Using Process Restarts to Improve Dynamic Provisioning

Raquel V. Lopes, Walfredo Cirne, and Francisco V. Brasileiro

Universidade Federal de Campina Grande,
Coordenação de Pós-graduação em Engenharia Elétrica
Departamento de Sistemas e Computação
Av. Aprígio Veloso, 882 - 58.109-970, Campina Grande, PB, Brazil
Phone: +55 83 310 1433
{raquel,walfredo,fubica}@dsc.ufcg.edu.br

Abstract. Load variations are unexpected perturbations that can degrade performance or even cause unavailability of a system. There are efforts that attempt to dynamically provide resources to accommodate load fluctuations during the execution of applications. However, these efforts do not consider the existence of software faults, whose effects can influence the application behavior and its quality of service, and may mislead a dynamic provisioning system. When trying to tackle both problems simultaneously the fundamental issue to be addressed is how to differentiate a saturated application from a faulty one. The contributions of this paper are threefold. Firstly, we introduce the idea of taking software faults into account when specifying a dynamic provisioning scheme. Secondly, we define a simple algorithm that can be used to distinguish saturated from faulty software. By implementing this algorithm one is able to realize dynamic provisioning with restarts into a full server infrastructure data center. Finally, we implement this algorithm and experimentally demonstrate its efficacy.

Keywords: dynamic provisioning, software faults, restart, n-tier applications.

1 Introduction

The desire to accommodate load variations of long running applications is not new. Traditionally, it has been made by overprovisioning the system [1,2]. Recently, dynamic provisioning has emerged, suggesting that resources can be provided to an application on an on-demand basis [3,4,5,6,7,8,9,10,11]. Dynamic provisioning is particularly relevant to applications whose workload vary widely over time (i.e. where the cost of overprovisioning is greater). This is the case of e-commerce applications, which typically are n-tier, long running applications that cater for a large user community.

We have experimented with a dynamic provisioning scheme that targeted a simple 2-tier application. To our surprise, we noticed that even when more

A. Sahai and F. Wu (Eds.): DSOM 2004, LNCS 3278, pp. 220–231, 2004.
© IFIP International Federation for Information Processing 2004

resources had been provided, application quality of service (QoS) had still remained low. Investigating further, we found out that the application failed due to hard-to-fix software bugs such as Heisenbugs [12] and aging related bugs [13]. Thus, the real problem was that dynamic provisioning systems use functions that relate system metrics (load, resource consumption, etc.) to the number of machines to be provided to the application [5,6,14]. However, when software faults occur, these functions may not reflect the reality anymore, since there are components of the application that are up, consuming resources, processing requests, but not performing according to their specifications anymore.

In fact, there is a close relationship between load and software faults. Saturated[1] applications are more prone to failures [1]. They are more susceptible to race conditions, garbage collector misbehavior, and so on, increasing the probability of occurrence of non-deterministic bugs, such as Heisenbugs. Because of this close relationship, we argue that a management system must deal with both of them in a combined fashion. This dual goal system must be able to decide between add/release resources (dynamic allocation actions) and restart software (software fault recovery).

The contributions of this paper are threefold. First, we introduce the importance of taking software faults into account when conceiving a dynamic provisioning scheme. Second, we define a simple algorithm that can be used to differentiate saturated from faulty software. Third, we implement this algorithm to experimentally evaluate its efficacy in the context of a full server infrastructure data center. Our results indicate that by taking into account both load variability and software faults, we can improve the quality of service (QoS) and yet reduce the resource consumption, compared to doing solely dynamic provisioning.

The remaining of the paper is structured in the following way. In the next section we discuss related work. Next, in Section 3, we show a control system that makes decision in order to identify the application status (saturated, faulty, optimal or underutilized) and act over an n-tier application. This system is named Dynamic Allocation with Software Restart (DynAlloc-SR). Then, in Section 4, we present some preliminary results obtained from experiments carried out to measure DynAlloc-SR efficacy. Finally, we conclude the paper and point future research directions in Section 5.

2 Related Work

Our research is related to two important areas: dynamic provisioning of resources and usage of software reboots as a remedy for soft software bugs. In the following we discuss related works in these areas and point out the novelty of our approach.

2.1 Autonomic Data Centers

Many research groups have been studying the issue of dynamic provisioning of resources in data centers (DCs). For an autonomic DC to come to reality,

[1] The words saturated and overloaded are used interchangeably in this paper.

some problems must be solved. One of them is to know the optimal amount of resources to give to an application on demand [3,5,6,14], which is our focus. Other issues involve DC level decisions such as whether agents requests for resources are going to be accepted, whether new applications are going to be accepted by the DC and whether DC capacity is appropriate [3,8,9,10]. Finally, technologies that enable rapid topology reconfiguration of virtual domains are needed [4,11].

In [5] authors present an algorithm that indicates the amount of machines a clustered application needs to accommodate the current load. The algorithm makes decisions based on CPU consumption and load. That work considers a full server infrastructure, where each server runs the application of only one customer at each time. A market-based approach that deals with the allocation of CPU cycles among client applications in a DC is presented in [3]. CPU consumption in servers is the monitored metric that must be maintained around a set point. A dynamic provisioning mechanism based on applications models is proposed in [6]. Application performance models relate application metrics to resource metrics and can be used to predict the effect of allotments on the application. Both [3] and [6] consider a shared server data center infrastructure, in which different applications may share the same server. An approach for dynamic surge protection is proposed in [14] to handle unexpected load surges. Resource provisioning actions are based on short and long term load predictions. The authors argue that this approach is more efficient than a control system based in thresholds. Clearly, both have advantages and disadvantages. A dynamic surge protection system is as good as the predictions it does. A threshold-based system is as good as the threshold values configured. Finally, other researchers proposed frameworks to help developers to write scalable clustered services [7,15]. Only applications in development can benefit from these frameworks.

As [5] we consider a full server infrastructure. However, we monitor application performance metrics instead of system consumption metrics, and act over the system as soon as possible in order to maintain the average availability and response times of the application around a set point. Our provisioning system is based only on QoS threshold values. Load tendency is taken into account only to reinforce a resource provisioning decision. Our approach needs neither application performance models nor specific knowledge about the implementation of the application. We also do not require modification in the application code nor in the middleware. Finally, our provisioning approach is able to detect when the degraded performance is due to data layer problems, in which case, actions in the application or presentation layers do not take effect.

2.2 Recovering from Software Faults

Some software bugs can escape from all tests and may manifest themselves during the application execution. Typically, they are Heisenbugs and aging related bugs. Both are activated under certain conditions which are not easily reproducible.

Software rejuvenation has been proposed as a remedy against the software aging phenomenon [16]. Rejuvenation is the proactive rollback of an application to a clean status. Software aging can give signs before causing a failure. As a

result, they can be treated proactively. A similar mechanism named restart has been prescribed for Heisenbugs recovery, however, on a reactive basis [17,18].

Rejuvenation can be scheduled based on time (eg every Monday, at 4 a.m.), on application metrics (eg memory utilization) or on the amount of work done (eg after n requests processed) [16]. [19] and [20], for instance, try to define the best moment to rejuvenate long running systems based on memory consumption. [19] defines multiple levels of rejuvenation to cope with different levels of degradation. [21] formally describes a framework that estimates epochs of rejuvenation. They distinguish memory leakage and genuine increase on the level of memory used by a leak function (each application may have its function) that models the leaking process. The amount of leaked memory of some application can be studied by using tools to detect application program errors [22]. However, these tools are not able to detect leaks automatically during the execution of the application. Methods to detect memory leaks still require human intervention.

The execution of micro-reboots is one technique proposed in [18] to improve the availability of J2EE (Java 2 Platform, Enterprise Edition) applications by reducing the recovering time. Candea et al consider any transient software fault, not only Heisenbugs or aging related bugs. Their technique is application-agnostic, however, it requires changes in the J2EE middleware.

Our restart approach is a simplification of the one presented in [18]. We perform reactive restarts when the application exhibits bad behavior. Our restarts are always at the middleware level. We try to differentiate saturation and software faults without requiring any knowledge of the application being managed neither modification in the middleware that supports its execution. We name our recovery method restart, not rejuvenation, because of its reactive nature.

3 Dynamic Provisioning with Software Restart

DynAlloc-SR is a closed-loop control system that controls n-tier applications through dynamic provisioning and process restart. Its main components are showed in Figure 1. The managed application is an n-tier Web based application. Typically, an n-tier application is compounded of layers of machines (workers). A worker of a layer executes specific pieces of the application. For a 3-tier application, for example, there is the load balancer and workers that execute presentation, application and data layer logic. A Service Level Agreement (SLA) specifies high level QoS requirements that should be delivered to the users of the application. It defines, for instance, response time and availability thresholds. The decision layer is aware of the whole application health status. Thus, it is the one who makes decisions and executes them by actuating over the execution environment of the application. It can choose among four different actions to execute: i) add workers; ii) remove workers; iii) restart a worker software, and iv) do nothing. At the monitoring layer there are the components that produce management information to the decision layer.

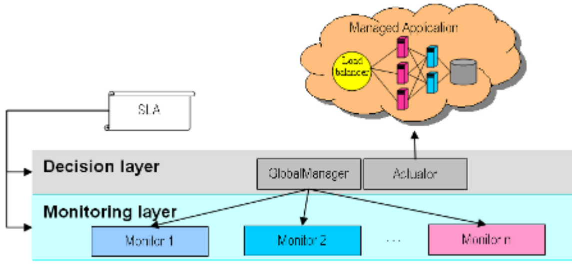

Fig. 1. DynAlloc-SR architecture

3.1 The Monitoring Layer

Monitoring components collect information from each active worker of the application. Monitoring information collected from the load balancer (LB) is related to the application as a whole because the LB is a central point on which the application depends. We call it application level monitoring information. For the other workers, monitoring information is called worker level information. These two levels of information give insight about the health status of the application and allow the detection of workers that are degrading the application QoS.

DynAlloc-SR gets monitoring information in two ways: (i) submitting probe requests, and (ii) analyzing application logs. Probe requests are used to capture the quality of the user experience. Success responses occur when the response time is less than a threshold (specified in the SLA) and does not represent an error. Logs provide information on load, response times and availability. Logs can also be processed to obtain tendencies of such metrics.

Both ways of gathering information have pros and cons. Application logs contain average response times and availability offered to the stream of real users. Moreover, they are available for free, since they are produced regardless of DynAlloc-SR and do not require modifications in the applications. However, logs may miss certain failures. For example, opening TCP connections to a busy server may fail, but the server will never know about it (and thus will not log any event). This problem does not affect probe requests, since they are treated by the application as a regular user request. On the other hand, probe requests bring intrusiveness because they add to the application load. Hence, to be as close as possible to the real user experience, and at the same time to be as unintrusive as possible, we combine both methods to infer the quality of user experience.

Each probe sends requests to a specific worker (WorkerProbe - WP) or to the load balancer (ApplicationProbe - AP) periodically. Some of these requests depend only on the services of the layer of the worker in question and others depend on services offered by other layers. Probes analyze the responses received and suggest actions to the decision layer instead of sending raw monitoring data.

WPs can suggest actions such as doNothing, addWorkers, otherLayerProblem and restart. The action doNothing means that all responses received did not exceeded the SLA threshold for response times and do not represent errors.

HTTP (Hyper Text Transfer Protocol) probes, for instance, consider good responses those who carry response code 2xx. A probe proposes addWorkers when at least one of the responses for the requests that do not depend on other layers' services do not represent errors but are exceeding the response time threshold specified in the SLA. A WP suggests restart when responses that represent errors are received even for the requests that do not depend on another layers. Finally, otherLayerProblem is proposed when all responses exceeding the SLA response time threshold or representing errors depend on other layers services.

The AP sends requests to the LB. If all responses received do not represent errors it proposes doNothing. Otherwise, it suggests addWorkers. The AP does not suggest restart actions nor otherLayerProblem because it has no idea about which worker may be degrading the QoS.

DynAlloc-SR also monitors other application metrics in the LB: availability and response times offered to the real users and load tendency. All of them are computed by processing logs. Availability and response times are well known metrics that do not need extra explanations. The tendency metric informs if the application is more or less loaded during a given monitoring interval in comparison with the previous one.

3.2 The Decision Layer

The GlobalManager (GM) is the decision layer component that periodically collects probes' suggestions and other metrics (availability, idleness, etc.) from monitoring layer components. Eventually, the GM can try to collect monitoring information while probes/monitors are still collecting new information. In this case, GM will use the last information collected. It correlates the information received, makes decisions and actuates over the application. More specifically, it correlates all monitoring metrics received and decides if the probes' suggestions must be applied.

In a first step, GM uses the application response times and availability computed by analyzing logs and the WPs suggestions to discover if the LB is a source of performance degradation. If response times or availability violates the SLA thresholds and all WPs suggest doNothing then GM restarts the LB software.

Next, the GM separates WPs' suggestions as well as the AP suggestion by layer into sets. There are 4 sets for each layer: doNothing, addWorkers, restart and otherLayerProblem. Probes' suggestions are organized as elements of these sets. A layer is considered overloaded if the cardinality of its addWorkers set is greater than a minimum quorum. One layer receives resources only if it is overloaded. We consider load balancing is fair among workers, thus, saturation happens to a set of workers at the same time. The minimal quorum is a mechanism to distinguish scenarios in which an increasing in capacity is actually needed and scenarios in which some pieces of the application are degraded. If less than the quorum has proposed an increase in capacity, the GM will restart the workers that proposed addWorkers instead of increasing the layer capacity. To give some time for the load to be rebalanced, the GM waits for some time before adding new workers, even if all conditions for a new increasing are satisfied.

The status of a layer could be degraded due to problems in other layers. The GM does not act over a layer if the cardinality of the otherLayerProblem set is greater than a quorum. It follows the same minimal quorum reasoning as for capacity increasing. If less than the quorum has proposed otherLayerProblem the GM decides to restart the software of those who suggested otherLayerProblem.

The restart process does not depend on the layer status, but on individual status of each worker. Thus, all restart suggestions are implemented by the GM as soon as possible to avoid the worker to degrade even more along the time.

Workers of the data layer do not depend on services of other layers. If problems in this layer are detected the GM forwards monitoring information to a database agent that can act over the database. A lot of new challenges are involved in the dynamic provisioning of the data layer and some systems are concerned with the load variability problem [23].

A worker is removed if DynAlloc-SR remains some period without the need to act over the application and the majority of the load tendencies collected during this period indicates load reduction. In this case, the load balancing stops sending requests to the oldest worker and when this worker has no requests to process it returns to the pool of free servers, as proposed in [5].

4 DynAlloc-SR Experimental Analysis

A prototype of DynAlloc-SR was implemented as well as a system named DynAlloc, which does not perform restarts. It increments the application capacity as soon as any signal of QoS degradation is seen. We compared the average availability and response times of the managed application as well as the number of machines used by each system in order to measure the efficacy of DynAlloc-SR.

4.1 DynAlloc-SR and DynAlloc Prototypes

Our prototypes were conceived to manage 2-tier applications. HTTP is used between the client and the presentation layer. There is a WP sending requests to each active worker of the presentation layer and an AP sending requests to the LB. All probes analyze the responses received as described in 3.1. Following the HTTP specification [24], only HTTP response codes 2xx are considered success.

A WP can suggest otherLayerProblem when it detects that the responses were unsuccessful due to poor QoS of data layer components. Each WP sends different kinds of requests to the probed worker: some that require data layer access and others that do not require[2]. If only the DB queries have delivered bad QoS, then the probe suggests otherLayerProblem.

The minimum quorum used by DynAlloc-SR is “majority”. We could have used “all”, but reach unanimity in such an asynchronous distributed environment is very unlikely. When one probe suggests addWorkers, others may be still waiting the workers’ replies.

[2] We plan to send requests directly to DB workers using Java Database Connectivity.

Currently, the database agent of the DynAlloc-SR prototype does nothing. Thus we are not acting over the data layer for now.

DynAlloc-SR knows a pool of machines it can use. Each of these machines is either an idle machine or is an active worker running pieces of the application. When restarting a worker, DynAlloc-SR first verifies if there is an idle machine. In this case it prepares one of them with the appropriate software, adds it to the pool of active workers and only then stops the faulty worker and returns it to the pool of idle machines. When the pool of idle machines is empty, the rejuvenation action stops the faulty software and then restarts it in the same machine. Since this operation takes some seconds, the number of active workers is temporarily reduced by one during restart. Clearly, the first way of restart is more efficient than the second. This is yet another advantage of combining dynamic provisioning with software restart in the same system.

DynAlloc has the same monitors as DynAlloc-SR, however, its GM does not take into account the minimal quorum. It increases the application capacity as soon as some QoS degradation is perceived and ignores suggestions of restart.

4.2 Testbed

The managed application is a mock-up e-commerce application named xPetstore[3]. In our experiments, after sometime running, we observed one of the following flaws (in order of frequency of occurrence): EJBException, ApplicationDeadlockException or OutOfMemoryError.

The testbed consists of 5 application servers running JBoss 3.0.7 with Tomcat 4.29, one database server running MySQL, one LB running Apache 2.0.48 with mod_jk, 3 load generators and a manager that executes either DynAlloc-SR or DynAlloc. We start an experiment using 2 workers.

Obtaining actual logs from e-commerce companies is difficult because they consider them sensitive data. Thus, we use synthetic e-commerce workloads generated by GEIST [25]. Three workload intensities have been used: the low load, with 120 requests per minute (rpm) in average, the medium load with 320 rpm and the high load with 520 rpm. Each workload lasts for around an hour and presents one peak. Based on the study reported in [26] we assume that the average number of requests per minute doubles during peaks.

DynAlloc-SR and DynAlloc availability and response times thresholds are 97% and 3 seconds respectively.

4.3 Experimental Results

We here present results obtained by running each experiment ten times. Average values of availability and response times measured during all experiments are presented in Table 1. DynAlloc-SR yielded better average application availability and response times than DynAlloc. This is an indication that although

[3] xPetstore is a re-implementation of Sun Microsystem PetStore, and can be found in http://xpetstore.sourceforge.net/. Version 3.1.1 was used in our experiments.

very simple, DynAlloc-SR is able to make good choices when adding/releasing resources and restarting software.

Table 1. Average availability and response times of xPetstore

	DynAlloc-SR	*DynAlloc*
Availability	88.04%	77.99%
Response times	49.65 sec.	136.37 sec.

Next we compare DynAlloc-SR and DynAlloc considering each load intensity individually. These results are illustrated in Figures 2 and 3.

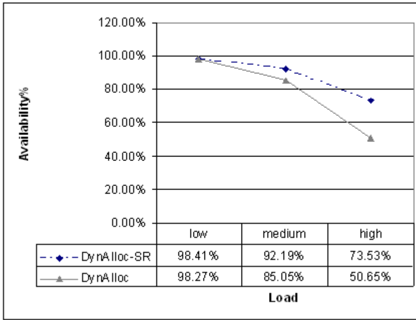

Fig. 2. Average availability

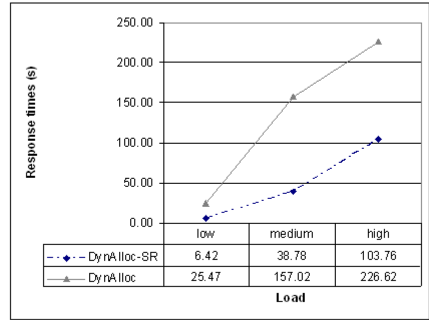

Fig. 3. Average response times

In average, DynAlloc-SR yielded 0.13%, 7.14% and 22.9% better availability than DynAlloc during the low, medium and high load experiments respectively. As we can see, the availability gain increases considerably when the load increases. The more intense is the load and the more saturated is the software, the greater is the probability of failures due to software faults. This is because, when load increases, the probability of race conditions, garbage collector misbehavior, acceleration of process aging, etc., also increases. This correlation makes the differentiation between faulty and overloaded software difficult.

Response time gains did not follow the same crescent pattern. For low and medium loads the gains were around 75%. For the higher load this gain was smaller (54%). Investigating further, we found out that the LB reached its maximum allowed number of clients and became a bottleneck during high load experiments. The Apache MaxClients directive limits the number of child processes that can be created to serve requests. Any connection attempt over this limit is queued, up to a number based on the ListenBacklog directive. Since DynAlloc-SR cannot actuate over the capacity of the LB, the response times increased and, thus, the gain around 75% was not achieved.

These high gains may be an indicative of the fragility of the application and its execution environment. It is likely that applications in production are more robust than the one we used here and thus these gains may be overestimated. However, applications will fail someday, and when this happens, a dynamic provisioning with software restart will deliver better results than a dynamic provisioning system that does not take software faults into account.

DynAlloc-SR also used less resources than DynAlloc. The mean number of machines used by DynAlloc was 14.6% greater than the mean number of machines used by DynAlloc-SR. This is due to the fact that workers that needed rejuvenation contributed very little to the application QoS, yet kept consuming resources. When looking at the load variation, the mean number of machines used by DynAlloc was 5.3%, 15.0% and 20.5% greater than the mean number of machines used by DynAlloc-SR, for low, medium and higher loads, respectively. We believe the raise from 5.3% to 15.0% in "resource saving" is due to the greater number of software failures generated by a greater load. Interestingly, however, this phenomena does not appear when we go from medium to high load: the increase in "resource saving" is of five percent points (from 15% to 20%). We believe that this is due to the maximum number of machines used. The maximum number of machines used (5) is more than enough to process the low load. However, when load increases DynAlloc tries to correct the degraded performance of the application by adding more machines, always reaching the maximum number of machines. If more machines were available, more machines would be allocated to the application. If the total number of machines was higher than 5, the "resource saving" for high load would likely be greater.

5 Conclusions and Future Works

We have shown that a dynamic provisioning system that takes into account software faults is able to deliver better application QoS using less resources than a similar dynamic allocation system that does not consider software faults. One of the most complex duties of such a dual goal management system is to differentiate saturated and faulty software. This is because aging related bugs produce effects in the application and the execution environment that are similar to those produced by load surges. Moreover, the probability of failure is proportional to the load intensity being hold by the application, turning the relationship between load and bugs still narrower. We here propose DynAlloc-SR, a control system that copes not only with capacity adjustment, but also with software faults of n-tier applications. DynAlloc-SR assumes that saturation is something that always happens simultaneously to a minimal quorum of workers of the same layer while software faults do not follow this clustered pattern.

Our experimental results indicate the efficacy of DynAlloc-SR decision algorithms. By combining a restart scheme with our dynamic allocation scheme, we could increase the average application availability from 78% to 88%. Maybe this improvement is overestimated due to the fragility of our demo application. However, even n-tier applications in production fail. When applications fail, dynamic

provisioning with software restart will deliver better results than a dynamic provisioning system that does not take software faults into account. The dual goal system also uses less resources than the dynamic allocation only system. DynAlloc used in average 15% more resources than DynAlloc-SR.

Our main goal here is not to propose a perfect dynamic provisioning algorithm but to demonstrate the importance of treating software faults in conjunction with dynamic provisioning. However, we emphasize two important features of our dynamic allocation scheme that, as far as we know, had not been applied by other schemes. Firstly, we consider dependency relationships among n-tier application layers. For instance, DynAlloc-SR does not try to add more machines in a layer L if another layer L' on which layer L depends presents poor performance. Secondly, DynAlloc-SR uses probes to infer the application health and do not depend on correct behavior of the application, since we do not use specific functions that relates QoS metrics with number of machines.

Before we proceed with the study of a combined solution to the problems of load variability and software faults, we plan to study deeper the interactions among dynamic provisioning systems, rejuvenation schemes and degradation/failure phenomena due to transient software faults. Based on the interactions discovered we hope to define new techniques in both areas (software faults recovery and dynamic provisioning), which maximize/create positive interactions or minimize/eliminate negative ones.

Acknowledgments. This work was (partially) developed in collaboration with HP Brazil R&D and partially funded by CNPq/Brazil (grants 141655/2002-0, 302317/2003-1 and 300646/1996-8).

References

1. Gribble, S.D.: Robustness in complex systems. In: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems. (2001) 21–26
2. Ejasent: Utility computing: Solutions for the next generation IT infrastructure. Technical report, Ejasent (2001)
3. Chase, J.S., Anderson, D.C., Thakar, P.N., Vahdat, A., Doyle, R.P.: Managing energy and server resources in hosting centres. In: Symposium on Operating Systems Principles. (2001) 103–116
4. Appleby, K., et al: Oceano - sla based management of a computing utility. In: 7th IFIP/IEEE International Symposium on Integrated Network Management. (2001) 855 –868
5. Ranjan, S., Rolia, J., Fu, H., Knightly, E.: Qos-driven server migration for internet data centers. In: Proceedings of the International Workshop on Quality of Service. (2002)
6. Doyle, R., Chase, J., Asad, O., Jen, W., Vahdat, A.: Model-based resource provisioning in a web service utility. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems USITS 2003. (2003)
7. Fox, A., Gribble, S.D., Chawathe, Y., Brewer, E.A., Gauthier, P.: Cluster-based scalable network services. In: Proceedings of the 6th ACM Symposium on Operating Systems Principles, ACM Press (1997) 78–91

8. Rolia, J., Zhu, X., Arlitt, M.F.: Resource access management for a utility hosting enterprise applications. In: Proceeding of the 2003 International Symposium on Integrgated Management. (2003) 549–562
9. Rolia, J., Arlitt, M., Andrzejak, A., Zhu, X.: Statistical service assurancecs for applications in utility grid environments. In: Proceedings of the Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telcommunication Systems. (2003) 247–256
10. Rolia, J., et al: Grids for enterprise applications. In Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., eds.: Job Scheduling Strategies for Parallel Processing. Springer Verlag (2003) 129–147 Lect. Notes Comput. Sci. vol. 2862.
11. Rolia, J., Singhal, S., Friedrich, R.: Adaptive internet data centers. In: In SS-GRR'00 Conference. (2000)
12. Gray, J.: Why do computers stop and what can be done about it? In: Symposium on Reliability in Distributed Software and Database Systems. (1986)
13. Vaidyanathan, K., Trivedi, K.S.: Extended classification of software faults based on aging. In: Proceedings of the 12th International Symposium on Software Reliability Engineering. (2001)
14. Lassettre, E., et al: Dynamic surge protection: An approach to handling unexpected workload surges with resource actions that have dead times. In: 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management. Volume 2867 of Lecture Notes in Computer Science., Springer (2003) 82–92
15. Welsh, M., Culler, D., Brewer, E.: Seda: an architecture for well-conditioned, scalable internet services. In: Proceedings of the 8th ACM Symposium on Operating Systems Principles, ACM Press (2001) 230–243
16. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.D.: Software rejuvenation: Analysis, module and applications. In: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing, IEEE Computer Society (1995) 381–390
17. Candea, G., Fox, A.: Recursive restartability: Turning the reboot sledgehammer into a scalpel. In: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems. (2001) 125–132
18. Candea, G., Keyani, P., Kiciman, E., Zhang, S., Fox, A.: Jagr: An autonomous self-recovering application server. In: 5th International Workshop on Active Middleware Services. (2003)
19. Hong, Y., Chen, D., Li, L., Trivedi, K.: Closed loop design for software rejuvenation. In: Workshop on Self-Healing, Adaptive, and Self-Managed Systems. (2002)
20. Li, L., Vaidyanathan, K., Trivedi, K.S.: An approach for estimation of software aging in a web server. In: International Symposium on Empirical Software Engineering. (2002)
21. Bao, Y., Sun, X., Trivedi, K.S.: Adaptive software rejuvenation: Degradation model and rejuvenation scheme. In: Proceedings of the 2003 International Conference on Dependable Systems and Networks, IEEE Computer Society (2003) 241–248
22. Erickson, C.: Memory leak detection in embedded systems. Linux Lournal (2002)
23. Oracle: Oracle database 10g: A revolution in database technology. Technical report, Oracle (2003)
24. Fielding, R., et al: Hypertext transfer protocol – http/1.1. Technical report, RFC 2616 (1999)
25. Kant, K., Tewari, V., Iyer, R.: Geist: A generator of e-commerce and internet server traffic. In: Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software, IEEE Computer Society (2001) 49–56
26. Arlitt, M., Krishnamurthy, D., Rolia, J.: Characterizing the scalability of a large web-based shopping system. ACM Trans. Inter. Tech. **1** (2001) 44–69