Can dynamic provisioning and rejuvenation systems coexist in peace?

Raquel Lopes, Walfredo Cirne, Francisco Brasileiro, Eduardo Colaço

Universidade Federal de Campina Grande,
Departamento de Sistemas e Computação
Laboratório de Sistemas Distribuídos
Av. Aprígio Veloso, 882 - 58.109-970, Campina Grande, PB, Brazil
Phone: +55 83 310 1365
{raquel,walfredo,fubica,eduardo}@dsc.ufcg.edu.br

Abstract. Dynamic provisioning systems change application capacity in order to use enough resources to accommodate current load. Rejuvenation systems detect/forecast software failures and temporarily remove one or more components of the application in order to bring them to a clean state. Up to now, these systems have been developed unaware of one another. However, many applications need to be controlled by both. In this paper we investigate whether these systems can actuate over the same application when they are not aware of each other, i.e., without coordination. We present and apply a model to study the performance of dynamic provisioning and rejuvenation systems when they actuate over the same application without coordination. Our results show that when both systems coexist application quality of service degrades in comparison with the quality of service provided when each system is acting alone. This suggests that some level of coordination must be added to maximize the benefits gained from the simultaneous use of both systems.
Keywords: interacting systems, dynamic provisioning, rejuvenation.

1 Introduction

There are many systems that aim at automating management tasks and decisions. We are particularly interested in two of them: dynamic provisioning systems (DPS) and Rejuvenation/Restart[1] systems (RRS). DPSs have been proposed to automatically adjust the application capacity to its demand [1,2,3]. RRSs have been proposed to tackle software failures by detecting such failures and bringing the system to a clean state [4,5,6,7]. These systems have been studied separated from each other. For instance, DPSs do not take software failures nor restarts into account, while RRSs do not consider the dynamic capacity changing and the strict matching between capacity and demand provided by a DPS.

Nevertheless, one would expect that some applications could benefit from the use of two or more automated management systems simultaneously. In fact, in [8]

[1] Restart and rejuvenation are used here as synonyms.

we have implemented a DPS with RRS features and have obtained good results. When both features coexist application quality of service (QoS) was better and resource savings were higher. Since we built that system from scratch, it was natural to introduce some coordination between the management systems. By coordination we mean that some information can be exchanged between the DPS and the RRS in order to improve their performance. Two coordination features were implemented: (i) whenever the capacity decreases the nodes more prone to failure were selected to be removed. This information is provided for the DPS component by the RRS monitor; and (ii) instead of restarting a node, the RRS component asks the DPS component to add one more node, if possible, and then to remove the faulty one.

Ideally, instead of developing new systems, we would like to harness the potential of legacy systems independently designed. Since these systems are designed independently, they do not assume the existence of the other and, therefore, do not exchange any information. In orther words, there is no imposed coordination between their actions; they do not interact directly, but the actions of one system may interfere in the actions of the other and their actions together may influence the QoS of the managed application and its operational cost. We argue it is crucial to identify whether some imposed coordination is really needed to make these systems harmoniously coexist. This work is a first step towards better understanding the effects of having an application managed by both a DPS and an RRS that act independently.

Contributions of this paper are twofold. First, we propose a component based model to study interactions between dynamic provisioning and restart systems. Instances of this model can consider different sets of components, allowing the evaluation of different situations. Second, we instantiate different model compositions to qualitatively identify interactions between DPS and RRS that actuate independently. Our results suggest that these systems do not provide good performance when they coexist without coordination.

The remaining of this paper is organized as follows. In the next section we present a model that can be used to study interactions between DPS and RRS. In Section 3 we present an instantiation of this model. In Section 4 we analyze the results of simulations of several possible compositions of the instantiated model. Then, in Section 5 we present some related work. Finally, in Section 6, we conclude the paper and point future directions.

2 A model to study interactions between dynamic provisioning and rejuvenation systems

In order to qualitatively identify uncoordinated interactions between a DPS and an RRS, we developed a model that encompasses an application, a software error injection system (SEIS), a DPS, an RRS and a load generator system (LGS), as illustrated in Figure 1. We see this model as a component-based model, in which the application and the LGS are mandatory components.

The application is a service that runs on a cluster with load balancing such as a Web based application. Both management systems (DPS and RRS) collect monitoring information which serves as basis for management decisions. While the DPS flexes the application capacity according to its demand, the RRS detects and restarts faulty components of the application. While a node is being rejuvenated, it is not able to service requests. The SEIS changes the application behavior in order to model the effects of software faults. Finally, the LGS sends requests to the application according to some trace. Each request comes with the time required to service it (inherent service time).

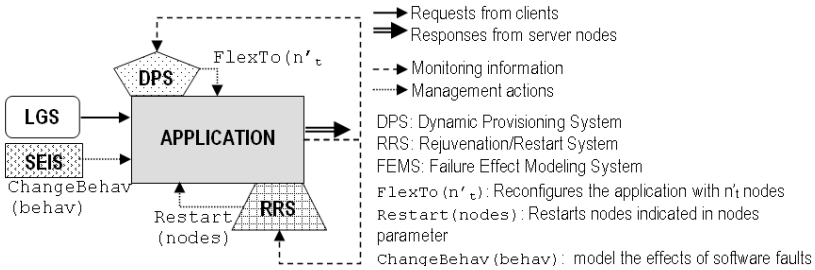

Fig. 1. Complete model view

2.1 The application model

We consider scalable applications, i.e applications whose capacity can be easily changed by changing the number of active nodes that run them. In fact, many applications satisfy this requirement; for instance, Web based applications such as e-commerce and auction sites. We consider an application with a load balancer LB and one tier with n_t active nodes at time t, as depicted in Figure 2(a). A node is active if it is expected to process requests. The LB receives requests from clients which are redirected to one of the n_t nodes following a round robin order. Requests arrival rate ($\lambda(t)$) can be highly variable.

Servers such as Apache and Tomcat run multiple processes or threads that process the requests. Each process/thread computes one request at a time. The maximum amount of processes/threads dictates the level of concurrency. The system administrator is allowed to change this number to tune the server to the amount of underneath resources. By following this idea, in each node i and every time t, there is an admission controller, composed by a pool of $m_{t,i}$ tokens. To be served, a request must first acquire a token. Requests that arrive when all tokens are in use wait in a queue called Backlog until tokens are released due to request completions. The Backlog queue has capacity K and follows a first-come, first-served (FCFS) discipline. Requests that arrive when the Backlog is full are dropped. This admission control system is illustrated in Figure 2(b).

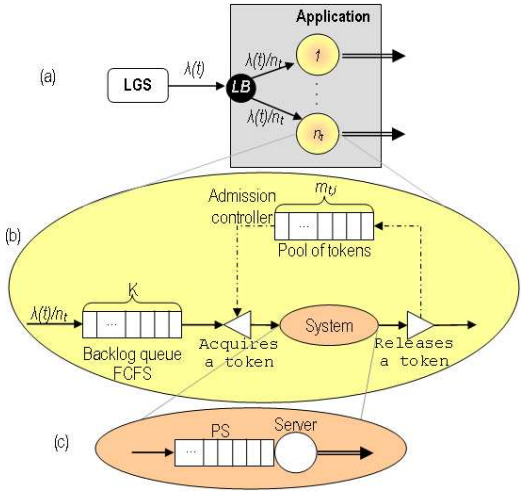

Fig. 2. Application model view

When a request gets a token it starts to be processed. In practice, this request would be served by a network of queues composed by hardware and software resources. We do not intend to find out what the bottlenecks of a system are, but to model the influence that the number of requests has in the system response time. Thus, we simplify the model by considering that all the admitted requests enter into a processor sharing (PS) system called Server and get equal share of system resources. If one request alone in the system is processed in t time units, when there are n requests each of them is processed in $t \times n$ time units. This system is illustrated in Figure 2(c).

2.2 The software error injection system

The SEIS changes the application behavior to model the effects of software faults. We use Laprie's dependability terminology defined in [9]. He considers faults as defects that exist in the system. Faults may stay in a dormant state, or they can be activated, leading to errors. Error conditions may lead the system to a failure when the expected behavior of the system is violated.

The SEIS introduces software errors into the managed application. Such errors can lead to performance degradation or to crash/hang failures. Software errors are modeled by changing the inherent service demands that come with requests based on a particular degradation function. Errors that lead to performance degradation failures occur when the application is available but does not accomplish its expected performance. They can be modeled by incrementing inherent service demands, as we exemplify in Section 3. When inherent service demands tend to infinite we consider the occurrence of a crash/hang failure.

2.3 The rejuvenation system

An RRS monitors each component of the application, detects/forecasts failures and restart components in order to bring them to a clean state. The granularity of these components may change from system to system. Candea *et al* [6] considers JavaBeans components, while [8,7] consider a whole process. The output of an RRS determines which nodes or components must be restarted.

2.4 The dynamic provisioning system

A DPS changes application capacity to accommodate the current load. It encompasses a feedback control loop. Monitoring information about the application and/or its environment is gathered and used to decide on the best number of machines to give to the application. DPSs differ from each other due to the monitoring information gathered and the objective function pursued. The output of a DPS indicates the number of nodes that must be active. Actions are needed when this number differs from the current number of active nodes. When this new value is greater than the current one, nodes must migrate from a pool of free machines to the application. Otherwise, nodes must be released. In this case, the LB immediately stops sending requests to that node, but the node remains active until all requests already in place are processed [1].

2.5 Metrics of interest

Management systems (including the automated ones) typically aim at delivering the expected QoS at the lowest cost. Thus, an automated management system can be measured by two classes of metrics: (i) QoS metrics and (ii) cost metrics.

Two metrics related to the application QoS are going to be computed: average response time (R) and average availability (A). For response times we consider only the amount of time a request stayed in the server side. Moreover, we only consider the response times of successful responses. Availability is computed as the percentage of requests successfully processed during a measurement interval.

When a node of the data center is allocated to the application, it is in the active state; otherwise, it is in the inactive state. In order to infer the operational cost of an application we use the average number of active nodes that run the application. If we know the the amount of time the application ran and the cost of an active node per time unit we can compute the application operational cost.

3 Instantiating the component-based model

In this section we present an instantiation of the generic model presented in the last section. In Subsection 3.1 we define the behavior of some components of the model and in Subsection 3.2 we present the simulation parameters used to instantiate the model compositions.

3.1 Components behavior

The DPS aims at maintaining nodes utilization around a target, as proposed in [1]. It periodically queries the application nodes for monitoring information. After each measurement interval the DPS queries nodes for the following measurements, computed for the previous interval: X, the number of request completions; A, the number of request arrivals; and U, the average CPU utilization (over all nodes). Given N, the current number of nodes, and ρ_{target}, the target utilization, the DPS computes the required number of servers for the next interval as follows: (i) it computes the average demand per completion as $D = U/X$; (ii) it computes the normalized utilization as $U' = max(A, X) \times D$; and (iii) it computes the number of servers needed to achieve ρ_{target} as $\lceil N' = N \times U'/\rho_{target} \rceil$.

The set of failures chosen to be modeled must be representative of the real world, since we want to uncover plausible interactions. We only consider performance degradation failures seen by other researchers. We use results of experiments carried out by Li *et al* [5] with Apache Web server. By generating a constant connection rate to the Web server they observed that response times become longer over time, degrading around 0.03 ms per hour. Based on these results we translated Li's equation into one that relates service time to the amount of requests already serviced by a node. Let $\delta_i(t)$ be the number of requests served by the node i since its last restart (or since it was activated). Each request r that arrives into a node has its service demand changed according to the following function: $S_{r,i}(t) = S_{0,r} + S_{aging}(t)$; where $S_{r,i}$ is the new service time; $S_{aging}(t) = 2.4 \times 10^{-8} \delta_i(t)$ and represents the addition due to aging in service time; and $S_{0,r}$ is the inherent service time of the request.

The RRS works as follows. After each measurement interval, the RRS gathers the current S_{aging} of each node and their availabilities. It restarts a node when: (i) the node's S_{aging} value reaches a threshold; or (ii) the node's availability is smaller than the minimum expected availability (A_{min}) for y consecutive measurement intervals. When a rejuvenation action is triggered, the RRS serves all requests already in place before restarting the node process [10].

3.2 Simulation parameters

Both DPS and RRS are configured with 5 minutes measurement intervals. The DPS target utilization may assume three values in different simulation experiments: 65%, 75% and 85%. The RRS's S_{aging} threshold is configured as 1 second. The minimum availability (A_{min}) and y are set to 99.99% and 6 respectively.

The application parameters are set as follows. The Server system timeslice is 110 ms. The Backlog queue capacity is 1024. Both are in accordance with Linux default. Migration time and effective restart time obey a normal distribution with averages of 60 and 120 seconds, respectively. This is in accordance with some experiments we conducted using JBoss [8]. The capacities of the nodes can assume different values in different simulation experiments: around 100, 300 and 500 rps (requests per second). Nodes of low, medium and high capacity respectively have a total number of tokens ($m_{t,i}$) of 200, 500 and 1000. These

numbers were chosen to avoid a request to stay more than two seconds in the Server queue.

Our simulation experiments use a 17-hour workload generated by GEIST [11]. This workload is variable and presents an average request rate of 670 rps.

The component based model allows us to build different compositions to reproduce different situations. The simplest composition is called AWoF (application without failure). The second one is called AWF (Application with failures) and is composed by the application and the SEIS. The third model is the AWF+RRS, which encompasses the application, the SEIS and the RRS. These three models use a static number of nodes calculated from results of simulations of the AWoF model. The number allows an application without failures to handle the workload without violating the 99.99% of minimum availability and the 2 seconds of maximum response times. The fourth and fifth model compositions are called AWoF+DPS and AWF+DPS. Finally, the most complete model is the AWF+DPS+RRS, which encompasses the application, the SEIS, the DPS and the RRS. In these models, DPS defines the amount of nodes to be operational on the fly. All these compositions are necessary in order to allow us to compare results of the most important model (the AWF+DPS+RRS) with results of the other intermediary models.

4 What happens when an application is controlled by a dynamic provisioning and a rejuvenation system?

We ran all simulation scenarios 5 times, which resulted in 180 simulation experiments. Average application availability and response times measured for all compositions are presented in Figures 3 and 4. Average utilizations and number of nodes used are presented in Tables 1 and 2 respectively.

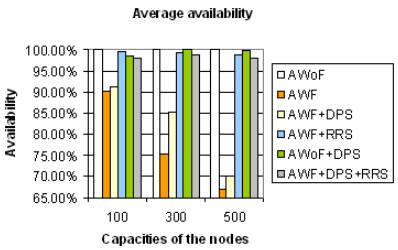

Fig. 3. Average availability

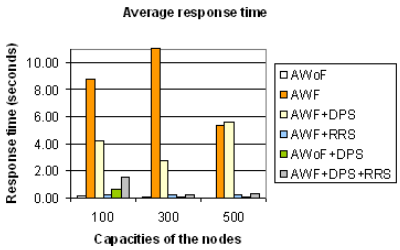

Fig. 4. Average response times

Our results show that the DPS is an efficient system. Application QoS measured for AWoF+DPS is near to the one delivered by AWoF, the model that provides the best result in terms of application QoS. However, AWoF+DPS, on average, uses 11.8% less nodes than AWoF which is statically overprovisioned.

	100 rps	*300 rps*	*500 rps*
AWF+DPS+RRS	9.25	3.64	2.39
AWoF+DPS	9.19	3.55	2.25
AWoF, AWF, AWF+RRS	10	4	3
AWF+DPS	9.46	3.76	2.19

Table 1. Average number of active nodes

	100 rps	*300 rps*	*500 rps*
AWF+DPS+RRS	70.1%	63.6%	59.9%
AWoF+DPS	70.2%	63.2%	59.6%
AWoF	66.2%	55.6%	44.8%
AWF+RRS	66.9%	57.8%	49.5%

Table 2. Average utilization

Thus, the DPS not only delivers a good application QoS but also reduces operational costs.

The RRS also proved to be efficient. When the application with failure ran without an RRS its QoS degraded, as the results of the AWF and AWF+DPS models show. These two compositions resulted the two worst QoS for the application. It is clear that the DPS is not able to manage applications with software faults. When the RRS runs (AWF+RRS composition) application QoS increased and was near that of AWoF.

Although DPS and RRS perform very well in isolation, application QoS degrades when they coexist. For nodes of low, medium and high capacities, application availability of the AWF+DPS+RRS model is respectively 0.59%, 0.30% and 0.73% worse than the minimum availability between the ones measured for AWoF+DPS and AWF+RRS. At a first glance, availability does not seem to vary substantially. However, small variations in availability represent big differences in terms of number of requests served. With a request arrival of 670 rps, an availability loss of 0.30% leads the application to drop almost 180,000 more requests per day. For nodes of low, medium and high capacities, response times of the AWF+DPS+RRS model are 60.4%, 30,8% and 17.9% higher than the the maximum response times between the ones measured for AWoF+DPS and AWF+RRS.

By comparing the number of nodes used by AWoF+DPS and AWF+DPS-+RRS we find out if the DPS changes its decisions due to the RRS actuation or aging. On average, the number of nodes used by AWF+DPS+RRS is greater than the number of nodes used by AWoF+DPS (Table 1): 0.6%, 2.5% and 5.9% greater for nodes of low, medium and high capacities. The AWF+DPS+RRS composition uses more nodes than the AWoF+DPS one because of the way failures are modeled. The SEIS models performance degradation faults. Requests in a faulty node require more time to be served. This failure presents two consequences. First, response times are greater, since service times are greater. Second, faulty nodes present greater utilizations, because requests stay longer in the system. Since the DPS goal is to maintain nodes' utilizations around a target, it adds more nodes when there are faulty nodes. Besides, when a node is selected for rejuvenation the DPS is influenced even more and augments the amount of nodes used. Nodes are added during or immediately after rejuvenation, probably to suppress the lack caused by the node being restarted, and are removed

shortly after being added. During a restart, the difference in terms of number of nodes used by AWoF+DPS+RRS and AWoF+DPS increases to 1.6%, 12.5% and 27.7% for nodes of low, medium and high capacities respectively (Table 3).

Let us now discuss what happens with the application QoS during the restarts. We present average application availability and response time during restarts in Figures 5 and 6. The average number of nodes used and their utilizations during restarts are presented in Tables 3 and 4 respectively.

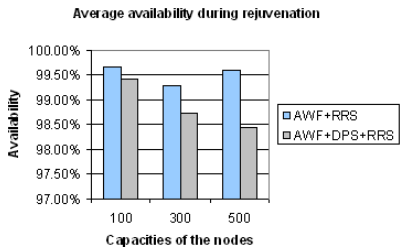

Fig. 5. Average availability (restart)

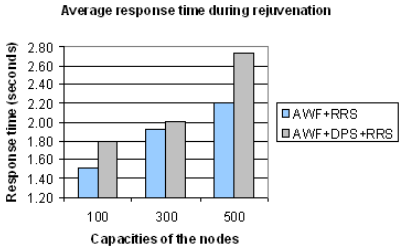

Fig. 6. Average response times (restart)

	100 rps	*300 rps*	*500 rps*
AWF+DPS+RRS	8.94	3.91	2.92
AWoF+DPS	8.80	3.42	2.11
AWF+RRS	10	4	3

Table 3. Average number of active nodes during rejuvenation

	100 rps	*300 rps*	*500 rps*
AWF+DPS+RRS	74.3%	77.2%	68.3
AWoF+DPS	71.8%	65.3%	60.2%
AWF+RRS	72.0%	72.7%	62.8%

Table 4. Average utilization during rejuvenation

Overprovisioned applications cope better with rejuvenation than the applications managed by a DPS. We found out that, during restarts, for low, medium and high capacity nodes respectively, average application availability measured for the AWF+RRS is 0.25%, 0.56% and 1.16% better than the availability measured for the AWF+DPS+RRS composition. The application response time during rejuvenation of nodes of low, medium and high capacities is 16.1%, 4.0% and 19.3% better for the AWF+RRS model than for the AWF+DPS+RRS one.

When both systems coexist the average utilization of the nodes during restart is greater than the utilization of the nodes from the AWF+RRS (Table 4). The overprovisioned AWF+RRS composition always uses more nodes than the AWF+DPS+RRS one. The number of nodes used by the DPS during the restart of a node for the AWF+DPS+RRS model is not enough to suppress the lack of the node being restarted. As a result, the active nodes of the AWF+DPS+RRS model become more saturated during restarts, increasing the probability of request rejection (when the Backlog is full) and increasing response times.

The rejuvenation time for the AWF+DPS+RRS composition is 0.33, 0.40 and 0.56 hours for nodes of low, medium and high capacity respectively. For the AWF+RRS model, the average restart time is almost equal those ones: 0.34, 0.41 and 0.55 hours for nodes of low, medium and high capacity respectively. Thus, the AWF+DPS+RRS composition spends, on average, the same time rejuvenating nodes, however, during these moments, application QoS degrades more when both DPS and RRS coexist then for the AWF+RRS composition.

To sum up, our results suggest that when both systems coexist application QoS may degrade in comparison with the QoS provided when each system is acting alone. This is an indicative that they are not orthogonal systems, in the sense that they are not independent. We believe that some level of coordination must be added to maximize the benefits gained from the simultaneous use of both systems. In fact, our previous experience [8] shows that some coordination between DPS and RRS can provide good results.

5 Related Work

The dissemination of overlay networks over IP (Internet Protocol) networks results in two independent systems coordinating data routing. In [12] interactions between these systems are studied. When failures occur, these interactions interfere in some traffic engineering tasks and, when there are overlay networks that span different autonomous systems, they allow the network status of a system to influence the network status of others, which is undesirable. We here also investigate interactions between systems that act over the same target.

Systems with conflicting goals is presented in [13]. Some conflicting relationships arise when a complex application presents both real time and fault tolerance requirements. In fact, some middlewares offer real time guarantees and others offer fault tolerance behavior. However, when real time requirements coexist with fault tolerance ones, the simple union of systems exclusively designed to deal with individual cases is not enough [13,14]. DPS and RRS systems have sometimes conflicting goals. For instance, RRS can restart a saturated node that was delivering low availability during a load surge.

Graupner *et al* [15] alert to interactions among traditional management systems and virtualization management systems, which recently arose as separated management systems. Associations among applications and underneath resources change more often in virtualized environments, under the control of the virtualization layer. If no information is exchanged between the traditional management system and the virtualization system, the traditional management system becomes unaware of these dynamic associations. It is difficult to separate the virtualization and the traditional management and these activities should be made in a combined way. This vision of various management systems which interact among them is exactly the vision we consider here. Interactions between these systems must be known in order to allow them to coexist synergistically.

Finally, control theory researchers study how independent control systems acting under the same plant (controlled system) and in the presence of uncer-

tainties actuate such as the desired global state of the system is reached. This problem is known as the decentralized adaptive control problem [16]. According to [16,17,18] distributed control systems need to exchange information in order to reduce the error of control actions. In [17], for instance, control systems actuate in different and pre-scheduled moments and when one system makes a decision about a control action the others must be aware of that. In [18] the control system which actuate over a subsystem must know the desired status of all subsystems that compounds the plant. It is clear that an extra effort is needed to coordinate the actions of control systems which actuate over the same plant. In the future, similar techniques may be introduced into DPS and RRS behavior in order to make their coexistence more efficient. This work provides some insights on how these systems coexist without these techniques.

6 Conclusions and future research

In this paper we propose a component based model used to better understand the interactions between dynamic provisioning and restart systems that act without coordination over the same application. We implemented the model and instantiated many different compositions. Then, we evaluated the model through simulation experiments.

Even in the very simple scenario studied, where only one node can be rejuvenated at a time, we can see that DPS and RRS may interact in a way that application QoS is degraded when both systems run simultaneously if compared to performance of systems that use each of the management services independently. Even worse, performance degrades when a slightly great number of resources are used. We believe that for more complex applications, with many tiers, and higher failure rates these unwanted interactions will be even more present.

According to our results, it is inefficient to join a DPS and an RRS that are not aware of each other. We believe they can exchange some information to allow the coordination of their actions in a synergistic manner. The way the application fail and the moments when one or more nodes are restarted influence not only the application QoS but also DPS actions. If DPS is aware of failure information and RRS actions it could act in a more efficient manner. One of these positive manners was presented in [8], where aged nodes were released when a capacity decreasing was performed.

Our next step is to propose an coordination module that makes the coexistence of these systems completely harmonic. This additional module is interesting because it allows us to harness mature dynamic provisioning and rejuvenation systems. We plan to analyze and validate this coordination module by carrying out simulations and measurement experiments.

Acknowledgments. We would like to thank Dr. Alexander Keller and the anonymous reviewers for their helpful comments. This work was developed in collaboration with HP Brazil R&D and funded by CNPq/Brazil grants 141655/2002-0, 302317/2003-1 and 300646/1996-8.

References

1. 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) 3 – 12
2. 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
3. Urgaonkar, B., Shenoy, P.: Cataclysm: Handling extreme overloads in internet applications. In: Proceedings of the Fourteenth International World Wide Web Conference (WWW 2005). (2005)
4. Garg, S., et al: Analysis of preventive maintenance in transactions based software systems. IEEE Transactions on Computers **47** (1998)
5. 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)
6. Candea, G., et al: Microreboot – a technique for cheap recovery. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation. (2004)
7. 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)
8. Lopes, R., Cirne, W., Brasileiro, F.: Improving dynamic provisioning systems using software restarts. In: Fifteenth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management. LNCS. Volume 3278., Springer (2004)
9. Anderson, T., ed.: Edpendability of Resilient Computers. Blackwell Scientific Publications, Oxford (1989)
10. 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
11. 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
12. Keralapura, R., Taft, N., Iannaccone, C.N.C.G.: Can isps take the heat from overlay networks? In: ACM SIGCOMM Workshop on Hot Topics in Networks. (2004)
13. Narasimhan, P.: Trade-offs between real-time and fault tolerance for middleware applications. In: Workshop on Foundations of Middleware Technologies. (2002)
14. Stankovic, J.A., F.Wang: The integration of scheduling and fault tolerance in real-time systems. Technical report, UM-CS-1992-049, Department of Computer Science, University of Massachusetts (1992)
15. Graupner, S., et al: Impact of virtualization on management systems. Technical report, Hewlett-Packard Laboratories (2003)
16. Mukhopadhyay, S.: Distributed control and distributed computing. SIGAPP Appl. Comput. Rev. **7** (1999) 23–24
17. Mukhopadhyay, S., Narendra, K.S.: Decentralized adaptive control using partial information. In: American Control Conference. Volume 1. (1999) 34 – 38
18. Narendra, K.S., Oleng, N.O.: Decentralized adaptive control. In: American Control Conference. Volume 5. (2002) 3407 – 3412