A Scalable Self-Managing Architecture for WSRF Services

Christoph Reich
Department of Computer Science
Hochschule Furtwangen University, Germany
reich@hs-furtwangen.de

Kris Bubendorfer
School of Mathematics, Statistics and Computer Science
Victoria University of Wellington, New Zealand
kris@mcs.vuw.ac.nz

Rajkumar Buyya
Grid Computing and Distributed Systems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
University of Melbourne, Australia
raj@csse.unimelb.edu.au

16 April 2007

Abstract

Service-Oriented Architectures provide integration of and interoptablity for independent and loosely coupled services. Web services and the WSRF standards are frequently used to realise such Service-Oriented Architectures. In such systems, autonomic principles of self-configuration, self-optimisation, self-healing and self-adapting are desirable to ease management and improve robustness. In this paper we focus on the extension of the self management and autonomic behaviour of a WSRF container to include its interconnection and interaction with other WSRF containers. In our architecture we incorporate a structured distributed hash table peer-to-peer overlay network within our autonomic WSRF service container. This approach enables the runtime construction of a scalable self-managing and self-configuring SOA system. Our architecture is inherently non-hierarchical and widely distributed. It enables reliable resolution of web service locations at any application server, allows balanced runtime deployment of WSRF services, migration to enable service level agreements to be satisfied and provides autonomic management of the overall system.

1 Introduction

Webservices and the associated WSRF [22] standards are the predominant choice for implementing service oriented architectures (SOA). Management of large SOAs is difficult, and autonomic principles of self-configuration, self-optimisation, self-healing and self-adapting [18, 19, 16] can be usefully applied to ease management and improve resilience and overall system performance. In addition, quality of service (QoS) needs to be expressed in such SOAs and can only be met if specific service level agreements (SLAs) are defined and adhered to. To this end we have developed an autonomic WSRF container that utilises MAPE (Monitor, Analyse, Plan and Execute) [6] to manage its internal functionality, detect SLA violations and trigger corrective actions.

However, this autonomic WSRF container is only part of the solution, as these containers need to be organised into a distributed network to enable deployment of distributed web services. Our approach is to extend the self management and autonomic behaviour of the WSRF container to include its interconnection and interaction with other WSRF containers. Hierarchical approaches such as [9] do not naturally fit with autonomic principles, as to be truly self managing, each WSRF node must be able to contribute to the management of the overlay network and yet avoid any specialised or static roles. These requirements fit well with recent advances in peer-to-peer networking [20, 21] and such systems typically permit a wide distribution of workload, decentralised management, failure tolerance and replica management. In our architecture we incorporate such a structured distributed hash table (DHT) peer-to-peer overlay network within our autonomic WSRF service container. Our work stands neatly at the junction of four areas of current research: service-oriented architectures, web services and the WSRF, service level agreements, and structured peer-to-peer overlay networks. Our architecture enables reliable resolution of web service locations at any application server, allows SLA compliant deployment of WSRF services, and provides autonomic management of the overall system.

Our contributions: firstly we have provided a scalable decentralised solution to the deployment of distributed web services, secondly we have simplified the management of such a system by adhering to autonomic principles, thirdly we maintain the performance of the system by tightly integrating SLA compliance and migrating service between containers to preserve QoS. In addition, as the architecture has been built using standardised modern technologies and with high levels of transparency, conventional webservices can be deployed with the addition of a SLA specification.

The paper is organised as follows, in section 2 we outline the basics of the autonomic WSRF container, in section 3 we describe the architectural design of the system, section 4 details the prototype and our experimental results that validate our approach, section 5 explores related work, and finally section 6 concludes this paper.

2 WSRF Container

In this paper we focus on the architecture for interconnecting our autonomic WSRF containers, and therefore we will only provide a brief outline of the WSRF service Container itself, see figure 1. The WSRF container is embedded within a Geronimo [2] application server, the WSRF services are deployed in Axis2 [1] running in Tomcat [3]. JSR-77 [15]provided by JMX [14] is used to monitor the WSRF services inside the service container (e.g. request counter, processing time, etc.). Remote access is provided by a management Web service (preferred) or by the RMI remote adapter from JMX. The management Web service contacts MBeans [14] to get or set management information, or to define policies, etc. MAPE [6] is an architecture for implementing such autonomic features and utilises SLAs and performance metrics to trigger self managing operations and is implemented using Geronimo's GBeans [12]. GBeans automatically generate MBeans, which are used by the management Web service. Using GBeans provides access to Geronimo's advanced features, such as, Inversion-of-Control [10].

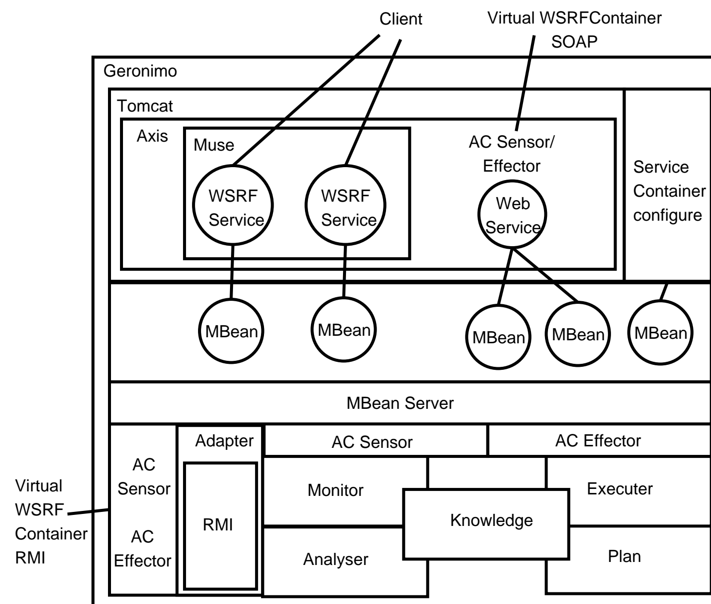

Figure 1: Internal structure of the WSRF container

Each container monitors its performance requirements, based on the SLAs it holds. If a container is not able to resolve an actual or a predicted SLA violation internally, it will generate a *help* message indicating which resource is causing the problem. The exact means of distributing this message is detailed in the next section 3, however, the important point here is that each recipient of this help

message will generate a response health status metric (H-metric). A H-metric is a simple approximation indicating the overall status of the WSRF container, and is weighted to highlight the resource responsible for the SLA violation. Essentially, each monitored resource is normalised, then all of the resources are summed and renormalised. This allows the state of the responding machine to be summarised in a single comparable number, but permits the particular resource of interest to carry more weight when selecting a destination for migration. The H-metric is specific to each help request. Two simultaneous requests with different violating resources, will result in two different H-metrics from the same container. The container with the lowest H-Metric, will typically be chosen as the new location of the service.

3 Architecture

The WSRF container as presented, utilises MAPE to manage its internal functionality, to detect SLA violations and trigger corrective actions. However, we saw an opportunity to extend the self management and autonomic behaviour of the WSRF container to include its interconnection and interaction with other WSRF containers. To build a truly scalable, flexible, self configuring autonomic WSRF overlay network, each WSRF node must be able to contribute to the management of the overlay network and yet avoid any specialised or static roles. The lack of specialised or static roles increases robustness in case of failure, and most importantly permits a wide distribution of workload. In particular we want services to be locatable within the network without relying on a centralized index server. This will avoid any single point of failure in the system. Structured peer to peer file systems that use distributed hash tables, such as Pastry [20] and Chord [21], are a good fit with these design requirements and naturally express some desirable autonomic behaviours. To store data each data object is mapped to the numerically closest live storage node (both node ID and data object ID are hashed). Thus, each storage node in the network is responsible for storing objects with numerically close IDs, and the resulting load is distributed evenly. However, such p2p file systems need adaptation and extension to provide the functionality required for interconnecting our autonomic WSRF containers. Figure 2 illustrates the extension of the WSRF container to integrate a customized p2p node and routing servlet within the Geronimo application server. The modified p2p Node stores a mapping of Service ID to Container ID. The actual service is not deployed on the container to which the service ID is hashed – otherwise services could not be migrated or deployed as required and transparency would be broken.

The following subsections 3.1 through 3.5 outline our architecture based on the operating phases of the system, e.g. service access, creation, deployment, and maintenance (observation and migration).

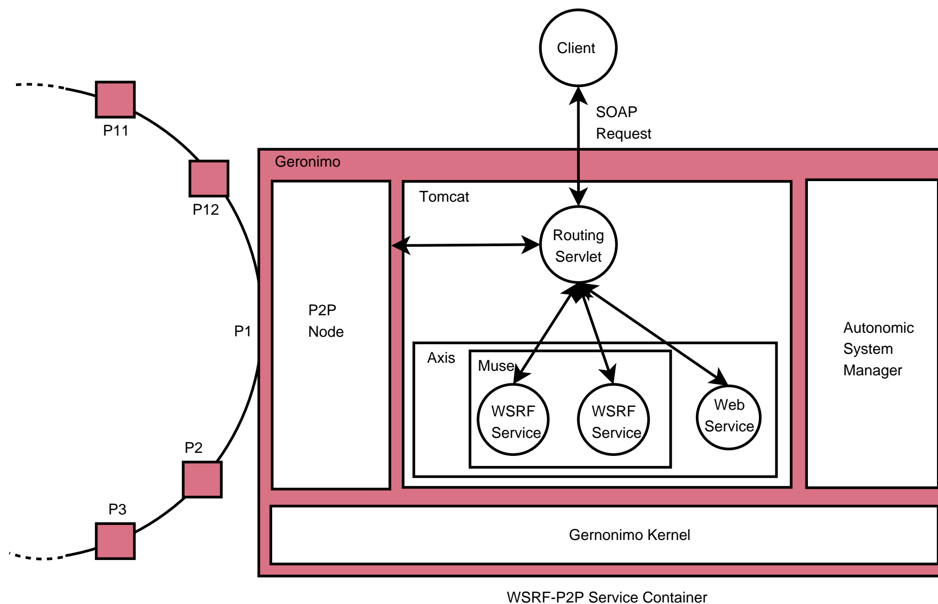

Figure 2: Architecture overview of the WSRF-p2p service container

3.1 WSRF Service Access

To obtain a service, the client only needs to know a single container in the WSRF container overlay network and the service's ID (name). Communication can take place either directly or via a local proxy. One advantage of the proxy is that it generates a random container ID and uses it to bind to that container. This step ensures that the workload is evenly distributed over the network due to the even randomisation properties of the DHT hash. Also, in the case of a failure of the container, the proxy will automatically rebind to a new randomly selected container. A typical request is shown in Figure 3. The client instantiates a proxy, passing in a container address and the service name. The proxy then picks a random ID and passes this to the known container. The known container resolves this address to the nearest live container and routes the service request to it. The address of the container is returned to the proxy and the proxy binds to it. From this point all queries go via the bound container. The service name is hashed, and the lookup is routed via the bound proxy to the container that holds the index for this service. The value returned is the actual container in which the service is hosted, the routing servlet then uses this to obtain the service via the routing servlet on the hosting container. Any changes to the distribution of services, or rebinding of the container to proxy take place transparently for the client.

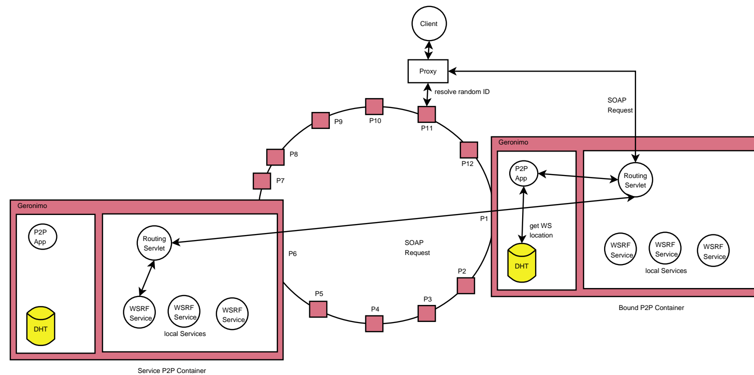

Figure 3: Client request routing

3.2 Creation and Initialisation Phase

The first WSRF container in the network is simply created. All subsequent containers that join the network are passed the ID of any existing container, which they then use to bootstrap themselves into the network. When more than one WSRF container is present, the degree to which the resources are normalised during the calculation of the H-metric needs to be common throughout the network. Otherwise the H-metrics of the various containers cannot be compared. As we do not assume a homogeneous set of machines, this information must be exchanged as each container joins. If the new container advises that it has more memory or a higher MIPS performance than the current maximums, then its values are selected as the new maximums for normalisation and are propagated to all containers in the network.

A container can join a new network, or an existing network. The difference between the two situations is that a new network will not have any pre-existing services deployed, whereas the existing network will have deployed services and runtime information stored in the containers. On a new network these is nothing else for the container to do other than wait for service deployment requests. However, when joining an existing network a new container must do additional work. First, it asks its new neighbours (left and right) what their observed (see section 3.5.1 for details) state of the load in the system is, it averages these and stores the result as it's view of the load on the overlay network. If this observed load is greater than some minimal threshold, the container queries (see section 3.3) the overlay network for an overloaded container that might wish to migrate a service to it. The container ceases its queries once it has a service migrated to it, and then starts operation as a normal member of the network. Additional applications are only migrated to the container in response to help requests or by new service deployments (see Section 3.4).

To summarise, to join an WSRF-p2p network all that a container needs to know is the ID of any member. After joining, a container knows the maximum

resource values present in the network, and how its direct neighbours view the average load of the system.

3.3 H-metric Queries

One aspect that initialisation, deployment and migration share is the need to query the WSRF-p2p container network to locate overloaded or underloaded containers. This is completely different to the problem of service location. The problem is that in a DHT ring overlay network there is no hierarchy to determine how the nodes should be queried for their H-Metrics. As we desire a scalable overlay network, solutions such as broadcast and a sequential ring traversal are not acceptable. Our solution is to treat the ID space of the nodes as evenly populated (which it is, given a reasonable hash function) and to compute a binary query tree with the initiating node as the root (for deployment we pick a random node as root to best balance the load over multiple deployments). Equation 1 gives the formula for the binary query as computed at each level.

$$newID_l = \begin{cases} \{ownID\}, & \text{if } l = 0.0 \\ \{ownID + \frac{maxID-of-ring}{2*l} * i | i = 1, 2, 3, ..., (2*l-1)\}, & \text{otherwise} \end{cases} \quad (1)$$

The search is then parameterised by depth and the H-metrics from the queried containers are returned. Due to the computation and the randomisation of container IDs within the DHT, duplicate queries can target the same node. This is resolved by utilising version numbers and discarding duplicate requests. A concrete examples of a H-metric query is given in the following section.

3.4 WSRF Service Deployment Phase

Services can be deployed at any time during the life of a WSRF-p2p container network, however bulk service deployment into a new network is somewhat of a special case as we desire to have a good initial distribution of services to containers to minimise the amount of maintenance and migration required later on. In either case however, we distinguish 2 types of services:

- Constrained services: These services have specific location dependencies. For example, if the service needs to be close to an existing database or other unique resource. These requirements are spelt out in the SLA governing the execution of this service along with other more flexible requirements such as the type of machine, hard disk speed, etc.
- Unconstrained services: These services have no special requirements for the location. They can be deployed anywhere, providing the container's performance is sufficient to meet the other SLA specifications.

When performing the initial bulk deployment of services to a newly created WSRF-p2p container network the order of the service placement is important. In this case, we select the constrained services first, place these such that their SLA requirements are met, and then deploy the unconstrained services slotting in around the deployed constrained services. However, were the same container to be used as the starting point for these deployments, then the H-metric queries would all follow the same search tree and give very poor efficiency. Our solution is to simply pick a random container for each unconstrained deployment and begin the deployment search from that point.

Deploying into an existing network is slightly different. In this case unconstrained services are placed on the first container for which the H-metric query returns a sufficiently low H-metric. For constrained services, there is no choice about the destination container, and as such the service must be placed there providing the requirements of all the *constrained* services on that container do not exceed its capacity (otherwise the deployment must fail and a new SLA or resources must be provided). Although this may result in potential SLA violations for the existing services on that container, normal maintenance (see section 3.5) will result in unconstrained services being migrated to alternative containers.

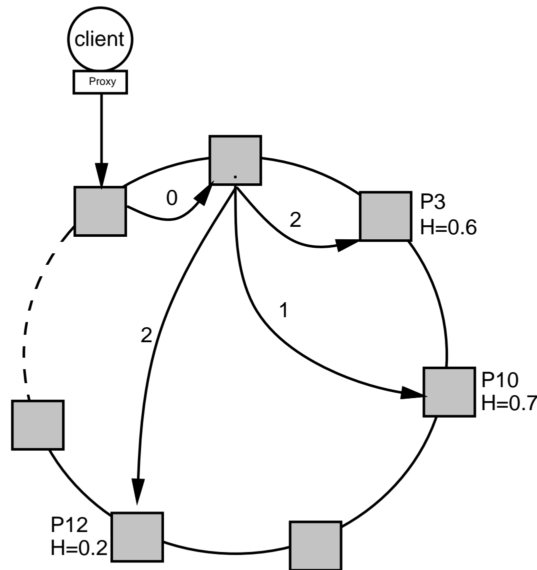

Figure 4: Deployment of a Single Service

Figure 4 shows an example of a service deployment. In this case the service deployer asks its local WSRF container (the one it usually interacts with via its proxy) to deploy a service. The WSRF container picks a random ID, resolves this to the nearest container, and initiates a two level H-metric query from this

root. The H-metric query in this case results in $H = 0.7$ from container P10 and $H = 0.6$ and $H = 0.2$ from its computed children P3 and P12 respectively. Note, the SLA is appended to the H-metric query and if a container cannot satisfy the SLA it will always return $H = 1.0$. If this search fails to return a suitable destination for the deployment, we increase the depth by one and reissue the query. The depth of the search will increase until a destination is found or the number of containers in the search tree exceeds the size of the overlay network. It is worth emphasising that when the depth is increased, there is no need to revisit containers that have already been queried, as the tree at each level is computed at the deployer's local container, and only the new leaf nodes are sent the H-metric requests.

3.5 WSRF Service Maintenance Phase

If the resource usage of a service can be characterised exactly, along with the number and location of its users, the available bandwidth, etc., and we have complete control over the resources on which it is executed - then the initial deployment of services to containers is sufficient. However, as this is not realistically the case, our architecture must be capable of maintaining the SLA compliance of the WSRF-p2p container network. There are four occasions when our system may migrate services: in response to a constrained deployment, in response to a new container joining the network, in response to a container having few services, or in response to a predicted or real SLA violation. The first two have already been discussed in earlier sections. It is also worth pointing out that service migration is expensive and should be only performed when it is really needed, so our approach should be considered degradation avoidance rather than load balancing.

3.5.1 H-metric Observation

To avoid excessive attempts to offload or obtain services, some information about the general load on the system needs to be obtained. Conventionally this is collected at a single point for easy access or by broadcast. Neither of these approaches are suitable for a scalable autonomic network, and in any case, the information need not be completely accurate as placement decisions are made in response to H-metric queries. Rather this information is only needed for determining wether a container ought to be attempting to offload or obtain services. Our novel solution to this problem is to note that in a structured p2p system, individual nodes act as routers, as well as storage devices. Hence H-metric messages that are generated by containers in response to queries from other containers, may be routed though otherwise uninterested containers. We *observe* these H-metric messages that pass through containers when operating as routers, and using a weighted decaying average gather an estimate on the current state of the overall system, *without* needing any additional messages for communicating the system's load.

3.5.2 Over-utilization

When a WSRF container predicts or detects a SLA violation, it will attempt to offload a service that consumes that resource. The first step is to check the observed H-metric, as there is no point trying to offload a service when the entire network is overloaded. Next, the container examines all of its unconstrained service's SLAs to identify a migration candidate with the highest usage of the resource. Once a candidate has been identified, the container issues a help message, appends the candidates SLA and performs a H-metric query of depth 2 as detailed in section 3.3. If this search fails to return a suitable destination for the deployment, we increase the depth by one and reissue the query. The depth of the search will increase until a destination is found or the number of containers in the search tree exceeds the size of the overlay network. This operates in exactly the same way as the deployment of a new service, see section 3.4. If this search fails, then an error is generated and logged, as the system is unable to resolve the problem.

3.5.3 Under-utilized WSRF containers

If a container has few services and determines that it is under-utilised, it will attempt to obtain services to execute. The first step is to examine the observed H-metric. If the H-metric indicates that the average load in the system is high (above a threshold), it will attempt to find a service to host by issuing an H-metric query (as in section 3.3 specifying it's least used resource. The responses to this query will be generated with reference to this resource and the container returning the highest H-metric will be offered the opportunity to offload a service onto this container. It is also worth noting that in a loaded system the query should easily find workload, hence in practice we do not allow the query to exceed a depth of 3.

4 Prototype and Evaluation

We have implemented the autonomic WSRF-p2p container in a Geronimo [2] application server. The p2p Node and the autonomic system manager are implemented as Geronimo Beans (GBeans; [11]). The p2p functionality is provided by an extension of freePastry [5]. Although earlier implementations used Chord, the simulation mode within freePastry resulted in its final selection. However, it is not critical which structured DHT package is used. The prototype is functional, but was run in simulation mode for the experiments to test scalability.

The rest of this section details the four main experiments that we conducted to test various aspects of our architecture. In the first set of experiments we were interested in the impact of query depth on the quality of the initial distribution of services, however this experiment also serves as a proof of concept. In the second set of experiments we investigate the impact of a simple migration policy during the maintenance phase. In the third set of experiments it was our aim to find out how much global information a container could observe passively,

and the forth set of experiments are to determine the cost of duplicates during our computed binary tree H-metric queries. All experiments use the freePastry simulation mode.

4.1 Deployment

The deployment experiments tests the sensitivity of the deployment to the depth of the H-metric search. The basic premise of the experiment is the insertion of 600 services into a clean 100 container network. The setup for the deployment distribution simulation is as follows:

- 100 nodes; ID generated randomly
- The initial H-metric value of all nodes is 0.0
- 600 unconstrained services that contribute load as follows: 200 services with H-metric=0.1; 200 services with H-metric=0.05; 200 services with H-metric=0.01

Figure 5 shows the result for H-metric query searches from depth 1 through 4. The x-axis shows the load on each node, numbered 1 through 100, while the y axis gives the H-metric computed for each node after all 600 services have been placed. The ideal distribution would be that each node gets two each of 0.1, 0.05, 0.01 services, which results in a H-metric of 0.32 for each node. A level 1 search simply selects a random node, and as can be seen from the graph the deviation from the ideal is large (std deviation of 0.22). A level 2 search obtains the H-metric from 3 containers and performs significantly better (std deviation of 0.17). As the depth of the search increases to 3 (std deviation of 0.11) and 4 (std deviation of 0.09) the graphs demonstrate a increasing convergence to the ideal. This data suggests that in practical terms, a search depth of 3 or 4 would be sufficient.

4.2 Maintenance

Once the initial workload has been deployed, the system needs migration to avoid SLA violations. In this set of experiments, the same configuration as above has been used, but after the initial level 1 deployment, the experiment was permitted to enter a maintenance phase with the following simple policy:

> IF H-metric $>$ 0.6 THEN deploy one service to the lowest utilized node.

The depth of the H-metric query in this case was limited to a depth of 2, and the container with the lowest H-metric was selected as the destination. Even with such a simple policy and limited search depth, figure 6 shows a clear improvement to the overall performance of the system with the standard deviation decreasing from 0.22 to 0.16. Clearly there is scope for an improved migration policy, and further improvements could be expected from increasing

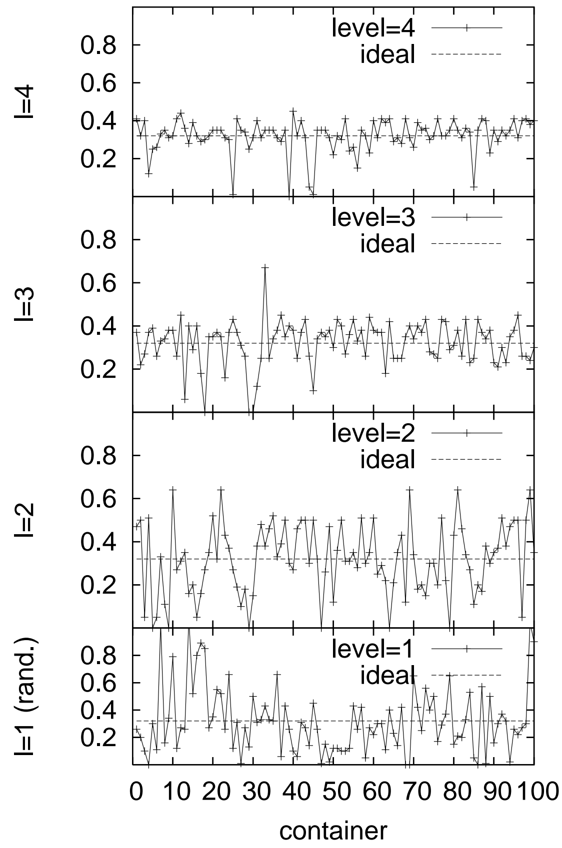

Figure 5: As the depth of the H-metric search increases the quality of the initial deployment also increases.

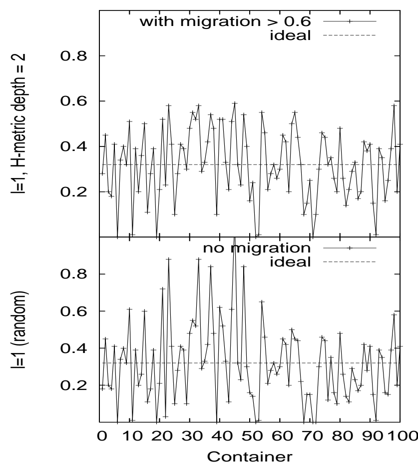

Figure 6: A simple migration policy with a restricted H-metric query depth of 2, still improves overall distribution quality.

the depth of the H-metric query. However, as migration is expensive and time consuming, we feel that the majority of the effort should be performed during service deployment.

4.3 Observation

One very interesting question that we had during the design of the WSRF-p2p architecture was the extent to which the *observation* of H-metric messages being routed through a node could provide a reasonable snapshot of the state whole of the system. It is worth emphasising that nodes that send help messages *do not* include their H-metric, so only lightly loaded containers that respond to help messages will be sampled. Figure 7 shows the H-metrics and the nodes that generated them as observed by an arbitrary container, in this case container number 90. The average of this data is 0.24 whereas the imposed load on the system is the ideal of 0.32. This confirms that only sampling the H-metric responses underestimates the total system load, however this information is obtained at no additional messaging cost. More complex models to estimate the system load more accurately are certainly possible and are worthy of future investigation.

4.4 Duplicates

When the H-metric queries are performed, a binary search tree is computed and overlayed on the DHT ring. This is dependent on the uniformity of the hash and therefore where the nodes happen to fall within the number space of the ring. It is clear that a simple computation to fold a binary search structure

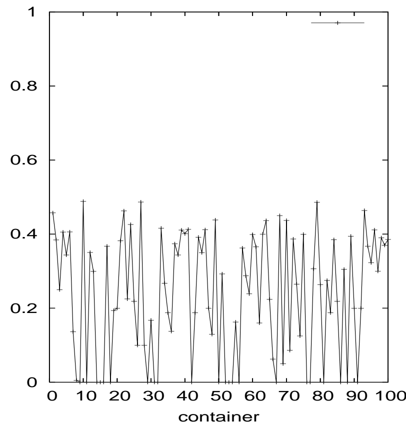

Figure 7: The view of the system that container number 90 has observed while routing H-metric messages.

over this ring will *likely* result in duplicate H-metric requests, and very early in the design version numbers were added to the H-metric responses so that any duplicates could be discarded. However, we had no feel for how often such duplicates would occur and while one can compute an estimate, this requires a number of assumptions about SHA-1 and its distribution. Clearly any duplicates represent pure overhead and this overhead is worthy of quantification. Figure 8 shows the percentage of duplicate messages. At a H-metric depth of 2 there are 7200 messages sent when deploying 600 services and at depth 3 there are 10800 messages.

As can be seen from these graphs, while the number of duplicates does increase with the depth of the H-metric search, the number of duplicates in both cases do not contributed significant overhead to the network.

5 Related Work

There have been a number of projects focusing on autonomic behaviour for managing web services, in particular Ecosystem [16] analyses and reconfigures a service-based system (with MAPE) to satisfy Service Level Agreements with minimal resource consumption. They conclude that migration is a heavy-weight exercise and should be avoided whenever possible and that migrating services to satisfy the minimal resource consumption can lead to unnecessary overhead. Like our approach, the principle is to migrate only when resource bottlenecks occur. Hao [13] carries out migration of weblets, specialized Web services, that can be migrated, according to the round trip time, message size, data location and load of the weblet containers.

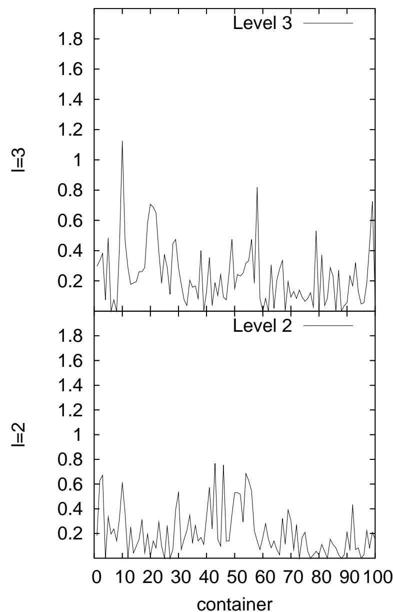

Figure 8: Duplicate Messages in per cent at each container, for H-metric depths of 2 and 3

Other projects have attempted to address scalability issues for, such as [9], which partitions resources into individual, cluster and grid resources. Dowlatshahi et. al [8] have developed an architecture that uses a hierarchical tree structure for participating nodes distant from the Internet backbone, and uses a single peer-to-peer structure for service discovery at the root layer of the underlying tree structures. The key characteristics of their architecture are optimal search for both distant and close services, minimal overhead traffic, scalability, robustness, and easier QoS support. A self-organizing p2p network of resource pools managed by CONDOR [4] has been implemented in [7]. Each resource manager periodically transmits a list of resources that it is willing to share to resource managers that are in close proximity. If a manager has insufficient resources to handle their jobs, they can forward some of their jobs to the advertising resource manager.

The closest work to ours is that of p2pWeb [17], which uses a p2p structured DHT, to deliver a SOA middleware platform. However, although many of the high level goals such as scalability, transparency and fault tolerance are the same, there are many differences in the architecture itself. Load balancing in p2pWeb is an exercise in selecting a replica, that is, p2pWeb does not place or migrate services to satisfy SLA requirements. Essentially p2pWeb and our WSRF-p2p target different domains, while relying on the same technological foundations.

6 Conclusions

In this paper we have presented a novel architecture that combines the principles of autonomic management, service oriented architecture, web services and service level agreements. We use the decentralised, fault tolerant and dynamic properties of a structured p2p DHT to create a scalable decentralised autonomic web service middleware that complies with service level agreements and strives to deliver QoS in response to client specifications. Management of the system is autonomic and therefore reduces and simplifies maintenance. Client workload is evenly distributed throughout the network, by ensuring that the client proxy always selects a random container through which it obtains its services. Otherwise one or two favoured entry points into the WSRF-p2p network could become overloaded. High levels of transparency, the use of current standards and technologies ensure that conventional webservices can be deployed within the system.

SLA aware deployment and migration maintain the performance of the system, and we utilise a novel H-metric query to locate suitable WSRF containers for specific web services. The H-metric query solves the problem of finding a suitable container for hosting a web service, and this search is shown to be effective experimentally. The results also show that the H-metric query does not suffer any significant overhead from duplicate requests. Another advantage of this approach is that WSRF containers can observe H-metric values as they are routed via the container, at no additional messaging cost. This allows the container to observe the state of the network and decide best how it should be operating, that is, looking for other containers on which to offload work or relieving other containers of their excess workload. Migration of services is shown to improve the workload balance within the system, and even a simple policy achieves a large improvement.

References

[1] Apache axis2/java. Home-Page: `http://ws.apache.org/axis2/`.

[2] Apache geronimo. Home-Page: `http://geronimo.apache.org/`.

[3] Apache tomcat. Home-Page: `http://tomcat.apache.org/`.

[4] Condor. Home-Page: `http://www.cs.wisc.edu/condor/`.

[5] freepastry software.

[6] An architectural blueprint for autonomic computing. IBM, 2003. Available at `http://www-3.ibm.com/autonomic/pdfs/ACwpFinal.pdf`.

[7] A.R. Butt, Rongmei Zhang, and Y.C. Hu. A self-organizing flock of condors. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 42–42, Purdue University, West Lafayette, IN, Nov. 2003. ACM Press.

[8] M. Dowlatshahi, G. MacLarty, and M. Fry. A scalable and efficient architecture for service discovery. In *The 11th IEEE International Conference on Networks, 2003. ICON2003.*, pages 51 – 56, September 2003.

[9] M. El-Darieby and D. Krishnamurthy. A scalable wide-area grid resource management framework. In *ICNS '06. International conference on Networking and Services, 2006.*, pages 76 – 86, Silicon Valley, USA, July 2006. IEEE Computer Society Press.

[10] Martin Fowler. Inversion of control containers and the dependency injection pattern. *http://www.martinfowler.com/articles/injection.html*, January 2004.

[11] Geronimo user guide. http://cwiki.apache.org/GMOxDOC11/apache-geronimo-v11-users-guide.html.

[12] J. Jeffrey Hanson. Manage apache geronimo with jmx. August 2006.

[13] Wei Hao, Tong Gao, I-Ling Yen, Yinong Chen, and Raymond Paul. An infrastructure for web services migration for real-time applications. In *SOSE '06: Proceedings of the Second IEEE International Symposium on Service-Oriented System Engineering (SOSE'06)*, pages 41–48, Washington, DC, USA, 2006. IEEE Computer Society.

[14] Sun's java management extensions (jmx) page. Home-Page: `http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/`.

[15] Jsr-77: J2ee management specification. http://jcp.org/en/jsr/detail?id=77.

[16] Ying Li, Kewei Sun, Jie Qiu, and Ying Chen. Self-reconfiguration of service-based systems: A case study for service level agreements and resource optimization. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 266–273, Washington, DC, USA, 2005. IEEE Computer Society.

[17] R. Mondejar, P. Garcia, C. Pairot, and A.F. Gomez Skarmeta. Enabling wide-area service oriented architecture through the p2pweb model. In *15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2006. WETICE '06.*, pages 89 – 94, Manchester, UK, June 2006.

[18] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, Frank Leymann, and Bernd J. Krämer. Service-oriented computing: A research roadmap. In Francisco Cubera, Bernd J. Krämer, and Michael P. Papazoglou, editors, *Service Oriented Computing (SOC)*, number 05462 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[19] M. Parashar and S. Hariri. Autonomic computing: An overview. In J.-P. Bantre et al., editor, *Unconventional Programming Paradigms*, volume 3566, pages 247–259, Mont Saint-Michel, France, 2005. Springer Verlag.

[20] A. Rowstron and P. Druschel. IFIP/ACM international conference on distributed systems platforms (middleware). In *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.*, pages 329–350, Heidelberg, Germany, Nov. 2001.

[21] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[22] Oasis web services resource framework (wsrf) tc. Web Page.