

# Web Services Usage Monitoring for Dynasoar

Salvatore Cavalieri, Fabio Scibilia  
*Dep. of Computing and Telecom. Engineering  
Faculty of Engineering, University of Catania  
Viale A.Doria, 6 95125 Catania, Italy  
e-mail: {Salvatore.Cavalieri,  
Fabio.Scibilia}@diit.unict.it*

Chris Fowler, Savas Parastatidis, Paul Watson  
*School of Computing Science  
University of Newcastle  
Newcastle upon Tyne, NE1 7RU, UK  
e-mail: {Chris.Fowler, Savas.Parastatidis,  
Paul.Watson}@newcastle.ac.uk*

## Abstract

*Dynasoar is a service-oriented framework for the dynamic deployment of services on the Grid or the Internet. This paper proposes a framework for monitoring the usage and messaging activity of Web Services deployed through and managed by Dynasoar. Message routing or service deployment-related decisions are made based on the analysis of deployment and run-time information collected during interactions with services. The framework can be extended by adding new algorithms to process and manage such information. This paper describes the architecture, design, implementation, and evaluation of the proposed monitoring framework for Dynasoar.*

## 1. Introduction

The management of Web Service deployments in changing runtime environments, and the need to dynamically adapt to the characteristics of load, quality of service, security, financial cost, etc. of the hosting environments and the network, has been the research topic of Dynasoar [1]. This paper focuses on the work done within the Dynasoar project on monitoring the usage of Web Services by collecting information about the messaging behaviour of Web Services and the load characteristics of the hosting environments. The collected information can be processed through custom decision-making algorithms which can be introduced to the framework. We have designed and built the service usage monitoring framework as a service which collects and maintains usage-related information. The resulting service also hosts the information processing algorithms used for the analysis of that information in order to make load-balancing, message routing, and deployment related decisions.

This paper is organized as follows: Section 2 introduces Dynasoar and explains why a service usage monitoring service is required; Section 3 describes the

architecture and implementation of the monitoring framework built for Dynasoar while Section 4 discusses the load balancing algorithm that has been implemented for the monitoring framework; Section 5 presents our performance evaluation of the system; Section 6 looks into related work and finally Section 7 offers conclusions.

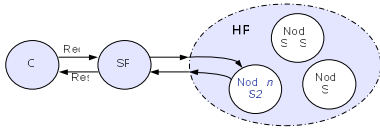
## 2. Dynasoar and Service Usage Monitoring

In high-performance computing the ‘job’ abstraction has been used extensively as the means to reason about the distribution of computation and the exploitation of remote computational resources. In a service-oriented world, current practice dictates the creation of service wrappers around job submission systems in order to enable interactions with them. Also, in a typical job-submission scenario, a job is submitted, executed, and then discarded. In a service-oriented architecture a service is deployed, rather than created, and once deployed it can deal with multiple interactions until it is explicitly undeployed [2]. This subtle difference represents a good example of the transition between the job-submission-based and service-oriented models of building distributed, large-scale applications. Dynasoar [1] describes a service-oriented architecture for the dynamic deployment and hosting of services on the Grid or the Internet. Like traditional job-scheduling systems, an implementation of the Dynasoar architecture must also be able to make deployment and message routing decisions based on load and security policy. We describe the architecture of Dynasoar in Section 2.1, while we detail on the need for monitoring Web Services usage in Section 2.3.

### 2.1. The Dynasoar Architecture

Dynasoar defines the roles of three actors in a service-oriented approach to the dynamic deployment of services: the *Consumer* (C), the *Service Provider* (SP)

and the *Host Provider* (HP). The Consumer is the originator of messages sent to services; the Service Provider makes services available to Consumers; and the Host Provider is responsible for hosting those services.



**Figure 1 - Dynasoar architecture**

The Service Provider does not need to know anything about the internal structure of the Host Providers to which it chooses to deploy a service. A Host Provider service may encapsulate anything from a single computer to an entire Grid of computational resources. In Figure 1 an example of a Host Provider service implemented to internally use a cluster is shown. The initial request to a service S2 causes the code for that service to be deployed at the Host Provider. Internally that HP may use one of its many available nodes transparently to the SP. After the deployment, messages to the service S2 are forwarded by the SP to the HP.

A Service Provider may use many Host Providers at the same time, even for the same service, according to the policies and service hosting requirements that are in place. There can even be a marketplace of Host Providers that are dynamically chosen to host services offered by the Service Provider.

## 2.2. The need for Service Usage Monitoring

In a typical deployment, a Service Provider will have to choose on which of the available Host Providers to deploy a particular service, or whether additional deployments are necessary to satisfy the Quality of Service (QoS) requirements of that service. Furthermore, there might be service-level agreements with its Consumers in place that should be honoured (e.g. maximum response time). While some of the characteristics of a particular deployment could be monitored by a Service Provider (e.g. the time taken for a response message to be routed back to the Consumer), additional information about the hosting environment of a deployment may be necessary. Also, a request message sent from a Consumer to a service offered by a Service Provider may not always involve a response, which makes it difficult to infer QoS related information.

To enable the sharing and management of host-utilisation and service-health information between a Service Provider and its Host Providers, a service usage monitoring infrastructure for Dynasoar is needed, preferably offered as a service. Such a service would

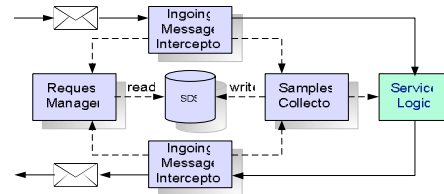
enable the periodic querying of Host Providers for hosting-specific information on deployed services such as runtime performance, service-health, etc. The data collected from the queries should become available for analysis by heuristic algorithms. The Service Provider makes use of the offered analysis algorithms in order to better manage the deployment of services (e.g. choose which Host Provider is better suited for a particular deployment by matching a service's QoS requirements and what is being offered, decide whether additional deployments are needed for load-balancing purposes).

## 3. The Architecture of a Service Usage Monitoring Framework

The framework is structured around three entities: *Samples*, *Filters* and *Algorithms*. A Sample is a set of information related to a service request and encapsulated within a data structure, which can then be stored in a Samples Data Store (SDS) for later analysis. These Samples relate to general info such as the time a request arrived, the time the response (if any) was sent back, and the successfulness of the service execution and other service-specific information such as either the list of input parameters in SOAP-RPC style messages or the entire SOAP [3] body part of the message.

A filter is an entity that decides whether or not to consider a Sample while computing an algorithm. This is done by basing filters on certain selection criteria (e.g. execution time). A typical filter, for example, could choose just the Samples collected in last few minutes and therefore discard older Samples.

An algorithm is a process that, given a set of filtered Samples, calculates some useful information related to a particular aspect of interest (e.g. the evaluation of the mean processing time for a service request).



**Figure 2 - Framework architecture**

Figure 2 shows the Architecture of the Framework. Solid lines show the path of the ingoing and outgoing SOAP messages, while dotted lines show interactions between the components. Shaded shapes are components of the Framework. The Framework comprises two main functional blocks: the first one, the *Samples Collector*, is concerned with the creation and storage of the Samples, while the second one, the *Request Manager*, is concerned with the management of re-

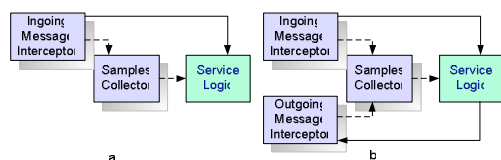
quests and the execution of a set of certain algorithms over Samples stored in the SDS.

The *Ingoing Message Interceptor* and the *Outgoing Message Interceptor* catch the ingoing and outgoing messages respectively, and pass them to the other components.

In Section 3.1 we expose some way for collecting Samples on the usage of the Web Service, while in Section 3.2 we describe how hosted algorithms worked over collected Samples.

### 3.1. Collecting Samples

Collecting historical information on the Web Service Usage is the most important function of the Framework. Information is considered service specific when it deals with the logical functionality of the service; it is not service specific in other cases. For example, the time when a SOAP message arrived and the time taken by the Web Service Container to process the message are examples of non-service specific information. The content of the body of the message, the list of input parameters (in SOAP-RPC messages style) and the content of the body of the SOAP message response, if any, are examples of service specific info. Each Sample must be collected while adhering to the logic functionality of the service. Once a Sample is collected, it is pushed in the SDS to be stored permanently.



**Figure 3 - Examples of collecting samples**

Figure 3 shows a non-exhaustive list of examples of the ways for Collecting Samples. Figure 3a shows the simplest case of one-way SOAP messages. This is the case when the logic of the Web Service is invoked with the processing of a single message and no response is needed. A typical example is a Web Service that writes a stream of data into a data store. Measuring the time taken to process the message is not service specific information, while the content of the message is service specific info (e.g. the length of the stream). Once the message has been processed, the Samples Collector has all the necessary information to create and store a new Sample.

Figure 3b shows a second interesting case, one where a SOAP message response is needed for the requested service. A SOAP message containing a request is received at a given time  $T=t_1$ . The response is sent back later at any time  $T=t_2$ . An example is a Web

Service that executes a heavy process (e.g. a simulation) and sends back to the service requester a SOAP message containing results of the execution (e.g. the output of the simulation) possibly some hours later. In this case the Samples Collector must be able to bind the SOAP request message with its right SOAP response to create the Sample. This might be done using message correlation mechanisms such as WS-Context [4] and WS-Addressing [5].

Looking at the above examples we make the following observations:

1. The Samples Collector has to be deployed in such a way to intercept ingoing and outgoing SOAP messages (e.g. in the same Web Services Container).
2. The way of collecting Samples is strictly related to the message exchange pattern.
3. Samples may contain both service specific information and non-service specific information.
4. An already deployed Web Service is unaware of being monitored. This means that it doesn't need any modification in its logic and that the Samples Collector can be deployed independently of it.
5. The Samples Collector is, in most of cases, not expensive in terms of computation as it works by just extracting information from SOAP messages as they pass through the intercepting point.
6. Despite collecting Samples being service specific, it is not implementation specific.

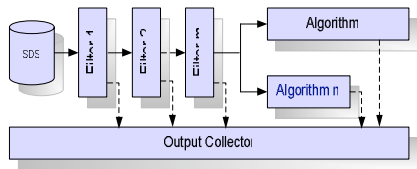
The examples above lead us to consider that the collection of Samples is still an open issue which needs to be investigated further in association with the emerging Web Services related standards. For example, a standard Samples Collector could be implemented using future specifications such as WS-Addressing that allows the correlation of a message request with the associated response.

### 3.2. Computing Algorithms

The Request Manager is the core component of the system. Its task is the interception of the ingoing requests, the execution of the requested algorithm over the Samples stored in the SDS and the delivering of the response. A request is inserted within a SOAP message as a SOAP header element and is characterised by a list of *queries* and a set of attributes that indicate the *urgency* of the response. It is important to point out that a pair of request/response messages is not contained within the same pair of request/response SOAP messages, first of all because a request SOAP message could be a one-way message. Moreover, the response to the query might be ready after the Web Service has already replied. In both cases an ad-hoc SOAP mes-

sage must be instantiated to send the usage response to the requester.

Now we focus on the execution of the requested algorithms. Samples are filtered using a set of filters as shown in Figure 4 prior to evaluation.



**Figure 4 - Evaluation of algorithms**

The SDS is the first source of Samples. A cascade of filters are connected in such a way that each filter is a source of Samples for the one below. The filter at the top of the stack extracts Samples directly from the SDS. The filter at the bottom of the stack passes out Samples to the algorithms for evaluation. Each filter as well as each algorithm can be initialized with proper parameters. For example, a filter that selects only recent Samples needs to know the maximum age of a Sample if it is to be considered interesting, while a filter that captures Samples of a given service needs to know the name of that service. Furthermore, each filter and each algorithm produce either an output, or a fault message, while the evaluation lasts. For example, a filter may give as output the percentage of Samples that satisfied its filtering criteria. The output produced by all of the involved filters/algorithms is collected to be delivered to the user's that requested the execution.

This way of collecting requests and delivering responses by encapsulating them within SOAP messages on the wire, avoids the need for new messages to be sent over the network. It is clear that when the result of the query is ready to be delivered, this has to happen as soon as possible. Therefore mechanisms to poll for these response messages are needed. The requester should be able to explicitly specify that the response must be delivered at the moment it is ready, or within a short time.

#### 4. Workload Balancing Algorithm

Now we provide details on a possible workload balancing algorithm and information that allows this algorithm to work correctly.

Let us assume one generic service that we call  $S$  and consider a set of  $m$  different HPs labelled with  $HP_1, HP_2, \dots, HP_m$  in which the service  $S$  is already deployed.  $T_i$  is the prediction of the time needed to execute the service  $S$  on the Host Provider  $HP_i$ . We define a fitness function  $K_i = 1/T_i$  to be the in-

verse of the prediction time. In this way, the worse the estimation is, the lower its fitness function is. Let us define  $K_{tot} = \sum_{p=1}^n K_p$  as the sum of the fitness functions for all of the Host Providers. We then define  $P_i = K_i/K_{tot}$  the fraction of the requests for service  $S$  that should be sent to the Host Provider  $HP_i$ . It is very simple to prove that  $\sum_{i=1}^n P_i = 1$  and, hence  $P_i$  can be considered a probability. Each time a request for the Service  $S$  arrives, the request has a probability  $P_1$  of being routed on the Host Provider  $HP_1$ , a probability  $P_2$  of being routed on the Host Provider  $HP_2$ , and so on. Apart from the different fitness functions adopted, the validity of this workload balancing algorithm has been demonstrated by simulations in [6]. This mechanism works very well in environments where changes in workload conditions can not be evaluated at run-time. Value  $T_i$  is indispensable in evaluating the fitness function for a given Host Provider. It cannot be collected on the Service Provider, but needs to be collected locally on each Host Provider. To enable this, each Host Provider must adopt the Framework in a way that allows it to be queried periodically to find out this predicted value calculated via some prediction algorithm.

The adopted prediction algorithm simply evaluates the weighted mean time for a given service over a number of previous executions. It works in a very simple way. Consider a set of  $n$  values labelled  $x_1, x_2, \dots, x_n$  which are collected at the time  $t_1, t_2, \dots, t_n$  respectively. Let us call  $T$  the time in which the algorithm started the execution. We define  $w_i = e^{-\lambda(T-t_i)}$  as the weight given to the value  $x_i$ . It is simple to note that the earlier the value was taken, the less its weight is. The *Weighted Mean Value* for the

set of given values is defined as  $\bar{x} = \frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i}$ . The

older the sample is, the less its importance is. The value of  $\lambda$  is the only parameter of the algorithm and specifies how fast the algorithm "forgets" older values. If  $x_i$  is the time taken to execute the request for the Service  $S$ , then  $\bar{x}$  is the weighted mean execution time for the service  $S$ . Evaluating this value on samples only collected in the last minutes can give us a

good estimation of the time needed to process the next service request.

## 5. Evaluation of Workload Balancing Algorithm

A prototype implementation of the Framework that uses the works exactly as described in the previous sections has been developed using the Tomcat/Axis Web Service Container. Tests have been done on the Giga cluster at the University of Newcastle (24 nodes each with dual hyperthreaded Intel P4 Xeon's at 2.8GHz, 1GB RAM each, 1 Gbit/s switched Ethernet between nodes). In these tests 10 service requests per second are dispatched over two or more Web Service endpoints according to the workload balancing algorithm explained in the previous subsection.

The adopted value of  $\lambda$  is such that Samples 15 seconds old have a weight of 0.5. In all of the tests the time since the test has begun (in minutes) is shown on the x-axis of each chart. Each figure in this section has a chart presenting the granularity of the service logic (the time, in milliseconds, required for processing a message) at a particular Host Provider. The histogram of each figure presents the percentage of request messages submitted to the endpoint where a Web Service has been deployed at a particular point in time. Each time a test starts, the system needs a number of seconds to collect the first workload information from the deployed Web Services. This occurs because at the beginning of each simulation the SDS doesn't contain any Samples, and all query attempts fail, so during this short cold-start time the workload is distributed randomly.

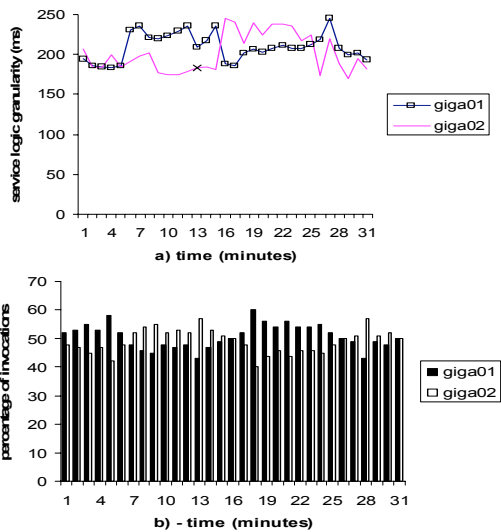


Figure 5 – Web Service deployed on two endpoints

Figure 5 shows the results obtained from the second test. The Web Service is deployed at two different endpoints. It computes the determinant of a 10x10 matrix. Performances of endpoints are similar and consequently the service requests are dispatched in a relatively balanced manner. When the time is between the 3<sup>rd</sup> minute and the 13<sup>th</sup> minute, the giga01 machine takes more time than giga02 to evaluate the determinant, and most of the requests are dispatched onto giga02. Between the 13<sup>th</sup> minute and the 22<sup>nd</sup> minute the situation swaps, and most of the jobs are submitted to the less-loaded giga01 Host Provider.

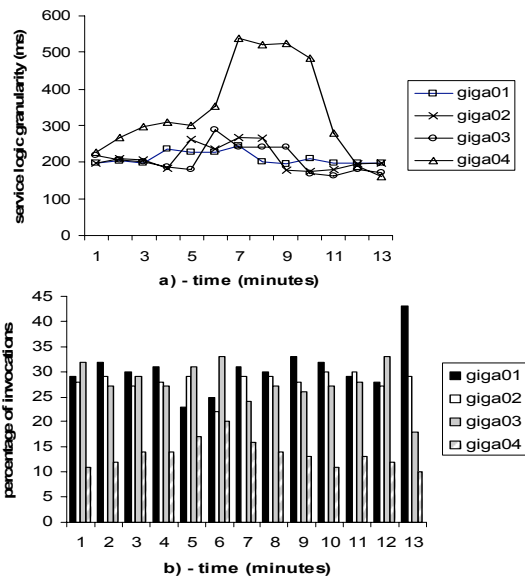


Figure 6 - Web Service deployed on four endpoints

Figure 6 shows the results obtained deploying the same Web Service of the previous case on four different endpoints of the Giga cluster. As shown in Figure 9a, giga04 presents the worst performances along the entire test. Between the 7<sup>th</sup> and 10<sup>th</sup> minute it takes twice the time of the other endpoints to process the message requests. In this case the balancing algorithm results in the routing of the fewest number of messages to giga04.

## 6. Related Work

Many researches are currently working on performance monitoring. Newman et al. described a distributed architecture to monitor a Grid using agents and JINI/JAVA technology based on Web Services [7]. Patel et al. implemented a set of algorithms to evaluate the Quality of Service of a Web Service to help the Workflow Manager to make decisions [8]. Hollingsworth et al. proposed an approach to detect bottlenecks

in parallel systems by monitoring the performance of nodes and then selecting the right performance data to analyse [9]. Tools to support the performance dynamic adaptation in parallel and distributed environments are presented in the literature [10-12]. Many other works in literature deal with the problem of scheduling in distributed systems [9, 13].

While previous works have proposed solutions to the problem of service usage monitoring, scheduling and balancing the workload in a distributed environment, the issue of collecting and analysing service interaction-specific information was not investigated. Our framework focuses on making load-balancing and message routing decisions based on the collection of information related to the message exchange patterns with the deployed services. Additionally, the QoS and policy requirements of the Consumers, the service, and the Service Provider can be considered separately or combined when making a message routing or a deployment decision.

## 7. Conclusions and Future Work

This paper has presented the motivation, design, and evaluation of our prototype implementation of a service-oriented framework for monitoring the usage of Web Services managed by Dynasoar. The results showed us that a Dynasoar Service Provider can utilise information collected from Host Providers in order to make decisions about the routing of messages to deployed Web Services.

Many issues need to be investigated in the future, such as a more detailed messaging system to query the Framework and a set of specifications to describe the syntax of the Samples stored in the SDS, the type of filters and algorithms recognized by the framework, their input and output parameters.

Mechanisms for rights management need to be investigated to prohibit anyone who is not authorized to query the Framework. Malicious people could, in fact, bombard the Framework with requests in order to impair its performance. Moreover, algorithms executed by the Framework might be computationally very intensive. A mechanism to limit resources consumed by the Framework needs to be adopted, for example, to avoid the execution of multiple evaluation algorithms at one time.

Different algorithms can have different end usages and also different priorities. It is clear that an algorithm whose task is the prediction of the execution time for a service must be executed, and the response delivered, as soon as possible. This is not true for algorithms that just provide a weekly report on the usage of the Web Service. The latter are, in fact, less time constrained.

Thus, a policy that gives different priorities to query requests is strongly encouraged.

## References

- [1] Watson P, Fowler C.P., "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. Nottingham, September 19-22 2005 Cox S.J. and Walker D.W. (eds).
- [2] S. Parastatidis, J. Webber, P. Watson, and T. Rischbeck, "WS-GAF: A Framework for Building Grid Applications Using Web Services," *Journal of Concurrency and Computation: Practice and Experience*, vol. 17, pp. 391-417, 2005.
- [3] W3C, "SOAP 1.2 Part 1: Messaging Framework," M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau, and H. F. Nielsen, Eds. <http://www.w3.org/TR/2003/REC-soap12-part1-20030624/>; W3C, 2003.
- [4] OASIS(WS-CAF), "Web Services Context (WS-CTX)." <http://www.iona.com/devcenter/standards/WS-CAF/WSCTX.pdf>.
- [5] W3C, "Web Services Addressing (WS-Addressing)." <http://www.w3.org/2002/ws/addr/>.
- [6] S. Cavalieri, S. Monforte, and F. Scibilia, "Proposing and Evaluating Allocation Algorithms in a Grid Environment," in *Lecture Notes in Computer Science*, vol. 2659, 2003, pp. 1083-1092.
- [7] H. B. Newman, I. C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu, "MonALISA: A Distributed Monitoring Service Architecture," presented at Computing in High Energy and Nuclear Physics, La Jolla, California, USA, 2003.
- [8] C. Patel, K. Supekar, and Y. Lee, "A QoS Oriented Framework for Adaptive Management of Web Service Based Workflows," *Lecture Notes in Computer Science*, vol. 2736, 2003.
- [9] J. K. Hollingsworth and B. P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems," presented at Proc. Seventh ACM International Conference on Supercomputing, New York, 1993.
- [10] F. Bergman, "High Performance Scheduler," in *The grid: blueprint for a new computing infrastructure*: Morgan Kaufmann Publishers, 1998, pp. 279-309.
- [11] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, "Application-Level Scheduling on Distributed Heterogeneous Networks," presented at Conference on High Performance Networking and Computing, Pittsburgh, Pennsylvania, USA, 1996.
- [12] H. D. Karatza, "Simulation of Parallel and Distributed Systems Scheduling," *Applied system simulation: methodologies and applications*, pp. 61-80, 2003.
- [13] A. Helsing, R. Lazarus, W. Wright, and J. Zinky, "Tools and Techniques for Performance Measurement of Large Distributed Multiagent Systems," presented at International Conference on Autonomous Agents, Melbourne, Australia, 2003.