A SLA-Oriented WSRF Container Architecture

Christoph Reich[1], Kris Bubendorfer[2], Matthias Banholzer[1], Rajkumar Buyya[3]

[1] Department of Computer Science
Hochschule Furtwangen University, Germany
reich@hs-furtwangen.de
[2] School of Mathematics, Statistics and Computer Science
Victoria University of Wellington, New Zealand
kris@mcs.vuw.ac.nz
[3] Grid Computing and Distributed Systems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
raj@csse.unimelb.edu.au

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 monitor and rectify its QoS to satisfy its SLAs. The SLA plays an important role during two distinct phases in the lifecycle of a WSRF container. Firstly during service deployment when services are assigned to containers in such a way as to minimise the threat of SLA violations, and secondly during maintenance when violations are detected and services are migrated to other containers to preserve QoS. In addition, as the architecture has been designed and built using standardised modern technologies and with high levels of transparency, conventional webservices can be deployed with the addition of a SLA specification.

1 Introduction

Webservices and the associated Web Services Resource Framework (WSRF) [1] 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 [2][3][4] 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) [5] to manage its internal functionality, detect SLA violations and trigger corrective actions. The containers themselves are connected in via a P2P (peer to peer) overlay to

achieve a wide distribution of workload, decentralised management, and failure tolerance.

The contributions that we make in this paper are: we have (1) developed an overall *health* metric for the WSRF container that is a single easily comparable value and is normalised to provide for heterogeneous resources, (2) provided differentiated service level domains (green, red and gold) in our SLAs, and (3) developed a decentralised migration algorithm that redistributes services between containers to meet agreed QoS using the health metric and service level domains. In addition the P2P overlay architecture is decentralised, highly distributed and scalable. We have implemented the architecture using standard modern technologies and with high levels of transparency, indeed, conventional webservices can be deployed with the addition of a SLA specification.

The rest of the paper is organised as follows. In section 2 we outline the basics of the autonomic WSRF container and we describe the general architecture of the system. Section 3 lays the foundation for the determination of the health of a WSRF container, section 4 presents the migration algorithms, section 5 details our experimental results that validate our approach, section 6 explores related work, and finally section 7 concludes this paper.

2 WSRF Container Architecture

In this paper we focus on the role of SLAs in the assignment and migration of services between 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 [6] application server, the WSRF services are deployed in Axis2 [7] running in Tomcat [8]. JSR-77 [9]provided by JMX [10] is used to monitor the WSRF services inside the service container (e.g. request counter, processing time, etc.). MAPE [5] is implemented using Geronimo's GBeans [11], provides autonomic management and utilises SLAs and performance metrics to trigger self managing operations such as service migration. Using GBeans provides access to Geronimo's advanced features, such as, Inversion-of-Control [12].

Individual WSRF containers are interconnected via a modified version of the Pastry [13] structured P2P overlay network. Each WSRF container contributes to the autonomic management of the overlay network and there are no specialised or static management roles. This architecture provides robustness and permits a wide distribution of workload. A container's pastry node stores an index for the service, and resolves this to the actual container on which the service is hosted.

2.1 WSRF Container Operation

The SLA plays an important role during two distinct phases in the lifecycle of a WSRF container. Firstly during service deployment when services are assigned to containers in such a way as to minimise the threat of SLA violations, and secondly during maintenance when violations are detected and services are migrated to other containers to preserve QoS. Each container monitors its performance requirements, and if it is unable to resolve the SLA violation internally, it will generate a *help* message indicating which resource is causing the problem.

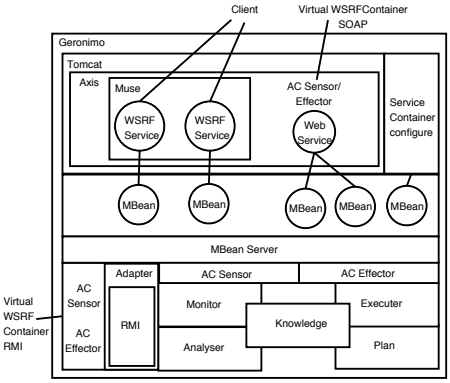

Fig. 1. Internal structure of the WSRF container

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 *health* 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 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 ideally result in two different H-metrics from the same container. Section 3 presents the H-metric in detail.

2.2 Example: Service Deployment

We distinguish two types of services: *constrained* services have specific location dependencies, while *unconstrained* services have no special location requirements. During the deployment of services, constrained services are prioritised over unconstrained services. For the initial deployment of services during container initialisation, this means placing all of the constrained services before the unconstrained services. For deployment of a new service into an existing container network, if the new service is unconstrained then it is simply deployed to the best container using a bounded depth H-metric query in the container overlay. If the new service is constrained, then we have no choice but to attempt to deploy it to the desired container. If that deployment results in an SLA violation, we then attempt to migrate unconstrained services from that container. Figure 2 shows an example of a service deployment.

In this example the service deployer asks its local WSRF container 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. If this search fails to return a suitable destination for the deployment, we increase the depth by

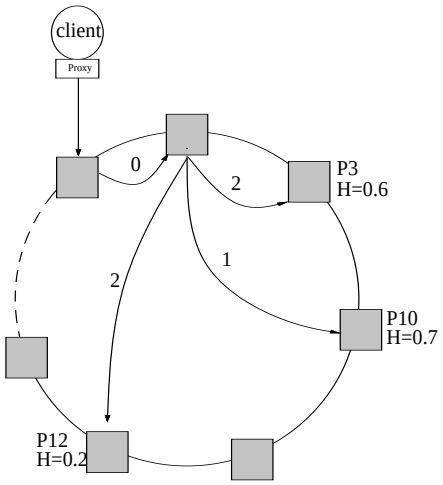

Fig. 2. Service deployment in a WSRF container

one and reissue the query. It is worth pointing out that if the container being queried has an unresolved SLA violation of its own, it will return an H-metric of $H = 100$ signalling that it is unavailable for inward service migrations.

3 Health Status Metric (H-metric) of a WSRF Container

As stated earlier, the H-metric gives an overview of a container's health. The SLA specifies a set of resource conditions that must be satisfied. All containers within the overlay network need to have their resources scaled to deal with heterogeneous hosts, otherwise the H-metrics of the various containers cannot be compared. This scaling information is exchanged when each container joins the container overlay network. If a 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. Each SLA resource condition is therefore scaled proportionally to the minimum and maximum values for the container overlay network and will not need to be changed if a service is subsequently migrated to another container. After scaling, the parameter is normalized (value $\in [0.0, 1.0]$) using a piecewise linear function (see Equations 1,2,3,4,5). The primary reasons for using this normalistion function is it allows us to adjust individual resources which behave in a non-linear way, e.g. memory usage, and because, pragmatically speaking, a server is fully loaded at about 80% to 90%.

$$H_{metric}() = f_{metric}(x, x_{10}, h_{10}, x_{90}, h_{90}) \tag{1}$$

$$H_{metric}() = \begin{cases} 0.0, & \text{if } x <= SLA_{min} \\ f_{10}(), & \text{if } SLA_{min} < x <= \\ & \quad SLA_{min} + x_{10} * SLA_{max} \\ f_{10-90}(), & \text{if } SLA_{min} + x_{10} * SLA_{max} < x <= \\ & \quad SLA_{min} + x_{90} * SLA_{max} \\ f_{90}(), & x < SLA_{min} + x_{90} * SLA_{max} \\ 1.0, & \text{otherwise} \end{cases} \quad (2)$$

$x :=$ measured metric value; $x =\in [SLA_{min}, SLA_{max}]$; $x_{10} \in [0.1, x_{90}]$; $x_{10} :=$ 10% metric default value; $h_{10} :=$ 10% H default value; $x_{90} \in [x_{10}, 0.9]$ $x_{90} :=$ 90% metric default value; $h_{90} :=$ 90% H default value;

$$f_{10}() = \frac{h_{10}x}{x_{10}SLA_{max}} - \frac{h_{10}SLA_{min}}{x_{10}SLA_{max}} \quad (3)$$

$$f_{10-90}() = \frac{(h_{90} - h_{10})x}{SLA_{max}(x_{90} - x_{10})} + h_{10} - \frac{(h_{90} - h_{10})(SLA_{min} + x_{10}SLA_{max})}{SLA_{max}(x_{90} - x_{10})} \quad (4)$$

$$f_{90}() = -\frac{(1.0 - h_{90})x}{SLA_{max} - SLA_{min} - x_{90}SLA_{max}} + 1.0 + \frac{(1.0 - h_{90})SLA_{max}}{SLA_{max} - SLA_{min} - x_{90}SLA_{max}} \quad (5)$$

Figure 3 illustrates the normalisation function for a memory metric: $SLA_{min} = 0MB$; $SLA_{max} = 300MB$; $h_{10} = 0.1$; $h_{90} = 0.9$ and by sweeping $x = [0, 300]MB$ and $x_{10} = [0.1, 0.5]$ with the constraint: $x_{90} = 1.0 - x_{10}$.

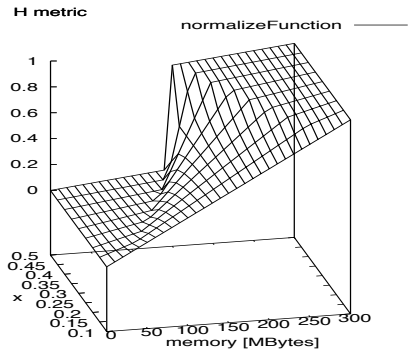

Fig. 3. Normalisation function for memory.

To calculate the overall health status of a container the normalised parameter values are combined and normalized again (see Equation 6).

$$H = \frac{1}{2} * \frac{\sum_{b=1}^{b=n} w_b * H_{metric}()_b}{\sum_{b=1}^{b=n} w_b} + \frac{1}{2} * h_{machine} \tag{6}$$

With $b :=$ machine hosting the WSRF Container and $w :=$ weights to emphasise particular differentiated SLAs. The $w :=$ weights represent SLA service level domains: gold $w = 4$, red $w = 2$, green $w = 1$. *Gold services* are the most important and are least likely to be migrated, *green services* can be thought of as *best-effort* services, while *red services* fall between gold and green services in priority and importance.

The machine specific health status ($h_{machine}$) is set relative to other containers. This permits the H-metric of two different machines (e.g. different MIPS) to be compared by considering the max values of all machines (see Equation 7).

$$h_{machine} = \frac{w_{mem} * H_{mem}() * c_{mem} + w_{cpu} * H_{cpu}() * c_{cpu}}{w_{mem} * c_{mem} + w_{cpu} * c_{cpu}} \tag{7}$$

Weights w_{mem} and w_{cpu} are the same differentiated service level (gold, red and green) weights. The correction factor for different machine resources R (e.g. cpu, mem etc.) is:

$$c_R = \frac{R_{overall-max}}{R_{container-max}} \tag{8}$$

4 Migration Algorithm

There are five occasions when our system may *migrate* services in response to a: (1) constrained deployment, (2) new container joining the network, (3) container having few services, (4) container leaves the network for maintenance, or (5) predicted or real SLA violation. Migration is similar to deployment, although in this case the query starts with a violating container detecting it has a problem, and then issuing a bounded H-metric query (see Figure 2) to locate a new container to host the service that caused the (or is expected to cause a) violation.

Algorithm 1 gives the pseudo-code for a responding to a service violation event. Firstly we stop registering SLA violations for this container, as subsequent violations will have the same outcome - migrate a service off this container. Secondly we stop accepting incoming service migrations. Next we pick a random unconstrained service with a green SLA. If none is available we then try selecting a service with a red SLA and then gold SLA.

Algorithm 2 gives the pseudo-code for the actual migration of the service S identified in algorithm1. Here we construct a 'help' message and append the H-metric and SLA for S. Next we do a bounded depth H-metric query and find the set of accepting containers A, from which we choose the minimum *dest*. The service S is then migrated to container *dest* and the algorithm terminates.

Algorithm 1 Pseudo-code for service violation events

Input: *violation*
Output: *success*, *failure*
STOP registering violations
START refusing inbound service migrations
$S \leftarrow pick_{unconstrained}(green)$
if no value for S **then**
  $S \leftarrow pick_{unconstrained}(red)$
  if no value for S **then**
    $S \leftarrow pick_{unconstrained}(gold)$
  else
    failure
  end if
end if
if migrate(S) **then**
  START registering violations
  STOP refusing inbound service migrations
else
  failure
end if

Algorithm 2 Pseudo-code for migrate

Input: S
Output: *success*, *failure*
$msg_{help} \leftarrow \{Hmetric_{container}, SLA_S\}$
$H \leftarrow HmetricQuery(msg_{help}, n)$
$A \leftarrow \forall h \in H \mid isAccepting(h)$
$dest \leftarrow min(A)$
return $move(S, dest)$

5 Prototype and Evaluation

We have implemented the autonomic WSRF-P2P container in a Geronimo [6] application server. The P2P Node and the autonomic system manager are implemented as Geronimo Beans (GBeans; [14]). The P2P functionality is provided by an extension of freePastry [15], however it is not critical which structured DHT package is used. The prototype has been deployed and tested on five machines. However, to properly test the performance of the architecture we ran the deployable prototype code but used the freePastry overlay simulation mode to scale the simulation to 100 containers.

5.1 Results

To simplify analysis the only SLA parameter we considered was response time. The experimental configuration was 450 services deployed over 100 containers. Each service had an expected response time of 10ms, with 75 gold, 150 red and

225 green level SLAs. To explore how our architecture responds to, and resolves SLA violations, all services were initially deployed to containers at random. It is worth pointing out however, that deployments are usually made much more carefully using the same H-metric query as migration. Hence these experiments show that the architecture can deal with a poor initial distribution and can resolve the distribution of services to provide the QoS dictated by the differentiated SLAs. The vast bulk of violations and subsequent rectification migrations are finished in 15-25 seconds after initial deployment when using a query depth of 3.

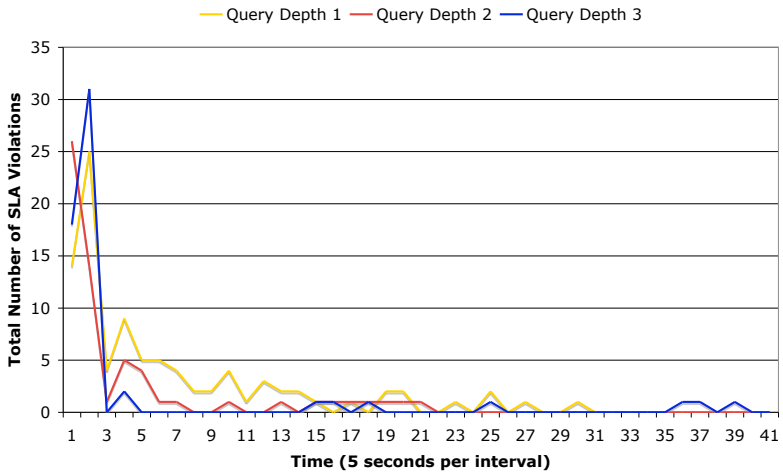

Fig. 4. Total number of SLA violations for bounded H-metric queries.

Figure 4 shows how quickly the system manages to stabilise at the agreed QoS after the initial random deployments. The slowest to stabilise on the agreed QoS was when the destination was selected using a query depth of 1 (random), with depth 2 and 3 improving things respectively. There are some single violations long after the bulk of the migrations have finished, and this is due to the same destination being chosen by two offloading containers (it is a fully decentralised algorithm). The deeper the query depth, the faster the system provides the agreed QoS to the vast majority of hosted services.

Figure 5 shows the number of gold, red and green service violations after initial deployment for a H-metric query depth of 2. The results for level 1 and level 3 both show similar decay curves but differ in the rate at which they stabilise on the agreed QoS (as in Figure 4). It is worth remembering that in response to any of these violations, a green (then red if no green, then gold) service is chosen to migrate from the host. Each of these violations results in a service migration, however the vast bulk of the migrated services are green, with few reds and minimal gold services migrating. Finally Figure 6 shows the total number of violations for each query depth and service level domain. Here the reduction in the number total of violations as more effort is put into finding the

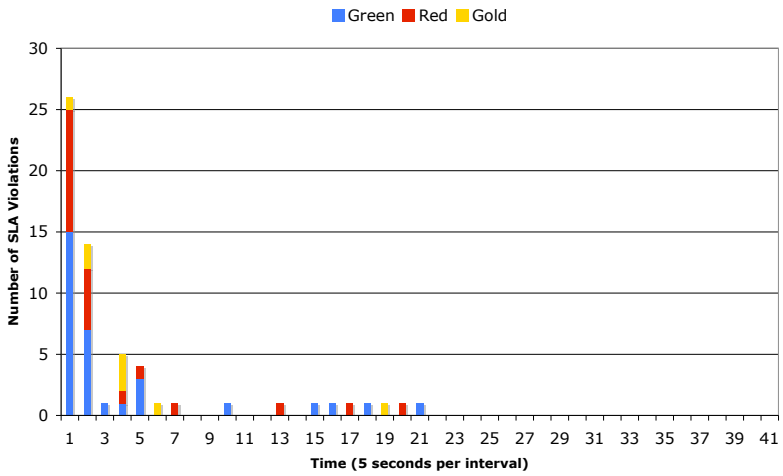

Fig. 5. SLA violations shown by service level for a H-metric query depth of 2.

best destination is clear, although the improvement due to the query depth is subject to diminishing returns.

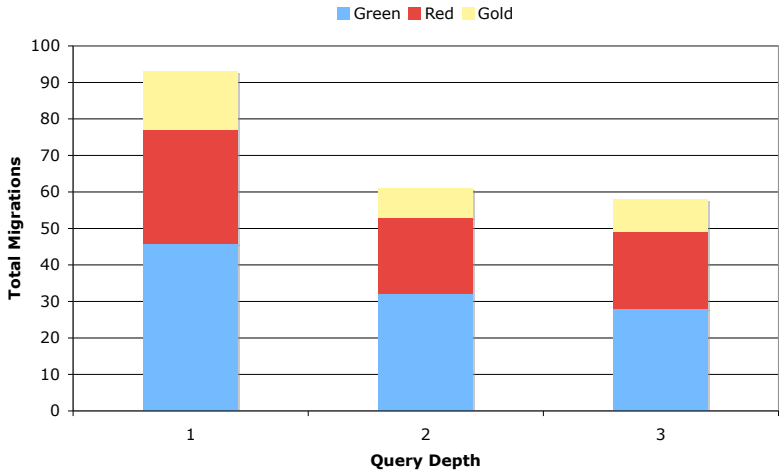

Fig. 6. Total number of violations for each service level for bounded H-metric queries.

6 Related Work

There have been a number of projects focusing on autonomic behaviour for managing web services, in particular Ecosystem [4] 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 [16] 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 for, such as that by El-Darieby and Krishnamurthy [17], which partitions resources into individual, cluster and grid resources. Dowlatshahi et. al [18] 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 [19] has been implemented by Butt et. al [20]. 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 [21] 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[22] 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. [23] present an applied self-reconfiguration approach to support disconnected operations by allowing the system to monitor and automatically redeploy itself.

Berenbrink et. al [24] 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 [25] aims at proving 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 [26], which uses a P2P structured DHT, 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.

7 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 SLA specifications.

We have demonstrated that our autonomic SLA aware containers, that monitor their SLA compliance and migrate excess services to other containers with spare capacity, can react to dynamic to runtime conditions. The rate at which the system is capable of redistributing services to find a QoS preserving distribution is very fast, and improves further as the H-metric query depth increases. We have also provided differentiated service level domains (green, red and gold) in our SLAs, and using the health metric and service level domains we have developed a decentralised migration algorithm that redistributes services between containers to meet agreed QoS. This is done in a distributed, scalable and robust way.

Finally, we have implemented the architecture using standard modern technologies and with high levels of transparency, indeed, conventional webservices can be deployed with the addition of a SLA specification.

References

1. OASIS: Web services resource framework (wsrf). Home-Page: `http://www.oasis-open.org/`
2. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F., Krämer, B.J.: Service-oriented computing: A research roadmap. In Cubera, F., Krämer, B.J., Papazoglou, M.P., eds.: Service Oriented Computing (SOC). Number 05462 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
3. Parashar, M., Hariri, S.: Autonomic computing: An overview. In et al., J.P.B., ed.: Unconventional Programming Paradigms. Volume 3566., Mont Saint-Michel, France, Springer Verlag (2005) 247–259
4. Li, Y., Sun, K., Qiu, J., Chen, Y.: 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), Washington, DC, USA, IEEE Computer Society (2005) 266–273
5. IBM: An architectural blueprint for autonomic computing. IBM (2004) Home-Page: `http://www-3.ibm.com/autonomic/pdfs/ACwpFinal.pdf`.
6. Apache: Geronimo Home-Page: `http://geronimo.apache.org/`.
7. Apache: Axis2/java Home-Page: `http://ws.apache.org/axis2/`.
8. Apache: Tomcat Home-Page: `http://tomcat.apache.org/`.
9. Sun: Jsr-77: J2ee management specification. Home-Page: `http://jcp.org/en/jsr/detail?id=77`
10. Sun: Sun's java management extensions (jmx) page Home-Page: `http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/`.

11. Hanson, J.J.: Manage apache geronimo with jmx. (August 2006)
12. Fowler, M.: Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html (January 2004)
13. Rowstron, A., Druschel, P.: IFIP/ACM international conference on distributed systems platforms (middleware). In: Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems., Heidelberg, Germany (Nov. 2001) 329–350
14. Apache: Geronimo user guide. Home-Page: http://cwiki.apache.org/GMOxDOC11/apache-geronimo-v11-users-guide.html
15. Druschel, P., Rowstron, A.: freepastry software. http://freepastry.org/
16. Hao, W., Gao, T., Yen, I.L., Chen, Y., Paul, R.: 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), Washington, DC, USA, IEEE Computer Society (2006) 41–48
17. El-Darieby, M., Krishnamurthy, D.: A scalable wide-area grid resource management framework. In: ICNS '06. International conference on Networking and Services, 2006., Silicon Valley, USA, IEEE Computer Society Press (July 2006) 76 – 86
18. Dowlatshahi, M., MacLarty, G., Fry, M.: A scalable and efficient architecture for service discovery. In: The 11th IEEE International Conference on Networks, 2003. ICON2003. (September 2003) 51 – 56
19. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. Concurrency - Practice and Experience **17**(2-4) (2005) 323–356
20. Butt, A., Zhang, R., Hu, Y.: A self-organizing flock of condors. In: Supercomputing, 2003 ACM/IEEE Conference, Purdue University, West Lafayette, IN, ACM Press (Nov. 2003) 42–42
21. Kang, C., Park, K., Kim, S.: A differentiated service mechanism considering sla for heterogeneous cluster web systems. In: Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop on. (27-28 April 2006) 6pp.
22. Lee, B.Y., Lee, G.H.: Service oriented architecture for sla management system. In: Advanced Communication Technology, The 9th International Conference on. Volume 2. (Feb. 2007) 1415–1418
23. Mikic-Rakic, M., Medvidovic, N.: Support for disconnected operation via architectural self-reconfiguration. In: ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04), Washington, DC, USA, IEEE Computer Society (2004) 114–121
24. Berenbrink, P., Friedetzky, T., Goldberg, L.A., Goldberg, P., Hu, Z., Martin, R.: Distributed selfish load balancing. In: SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, New York, NY, USA, ACM Press (2006) 354–363
25. Zeid, A., Gurguis, S.: Towards autonomic web services. In: The 3rd ACS/IEEE International Conference on Computer Systems and Applications. (2005) 69
26. Mondejar, R., Garcia, P., Pairot, C., Gomez Skarmeta, A.: 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., Manchester, UK (June 2006) 89 – 94