

Dynamically Deploying Web Services on a Grid using Dynasoar

Paul Watson, Chris Fowler, Charles Kubicek, Arijit Mukherjee, John Colquhoun, Mark Hewitt,
Savas Parastatidis
School of Computing Science, University of Newcastle-upon-Tyne, UK
paul.watson@ncl.ac.uk

Abstract

Dynasoar is an infrastructure for dynamically deploying Web Services over a Grid or the Internet. It enables an approach to Grid computing in which distributed applications are built around services instead of jobs. Dynasoar automatically deploys a service on an available host if no existing deployments exist, or if performance requirements cannot be met by existing deployments. This is analogous to remote job scheduling, but offers the opportunity for improved performance as the cost of moving and deploying the service can be shared across the processing of many messages. A key feature of the architecture is that it makes a clear separation between Web Service Providers, who offer services to consumers, and Host Providers, who offer computational resources on which services can be deployed, and messages sent to them processed. Separating these two components and defining their interactions, opens up the opportunity for interesting new organisational/ business models.

1. Introduction

Distributed job scheduling systems that can dynamically route client jobs to remote computing resources for execution are now widely available (e.g. Condor [1], Globus [2], SunGridEngine [3]), and found at the heart of most Grid computing infrastructures. The job – a combination of code to be executed and (in most cases) the data on which it is to operate – is created by a client and given to distributed job scheduling systems that aim to route it to a suitable host which has the resources available to execute it.

In recent years, there has been a move towards utilising Web Services to build Grid and other distributed applications [4-6]. An application is represented as a set of services that communicate through the exchange of messages, and there are now widely accepted standards for describing service interfaces (WSDL [7]) and the transfer of those

messages (SOAP [8]). When the computational requirements of such services cannot be met by their hosting environment then the current practice is to create a job and send it to a distributed job scheduling system for execution on a suitable host. The consequence is that application writers must deal with two types of computational entities: services and jobs.

This paper describes work in the Dynasoar project which is exploring an alternative, “service-only” approach that promotes service-oriented application design free of the ‘job’ abstraction. To enable this, if no existing service deployments can meet the computational requirements then a service is dynamically deployed on an available host in order to utilise its computational resources. The key architectural feature is to make a clear separation between *Web Service Providers*, who make services available to consumers by exposing service endpoints, and *Host Providers*, who offer computational resources on which services can be deployed, and messages sent to them processed. These components are supported by *Service Repositories* that hold deployable versions of services, and *Brokers* that decide to which of a set of Host Providers a message should be routed. All these components are themselves realized as loosely-coupled Web Services, so enabling a wide range of deployment options.

This approach has the potential to provide three main advantages over existing approaches that utilise both jobs and services:

- a) it simplifies the development of applications and services by allowing designers and implementers writers to work entirely in a service-oriented framework;
- b) it can improve performance by retaining the deployment of the code (in this case the deployed service) on a host. This allows the deployment cost to be shared over the processing of many messages sent to the service. In contrast, because jobs represent self-contained, often “one-off” executions, job schedulers lack the ability to share

the cost of the movement and installation of code across multiple executions; and

- c) the clear distinction between Service Providers and Host Providers enables new organisational/business models.

The paper describes this investigation into the design and application of a generic infrastructure for the dynamic deployment of Web Services. Section 2 describes the system architecture, Section 3 describes various scenarios for how it could be used, while Sections 4 and 5 explain the prototype implementation and evaluate its behaviour in an example scenario. Section 6 discusses related work before Section 7 draws conclusions.

2. The Dynasoar Architecture

Dynasoar provides a generic infrastructure for the dynamic deployment of Web Services (from now on the terms Service and Web Service will be used interchangeably). This is achieved by dividing the handling of the messages sent to a service between two components – a *Web Service Provider* and a *Host Provider* – and defining an interface through which they interact.

- The *Web Service Provider* accepts incoming SOAP messages sent to an endpoint associated with a particular service and is responsible for arranging for their processing. It does this by choosing a Host Provider and forwarding the SOAP message to it together with any associated Quality of Service (QoS) information (as will be described below).
- The *Host Provider* controls computational resources (e.g. a cluster or a Grid) on which services can be deployed and messages to them processed. It accepts SOAP messages from the Web Service Provider (along with any associated information). If a response is generated after the processing of a message, the Host Provider returns it to the Web Service Provider.

When a message reaches the Host Provider, there are two main classes of interactions, depending on whether or not the service is already deployed on the node on which the message from the consumer is to be processed:

If the service is already deployed on the node on which the SOAP message is to be processed then the Host Provider simply routes the SOAP message to the service. This case is shown in Figure 1; a request for service *s5* is sent by the Consumer to an endpoint at the Web Service Provider which passes it on to a Host Provider. The Host Provider already has the service *s5* deployed (on nodes 1 and 2 in Fig. 1) and so, based on current loading information it chooses to route the request to node 2 for processing. Note that the Web

Service Provider is not aware of the internal structure of the Host Provider - e.g. the nodes on which the service is deployed, nor the node to which the message is sent – this is managed entirely by the Host Provider.

The other case to be considered is where the service is **not** already deployed on the Host Provider (Fig. 2).

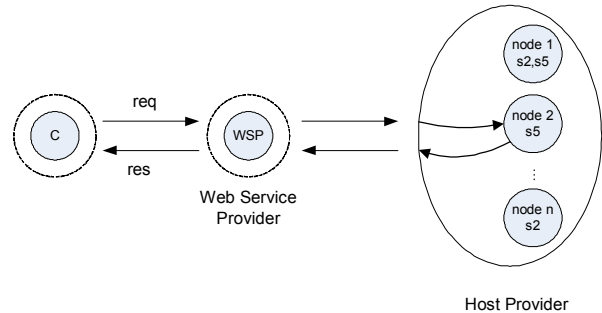


Figure 1. A request is routed to an existing deployment of the service.

To allow for the dynamic deployment of services, there are “Service Repositories” that hold deployable versions of services and associated metadata. The architecture makes no assumptions about the nature of the deployable services except that, once deployed, they act as Web Services, accepting and processing SOAP messages. For example they may be Java services that are deployed by having their “.war” file copied into the appropriate directory of a Tomcat Axis server. The current system also deploys services in Virtual machines (both VMWare and Virtual PC) [9, 10], .Net services and even services that are dynamically deployed as stored procedures in SQLserver. Each service may have multiple entries in the Service Repository, so allowing a single service to be deployed on several different types of hosts (e.g. Windows, Linux, VMWare).

Fig. 2 shows an example of a dynamic service deployment. A request for a service (*s8*) is sent by the Consumer to the endpoint at the Web Service Provider which passes it on to a Host Provider (step 1 in the Figure). The Host Provider does not have the service *s8* deployed on any of the nodes it controls and so, based on current loading information it chooses one node (node 2), fetches the service code from a Service Repository and installs the service on that node (step 2). It then routes the request to it for processing (step 3). Any response is then routed back to the consumer.

Once a service is installed on a node it can remain there ready to process future messages until the Host Provider decides to reclaim it. This has the potential to generate large efficiency gains when compared to job-based scheduling systems in which each job execution requires its own installation.

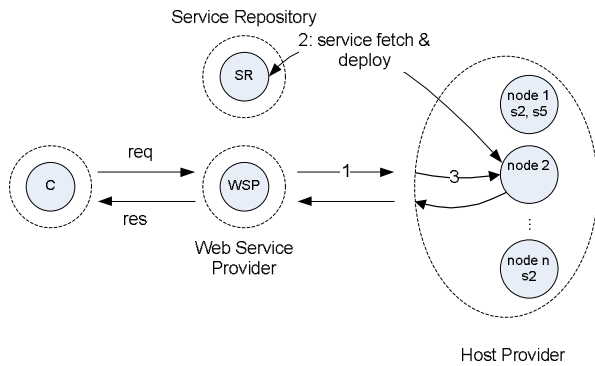


Figure 2. A service is dynamically deployed to process a request

It should be noted that the Consumer is not aware of the fact that a service is dynamically deployed – their interaction with the service is through the standard mechanism of sending a message to the service endpoint offered by the Web Service Provider.

In the above description, services are deployed in the Host Provider in order to meet changing demands, triggered by the arrival of messages from consumers. This does not however preclude pre-emptive service deployment by a Host Provider, for example where changing demands are predictable (e.g. regular peak times for a service).

The Web Service Provider, Host Provider and Service Repository are themselves implemented as Web Services. This opens up a range of deployment options. For example, the Service Repository may be owned by the Web Service Provider, and access restricted to Host providers it trusts, it could be a public repository that is independent of any one provider. It could, for example, allow those working in a particular domain (e.g. bioinformatics) to share deployable versions of services.

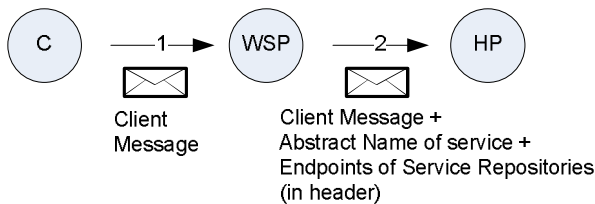


Figure 3. Basic information flow in Dynasoar

The way in which information is conveyed between the services is shown in Figure 3. The Client sends a SOAP message (step 1 in Fig. 3) to the endpoint exposed by the Web Service Provider. There is nothing Dynasoar-specific about this message. The WSP inserts into the header of the message: a) a URI representing the abstract name of the service, b) zero or

more endpoints of Service Repositories from which a deployable version of the service can be obtained. The message is then forwarded to the Host provider (2 in Figure 3). This uses the abstract name of the service to check whether or not a version of that service is already deployed. If a further deployment is required then a request can be sent to the Service Repository for a deployable version that can then be installed. Once this is done then the message can be processed and any result returned to the client. As an optimization, Host Providers may also choose to cache deployable versions of services that they have already installed so as to avoid the need to re-request the service from a Repository. Other techniques could also be used by Host Providers to retrieve copies of a service: for example P2P filesharing techniques may prove effective for transferring large Virtual Machine images.

2.1. Dynamic Requirements' Matching

Figures 1-3 showed only a single Host Provider. However, there may be several Host Providers available to a Web Service Provider, and there could be an advantage in making a selection between them based on criteria such as cost, performance, dependability, security.

These decisions are made by the final component in the Dynasoar architecture - the *Broker*. This has the same interface as the Host Providers, but has knowledge of one or more of them, and so can make a decision on where to route a message based on their current characteristics (Figure 4).

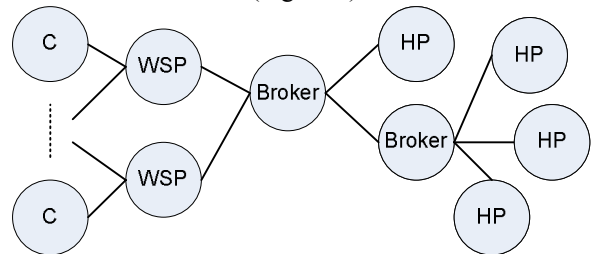


Figure 4. Brokers routing messages to Host Providers

In order to achieve this, the broker must know what requirements it is attempting to meet (best price or performance? dependability etc.). We are therefore experimenting with having the requirements placed in the header of the SOAP message by the Client and/or the Web Service Provider. The broker can then use information provided dynamically by the Host Providers, or third-parties (e.g. independent monitors of service availability) to decide how best to meet those requirements. To date, we have investigated the meeting of security requirements specified by the Consumer, Web Service Provider and Host provider as

XACML assertions. This allows, for example, satisfying the desire of a Web Service provider that services they own should only be deployed on Host providers they trust.

3. Usage Scenarios

The clean separation of the Web Service Provider from the Host Provider enables a number of business/organisational usage models. This section describes three example scenarios.

3.1. Scenario 1: The Dynamic Outsourcing of Web Service Hosting

A bioinformatics company (BioCorp) wishes to earn revenue by writing computational services and charging customers to use them (perhaps by taking a micro-payment for each message processed). However, it does not wish to host the services as this is not its area of expertise. It therefore contracts a Hosting company (Hosting Inc) to host its services (this is analogous to an ISP hosting web sites). To achieve this, BioCorp runs the Web Service Provider component on their system, while Hosting Inc deploy the Host Provider component (Figure 5). When BioCorp produce a new Web Service, they advertise its endpoint to potential consumers and place a deployable version of it in their Service Repository. When Hosting Inc receive the first SOAP message for the service, they dynamically deploy it on one of their nodes as described in the previous Section.

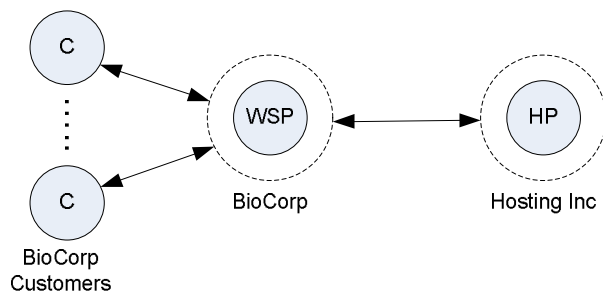


Figure 5. BioCorp dynamically outsource the service to Hosting Inc

After some time, the CTO of BioCorp realises that other hosting companies sometimes offer a lower cost than Hosting Inc. Therefore, BioCorp use a Broker to dynamically choose the cheapest hosting company on a per-message basis.

3.2. Scenario 2: Moving Computation to Data

A bioinformatics service analyses data from a database that is made available as a web service through an interface such as OGSA-DAI [11]. The analysis service sends a query to the database and receives back from it a large quantity of data. It then processes this data and sends a small result set back to the consumer

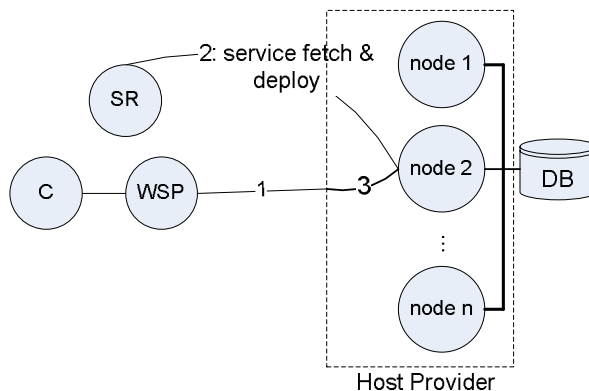


Figure 6. An analysis service is deployed close to a database

In order to avoid transferring large amounts of data over long distances, it is advantageous to deploy the service as close as possible to the database on which it operates. The database provider realises that many services will benefit from deployment close to the database, and so provides a cluster of compute servers close to the database (Figure 5) [12]. The cluster runs the Host Provider component. The provider of the Analysis Service runs the Web Service Provider component, and makes the service available to consumers. When a message arrives for the Analysis Service, it passes it to the Host Provider to ensure that it is processed close to the data (Figure 5, step 1). In the Figure, the Host Provider fetches and deploys the Analysis service from the Web Service Provider (Figure 5, step 2) before the request is processed (on node 2). This ensures that interactions between the database and the analysis service only involve data transfer within the cluster.

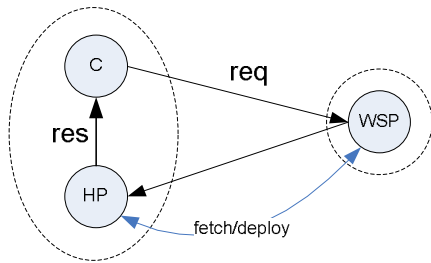


Figure 7. The Consumer Specifies a Local Host Provider

3.2. Scenario 3: Consumer selected Host Providers

It is also possible for the Consumer, rather than the Web Service Provider to choose the Host Provider. This can be done by the consumer passing additional information on the chosen Host Provider in the header of the SOAP message it sends to the Web Service Provider. A Consumer may for example wish to do this if they have significant compute resources available to them. Alternatively, a Web Service Provider may mandate that the Consumer is responsible for identifying a Host Provider.

Figure 6 shows an example in which the consumer has access to local compute resources that are running the Host Provider software. Therefore, the consumer's tooling inserts the location of the local Host Provider in the header of the message sent to the service offered by the Web Service Provider. The message is then sent to the Host Provider for processing. If the service is not already available on the specified Host Provider then it will be dynamically deployed in the normal manner.

4. Evaluation

In order to explore and evaluate the Dynasoar approach, a prototype was built by extending the existing GridSHED resource management system for hosting Grid applications [13, 14].

In the testbed that was used to obtain the results given in the following Section, Apache Tomcat and Apache Axis are used as the Web Services containers throughout the system (we have also produced a version of the system using .Net containers – as all the components are Web services with well defined interfaces, their internal implementation is not architecturally important). The Service Repository is a Web Service that encapsulates a file server and makes available service code that can be downloaded and deployed. For the experiments described in the next section, the dynamically deployed services all had Java implementations, and were designed to run under Tomcat. Service code is stored together with

information on dependencies, such as Java libraries and property files, in a single Web archive (or *war*) file which is the standard file packaging format used by Tomcat for Web applications. The war file must contain the Apache Axis libraries as each Web application requires its own SOAP processor. The war files are automatically hot-deployed inside Tomcat, avoiding the need to stop Tomcat and manually install the service.

Deploying a Web Service consists of copying the war file onto a host managed by the Host Provider so Tomcat can install the service. While these steps are happening, incoming SOAP messages for the service being deployed are temporarily queued. When the Web Service has been successfully deployed, the messages are sent to it for processing.

4.1. Computational resource allocation

We wished to investigate whether Dynasoar could be built as a higher layer over existing Grid fabrics. Consequently, for the evaluation, Condor is used in the Host Provider. Each SOAP request is “wrapped” in a generic Condor client job which, upon execution, sends the request to a suitable Web Service deployed on the Host Provider (if one exists). When service deployment is required, Condor daemons are reconfigured to run as central managers instead of a slave, and the new Web Service is downloaded and installed on the host.

5. Results

In order to demonstrate the viability of the Dynasoar architecture, this section presents results for dynamically deploying a service close to a database. The aim is to test the argument made in Scenario 2 above that it can be advantageous to deploy analysis services close to the database on which they operate. We also test the resource allocation system within a Host Provider to demonstrate how computational resources are reconfigured using dynamic Web Service deployment to match increasing demand.

5.1. Dynamic deployment results

The first experiment consisted of measuring the behavior of a bioinformatics application in which an analysis Web Service extracted and processed genetic data from a database. This represents an example of the scenario described in Section 3.2. Three tests were conducted in which messages were sent to the same Web Service in order to compare response times with and without the Dynasoar architecture. Within each

test, the number of tuples requested from the database was varied so that the effects of changing the quantity of data transferred could be observed. Each separate test was conducted 10 times and the results averaged to remove unpredictable network and scheduling delays. The tests were:

1. No dynamic service deployment. The analysis service communicates remotely with the database through a domestic 512Kbps broadband connection.

2. Dynamic service deployment. The Web Service Provider and Service Repository are remote from the Host Provider, whose nodes are connected to the database by a 100Mbps network. When a request for the service is sent to the Web Service Provider, the service is dynamically deployed on the Host Provider by the Dynasoar infrastructure.

3. Invoking a Service that has already been dynamically deployed. Dynasoar is also employed in the same configuration as in test 2, but with the Web Service having already been dynamically deployed on the Host Provider.

The graph in Figure 8 shows the results for the three different tests. The main feature is the significant increase in the time taken to retrieve and process data from the database for a larger amount of data when the analysis service is remote from the database (test 1). In contrast, as the number of tuples requested from the database increases, the processing time in the scenarios employing the Dynasoar architecture rise significantly more slowly. Once the number of tuples exceeds 800,000, the Dynasoar approach gives considerable performance benefits due to the fact that the cost of transferring data over a local 100Mbps network is significantly less than over a 512Kbps connection. For example, at 6.4M tuples, a roughly ten times performance improvement was observed. In contrast, when only a small amount of data was requested from the database, the remote service had a faster response time than when the Dynasoar infrastructure was used, due to the costs of deploying the service. This cost is however, incurred only once by the Dynasoar infrastructure - on the first access to the service. For subsequent accesses, the analysis service is already deployed, and so the only cost is in routing the SOAP message from the client to the service in the Host Provider. The cost of dynamically deploying a service can therefore be seen in the relatively constant performance difference between the two tests taken with Dynasoar in use.

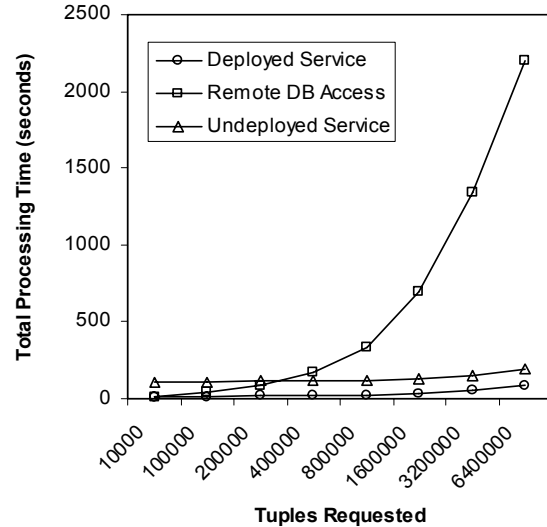


Figure 8. Measured response times for the three tests

Performance Scalability

The ability to dynamically deploy services on its internal Grid infrastructure gives the Host Provider the opportunity to match increases in the demand for a service by deploying it on more nodes. The prototype Host Provider implementation exploits the GridSHED [13, 14] resource allocation system that determines when additional service deployments are made within the Dynasoar implementation. Experiments were run to investigate the effectiveness of this approach.

In the experiment, a deployed Web Service receives an increasing rate of requests (in the form of SOAP messages) over time. Consequently, in order to share out the load, the service is dynamically deployed on other nodes so that requests can be balanced across the set of deployed services. The Web Service used here was the same as that used in the previous experiment. Six different arrival rates were used, starting at 0.03125 and doubling continuously up to 1 job per second. Ten messages were sent at each arrival rate. Each request returned 3,200,000 database tuples (this took 48.4 seconds in the previous experiment). Each node in the experiment was a dual processor, and so the results are presented in processors per pool.

The graph in Fig. 9 shows that as the arrival rate is increased, the number of nodes on which the service is deployed is automatically and dynamically increased by the GridSHED Host Provider software (this is labeled "Processors in pool"). The results show that this is effective at keeping the response time fairly stable until the arrival rate reaches 0.5 jobs per second. At this point, the system becomes over-utilised, with

too many requests for the system to process even when more processors are added, up to a total of 16. If the arrival rate of requests for the service reduced over time, then the GridSHED manager would reduce the number of processors in the pool if that would allow another service to better meet its performance demands.

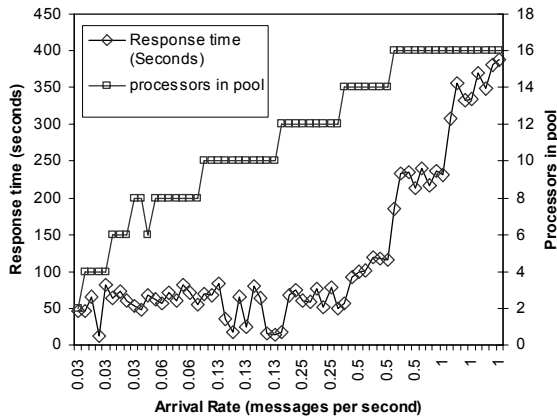


Figure 9. Servers processing a newly deployed service with an increasing arrival rate

6. Related Work

The work described in this paper benefits from exploiting the results of work that is producing components on which dynamic deployment implementations can be built. In particular, the Host Provider can exploit existing job scheduling fabrics such as Condor [1] and Globus [2]. These provide ways to gather information on machine characteristics (type, software installed) and CPU loadings. Our aim is that the Host Provider can sit on top of existing Grid infrastructure as a high level service.

However, the shift in focus from job scheduling to service deployment requires other issues to be addressed. A key decision is to decide when to deploy a service on a new node within a Host Provider, rather than utilising an existing (though perhaps overloaded) deployment. This area has been extensively investigated in the GridSHED project for job scheduling [13, 14], and the results are now being utilised in the implementation of the Host Provider, as described in the last section.

There are existing systems that are able to perform dynamic service or application deployment within a local network. An example is IBM's Websphere XD.

However, of particular relevance is the work described in [15] which also includes a store for service code and dynamic deployment for loading and dependability reasons. However, a key difference is that this work does not address the separation of the Web Service Provider from the Host Provider such that they can be provided by independent parties, potentially distributed over a Grid or the Internet. Our view is that making this separation, defining the interactions between the parties, and introducing decision-making brokers, opens the opportunity for a range of new opportunities for Grid and Internet Scale distributed computing. Therefore, the two projects have a different focus.

In the Brokering area, of particular interest is work on payment and negotiation [16] in which clients can use the cost of processing an invocation of a service to choose between a set of possible hosts. Once a decision has been made then the service is deployed on the chosen host.

The architecture of the Dynasoar deployment infrastructure is designed to be independent of the way in which the service is deployed. The initial prototype utilised the dynamic service deployment facilities of Axis/Tomcat, but a range of other options has been investigated, including services encapsulated within Virtual Machines and database stored procedures. For these reasons, a medium term goal is to utilise one of the emerging, generic distributed system deployment mechanisms such as SmartFrog [17], or CDDLM [18], when these have stabilised.

7. Conclusions

This paper has given an overview of work in the Dynasoar project. We believe that separating out the Host Provider and Web Service Provider components, and defining interfaces through which they can interact, offers opportunities to implement a range of different options for processing messages sent to Web Services on a Grid or Internet-scale distributed system. This includes opportunities to create a marketplace for Host Providers that are dynamically chosen based on cost and/or quality of service guarantees. The work has shown that Host Providers can be built as high-level services sitting on top of an existing Grid infrastructure (e.g. Condor [1] which has been used for our evaluation), exploiting the large investment that has already been made in the design and deployment of distributed job scheduling systems. However, the results also show the importance of having Host Providers that can make good decisions on when a service should be deployed on an extra node in order to meet changing loading requirements. This has been

provided in Dynasoar by exploiting the results of the GridSHED project [13, 14].

The performance results show the effectiveness of Dynasoar for improving application performance. In the first experiment, costs were dramatically reduced by dynamically deploying an analysis service close to the data on which it operates. In the second, the ability to dynamically deploy multiple instances of a service was used to maintain response times as the arrival rate of requests increased.

The project is moving on to further investigate some of the issues. Current work includes tuning the cost model [13] that governs when and where services are deployed; experimenting with services that maintain internal state (e.g. dynamically deploying replicas of databases to act as structured data caches), and the co-deployment of sets of services, for example in workflow execution).

8. Acknowledgements

We are indebted to our collaborators in the GridSHED project: Isi Mitrani, Jennie Palmer, Rob Smith, Paul McKee (BT) and Mike Fisher (BT). We would like to acknowledge the support of the UK Engineering and Physical Sciences Research Council and the DTI for the GridSHED and DAIT projects. This has been provided through the UK e-Science core programme.

10. References

- [1] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor - A Distributed Job Scheduler," in *Beowulf Cluster Computing with Linux*, T. Sterling, Ed.: The MIT Press, 2002.
- [2] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputing*, vol. 11, pp. 115-128, 1997.
- [3] Sun, "Sun Grid Engine." <http://www.sun.com/software/gridware/>.
- [4] S. Chatterjee and J. Webber, *Developing Enterprise Web Services: An Architects Guide*: Penguin Books, 2003.
- [5] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "Grid Services for Distributed System Integration," *IEEE Computer*, vol. 35, pp. 37-46, 2002.
- [6] S. Parastatidis, J. Webber, P. Watson, and T. Rischbeck, "WS-GAF: A Grid Application Framework based on Web Services Specifications and Practices," *Journal of Concurrency and Computation: Practice and Experience*, vol. 17, pp. 391-417, 2005.
- [7] W3C, "Web Services Description Language (WSDL)." <http://www.w3.org/2002/ws/desc>.
- [8] W3C, "Simple Object Access Protocol (SOAP) 1.1," in *W3C Notes*, D. Box, D. Ehnebuske, G.

- Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer, Eds. <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>, 2000.
- [9] VMWare, "VMWare," 2006.
- [10] Microsoft, "Virtual PC," 2006.
- [11] OGSA-DAI, "OGSA-DAI." <http://www.ogsadai.org.uk/>.
- [12] P. Watson and P. Lee, "The NU-Grid Persistent Object Computation Server.," presented at 1st European Grid Workshop, Poznan, Poland, 2000.
- [13] J. Palmer and I. Mitrani, "Optimal Server Allocation in Reconfigurable Clusters with Multiple Job Types," presented at Computational Science and its Applications (ICCSA 2004), Assisi, Italy, 2004.
- [14] C. Kubicek, M. Fisher, P. McKee, and R. Smith, "Dynamic Allocation of Servers to Jobs in a Grid Hosting Environment," *BT Technology Journal*, vol. 22, pp. 251-260, 2004.
- [15] M. Keidl, S. Seltzsam, and A. Kemper, "Reliable Web Service Execution and Deployment in Dynamic Environments," presented at Technologies for E-Services, Berlin, 2003.
- [16] J. Cohen, J. Darlington, and W. Lee, "Payment and Negotiation for the Next Generation Grid and Web," presented at UK e-Science All Hands Meeting 2005, Nottingham, 2005.
- [17] HP, "The SmartFrog Reference Manual," 2004.
- [18] Global Grid Forum, "Configuration Description, Deployment, and Lifecycle Management. XML Configuration Description Language Specification," 8/9/2004 2004.