specified in chapter 3). The service interaction in DynaSOAr is *tripartite*. In the light of this tripartite nature, and the introduction of dynamic service deployment, a brief review of dynamic code/service-based security relationships is felt necessary, and is now presented.

At a *service* level, a dynamic system that enables the deployment of code (services) between a service provider and an execution host, consists of three parties: a service consumer (C), a service implementation (S), and an execution host (H). Examining the resulting relationships between these parties, helps to clarify the potential security issues which need to be addressed. Figure 2.5, illustrates the possible dynamic code scenarios that arise from this tripartite scenario; these are explained below with reference to some existing work in the area.

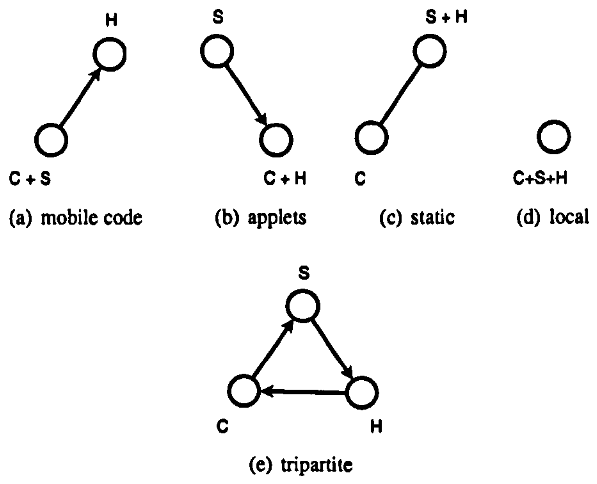


Figure 2.5: Security Relationships for Dynamic Code

(a) mobile code With user-enabled *migration* of mobile code to execute at a remote site we have **mobile agents**. This is a *push* model, instigated by the user, where code is pushed onto a host. There are clear issues of authorisation from the perspective of the host. For example, the host will want to be sure that the consumer pushing the code is authorised to do so. An identity for authentication will need to be presented to the host by the agent on behalf of the consumer, to allow authorisation. Much work has been done in the area of mobile agent security [61, 62, 63]. Example (a) in the figure, also illustrates job scheduling, if we consider the push of the service as remote evaluation. Job scheduling has the same issues of authentication and authorisation as mobile agents.

(b) applets The user provides the execution environment, *pulling* the code (service) to run on the local resources. This example is typified by Java **Applets**, or software patches. The acceptance of a patch is usually done via trusted certificates, which establish origin and authenticity. The choice as to whether to accept the download is made by the consumer (user) of the code. Applets rely on a well defined security models, such as *sandboxing* [64].

(c) static With static code and static execution, this example illustrates **Web Services**. There is a wealth of standards to cover many aspects of Web Service security (discussed in [65]); the eXtensible Access Control Markup Language (XACML) [66] can be used authorisation, and the Security Assertion Markup Language (SAML) [67] for authentication. These are just a few of the proposed standards.

(d) local For completeness, **localhost** execution is shown, although the security concerns are limited - unless a consumer does not trust himself!

(e) tripartite The separation of service from hosting resource, results in a **tripartite** relationship for security concerns. This is the focus of the work described in chapter 4 of this thesis, where a new tripartite security model for DynaSOAr is presented.

The DynaSOAr system is Web Services based, therefore, guided by the relationships detailed in figure 2.5, an examination of Web Service security is a first logical step to begin a review of current work. This is limited still further, as the only concern at this early stage is for authentication and authorisation.

2.5.1 Web Service Security

Web Service security is predicated on the needs of business/commercial interaction. The security concerns involved are commonly understood to span the following areas (described in terms of messages), and form the full *security stack*:

Integrity Integrity refers to the ability to protect a message from unauthorised changes.

Confidentiality Confidentiality refers to the ability to keep a message private from any party other than the intended recipient.

Authentication Authentication refers to the act of confirming a party's unique identity.

Authorisation Authorisation refers to the act of giving access to a resource. It is usual that access is granted to an authenticated identity.

Non-repudiation Non-repudiation refers to the inability of one party to deny an action.

Much work has been published in the form of standards, technical guides, Web Service specification documents, and the like. It would be unrealistic to review this literature. The security stack discussed above is supported by the Web Services security suite of standards, and is one of the more stable areas of the WS-* super-set of standards.

The key concepts, requirements and standards for Web Service security are discussed in [68]. The discussion of standards states that:

Standardisation enables agreement and interoperability at a particular technical level, beyond or upon which proprietary and differentiating ideas can be added. [68]

What Mysore suggests is that once the open standards are agreed upon, then securing disparate systems in an interoperable way is technically possible. As is suggested in section 6.2.1, it is not always easy, or given the appropriate support to encourage collaboration.

Securing web services using open standards is discussed by Hondo [65] and provides a very clear guide to the issues and possible solutions.

The security issues of DynaSOAr at this stage are focused on access-control constraints between the three actors in an invocation/deployment. A brief discussion below is presented as a rationale for this aim, with some reference to existing work.

2.5.2 Access Control

Authorisation, or access control, is concerned with granting access to a resource. Examples of access may be in the form of read, enter, invoke, or remove; an example of a resource may be a file, service, person, or a building. Collectively, access control in DynaSOAr may affect the topology of a deployment by restricting a deployment to a particular host, or it may concern the subject of the access control request (are they trusted, or credit worthy for example). In the context of DynaSOAr, access control is a concern at two stages: *service deployment*, and *service invocation*; and is expressed using control statements. These control statements are modelled as *constraints*. (These issues are discussed in much greater detail in chapter 4, however, to aid the discussion of related work it is important to introduce a brief overview of the objectives of the DynaSOAr approach - so that a more useful comparison can be made).

The constraints associated with *service deployment* are concerned with access to the resources of a host provider (to enable deployment of a Web Service to satisfy a consumer's request), whereas the constraints associated with *service invocation*, are concerned with access to a deployed service (again to satisfy a consumer's request). In DynaSOAr these two interactions potentially concern all three parties: consumer, service provider, host provider. This extends the usual two-party access control relationship of grantee and granter, to include a third party.

In a *traditional* access control situation a policy describes the access rules governing a particular resource. A request for access is made against this policy, and dependent upon the outcome of the rule evaluation, access is granted or denied. DynaSOAr requires a more dynamic approach.

The DynaSOAr architecture exploits late binding to enable the best possible deployment/invocation to be made at run-time. Static policy is created that governs access to service provision and host provision, and also describes any constraints made by the consumer. The policies that govern deployment and/or invocation must be generated dynamically as a deployment or invocation request is being handled. The dynamic policies are short-lived, existing only long enough for the access control decision to be made. Therefore, the architecture for DynaSOAr security, supports: propagation of policy, dynamic policy generation, decision evaluation and enforcement.

Of the dynamic deployment systems reviewed above (see sections 2.3 and 2.4), only one made *concrete* reference to the security concerns of dynamic deployment. This is not unusual - security is not *sexy*, so it is more often than not left for *others to tackle*. Smith et al [49], however, did introduce a security model to address some issues in dynamic service deployment. The key concern raised is that of inter-service attack. They assume no central security authority, so trust between separate consumers is not a given, i.e. if A trusts B and A trusts C, it does not hold that B trusts C. This quite rightly raises the issue of inter-service trust. If A and C have a service deployed and B subsequently deploys a service then trust becomes an issue for all concerned.

DynaSOAr assumes an existing trust relationship between all parties, so inter-service trust is not a concern addressed by the research here. In the DynaSOAr model, it can be envisaged that a policy stipulation by the consumer of a service states which other consumers are acceptable to share a common resource. The host provider making the deployment may then decide to partition the resources in such a way as to satisfy these constraints. This is not, however, a scalable solution to the problem of inter-service trust. If the consumer requires this level of separation (to instill trust), then a premium rate may be charged for the dedicated resources.

It is important for the future of dynamic deployment infrastructures that these issues are being raised, as trust may not be present in a truly open and dynamic grid. Trust in a decentralised environment is complex and not debated here. The mobile agent community have been investigating these issues [69, 61], though it is not clear whether solutions have been found.

More specifically on access control, the PRIMA system [70, 71] (and supporting work [72]) is designed to tackle three issues: static user accounts, coarse granularity, and application-dependent enforcement mechanisms. In considering the first of these, Lorch argues that the use of static user accounts, set-up on the potential resource providers a consumer may access, is not scalable. Accounts are often not used in a timely way, and may exist well beyond the time of a service invocation. Secondly, the coarse granularity referred to is caused by reliance on restriction mechanisms at the operating system level, which limit the expressiveness of any constraints. Finally, Lorch discusses the reliance on external enforcement outside the local operating system.

The PRIMA system tackles these issues by providing a grid-layer privilege management system to support fine grained access rights and delegation. It uses X.509 Attribute Certificates to carry privilege and policy statements. These are held against resources and used by consumers to propagate intentions. Access Control Lists are used to challenge resource requests. Policy creation, Privilege Management and Policy Decision mechanisms are included in the architecture. Dynamic account creation is also enabled, to ensure timely use and avoid the issue of lingering accounts. The PRIMA system is built around static policies. The policies are created by resource authorities (the person/system responsible for a resource) and stored.

DynaSOAr has two sets of *resources*: service and hosts (both these are viewed at the architectural level as Web Services; the term *resources* is used simply to aid understanding when considering the two systems). The dynamic creation and enforcement of deployment and invocation-policy requirements, discussed above, is not dealt with by the PRIMA system - no dynamic deployment is envisaged. DynaSOAr does not need the creation of user accounts to access a Web Service so this is not considered. The need for expressiveness in the policy language is dealt with by the adoption of the eXtensible Access Control Markup Language (XACML) [66]. This is in common with PRIMA. The X.509 Attribute Certificates used in PRIMA are exchanged for security assertions in DynaSOAr, encoded using the Security Assertion Markup Language (SAML) [67], which allows single sign-on with security assertion propagation, and delegation. DynaSOAr creates the required dynamic-policy at run-time, to constrain both deployment and invocation requests within the infrastructure in support of consumer requests. The dynamic creation of policy is done using the XACML policy combining mechanisms (discussed in detail in section 4.3).

Another policy language considered for the security requirements of DynaSOAr was Ponder [73]. Ponder is a rules-based policy language, management system and run-time support for distributed policies. In Ponder, policy is defined as a *rule governing the choices in behavior of a managed system* [74]. The system is claimed to provide the flexibility needed for dynamically modifying behavior of a system by changing the policies governing its behavior without the need for recoding the management system. This separation of control and management is the same as that provided by the XACML and is a requirement in the DynaSOAr system. Ponder has a much greater run-time support when compared with XACML, providing support for the distribution and instantiation of policies, as well as the disabling, unloading and deletion of policies. These are capabilities necessary - and are identified later in chapter 4 as requirements for a wider security system for DynaSOAr - but at this stage of *proof-of-concept*, the simpler language and run-time mechanisms provided by XACML are considered sufficient to meet the needs of the research presented by this thesis. In addition to this reason, Ponder assumes and object-oriented view of the underlying system [75], where system interaction occurs via remote object invocations; DynaSOAr is a service-oriented system and is concidered to be better supported using the XACML as it makes no such assumption regarding the underlying system. Also, XACML has close integration with the Security Assertion Markup Language (SAML).

The access control issues of expressiveness, and static account creation discussed by Lorch [72] are common in many grid access control systems. The delegation of rights and the propagation of credentials, assertions, and enforcement via access control lists (ACLs), are consistent across a large number of existing systems (see debates in [76, 60, 59, 77]). This debate has centered around virtual organisations. The dynamism of DynaSOAr enables virtual organisations to be formed for a single service deployment/invocation; or deployment topologies may be formed with a longer time-frame in mind. DynaSOAr uses the notion of constraints, expressed using fine-grained rules in XACML, to allow interaction participants from across different domains. The XACML rules are then grouped to form policies. Policies can be combined to cover all aspects of the tripartite, potentially distributed relationship. The system is very flexible in this respect.

An excellent summary of other access control work using XACML, but not addressing the issues of tripartite relationships and dynamic policy creation, is available in [78] and is well worth reading.

The need for dynamic security models and enforcement architectures is endorsed by Humphrey et al in [59] and Qi et al in [50], who both call for security systems able to tackle the issues of dynamic deployment in grids. The security debate is now moving towards dynamic systems, and DynaSOAr is the first system to address this directly in the Web Services arena.

DynaSOAr has introduced dynamic Web Service deployment. In line with the premise that security should be factored into the start of a new system, the research presented in this thesis also take a first step in addressing the security issues of dynamic deployment called for by Qi [50] and Humphrey [59]. This is debated further in chapter 4, section 4.1.

2.6 Summary Discussion

The work presented in this thesis adopts a wholly Web Services approach to grid computing infrastructures in line with [41, 37]. This maintains a loosely coupled system that encourages late binding with optimal discovery and matching of Web Service requirements to host capabilities. No reference pointers are used in DynaSOAr that expose the resources upon which services are execute, thereby maintaining execution transparency. This is a key architectural difference to the HAND [50], Dynamic Service Architecture [47, 48] systems reviewed. The DynaSOAr infrastructure does allow for semi-transparent deployment of Web Service. This is achieved by enabling a service consumer to stipulate a hosting service upon which a deployment/request should be executed. However, the choice of execution resource, upon which a Web Service executes, is under the full control of the hosting service. The consumer stipulates only a host provider service, not a specific resource. All providers in DynaSOAr are Web Services.

The following issues are common in all the systems reviewed, and seen as common to the wider distributed systems domain. The above related-work has been discussed in these terms. To summarize, each issue is now discussed to emphasize the approach taken by DynaSOAr:

scalability Scalability is the ability to scale in response to demand. Scalability is handled in DynaSOAr by enabling multiple deployments of a Web Service to be made to multiple, possibly distributed, host providers at run-time. If the request traffic increases, DynaSOAr is able to deploy new services in response (this is evaluated in section 5.2.3).

adaptability Adaptability is the ability to adapt to changes in the environment. Adaptability is handled in DynaSOAr by late binding and the loosely coupled nature of the architecture. This allows the system to respond to changes in resource availability and performance; if a host provider is performing poorly, then a new deployment can be made to avoid degradation in the consumer quality-of-service. Work in this area has been published regarding Web Service usage monitoring in DynaSOAr [79].

dependability A correctly functioning system is considered to have a high level of dependability. Dependability is handled in DynaSOAr at the architectural level. For example, if a host provider fails, or is not responding as expected, then a new deployment can be made to a different Host Provider. Failures are handled at each stage of a deployment/invocation in accordance with the usual Web Service failure protocols.

security Security is a complex issue. In the context of this work, security refers to the ability to constrain a deployment/invocation in a way that satisfies the constraints of all three actors in the system. To do this DynaSOAr enables dynamic policy creation and evaluation on a per-request basis at two levels: deployment and invocation.

This thesis now moves on to present the research undertaken to design, implement and evaluate the DynaSOAr system.

Chapter 3

Dynamic Web Service Deployment

This chapter presents a Dynamic Service-Oriented Architecture (DynaSOAr). DynaSOAr is a generic service-based infrastructure for the dynamic deployment of Web Services on a grid or the Internet. It uses an approach to grid computing in which there are no jobs, but only services [3]. Rather than scheduling jobs, DynaSOAr automatically deploys a Web Service on an available host in response to requests from consumers.

The chapter is organised as follows: section 3.1 discusses the convergence of ideas and opportunity, making the case for dynamic service deployment; section 3.2 discusses the opportunities for the DynaSOAr system; section 3.3 presents a detailed specification of the DynaSOAr services, components and interaction scenarios; section 3.4 concludes the chapter with a brief summary.

3.1 Towards Dynamic Web Services

Web Services is an instance of a service-oriented architecture. At the simplest level a Web Service is a *client - server* interaction, in which a consumer sends a request to a service provider which processes it and provides a response.

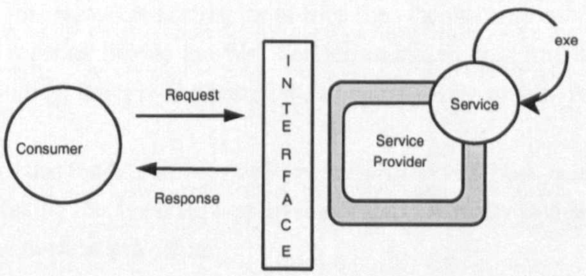


Figure 3.1: Anatomy of a Web Service invocation.

Figure 3.1 shows a high level view of a Web Service invocation. It involves the exchange of SOAP messages between a SOAP Client (Consumer in figure 3.1) and a SOAP server (Service Provider in figure 3.1) via an interface. To invoke the Web Service, the SOAP Client transforms an application level invocation message into the correct SOAP format, as prescribed by the interface. The SOAP request is then sent by the consumer using the *transport binding* specified in the interface. Upon receipt of the message the Service Provider must first check for conformity with the requested service's Web Service Description Language (WSDL) interface, then transform the SOAP *request* into the appropriate proprietary message format necessary to invoke the service logic. The response (if any) is then sent back through the same message processing layer.

The message processing layer of the Service Provider, provides a clean separation between the Web Service's infrastructure and the underlying logic and resources used to execute the service. Therefore, the point of execution is outside the boundary of the Web Services framework. This separation between invocation and execution gives rise to a number of key properties of interest to this thesis: *execution transparency* and *loose coupling.*

execution transparency The consumer's only view of the service is its WSDL interface and the endpoint address (stipulated in the WSDL) to which the request message is sent. The consumer has no notion of the final execution location for the request message, being aware only of the published service endpoint address (i.e., the execution of the request takes place beyond the interface boundary). This is execution transparency.

loose coupling Loose coupling is a consequence of separating the interface of a service from its execution. It relies on having agreed semantics and message exchange protocols between the interacting parties, namely: Consumer and Service Provider. For Web Services, this is provided by the Web Service Description Language, SOAP, and an appropriate transport mechanism. Loose coupling allows the implementation of a service to be managed by the Service Provider in any manner sufficient to satisfy the agreed service interface. For example, a Web Service implementation in Java, could be exchanged for a C++ implementation, with no disruption to the WSDL interface semantics, or the Consumer's interaction. The advantage of this loose coupling is that the Service Provider is able to optimise the service provision, while maintaining service semantics.

The clear separation between the message processing and service execution, highlighted above, is a key point to note from this discussion. It is the *execution transparency* and *loose coupling* of the Web Service endpoint from the location of the execution resources that ultimately process the request message, that is exploited in the architecture of DynaSOAr.

3.1.1 Separation of Concerns

Execution transparency highlights an additional consequence: from the point of view of the Consumer, the Service Provider and the resources hosting the service (i.e., the machine upon which the service logic executes) are all bound together behind the Web Service endpoint. It is true that the Consumer is not directly aware of any binding, but it is implied by the bipartite nature of Web Service interactions via a single endpoint address.

The implicit nature of this binding of Service Provider with service host, is seen in the context of this thesis as a limitation. Making this bond explicit presents the opportunity to make a *separation* between the service provision and the host provision.

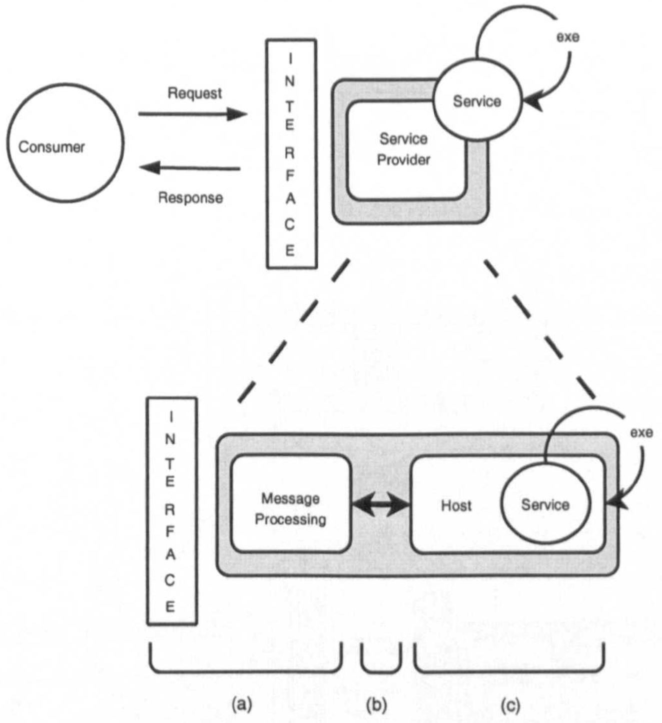


Figure 3.2: Anatomy of a Service Provider

Figure 3.2 allows closer scrutiny of the Service Provider to bring into focus the point of separation between service provision and host provision. The upper portion of figure 3.2 shows a standard Web Service interaction. The lower portion shows an exploded view of the Service Provider to illustrate its components: (a) the interface and SOAP message processing layer, (b) the proprietary message exchange between the message processing and the service implementation, and (c) the hosting resource upon which the service is executed. Point (b) marks the boundary of the Web Services framework and the proprietary implementation of a Web Service. It is this point that DynaSOAr exploits.

By making a clear separation between the service provision and host provision, DynaSOAr introduces a new component into the invocation model, namely that of a Host Provider.

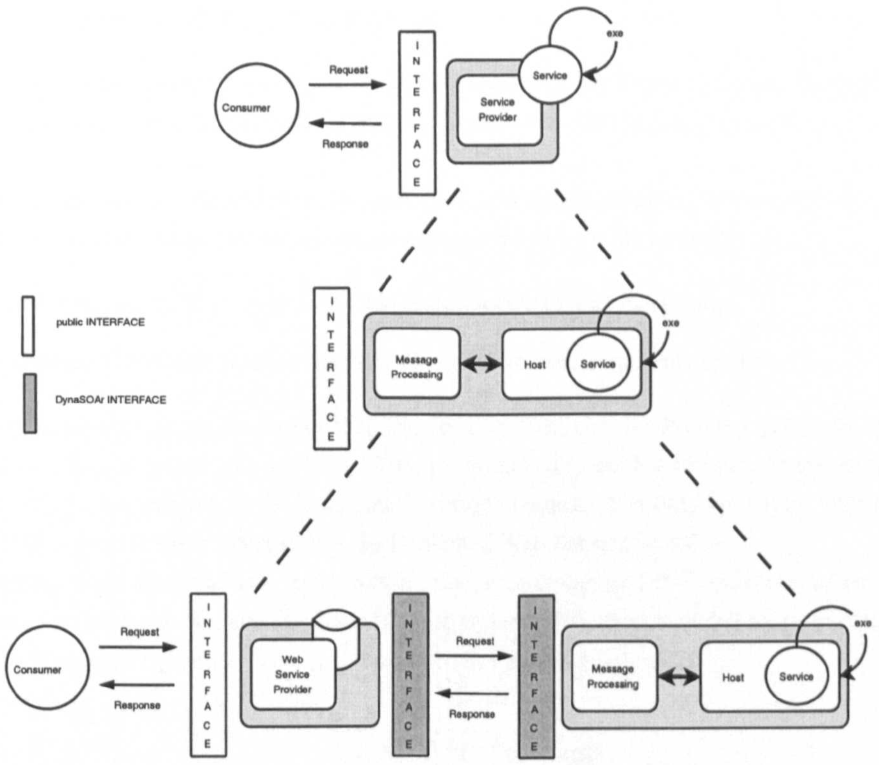


Figure 3.3: Evolution of DynaSOAr

Figure 3.3 shows the final stage in the evolution of DynaSOAr. The upper portion of the figure once again shows the current Web Service model (also shown is the expanded Service Provider from figure 3.2). The lower portion now introduces the separate *Host Provider* into the interaction. The key points to note here are:

1. From this point onwards, the Service Provider is refereed to as the Web Service Provider to reflect the change introduced by DynaSOAr.
2. The public interface (i.e., that published by the Web Service Provider), through which the Consumer invokes the Web Service, is the same as before.
3. Separate interfaces between the Web Service Provider and Host Provider now exist. These interfaces do not relate to the published service; they are concerned with the semantics and message exchange required to enable the Web Service Provider to communicate with the Host Provider to satisfy the Consumer's request.
4. The service is hosted on a host resource under the control of the Host Provider. The message exchange between the Host Provider and the service is now the proprietary exchange.

The tripartite nature of this new architecture has changed the role of the Web Service Provider. The Web Service Provider is now responsible for providing services, and the Host Provider is responsible for providing the resources to execute these services. This shift in roles and their explicit nature is explained in detail in section 3.3 below. Also explained are the necessary interfaces to make this separation possible and still satisfy a Consumer's service request. The Web Service Provider and Host Provider are implemented as Web Services.

3.1.2 Convergence of Enabling Factors

We have shown in section 3.1.1 the separation of concerns. This separation is the key factor that enables the dynamic nature of Web Service provisioning in DynaSOAr. This is now discussed.

Recall the discussion on code mobility (see section 2.2) which highlighted the benefits of dynamically moving code. Consider these two important properties offered by code mobility:

- better use of resources by *matching* code requirements to host capabilities
- deployment at run-time *without* the need to shutdown and reconfigure the host

Also recall the loosely coupled nature of Web Services; the Web Service Provider in DynaSOAr exploits this loose coupling, together with code mobility, to dynamically deploy the service implementation at run-time, without interrupting to the Consumer's service request. The only constraint imposed on this dynamic deployment is that it must satisfy the published Web Service interface.

Harnessing these three factors: *code mobility, loose coupling,* and the *separation of service provisioning and host provisioning,* enables DynaSOAr to dynamically deploy Web Service implementations at run-time. This convergence of factors is illustrated in figure 3.4 below.

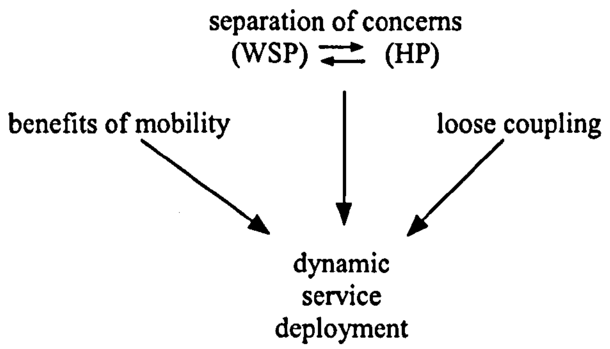
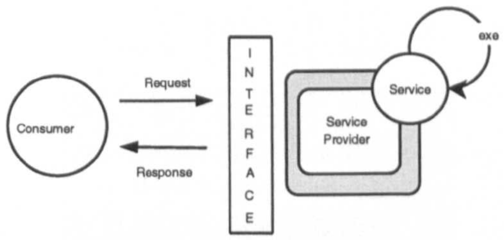
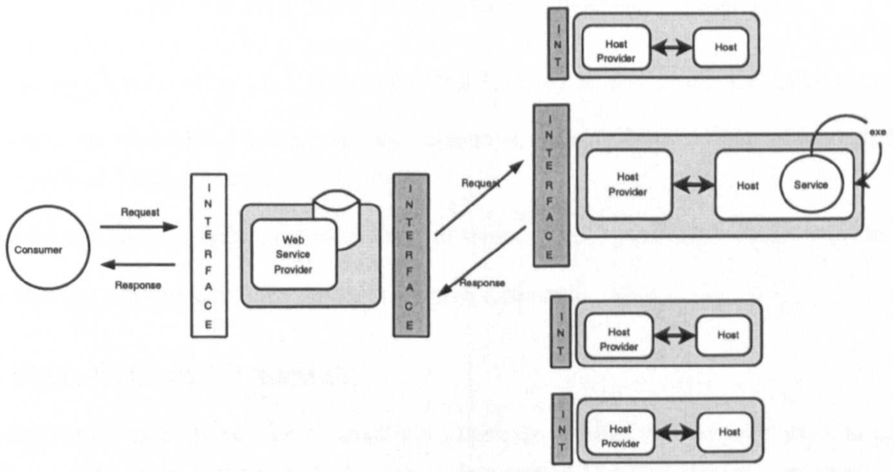


Figure 3.4: Convergence of enabling factors.

Another key advantage of the separation introduced by the DynaSOAr approach is that the *one-to-one* mapping of a Web Service (viewed as a single endpoint address) may be extended to form a *one-to-many* mapping between the Web Service Provider and multiple Host Providers. This allows the Web Service Provider to consider all Host Providers willing to host a particular Web Service deployment. This advantage is illustrated in figure 3.5 below and discussed further in section 3.2.2.



(a) Web Service (1-1)



(b) DynaSOAr (1-many)

Figure 3.5: The one-to-many mapping advantage of DynaSOAr.

By taking the standard Web Services model, recognising the opportunity to separate the elements of current service provisioning into a Web Service Provider and a Host Provider, and introducing code mobility, it has been shown that a one-to-many model for dynamic Web Service deployment is possible. This separation and definition of the DynaSOAr architectural components (Web Service Provider and Host Provider), represents one of the key contributions of this thesis.

Before discussing the specification of DynaSOAr in terms of services and interactions, it is opportune to discuss the benefits arising from adopting the DynaSOAr approach. This discussion will also begin to suggest some application scenarios.

3.2 Opportunities for DynaSOAr

DynaSOAr has introduced a dynamic tripartite model for Web Service deployment and invocation. It has clear benefits, namely it:

- leverages the benefits of *code mobility*.
- maintains the properties of a Service-Oriented Architecture (execution transparency and loose coupling).
- gives the opportunity for improved performance by allowing specific deployments that match the Web Service requirements with the resource (host) capabilities.
- allows consideration of quality of service metrics such as: cost, speed, data-locality, and security, when making a deployment.

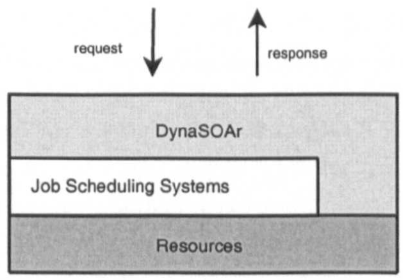


Figure 3.6: DynaSOAr deployed above job-scheduling systems.

- explicitly allows the Consumer instruct the Web Service Provider on service deployment decisions.
- opens up new models for service and host brokerage, creating the possibility of markets for service provision and host provision.
- maintains consistent failure recovery from all levels of the DynaSOAr invocation cycle.

The opportunities presented by these benefits are now discussed.

3.2.1 SOA, Grid and DynaSOAr

Service-orientation uses the service abstraction to describe entities that perform work in response to consumer requests. In recent years, grid research has moved to adopt service-orientation. However, job-scheduling-based systems such as Condor, Globus, and Sun Grid Engine, still form the majority of existing grid systems. As service-based systems come on-line, grid developers must work with both *jobs* and *services*. DynaSOAr allows developers to work in a single Web Service abstraction.

DynaSOAr does not exclude the use of jobs to take advantage of existing job scheduling systems. Using DynaSOAr, a job-scheduling Web Service can be dynamically deployed. This service then handles the transformation of the consumer's Web Service request into a *job* for scheduling. The job is scheduled by the Web Service logic on the existing job-scheduling system. The response from the scheduled job is then transformed by the service logic into the consumer's Web Service response and is then returned to the consumer. This running of DynaSOAr on top of existing job scheduling systems and resources, is illustrated in figure 3.6.

A key advantage of using Web Services instead of jobs is the opportunity for DynaSOAr to take advantage of the *one-to-many* invocation semantics they offer. For example, a dynamically deployed Web Service can be used to satisfy many requests for that service, as once the Web Service is installed it is available for other consumers to invoke. In contrast, a job is a *one-to-one* invocation, whereby a job is scheduled, executed, and then discarded, with no possibility for multiple requests. The saving made by DynaSOAr is in avoiding the cost in having to make a new deployment for each new request. In effect, DynaSOAr is caching Web Service implementations across a potentially large pool of remote Host Providers, each of which is able to satisfy multiple requests for the deployed services.

3.2.2 Improved Performance

DynaSOAr allows the Web Service Provider to deploy a service to a Host Provider. It is perfectly possible that a Web Service Provider will have available many Host Providers capable of resourcing a particular service deployment. This illustrated by the *one-to-many* mapping of service-to-host-deployment options discussed above.

The heterogeneous nature of grid means that the characteristics of the Host Providers are likely to be diverse. The Web Service Provider may, therefore, take into account performance characteristics when selecting the optimum Host Provider. Deployment based on the quality of service required is therefore possible. Examples of these quality-of-service driven deployments are described below:

Consumer Constraints In DynaSOAr a consumer interacts with a Web Service Provider, who in turn interacts with a Host Provider. The opportunity exists for consumers to pass deployment constraints, such as: where to deploy; cheapest deployment; trusted deployment; on to the Web Service Provider. It is noted that this breaks the execution transparency of the Web Service's model, however, in a grid context this possibility for greater consumer choice is seen as an advantage that has potential to improve the consumer's quality of service. (Bear in mind that this is an option, and the usual execution-transparent model is still supported).

Speed If speed is an issue the Web Service Provider should choose the Host Provider offering the fastest-performing resources to deploy the service. For example, consider a researcher who is trying to meet a conference paper deadline and must have the results as fast as possible. By making this constraint known to the Web Service Provider, a Host Provider offering the fastest turn-around can be selected. There is clearly an opportunity for a cost mechanism to be put in place here, to allow a Host Provider the opportunity to charge for better performing resources.

Cost A Host Provider may associate a charge with the resources offered to a Web Service Provider. Consider the same example given above where a researcher is seeking results for a particular experiment. This time there is no deadline. Instead, the research is more concerned with the costs involved in the service invocation. The Web Service Provider identifies the Host Provider offering the cheapest resources.

Data-locality Many grid application scenarios are based around the processing of large data sets. (This was discussed in chapters 1 and 2, and was a key motivation for DynaSOAr.) It will be shown later that the deployment of services close to the data upon which it operates gives a considerable saving in terms of processing time. DynaSOAr negates the need to transfer large amounts of data across the network. The principle adopted is to move the processing, in the form of a Web Service deployment, to the data.

Pre-emptive Deployment In the DynaSOAr model a Web Service Provider deploys a service in response to a consumer's request for that service. The deployment is triggered by the arrival of a consumer's request message at the Web Service Provider offering that particular service. This model does not preclude the pre-emptive deployment of a popular service in order to cope with a high demand. For example, if a service is know to be popular at certain times - possibly owing to an analysis of the past history of invocations - then, the Web Service Provider might deploy certain services ahead of a known peak demand.

Security Security is a key concern for any distributed system - DynaSOAr is no exception. The tripartite nature of the architecture means that each separate party could possibly be deployed within a separate security domain. By giving choice to both the Consumer and Web Service Provider over deployment locations, there is a need to ensure that each party (including the Host Provider) is satisfied that their own security constraints are being observed. Security may, therefore, be an option when considering a deployment choice. An example might be a consumer performing some security-classified experiment and requiring only a trusted Host Provider to perform the service execution, this constraint being passed to the Web Service Provider to ensure a trusted deployment.

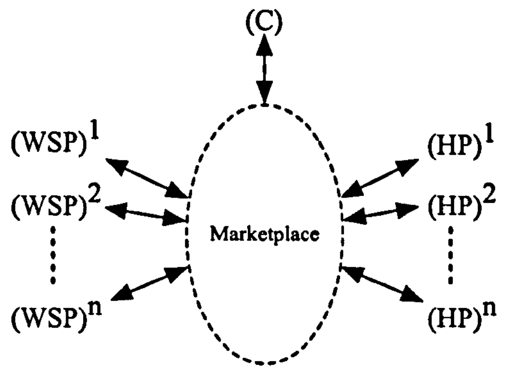


Figure 3.7: Consumer Market for Web Service Providers and Host Providers.

3.2.3 Brokerage and Markets

By making the separation between service provision and host provision, and providing clearly defined interfaces for both service and host providers, DynaSOAr opens up the possibility of markets. Organisations, or individuals, are able to offer both services and resources for use by consumers. For example, a keen researcher has written a new algorithm for processing genomic data. The results are a degree better than any service currently on offer. By hosting the Web Service Provider service and publishing a WSDL document to a suitable *service-broker*, the researcher is able to make available the higher performing service - for a fee of course. The same could be said of a company wanting to make hosting resources available, perhaps after office hours. In this case the company need support only the Host Provider service and publish the Host Provider WSDL to a *resource-broker*. Conceivably, service-brokers and resource-brokers form a marketplace. Consumers are then free to access this market in order to satisfy service requests in a way best suited to their needs. This scenario is illustrated figure 3.7.

3.2.4 Handling Failure

With any distributed system, failure is a concern and may occur at many levels:

1. *At the Host Provider*: failure of a node (host machine) behind the Host Provider interface can be managed internally by the Host Provider directing messages to other nodes with the same service deployed, or, by deploying the service to a new node. With the correct monitoring, those messages currently being processed on a failed node, can be re-sent to another deployment, or, a new deployment can be made to satisfy the requests.

2. *At the Broker or Web Service Provider*: failure of a complete Host Provider can be handled by the broker, or, managed by the Web Service Provider, simply by selecting an alternative Host Provider.

3. *By the Consumer*: failure of a Web Service Provider can be handled through the normal Web Services mechanism of the Consumer querying a registry to find an alternative WSDL for the required service.

Having discussed the rationale, evolution and opportunities afforded by the dynamic service-oriented approach made possible through DynaSOAr, the next step is to discuss use-cases and requirements, before defining the interactions and interface specifications that make-up the architecture.

3.3 Service Deployment in DynaSOAr

The interfaces between the two DynaSOAr components must support two deployment scenarios in response to a consumer's request for a given service:

1. *The service is already deployed*: in this case, the Web Service Provider routes the consumer's request to the Host Provider currently deploying the service where it is then processed.

2. *The service is not deployed*: in this case the Web Service Provider must identify a Host Provider willing to support the service and request that the Host Provider deploy the required service and process the consumer's request.

These two deployment scenarios are discussed below.

3.3.1 Deployment Scenarios

The two possible deployment scenarios are as follow:

Case 1. The service requested by the consumer is already deployed on a Host Provider. This case is shown in figure 3.8. The Consumer sends a *request* to the Web Service Provider (in this case for service S^2). Searching a list of deployments the Web Service Provider determines that service S^2 is already deployed. The Web Service Provider forwards the *request* message to the Host Provider hosting the service. Upon receipt of the message for processing, the Host Provider sends the message to the node hosting the required service (in this case H^n). The hosted service processes the message sending any response back to the Consumer. Notice that from figure 3.8 the Host Provider currently has two deployments of service S^2. The decision on which host to use in routing the request is the choice of the Host Provider. In this case, the decision taken may be due to additional loading on node H^1.

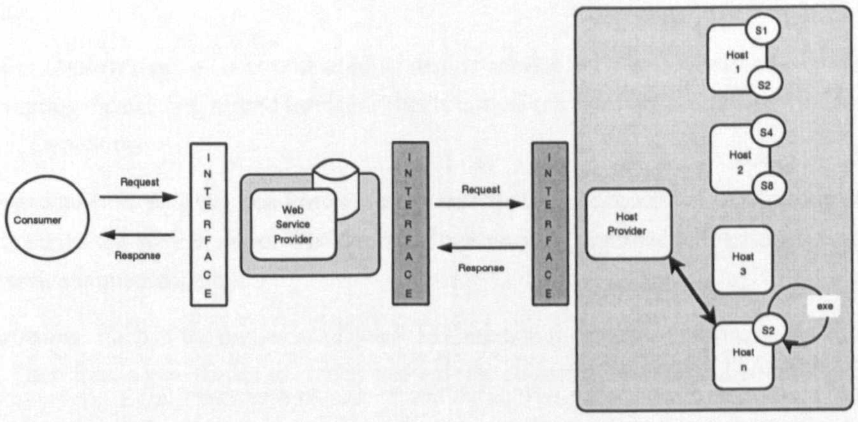


Figure 3.8: Request is routed to an existing service deployment.

Case 2. The service requested by the consumer is *not* deployed on a Host Provider. This case is shown in figure 3.9. The Consumer sends a *request* to the Web Service Provider for service S^2. The Web Service Provider must identify a suitable Host Provider to process the message. Searching a list of deployments the Web Service Provider discovers no current deployment for the service S^2. The Web Service Provider identifies a Host Provider from an available list and sends to it the consumer's message together with a deploy-service *request* (figure 3.9.1). The deploy-service *request* contains an identifier for the service S^2, and a location from where the service can be *fetched*. The Host Provider, upon receipt of the deploy *request*, fetches the service implementation from a service store at the Web Service Provider and deploys the service (in this case, on to host H^1 (see figure 3.9.2)). The choice of host is controlled by the Host Provider and may, for example, be based on load, speed, or cost. The consumer's request message is now processed (figure 3.9.3) and the response sent back to the Consumer via the Web Service Provider.

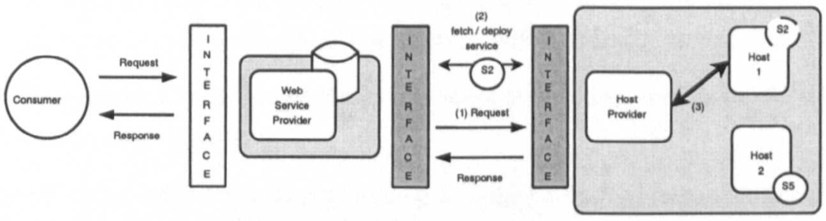


Figure 3.9: A service is dynamically deployed to process a Consumer's request

3.3.2 Requirements for DynaSOAr

Two scenarios have been presented. Before proceeding to discuss the architectural services and interfaces for the Web Service Provider and Host Provider, a set of requirements for DynaSOAr sufficient to support the two scenarios are listed:

1. ***Invocation Routing***: DynaSOAr must be able to route a consumer's *request* to an already deployed service while maintaining synchronicity.

2. ***Mobility***: A service implementation must be able to be transferable from the Web Service Provider to the Host Provider for deployment.

3. ***Host Allocation***: DynaSOAr needs to identify a suitable Host Provider upon which to deploy a service.

4. ***Service Deployment***: A host will need to deploy service code into a service container without interrupting the existing hosted services. This is critical to achieving acceptable service dynamism within DynaSOAr.

5. ***Service Store***: A Web Service Provider must be able to store the set of service implementations for its published Web Services. The Service Store must be available to the Host Provider to *fetch* the service implementation.

6. ***Constraints***: Each of the parties in a DynaSOAr interaction may impose constraints upon the other two. Therefore, a mechanism to convey and enforce constraints is required within DynaSOAr.

7. ***Security***: DynaSOAr has introduced a new tripartite relationship between the Consumer, Web Service Provider, and Host Provider. This raises some key security issues. A new Tripartite Security Model for DynaSOAr is introduced in chapter 4.

8. ***Registration***: A registration mechanism needs to be in place to allow the formation of trust relationships between the Web Service Provider and Host Provider prior to service deployment. In the case of consumer involvement in the choice of Host Provider (discussed above), then consumer registration is also needed. It is recognised that this is a static method of collaboration. A more dynamic method, and one in line with the dynamic nature of DynaSOAr, is the use of UDDI to form virtual organisations.

The requirements listed below are not dealt with in the current specification of DynaSOAr, however, in view of the discussion so far, it is considered useful for the sake of completeness to include them here:

1. ***Accounting***: When services and hosts are made available to consumers there is often a cost involved which the providers will want to recoup. An accounting mechanism needs to be in place to monitor and log invocations and deployments, and collect the required revenues.

2. ***Service Version Control***: Dynamic service deployment is bound to lead to versioning issues with code that is distributed between a number of caches (service stores, Host Providers). Peer-to-peer technologies may provide a solution to this, together with a notion of service lifetime.

3. ***Service Revocation***: Closely linked to version control, the need to revoke a service whose lifetime has expired, becomes important.

4. ***Message Routing***: Policy control over routing messages is seen as important when dealing with sensitive result sets, or service requests from which intent may be inferred. For example, a researcher from a pharmaceutical company using a shared data-resource would be wary of routing any message via an untrusted intermediary, for fear of revealing the aims of their research to a competitor, and therefore adopts a message routing standard, such as WS-Addressing [45]. It is clear that message level encryption would add a layer of confidentiality independent of routing.

5. ***Service Level Agreements***: A Host Provider, in registering its resource with a Web Service Provider, may offer its resource at a certain performance level based on the characteristics of its resources. This level of service may be contractually bound by using service level agreements. This would be necessary in a situation where the Web Service Provider is making deployment decision based on quality of service metrics.

3.3.3 Design Issues

This section discusses design issues arising from the discussion of Dynasoar which impact on the specification of the Dynasoar infrastructure.

3.3.3.1 Message Headers

The SOAP Envelope illustrated in figure 3.10 on the following page shows the components of a SOAP message. Of interest here is the header element. The header element is used to hold information associated with, but not necessarily part of, the intended message. Header information is passed from one intermediary to another, for example, the SOAP header is often used to convey security information. Intermediaries and the intended recipient of a SOAP message can ignore, or read, the header information of interest. DynaSOAr uses the SOAP header mechanism to convey information on deployment and constraints between the parties involved.

3.3.3.2 Service Store

Services implementations must be available for deployment on to Host Providers. The Web Service Provider, therefore, must maintain a *service store* of the service implementations and associated files for all the WSDL descriptions published by the Web Service Provider. The service store must be available for Host Providers to fetch service implementations.

```
<soap:envelope>
  <soap:header>
    <!-- information intended for intermediary -->
  </soap:header>
  <soap:body>
    <!-- message intended for recipient -->
  </soap:body>
</soap:envelope>
```

Figure 3.10: SOAP Envelope

3.3.3.3 Meta-data

To facilitate the matching of service requirements with host capabilities it is necessary to associate meta-data with each service available. Suggested meta-data might include: service type, author of the service, cost per invocation, and access policy.

Likewise it necessary to associate meta-data with each available Host Provider. Suggested meta-data might include: available machine types (memory, CPU, software libraries) and cost per invocation.

These lists are by no means exhaustive. There is a large range of possible data that could be held and then used to influence deployment decisions, or optimise performance, however, the storage and use of this data for adaptive scheduling is beyond the scope of this thesis.

3.3.3.4 Push/Pull Model

An issue that arose in the DynaSOAr architecture design was the choice between a push or pull model for deploying the requested service implementation.

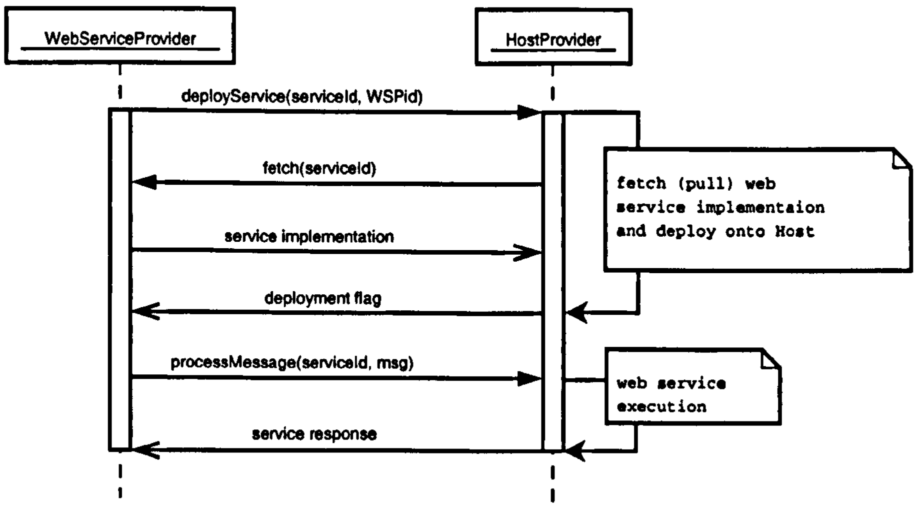
A pull model was adopted. This involves the Web Service Provider requesting a *deploy* operation on the Host Provider, passing the required service identifier, the Host Provider then responds by calling a *fetch* operation on the Code Store, pulling the service implementation code and deploying the service onto local resources. Once deployed, the Web Service Provider is notified and the consumer's request messages is passed on for *processing*. The author wanted to maintain a clean separation between the deploy phase and the process phase in the architecture as it opens a number of possibilities: pulling the code means there is potential to use a third party code store service for supplying the implementation; a two phase deployment allows security to be handle at each stage, which was considered an advantage; if a code deployment fails, then no message has been passed with no associated security risk from snooping, a new deployment can then be requested. It is recognised that an additional phase is required, but this was considered an acceptable trade-off.

The alternative *push* model, where the Web Service Provider pushes the required service code, together with the consumer's request message, on the selected Host Provider was also considered and adopted in later implementations of DynaSOAr. The service is deployed and the request message processed with only one operation, with no *fetch* operation. The different implementations of DynaSOAr are discussed in section 5.1.

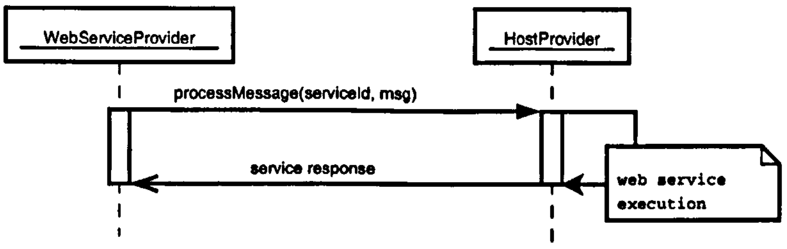
The description of the architecture shown below is using the pull model.

3.3.4 DynaSOAr Services

This section describes the services of the DynaSOAr infrastructure. Each high-level component of the architecture is realised as a Web Service. Each Web Service presents the operations available to a calling consumer via a published WSDL interface. The combined operations of the infrastructure enable the dynamic deployment of a Web Service in accordance with the two scenarios presented in section 3.3.1.


WebServiceProvider
HostProvider
deployService(serviceId, WSPid)
fetch(serviceId)
service implementation
deployment flag
processMessage(serviceId, msg)
service response
fetch (pull) web service implementaion and deploy onto Host
web service execution

(a) Service is not already deployed.



(b) Service is already deployed.

Figure 3.11: DynaSOAr Service Interactions.

Each of the DynaSOAr Web Services presented is then broken down into the underlying components that form the logic of each service. The role of these internal components and the interface of each service are discussed below; the service/component interactions illustrating the two scenarios are also discussed. To aid the reader in understanding these interactions, two high level diagrams are shown in Figure 3.11. Figure 3.11 (a) shows the case where a service is not already deployed. This requires the Host Provider to fetch the service implementation from the requesting Web Service Provider before processing the message. Figure 3.11 (b) shows the case where the requested service is already deployed. In this case the message sent to the Host Provider to be processed and the response returned. These figures illustrate the DynaSOAr operations called during the two deployment cases dicussed in section 3.3.1 - these operations are dicussed in detailed in the following sections describing each DynaSOAr service.

3.3.4.1 Consumer

The consumer is not a service within the DynaSOAr infrastructure, but clearly plays an integral part in any service invocation.

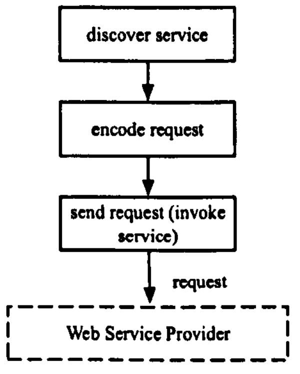


Figure 3.12: Consumer invocation work flow.

The Consumer initiates a service invocation by first discovering the required service. This is done in the same way as a standard Web Service is discovered. After the discovery phase, the Consumer then interacts with the Web Service Provider by encoding a request message that conforms with the discovered WSDL interface, and then sends it to the service endpoint. This interaction is shown in figure 3.12 above and is a standard Web Service call.

3.3.4.2 Web Service Provider

To satisfy the DynaSOAr infrastructure, the Web Service Provider needs to support the following requirements. It must:

- receive incoming request messages from consumers and arrange for them to be processed by an available Host Provider.
- store the service implementations for the services it has published.
- store a set of registered Host Providers sufficient to satisfy requests for all the published services.
- allow Host Providers to register, update, or remove their details.

These requirements give rise to the high-level Web Service operations illustrated in figure 3.13; a description of each operation is given below:

- **`serviceImplementation`** **`fetchService(+serviceId)`**

  inputs
  `serviceId` - URI of the service to be fetched
  outputs
  `serviceImplementation` - the service implementation bundle

- **`registered`** **`registerHostProvider(+hostProviderId, +hostProviderDetails)`**

  inputs
  `hostProviderId` - URI of the Host Provider (endpoint)
  `hostProviderDetails` - details to be registered
  outputs
  `registered` - Boolean indicating success/failer

- **`updateded`** **`updateHostProvider(+hostProviderId, +hostProviderDetails)`**

  inputs
  `hostProviderId` - URI of the Host Provider
  `hostProviderDetails` - update details
  outputs
  `updateded` - Boolean indicating success/failer

- **`removed`** **`removeHostProvider(+hostProviderId)`**

  inputs
  `hostProviderId` - URI of the Host Provider
  outputs
  `removed` - Boolean indicating success/failure

Figure 3.13: Web Service operations for the Web Service Provider.

– *fetchService()* - to complete a Web Service deployment, the Host Provider must be able to *fetch* the requested service implementation. The *fetchService()* operation allows the Host Provider to retrieve an implementation bundle. This bundle will contain all the necessary code, associated libraries, configuration files, and other associated items required to deploy and run the requested service as a Web Service.

– *registerHostProvider()* - a Web Service Provider must have a set of available Host Providers to satisfy a Consumer's request. The *registerHostProvider()* operation allows a Host Provider to register its details. The result of a *registerHostProvider()* call is that the registering Host Provider is now available to satisfy calls to the *processMessage()* and *deployService()* operations (see below, section 3.3.4.3 Host Provider). The details a Host Provider must include when registering, are:

  * the endpoint address (URI identifier) of the Host Provider service being offered;
  * the capabilities of the Host Provider, to allow the Web Service Provider to match service requirements with host capabilities;
  * the policy constraints the Host Provider has placed upon access to the hosting resources offered.

– *updateHostProvider()* - a Host Provider must be able to *update* the registered details held by the Web Service Provider. The *updateHostProvider()* operation allows a Host Provider to update its registered details. The result of a call to the *updateHostProvider()* operation is that the Host Provider details are updated. Updating should include options to ADD, REMOVE, EDIT elements of the Host Provider's details. The Host Provider's endpoint is unique and immutable so cannot be updated. A Host Provider may only have one URI identifier (endpoint). The update details can include any of the following:

  * the endpoint address (URI identifier) of the Host Provider service being offered;

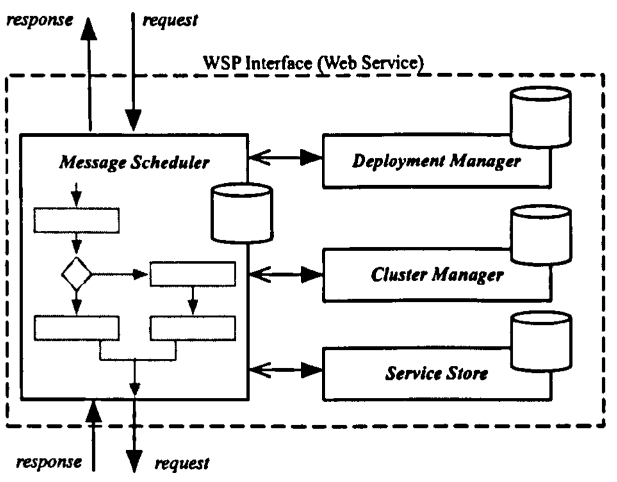


Figure 3.14: Internal components of the Web Service Provider.

- * the capabilities of the Host Provider, to allow the Web Service Provider to match service requirements with host capabilities;
- * the policy constraints the Host Provider has placed upon access to the hosting resources offered.

- ***removeHostProvider()*** - a Host Provider must be able to remove its registered details from a Web Service Provider. The *removeHostProvider()* operation allows a Host Provider to *remove* its details from the Web Service Provider. The result of a *removeHostProvder()* operation is to *remove* the Host Provider matching the *hostProviderId* as an available resource, from the Web Service Provider. Also, any current deployments to Host Providers matching the *hostProviderId* are also *removed*. (This does not affect any current calls to the *processMessage()* operation - these will respond as expected.)

Internal Components of the Web Service Provider (refer to figure 3.14)

1. **Deployment Manager**: The Deployment Manager maintains a view of all currently deployed services. This *service-view* is a mapping between service identity and all endpoints at which the service is currently deployed; this is potentially a one-to-many mapping for a given service. At its most basic level this is a look-up table, or database, although this view could be implemented as a private UDDI registry. The key used by the Deployment Manager to query the database is the URI of the service being requested. This URI is unique to the service in question. Any query for the current deployment(s) of a given service requires the service URI to be passed to the Deployment Manager. The Deployment Manager queries the database using the URI, and returns a set of all current deployments. This set will be a set of the Host Provider endpoints currently hosting the service that matches the query. An empty set indicates no current deployments. Figure 3.15 shows the internal operations of the Deployment Manager.

```
-[]hostProviderEndpoints ←—— getDeployedServiceEndpoints(+serviceId)

    inputs
      serviceId - URI of requested service
    outputs
      hostProviderEndpoints - set of URIs of Host Providers

-success ←—— updateServiceViewEndpoints(+serviceId, +[]hostProviders, +action)

    inputs
      serviceId - URI of requested service
      []hostProviders - array URIs of HPs with currently deployment of serviceId
      action- update action (ADD, REMOVE)
    outputs
      success - boolean indicating successful update

-success ←—— removeHostProvderServices(+hostProviderId)

    inputs
      hostProviderId - URI of Host Provider
    outputs
      success - boolean indicating successful removal
```

Figure 3.15: Deployment Manager internal interface.

2. **Cluster Manager**: The Cluster Manager maintains a view of all registered Host Providers. At its most basic level this is a look-up table, or database. When asked to do so by the Message Scheduler (discussed below), the Cluster Manager must match service requirements with Host Provider capabilities. The key used by the Cluster Manager is the URI for the Host Provider. The URI is unique. The Cluster Manager uses the key to query the database and returns a set of Host Providers capable of satisfying the consumer's request. The Cluster Manager must also support the *register*, *update*, and *remove* operations offered by the Web Service Provider. Figure 3.16 shows the internal operations of the Cluster Manager.

```
-[]hostProviderEndpoints ←—— matchServiceToHostProviders(+serviceId)

    inputs
      serviceId - URI of requested service
    outputs
      hostProviderEndpoints - set of URIs of Host Providers

-success ←—— updateHostProvider(+hostProviderId, +hostProviderDetails)

    inputs
      hostProviderId - URI of Host Provider to update
      hostProviderDetails - new Host Provider details
    outputs
      success - Boolean indicating successful update

-success ←—— removeHostProvder(+hostProviderId)

    inputs
      hostProviderId - URI of Host Provider
    outputs
      success - Boolean indicating successful removal

-success ←—— addHostProvder(+hostProviderId, +hostProviderDetails)

    inputs
      hostProviderId - URI of Host Provider
      hostProviderDetails - new Host Provider details
    outputs
      success - Boolean indicating successful addition
```

Figure 3.16: Cluster Manager internal interface.

3. **Service Store**: The Service Store holds an implementation for each of the services offered by the Web Service Provider. In addition, it holds descriptions of the service requirements (used to match a service to a Host Provider), and any additional libraries required by the service. (No provision has been provided for updating a service at this stage). Figure 3.17 shows the internal operations of the Service Store.

- **`serviceImplementation`** ⟵ **`getServiceImplementation(+serviceId)`**

  inputs
  `serviceId` - URI of the service
  outputs
  `serviceImplementation` - the service implementation bundle

- **`serviceRequirements`** ⟵ **`getServiceRequirements(+serviceId)`**

  inputs
  `serviceId` - URI of the Service requested
  outputs
  `serviceRequirements` - the requirements of the service sufficient to match with a Host Provider

- **`removed`** ⟵ **`removeService(+serviceId)`**

  inputs
  `serviceId` - URI of the Service
  outputs
  `removed` - Boolean indicating success/failure

Figure 3.17: Service Store internal interface.

4. **Message Scheduler**: The Message Scheduler is the decision point within the Web Service Provider, and the component responsible for scheduling consumer requests using either of the two scenarios presented in section 3.3.1. The Message Scheduler must also maintain a view of all consumer messages currently being processed. This is to ensure that responses from the *processMessage()* operation on the Host Provider are routed to the correct consumer. The Message Scheduler interacts with the other internal components and warrants a more detailed discussion.

   Figure 3.18 below illustrates the three key stages of the Message Scheduler in handling a request. These are discussed below:

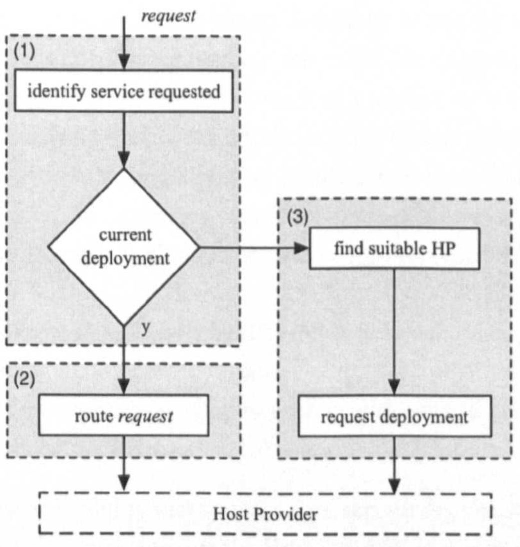


Figure 3.18: Stages in scheduling a request message

stage (1): Common to both scenarios is the identification of any already deployed service that matches the Consumer's request. This involves a look-up in a table of current deployments. Using the URI of the service requested as a key, the Web Service Provider queries the Deployment Manager. The query returns a set of current service deployments. The returned set will contain the URIs (endpoint addresses) of the Host Provider service(s). This set will contain zero-to-many Host Provider endpoints and is considered to be the candidate set. If the candidate set contains one or more Host Providers then Message Scheduler proceeds to stage (2), otherwise it proceeds to stage (3).

stage (2): If the candidate set returned by the Deployment Manager has a cardinality of one, the consumer's request message is routed to the Host Provider hosting the service (by calling the *processMessage()* operation), where it will be processed and any response sent back to the consumer. If the candidate set has a cardinality greater than one, then each member of the set already has the requested service running and is a possible Host Provider. Where more than one Host Provider is available, there is an opportunity for further refinement in the choice of Host Provider. This can be achieved using a number of different metrics (this was discussed previously in section 3.2.2, Improved Performance). It suffices to say that a Host Provider is chosen, and the message is routed on to be processed (by calling the *processMessage()* operation); any response is sent back to the consumer. If the candidate set has a cardinality of zero, then there is no current deployment available at a Host Provider. In this instance the Web Service Provider proceeds to stage (3).

stage (3): This stage involves three actions: finding a suitable Host Provider, requesting a new deployment, and requesting a *processMessage()* operation. Finding a suitable Host Provider involves matching a service's requirements to a Host Provider's capabilities. The URI of the service being requested is used by the Web Service Provider to query the Cluster Manager. This query will return a set of candidate Host Providers. In this instance, a candidate Host Provider is one able to support the service being deployed. The deployment now continues in one of two ways depending upon the cardinality of the candidate set: if the candidate set has a cardinality of one, then the deployment continues by calling the *deployService()* operation on the Host Provider. A successful response to this triggers the Message Scheduler to call the *processMessage()* operation with the Consumer's request. The response to this call is then sent back to the Consumer. If the candidate set has a cardinality greater than one, then a choice must be made. Where more than one Host Provider is available, there is an opportunity for further refinement in the choice of Host Provider. This can be achieved using a number of different metrics (this was discussed previously in section 3.2.2, Improved Performance). It suffices to say that an Host Provider is chosen and the deployment continues as above, with any response being sent back to the Consumer.

It is necessary to update the service-view held by the Deployment Manager in response to a deployment of a new service. Updates allow a consistent view of the deployed services to be maintained. An *update* is required in three cases: (a) after each new successful service deployment; (b) when a service deployment becomes *inactive*; or, (c) when a Host Provider withdraws its resources.

(a) A new service deployment is successful: a new service deployment is considered successful when the service becomes *active* at the Host Provider to which the *deployService()* request was sent. The response from the Host Provider to the *deployService()* request is therefore used to notify the Web Service Provider of the state of the deployment. The implementation of the Host Provider service, therefore, must handle exceptions being raised during the service deployment phase. If an exception is raised and the Host Provider cannot recover from it, a deployment *failed* response is sent to the Web Service Provider, otherwise, a deployment *success* response is sent. The Web Service Provider will handle the response in one of three ways:

    i. In the case of a *success* response, the Web Service Provider will update the service-view with the Host Provider URI of the successful deployment. This is done by calling *updateServiceViewEndpoint()* on the Deployment Manager.

    ii. In the case of a *failed* response, the Web Service Provider will try the next Host Provider in the set. If no more Host Providers are in the set, then the WSP has failed to deploy the service and must, therefore, respond to the consumer's request with an *exception*. The consumer must then discover a new Web Service Provider to satisfy the service request (see section 3.2.4, Handling Failure).

    iii. If a timeout occurs (i.e. the Host Provider has failed), this is dealt with in the same way as a *failed* response.

(b) A service deployment becomes *inactive*: a service is considered *inactive* when the Host Provider, upon which the service is deployed, responds to the *processMessage()* operation with an *exception*. (This is an over simplification, as an exception could indicate many different errors. For the purpose of this discussion, however, an exception indicates a failure in processing the consumer's message). The Web Service Provider responds to this exception by updating the Deployment Manager to indicate that the service in question is *inactive*. At this stage there is still a need to process the consumer's request. The Web Service Provider must therefore find a different service deployment, or request a new deployment to satisfy the consumer's request. If no deployment, or new Host Provider can be found, then an exception is raised and sent to the consumer, i.e. the Web Service Provider has failed to satisfy the original request. (This is handled as described in section 3.2.4, Handling Failure). If a timeout occurs in the *processMessage()* operation (i.e. the Host Provider has failed), then this is handled by the Web Service Provider by selecting a new Host Provider (see section 3.2.4, Handling Failure).

An *inactive* service deployment does not indicate a failure of the Host Provider logic behind which the service is operating, merely that the service is not responding as expected. To clear the *inactive* status from a service deployment, the Web Service Provider must contact the Host Provider to request an investigation of the service; this may involve a restart of the host machine for example. This management mechanism is outside the boundary of this work.

(c) The Host Provider withdraws its resources: in the case of a Host Provider withdrawing resources, perhaps outside the limits of a timed access widow, then the Cluster Manager must remove the Host Provider from the view of available Host Providers, and the Deployment Manager must update the service-view by removing all service deployments associated with the withdrawing Host Provider.

3.3.4.3 Host Provider

The requirements the Host Provider needs to support in order to satisfy the DynaSOAr infrastructure are, to:

- deploy a new Web Service as instructed by the Web Service Provider, and
- process a consumer's message on an already running Web Service deployment.

These requirements give rise to the high-level Web Service operations illustrated in figure 3.19; a description of each operation is given below:

```
-success ←— deployService(+serviceId, +webServiceProviderId)

  inputs
    serviceId - URI of requested service
    webServiceProviderId - URI of WSP holding Service Store
  outputs
    success - Boolean indicating success of deployment

-reponse ←— processMessage(+serviceId, +message)

  inputs
    serviceId - URI of requested service
    message- the Consumer's message
  outputs
    response - result of processing Consumer's message
```

Figure 3.19: Web Service operations for the Host Provider.

- *deployService()* - to complete a Web Services deployment, the Host Provider must be able to *deploy* the requested service implementation. The *deployService()* operation allows the Web Service Provider to deploy a service on to a host node under the control of a Host Provider. Once the service is deployed, a *flag* is returned indicating the *success/failure* of the deployment.
- *processMessage()* - to *process* a consumer's request, the Web Service Provider calls the *processRequest()* operation on a Host Provider with a current deployment of the service requested. The *serviceId* and the *message* are passed to the Host Provider. The message is then processed and the result is sent back as the *response*.

Internal Components of the Host Provider (refer to figure 3.20)

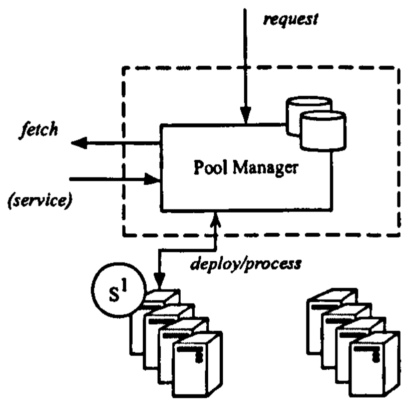


Figure 3.20: Internal components of the Host Provider.

1. **Pool Manager**: The Pool Manager has control over the resources offered by the Host Provider. It must maintain a look-up table, or database, mapping the current service deployments to specific hosts, and also maintain an additional table of Web Service Providers (those where the Host Provider is registered). The Pool Manager acts in one of two ways:

   (a) A request is received to *deploy* a service. The request message will contain the service identity and the Web Service Provider identity (endpoint). The Host Provider uses the service identity and Web Service Provide endpoint to invoke the *fetchService()* operation. The Web Service Provider responds with the service implementation. The service implementation is deployed as a Web Service running on a machine under the control of the Host Provider. When the Web Service is deployed, the Host Provider passes the original consumer's request for processing. When the processing is complete the response is sent back to the Web Service Provider.

   (b) A request is received to *process* a message. The request will contain the service identity and the consumer's original message. The Host Provider uses the service identity to find the deployed service. It then sends the consumer's request message for processing. When processing is complete the response is sent back to the Web Service Provider. There is an opportunity here to balance the load between multiple deployments for the same service. For example, if the Host Provider has four instances of service S^1 running on separate nodes, then the Cluster Manager can optimise the processing by using some performance metrics before routing the message.

3.4 Summary

This chapter presented a dynamic service-oriented architecture (DynaSOAr) capable of dynamically deploying Web Services in response to consumer requests. The rationale and requirements for the architecture were discussed. These drew from the opportunities inherent in service-orientation, code mobility, and a separation of service provision and resource provision. Opportunities for the DynaSOAr infrastructure were then discussed to illustrate the benefits of dynamic service deployment in the context of grid computing. Finally, a detailed specification of the architectural components and interfaces of DynaSOAr are described. The architecture supports two invocation scenarios which were also presented. An implementation of DynaSOAr is used for evaluation in chapter 5.

As discussed in chapter 1, a dynamic deployment system that moves code between a service provider and separate resource raises security issues. These issues are discussed in the following chapter.

Chapter 4

Tripartite Security

In chapter 3 the separation of concerns into service provision and resource provision was introduced. The specification of a Web Service Provider and a separate Host Provider, together with the invoking consumer, forms a three-party relationship in each DynaSOAr service invocation. This chapter now presents a tripartite security model for DynaSOAr which represents, unifies, and enforces the combined security constraints of all these three parties in any one Web Service invocation. It focuses on access control (authorisation) constraints.

Section 4.1 discusses security constraints arising from a typical DynaSOAr invocation; section 4.2 presents the design of a tripartite security model for access control constraints; section 4.3 describes the application of XACML in implementing the proposed security model; section 4.4 describes the architecture of the security components used to integrate the model into the DynaSOAr infrastructure; and finally, section 4.5 briefly summarizes the work presented in this chapter.

4.1 Security Constraints

A constraint is a limitation or restriction placed on someone or something [80]. In the context of DynaSOAr, a constraint refers to a limitation placed on one party in a service interaction by another party. The following sections describe and discuss a typical DynaSOAr scenario, firstly by considering constraints that may be imposed upon them, and second by analyzing these constraints. This analysis reveals the concepts that must be embodied in a tripartite security model capable of enforcing each party's constraints in any given invocation.

4.1.1 Typical Scenario

The principle of an Active Information Repository [5], where processing is carried out as close to the data as possible, is a good application for DynaSOAr. Presented below is a typical scenario, which uses an Active Information Repository (AIR) to provide remote processing and access to a large data store. The scenario is built around a DynaSOAr service deployment that brings together the three entities involved - this is illustrated in figure 4.1 below:

- a consumer (a bio-informatics researcher working through a client application)
- a Web Service Provider offering data analysis services (herein referred to as Bio-Services Limited)
- and a Host Provider (herein referred to as Data-Resources Limited)

Dynasoar Scenario: An independent researcher is working on gene-data analysis. Part of the experiment requires processing of gene-sequence data located at a remote data provider (Data-Resources Ltd). The researcher aims to exploit the benefits of DynaSOAr and the Active Information Repository by requesting the required Web Service to be deployed locally with the data (i.e. at Data-Resources Ltd). The benefit of this being that large amounts of gene-sequence data need not be transferred across the network between the data-source and the processing resources. The researcher, therefore, identifies the required analysis service, published in a publicly available registry (see figure 4.1.1), and sends a service request message to the Web Service Provider stipulating that the required deployment should take place at Data-Resources Ltd (see figure 4.1.2). The Web Service Provider instructs the Host Provider to deploy the service (see figure 4.1.3). The researcher's request message is processed close to the data and the execution response is sent to the researcher's application (see figure 4.1.4).

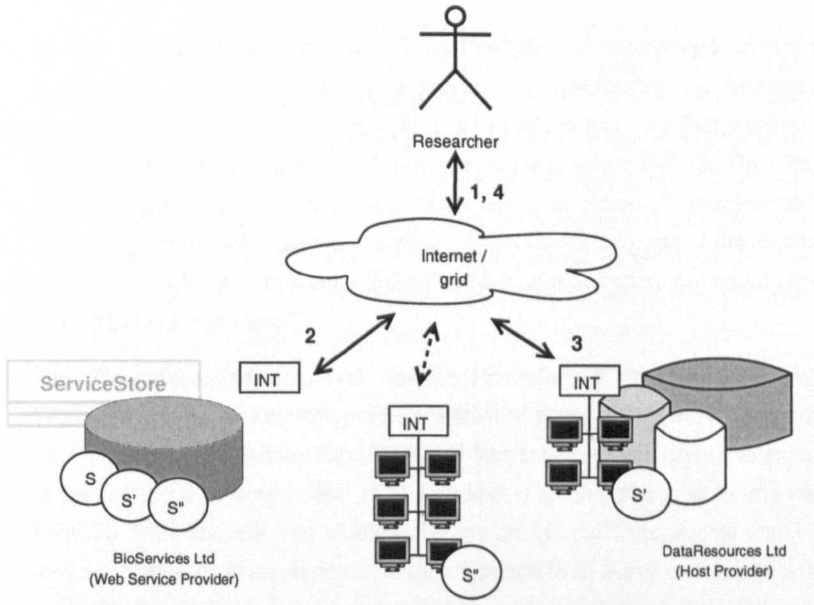


Figure 4.1: Dynamically deploying a service onto an AIR architecture.

4.1.2 Example Constraints

In DynaSOAr there are three bipartite relationships that give *2 x 3 = 6* one-way relationships. Each relationship may give rise to a set of constraints. This is illustrated in figure 4.2:

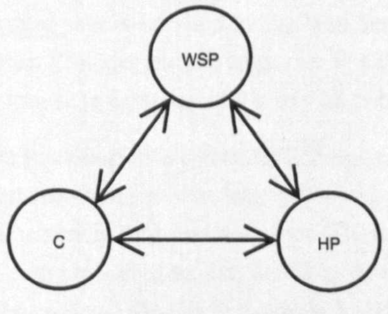


Figure 4.2: Tripartite constraint relationships.

Considering the scenario in subsection 4.1.1, a number of example constraints are now presented that show the typical security considerations each party may make, given the prior relationships and understanding between the parties involved.

1. **Consumer constrains Web Service Provider**: One of the key benefits of DynaSOAr is that it allows a consumer to specify a set of potential Host Providers. By doing so the consumer is constraining the Web Service Provider to a set of Host Providers upon which a message may be executed. The consumer may impose this constraint for a number of reasons: security, data locality, processing speed.

2. **Web Service Provider constrains Consumer**: A Web Service Provider makes available a service by publishing its endpoint. The Web Service Provider may wish to constrain a Web Service consumer by allowing only certain consumers access to a given Web Service. This may be for trust issues, or non-payment for previous invocations in the case of a chargeable service. To be able to enforce these constraints the Web Service Provider has a set of known consumers who are trusted and have paid their fees. This is a typical Web Service constraint scenario for authentication.

3. **Web Service Provider constrains Host Provider**: In deploying a service to a Host Provider, the Web Service Provider is passing the service implementation to a third party for hosting (deployment) and subsequent execution. The Host Provider has, in effect, cached the service implementation. In this context, the Web Service Provider is aware that the Host Provider could make this service implementation available to other users, or process messages not sent by the Web Service's originating Web Service Provider. This breach of *trust*, if discovered, may cause the Web Service Provider to constrain a Host Provider, and therefore not make further deployments until any conflicts are resolved.

4. **Host Provider constrains Web Service Provider**: A Host Provider may wish to constrain the Web Service Provider on the grounds of trust. For example, a service recently deployed may have caused a denial-of-service for other Web Service Providers owing to excessive database activity in processing a message. The Host Provider may, therefore, move the originating Web Service Provider from the allowed to the not-allowed list until the anomalous Web Service can be reviewed. Subsequent messages from the restricted Web Service Provider will therefore be rejected. A further example of a constraint might be a temporal constraint. For example, a Host Provider may wish to make available a set of resources that are not always in use (e.g. out of office hours) and therefore, want to enforce a window of access during which all requests may be accepted. Access requests outside this time window will be rejected, or, a premium charge may be payable for access outside the specified window.

5. **Host Provider constrains Consumer**: A Host Provider may wish to constrain a particular consumer from using the resources being offered. This constraint is not imposed directly between the Host Provider and the consumer, but is imposed by the Web Service Provider, the latter being the intermediary between the Host Provider and the consumer in a Dynasoar interaction. In effect, the Web Service Provider constrains the Consumer by proxy on behalf of the Host Provider.

6. **Consumer constrains Host Provider**: The consumer imposes a constraint of a set of Host Providers on the grounds of trust, perhaps owing to past leakage of important information that resulted in the loss of considerable investment in time and resources. These constraints are passed to the Web Service Provider encoded in the message header, and it is the responsibility of the Web Service Provider to enforce these constraints. Similar to example 5 above, these constraints are imposed on the Host Providers on behalf of the consumer by the Web Service Provider.

4.1.3 Analysis of Natural Language Constraints

As described in subsection 4.1.2, a constraint is a limitation placed upon one party by another party. Examining the examples above, a constraint can be seen as a *rule*. For example, a Host Provider constrains access to resources by putting in place a *rule* that states:

> Resources are only available between the hours of 18:00 and 06:00, Monday to Friday.

Rules may apply to *attributes* associated with the resource for which access is being requested; they may also apply to the requester, and to the environment in which the request is being made. The rule above is an example of the Host Provider imposing an environmental constraint by restricting access to resources to a particular time window, also imposing a condition on the day of the week the request is made - this is another environmental constraint. By using attributes, parties are not restricted in the nature of the rules they impose, as a wide range of attributes can be described. For example, a further rule may state:

> The invoking consumer must have an available credit balance of not less than GBP300.

This rule requires that a consumer (requester) has a minimum credit balance.

To determine if a requester adheres to a rule, a conditional operation must be performed on the request information and the constraints encoded in the rule. This suggests that *requests, rules, attributes*, and *conditional operators* must be expressed in a common language. A mechanism associated with the language enables the evaluation of the necessary conditional operators, thereby giving an *outcome* to a rule.

A party may wish to combine a set of rules into a policy. For example, a number of rules that pertain to a particular resource might be grouped to form a policy controlling that resource. Each rule, or policy, may be concerned with a different set of attributes.

Considering the full set of rules, across all the six possible relationships of a tripartite system, a key issue is the determination of the specific combination of rules (policies) which must be imposed for a given invocation. Different combinations of rules will apply depending upon the parties involved in an invocation and the attributes and actions associated with the request. Therefore, the *applicability* of a set of rules for a given invocation (C, WSP, HP combination) needs to be resolved.

Describing, combining and matching applicable rules is only useful if the applicable rules can be *enforced.*

4.1.4 Requirements for Tripartite Security Model

The analysis described above gives rise to a number of requirements needed in a system able to deliver a tripartite security model for DynaSOAr.

1. ***Encoding of Rules***: natural language constraints must be able to be expressed as rules in a policy language that allows run-time evaluation and enforcement.
2. ***Attributes***: the designation and encoding of attributes is needed for the requester, the resources, and the environment in which a request is made.
3. ***Policy Combining***: a tripartite system that aims to enforce the policy constraints of all three parties must be able to combine the rule(s) imposed by each party involved for a particular invocation.
4. ***Policy Discovery***: given the wide range of possible policies, a suitable discovery mechanism is required to match an access request to an applicable policy set. This allows the set of all constraints in the system to be reduced to the applicable set for a given invocation.

5. ***Decision Enforcement***: a request made against a policy set must yield a definitive access decision (access/no access); this decision must always be enforced.

4.2 Tripartite Security Model (Sets and XACML)

This section describes, using simple sets, how the requirements of tripartite access-control can be expressed. The section continues by describing an available policy language and protocol, based on sets and Boolean algebra, which is used to implement the requirements for a tripartite security model for DynaSOAr.

The following notation is used when describing sets.

Symbol	Meaning	Example	Read as:
$\in$	element of	$x \in A$	"x is an element of A" or "x in A"
$\notin$	not element of	$x \notin A$	"x is not an element of A"
:	such that	$x \in A : R(x)$	"x in A such that $R(x)$ is true"
$\cup$	union	$A \cup B$	"A union B"
$\cap$	intersection	$A \cap B$	"A intersection B"
$'$	complement	A'	"A complement"

Table 4.1: Notation used to describe sets.

4.2.1 Modeling Constraints using Sets

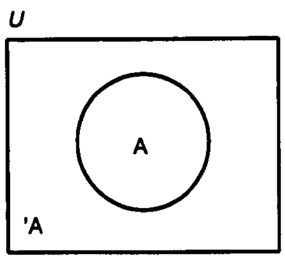
The following constructs and concepts are considered and modeled: rules, policy, outcomes, rule/policy combining, and matching.

Rules: A rule is seen as a constraint placed upon one set of users by another set. By examining a single rule a clearer picture of this can be shown and illustrated using sets.

Consider the example of a consumer making a request to a Web Service Provider for access. A rule is in place that states: *access is only granted to requests from consumers of the group 'developer'*. For a consumer to be compliant with this rule, they must satisfy the conditions of the rule. The condition states: *consumers must be of group 'developer'*. If we consider the set of all consumers, then the set of allowed consumers (allowed meaning that they comply with the rule) is the set of all consumers that comply with the condition of the rule. This can be expressed as a general case, thus:

$A\,(the\,set\,of\,allowed consumers) = \{x|x \in A : R(x)\}$ where $R(x)$ is a condition that applies to x.

This can be represented using a Venn Diagram, as shown in figure 4.3 below:



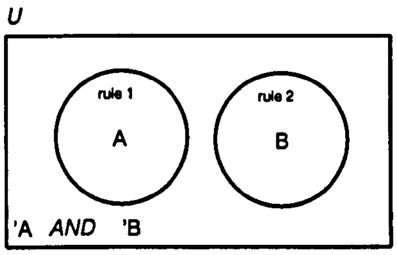
U	the set of all consumers
A	the set of all consumers that comply with the rule R(x)
'A	the set of all consumers that do not comply with R(x)

Figure 4.3: Rule modeled as a subset with conditional membership.

Policy: A policy wraps a set of rules. For example, two rules that constrain a Web Service Provider might be: *access is only granted to trusted consumers*, and, *access is only allowed at weekends.* These two rules set conditions on an attribute of the requester (that it is trusted[1]), and an attribute on the environment (a weekend). The two rules can be described thus:

$$A\,(the\,set\,of\,trusted\,consumer\,requests) = \{x|x \in A : R(x)\}$$
$$B\,(the\,set\,of\,weekend\,requests) = \{x|x \in B : P(x)\}$$

This can be represented using aVenn Diagram, as shown in figure 4.4 below:



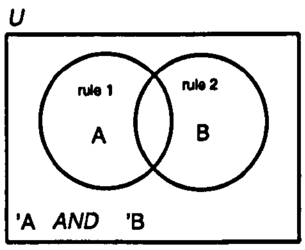
U	the set of all consumers
A	the set of all consumers that comply with rule R(x)
B	the set of all consumer making requests that comply with rule P(x)
'A AND 'B	the set of all consumers/requests that do not comply with either rule

Figure 4.4: Policy modeled using multiple Rules.

Outcomes: An outcome is the result of applying a rule to a given consumer request. Shown in the above two examples, the outcome of a rule produces two sets of consumers: those that comply with a rule, and those that do not. Therefore, membership of the set of those that comply means that access may be granted for that rule. Compliance with a policy will depend upon how individual rule-outcomes are combined.

[1]Trust is a complex issue that is not debated in this thesis. In the context of this work, the issues of trust have been resolved outside the boundary of DynaSOAr. It is sufficient to assume that a trusted consumer is one who is a member of the set of trusted consumers.

Combining: An access request may require more than one rule, or more than one policy, to be applied to determine an overall outcome. The combination of rules can be shown by the combination of sets.



A∩B	the set of all consumers that comply with both *rule 1 AND rule 2*

Figure 4.5: Combining rule outcomes using sets.

Shown by the Venn Diagram in figure 4.5, the combination of two rules gives rise to an additional set of consumers at the intersection. There are three set combinations that will satisfy all outcomes for two rules, these can be combined to satisfy multiple sets of rules and policy outcomes. These can be categorised by the following sets: 1) the Intersection, 2) the Union, and 3) the Relative Complement.

1. $C\,(the\,set\,compliant\,with\,Rule\,1\,AND\,Rule\,2) = \{x|(x \in A) \cap (x \in B)\}$
2. $D\,(the\,set\,compliant\,with\,Rule\,1\,OR\,Rule\,2) = \{x|(x \in A) \cup (x \in B)\}$
3. $E\,(the\,set\,compliant\,with\,Rule\,1\,but\,not\,compliant\,with\,Rule\,2) = \{x|(x \in A) \cap (x \notin B)\}$

In addition to combining the outcomes of rules, it may be the case that a given request is controlled by a set of policies. This requires that the outcomes of different policies be combined. This can be achieved in the same way as rule outcomes are combined; the final outcome being the result of the combination of the outcome of each policy. This is illustrated in figure 4.6, which shows a set of two policies where all four rules apply:

The policy combination is shown as the intersection between the policy outcomes F and G. This results in the set of consumer requests that comply with all four rules stipulated in the policy set. The members of this set (H), may be granted access. This can be shown thus:

$H = F \cap G$

where

$F = A \cap B$

$G = C \cap D$

and

$A = \{x|x \in A\ :\ Rule1(x)\}$

$B = \{x|x \in B\ :\ Rule2(x)\}$

$C = \{x|x \in C\ :\ Rule3(x)\}$

$D = \{x|x \in D\ :\ Rule4(x)\}$

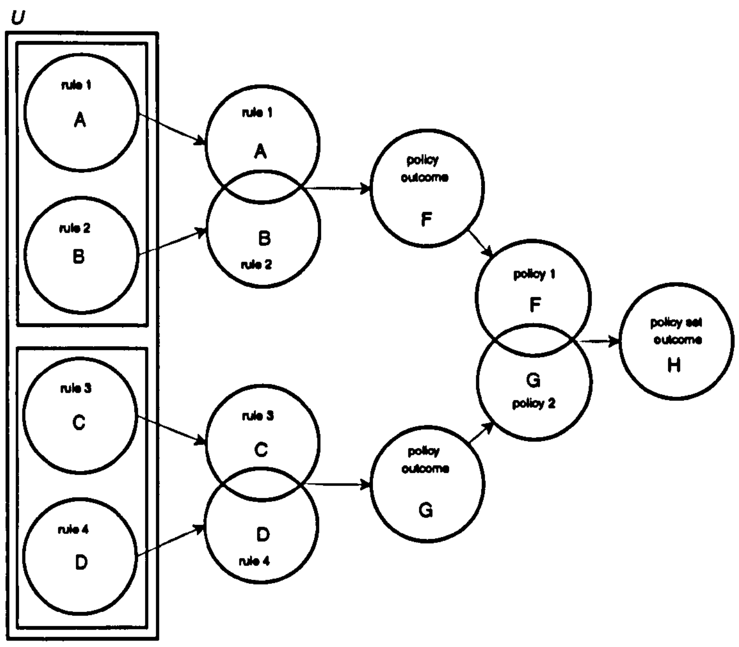


Figure 4.6: Combining rules and policy outcomes as sets to reach a final outcome.

Matching: The universal set (U) in the examples above is the set of applicable consumers. The applicability of a consumer request to a rule, or policy, is determined by matching the request to the attributes of the rule or policy. The set U is therefore formed by those consumers whose request attributes match the rule/policy attributes governing applicability.

4.2.1.1 Modeling a Tripartite Relationship

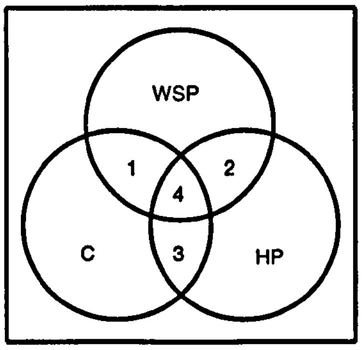
In section 4.2.1, rules, policies, and combinations were modeled to show how policy constraints can be expressed using sets. DynaSOAr is a tripartite system. The purpose of the security model presented in this chapter is to allow the expression and enforcement of the constraints of all three parties for a given invocation. By extending the number of parties from two, as in a standard Web Service setup, to three, merely requires an additional set of rules to be considered when matching and enforcing the constraints of the system.

Consider three sets:

- A, the set of constraints imposed by the consumer
- B, the set of constraints imposed by the Web Service Provider
- C, the set of constraints imposed by the Host Provider

Each of these sets contains the rules imposed by one party on each of the other parties. So, A will contain the constraints on the Web Service Provider, and the constraints on the Host Provider, both imposed by the consumer.

Now consider the combination of these three sets. This is best illustrated using a Venn Diagram (see figure 4.7 below) :



(a) The sets A, B, and C

Region	Triple Intersection	Description
C	C	Consumer Policy
WSP	WSP	Web Service Provider Policy
HP	HP	Host Provider Policy
1	$C \cap WSP \cap HP'$	Constraints between C+WSP
2	$C' \cap WSP \cap HP$	Constraints between WSP+HP
3	$C \cap WSP' \cap HP$	Constraints between C+HP
4	$C \cap WSP \cap HP$	Tripartite Constraints C+WSP+HP

Figure 4.7: Tripartite Policy Modeling using sets and set combinations.

The descriptions of the intersections above, show how the combination of three sets of rules across a tripartite relationship can be expressed. The intersection sets (1, 2, 3, 4 in figure 4.7) will contain those consumer requests that conform to the rules imposed by the main sets (C, WSP, HP).

Using the modeling constructs shown above in subsection 4.2.1, rules and policy can be expressed, and outcomes can be derived, by matching and combining the three separate sets of constraints for a given consumer invocation.

To implement the tripartite model described above, a mechanism is required that encapsulates the requirements described in subsection 4.1.4, and the modeled behavior shown above; this mechanism is described below.

4.2.2 XACML

The eXtensible Access Control Markup Language (XACML) is an OASIS standard [66] that describes a general-purpose access control policy language. The standard is in two parts:

- a language for describing a set of rules as policies
- a request/response protocol that describes functionality for discovering, combining and enforcing policy/rule outcomes.

In combination, the two components of the XACML standard enable the combining of rules and the combining of policies to determine the outcome (access control decision) of a given request against a set of applicable policies.

Policies may be written to control access to distributed resources. Policies can refer to other policies and be grouped together to form policy sets that control access to a set of resources. The benefit of this is that resource administrators need not manage a single policy for all resources, but divide access control amongst a set of specific policies. This division is equally applicable to a system topology where the parties involved in a system interaction are distributed across a number of distinct domains. This would allow the consumer, Web Service Provider, and Host Provider to express separate policies, which can then be combined at run-time.

The following subsections describe the XACML language and protocol.

4.2.2.1 XACML Policy Language

XACML is written using XML. It gives a detailed and extensible syntax for expressing access control policy. The specification also allows for extensions to the functions, attribute designation, and combining algorithms that enforce the policy decisions. This permits detailed application-specific functions to be written and enforced. This general mechanism gives fine grained expression and control of policy, rules, and outcomes.

The XACML policy language is constructed from the following components. (The definitions are summerised from the XACML 1.0 Specification [66]).

Rule: A rule is the main component of a policy. Rules allow policy authors to encapsulate constraints by defining *conditions*, which are then evaluated against a given request. Rules may stipulate a number of conditions that must be met in order to fulfill the requirements of the rule. Conditions are evaluated by performing functions on a set of *attributes*. If the conditions are met (the functions evaluate to *true*), then the effect of the rule is the permitted *outcome*. The three components of a rule are:

1. **Target**: The target of a rule defines three areas that describe the context in which a rule must be applied:
   (a) **Subject**: the name/identifier of the entity making a request. For example this might be a machine name, or an email address of the requester. (A number of standard identifying structures have been encoded into the language). Inside a subject it is possible to specify a subject-category. This will allow multiple subjects to be specified, each of which has an accompanying category.
   (b) **Resource**: the resource (a Web Service, a document, or a building) to which access is being requested.
   (c) **Action**: the action being requested, for example, invoke in the case of a Web Service, or read in the case of a file.
2. **Effect**: the result of a *true* evaluation of the rule conditions. This can be either *PERMIT* or *DENY.*
3. **Condition**: an optional Boolean expression used to refine the predicates implied by the rule target.

Policy: a policy encapsulates a set of rules. There are four components to a policy:

1. **Target**: the policy target defines the three areas which apply to a policy. A policy target has the same components as a rule target (described above).
2. **Rules**: the set of rules associated with a policy.
3. **Rule-combining algorithm**: the rule-combining algorithm defines how the outcome of each rule evaluated as part of a policy decision, is combined to give the final outcome to a policy.
4. **Obligations**: obligations are constraints placed on the Policy Enforcement Point (explained further in subsection 4.2.2.2) which must be enforced before allowing access.

Policy Set: a policy set may be used to encapsulate a set of policies and other policy sets. A policy set consists of four components:

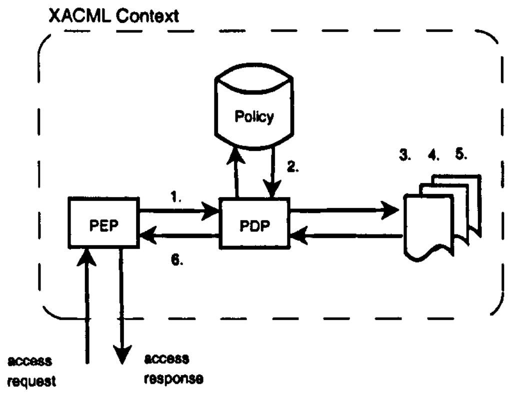
1. **Target:** the policy-set target is constituted in the same way as that for a policy or rule target.
2. **Policies:** the set of policies associated with a policy set.
3. **Policy-combining algorithm:** the policy-combining algorithm of a policy set defines how the decision outcome of each policy evaluation is combined to give a final outcome to a policy set.
4. **Obligations:** Obligations provide a feedback mechanism that allows the access controller to stipulate certain conditions on the access being granted. For example, rather than simply saying: "*OK, you can pass.*", the XACML obligations mechanism allows for: "*OK, you can pass, but only if...*"

Matching: Targets (subject, resource, action) are used to match a context request against an applicable policy and/or rule. If all three predicates in a policy, policy-set, or rule-target match those encoded in the request context, then a positive match is made and the rule/policy/policy-set is then evaluated. The predicates associated with a target match can be extended with additional *attributes*. Attributes refine the matching of targets by adding meta-data to the subject, resource, action, or environment. For example, a subject's name, security clearance, or the time a request is made, are all attributes and can be used to enhance the richness of the data available for matching requests to rules, policy, or policy-sets.

4.2.2.2 XACML Protocol

An XACML system is wholly encapsulated by the language and protocol, and can be deployed at a point within a system to provide a policy enforcement point (PEP) to enforce policy outcomes. Its generic nature is such that it can be deployed for many application and system set-ups.

A relevant set-up for access control, and one which illustrates the control mechanisms described above, might be one where an entity wishes to access a resource to perform a particular action, for example, invoke a Web Service. In XACML, the requesting entity, or subject, makes an application-specific request to a Web Service. The request is routed to the policy enforcement point where it is encapsulated as an XACML context request. (The XACML request encodes the subject, resource and action that formed the consumer's application request in a standard way understood by the XACML system components). The request is then routed to a policy decision point (PDP). The policy decision point is responsible for parsing the context request, finding and matching the request target (subject, resource, action) to an applicable policy, policy-set, or rule and evaluating the applicable rule(s). The policy decision point then replies to the policy enforcement point with a decision (outcome) encoded in an XACML response. The policy enforcement point then enforces the policy decision, obeying any obligation stipulated by the policy decision point. Figure 4.8 below illustrates this simple protocol:



1. XACML Context Request
2. Target Matching
3. Rule Evaluation
4. Policy Evaluation
5. Policy Set Evaluation
6. XACML Context Response

Figure 4.8: The XACML protocol stages.

4.3 XACML and DynaSOAr

Clearly XACML can be used to encode and enforce the constraints of the tripartite security model described above in subsection 4.2.1.1. Sets of resources (the published Web Services, the Web Service Provider services, or the Host Provider services) may have access to them controlled via different policies, which may be combined to form a single policy for a given invocation/deployment request. Each party in a DynaSOAr request may encode a separate sets of constraints. These constraints are then combined into a policy-set with rules that may cover: subject, resource, action, and environment. The mechanism for matching requests to applicable rules is then used to ensure that the intended constraints are enforced for a given request.

4.3.1 Policy Representation

The policy and policy-set constructs of XACML are used to wrap a set of rules and policies respectively. The key language construct to focus on, however, is the rule, as this forms the foundation of all policy.

An example of rule representation is given below (see figure 4.9). The `<Target>` of this rule stipulates a subject to which the rule applies. The natural language constraint represented by the condition of the rule, states:

all requests from cs.ncl.ac.uk must be from the research-group 'escience'.

```
<Rule RuleId="Rule 1" Effect="Permit">
<!-- Controlling access for cs.ncl.ac.uk requestors>
<Target>
   <Subject>
     <SubjectMatch
       MatchId="xacml:function:rfc822Name-match">
      <AttributeValue DataType="#string">
      cs.ncl.ac.uk
      </AttributeValue>
      <SubjectAttributeDesignator
         DataType="xacml:data-type:rfc822Name"
         AttributeId="xacml:subject:subject-id"/>
     </SubjectMatch>
    </Subject>
</Target>
<Condition FunctionId="xacml:function:string-equal">
 <Apply FunctionId="xacml:function:string-one-and-only">
  <SubjectAttributeDesignator DataType="#string"
    AttributeId="research-group"/>
 </Apply>
 <AttributeValue DataType="#string">
  escience
 </AttributeValue>
</Condition>
</Rule>
```

Figure 4.9: Representing constraints as an XACML rule construct.

By stipulating a matching function on an attribute of the subject of the rule target, the XACML matching mechanism will only apply this rule to requests that match the stipulated requirement. In the case of the example presented in figure 4.9, this states that the subject identifier (an email address) must be of the form:

anyname@cs.ncl.ac.uk

If the policy decision point is presented with requests and cannot find a match, an error that states no applicable policies found will be returned to the policy enforcement point. The policy enforcement point must decide how this will be handled.

Rules are expressed in terms of attributes of the target with conditions being imposed on those attributes. Boolean functions are used to determine compliance with the condition. In the case of the example given in figure 4.9, the condition states that:

the request must have an attribute 'research-group', and the value for this must be 'escience'

The representation of the tripartite constraints of a DynaSOAr invocation can be expressed using XACML. The following subsection describes how these constraints can then be unified to form the controlling policy for a given invocation.

4.3.2 Policy

One of the key requirements for the DynaSOAr security model (see subsection 4.1.4) is the run-time unification of separate sets of constraints. It has been shown that the mechanism for combining policies and rules exists as part of the XACML infrastructure. The purpose of this subsection is to consider the sources, propagation through the system, and the unification of these tripartite constraints. This will lead to the definition of a flow-diagram that shows the policies which need to be enforced.

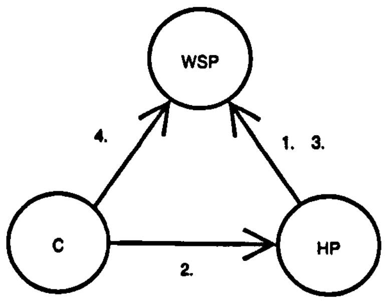
4.3.2.1 Sources

Three actors are involved in a DynaSOAr invocation; each will provide a set of constraints. Each set may contain zero-to-many constraints, and be sub-divided into two sets that constrain each of the other parties in an invocation. The sub-division is only logical, as the target and matching mechanisms discussed previously allow rules and policies to be matched against specific subjects. This mechanism allows for a single policy containing multiple rule-targets. For example, the Web Service Provider's policy may govern all Web Services being offered, however, each rule in the set may have an individual target; some of the rules may cover consumer access, and some Host Provider deployment.

4.3.2.2 Propagation

Six sets of constraints have been identified (see subsection 4.1.2) in a DynaSOAr invocation. To be able to unify these into an over-arching invocation policy, these constraints need to propagate through the system to be available to the policy decision point.

It has been stated (see subsection 3.3.4) that for a Host Provider to be eligible to receive Web Service deployments, it must first register with a Web Service Provider. Likewise, if a consumer wishes to stipulate a specific Host Provider upon which a Web Service must be deployed, then it must have registered with that Host Provider. Any consumer stipulation is passed in the header of the SOAP request as a constraint on the Web Service Provider offering the service requested. These communication channels provide a means of propagating constraints (either as policy at registration, or constraints at run-time). Figure 4.10 below details the stages and channels of constraint propagation in a DynaSOAr invocation:



1. Host Provider registers resources with the Web Service Provider. An initial constraints policy is passed to be stored by the Web Service Provider.
2. Consumer registers with the Host Provider.
3. The Host Provider updates the stored constraints policy with all Web Service Providers with which it is registered.
4. The consumer invokes the Web Service, passing its constraints policy encoded in the SOAP header of the request message.

Figure 4.10: Constraint propagation stages for each party relationship.

Constraint propagation between all three actors is directed towards the Web Service Provider. This is intentional, as it is the Web Service Provider who must unify and enforce the tripartite constraints.

4.3.2.3 Unification

As discussed in subsection 4.2.2, the XACML specification provides rule and policy combining algorithms used to combine the decisions of separate policies. Policies may be combined into policy-sets.

A rule-combining mechanism exists that allows the outcomes from multiple rules to be combined into a single outcome. Also, a target-matching mechanism ensures that only applicable rules/policies are evaluated against a given access request. This mechanism allows the specification of subjects and subject-catagories.

Figure 4.11 below shows how policy unification in DynaSOAr occurs for a service invocation. Because the target matching mechanism allows applicability to be defined at the policy level and/or at the rule level, the *combination policies* (those formed by two or more individual policies) are logical rather than explicit - they exist at run-time as a result of combinning rules from multiple policies. For example, the set of rules forming the invocation policy is made up from the applicable rules from the deployment policy and the consumer's policy for a specific invocation request.

A top level policy with no target specified results in the rule-targets being used to determine applicability of the rules. For example, a Web Service Provider policy may stipulate no Policy Target, but define fine-grained Rule Targets. If this policy covers all the Web Services the Web Service Provider has published, then the policy that covers a specific Web Service call is formed by a subset of the rules of this single policy. This subset could be referred to as the policy governing the specific Web Service targeted in the invocation - although it is not written as a separate policy in its own right.

The policies defined in a tripartite DynaSOAr interaction are as follows. (Note that those marked with an asterisk are a combination of two or more of the other policies).

- Consumer Policy
- Web Service Provider Policy
- Host Provider Policy
- Deployment Policy*
- Invocation Policy*

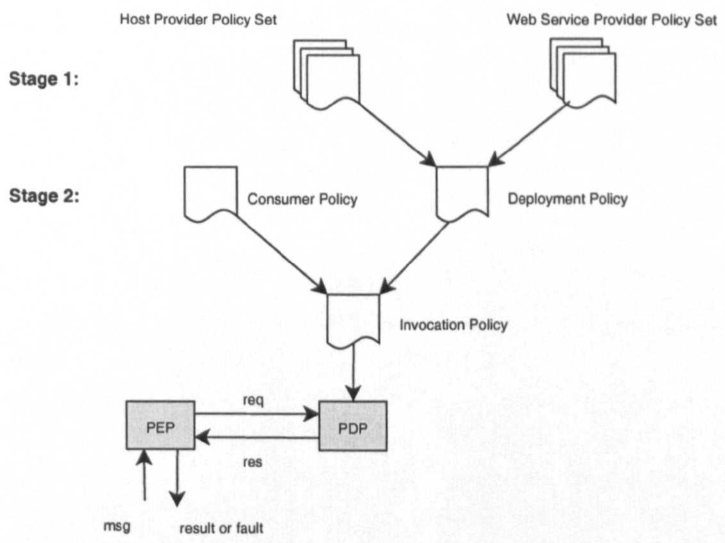
Consumer Policy: the policy formed by the consumer's constraints passed in the header of the consumer's SOAP request, and encoded using XACML.

Web Service Provider Policy: the combination of all policy and policy-sets covering all published Web Services for this Web Service Provider.

Host Provider Policy: the combination of all policies registered by a Host Provider. These will include any updates made as a result of a Consumer's registration with a Host Provider.

***Deployment Policy:** the result of combining the Host Provider Policy with the Web Service Provider Policy.

***Invocation Policy:** the result of combining all three sets of constraints for a given DynaSOAr invocation; it will be specific to the three parties involved. The outcome from the evaluation of this policy is the outcome of the access request for a single invocation.



Stage 1: The inputs to this stage are the Host Provider Policy and the Web Service Provider Policy. The output of this stage is the Deployment Policy relevant to both parties. This will reflect the combined constraints of the Host Provider and the Web Service Provider.

Stage 2: The inputs to this stage are the Deployment Policy and the Consumer Policy. The output of this stage will be the Invocation Policy. This is enforced by the policy decision point. This enforcement evaluates the service access XACML request message against the combined policies of consumer, Web Service Provider and Host Provider (Invocation Policy).

Figure 4.11: Policy unification in DynaSOAr.

4.3.2.4 Enforcement Enactment

To unify and enforce a set of constraints using the XACML they must be encoded to ensure:

- the objective of the constraint is encoded using the extensible *function* capabilities of the XACML language;
- the policy *rules* apply the extended functions to evaluate the required constraint;
- the rules are *targeted* to apply to subjects, which may be described further using subject-catagories;
- the *resources* and *actions* are consistent with the service-list held by the Web Service Provider and Host Provider;
- the *policy-combining* is defined in such a way that all three sets of constraints are applied to any one service invocation request.

The policy sources, policy propagation, policy unification and enactment described above, jointly ensure that the policy decision point (located at the Web Service Provider) has access to the targeted XACML policies/rules for each of the three actors involved in an invocation.

4.4 Dynasoar Security Components

The DynaSOAr infrastructure is a set of strictly defined Web Services (described in subsection 3.3.4) that enables the dynamic deployment and invocation of Web Services in response to consumer requests for those services. The propagation and unification of a tripartite policy has been described (see subsection 4.3.2) and the supporting mechanisms of the XACML that allows the expression, combination, and enforcement of this policy, have also been discussed.

The following subsections describe issues of design, the architectural components required to build a tripartite security infrastructure, and the way these components are integrated into the current DynaSOAr infrastructure. The architecture is concerned only with authorisation at this stage, however, before authorisation can be given and subject must be *authenticated.* This is discussed briefly below.

4.4.1 Authentication

An authorisation system that enforces control-policy based on subject, action, and resource, requires authentication. To authenticate one party against another, identity needs to be exchanged and then confirmed within the context of a predetermined trust relationship.

There are six relationships in DynaSOAr (see subsection 4.1.2). These can be grouped as three two-way relationships. It has been described how identity is exchanged in two of these three sets (C<>HP and WSP<>HP) as part of the DynaSOAr registration. This means that during invocation, identity assertions, perhaps in the form of digital-certificates, or Security Assertion Markup Language (SAML) tokens, may be passed between parties and authentication established by comparing stored identities against those passed at run-time. The third relationship (C<>WSP) is not dealt with by registration.

The relationship between a calling consumer and a Web Service Provider is established at run-time. The need for the Web Service Provider to establish the identity of a calling consumer is required in order to enforce the targeted constraints imposed by the two other parties. The Security Assertion Markup Language [67] is an OASIS standard that specifies the transportation and verification of security assertions across administrative domains.

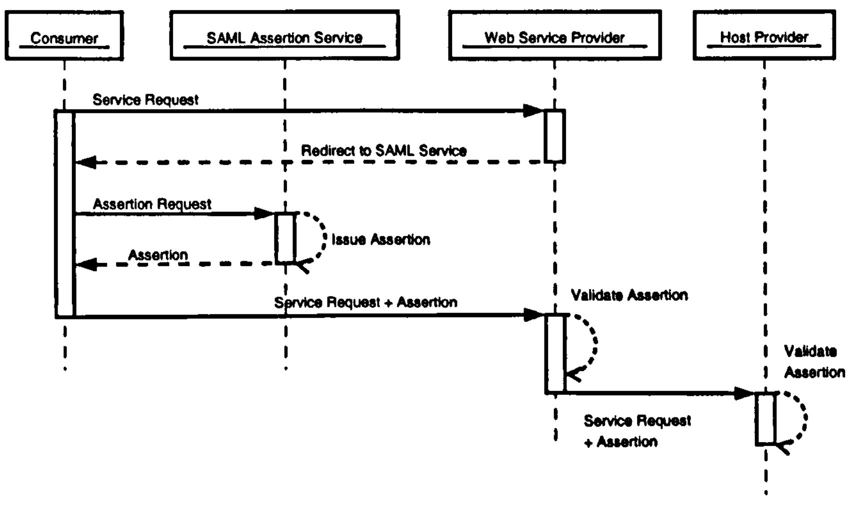


Figure 4.12: SAML protocol for authentication between multiple DynaSOAr parties.

Figure 4.12 above shows how a security assertion made by an asserting party is propagated and used by a relying party to determine identity. The assumption upon which SAML is based is that a trust relationship already exists between the parties in an interaction. This mechanism is an accepted solution to both identity assertion and single sign-on for multi-party interactions similar to DynaSOAr.

4.4.2 Security Manager

The Security Manager is the authorisation component within the Web Service Provider. It is responsible for authorising consumer requests against the combined policy constraints of the Web Service Provider, the Host Provider and the calling consumer. The authorisation gives rise to a set of authorised Host Providers upon which the Message Scheduler (one of the Web Service Provider's internal components, see section 3.3.4.2) may deploy a Web Service in-order to process a consumer message, or process a message at an already deployed service.

4.4.2.1 Interface

The Security Manager is responsible for providing a set of authorised Host Providers. The authorisation is based upon the evaluation and enforcement of the security constraints of all three parties in an invocation. The interface to this component hides much of the policy control and handling of authorisation requests. The interface is illustrated in figure 4.13 and described below.

```
[]hostProviderEndpoints        <——  getAuthorisedHostProviders(+consumerRequest,
                                                                +[]candidateHPSet)
   inputs
        consumerRequest the calling consumer's invocation request
   outputs
        hostProviderEndpoints set of URIs of candidate Host Providers
```

Figure 4.13: Security Manager interface operations.

- *getAuthorisedHostProviders()* - this high-level interface accepts the consumer's request message, which may or may not contain an encoded policy, and a set of at least one candidate Host Provider identifier(s) (URIs). This operation uses these two pieces of data to interact with the components of the Security Manager to return a set of zero-to-many *authorised* Host Providers.

4.4.2.2 Components

The following components, illustrated in figure 4.14 below, are defined to satisfy the requirements of the Security Manager's interface.

Access Controller The Access Controller is responsible for handling a request for authorisation to determine a set of authorised Host Providers. A call to the *getAuthorisedHostProviders()* operation on the Security Manager passes the consumer's request and the set of candidate Host Providers, determined by the Deployment Manager, to the Access Controller. The Access Controller must parse the request to determine: a) the consumer's policy constraints, b) the target service. The candidate set of Host Providers is used to specify the relevant set of Host Provider policies for this invocation.

Policy Writer The Policy Writer is responsible for transforming any specified consumer constraints into a valid XACML policy. This will be the Consumer Policy for this invocation. The Consumer Policy stipulates the target to which the policy applies, and the rules that must be enforced. The Consumer Policy is given a unique name and stored in the Policy Store.

Policy Store The Policy Store holds XACML policy for all parties. For the purposes of this work, two simple operations are required to maintain the store:

createPolicy(consumerRequest) creates a new XACML policy from the constraints passed in the consumer's request. The policy is given a unique name; this is returned to the calling operation.

deletePolicy(policyName) deletes a redundant policy. Given that each invocation is stateless, then once a authorisation decision has been given, the Consumer Policy is deleted. Any later request will contain a new set of constraints. These will be encoded and stored temporarily as before.

Request Writer The Request Writer is responsible for encoding a valid XACML request for authorisation. The request contains the target of the original consumer's request (subject, resource, action).

PEP The Policy Enforcement Point is responsible for enforcing the access decision of the Policy Decision Point. To determine the access decision, the PEP makes an XACML context request to the PDP.

PDP The Policy Decision Point is responsible for making an access decision based on the policies of the three parties involved.

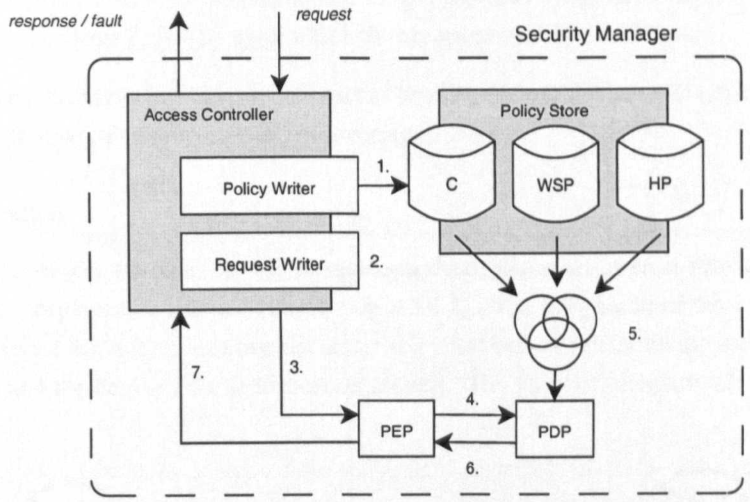


Figure 4.14: Message flows within the Security Manager.

The relationships between the components described and the message flows are illustrated in figure 4.14 and described below:

Figure 4.14-1 The Access Controller parses the consumer's request. If a set of constraints have been specified in the SOAP header of the request, then the Access Controller makes a call to the Policy Writer. The Policy Writer encodes the consumer's constraints as an XACML policy and stores it in the Policy Store. The Policy Store returns a unique identifier for the newly created policy. In addition the Access Controller needs the set of Host Provider policy identifiers that match the candidate set passed by the Deployment Manager. This set of named policies will be used by the Policy Enforcement Point. The set is determined by the Policy Store from the set of candidate Host Providers passed to the Security Manager. The Access Controller must now encode an XACML context request to send to the Policy Enforcement Point.

Figure 4.14-2 The Access Controller calls the Request Writer. It passes the consumer's request from which the Request Writer determines the target of the access request. The requested target is encoded as an XACML context request suitable for the Policy Enforcement Point.

Figure 4.14-3 The call to the Policy Enforcement Point is made with two pieces of information: the targeted access request and the set of relevant polices. The relevant policies are the identifiers for the consumer's, the Web Service Provider's and the set of Host Provider policies.

Figure 4.14-4 The Policy Enforcement Point requests access from the Policy Decision Point and specifies a set of named policies.

Figure 4.14-5 The Policy Decision Point uses the request to gather the relevant XACML policies to determine the access control decision.

Figure 4.14.6 A response from the Policy Decision Point is encoded as an XACML context response. The response contains the decision and any obligations which must be adhered to. Given that the purpose of the access request is to determine a set of authorised Host Providers, it is appropriate to use the *obligations* mechanism provided by the XACML system, to determine the set of Host Providers upon which the access control decision stands.

Figure 4.14.7 The Access Controller receives a set of Host Providers. This set will satisfy the constraints of all three parties involved in the invocation.

4.4.2.3 Integration

The current Web Service Provider service-components determine a set of Host Providers upon which a service may be deployed or invoked (see section 3.3.4.2). The integration of the Security Manager into the Web Service Provider must therefore occur at a point between determining the set of candidate Host Providers and the deployment or invocation request. The point of integration is shown below in figure 4.15.

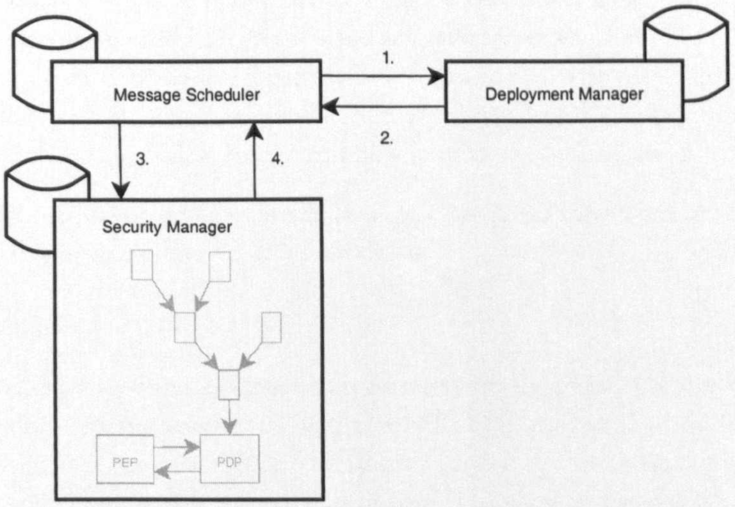


Figure 4.15: Security Manager integration with DynaSOAr components.

Figure 4.15-1 The Message Scheduler requests a set of Host Providers from the Deployment Manager that satisfy the requirements of the Web Service being requested.

Figure 4.15-2 A candidate set of Host Providers is returned.

Figure 4.15-3 The Message Scheduler requests a set of *authorised* Host Providers from the Security Manager. The Message Scheduler passes on the candidate set returned from the Deployment Manager to the Security Manager (from which the authorised set will be determined).

Figure 4.15-4 The response from the Security Manager contains an authorised set of Host Providers as subset of the candidate set passed. If this set is empty (i.e. access control has failed) then the Web Service Provider has failed to find a suitable Host Provider that satisfies all the constraints of the invocation. This fault is handled as described in subsection 3.2.4 Handling Failure.

4.5 Summary

This chapter has described a tripartite security system capable of enforcing the constraints of three parties in a dynamic Web Service deployment scenario. An example was used to illustrate the natural language constraints that the system would need to enforce, such constraints being typical of those encountered in grid scenarios common to many application domains. These natural language constraints were then modeled using simple sets to show how *rules* and *policies* could be defined, combined, and enforced, using set combinations and Boolean conditions, in order to reach a decision-outcome for a given request. The XACML Policy Language and infrastructure was described and shown to be a suitable candidate for implementing the model described. A wider set of components was then defined which enable the tripartite security model to be integrated into the wider DynaSOAr infrastructure to enforce the constraints of the three parties involved in any one DynaSOAr invocation.

This thesis now goes on to describe the evaluation of both the DynaSOAr infrastructure and the security model presented above.

Chapter 5

Evaluation

This chapter presents the evaluation of the research presented in chapters 3 and 4. The purpose of the evaluation is to challenge the claims made in the introductory chapter to this thesis and test their validity. DynaSOAr is evaluated using a prototype implementation of the architecture presented. Scenarios are used to drive the experiments presented. Quantitative results are presented and discussed. The security model and architecture presented in chapter 4 are evaluated using a limited implementation of the key components. Test scenarios are presented with qualitative results discussed to show correct operation of the system.

Section 5.1 discusses the implementation of DynaSOAr; section 5.2 presents the evaluation of DynaSOAr; section 5.3 presents the evaluation of the tripartite security model.

5.1 Implementation

DynaSOAr has been through a number of implementation reviews and releases since its first inception and the work presented in this evaluation uses a number of these implementations. These additional versions were produced by a number of researchers as part of their own research. The following list details the different implementation releases and the major differences between them. An indication of which version was used for each section of the evaluation presented in this chapter is given at the head of each section.

version 1.0 Produced by the author in collaboration with DynaSOAr Project members: Charles Kubicek, Arijit Mukherjee, John Colquoun, and Savas Parastididis, this first version of DynaSOAr leveraged work from the GridShed project. It used the Condor scheduling system and Class Ads mechanism to schedule jobs onto the underlying hardware upon which the Host Provider service is running. These jobs triggered the deployed Web Service implementation. This early prototype was the first to implement the DynaSOAr Web Service Provider, Host Provider and Code Store. It was an implementation of the *pull* model discussed in section 3.3.3.4. The work published in [34, 3] used this implementation.

version 2.0 This upgrade to the DynaSOAr implementation was written to support work on Distributed Query Processing. Version 2.0 implemented a full Web Service approach with no use of the GridShed libraries. It incorporated the Grimoires UDDI directory for the Code Store, and used the *push* model discussed in section 3.3.3.4. The work published in [81] used this version of the implementation.

version 2.2 This further upgrade incorporated support for the dynamic deployment of Virtual Machines.

The architectural principles presented in this thesis are not exclusive to Web Services. Deploying a service may just as well refer to a Web Service as to a Virtual Machine; a virtual machine carries the Web Service in the same way as a *.war* file deployed into a Web Service container carries a Web Service. The work published in [52] used this version of the implementation.

custom version This version was produced by Dr Jim Smith in the School of Computing Science at Newcastle in support of work on adaptive work flow deployment. It involved only a few minor changes to version 2.0, and permitted concurrent deployments to be made in response to multiple requests for a published service.

5.2 DynaSOAr Evaluation

The evaluation of DynaSOAr shows the benefits of dynamic deployment in support of the claims made in chapter 1 of this thesis.

5.2.1 Evaluation Platform

To evaluate DynaSOAr various hardware resources were used.

localhost Laptop computer with Pentium III 1.19 GHz single processor, 512 MBytes RAM, running Windows XP Professional, version 2002 with Service Pack 2.

paston.ncl.ac.uk Desktop computer with Pentium III 1.00 GHz single processor, 512 MBytes RAM, running Windows XP Professional, version 2002 with Service Pack 2.

giga cluster.ncl.ac.uk 25-node cluster, each node has four Intel Xeon 3.06 GHz processors, 1 GByte RAM, running Fedora Core 4 Linux.

networking *localhost* had a domestic ADSL connection to the Internet. At the time of the experiments the recorded connection speed was: 256Kbps download and 2Mbps upload; the difference in download compared to upload is a factor of 8. *paston* was connected to the Internet via an internal 100Mbps connection with a balanced upload and download speed. Between *paston* and the *giga cluster* was an internal 100Mbps connection.

The following subsections detail the evaluation experiments carried out. Each experiment has a description, the expected results, the results achieved, and a discussion.

5.2.2 Data Locality Experiments.

The purpose of this set of experiments was to investigate the benefits of deploying a Web Service close to the data upon which it operates. This deployment scenario is discussed in section 4.1.1, where a researcher requests deployment of a Web Service to an Active Information Repository close to the required data-source. The rationale behind the scenario discussed was to reduce the cost of moving large amounts of data over a network (this movement of data would be necessary where data-intensive services are deployed away from the data source). Therefore, this experiment compared the costs of invoking a data-intensive Web Service executing remotely from the data, and the same service deployed close to the data upon which it operates. Version 2.0 of the DynaSOAr implementation was used for this set of experiments.

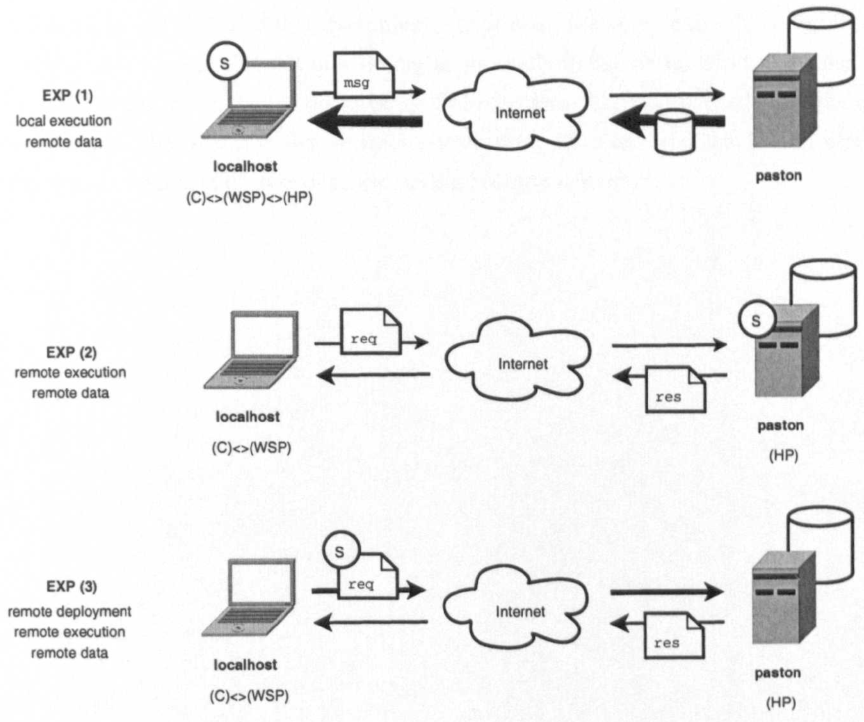


Figure 5.1: Data locality experiments.

Description Three experiments were conducted: (1) an invocation was sent to a local service running on *localhost*, which downloaded data from a remote database for processing (the database was running on *paston*; *localhost* was running from a shed in rural Cambridgeshire, some 200 miles from Newcastle, which housed *paston*). Note the download speed of 2Mbps; (2) a call to the same service deployed locally to the database (on *paston*), and (3) a call to the same service which was first dynamically deployed locally to the database on *paston* (to show the cost of deployment). Note the upload speed of 256Kbps.

A combined Data Analysis Service was used for each experiment. This service wrapped a Data Access Service, which extracted data from the database over the network, and an Entropy Analyzer Service, which performs processing on the data to determine an entropy value for the given data-set. The data size extracted and processed was increased on a logarithmic scale. For each service request, the invocation message stated the number of tuples to process (in the form of an SQL query extracting up to 640,000 tuples). Each tuple was 512 Bytes. The data was extracted from the database as one *chunk* (tuple-set); each chunk was then processed to give a final entropy result. The result was returned as the service response. Each experiment was repeated 10 times for a given chunk size, and an average *round-trip* time recorded in the client application running on *localhost*. A round-trip time is defined as the time taken from service request to service response.

Using a fixed database situated in Newcastle, meant that for experiment (1), the data was downloaded over the network from Newcastle to Cambridgeshire where it was processed by a locally running Host Provider. The results of Experiment (1) were then compared with those of experiment (2), in which a Host Provider was running in Newcastle on *paston*. Experiment (2) was designed to show the potential reduction in download time by not having to download the data to the locally running Host Provider on *localhost*.

Expected Outcome It was expected that dynamically deploying the service to a Host Provider located close to the database would result in a saving being made in the round-trip time when compared with a service deployed on a local host remote from the data. The saving made is in the time taken to move the data chunk across the network between the database and the hosted service. One question was, at what size of data does the saving become apparent?

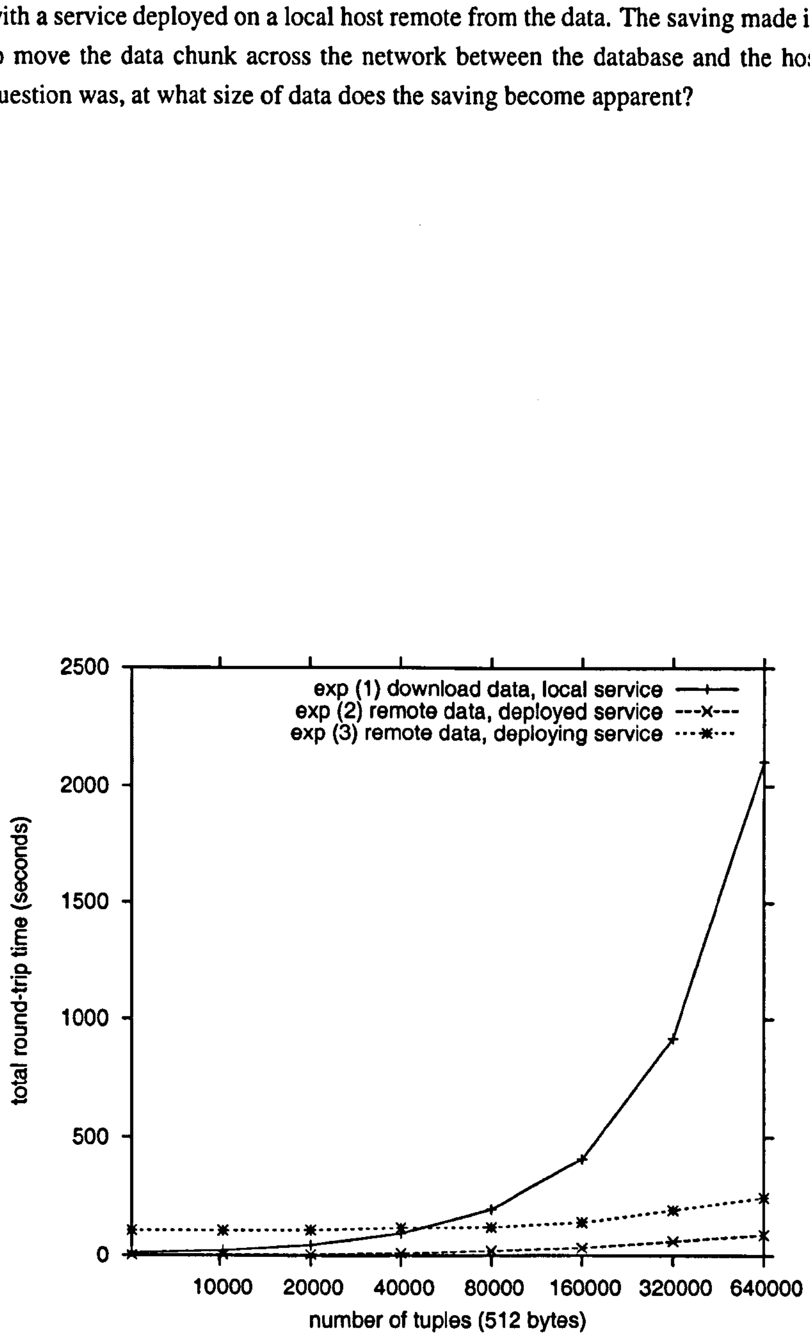


Figure 5.2: Round-trip time for data-intensive Web Service deployments.

Discussion It can be seen from figure 5.2 that a *significant* saving in the round-trip time can be made when deploying a data intensive Web Service close to the data upon which it operates. This is shown by the difference between the results of experiment (1) and experiment (2). The speedup in round-trip time for processing 640,000 tuples is in the order of 21 for a dynamically deployed service, and 8.4 for a service deployed and then invoked, when compared with a remotely operating service. By deploying the service close to the source of the data-set, a reduction in the time taken to transfer the data over the network is made. This saving becomes greater the larger the data-set being processed. The point at which this saving becomes apparent is when the data-set being transfered over the network is larger than the size (in bytes) of the service being deployed (transfered). Therefore, the point at which the transfer rate of a service or data-set are balanced, is the point where the saving becomes apparent.

In this experiment, the Data Analysis Service is 2.4MB in size, this corresponds to 4848 tuples at 512 bytes each. However, given the difference in upload and download bit-rates for the *local-host* Internet connection discussed above in section 5.2.1, the results from experiment (3) must account for the slower upload time (in the order of 1/8 download time). Therefore, the intercept shown in figure 5.2 shows a resultant higher round-trip time when the Data Analysis Service is uploaded to the remote Host Provider. This higher rate forms part of the deployment cost. In a balanced network with equal upload and download rates, it is expected that the intercept point would be at 4848 tuples for this experiment, to match the size of the service deployed.

One of the benefits of the DynaSOAr approach to dynamic service-provisioning is the one-to-many invocation semantics discussed in section 3.2.2, and reviewed in chapter 2. It is clear, therefore, that it may also be beneficial to deploy a service below the intercept point, if it is anticipated that multiple requests for the service will be made. This will, in effect, distribute the deployment costs between more than one request. This is in direct contrast to the job-scheduling, one-to-one model, in which the full cost of deployment is incurred by each individual consumer request. The cost of deployment is illustrated in figure 5.2 as the difference between the results of experiments (3) and (2); this difference represents the cost of deploying the Data Analysis Service used in these experiments and is in the order of 102 seconds.

5.2.3 Parallel Deployment Experiments

The purpose of these experiments was to support the claim that a parallel deployment of a computationally intensive Web Service to multiple Host Providers, would provide an improved quality of service for the calling consumer. This deployment being intended to support concurrent requests for the same Web Service. This scenario is designed to show a more efficient use of the available resources for an increased request load.

This experiment was in support of computationally intensive services as opposed to data-intensive services. Therefore, a version of the Data Analysis Service was written which extracts only a small amount of data and performs a large amount of processing upon it. (The split between data-extraction and data-processing time is in the order of 1:1000).

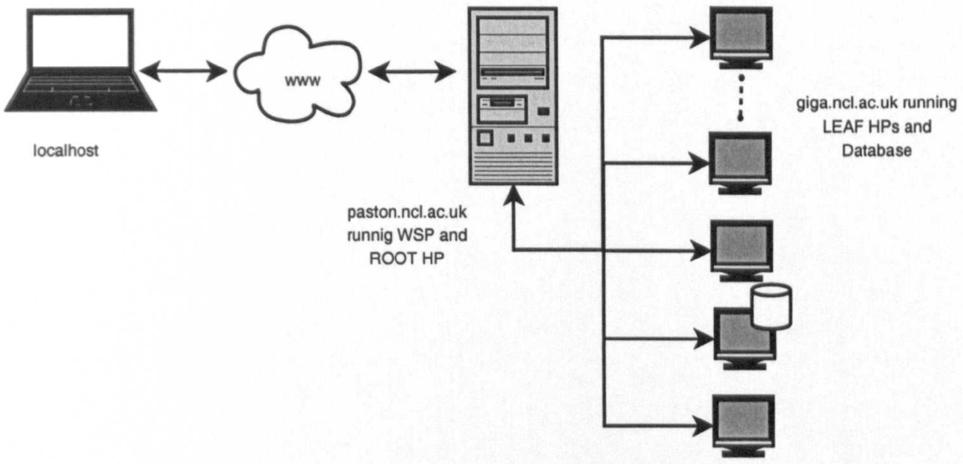


Figure 5.3: Hardware setup for parallel deployment.

Figure 5.3 illustrates the use of ROOT and LEAF Host Providers. A ROOT Host Provider registers with the Web Service Provider and receives deployment/process message-requests. A LEAF Host Provider registers with a ROOT Host Provider and accepts deployment/process message instructions from the ROOT. This mechanism allows the ROOT provider to farm-out the work-load.

Description The combined round-trip time to satisfy 50 concurrent requests was recorded in the client application running on *localhost*. A fixed data-set of one tuple (128 bytes) was processed 10,000,000 times during each service request. Each experiment was repeated 10 times and an average combined round-trip time was recorded in the client application for processing all 50 requests. The number of Host Providers supporting the requested service was increased (1, 3, 5, 7, 9) for each set. The setup for the experiment is illustrated in figure 5.3. The Web Service Provider was located on *paston* with a ROOT Host Provider also running on *paston*; the ROOT Host Provider was registered with the Web Service Provider. Running on the nodes of the *giga cluster*, LEAF Host Providers register with the ROOT Host Provider on *paston*, this created a tree-structure to the DynaSOAr configuration. Requests forwarded to the ROOT Host Provider, were distributed to the LEAF Host Providers. At this stage distribution was effected on a random basis. The custom version of the DynaSOAr implementation was used.

Expected Outcome It was expected that by increasing the number of supporting Host Providers and distributing the concurrent consumer requests between them in parallel, a speed-up in overall round-trip time to satisfy all requests would be achieved. The higher the number of supporting LEAF Host Providers, the greater the expected speed-up.

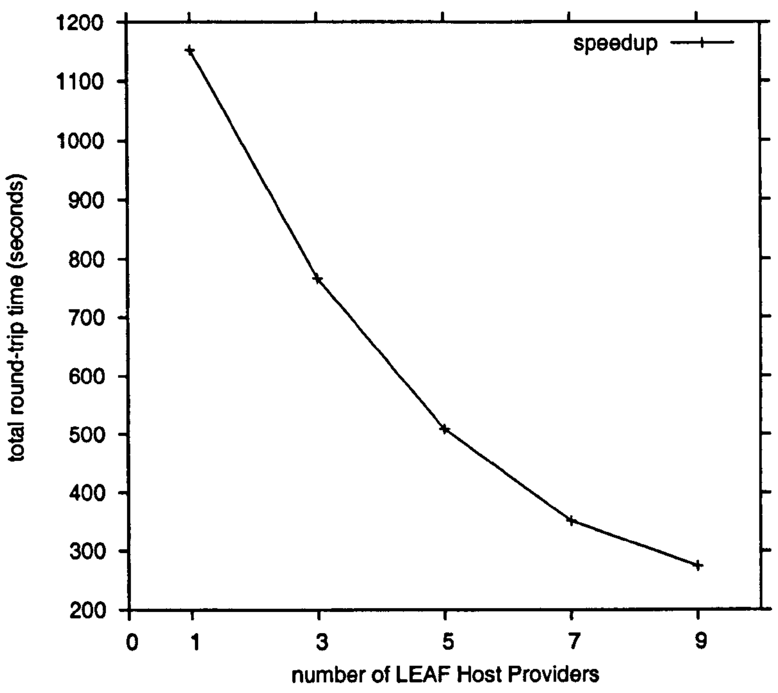


Figure 5.4: Parallel deployment in support of concurrent requests.

Discussion The results plotted in figure 5.4 show a speed-up of up to 4.11 when nine LEAF Host Providers are used to process the 50 concurrent requests. There are a number of factors that influence the result of this experiment, for example, the access time to retrieve the data from the database. By using a small amount of data, the majority of the execution time is spent processing by the individual LEAF Host Providers rather than accessing the database; this situation makes for a better result and is a more representative use of parallelism.

The choice of which deployment strategy to adopt for a given service deployment clearly depends upon the nature of the service - whether it is computationally intensive, or data intensive. The evaluation above shows the clear benefit of adopting the DynaSOAr approach in both of these scenarios. A more detailed evaluation, with the aim of characterising deployment strategies, would result in better information to aid deployment decisions.

5.2.4 Further Cost-factors

The discussion above refers to the evaluation of DynaSOAr usage scenarios (data locality, parallel deployment) and shows a considerable improvement in the round-trip time of service invocations in both cases. There are a number of further cost-factors that will influence the round-trip time. Although these are not investigated here, a brief discussion is given below.

5.2.4.1 Cost of Deployment

The results from the experiment in section 5.2.2 above show the cost of deployment (see figure 5.2). This cost, for a given network, is proportional to the size (in bytes) of the service being deployed. Therefore, a larger service will take longer to deploy. This cost may become far more critical if the unit of deployment were, for example, a virtual machine at 5GBytes rather than a Web Service implementation of 2.5MBytes.

However, in a DynaSOAr system the cost of deployment is spread between the total number of consumer invocations for that particular service deployment. This is due to DynaSOAr being able to exploit the benefits of one-to-many Web Service interaction semantics. As such, the cost of deploying even a very large service may be recouped. A greater understanding of this cost characteristic and its contributing factors would enable better deployment choices to be made for a given network topology and a given number of anticipated invocations. (This is beyond the scope of this thesis, it is, however, discussed as part of further work presented in chapter 6).

5.2.4.2 Service Performance

A number of factors influence the performance of a particular service. Different algorithms designed to achieve the same goals may perform in different ways owing to efficiencies in their coding, or their dependency on additional libraries. Characterisation of these influences will again provide a means for better deployment choices.

5.2.4.3 Hardware Specification

The heterogeneous nature of the resources upon which a service may be deployed will have an influence on the performance of the service. Characterisation of Web Service performance on hardware of a given specification, may provide useful data for predicting the performance of future Web Service deployments.

5.2.5 External Evaluation of Dynasoar

It is fortunate that the results of this research have been incorporated into other research projects. In this section a summary is presented of the lessons learned from these evaluations.

Distributed Query Processing (DQP) uses a service-oriented abstraction to combine data access and data processing services to form declarative statements for long running queries. These queries run over potentially distributed data sources.

> Because the services can be potentially located on computational resources distributed across the Internet, communication costs can play a major role in the performance of the system. Co-locating the query evaluation service and the analysis service with the data, even with an on-the-fly deployment may prove to be potentially beneficial in the long run, especially when frequent, long-running queries are executed [81].

As major contributors to the implementation of DynaSOAr, the researchers investigating Distributed Query Processing have incorporated the DynaSOAr framework into their DQP research.

> The results clearly show that for the queries using analysis services over the data retrieved from the databases, the performance of DynaDQP (the combination of DynaSOAr and DQP) is much better than the standard OGSA-DQP where the analysis service can be remote from the data [81].

The standard OGSA-DQP referred to uses static services which require the transit of data for processing between the data-access service nodes and the data-processing service nodes. The incorporation of DynaSOAr has allowed the co-location of analysis services with the data sources by dynamically deploying them at run-time, thereby dramatically reducing the cost of moving data between nodes.

These findings further confirm the validity of the initial hypothesis of this thesis.

5.3 Evaluation of Tripartite Security Model

In chapter 4 a tripartite security model and component architecture was presented to enforce the combined constraints of the three parties involved in a DynaSOAr invocation. This section illustrates two scenarios which embody the aspects covered by the security model presented. These illustrations will be restricted to access control scenarios which refer to: the subject, resource, action, and environment.

The two scenarios are: an execution transparent interaction that hides the execution behind the published service endpoint; and, a visible execution interaction, where the consumer constrains the hosting of the service execution in some way (i.e. the consumer has knowledge of a separate Host Provider).

5.3.1 Scenario One: Execution Transparent Interaction

A researcher aims to model rainfall patterns in a particular location. As part of the work flow a climate data service is required from which to retrieve rainfall data based on the geo-reference given. At this stage the researcher places no constraint on this service call in terms of processing location, processing time, cost, or other factors - being concerned only with the result.

Given that no constraint is made by the service *consumer*, this scenario illustrates an execution-transparent DynaSOAr interaction from the perspective of the consumer. Any constraints handled by the security model in this case are those imposed by the other two actors: the Web Service Provider and the Host Provider.

5.3.1.1 Model Enactment

The consumer's request is used to build an XACML access control request. This request is challenged by the security architecture against the Invocation Policy. The Invocation Policy is a logical collection of the XACML rules encoding each set of constraints.

A representation of the constraints, and the invocation outcome, are shown using the three tables below. A discussion of the implementation of the function requirements used in the policy evaluation is also shown.

Firstly, table 5.1 shows a set of natural language constraints. These are shown against the constraining relationship and followed by a listing of the XACML rules for each. For the sake of focus, only the XACML rule *conditions* are shown, as it is these that are evaluated. The assumption is made that the target-matching mechanism has identified the correct rules consistent with the original request-message details.

Recall that the XACML specification allows for extensions to the functions, attribute designation, and combining algorithms that enforce the policy decisions. These extensions allow for detailed application-specific functions to be written and enforced. The functions to control the day-access windows and credit assessment stipulated in the constraints below, are invoked through the XACML enforcement mechanisms. Also shown in table 5.3 is the Policy Outcome for each policy. The outcome of the policy is determined by applying the outcome of each Rule to the Rule Combining Algorithm. The Deployment Policy Outcome is determined by applying the Policy Outcome for each constituent policy to the Policy Combining Algorithm. Recall that the combination of the Web Service Provider Policy and the Host Provider Policy is the Deployment Policy (see section 4.3.2.3).

Consumer Constraints	
C > WSP	no constraints
C > HP	no constraints
Web Service Provider Constraints	
WSP > C	no constraints
WSP > HP^n	The HP is listed as a credit worthy provider.
Host Provider[1] Constraints	
HP^1 > WSP	Only allowed access between 01:00 and 07:00 hrs.
HP^1 > C	no constraint
Host Provider[2] Constraints	
HP^2 > WSP	Only allowed access Monday to Friday inclusive.
HP^2 > C	no constraint
Host Provider[3] Constraints	
HP^3 > WSP	Only allowed access at weekends.
HP^3 > C	no constraint

Table 5.1: Natural Language Constraints.

WSP > HP^n Rule 1 *The HP is listed as a credit worthy provider.* This involves a custom function which might use an on-line assertion service that asserts the credit worthiness of one party to another based on mutually agreed terms. A call to this function via an XACML policy decision point requires a subject identifier to be passed.

XACML Code Listing for this Rule:

```
<Rule effect=Permit RuleId=trusted_host_provider>
  <Condition FunctionId="function:dynasoar:credit-check">
    <Apply FunctionId="string-one-and-only">
      <SubjectAttributeDesignator
       AttributeId="credit-worthy"/>
    </Apply>
    <AttributeValue>true<AttributeValue>
    <Apply FunctionId="string-one-and-only">
      <SubjectAttributeDesignator
       AttributeId="subject-id"/>
    </Apply>
    <AttributeValue>host provider<AttributeValue>
  </Condition>
</Rule>
```

HP[1] > WSP Rule 1 *Only allowed access between 01:00 and 08:00 hrs.* This function is possible through pre-defined functions within the XACML implementation used. It involves a time matching function which defines the start and end times of the access window in question.

XACML Code listing for this Rule:

```
<Rule effect=Permit>
  ...
  <Condition FunctionId="xacml:function:AND">
    <Apply
      FunctionId="xacml:function:time-greater-than-or-equal">
      <Apply FunctionId="xacml:function:time-one-and-only">
        <EnvironmentAttributeDesignator
         AttributeId="current-time"/>
      </Apply>
      <AttributeValue>01:00:00</AttributeValue>
    </Apply>
    <Apply
     FunctionId="xacml:function:time-less-than-or-equal">
      <Apply FunctionId="xacml:function:time-one-and-only">
        <EnvironmentAttributeDesignator
         AttributeId="current-time" />
      </Apply>
      <AttributeValue>08:00:00</AttributeValue>
    </Apply>
  </Condition>
  ...
</Rule>
```

HP[2] > WSP Rule 1 *Only allowed access Monday to Friday inclusive.* This function is similar to the above time-based function except it uses days of the week with the start and end day defined for the allowed window.

XACML Code listing for this Rule:

```
<Rule effect=Permit>
  ...
  <Condition FunctionId="xacml:function:AND">
    <Apply
     FunctionId="function:dynasoar:day-greater-than-or-equal">
      <Apply
        FunctionId="xacml:function:time-one-and-only">
        <EnvironmentAttributeDesignator
         AttributeId="today"/>
      </Apply>
```

```
          <AttributeValue>Monday</AttributeValue>
        </Apply>
        <Apply
         FunctionId="function:dynasoar:day-less-than-or-equal">
          <Apply FunctionId="xacml:function:time-one-and-only">
            <EnvironmentAttributeDesignator AttributeId="today"/>
          </Apply>
          <AttributeValue>Friday</AttributeValue>
        </Apply>
      </Condition>
      ...
    </Rule>
```

HP^3 > WSP Rule 1 *Only allowed access at weekends.* As above except that Saturday and Sunday are used as attribute values.

The XACML request generated by the Request Writer has the form detailed in figure 5.5. The rules governing environmental conditions will draw on data from the built-in XACML functions. The policy decision point may use access to the security identity and third-party services to implement the credit-worthy function invoked in Rule 1.

```
<Request>
  <Subject>
    <Attribute AttributeId="subject-id">
      <AttributeValue>
        subject identity
      </AttributeValue>
    </Attribute>
  </Subject>
  <Resource>
    <Attribute AttributeId="resource-id">
      <AttributeValue>
        service-endpoint
      </AttributeValue>
    </Attribute>
  </Resource>
  <Action>
    <Attribute AttributeId="action-id">
      <AttributeValue>
        invoke
      </AttributeValue>
    </Attribute>
  </Action>
</Request>
```

Figure 5.5: General form of XACML request message.

Assume the request shown in figure 5.5 was made at 9.30am on a Wednesday and all Host Providers are known to be credit worthy; table 5.3 shows the outcome of the rules described above. The Rule Combining Algorithms are shown in the second table below.

Web Service Provider Policy	
Rule 1 outcome	PERMIT
Rule Combining Algorithm	deny-overrides
Policy Outcome	PERMIT

Host Provider Policy	
Rule $1^{Host\ Provider\ 1}$ outcome	DENY
Rule $1^{Host\ Provider\ 2}$ outcome	PERMIT
Rule $1^{Host\ Provider\ 3}$ outcome	DENY
Rule Combining Algorithm	permit-overrides
Policy Outcome	PERMIT

Deployment Policy Outcome	
Policy Combining Algorithm	deny-overrides
Policy Outcome	PERMIT

Table 5.3: Policy Outcomes and XACML Rules

The third, table 5.4, shows the overall outcome of combining the Deployment Policy with the Consumer's Policy using the Invocation Policy combining algorithm. This represents the result of the tripartite policy combination and enforcement, and determines if access is granted or denied.

Invocation Policy$^{Consumer+DeploymentPolicy}$	
Policy Combining Algorithm	deny-overrides
Policy Outcome	PERMIT
Overall DECISION	PERMIT = access granted

Table 5.4: Tripartite Policy Outcome.

5.3.2 Scenario Two: Consumer-Constrained Host Selection

A researcher must process a large data set to meet a deadline for publication. DynaSOAr is adopted to allow the dynamic deployment of the processing Web Service close to the specific data source. The data source is available only to registered researchers working in collaboration with the data owner. The Web Service Provider must be a trusted provider of security cleared Web Service implementations to prevent data leakage.

A typical consumer request message structure for this scenario is shown in figure 5.6.

```
<env:Envelope> - SOAP Envelope
  <env:Header> - SOAP Header
    <wsse:Security> - Security Header
      <saml:Assertion> - SAML Assertion
        <saml:AssertionStatement/> SAML Assertion Statement
        <signature/> - signed SAML Assertion
      </saml: Assertion>
    </wsse:Security>
    <dynasoar:Constraints> - Dynasoar Constraints
      <dynasoar:ConsumerPolicy> - set of Consumer Rules
        <xacml:Rule/>
```

```
          <xacml:Rule/>
            ...
        </dynasoar:ConsumerPolicy>
      </dynasoar:Constraints>
    </env:Header>
    <env:Body>
      <Web Service Request/> - Web Service message content
    </env:Body>
  </env:Envelope>
```

Figure 5.6: Consumer request message structure.

The structure shown in figure 5.6 reflects the need for both authentication using SAML Assertions by the Web Service Provider and Host Provider, and the fact that the consumer is setting constraints on the DynaSOAr infrastructure.

5.3.2.1 Scenario Enactment

The invocation is shown using the tables below; firstly, table 5.5 shows the natural language constraints, followed by the associated XACML function and rule descriptions.

Consumer Constraints	
C > WSP	Must deploy service to a specific HP.
Web Service Provider Constraints	
WSP > C	Is a member of 'e-science' group
WSP > HP	The HP is listed as a secure provider.
Host Provider Constraints	
HP > WSP	Only security cleared Web Services.
HP > WSP	Minimum $300 available balance.
HP > C	Must be a trusted Consumer.

Table 5.5: Natural Language Constraints

C > WSP Rule 1 *Must deploy service to a specific HP.* This is an example of an obligation imposed upon the Web Service Provider. Recall that the XACML language allows for the policy decision point to grant access and impose an *obligation* upon the policy enforcement point. This mechanism is used to impose conditional access to the resource (i.e. access is granted *if and only if* a certain obligation is carried out - in this case the Web Service is deployed to a specific Host Provider). The Web Service Provider must ensure that the XACML request is encoded to define the specific Host Provider resource. This is possible as each service deployment is uniquely identified using a URI endpoint address that refers to both service and host/port.

The obligation imposed on the Web Service Provider is encoded using XACML. To ensure it is adhered to a default Rule is written into the policy by the policy writer with a Target set to cover all requests. No condition is applied in the rule. The obligation will be returned to the policy enforcement point if the policy outcome is *permit*. The *FulfillOn=true* ensures that the obligation must be met before access is granted. This information can be sent back to the Message Scheduler to take action.

```
<Obligation name="deploy-host-specified">
  <AttributeAssignment name="specific-host-provider">
    <AttributeValue>hosting inc</AttributeValue>
  </Attribute>
  <AttributeAssignment name="FulfillOn">
    <AttributeValue>true</AttributeValue>
  </Attribute>
</Obligation>
```

WSP > C Rule 1 *Is a member of 'e-science' group.* This function is required to check that the requesting consumer is a member of the specified group. It requires a valid consumer-identity assertion.

XACML Code listing for this Rule:

```
<Rule effect=permit>
  <Condition FunctionId="xacml:function:string-equal">
    <Apply FunctionId="xacml:function:string-one-and-only">
      <SubjectAttributeDesignator
       AttributeId="research-group"/>
    </Apply>
    <AttributeValue>
      escience
    </AttributeValue>
  </Condition>
</Rule>
```

WSP > HP Rule 2 *The HP is listed as a secure provider.* Similar to the above example, this function will check that the Host Provider hosting the requested Web Service, or potential deployment, is a *secure* provider. The definition of secure is at the discretion of the Web Service Provider, and the function's implementation will reflect this.

XACML Code listing for this Rule:

```
<Rule effect=permit>
  <Condition
   FunctionId="function:dynasoar:security-check">
    <Apply FunctionId="function:string-one-and-only">
      <ResourceAttributeDesignator
       AttributeId="security-status"/>
    </Apply>
    <AttributeValue>
       secure
    </AttributeValue>
  </Condition>
</Rule>
```

HP > WSP Rule 1 *Only security cleared Web Services.* This function must check that the Web Service being requested has been cleared by the Host Provider, possibly because of the sensitive nature of the processing or of the data upon which it operates.

XACML Code listing for this Rule:

```
<Rule effect=permit>
  <Condition FunctionId="function:security-check">
    <Apply FunctionId="xacml:function:string-one-and-only">
      <ResourceAttributeDesignator
       AttributeId="security-clearance"/>
    </Apply>
    <AttributeValue>
      cleared
    </AttributeValue>
  </Condition>
</Rule>
```

HP > WSP Rule 2 *Minimum $300 available balance.* Perhaps by using an internal account management service, this function will check for a minimum available balance in the requester's account. In this case the Host Provider needs confirmation that the Web Service Provider has funds sufficient to pay for the requested Web Service deployment/invocation. This rule may have been triggered by a previous history of poor payment.

XACML Code listing for this Rule:

```
<Rule effect=Permit>
  <Condition FunctionId="function:dynasoar:credit-check">
    <Apply FunctionId="string-one-and-only">
      <AttributeDesignator
       AttributeId="minimum-credit-balance"/>
    </Apply>
    <AttributeValue>300<AttributeValue>
    <Apply FunctionId="string-one-and-only">
      <AttributeDesignator AttributeId="subject-id"/>
    </Apply>
    <AttributeValue>web service provider<AttributeValue>
  </Condition>
</Rule>
```

HP > C Rule 3 *Must be a trusted Consumer.* Here the Host Provider must be assured that the original requesting consumer is trusted. Recall that a consumer wanting to specify a particular Host Provider must have registered with the Host Provider prior to making any request. This allows the two parties to agree on an identity assertion mechanism. The SAML protocol (one such mechanism) allows for single sign-on, meaning that the authenticated consumer can be cleared by the Web Service Provider on behalf the Host Provider.

XACML Code listing for this Rule:

```
<Rule effect=Permit>
  <Condition FunctionId="function:dynasoar:trusted-access">
    <Apply FunctionId="function:dynasoar:valid-assertion">
```

```
      <SubjectAttributeDesignator AttributeId="role"/>
        <AttributeValue>consumer<AttributeValue>
      <SubjectAttributeDesignator AttributeId="assertion"/>
        <AttributeValue><saml assertion/><AttributeValue>
    </Apply>
  </Condition>
</Rule>
```

An example XACML request for scenario two is shown in figure 5.7. For clarity some of the detailed XACML encoding has been removed.

```
<Request>
  <Subjects>
    <Subject> - Consumer details
      <Attribute AttributeId="subject-category" >
        <AttributeValue>consumer</AttributeValue>
      </Attribute>
      <Attribute AttributeId="subject-id">
        <AttributeValue>Johnny Bacon</AttributeValue>
      </Attribute>
      <Attribute AttributeId="name-format">
        <AttributeValue><saml-assertion/></AttributeValue>
      </Attribute>
      <Attribute AttributeId="role">
        <AttributeValue>researcher</AttributeValue>
      </Attribute>
    </Subject>
    <Subject> - Web Service Provider details
      <Attribute AttributeId="subject-category">
        <AttributeValue>
         web service provider
       </AttributeValue>
      </Attribute>
      <Attribute AttributeId="subject-id">
        <AttributeValue>
          particle physics labs inc.
        </AttributeValue>
      </Attribute>
      <Attribute AttributeId="name-format">
        <AttributeValue>
          <saml-assertion/>
        </AttributeValue>
      </Attribute>
      <Attribute AttributeId="role">
        <AttributeValue>
         web service provider
        </AttributeValue>
      </Attribute>
```

```
        </Subject>
        <Subject>
          <Attribute AttributeId="subject-category">
            <AttributeValue>host provider</AttributeValue>
          </Attribute>
          <Attribute AttributeId="subject-id">
            <AttributeValue>hosting inc.</AttributeValue>
          </Attribute>
          <Attribute AttributeId="name-format">
            <AttributeValue><saml-assertion/></AttributeValue>
          </Attribute>
          <Attribute AttributeId="role">
            <AttributeValue>
             hosting service provider
            </AttributeValue>
          </Attribute>
        </Subject>
      </Subjects>
      <Resource>
        <Attribute AttributeId="resource-id">
          <AttributeValue>
            //hostprovder/service-endpoint
          </AttributeValue>
        </Attribute>
      </Resource>
      <Action>
        <Attribute AttributeId="action-id">
          <AttributeValue>invoke</AttributeValue>
        </Attribute>
      </Action>
      <Environment>
        <!-- Environmental Attributes of Request Context -->
        <Attribute/>
           ...
        <Attribute/>
      </Environment>
    </Request>
```

Figure 5.7: Example of a tripartite XACML Request.

Assuming the request shown in figure 5.7 is made on a Sunday, the Host Provider specified in the original SOAP message is available, and that the other constraints when encoded and challenged equate to *permit*, then the outcome is shown in table 5.7 below. The Rule Combining Algorithms and policy outcomes are shown in the second table below.

Web Service Provider Policy	
Rule 1	PERMIT
Rule Combining Algorithm	deny-overrides
Policy Outcome	PERMIT

Host Provider Policy	
Rule 1	PERMIT
Rule 2	PERMIT
Rule 3	PERMIT
Rule Combining Algorithm	deny-overrides
Policy Outcome	PERMIT

Deployment Policy Outcome	
Policy Combining Algorithm	deny-overrides
Policy Outcome	PERMIT

Table 5.7: Policy Outcomes and XACML Rules.

The third table, table 5.8, shows the overall outcome of combining the Deployment Policy with the Consumer's Policy using the Invocation-Policy-combining algorithm. The overall outcome represents the result of the tripartite policy combination and enforcement, and will determine if access is granted or denied.

Invocation Policy$^{Consumer+DeploymentPolicy}$	
Rule 1	PERMIT
Policy Combining Algorithm	deny-overrides
Policy Outcome	PERMIT
Overall DECISION	PERMIT=access$^{with\ obligations}$

Table 5.8: Tripartite Policy Outcome

5.3.3 Discussion

Two example scenarios have been presented. A full XACML policy listing for each of these is unsuitable for inclusion in this thesis as it runs to many pages of XML with little benefit, however, the key aspects have been represented, showing both *rules* and *requests*. For a given request context, the outcome of the applicable rules has been shown. The rule and policy combinations and their outcomes are then shown to give a final outcome for the three sets of constraints placed upon the DynaSOAr infrastructure.

It is challenging to demonstrate an abstract model using XML interaction - nothing really happens - particularly when the result of the interaction is either access or no-access. Further work to develop a more readily configurable policy framework for DynaSOAr deployment and invocation control is necessary, but is beyond the scope of this thesis.

5.4 Summary

In this chapter a brief discussion of the DynaSOAr implementation was given, before considering the evaluation of the DynaSOAr infrastructure itself (presented in chapter 3), and the tripartite security model presented in chapter 4.

The evaluation of DynaSOAr was performed using two experimental scenarios. Firstly, a set of experiments was described, designed to test the benefits of dynamically deploying data-intensive Web Services close to the data upon which they operate. Second, a set of experiments was described, designed to test the benefits of parallel deployment of a Web Service in response to concurrent requests. Both experiments aimed to confirm that better quality of service, and making more efficient use of the resources available, is indeed possible with dynamic Web Service deployment using DynaSOAr. The results of both experiments showed a significant speed-up in the round-trip time for consumer requests. The experiments also demonstrated how DynaSOAr responded to the issue of scalability, by enabling multiple service to be deployed in response to a higher demand. Adaptability is possible within the architecture by enabling a new service to be deployed to a new host in the event of failure.

Additionally, an evaluation of the tripartite security model was described. This presented two scenarios to illustrate access control constraints within a DynaSOAr invocation. Firstly, an execution transparent deployment was considered; second, a consumer-specified host deployment was considered. In both cases the natural language constraints were shown, followed by descriptions of XACML functions and XACML rule conditions; these encoded the required constraints. The outcomes of the tripartite security constraints were then shown. Finally, the integration of the security model architecture within the wider DynaSOAr infrastructure was described.

This thesis now goes on to discuss the findings of the research, and the opportunities for further work identified during the course of the research presented.

Chapter 6

Discussion and Further Work

This chapter concludes the thesis. It presents a summary and discussion of the main chapters and contributions; some additional thoughts on opportunities for further work are also discussed.

6.1 Summary and Discussion

Any research should be a *stepping-stone* towards new research. However, before moving on, it is necessary to gain a clear understanding of the current position. Therefore, this section presents the work and contributions made by each main chapter. This is done to challenge the claims made in the thesis introduction and show that the claims are valid.

6.1.1 Dynamic Web Service Deployment (Chapter 3)

The principle objective of the research presented in chapter 3 and evaluated in chapter 5, was:

> *To determine if the introduction of mobility into the service-oriented architecture can allow the dynamic deployment of Web Services, leading to a more efficient use of available resources and better performance for Web Service clients.*

To understand this objective better, an analysis of the rationale and convergence of thought underlying the research was presented (section 3.1, Towards Dynamic Web Services). This analysis drew on a number of important concepts and opportunities: the introduction of mobility; loose coupling; and a separation of concerns between service-provision and resource-provision. If the claim that *better performance for Web Service clients* was to be evaluated, then it was important to understand the areas in which these improvements could be made. Therefore, a discussion was presented which assessed the opportunities of dynamic Web Service deployment (section 3.2, Opportunities for DynaSOAr). The main contribution of the chapter was then described: a new Dynamic Service-Oriented Architecture (DynaSOAr) for Web Service provision (section 3.3, Service Deployment in DynaSOAr). Two Web Service interaction scenarios are supported: 1) a Web Service is already deployed, therefore, a request is routed to a running deployment; and 2) the Web Service is not already deployed, therefore, the requested service is dynamically deployed to an available resource before processing the consumer's request. Consumers interact with DynaSOAr in the same standard way as for Web Service interaction, by sending a SOAP request that conforms to the published Web Service interface. All components of the DynaSOAr architecture are described (section 3.3.4, DynaSOAr Services) and implemented as Web Services.

The above briefly describes the work completed, but did it fulfill its objectives?

To understand if DynaSOAr meets the primary objective of the thesis, this objective is now broken down into three parts, and discussed in the context of the evaluation presented in chapter 5:

Firstly Does *the introduction of mobility into the service-oriented architecture allow the dynamic deployment of Web Services?* The simple answer to this is yes. The introduction of a Web Service Provider (section 3.3.4.2), and separate Host Provider (section 3.3.4.3), and the use of remote evaluation as a means of *mobility of code* between these two components, allows the dynamic deployment of Web Services. The system exploits loose coupling and late binding.

Second Does the dynamic deployment of Web Services *lead to a more efficient use of available resources?* The answer again is yes. This assertion is backed-up by the work presented in chapter 5. An experimental scenario was evaluated using parallelism to dynamically deploy multiple services to handle 50 concurrent consumer requests (section 5.2.3). The experiment shows that the greater the number of deployments (onto different Host Providers), the less time is required to process all 50 requests (a speed up of 4.11 was recorded, see figure 5.4). This experiment was in support of computationally expensive Web Services.

Third The final part of the main objective was to consider if the dynamic deployment of Web Services gave *better performance for Web Service clients.* The evaluation in chapter 5, presented three experiments that backed-up this claim (section 5.2.2). The results of these experiments show that by dynamically deploying a data-intensive Web Service close to the data upon which it operates, a considerable saving is made in the round-trip invocation time giving better performance for Web Service clients (see figure 5.2).

It is believed that from the above three conclusions, that the contribution made by the author's research has successfully answered the principal questions posed by the first of the research objectives.

Additional contributions, arising from the work presented in chapters 3 and 5, are that grid application developers can now work with a single abstraction. For escience to become more readily a part of the *tools of science,* then there is a need for higher-level development abstractions. These abstractions must provide dependable, adaptive, and secure access to the underlying grid resources and systems, but they should also be accessible. DynaSOAr has raised the level of abstraction, from job-scheduling systems such as Condor and Globus, to Web Services. DynaSOAr does not preclude job scheduling, as job-scheduling based Web Services, such as GridSAM [82], can now be deployed using DynaSOAr and used to submit jobs via the Web Services abstraction. This simplifies things for developers, who need not now concern themselves with the minutiae of job-scheduling resource management systems. Also, working wholly within the Web Services domain, developers and scientists have access to a vast commercial investment in tools and documentation in support of Web Services and service-orientation.

Further benefits are that DynaSOAr exploits one-to-many Web Service interaction semantics, allowing the cost of deployment to be spread across potentially many consumers. This is in contrast to the job-scheduling approach where each user carries the full cost of deployment, with every job discarded after execution. Also discussed was the use of separate service and host provisioning in DynaSOAr and how these present the opportunity for brokers, and a wider marketplace for both service and resource-provisioning. (This is discussed further in section 6.2.3).

6.1.2 Tripartite Security Model (Chapter 4)

The objective of the research presented in chapter 4 and evaluated in chapter 5, was:

> *To investigate, design, and evaluate a tripartite security model able to satisfy the security constraints of all three parties in a dynamic Web Service deployment system.*

DynaSOAr has extended the usual bipartite Web Service interaction. The introduction of dynamic service deployment between a service provider and a host provider (on behalf of a third party consumer) raises security issues as the interaction is now tripartite, with each party having separate concerns.

The first part of the research presented in chapter 4, and evaluated in chapter 5, investigated what the security issues for a DynaSOAr invocation were from the perspective of the three parties involved (section 4.1, Security Constraints). (This was restricted to access control). Following this discussion an analysis was undertaken to understand the concepts and interactions required to implement a tripartite access control architecture for DynaSOAr (section 4.1.3, Analysis of Natural Language Constraints); a set of requirements was presented (section 4.1.4). Simple sets were used to model the necessary concepts of a system: rules, policy, combining and matching (section 4.2.1, Modelling Constraints using Sets). The eXtensible Access Control Markup Language (XACML) was used to implement the model's design requirements. An explanation of the XACML and its suitability was given in detail (section 4.2.2, XACML). An example was used to show how the combined constraints of all three parties in a DynaSOAr interaction are represented, propagated through the system, combined, and enforced (section 4.3, XACML and DynaSOAr). Finally, a simple component architecture and interface was presented to enable the model to be added to the wider DynaSOAr architectural components (section 4.4, DynaSOAr Security Architecture). Key components of this architecture were implemented to allow for evaluation.

To understand the result of the research presented in chapter 4, its objective was evaluated in chapter 5, section 5.3, by way of two application scenarios (section 5.3.1 Execution Transparent Interaction, and section 5.3.2 Consumer-Constrained Host Selection). Each example gave rise to a series of natural language constraints. Three constraint-sets were produced, one for each party to the interaction. The constraints were then encoded as XACML rules and three policies were developed relating to: consumer, Web Service Provider and Host Provider. The policies were then combined to form policies that govern deployment (if a service can be deployed given the constraints of the Web Service Provider and Host Provider) and invocation (if a service can be invoked given the constraints of all three parties). Empirically, the evaluation presented in chapter 5, section 5.3, shows that the tripartite model investigated and designed in chapter 4 was successful.

6.1.3 Contributions of the Research

The following list summarizes the contribution of the research presented by this thesis:

1. **Dynamic Web Service deployment** through the introduction of code mobility in the form of remote evaluation.

2. **Explicit Separation of concerns** leading to a clear division between service-provision and resource-provision in a wholly service-oriented system, exploiting loose-coupling and late-binding, allowing greater opportunity for improved scheduling and brokered marketplaces.

3. **Design, implementation and evaluation of DynaSOAr,** a Dynamic Service-Oriented Architecture that allows the dynamic deployment of Web Services at run-time in response to consumer requests.

4. **Simpler development abstraction** for grid application developers and scientists to benefit from a wholly Web Service based infrastructure, and to leverage industry tools and investment in the Web Services domain.

5. **Improved quality-of-service** giving faster round-trip invocation times for both computationally-expensive and data-intensive Web Service deployments.

6. **Tripartite security model and architecture for access control** that allows the representation, propagation, unification and enforcement of three separate sets of constraints in a DynaSOAr invocation.

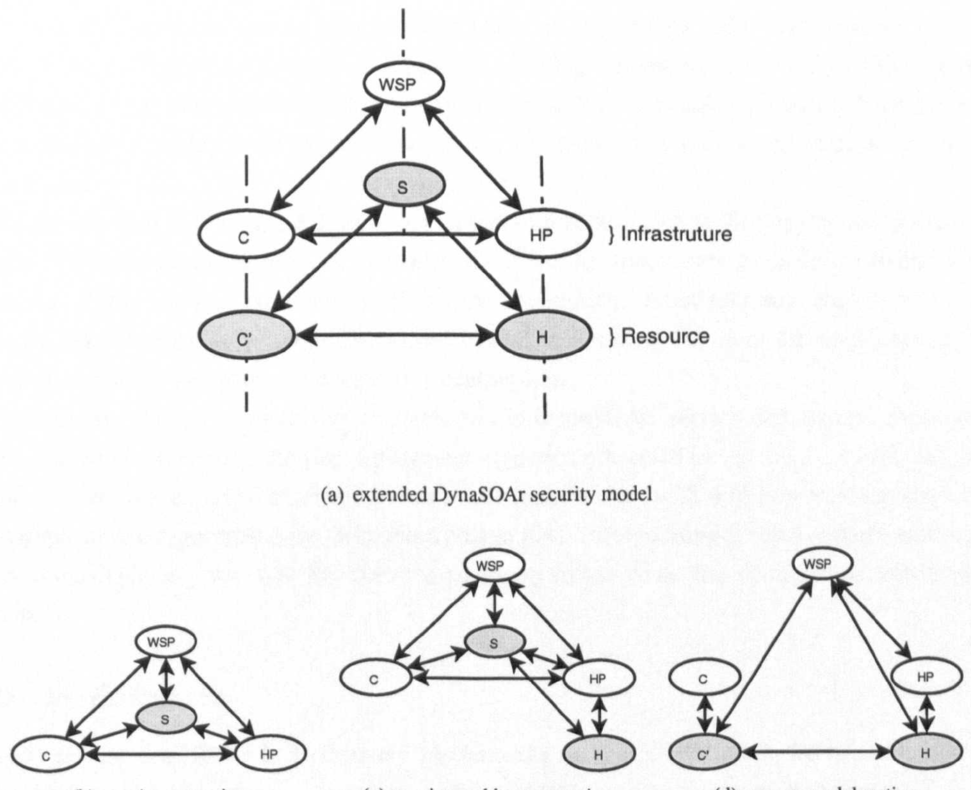
6.2 Opportunities for Further Work

As with any journey the next step requires thought and direction. This section, therefore, presents a number of opportunities for further work that have emerged during the course of the research presented here. Some of the areas are focused on improvements and additional aspects to the research already presented, others discuss the application of DynaSOAr in the wider domain.

6.2.1 Extended Policy Framework

Security is a difficult, complex problem, and often hinders collaboration between systems beyond administrative boundaries. In support of a truly dynamic grid environment, new techniques to specify security requirements are needed. The supporting security standards in Web Services are well documented, yet typically security is left to the end to be *bolted on*, almost as an after thought. Providing simple interfaces for security requirements description for dynamic distributed grid architectures, that provide robustness, instill trust, and auto generate policy, protocols and messages, is in the author's view an essential step in grid security.

Examining an extended security model for DynaSOAr (illustrated in figure 6.1), the key relationships between the three DynaSOAr parties are shown at the *infrastructure* layer. In addition a further *resource* layer can be seen. In this context, the relationships at the infrastructure layer refer to the security contraints placed by the three parties in a DynaSOAr invocation request at the application layer. In DynaSOAr, applications may be deployed as workflows onto virtual networks of resources. Resources may be under the control of potentially different resource administrators. At the resource layer therefore, the relationships are at a lower level, refering to specific machines and invocations instances, giving greater granularity of control over the resource provisioning of a DynaSOAr application deployment.



(a) extended DynaSOAr security model

(b) service extension (c) service and host extension (d) consumer delegation

Figure 6.1: Extended security relationships for DynaSOAr.

The two abstractions target a different level of user. At the resource level, resource administrators and computing experts monitor and control specific resources; at the infrastructure level an application developer specifies higher-level security requirements for the application in question. This clean separation is seen as a benfit to aid modelling of interactions at the two levels.

Extending the security model in chapter 4 in this fashion presents an opportunity to build a detailed security infrastructure for DynaSOAr, with two levels of abstraction. Detailed analysis of typical interactions would be needed to understand the nature and constructs required to specify a model that maps between the infrastructure and resource layers (some useful first steps can be seen in [60, 59]).

The example interactions in figure 6.1 a, b, c, show how these security interactions might be modeled - the arcs representing the security requirement. Modeling these possible interactions at all five layers of the security stack (see section 2.5.1) would provide a comprehensive foundation to securing applications and processing resources. A first step to tackling the delegation illustrated in figure 6.1 (c) is suggested by Wang and Humphrey (see [83]) with extensions to the Security Assertion Markup Language. Domain specific languages might be used to provide an accessible abstraction for the scientists to describe security requirements, with a visual language and interface representation also improving accessibility. These language descriptions would be mapped onto the required security technology, with code generators providing robust implementation and verification.

6.2.2 Service Characterisation and Scheduling

The deployment cost characteristics discussed by Qi et al [50] and reviewed in section 2.4 show a detailed understanding of the time-costs involved in a grid service/container deployment in the HAND system. Although not the motivation for the work presented by this thesis, a fine-grained understanding of the costs in a DynaSOAr deployment cycle is also considered an essential part of a wider dynamic scheduling system. Work published that focused on a usage monitoring framework [79] for DynaSOAr proposed the collection of run-time data to feedback and influence deployment choice in-order to balance the load across a number of deployments for the same service. The techniques used would be useful in gathering run-time data.

The experiments in section 5.2.3 relied upon random scheduling of service deployments and service requests to already deployed services. There is, therefore, an opportunity to explore scheduling algorithms, that might be based on a number of factors: deployment characterisation, current load, quality of service requirements, or security. The loose-coupled late-binding nature of a DynaSOAr invocation gives great flexibility and potential choice of executing host.

There are two layers to scheduling opportunities in DynaSOAr: service deployment, message processing. These correspond to the two deployment scenarios presented in section 3.3.1. Job scheduling heuristics have been explored examining server-switching to cope with differing message arrival rates for multiple service types within the GridSched project [84]. Incorporating predictive performance capabilities to aid deploying new Web Services pre-emptively to handle service demand is therefore an area to explore.

6.2.3 Marketplaces

Dynamic service description and discovery mechanisms such as UDDI allow for run-time discovery and binding to service providers. Providers publish available services in network registries. Using this approach means DynaSOAr is able to support brokers for both service-provision and host-provision. Late binding means that a provider can be found at the time of deployment, or invocation. Figure 6.2 illustrates the potential of the DynaSOAr infrastructure for creating *grid markets*.

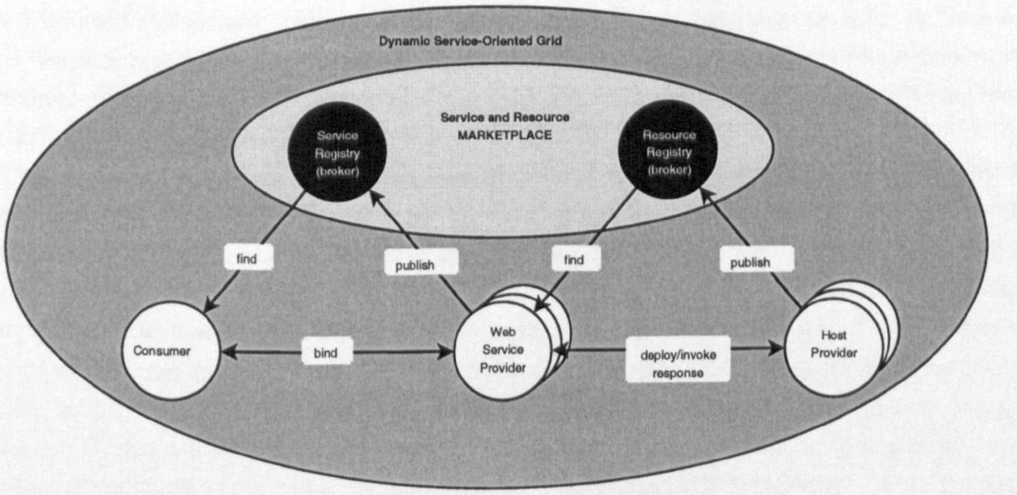

Dynamic Service-Oriented Grid
Service and Resource MARKETPLACE
Service Registry (broker)
Resource Registry (broker)
find
publish
find
publish
Consumer
bind
Web Service Provider
deploy/invoke response
Host Provider

Figure 6.2: Dynamic service-oriented grid marketplace.

The Computational Markets project at Imperial College London (described in [85]) is investigating different economic models in support of an open computational grid environment. A negotiation and payment framework for Web Services has been developed and tested. The negotiation handles fine grained negotiation between service consumers and service providers. Once an agreement is reached on a *price*, then a signed token is issued by the service provider that gives access to the service under the terms of the agreement. Once interaction is complete, payment is made. Client and server-side APIs have been produced to handle negotiation and payment.

It is not clear if these APIs could be used to handle a tripartite negotiation. In the context of DynaSOAr there are six two-way relationships (see section 4.1.2), so it would appear possible that these could be handled by the client-server tools offered. The uncertainty is if the dynamic relationship formed at deployment/invocation time could be resolved.

A combination of negotiation, payment, and the Matchmaking and Brokerage Framework being developed by the KNOOGLE Project [86], together with the dynamism discussed above may provide a starting point to investigate and implement a dynamic service-oriented marketplace for Web Services in DynaSOAr.

6.2.4 Dynamic Virtualisation

The separation of concerns and introduction of dynamic service provisioning allows for dynamic virtualisation. The dynamic deployment of virtual machines using DynaSOAr has been investigated [52] to deploy groups of services encapsulated within a virtual machine. Being able to deploy virtual machines of services close to data, perhaps within existing data centers, would effectively turn a data center into an Active Information Repository [5]. Virtualisation allows grids to be extended dynamically beyond academic domains to include enterprise infrastructure. Extending this work would allow greater investigation of the potential issues. A virtual machine is considerably larger is terms of size when compared to a Web Service deployment (perhaps in the order of 10^3), which will impact on the performance of the system.

6.2.5 Interaction Design

The separation of software from hardware (service from host) and the dynamic deployment of secure, on demand services, facilitated by brokers, governed by computational or economic markets, effectively frees distributed applications to organically adapt and respond to both consumer, provider, and resource usage demands and application quality of service requirements. Separate systems of services would collaborate using well understood languages, agreeing on terms, delivering functionality as a service economy, designed to respond to and satisfy demand.

This is a heady vision; but is it really so far fetched? DynaSOAr represents a small step forward towards this goal. The abstraction layer has been raised and the infrastructure shown to satisfy the demands of computation and data-intensive applications. The infrastructure is able to adapt to run-time conditions. Utility computing systems such as 3tera [87], which allow dynamic drag-and-drop configuration of Web application deployment via a graphical browser interface have begun to develop more accessible, and intuitive interfaces. New, accessible interfaces for grid applications are clearly possible, perhaps with *dashboard* like control over a deployed service application workflow. Areas of Interaction Design and Human Factors research will perhaps provide solutions to communicate better the needs of the scientists the grid is designed to serve. These, together with research from the Semantic Web community, may provide the connections needed to move closer towards a fully dynamic, adaptive, self-referential and accessible grid/Internet.

Bibliography

[1] M. Litzkow, M. Livny, and M. Mutka, "Condor: A Hunter of Idle Workstations," in *In Proceedings of 8th International Conference on Distributed Computing Systems*, 1988.

[2] "Globus Toolkit." available at http://www.globus.org/toolkit/ (last visited April 2007).

[3] P. Watson and C. Fowler, "DynaSOAr: An Architecture for the Dynamic Deployment of Web Services on a Grid or the Internet," in *Proceedings of the UK e-Science All Hands Meeting 2005* (C. S.J. and W. D.W., eds.), Septmeber 2005.

[4] P. Watson, "Databases in Grid Applications: Locality and Distribution," in *Proceedings of the Database: Enterprise, Skills and Innovation* (N. D. Jackson, M. and S. Stirk, eds.), vol. Lecture Notes in Computer Science Volume 3567, pp. 1–16, Springer-Verlag, 2005.

[5] P. Watson and P. Lee, "The NU-Grid Persistent Object Computation Server," in *In Proceedings of the 1st European GRID Forum Workshop, Poznan, Poland (part of the ISTHMUS 2000 Conference)*, (Instytut Informatyki, Politechnika Poznánska, Poznan), pp. 357–364, 12-13 April 2000.

[6] "The OGSA-DAI Project." available at http://www.ogsadai.org.uk/ (last visited April 2007).

[7] "The TAVERNA Project." available at http://taverna.sourceforge.net/ (last visited April 2007).

[8] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, 1998.

[9] G. Cabri, L. Leonardi, and F. Zambonelli, "Weak and Strong Mobility in Mobile Agent Applications," in *In Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java, Manchester, UK*, April 2000.

[10] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing," in *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, (Washington, DC, USA), p. 140, IEEE Computer Society, 1998.

[11] "GT Execution Management: GRAM." available at http://www.globus.org/toolkit/gram (last visited May 2007).

[12] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington, "ICENI: Optimisation of Component Applications within a Grid Environment," *Parallel Computing*, vol. 28, no. 12, pp. 1753–1772, 2002.

[13] "Reference Model for Service Oriented Architecture 1.0, August 2006." available at http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf (last visited May 2007).

[14] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," 2002.

[15] "Web Service Architecture Specification." available at http://www.w3.org/TR/ws-arch/ (last visited April 2007).

[16] P. Maes, "Agents that Reduce Work and Information Overload," *Communications of the ACM*, vol. 37, no. 7, pp. 30–40, 1994.

[17] M. R. Genesereth and S. P. Ketchpel, "Software Agents," *Communications of the ACM*, vol. 37, no. 7, pp. 48–53, 1997.

[18] N. R. Jennings and M. J. Wooldridge, eds., *Agent Technology: Foundations, Applications and Markets*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1998.

[19] D. Milojicic, F. Douglas, and R. Wheeler, eds., *Mobility*. Addison Wesley, 1999.

[20] M. Keidl, S. Seltzsam, and A. Kemper, "Flexible and Reliable Web Service Execution," in *1st Workshop on Entwicklung von Anwendungen auf der Basis der XML Web-Service Technologie*, pp. 17–30, 2002.

[21] M. Keidl, S. Seltzsam, and A. Kemper, "Reliable Web Service Execution and Deployment in Dynamic Environments," in *Technologies for E-Services*, vol. 2819 of *Lecture Notes in Computer Science*, pp. 104–118, Springer Berlin / Heidelberg, 2003.

[22] M. Keidl and A. Kemper, "Towards Context-Aware Adaptable Web Services," in *Proceedings of the 13th International World Wide Web (WWW04)*, (New York, NY, USA), pp. 55–65, ACM Press, 2004.

[23] "Universal Description, Discovery and Integration Specification (UDDI)." available at http://www.uddi.org (last visited April 2007).

[24] L. Chunlin and L. Layuan, "An Agent-Oriented and Service-Oriented Environment for Deploying Dynamic Distributed Systems," *Comput. Stand. Interfaces*, vol. 24, no. 4, pp. 323–336, 2002.

[25] A. Omicini and F. Zambonelli, "Coordination for Internet Application Development," *Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 3, pp. 251–269, 1999.

[26] Y. Yang, O. Rana, C. Georgousopoulos, D. Walker, and R. Williams, "Mobile Agents and the SARA Digital Library," in *Advances in Digital Libraries*, pp. 71–77, 2000.

[27] J. Cao, D. Kerbyson, and G. Nudd, "Performance Evaluation of an Agent-based Resource Management Infrastructure for Grid Computing," in *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid, Brisbane, Australia*, pp. 311–318, IEEE, 2001.

[28] O. Rana and D. Walker, "The Agent Grid: Agent-based resource integration in PSEs," in *Proceedings of the 16 th IMACS World Congress on Scientific Computing, Applied Mathematics and Simulation, Lausanne, Switzerland*, 2000.

[29] K. Bryson, M. Luck, M. Joy, and D. T. Jones, "Applying Agents to Bioinformatics in GeneWeaver," in *Cooperative Information Agents*, pp. 60–71, 2000.

[30] F. Bellifemine, A. Poggi, and G. Rimassa, "Developing Multi-Agent systems with a FIPA Compliant Agent Framework," *Software - Practice And Experience*, vol. 31(2), pp. 103–128, 2001.

[31] B. Burg, "Agents in the World of Active Web-Services," in *Lecture Notes In Computer Science; Revised Papers from the Second Kyoto Workshop on Digital Cities II, Computational and Sociological Approaches*, vol. 2362, (London, UK), pp. 343–356, Springer-Verlag, 2002.

[32] L. Moreau, "On the use of Agents in a Bioinformatics Grid," in *Proceedings of the 3rd IEEE/ACM CCGRID* (S. Lee, S. Sekguchi, S. Matsuoka, and M. Sato, eds.), 2003.

[33] Y. Li, F. Rao, Y. Chen, D. Liu, and T. Li, "Services Ecosystem: Towards a Resilient Infrastructure for On Demand Services Provisioning in Grid," in *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, (Washington, DC, USA), p. 394, IEEE Computer Society, 2004.

[34] P. Watson, C. Fowler, C. Kubicek, A. Mukherjee, J. Colquhoun, M. Hewitt, and S. Parastatidis, "Dynamically Deploying Web Services on a Grid using DynaSOAr," in *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pp. 151–158, IEEE Computer Society Press, April 2006.

[35] I. Foster and C. Kesselman, eds., *The Grid: Blueprint for a New Computing Infrastrcture.* Morgan Kaufman, 1999.

[36] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputer Applications*, vol. 11(2), pp. 115–128, 1997.

[37] M. Atkinson, D. DeRoure, A. Dunlop, G. Fox, P. Henderson, T. Hey, N. Paton, S. Newhouse, S. Parastatidis, A. Trefethen, P. Watson, and J. Webber, "Web Service Grids: An Evolutionary Approach," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 2-4, pp. 377–389, 2005.

[38] "Web Services Interoperability Organisation." available at http://www.ws-i.org/ (last visited April 2007). (last visited May 2007).

[39] "Open Grid Services Infrastructure (OGSI) Version 1.0." available at http://www.globus.org/alliance/publications/papers/Final-OGSI-Specification-V1.0.pdf (last visited May 2007.

[40] S. Tuecke, K. Czajkowski, I. Foster, J. Rey, F. Steve, and G. Carl, "Grid Service Specification," 2002.

[41] S. Parastatidis, J. Webber, P. Watson, and T. Rischbeck, "WS-GAF: A Framework for Building Grid Applications using Web Services," *Concurrency and Computation: Practice & Experience*, vol. 17, no. 2-4, pp. 391–417, 2005.

[42] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke, "From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring and Evolution." available at, http://www.globus.org/wsrf/specs/ogsi-to-wsrf-1.0.pdf.

[43] "WS-RF: Web Services Resource Framework." available at www.globus.org/wsrf/specs/ogsi-to-wsrf-1.0.pdf (last visited April 2007).

[44] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, and W. Vambenepe, "Modeling Stateful Resources with Web Services v. 1.1." availabel at, http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf.

[45] "Web Services Addressing (WS-Addressing) Specification, August 2004." available at http://www.w3.org/Submission/ws-addressing/ (last visited May 2007).

[46] "Web Services Resource Transfer (WS-RT) Specification." available at, http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-rt/ws-rt-spec.pdf (last visited May 2007).

[47] J. Weissman, S. Kim, and D. England, "Supporting the Dynamic Grid Service Lifecycle," in *Cluster Computing and the Grid*, vol. 2, pp. 808–815, IEEE, May 2005.

[48] J. B. Weissman, S. Kim, , and D. England, "A Framework for Dynamic Service Adaptation in the Grid," in *Proceedings of the IPDPS NSF Next Generation Software Workshop*, (Denver, Colorado), April 2005.

[49] M. Smith, T. Friese, and B. 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)*, (Washington, DC, USA), pp. 201–208, IEEE Computer Society, 2004.

[50] L. Qi, H. Jin, I. Foster, and J. Gawor, "HAND: Highly Available Dynamic Deployment Infrastructure for Globus Toolkit 4," in *In Proceedings of 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing*, 2007.

[51] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," in *IFIP International Conference on Network and Parallel Computing*, pp. 2–13, Springer-Verlag LNCS 3779, 2005.

[52] A. Mukherjee and P. Watson, "CS-TR: 1002 Virtual Machines in DynaSOAr: Creating an on-demand ad-hoc Virtual Grid," tech. rep., School of Computing Science, Newcastle University, 2007.

[53] K. Krauter, R. Buyya, and M. Maheswaran, "A taxonomy and survey of grid resource management systems for distributed computing," *Software Practice & Experience*, vol. 32, no. 2, pp. 135–164, 2002.

[54] L. Boloni and D. Marinescu, "An Object-Oriented Framework for Building Collaborative Network Agents," pp. 31–64, 2000.

[55] K. Jun, L. Bni, K. Palacz, and D. C. Marinescu, "Agent-Based Resource Discovery," in *Heterogeneous Computing Workshop*, pp. 43–52, 2000.

[56] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra, "NetSolve: Grid Enabling Scientific Computing Environments," in *Grid Computing: The New Frontier of High Performance Computing (Volume 14)* (L. Grandinetti, ed.), Elsevier, 2005. Also available as CoreGRID TR-0001.

[57] A. YarKhan, K. Seymour, K. Sagi, Z. Shi, and J. Dongarra, "Recent Developments in Gridsolve," *International Journal of High Performance Computing Applications*, vol. 29, pp. 131–141, 2006.

[58] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organization," *International Journal on High Performance Applications*, vol. 15(3), 2001.

[59] M. Humphrey, M. Thompson, and K. Jackson, "Security for Grids," *Proceedings of the IEEE*, vol. 93 (3), pp. 644– 652, 2005.

[60] M. Humphrey and M. Thompson, "Security Implications of Typical Grid Computing Usage Scenarios," *Cluster Computing*, vol. 5, no. 3, pp. 257–264, 2002.

[61] K. Ono and H. Tai, "A Security Scheme for Aglets," *Software Practice & Experience*, vol. 32, no. 6, pp. 497–514, 2002.

[62] G. Karjoth, D. Lange, and M. Oshima, "A Security Model for Aglets," *IEEE Internet Computing*, vol. 1, no. 4, pp. 68–77, 1997.

[63] D. Milojicic, G. Agha, P. Bernadat, D. Chauhan, S. Guday, N. Jamali, D. Lambright, and F. Travostino, "Case Studies in Security and Resource Management for Mobile Object Systems," in *Autonomous Agents and Multi-Agent Systems, vol 2.1, pp 45-79*, 2002.

[64] L. Gong, G. Ellison, and M. Dageforde, *Inside Java 2 Platform Security, Second Edition: Architecture, API Design and Implementation.* Addison-Wesley, 2003.

[65] M. Hondo, N. Nagaratnam, and A. Nadalin, "Securing Web Services," *IBM Systems Journal*, vol. 41(2), pp. 228–241, 2002.

[66] "eXtensible Access Control Markup Language v1.0 Specification." February 2003, available at www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf (last visited May 2007).

[67] "Security Assertion Markup Language (SAML) v1.0." available at www.oasis-open.org/committees/security/docs/cs-sstc-core-01.pdf (last visited May 2007), May 2002.

[68] S. Mysore, "Securing Web Services - Concepts, Standards, and Requirements." October 2003.

[69] T. Taka, T. Mizuno, and T. Wantanabe, "A Model of Mobile Agent Services Enhanced for Resource Restrictions and Security," in *ICPADS '98: Proceedings of the 1998 International Conference on Parallel and Distributed Systems*, (Washington, DC, USA), p. 274, IEEE Computer Society, 1998.

[70] M. Lorch, D. Kafura, and S. Shah, "An XACML-based Policy Management and Authorization Service for Globus Resources," in *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing*, (Washington, DC, USA), p. 208, IEEE Computer Society, 2003.

[71] M. Lorch, D. B. Adams, D. Kafura, M. S. R. Koneni, A. Rathi, and S. Shah, "The PRIMA System for Privilege Management, Authorization and Enforcement in Grid Environments," in *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing*, (Washington, DC, USA), p. 109, IEEE Computer Society, 2003.

[72] M. Lorch and D. G. Kafura, "Supporting Secure Ad-hoc User Collaboration in Grid Environments," in *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, (London, UK), pp. 181–193, Springer-Verlag, 2002.

[73] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," in *International workshop on policies for distributed systems and networks (POLICY 2001), Hewlett-Packard Lab, Bristrol, England*, vol. 1995, pp. 18–39, 2001.

[74] N. Dulay, E. Lupu, M. Sloman, and N. Damianou, "A Policy Deployment Model for the Ponder Language," in *Integrated network management: 2001 IEEE/IFIP integrated management strategies for the new millennium, Seattle*, pp. 529–544, May 2001.

[75] N. Dulay, N. Damianou, E. Lupu, and M. Sloman, "A Policy Language for the Management of Distributed Agents," in *Agent-oriented software engineering, AOSE 2001*, vol. 2222 of *Lecture Notes in Computer Science*, pp. 85–102, 2002.

[76] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke, "Security for Grid Services," in *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, (Washington, DC, USA), p. 48, IEEE Computer Society, 2003.

[77] G. Wasson and M. Humphrey, "Policy and Enforcement in Virtual Organizations," in *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing*, (Washington, DC, USA), p. 125, IEEE Computer Society, 2003.

[78] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah, "First Experiences using XACML for Access Control in Distributed Systems," in *XMLSEC '03: Proceedings of the 2003 ACM workshop on XML security*, (New York, NY, USA), pp. 25–37, ACM Press, 2003.

[79] S. Cavalieri, F. Scibilia, C. Fowler, S. Parastatidis, and P. Watson, "Web Service Usage Monitoring for DynaSOAr," in *Proceedings of the IEEE Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW2006)*, (Gosier, Gaudeloupe), IEEE Computer Society Press, February 2006.

[80] F. H. Adapted by and F. F. from The Oxford Dictionary, *The Concise Oxford Dictionary of Current English.* Oxford University Press, 1946.

[81] A. Mukherjee and P. Watson, "Adding Dynamism to OGSA-DQP: Incorporating the DynaSOAr Framework in Distributed Query Processing," Technical Report Series CS-TR-979, Newcastle University, August 2006.

[82] J. Darlington, W. Lee, and S. McGough, "Performance Evaluation of the GridSAM Job Submission and Monitoring System," in *In UK e-Science All Hands Meeting (Nottingham)*, 2005.

[83] J. Wang, D. D. Vecchio, and M. Humphrey, "Extending the Security Assertion Markup Language to Support Delegation for Web Services and Grid Services," in *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, (Washington, DC, USA), pp. 67–74, IEEE Computer Society, 2005.

[84] J. Palmer and I. Mitrani, "Optimal and Heuristic Policies for Dynamic Server Allocation," *J. Parallel Distrib. Comput.*, vol. 65, no. 10, pp. 1204–1211, 2005.

[85] J. Cohen, J. Darlington, and W. Lee, "Payment and Negotiation for the Next Generation Grid and Web," *Concurrency and Computation: Practice and Experience*, Septmeber 2005.

[86] "Project: KNOOGLE: Matchmaking and Brokerage Framework." available at http://www.omii.ac.uk/downloads/project.jsp?projectid=77 (last visited May 2007).

[87] "3Tera - Utility Computing." available at, http://www.3tera.com/ (last visited May 2007).