Eighth IEEE International Symposium on Cluster Computing and the Grid

An Autonomic Peer-to-Peer Architecture for Hosting Stateful Web Services

Christoph Reich*, Kris Bubendorfer† and Rajkumar Buyya‡
*Department of Computer Science
Hochschule Furtwangen University, Germany
reich@hs-furtwangen.de
†School of Mathematics, Statistics and Computer Science
Victoria University of Wellington, New Zealand
kris@mcs.vuw.ac.nz
‡Grid Computing and Distributed Systems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
University of Melbourne, Australia
raj@csse.unimelb.edu.au

***Abstract*—In this paper we present an autonomic web services architecture that manages both the performance of service containers and the interconnection of those containers into a service overlay network. The advantages of this approach include the easing of management tasks through the autonomic systems ability to self-configure, self-optimise and self-heal. We also benefit from improved resilience and anticipate an improvement in overall performance. In our architecture we incorporate a structured distributed hash table peer-to-peer overlay network within our autonomic web services container. Our architecture is inherently non-hierarchical, widely distributed and enables SLA compliant deployment of WSRF services. We have simplified the management of such a system by adhering to autonomic principles, and we maintain the performance of the system by tightly integrating SLA compliance and migrating services between containers to preserve QoS. We have developed a workable system for both service deployment and migration without the need for global state.**

I. Introduction

It is not simple to organise a collection of web services for a Service Oriented Architecture (SOA), especially when we desire to meet a set of Quality of Service (QoS) requirements. Various management tasks can be eased by engineering autonomic service containers. Autonomic service containers enable self-configuration, self-optimisation, self-healing and self-adaption [1], [2], [3] to improve resilience and overall system performance. QoS is often seen as the satisfaction of Service Level Agreements (SLAs), and therefore the autonomic service container must monitor for and correct any SLA violations.

Organising the containers in a hierarchy [4], [5] makes it difficult to manage the resulting network as the management role each container plays is dependent on its position within the hierarchy. We believe that each autonomic WSRF container should contribute to the management of the overlay network and avoid any specialised or static hierarchical roles. These requirements fit well with the principles of peer-to-peer networking [6] and such systems typically permit a wide distribution of workload, decentralised management, failure tolerance and replica management. In our architecture we incorporate a modified Pastry node within each of our autonomic WSRF service containers.

Our contributions: (1) We have developed a scalable decentralised solution to the deployment of distributed web services. (2) We have simplified the management of such a system by adhering to autonomic principles, and we maintain the performance of the system by tightly integrating SLA compliance and migrating services between containers to preserve QoS. (3) We have developed a workable system for both service deployment and migration without the need for global state. (4) We collect load information by observation of the peer-to-peer routing packets and find resources using a computed search tree that overlays the network of autonomic WSRF service containers. (5) Finally, we have constructed the architecture using standard modern technologies which allows us to hist legacy webservices with only minor adaption.

The rest of the paper is organised as follows: in section II we describe the architectural design of the autonomic WSRF container, in section III we describe the peer-to-peer network architecture and how it is integrated with and managed autonomically by the WSRF container, in section IV we detail the prototype and our experimental results, in section V we explore related work, and finally in section VI we conclude the paper.

II. WSRF Container Architecture

In this paper we focus on the peer-to-peer architecture for interconnecting our autonomic WSRF containers. A more detailed treatment of the engineering of the WSRF container is available in [7]. For brevity the autonomic WSRF container will be simply referred to as the *container* for the remainder of this paper. Figure 1 outlines the internal architecture of the container. The container is embedded within a Geronimo [8] application server and WSRF services are deployed in Axis2 [9] running in Tomcat [10]. JSR-77 [11] provided via JMX [12] is used to monitor the WSRF services inside the

978-0-7695-3156-4/08 $25.00 © 2008 IEEE
DOI 10.1109/CCGRID.2008.66

service container (e.g. request counter, processing time, etc.). MAPE [13] is implemented using Geronimo's GBeans [14] and provides autonomic management and utilises SLAs and performance metrics to trigger self managing operations such as service migration.

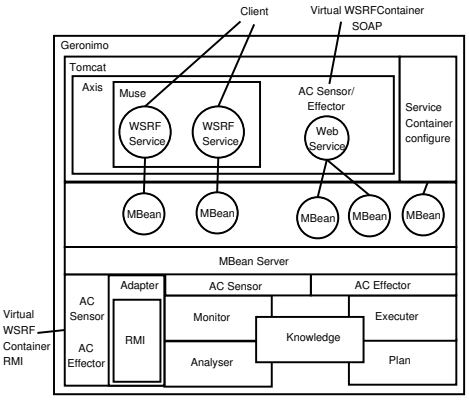

Fig. 1. Internal structure of the WSRF container

An SLA for each service is registered with the host container when the service is deployed. The SLA encodes three differentiated service levels, gold, red and green (priority is in the given order). These service levels define the acceptable performance limits for each service level. In addition to QoS monitoring, we also use these service levels to rank services for deployment or migration. For example, gold services are the most valuable and are therefore prioritised during deployment and are only migrated as a last resort. A more detailed description of the SLA and differentiated service levels is available in an earlier paper [15].

If a container is not able to internally (within the container) resolve an actual or a predicted SLA violation detected via the MAPE-K loop, it will generate a *help* message indicating which resource is oversubscribed. This message is delivered to a subset of containers, which in response generate a specific health value (H-value) computed using a health metric (H-metric) and biased towards the oversubscribed resource. The H-metric is detailed in [15], but in essence, each monitored resource is normalised and then all of the resources are summed and normalised. 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. To emphasise, two simultaneous requests with different violating resources, will result in different H-value responses from the same container.

III. Container Peer-to-Peer Architecture

To build scalable, flexible, self configuring autonomic service overlay network, each container contributes to the management of the overlay network and avoids all specialised and 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. Pastry [6] is a good fit with our design requirements and was used as the basis for our peer to peer network. To store data each data object Pastry maps to the numerically closest real 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 by the SHA-1 hash function. Figure 2 illustrates the extension of the container to integrate a customized peer-to-peer node and routing servlet within the Geronimo application server. The modified peer-to-peer node stores a mapping of Service ID to Container ID. For this paper we have made the simplifying assumption that the containers and services all belong to the same organisation. With our simplifying assumption, peer-to-peer problems such as churn [16] are not a significant. Service discovery takes place outside of our architecture, using any of the many available systems (such as the Globus MDS). The following subsections III-A through III-E outline our architecture based on the operating phases of the system, e.g. service access, creation, deployment, and maintenance.

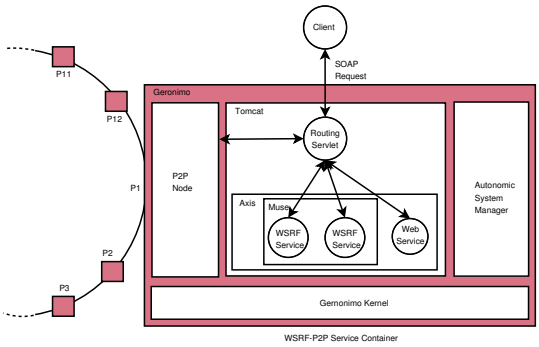

Fig. 2. Peer-to-peer network interconnection of the WSRF service container

A. Creation and Initialisation Phase

New containers join the network via any existing container, through which they bootstrap themselves into the network. The degree to which resources are normalised during the calculation of the H-value needs to be common throughout the network so that containers may be compared. 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 via sequential ring traversal to all containers in the network. The maximums have associated version numbers (derived from the source machine ID) and only normalized values with the same version numbers are comparable. This is to prevent incorrect migration decisions between nodes using different normalization maximums that could occur during the propagation of new values through the ring.

B. Container Discovery

In a ring overlay network there is no hierarchy to determine how a potential host container should be queried for its H-value. As we want a scalable overlay network, solutions such as broadcast and sequential ring traversal are not plausible. Our solution is to treat the ID space of the containers as evenly populated and to compute an overlay binary query tree. Algorithm 1 `findIDs` gives the algorithm used to find the children of a container (id) to depth $level$. In order to balance load and void containers becoming hot-spots, we often pick a random container as root and balance the load by computing different search trees for subsequent queries.

Algorithm 1 Pseudo-code for findIDs

input $level$, id
output C
if $level < maxlvl$ **then**
  $o = \frac{max}{2^{level+1}}$
  $C1 \leftarrow$ findIDs($level + 1$, mod($max + id - o$, max))
  $C2 \leftarrow$ findIDs($level + 1$, mod($max + id + o$, max))
end if
return $C1 \cup C2$

For example, for a maximum container ID of 191 (max), a maximum level of 3 ($maxlvl$) and using container `31` (id) as the root, the resulting set C is {31, 175, 143, 189, 69, 37, 83}. The algorithm is also easily modified to find only the containers within a specific level. This is very useful and efficient; say your query was unsuccessful at a depth of 2, you can simply compute the containers for level 3 and you need not revisit the containers from level 2 during the level 3 query. The example given in Figure 3, in which container `31` has experienced an SLA violation, illustrates this.

C. WSRF Service Deployment Phase

Services can be deployed at any time during the life of a WSRF-peer-to-peer 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 QoS maintenance required later on. This phase of operation is essentially *initial placement* [17]. We distinguish 2 types of services that apply during initial placement:

- 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. Unconstrained SLAs are differentiated into three classes, Gold, Red and Green [15].

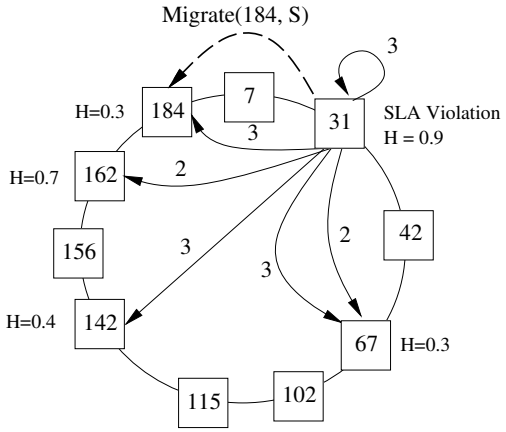

Fig. 3. H-value query and resulting migration as a response to an SLA violation at container `31`. Container `31` starts by computing a level 2 overlay tree and issues a help message to containers {`69` and `175`}. These container IDs are resolved to real containers `67` and `162`, which return their H-values and willingness to receive a service migration. In this case container `67` is refusing incoming migrations and `162` is overloaded. Container `31` then computes the container IDs for level 3, {`37, 83, 143` and `189`} and sends the help message to these containers. When the container IDs are resolved to real containers the duplicates (`37` $\rightarrow$ `31`) and (`83` $\rightarrow$ `67`) are detected as having duplicate message version numbers and are discarded by the containers themselves. Container `184` is selected as the migration destination and service S is migrated.

When performing the initial bulk deployment of services to a new container network, the order of the service placement is important. In this case, we select the constrained services first and place these such that their SLA requirements are met. We next deploy the unconstrained services slotting in around the deployed constrained services. However, were the same container to be used as the root for these deployments, then the H-value queries would all query the same nodes and give very poor efficiency and resulting distribution. So, during deployment we pick a random container as the root for each unconstrained deployment and begin the search from that point.

Deploying into an existing network is slightly different. Unconstrained services are placed on the first container for which the H-value query returns a sufficiently low H-value and will therefore not cause an SLA violation. For constrained services, there is no choice about the destination container, and as such the service must be placed there. This can have the effect of introducing SLA violations for the chosen container, and we rely on normal maintenance (see section III-D) to remove enough unconstrained services to return the container to SLA compliance.

D. Service Migration

After the services have been initially deployed, we use service migration to ensure SLA compliant performance. At this point it is worth pointing out that not all services are candidates for migration. The constrained services mentioned in section III-C are not. Also, some of the initially unconstrained services can also not be moved at certain times due to run-time

constraints such as open files, locally generated data and other resources that they are consuming. We note that these services are *temporarily constrained*, and rely on the service to report to the container when it changes state between unconstrained and temporarily constrained.

There are four occasions when our system may attempt to migrate services: (1) in response to a constrained deployment (migrate unconstrained services away), (2) in response to a new container joining the network (opportunity to offload services), (3) in response to a container having few services (again, opportunity to offload services), or (4) in response to a predicted or real SLA violation. In essence our migration system is symmetrically initiated [17] where; during light loading of the system heavily loaded containers operate to offload excess workload and during heavy loading when under-loaded containers operate to import workload. The following sections III-D1 through III-D3 provide algorithmic equivalents to the maintenance actions taken in the MAPE-K loop – including violations and load initiated actions.

1) H-value Observation: To avoid non-productive 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 place for easy access, however solutions are not usually scalable or robust. As H-value queries are in essence directed polls and are used to make the final placement decisions, any global information need not be completely accurate. We in fact only use this information to determine if a service migration is worth pursuing.

Our novel solution to this problem is to note that in Pastry and many other peer-to-peer systems, individual containers act as routers, as well as storage devices. When acting as a router, messages for other containers will pass through this container and we can therefore record any H-values contained in those messages. These H-values are recorded for each container (the source of the message) using a weighted decaying average ($L_{msg.src} \leftarrow (1-\alpha)L_{msg.src} + \alpha \times msg.Hvalue$). The value α is a tuneable parameter, and for our experiments was set to 0.5. The resulting load vector L can be used to provide an estimate on the current state of each container and the overall loading in the system. To emphasise the point, collection of this data requires *no* additional messages for communicating the system's load. The accuracy of the observation based estimates is presented in section IV-C. When a new node joins the network it has not yet been able to observe any load values in the system and therefore will have difficulty making migration decisions. This is resolved during bootstrapping, when the new node obtains the observed load values from its immediate neighbors and averages them.

2) Under-utilized Containers: If a container has few services or has just joined the network it may determine that it is under-utilised and will attempt to obtain services to execute. This is first step in the receiver initiated [18] phase of the `receiverInitiated` Algorithm 2. The next step is to examine the observed H-value. If the H-value of the current container is less than the average load in the system less the threshold T_{pull} (The threshold T_{pull} defines the amount a container must be below the average load before it will attempt to import workload), it will attempt to find a service to host by issuing a query (see section III-B) specifying it's least used resource r and query depth n. The responses to this query will be generated with reference to the specified resource and the responses received are placed in vector H. Vector H is then filtered based on whether the container is willing to offload a service to produce the vector A. The query will continue to loop increasing the depth until a source candidate has been located or until the depth of the query exceeds the maximum. The container in A returning the highest resource targeted H-value will be offered the opportunity to offload a service to the requesting 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 $max = 3$.

Algorithm 2 Pseudo-code for receiverInitiated

```
input L
output success, failure
if this.Hvalue < T_noload then
  if this.Hvalue < average(L) − T_pull then
    r ← min(∀R)
    n ← 2
    while A is empty & n ≤ max do
      H ← HvalueQuery(msg.request(r, n))
      A ← ∀h ∈ H | isAccepting(h)
      inc(n)
    end while
    source ← max(A)
    if source is not NULL then
      return source.migrationRequest(this, r)
    end if
  end if
end if
return failure
```

The `migrationRequest` Algorithm 3 is then instantiated on the selected source container. The source container verifies that it is still overloaded using threshold T_{loaded}, and then chooses and migrates an unconstrained service ranked preferentially by the target resource r.

Algorithm 3 Pseudo-code for migrationRequest

```
input destination, r
output success, failure
if this.Hvalue > T_loaded then
  S ← choose(r)
  if S is NULL then
    return failure
  end if
end if
return migrate(S, destination)
```

The `choose` Algorithm 4 selects an unconstrained service in decreasing preference from green, red and gold services.

Algorithm 4 Pseudo-code for choose

```
input r
output S
S ← NULL
S ← selectUnconstrained(green, ranked by r)
if S is NULL then
  S ← selectUnconstrained(red, ranked by r)
  if S is NULL then
    S ← selectUnconstrained(gold, ranked by r)
  end if
end if
return S
```

3) Over-utilization: A container may also attempt to offload services either in response to a constrained service deployment, or in response to a predicted or real SLA violation. In the case of a SLA violation, the `violation` Algorithm 5 is called from the MAPE-K loop in response to a monitored service violation. The algorithm stops registering violation events and prevents any incoming service migrations. We then obtain the violating resource r from the SLA violation event and use this resource when issuing the H-value query to locate a migration destination. The remainder of the algorithm is similar to the `ReceiverInitiated` Algorithm 2, except we select the destination with the lowest H-value with reference to the violating resource r, and the value of max is higher. If this search fails, then an error is generated and logged, as the system is unable to resolve the problem.

Algorithm 5 Pseudo-code for violation

```
Input: violation
Output: success, failure
STOP registering violations
START refusing inbound service migrations
r ← violation.getResource()
n ← 2
while A is empty & n ≤ max do
  H ← HvalueQuery(msg.request(r, n))
  A ← ∀h ∈ H | isAccepting(h)
  inc(n)
end while
destination ← min(A)
if destination is not NULL then
  if migrationRequest(destination, r) then
    START registering violations
    STOP refusing inbound service migrations
    return success
  end if
end if
return failure
```

When we are responding to a constrained deployment, we call the `SenderInitiated` [19] Algorithm with the observed global load vector L rather than the SLA violation. Otherwise the only differences between this algorithm and the `violation` Algorithm 5 are that we test that the local H-value is in excess of the threshold T_{loaded} and that it is also in excess of the average observed global load plus the threshold T_{send}. If a service migration is worth attempting, the container will issue identify its most oversubscribed resource r and issue an H-value query to locate a migration destination.

E. WSRF Service Access

To obtain a service, the client only needs to know a single container in the container overlay network and the service's name. Binding to the service can take place either via any known container or via a proxy. the advantage of the proxy is that we can hide rebinding when services migrate or fail or when containers fail. Another advantage of the proxy is that by picking random container IDs we can spread the load on well known containers which may become hot-spots for client requests.

IV. Prototype and Evaluation

The architecture has been implemented and installed on five machines with a small test set of webservices. However to properly exercise the prototype, we elected to use the freePastry **simulation** mode with **100 containers** and deploy a synthetic webservice workload. The code executed in the simulation was the same container code that was deployed and tested on the real machines.

The focus of this paper is on the peer-to-peer architecture for interconnecting our containers, and therefore the experiments herein are designed to support this focus. We have also investigated the QoS in terms of SLA violations in our system, and these results are available in the paper [15]. The In the first set of experiments we examined the impact of query depth on the quality of the initial distribution of services. In the second set of experiments we investigated the impact of query depth on 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 with no additional effort, and in the forth set of experiments we looked at the cost of duplicates during our computed binary tree H-value queries.

A. Deployment

The deployment experiments test the sensitivity of the service distribution to the depth of the H-value search. The basic premise of the experiment is the insertion of 600 services into a clean 100 container network. The 600 services were all unconstrained services and contributed load as follows: 200 services each with a H-value of 0.1; 200 services each with a H-value of 0.05; 200 services each with a H-value of 0.01. The ideal distribution is two services of H-value 0.1, 0.05 and 0.01 deployed to each container.

Figure 4 shows the result for H-value query searches from depth 1 through 4. A level 1 search simply selects a random container and the results exhibit a large std deviation of 0.22. A level 2 search obtains the H-value from 3 containers and performs significantly better (std deviation of 0.17). As the

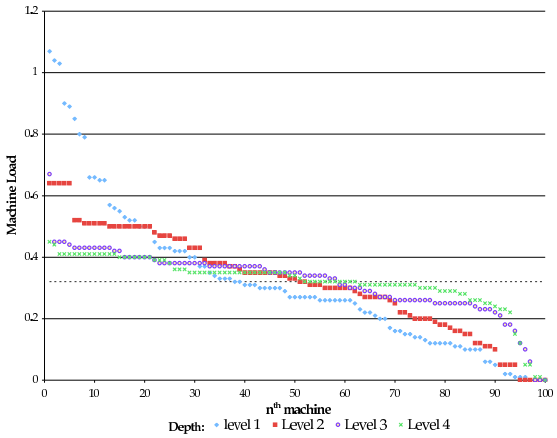

Fig. 4. This graph shows the load on each of the 100 machines as the H-value query depth is increased from 1 (random) to a tree of depth 4. The load at each machine is measured in terms of H-value, and the machines are sorted by load to enable a direct comparison of the results for each depth. The dashed line shows the ideal load, that would be seen by each machine if the distribution was perfect. As can be seen from the graph, the more levels in the H-value search, the closer the load presented to each machine approaches the ideal. For example, looking at search depth 1 and search depth 4, search depth 1 has many more overloaded and underloaded than search depth 4, this is especially evident in the first and last 20% of the machines.

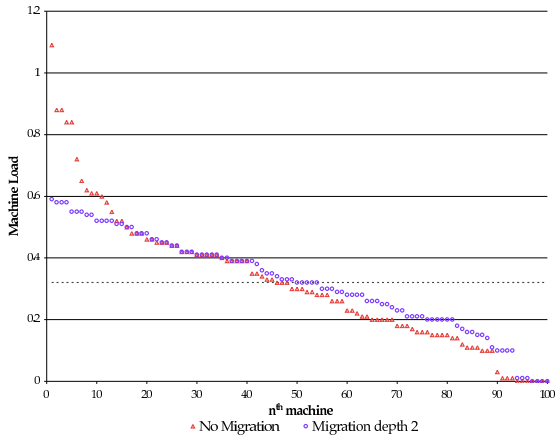

Fig. 5. This graph shows the load on each machine as the migration policy becomes more aggressive in finding a better host. The policy in this experiment used a load threshold of 0.8, so that only machines with a load worse than this would initiate a service migration. The load at each machine is measured in terms of H-value, and the machines are sorted by load to enable a direct comparison of the results for each depth. The dashed line shows the idealised load, that would be seen by each machine if the distribution was perfectly even. As can be seen from the graph, even with the same threshold, increasing the query depth from 1 to 2 significantly improves the distribution of services in the system.

depth of the search increases to 3 (std deviation of 0.11) and 4 (std deviation of 0.09) the graph demonstrates an increasing convergence to the ideal. This data suggests that in practice, a search depth of 3 or 4 would usually be sufficient.

B. Maintenance

Once the initial workload has been deployed, the system needs migration to avoid SLA violations. In this experiment, the same configuration as above has been used, but after the initial deployment to a depth of 1, the experiment was permitted to enter a maintenance phase.

The depth of the H-value query in this case was limited to a depth of 2 as the results given in figure 4 show only a minor improvement in our experiments at greater depths than this, however we would expect larger networks to benefit from greater search depths. The container with the lowest H-value was selected as the destination. Even with such a simple policy and limited search depth, figure 5 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 the depth of the H-value query.

C. Observation and Determining Global Load

Before we can enter either sender-initiated or receiver initiated migration mode, we need to have an estimate of the overall load in the system. This information could be obtained via a central load information repository – or via explicit polling. However, a very interesting question that arose during the design of the architecture was the extent to which the *observation* of H-value messages being routed through a container could provide a reasonable snapshot of the state whole of the system. Figure 6 shows the H-values and the containers 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 average load in the system is 0.32. We believe that this is due to the fact that one help message will generate many more responses, giving a greater weighting to underloaded hosts in the system. Overloaded hosts need not respond to a help message. However this information is obtained at no additional messaging cost. More complex models with more communication between groups of observing peers that could estimate the system load more accurately are certainly possible and are worthy of future investigation. In addition, 100 nodes is a rather small network. It is not clear if there would be sufficient observed load samples in a very large network, and this is an interesting problem for future work.

D. Duplicate Messages

When the H-value queries are performed, a binary search tree is computed and overlaid on the ring. This is dependent on the uniformity of the hash and therefore where the containers happen to fall within the number space of the ring. It is clear that a simple computation to fold a binary search structure over this ring will *likely* result in duplicate H-value requests. This was emphasised in the example given in Figure 3. To resolve this problem all H-value queries have version numbers so that any duplicates can be discarded. However, we had no practical

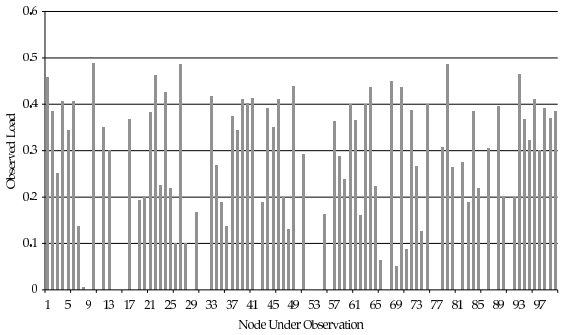

Fig. 6. This graphs shows view of the global load in the system as observed by container 90 (its load is not included in the observation data) at the end of the simulation run. This information is collected by observing H-value messages routed through container 90 on their way to other containers.

feel for the number of duplicates and how much overhead would result. Figure 7 shows the percentage of duplicate messages. With 100 containers and a query depth of 2 there are 7200 messages sent when deploying 600 services and at depth 3 there are 10800 messages.

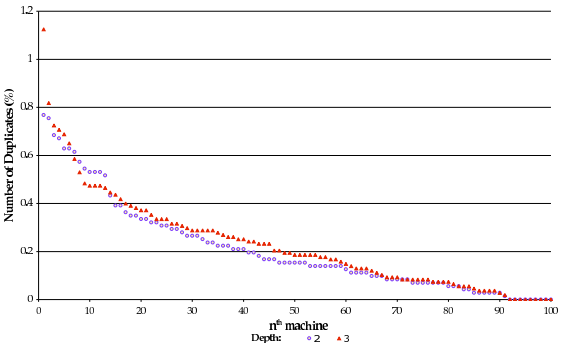

Fig. 7. When the H-value query tree is computed, there is the possibility for some containers to occur in multiple positions within the resulting binary tree due to hash ranges resolving a single container. This graph quantifies the percentage of duplicate packets received at each container (sorted by frequency to aid comparison) for for H-value query depths of 2 and 3.

As can be seen from these graphs there is very little percentage increase in the number of duplicates as the depth of the H-value query increases. In both cases the number of duplicate packets seen by the containers is very small, and will not contribute significant overhead to the network.

V. Related Work

There have been a number of projects focusing on autonomic behaviour for managing web services, in particular Ecosystem [3] analyses and reconfigures a service-based system (with MAPE) to satisfy SLAs 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 [20] 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.

Other projects have attempted to address scalability issues including El-Darieby and Krishnamurthy [5], who partition resources into individual, cluster and grid resources. Haas et. al [4] examine the use of hierachical structures for the automated deployment of services over heterogeneous networks. They utilise QoS aware routing in the network, but do not provide any mechanisms for supporting the choice of host machines or for maintaing a larger QoS picture. Dowlatshahi et. al [21] have developed an architecture that uses a hierarchical tree structure for participating containers 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 peer-to-peer network of resource pools managed by CONDOR has been implemented by Butt et. al [22]. 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.

Kang et. al [23] divide SLAs into function domains (low, medium and high function domains). The 95-percentile response time of the real server is used as base for determining whether to allocate more computing resources to clients demanding a high level of service. They do not consider service migration to meet the QoS targets. Lee and Lee[24] discuss how to integrate a service provider in a negotiation framework. An important aspect is the need for a quality measurement like the H-value developed in this paper. Mikic-Rakic et. al. [25] present an applied self-reconfiguration approach to support disconnected operations by allowing the system to monitor and automatically redeploy itself.

Gokhale and Natarajan [26] have developed a model driven architecture for Grid Webservices, that ensures end-to-end QoS and tie this in to a Meta Resource broker to meet more general QoS requirements. They do not look at SLA maintenance and do not migrate services to preserve QoS. They also do not use any form of overlay network, and have not addressed the issue of scalability.

Berenbrink et. al [27] introduce a game-theoretic mechanism which they use to find suitable allocations. Each task is associated with a “selfish agent”, and requires each agent to select a resource, with the cost of a resource being the number of agents to select it. Agents would then be expected to migrate from overloaded to under loaded resources until the allocation becomes balanced. This system is unlikely to scale well, as the resource discovery is centralised. The research of Zeid and Gurguis [28] aims to prove that with autonomic Web services, computing systems will be able to manage themselves as well as their relationships with each other. To achieve this objective,

the research proposes a system that implements the concept of autonomic Web services but without service migration.

The closest work to ours is that of P2PWeb [29], which uses a peer-to-peer structure to deliver a SOA middleware platform. However, although we share many of the high level goals such as scalability, transparency and fault tolerance, there are many significant differences in the architecture itself. Load balancing in P2PWeb is an exercise in selecting a replica, that is, P2PWeb does not deploy or migrate services to satisfy QoS requirements.

VI. Conclusions

In this paper we have presented a novel architecture that combines the decentralised, fault tolerant and dynamic properties of a structured peer-to-peer overlay to create a scalable decentralised autonomic web service middleware that monitors SLA compliance to achieve QoS goals. Management of the system is autonomic and therefore reduces and simplifies maintenance. SLA aware deployment and migration maintain the QoS of the system, and we utilise a novel H-value query to locate suitable WSRF containers for deploying specific web services. The H-value query solves the problem of finding a suitable container for hosting a web service, and this search is shown to be efficient experimentally. The results also show that the H-value query does not suffer any significant overhead from duplicate requests. Another advantage of this approach is that WSRF containers can observe H-values as they are routed via the container, at no additional messaging cost in small to medium networks. 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. 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. This avoids one or two favoured entry points into the container overlay network could become overloaded. Our use of high levels of transparency, current standards and technologies ensure that legacy webservices can be deployed within the system with minimum alteration.

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer, "Service-oriented computing: A research roadmap." Dagstuhl, Germany: Internationales Begegnungs und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[2] M. Parashar and S. Hariri, "Autonomic computing: An overview," in *Unconventional Programming Paradigms*, vol. 3566. Mont Saint-Michel, France: Springer Verlag, 2005, pp. 247–259.

[3] Y. Li, K. Sun, J. Qiu, and Y. Chen, "Self-Reconfiguration of Service-Based Systems: A Case Study for Service Level Agreements and Resource Optimization," in *Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, Washington, DC, USA, 2005, pp. 266–273.

[4] R. Haas, P. Droz, and B. Stiller, "Cost and Quality-of-Service Aware Network-service Deployment," in *Proceedings of Internet Charging and QoS Technology*, September 2001, pp. 166–171.

[5] M. El-Darieby and D. Krishnamurthy, "A Scalable Wide-Area Grid Resource Management Framework," in *International conference on Networking and Services*, Silicon Valley, USA, July 2006, pp. 76 – 86.

[6] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November 2001, pp. 329–350.

[7] C. Reich, M. Banholzer, R. Buyya, and K. Bubendorfer, "Engineering an Autonomic Container for WSRF-based Web Services," in *proceedings of the 15th International Conference on Advanced Computing and Communication (ADCOM)*, Bangalore, India, December 2007.

[8] "Geronimo." [Online]. Available: http://geronimo.apache.org/

[9] "Axis2/Java." [Online]. Available: http://ws.apache.org/axis2/

[10] "Apache tomcat." [Online]. Available: http://tomcat.apache.org/

[11] "JSR-77: J2EE Management Specification." [Online]. Available: http://jcp.org/en/jsr/detail?id=77

[12] "Java Management Extensions (JMX)." [Online]. Available: http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/

[13] "An Architectural Blueprint for Autonomic Computing," IBM and autonomic computing, 2003.

[14] J. J. Hanson, "Manage apache geronimo with jmx," August 2006. [Online]. Available: http://www.ibm.com/developerworks/library/os-ag-jmx/index.html

[15] C. Reich, K. Bubendorfer, and R. Buyya, "SLA-Oriented Management of Containers for Hosting Stateful Web Services," in *proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing*, Bangalore, India, December 2007.

[16] D. Stutzbach and R. Rejaie, "Understanding Churn in Peer-to-Peer Networks," in *proceedings of the Internet Measurement Conference*, Rio de Janeiro, Brazil, October 2006.

[17] D. Eager, E. D. Lazowska, and J. Zahorjan, "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing," *Performance Evaluation*, vol. 6, pp. 53–68, March 1986.

[18] M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Proceedings of the ACM Computer Network Performance Symposium*, pp. 47–55, April 1982.

[19] B. Bershad, "Load Balancing with Maitre d'," University of California at Berkeley, Tech. Rep. CSD-85-276, Dec. 1985.

[20] W. Hao, T. Gao, I.-L. Yen, Y. Chen, and R. Paul, "An Infrastructure for Web Services Migration for Real-Time Applications," in *the Second IEEE International Symposium on Service-Oriented System Engineering (SOSE)*, Washington, DC, USA, 2006, pp. 41–48.

[21] M. Dowlatshahi, G. MacLarty, and M. Fry, "A Scalable and Efficient architecture for Service Discovery," in *The 11th IEEE International Conference on Networks (ICON)*, September 2003, pp. 51 – 56.

[22] A. R. Butt, R. Zhang, and Y. C. Hu, "A self-organizing flock of condors," *J. Parallel Distrib. Comput.*, vol. 66, no. 1, pp. 145–161, 2006.

[23] C. Kang, K. Park, and S. Kim, "A Differentiated Service Mechanism Considering SLA for Heterogeneous Cluster Web Systems," in *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems.*, April 2006.

[24] B.-Y. Lee and G.-H. Lee, "Service Oriented Architecture for SLA Management System," in *Advanced Communication Technology*, Febuary 2007, pp. 1415–1418.

[25] M. Mikic-Rakic and N. Medvidovic, "Support for Disconnected Operation via Architectural Self-Reconfiguration," in *the First International Conference on Autonomic Computing (ICAC)*, Washington, DC, USA, 2004, pp. 114–121.

[26] A. S. Gokhale and B. Natarajan, "Composing and deploying grid middleware web services using model driven architecture," in *On the Move to Meaningful Internet Systems, CoopIS*. London, UK: Springer-Verlag, 2002, pp. 633–649.

[27] P. Berenbrink, T. Friedetzky, L. A. Goldberg, P. Goldberg, Z. Hu, and R. Martin, "Distributed Selfish Load Balancing," in *the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, New York, NY, USA, 2006, pp. 354–363.

[28] A. Zeid and S. Gurguis, "Towards Autonomic Web Services," in *The 3rd ACS/IEEE International Conference on Computer Systems and Applications*, 2005.

[29] R. Mondejar, P. Garcia, C. Pairot, and A. F. G. Skarmeta, "Enabling Wide-Area Service Oriented Architecture through the p2pWeb Model," in *15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Manchester, UK, June 2006, pp. 89 – 94.