Dynamic handling for cooperating scientific web services

Cushing, R.; Koulouzis, S.; Belloum, A.; Bubak, M.

DOI
10.1109/eScience.2011.40

Publication date
2011

Document Version
Final published version

Published in
Proceedings 2011 IEEE Seventh International Conference on eScience: 5-8 December 2011, Stockholm, Sweden

Dynamic Handling for Cooperating Scientific Web Services

Reginald Cushing*, Spiros Koulouzis*, Adam Belloum*, Marian Bubak*†
**University of Amsterdam, Institute for Informatics, Amsterdam*
†*AGH University of Science and Technology, Department of Computer Science, Kraków*

***Abstract*—Many e-Science applications are increasingly relying on orchestrating workflows of static web services. The static nature of these web services means that workflow management systems have no control over the underlying mechanics of such services. This lack of control manifests itself as a problem when optimizing workflow execution since techniques such as data-locality aware deployment and service-to-service communication are very difficult to achieve. In this paper we propose a novel approach for mobilizing scientific web services onto common distributed resources and as such enable back-to-back communication between cooperating web services, autonomous web service scaling through fuzzy control and autonomous web service workflow orchestration.**

***Keywords*-services; workflows; auto; scaling; autonomous; orchestration; communication; fuzzy**

I. Introduction

E-Science applications are increasingly becoming data-centric and service oriented in nature. The huge volumes of scientific data being produced creates extraordinary challenges for data processing. Current scientific web services are not adequately equipped to deal with this data processing influx. Web services lack the mobility and dynamics to move closer to data sources, direct data communication between services and replication to process data at faster rates. The static nature of current web services also inhibits the use of distributed resources such as grids.

The common approach for utilizing scientific web services is through high-level systems such as Scientific Workflow Management Systems (SWMSs). These systems rely on catalogs such as BioCatalogue [1] for searching and referencing static web services to compose workflows. Typical SWMSs orchestrate web services by iteratively invoking the static services and handling the data transport between services. With complex workflows consisting of many services, the central coordination of huge data just does not scale.

Many web service based e-Science applications can, thus, benefit from an architecture whereby cooperating services can be mobilized, and dynamically scaled. The goals of the proposed framework are: (1) mobilize web services so that they can be deployed anywhere on-demand, (2) choreograph cooperating web services so that they can communicate back-to-back, (3) autonomously orchestrate web service workflows, (4) autonomously scale web services to meet high data load, (5) a non-intrusive approach for deploying current web services to new architectures.

This paper is organized as follows. Section III and IV introduce the architecture that tackle the goals set out above. Section V describes a bio-informatics sequence alignment application that drives the prototype system. This is followed by results from the execution profile of the application. In section II we give an overview of the related work in the area. Section VI discusses future work and concludes.

II. Related Work

Circulate [2] is a web service choreographic and orchestration system which decentralizes web service choreography through a system of proxies which aids the web services to directly talk to each other without going through a central coordinator. Orchestration is still centralized and is only used to control the overall execution. In our system choreography and orchestration are purely decentralized and autonomously achieved by each web service container. Furthermore, communication is achieved through messaging which eliminates the need for proxies. Circulate does not implement any scaling mechanisms for dealing with excess load and neither provides a framework for mobile services.

DynaSched [3] provides a framework for dynamic WSRF service deployment on Grid resources. A central orchestration engine overlooks the whole workflow execution. A scheduler is responsible for deploying services into WS-containers. The WSRF services communicate with files over GridFTP or RFT servers. With dynamic deployment, DynaSched achieves service mobility. It is not clear if deployed services are able to communicate back-to-back over the file based approach. When compared to our approach, DynaSched lacks web service scaling, autonomous orchestration, and possibly back-to-back communication. Furthermore, DynaShed only manages WSRF services which could be a limitation for the system.

ServiceGlobe [4] only aims at dynamic web service deployment with replication and load balancing. ServiceGlobe differentiates between dynamic and static services. The latter being those services which can not be moved around due to some dependency. The architecture relies on a dispatcher which is described as a software-based layer 7 switch. The dispatcher balances the load on a set of replicated web services and can initiate replicas on-demand when the load increases. This system achieves service mobility and scaling although scaling is not based on data load prediction. Since it not a intended for scientific workflows, it lacks web service back-to-back communication and orchestration.

Fuzzy logic has had wide spread use in controlling systems from hardware to software. Some recent works in the area of distributed computing and fuzzy logic include load balancing for a distributed service process engine [5]. In [6] fuzzy logic is used in master/slave application approach on MPI clusters to balance the work units among slaves. The main difference we notice is that our system is composed of multiple simultaneous fuzzy controllers (one for each web service) influencing each other with their outputs thus further adding to the complexity of the control space.

Most of the common workflow system within the scientific community such as Taverna [7], Triana [8], Kepler [9], Pegasus [10], WS-VLAM [11], [12] and GWES [13] focus on orchestrating service-based workflows by contacting statically located services and coordinating the communication between them. This technique involves data being passed through the central coordinator which can easily result in a bottleneck for large web service workflows. The reviewed SWMSs (Table I) do not have any provisions for supporting web service choreography and dynamic deployment. Our system does not intend to be another workflow system since it is not a full SWMS but a subsystem for dealing with dynamic handling of web services. The aim of our system is to extend current SWMSs to ameliorate their feature set with dynamic web service handling.

Table I
O:MISSING X:SUPPORTED

	Orchestration	Mobile	B2B Comm	Auto.Orch	Scaling
Taverna	X	O	O	O	O
Triana	X	O	O	O	O
Kepler	X	O	O	O	O
GWES	X	O	O	O	O
Pegasus	X	O	O	O	O
WS-VLAM	X	O	O	O	O
Circulate	X	X	Proxy	O	O
ServiceGlobe	O	X	O	O	X
DaynaShed	X	X	O	O	O

III. Design Considerations and Architecture

Web services are passive program objects which are hosted in service containers such as Apache Axis2 [14]. Containers are responsible for managing the service lifecycle including starting, stopping and invoking methods. WSDL is a descriptive language that abstractly describes the operations a web service exposes without any knowledge of the underlying implementation. Web services are addressed through an End Point Reference (EPR) which is a location-based addressing scheme using URLs. Common methods for invoking a web service are either using SOAP or REST. In this paper we only consider SOAP services, though the methods can be equally applied to REST services.

The above exposes the first two challenges for realizing our architecture. The EPR system of addressing a web service binds a service to a location using URLs. This hinders

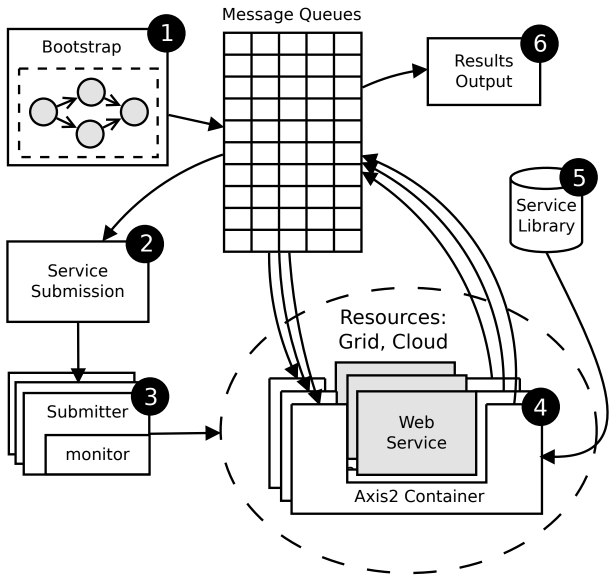


Figure 1. Loosely coupled core modules revolving around the message broker. The Axis2 container is the actual entity that is submitted to resources. The server side components are the message broker, submission service and submitters.

the web service mobility since re-allocating a web service will change its URL. The passive mode of communication means that web services residing behind firewalls, as is the case with the majority of distributed shared resources, have no way of being accessed from outside the network as inbound connections are usually blocked.

Both these challenges are tackled using the same basic idea of messaging. For the communication problem, message queues allow containers to poll and pull SOAP invocation messages off a queue which itself is accessible outside the network. By systematically setting up different message queues for each deployable web service, the queue id becomes the service EPR. Queue ids replace URLs as EPRs and de-localize web services. De-localized services can migrate to different resource without clients being aware of it.

The pull communication model and the location-agnostic service addressing are the basic foundation for our architecture. With these two characteristics, services can be dynamically deployed anywhere on the Internet having at least outbound communication capabilities. Based on the same notion of message passing, the system is further capable of achieving web service back-to-back communication and elastic scaling.

The architecture depicted in Fig. 1 revolves around a message brokering system which loosely couples all other sub modules. The messaging system exposes two types of queues: data transport queues and workflow control queues. Control queues include: a global run-queue where services awaiting execution are queued, an events queue for gathering events from running services and other queues for describing the workflow topology and send commands to web services.

Fig. 1 illustrates the steps in which a web service based workflow is orchestrated. In step **1**, a workflow is bootstrapped. In bootstrapping, the first web service is put on the run queue and the web service connections are also made available on the messaging system. These connections allows web services to autonomously know to whom they are connected which in turn allows back-to-back communication. In step **2** the service submission picks up the bootstrapped service and submits it to one of the configured resource submitters step **3**. Submitters abstract the actual resources and are responsible for monitoring the available free slots on the resources.

In step **4** a submitted service container lands on a worker node. The service container is initially void of a web service to host. The first step for the container is to check the message queue for any available web service to host. Runnable services are loaded from the service library step **5**. The container then starts consuming SOAP messages from the designated queues on the messaging system and pushes them up to the web service.

Web service containers can deduce the neighboring workflow topology for the hosted services through control queues on the messaging system. This information is used by the container to transform SOAP output messages directly to input messages for connected web services. On the fly SOAP transformation allows web service back-to-back communication. The transformed SOAP messages are transported through the messaging system which unburdens the central SWMS from coordinating communication between services.

Web service orchestration is modeled on a dataflow approach. This model dictates that only those web services having input data to consume may become active. The advantages of such a model is that resources are not waisted by idling services. Furthermore, by combining dataflow models with messaging back-ends, communication between services is decoupled in time. This reduces the need for co-allocating resource which have been shown to degrade a system due to increased run queue waiting times [15], [16].

A. Technologies Used

The central messaging system is the Apache ActiveMQ [17]. ActiveMQ is an enterprise messaging system with many features that can be used to tune the performance of the architecture. Noticeable features include; fail-over setup where web services can connect to different brokers in the event that one fails, and networks of brokers where messages travel from one broker to the next until they reach a consumer.

The web service container used in this architecture is the Apache Axis2 [14]. Axis2 is a lightweight extensible container suitable for submitting container-level jobs. The proposed architecture relies heavily on the modifications to the default Axis2 container. Although Axis2 supports both SOAP and REST services, we only tackled SOAP services in our implementation.

The web service library is a simple HTTP server where web service bundles are kept. For the rest of the architecture including bootstrapping, submission, and results client, Java was used as the programming language of choice.

IV. Container Architecture

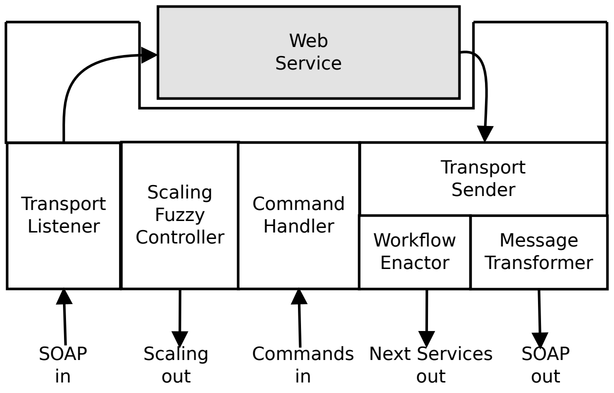


Figure 2. Modified Axis2 container including transport handlers for pulling messages, autonomous workflow enactor, fuzzy controlled scaling, message transformer, and a command handler

Most of the management routines reside inside the Axis2 container which executes alongside the web service on worker nodes. The modified Axis2 container transforms a traditional web service into a mobile object with smart orchestration and scaling mechanisms. Fig. 2 highlights the main components added to a standard Axis2 container. The customized transport handlers are the main entry and exit points for the web service. The workflow enactor component implements the autonomous orchestration (Section IV-B). The message transformer implements back-to-back communication (Section IV-A) and the fuzzy controller implements autonomous scaling (Section IV-C). The command handler consumes command messages from the control queues.

Fig. 3 illustrates the round trip path of events for a SOAP message through the container. On reception of a SOAP message the transport listener processes the message such as adding timestamp information and moves it up the to the Axis2 stack. The container unmarshals the SOAP message and invokes the web service method with the parameters extracted from the message. The container stack returns the SOAP response to the transport sender. The latter updates the message round trip time. Message round trip times are used by the service replication routine to deduce the load on the service. The message transformer transforms response messages to input SOAP messages for successor services. The workflow enactor checks if any successor services need to be initiated. Finally, the transport sender sends out the transformed messages or the default response message if no transformation took place to the designated queues. The fuzzy controller and the command handler continuously execute in a loop. The former elastically scales the service instances while the latter listens for commands.

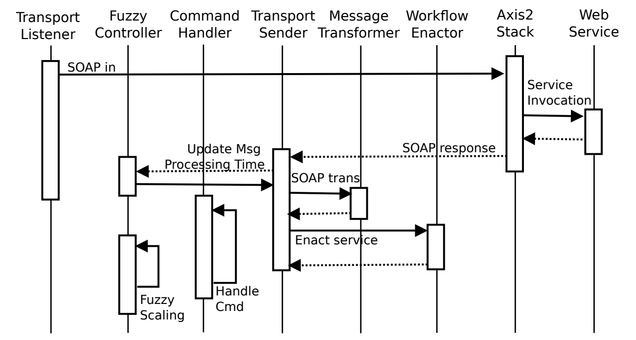


Figure 3. Sequence of SOAP message processing on the modified Axis2 container. Most of the processing is done on the SOAP response path as it triggers the message transformer, workflow enactor and fuzzy controller.

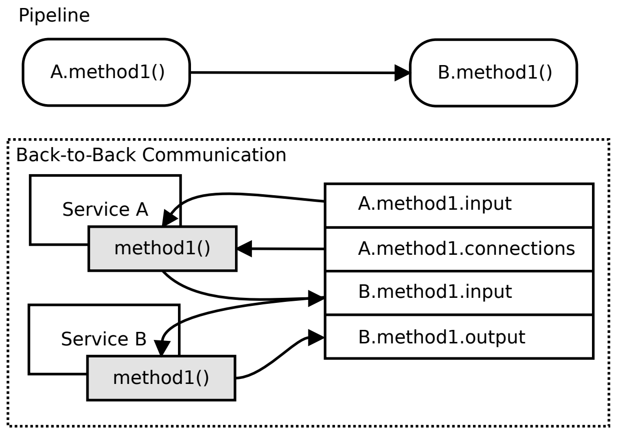


Figure 4. Back-to-back communication for a two service pipeline. Service *A* knows about the connection between *A*.method1() and *B*.method1() through the connections queue. Any output from *A*.method1() is sent directly to *B*.method1().

A. Back-to-back Communication

In cooperating web services such as those in pipelines or workflows it is often advantageous to allow web services to directly talk to each other without the need for a client to coordinate the communication. This is especially the case for complex workflows where it is not feasible to manage all the inter-service communication. For this reason the modified Axis2 container allows web services to directly talk to each other through the message broker.

At the bootstrapping stage, the topology of the workflow is known. The topology is synthesized into messages on dedicated connection queues thus, in the pipeline topology depicted in Fig. 4, there exists a connection between *A*.method1() and *B*.method1(). The connection would translate into a message on *A*.method1.connections queue. This designated queue is used by service *A* to deduce to whom it is connected hence giving *A* the knowledge of its neighbors. The messages on the connections queue describe the SOAP template expected by the successor (in this case *B*.method1()).

When *A*.method1() returns a SOAP message it is picked up by the message transformer inside the Axis2 container (see Fig. 2). The SOAP template present on the connections queue is used to transform the response message from *A*.method1() to the input of *B*.method1(). This transformed message is then written directly to *B*.method1.input queue by the transport sender for *A*.method1().

Since *B*.method1() has no successor connections, any output by this method is written to the method's default output queue *B*.method1.output which can then be consumed by a client waiting for output from the pipeline.

In the scenario of a fan-in topology, multiple services connected to *B*, write their messages to the same input queue for *B*. Similarly in a fan-out approach where *A* is connected to multiple services, the message transformer transforms the SOAP response for each successor. In both cases message ordering is not guaranteed but can be accomplished through message sequence ids on the container. Message ordering is a very expensive operation and can lead to memory exhaustion due to buffering messages in order to re-sequence them.

B. Autonomous Orchestration

The connections queues described for back-to-back communication are also used to enable autonomous orchestration. Connections between services represent a data dependency thus from Fig. 4 service *B* is data dependent on service *A*. The dataflow model approach dictates that service *B* should only become active when it has data to process. This model is enforced autonomously by the individual containers.

From Fig. 4, *A*.method1() produces data for *B*.method1(). This satisfies the dataflow model that *B*.method1() should become active since data is now available. The workflow enactor component in the container for service *A* can deduce if any instance of *B* is running. This is done by checking the number of consumers on *B*.method1.input. If no consumer is active on the queue, the workflow enactor submits and instance of *B* to the global run queue. The instance is picked up by the service submission (Fig. 1) and submitter to a resource. In the case that service *A* has multiple successors the procedure is repeated for every successor. This approach differs from the common scenario of having a central SWMS which has to orchestrate the whole workflow. Typical SWMS are far-sighted i.e. they have knowledge of the whole workflow and hence have to maintain the whole workflow which could be a limitation for large complex workflows. With autonomous orchestration, services are myopic as they only have knowledge of their immediate successors thus no central entity is coordinating the whole workflow execution.

C. Fuzzy Controlled Elastic Scaling

A characteristic of many e-Science applications is that they are embarrassingly parallel and therefore can be easily scaled up with simple data partitioning techniques. The main goal of partitioning an embarrassingly parallel application is to achieve better throughput and hence reduce the makespan. This is usually done in a greedy manner where the application consumes as many resources as possible to reduce overall execution time. This premise is not always an ideal solution when dealing with cooperating tasks since one's greed to consume as many resources as possible will result in starvation for other tasks in the workflow.

Starving tasks can degrade the whole system because progress is hampered and data gets piled up waiting to be processed. For this reason we propose a fuzzy controlled elastic scaling mechanism for individual services taking part in a workflow. The fuzzy controller can autonomously scale up and down the service depending on the predicted service load and the resource load.

Through the messaging system web services can be replicated as many times as needed. Every instance of the same web service is attached to the same queues. From Fig. 4, if multiple instances of service *A* are initiated then all instances read data from *A*.method1.input thus the input data is said to be partitioned amongst all instances of the same service. This implements data parallelism. The assumption here is that there is no causal dependency between data messages on the input queue as this would impede data partitioning. Similarly, all instances of the same service write data to the same output queues. When data parallelism is not possible such as services that need the whole data set to accomplish their task can not exploit such replication and would have their fuzzy controller disabled.

Within a single workflow, cooperating services are competing for resources. This is especially evident when the resource pool is apparently finite as would be the case in many distributed shared resources. Thus to achieve adequate workflow progress, services must not replicate themselves greedily when not enough resources are available. Conversely, service scaling must take an abstemious approach to resource consumption so as to guarantee whole workflow progression. Such an approach is implemented by means of a fuzzy controller whereby each Axis2 container runs a fuzzy controller for each hosted service to scale up or down the replicated instances of the same service. The bases of the controller is that a web service should be able to aggressively replicate itself when its load is high and resources are free but scale down when its load diminishes and the resource are quite occupied. The latter is intended to make space for other services to take hold of the resources.

The decision of *when a task is overloaded* or *enough resource are available* is difficult to simplify using a simple thresholds since service load and resource load are very dynamic especially when cooperating service are influencing each others view of the load. For this reason, calculating the scaling factor of a service such that it does not overuse the resources but at the same time does not under utilize them is a problem well suited for fuzzy logic. In fuzzy logic, terms like *high load* do not represent a single threshold but a range of thresholds with varying membership probabilities.

Fig. 5 illustrates the inputs (taskLoad, resourceLoad) and output (replication) for the fuzzy controller. The taskLoad input defines a set of fuzzy membership function for the terms *very_low, low, ideal, high,* and *very_high*. Similarly the same terms are defined for the resourceLoad. The output from the fuzzy controller is the scaling count which ranges from -15 to 15 so if the output is -10 then the number of instances for a particular service should be scaled down by 10. These adjustment are done at timed intervals hence the controlling is progressive. The fuzzy output defines membership functions for controlling how aggressive the scaling should be done hence terms like *positive_aggressive, positive_slow, negative_aggressive,* and *negative_slow* are defined.

The taskLoad defines the web service load and is a prediction-based load calculation. Given that at any point in time we know the input queue size and the average message processing time, we try to predict the total processing time for the whole input data queue. For every message that exits the container, the average message processing time is update. A message processing time T_i is defined as the round trip time from when the message enters the container up till it exits the container hence $T_i = (t_i^{out} - t_i^{in})$.

The mean message processing time T_i^{avg} is defined as the weighted mean of the current and last message processing time hence $T_i^{avg} = (T_i^{current} w_k + T_{i-1}^{avg} w_p)$, where $i > 0$, $T_1^{avg} = 0$ and $w_k + w_p = 1$. The weights w_k and w_p are always set to favor the highest load therefore if $T_i^{current} > T_{i-1}^{avg}$ then $T_i^{current}$ has a higher weighting and vice-versa. This smooths out *flip-flop* scenarios when the message processing time continuously fluctuates between a high and a low. Favoring the highest message processing time in the weighted mean ensures that an increase in load is rapidly evident while a decrease in load is gradual. Having calculated T_i^{avg}, the predicted processing time P_i for the whole message queue is then calculated as $P_i = (T_i^{avg} \times S_i)$ where S_i is the input queue size at the moment of calculation.

Given a time quantum Q for a web service which could either be derived from a budget to use a resource or an allocated time quantum by a resource manager, the web service load can be calculated as $L_i = P_i/(Q - E_i)$ where E_i is the elapsed time since the web service started. When $L_i \approx 1$ the service is in an ideal load since it should manage to process all the data within the allocated time quantum. A load much lower than 1 indicates the web service is underloaded while a load much greater than 1 indicates the web service is overloaded. L_i is the input value for the taskLoad in the fuzzy controller.

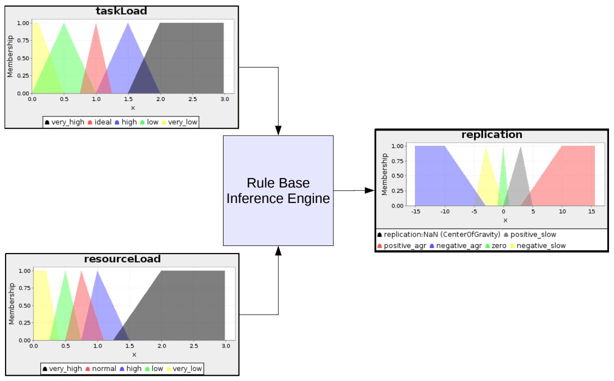


Figure 5. Fuzzy controller membership functions for inputs ant output

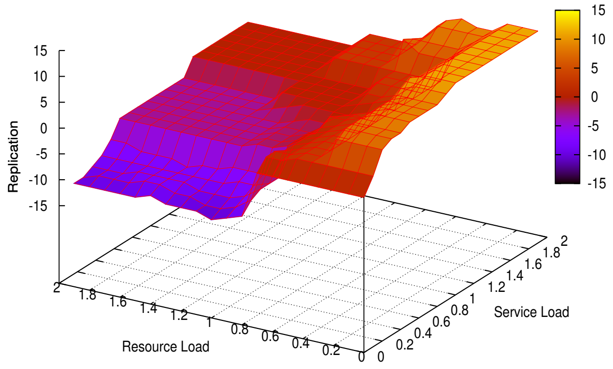


Figure 6. Scaling fuzzy controller surface plot

The resourceLoad R_i is defined as a ratio $R_i = (U_i + W_i)/A_i$ where U_i is the amount of used resources, W_i is amount of queued tasks waiting for a free resource and A_i is total available slots. A resource pool is fully used when $R_i = 1$. When $R_i > 1$, the resources are overbooked since a number of tasks are queued waiting for a free slot. R_i is the second input to the fuzzy controller.

We refer to the set of all replicated instances of the same service as the service farm. Service replication routine is restricted to one per service farm. The designated service instance which is currently responsible for running the fuzzy controller is referred to as the master of the service farm. Since the size of the service farm starts out as one, the first service is automatically elected to a master. The service knows its the only instance running by querying its own input queue and query the number of consumers on the queue. Subsequent instances created by the master do not, themselves, become masters since they are note the sole consumers on their queue. Before a master terminates it relinquishes it own mastership by putting a master token on its own command queue.

Since all instances are consumers on the same command queue and the message broker guarantees that only one instance will consume the message, the instances that gets hold of the master token elects itself as the new master. If a master abruptly dies without relinquishing its mastership then the only way a new master is elected is when the service farm is reduced back to one. A better solution, although not implemented, is for the master to elect a secondary master who will periodically challenge the mastership by sending a command to all instances asking who is the master. If no master replies then it takes over the mastership and relinquishes the secondary master.

Fig. 6 shows illustrates the surface plot for taskLoad L and resourceLoad R. The output, replication indicates how to scale the number of services. The plot is derived from a set of fuzzy rules such as:

IF *taskLoad* **IS** *very_high* **AND** *resourceLoad* **IS** *very_low* **THEN** *replication* **IS** *positive_aggressive;*
IF *taskLoad* **IS** *very_low* **AND** *resourceLoad* **IS** *high* **THEN** *replication* **IS** *negative_aggressive;*

V. Experiments and Results

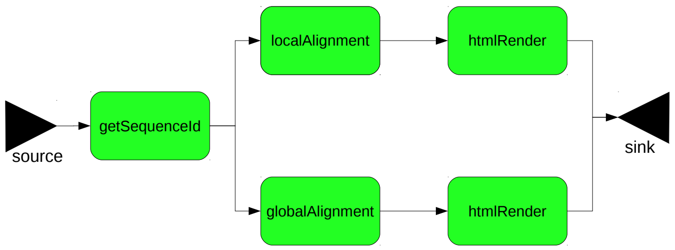
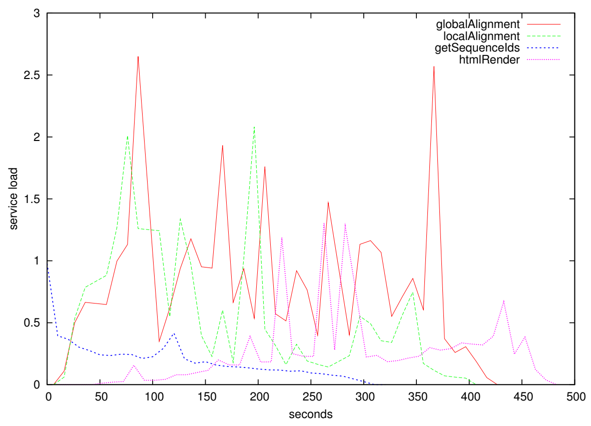


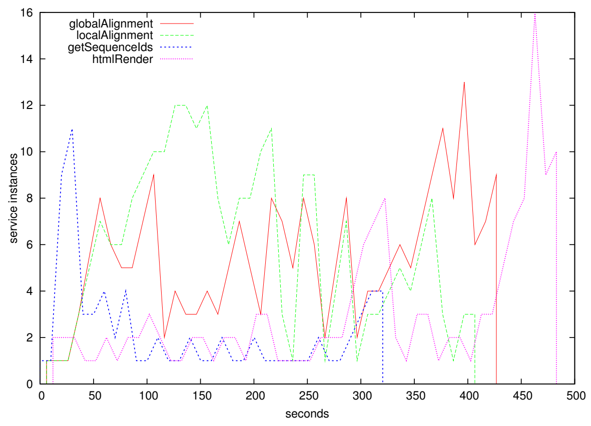
Figure 7. BioJava sequence alignment workflow

The workflow depicted in Fig. 7 illustrates a typical bioinforamtics workflow. The workflow consists of two independent pipelines. The pipelines compute sequence alignment using data supplied by the UniProtKB [18]. Each component is a SOAP Axis2 web service. The source represents the bootstrapping component while the sink represents the result gathering client. The workflow is induced with 22550 alignments for each pipeline therefore the whole workflow computes 45100 alignments. The `getSequenceIds` web service reads a list of sequence ids and returns the actual sequence data for the ids. `localAlignment` performs a local alignment on the passed sequences while `globalAlignment` performs a global alignment. Both alignment web services use the BioJava API [19] for processing the biological data. `htmlRender` transforms the results into HTML tags which are then made accessible through a web browser. The sink concatenates the results into HTML pages.

As a resource pool back-end we had access to the Distributed ASCI SuperComputer 3 (DAS3) which is five wide area distributed system. For purpose of testing the resource competitiveness between web services we used a single 29 node cluster from the University of Amsterdam (UvA). The UvA cluster nodes each have 2 2.2GHz AMD Opteron DP275 processors with 4GB of main memory.



(a) Web service load for all workflow components. A service load of 1 means that the web service is expected to complete its task within the specified time, service load of 2 means it will take twice as much to complete.



(b) Number of web service instances running for each workflow module at any given time.

Figure 8. Results showing the calculated service load 8(a) and the number of web service instances initialized by the fuzzy controller 8(b) to control the service load.

The results in Fig. 8(a) and Fig.8(b) illustrate the execution pattern of the workflow in Fig. 7. Fig. 8(a) shows predicted input load for each web service during the execution lifetime at intervals of 5 seconds. Spikes in the load graph signify when a considerable amount of data has been queued on the web service input queues. The spikes in the service load are short lived since the fuzzy controller immediately responds by initiating multiple instances to deal with the increased load. The response to the service load spikes is illustrated in Fig. 8(b) which shows the number of web service instances simultaneously running at any particular time. Thus spikes in the service-load graph 8(a) are shortly followed by spikes in the service-instances graph 8(b).

Dissecting some notable regions within these results we can note that at the beginning of the execution `getSequenceIds` starts with a load close to 1. Since no other web service is running at this stage, the fuzzy controller does not waist time and aggressively scales the service up. This can be noted with a spike in 8(b). With the autonomous orchestration feature, as soon as `getSequenceIds` produces output it also initiates its dependent successors. Since `getSequenceIds` produces output for both `localAlignment` and `globalAlignment`, the multiple instances immediately increase the input load on both these web services. The spikes for the simultaneous load increase is illustrated between the 50-100 second mark in 8(a). As expected, the fuzzy controllers take action and respond by replicating the instances. At this point the controller on `getSequenceIds` senses the increase in resource load and also notes its own load has diminished hence it downscale itself to make way for other services. Whilst still having a light load, `getSequenceIds` will tentatively replicate itself slowly when it detects dips in resource usage. This can be noted in the region 100-200 seconds in 8(b) where sudden dips by `globalAlignment` result in slow increase by both `getSequenceIds` and `htmlRender` simultaneously.

As was the case for `getSequenceIds` at the start of execution, a relative small spike in the load for `htmlRender` at the end of execution triggers an aggressive replication since it is the only running web service at that time. These results show that the workflow of cooperating web services cooperate on three fronts: cooperation through communication, cooperation through orchestration, and cooperation through fair resource usage. During the whole execution, the load on the resources was at an average of 72%. This is very close to the ideal with regards to the fuzzy controller where 75% had the highest probability in the *normal* membership function.

VI. Conclusion and Future Work

Through this architecture and its implementation we have shown how web services can be handled dynamically. Web services have been made mobile by using queue ids as their EPR instead of the URL based EPR. A pull model allows web service to be deployed deep within a network. Back-to-back communication has been achieved through a system of message brokering. Also, autonomous scaling has been achieved using fuzzy controllers. Autonomous orchestration has been achieved with web services containers having myopic view on the workflow. The implementation of the architecture demonstrated that the above attributes to dynamic web service handling can be achieved in a nonintrusive manner thus not modifying the actual web service code.Through the described architecture and the obtained results we have shown that intelligence within service containers can transform a web service into mobile, replicable, and cooperative service.

A limitation in the current system is that all SOAP data is passed through the message broker. This is not ideal as it puts a lot of strain on the broker which has to keep track of the huge amounts of data in the messages. A better solution is to have the Axis2 container automatically swap out the outgoing large SOAP data with a reference. The container can distribute the SOAP messages to reliable dedicated data stores. The message sent to the broker would only contain the reference to the data and the protocol to access it. When a containers receives a message with a reference, it can automatically get the SOAP data and push it up to the web service. The same techniques applied to SOAP web services can also be applied to REST services. With minor modifications to the Axis2 container, REST services can also be handled dynamically in the same manner as SOAP services.

Another area of interest is the possibility for peer-to-peer web service communication. Rather than a technical problem it is more of a security problem due to access restriction on inter-cluster communication. The possibility here is for a web service to detect which services are running on the same network and then open direct socket connections between services. This capability would lead to another challenge that is locality scheduling. By profiling inter-service communication a scheduler can decide to move both services on the same network so that they can open direct socket connections and therefore improve communication.

Acknowledgment

This research was partially funded by the COMMIT project [1].

References

[1] "BioCatalogue," http://www.biocatalogue.org.

[2] A. Barker, J. Weissman, and J. van Hemert, "The Circulate architecture: avoiding workflow bottlenecks caused by centralised orchestration," *Cluster Computing*, vol. 12, pp. 221–235.

[3] S. Shahand, S. J. Turner, W. Cai, and M. K. H., "DynaSched: a dynamic web service scheduling and deployment framework for data-intensive grid workflows," *Procedia Computer Science*, vol. 1, no. 1, pp. 593 – 602, 2010, iCCS 2010.

[1] www.commit-nl.nl

[4] M. Keidl, S. Seltzsam, and A. Kemper, "Reliable web service execution and deployment in dynamic environments," in *In Proceedings of the International Workshop on Technologies for E-Services (TES*, 2003, pp. 104–118.

[5] J. Cao, H. Zhao, and M. Li, "A fuzzy rule based load balancing model for a distributed service process engine," in *Grid and Pervasive Computing Workshops, 2008. GPC Workshops '08. The 3rd International Conference on*, may 2008, pp. 9 –14.

[6] S. Sanchez-Solano, A. Cabrera, I. Baturone, F. Moreno-Velo, and M. Brox, "Fpga implementation of embedded fuzzy controllers for robotic applications," *Industrial Electronics, IEEE Transactions on*, vol. 54, no. 4, pp. 1937 –1945, aug. 2007.

[7] D. Hull, *et al.*, "Taverna: a tool for building and running workflows of services," *Nucleic Acids Research*, vol. 34, no. Web Server issue, pp. W729–W732, 2006.

[8] A. Harrison, I. Taylor, I. Wang, and M. Shields, "WS-RF workflow in Triana," *Int. J. High Perform. Comput. Appl.*, vol. 22, no. 3, pp. 268–283, 2008.

[9] I. Altintas *et al.*, "Kepler: an extensible system for design and execution of scientific workflows," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, june 2004, pp. 423 – 424.

[10] E. Deelman *et al.*, "Pegasus: Mapping scientific workflows onto the grid," in *Grid Computing*, ser. Lecture Notes in Computer Science, M. Dikaiakos, Ed. Springer Berlin / Heidelberg, 2004, vol. 3165, pp. 131–140.

[11] V. Korkhov, D. Vasyunin, A. Wibisono, V. Guevara-Masis, A. Belloum, C. de Laat, P. Adriaans, and L. Hertzberger, "Ws-vlam: towards a scalable workflow system on the grid," in *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*. New York, NY, USA: ACM, 2007, pp. 63–68.

[12] A. Belloum, M. Inda, D. Vasunin, V. Korkhov, Z. Zhao, H. Rauwerda, T. Breit, M. Bubak, and L. Hertzberger, "Collaborative e-science experiments and scientific workflows," *Internet Computing, IEEE*, vol. 15, no. 4, pp. 39 –47, july-aug. 2011.

[13] A. Hoheisel, "User tools and languages for graph-based grid workflows: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1101–1113, 2006.

[14] "Axis2," http://axis.apache.org.

[15] E. Elmroth, F. Hernndez, and J. Tordsson, "Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment," *Future Generation Computer Systems*, vol. 26, no. 2, pp. 245 – 256, 2010.

[16] W. Smith, I. Foster, and V. Taylor, "Scheduling with advanced reservations," in *Proceedings of IPDPS*, 2000, pp. 127–132.

[17] "ActiveMQ," http://activemq.apache.org.

[18] "UniProtKB," http://www.uniprot.org.

[19] "BioJava API," http://biojava.org.