

Dynasoar: An Architecture for the Dynamic Deployment of Web Services on a Grid or the Internet

Paul Watson & Chris Fowler

School of Computing Science, University of Newcastle, Newcastle-upon-Tyne, NE1 7RU

Abstract

This paper describes an architecture for dynamically deploying Web Services, on-demand, over a grid or the Internet. The Dynasoar project is investigating an approach to grid computing in which there are no jobs, but only services. Rather than scheduling jobs, it automatically deploys a service on an available host if no existing deployments can meet the computational requirements. The architecture 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. The paper describes the architecture, a prototype implementation, and new e-science models that it supports.

1. Introduction

Distributed job scheduling systems that can dynamically route client jobs to available, remote computing resources for execution are now widely available (e.g. Condor [1], Globus [2] and the SunGridEngine [3]), and found at the heart of most grid computing infrastructures. The job – a combination of code to be executed and (in many 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. An application is represented as a set of services that communicate through the exchange of messages. Consequently, Grid and other distributed applications are constructed by combining a set of services. However, if the computational requirements of a service cannot be met by its hosting environment then a job must be created and sent 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 (named as a short form of DYNAMIC Service Oriented ARchitectures), exploring an alternative architectural approach that removes the concept of jobs, and allows application and service writers and consumers to remain wholly within the service-based framework. Here, the equivalent to a job is the combination of a service, and a message sent to it for processing.

To enable this approach, the architecture allows a service to be dynamically deployed on an available host in order to utilise its computational power, if no existing service deployments can meet the computational requirements.

We believe that this approach has potentially three main advantages:

a) it simplifies the writing of applications and services by allowing writers to remain entirely in a service-oriented framework, rather than within one in which they have to deal with both services and jobs.

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, “one-off” executions, job schedulers lack the ability to share the cost of the movement and installation of code across multiple executions.

c) it opens up opportunities for interesting new organisational/business models for Service Providers, and Host Providers.

In order to investigate these ideas, a prototype of Dynasoar has been built and evaluated. This paper describes the Dynasoar architecture (Section 2), some of the e-science organisational models that it enables (Section 3) and presents some performance results (Section 4) before drawing conclusions (Section 5).

2. The Dynasoar Architecture

The aim of this work was to design a generic infrastructure for the dynamic deployment of web services, in response to the

arrival of messages for them to process. 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 a well defined interface through which they interact.

The *Web Service Provider* accepts the incoming SOAP [4] message sent to the endpoint and is responsible for arranging for it to be processed. It does this by choosing a *Host Provider* and passing to it the SOAP message and any associated QoS information. It holds a copy of the service code (in a “Service Store”) ready for when dynamic service deployment is required.

The *Host Provider* controls computational resources (e.g. a cluster or a grid) on which services can be deployed, and the messages set to them to be process. Therefore, it accepts the SOAP message from the *Web Service Provider* (along with any associated information) and is responsible for processing it and returning a response to the consumer (if one is required).

When the 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:

Case 1: The Service is already deployed on the node on which the SOAP message is to be processed (to assist in the choice of node, it is likely that the *Host Provider* will maintain a registry of which services are deployed on which nodes). Here the *Host Provider* simply routes the SOAP message to a node on which the service is deployed and passes it to the local endpoint of the deployed service that will process it. Any response message is sent back to the consumer (currently this is routed via the *Web Service Provider*, but an implementation that used WS-Addressing could send it directly). This case is shown in Figure 1. A request for a service (s2) is sent by the *Consumer* to the endpoint at the *Web Service Provider* which passes it on to a *Host Provider*. The *Host Provider* already has the service s2 deployed (on nodes 1 and n in the diagram) and so, based on current loading information it chooses one (node n) and routes the request to it for processing. A response is then routed back to the consumer. 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. It simply communicates with the *Host Provider*, which manages its own internal resources.

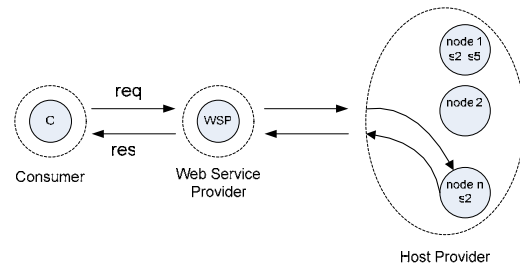


Figure 1 A request is routed to an existing deployment of the service

Case 2: The Service is **not** already deployed on the node on which the SOAP message is to be processed. This may be because the *Host Provider* has no nodes on which the service is deployed, or it may be because demand for the service is rising and so in order to maintain acceptable performance levels the service must be deployed on another node.

To allow for the dynamic deployment of services, the *Web Service Provider* maintains a “Service Store” that holds deployable versions of services and associated metadata (in the prototype implementation, the *Service Store* is actually itself a *Web Service*, and so potentially it could be shared by several *Web Service Providers*). The architecture makes no assumptions about the nature of the deployable services except that, once deployed, they are *Web Services* that accept SOAP messages. In the simplest case, for Java-based services they may be files that are deployed by being copied into the appropriate directory of an Axis Tomcat server. However, for more complex cases they could, for example, be a set of executable commands that install a database, populate it by replicating the contents of another database, and then install a *Web Services* wrapper (e.g. OGSA-DAI [5]). Alternatively, they may be a “frozen” virtual machine containing a complex environment that can be copied to a host and installed there. Each service may have multiple entries in the *Service Store*, e.g. different versions for different types of host (e.g. Windows, Linux and Solaris).

Included in the information that the *Web Service Provider* sends to the *Host Provider* along with the SOAP message is an identifier for the service being called and the endpoint of the *Service Store*. If the *Host Provider* decides to deploy a service on a new node then it sends a message to the *Service Store* requesting the code for the service. When this returns, the *Host Provider* installs the *Service* on the node and routes the message to its new local endpoint. The message is then processed and the result (if any) returned to the consumer. An example of

this case is shown in Figure 2. A request for a service (s3) is sent to 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 s3 deployed on any of the nodes it controls and so, based on current loading information it chooses one (node 2), fetches the service code from the Web Service Provider and installs the service on that node (step 2). It then routes the request to it for processing (step 3). A response is then routed back to the consumer. Node 2 may continue to host service s3, and so be ready to process any other messages routed to it by the Host Provider.

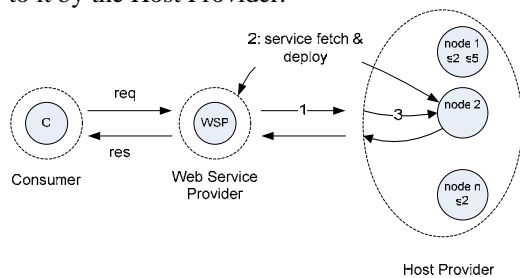


Figure 2 A service is dynamically deployed to process a request

It should be noted that the Consumer need not in any way be 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.

Once a service is installed on a node it can remain there ready to process future messages until it needs to be reclaimed, perhaps to make way for other services that have become more in-demand. This retention of deployed services has the potential to generate large efficiency gains when compared to job-based scheduling systems in which each job execution requires its own installation. A related project, the GridSHED project [6, 7] uses mathematical models to generate heuristics to determine when it is desirable to install a service on another node, versus when it is better just to use the existing installations.

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).

In Figure 1 and Figure 2, only a single Host Provider was shown. However, because there is

a well defined interface for the interactions between the Web Service Provider and the Host Provider, it is possible for the Web Service Provider to make a dynamic choice between Host Providers, for example on the basis of cost, availability of resources, or quality of service. This also opens the way for organisations or third parties to build marketplaces to match the requirements of the Web Service Providers against the offerings of the Host Providers (Figure 3). The Marketplace would offer the same interface as a Host Provider but, on receiving a message, would dynamically select a Host Provider that could best meet the needs of the Web Service Provider. It would then pass the request message on to the chosen Host Provider for processing (this would therefore be transparent to the Consumer or Web Service Provider). As before, if required the Host Provider could download a service from the Web Service Provider and dynamically deploy it.

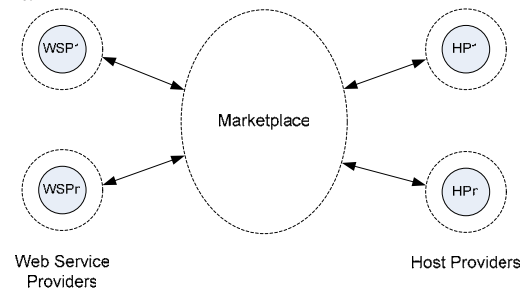


Figure 3 A Marketplace for Matching Web Service Providers to Host Providers

In implementing the Host Provider, the aim should be to build on the facilities provided by existing cluster and grid job scheduling solutions. Information management services (e.g. MDS [8]) allow loadings to be determined, so providing guidance on to which nodes messages should be routed, or new services deployed. At a higher level, the functionality of job scheduling systems can be used to identify hosts that already provide a particular service (e.g. using Condor) and also a means to route messages to those nodes. Consequently, the Host Provider can be viewed as a high level service building on the functionality offered by lower level scheduling and information management services. For example, where the lower-level infrastructure is relatively sophisticated, QoS requirements could be provided by the Consumer in the header of the request and used in the selection of a node for processing the message.

The introduction of a market-place allows failure to be handled at three levels. Failure of a node in a Host Provider can be managed

internally by directing messages to other nodes, and/or deploying the service on another node. Failure of an entire Host Provider can be handled by the marketplace or Web Service Provider selecting an alternate Host Provider. Failure of a Web Service Provider can be handled through the normal Web Services mechanism of the Consumer going to a registry to find an alternative endpoint for the same service.

2.1. Allowing the Consumer to select the Host Provider

In some cases, it may be desirable, or absolutely necessary, for the consumer to specify the Host Provider, rather than leave this choice to the Web Service Provider. Examples of this are when:

- the Web Service Provider does not wish to use their own associated Host Provider, nor enter into a relationship with one. For example, a company may wish to sell access to their Web Services, perhaps by charging per message processed. However, they may not wish to be responsible for actually processing the messages sent to those services.

- the consumer has their own powerful compute resources that they wish to utilise. They therefore deploy their own, local Host Provider component. This may, for example, give them faster or cheaper hosting than they would get by using an external Host Provider.

- the consumer wishes to control the choice of Host Provider in order to exploit their knowledge of the behaviour of their application. For example, a consumer may know from experience that they get better performance from a service if it is deployed on a host that is close to a database on whose data the service operates.

These scenarios are described in more detail in the next Section. However, the solution is the same: Host Provider information is included in the header of the SOAP messages sent by the Consumer to the Web Service Provider. When the Web Service Provider receives the messages it uses this information to select the Host Provider specified by the consumer. For example Figure 4, shows an example where a Consumer sends a message to a service offered by the Web Service Provider. The service is computationally expensive, but the consumer has access to local compute resources that is 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 provider then utilises that Host Provider to process the message. If the service is not already available on the specified Host Provider then it will be dynamically deployed in the normal manner.

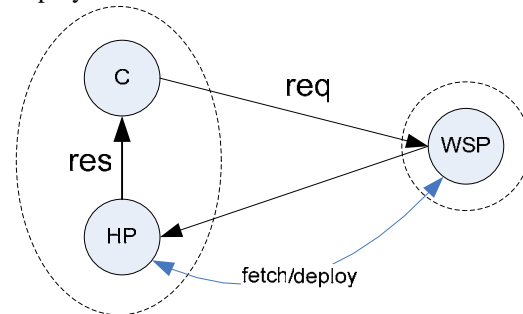


Figure 4 The Consumer Specifies a Local Host Provider

The dynamic deployment infrastructure must respect the security and policy requirements of the three parties: the Consumer, the Web Service Provider and the Host Provider. To satisfy this, a tripartite trust model is being explored. This has the aim of managing the relationships between all three parties using security and policy mechanisms. Dynasoar allows each party to express their security policies, which are then enforced at run-time.

Examples of the relationships that the trust model must capture are:

- the Web Service Provider may wish to restrict the Host Providers it utilises. For example it may only wish to use Host Providers that it trusts to not offer the deployed services directly to consumers as a ploy to avoid paying the Web Service Provider for each use of the service.

- the Host Provider may only wish to deploy services from a restricted set of Web Service Providers. This may be to reduce the risks of hosting malicious code.

- the Consumer may wish to have their messages processed only by Providers they trust. An example would be a pharmaceutical company wishing to interact only with providers who are trusted not to monitor their requests and responses.

- the Web Service Providers may only wish to process messages from certain Consumers. For example, only those with whom they have a business relationship, so that they know they will receive payment.

3. e-Science Scenarios

In this section we describe some common e-science usage scenarios that can be supported

by the Dynasoar infrastructure described in the previous section.

3.1. Offering a Service without providing hosting

A researcher has written a web service that they host on their own desktop computer, and use successfully for their own research. At a conference, they describe the service, which generates some interest and has others asking if they may use it. However, the researcher does not own the compute resources required to host access to the service by others - it is computationally intensive and so large CPU resources would be required to meet the requirements of the community of potential users. Nor could the researcher pay a Host Provider company to host the service as the researcher has no way of recovering the cost from users. Therefore, Scenario 1 is not appropriate.

The researcher could choose to make the code available for downloading on their website, but this would require potential users to be able to manually download and install it on suitable local resources. This requires a level of knowledge that many users do not have. Further, if the researcher upgraded the service, perhaps to fix a bug, then those who had already downloaded and installed it would not see the benefit.

Instead, the researcher utilises Dynasoar. They deploy the Web Service Provider component locally and add the service into the Service Store. Any interested consumers must arrange their own Host Provider (perhaps by deploying it on one of their own machines, or by finding a Host Provider that they are allowed to use, perhaps one provided by their organisation). When the consumer sends a message to the endpoint, it sends in the header information on the location of the Host Provider that can be used (this could be done by a client library). This allows the Web Service Provider to route the message to the Host Provider, which can, if necessary, retrieve the Service from the Service Store and install it as described in Section 2.

3.2. A Grid supporting a community of researchers

An organisation such as a university, corporate lab, or national body wishes to make a pool of computational resources available for researchers. They provide a grid infrastructure (e.g. the UK National Grid Service) that can

accept and dynamically schedule jobs over a set of nodes. However, the researchers they support also wish to utilise computationally intensive services that they or others have written. Therefore, the Host Provider component is deployed on top of the grid. Researchers can then deploy the Web Service Provider infrastructure locally, and configure it to utilise the grid for dynamically deploying their services (Figure 5). This has the advantage that researchers can make their services available to colleagues, but leave the job of hosting them to a grid provider.

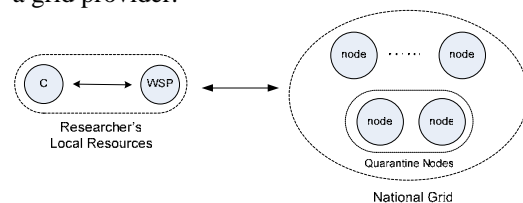


Figure 5 A National Grid Providing Resources to a Remote Researcher

The managers of the grid may be concerned about deploying arbitrary services on their nodes (though currently many grids accept arbitrary jobs from users). Therefore, they may choose to implement a policy such that when the Host Provider component receives a request to process a message for a new service, they deploy that service only on a subset of their resources ("quarantine nodes") where its behaviour can be monitored to check that it will not disrupt other users. Once the managers are comfortable with the service then the restriction can be lifted, and the service would be allowed to be deployed on any suitable node on the grid. This could all be controlled automatically through the use of policies set in the Host Provider, which would specify any restrictions on the nodes on which a service could run. The policy for a service could be changed either automatically (as a result of monitoring the behaviour of a new service) or through manual intervention by a manager of the Grid.

This scenario still contains the danger that if a researcher makes available a service that proves popular, the researcher's local Web Service Provider component will have to be powerful enough to accept all the incoming messages, and forward them to the Host Provider. To prevent this, the managers of the grid could also deploy a local instance of the Web Service Provider component on scalable resources (c.f. scalable implementations of a Web Server), and allow popular services to be made available (e.g. have published endpoints) directly from there.

3.3 Moving Computation to Data

A bioinformatics service is written that analyses data from a database that is also made available as a web service through an interface such as OGSA-DA [5]. 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 as shown in Figure 6 (the thickness of the arrows indicates the quantity of data transferred).



Figure 6. Interactions between an Analysis Service and a Database Service

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 an AIR architecture [9], with a cluster of compute servers close to the data (Figure 7). The cluster runs the Host Provider component. The provider of the Analysis Service (which may be a lone researcher or a commercial company that specialises in writing analysis services) runs the Web Service Provider component, and makes the service available through this. Consequently, either the consumer (through providing the information in the request header) or the Analysis service provider can specify that the service should run on the cluster close to the database service. Therefore when the Web Service Provider receives a message for the Analysis Service, it passes it to the cluster's Host Provider to ensure that it is processed close to the data (Figure 7:1). In the Figure, the Host Provider fetches and deploys the Analysis service from the Web Service Provider (Figure 7:2) before the request can be processed (on node 2).

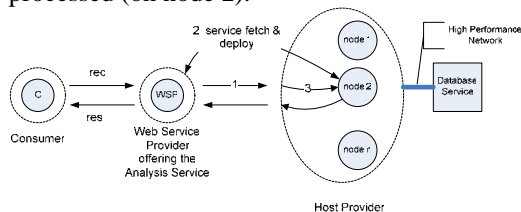


Figure 7. The Analysis Service is dynamically deployed close to the Database Service

4. Results

The behaviour of a bioinformatics application was analysed in order to explore the behaviour of Dynasoar. An analysis Web Service extracts and processed genetic data from a database.

Three different tests were run:

1. No Dynamic Service Deployment. The Web service is manually deployed remotely from the database, with which it interacts through a 512Kbps broadband connection.

2. Dynamic Service Deployment. The Service Provider is remote from the Host Provider, whose nodes are connected to the database over 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.

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 already dynamically deployed on the Host Provider.

Within each test, the number of tuples requested from the database was varied so that the effects could be observed. Each separate test was conducted 10 times and the results averaged to remove unpredictable network and scheduling delays. Figure 8 shows the results. It can be seen that there is a significant increase in the time taken to retrieve and process data from the database when the Web service is deployed remotely. This is in contrast to the behaviour when Dynasoar is used to dynamically deploy the service. The difference in time between the two Dynasoar tests represents the cost of service deployment.

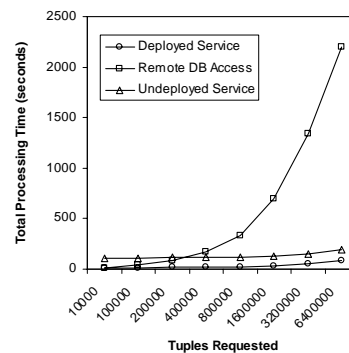


Figure 8. Measured response times for the three tests

5. Conclusions

This paper has given an overview of work in the Dynasoar project on dynamically deploying services in response to changing demand and

host availability. It does so by defining a generic infrastructure consisting of two components: the Web Service Provider, and the Host Provider. By separating out these two components, and defining interfaces through which they can interact remotely, this approach 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 spread over the internet. Further, it offers the possibility for specialist service writers to make available their services for others to use, without having to provide any hosting capability.

The current prototype can dynamically deploy .war files into Tomcat/Axis containers. However, the service store is not tied to any one deployment mechanisms, and we are currently exploring the use of VMs to dynamically deploy more complex computational environments [10, 11].

We are also currently developing the prototype to include the use of QoS to inform decisions on the choice of Host Provider.

6. 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, C. Kesselman, and C. K. I. J. S. Globus: A Metacomputing Infrastructure Toolkit. I. Foster, "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] W3C, "SOAP 1.1." <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>.
- [5] OGSA-DAI, "OGSA-DAI." <http://www.ogsadai.org.uk/>.
- [6] 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.
- [7] 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.
- [8] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman., "Grid Information Services for Distributed Resource Sharing," presented at Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), 2001.
- [9] P. Watson and P. Lee, "The NU-Grid Persistent Object Computation Server.," presented at 1st European Grid Workshop, Poznan, Poland, 2000.
- [10] VMWare, "VMWare." <http://www.vmware.com>.
- [11] Microsoft, "Virtual PC." <http://www.microsoft.com/windows/virtualpc/default.mspx>.