

Achieving and Assuring High Availability

Kishor Trivedi¹, Gianfranco Ciardo², Balakrishnan Dasarathy³, Michael Grottke⁴,
Rivalino Matias¹, Andy Rindos⁵, and Bart Varshaw⁵

¹ Duke University

² UC Riverside

³ Telcordia

⁴ University of Erlangen-Nuremberg

⁵ IBM

kst@ee.duke.edu, ciardo@cs.ucr.edu, das@research.telcordia.com,
rivalino@ece.duke.edu, michael.grottke@wiso.uni-erlangen.de,
{rindos,vashaw}@us.ibm.com

Abstract. We discuss availability aspects of large software-based systems. We classify faults into Bohrbugs, Mandelbugs and aging-related bugs, and then examine mitigation methods for the last two bug types. We also consider quantitative approaches to availability assurance.

Keywords: High-Availability, Proactive Fault-Tolerance, Software Aging.

1 Overview

Bugs invariably remain when an application is deployed. A good, albeit expensive, development process can reduce the number of residual bugs to the order of 0.1 defects per 1000 lines of code [1]. There are broadly two classes of residual bugs in an application, known as Bohrbugs and Mandelbugs [2]. Bohrbugs are easily isolated and manifest themselves consistently under well-defined sets of conditions; thus, they can be detected and fixed during the software-testing phase, although some of them do remain in production. Preliminary results from investigation of a NASA software project suggest that 52% of residual bugs were Bohrbugs [3]. Mandelbugs instead have complex causes, making their behavior appear chaotic or even non-deterministic (e.g., race conditions), thus are often difficult to catch and correct in the testing phase. Retrying the same operation might not result in a failure manifestation. A third type of bugs has the characteristic that its failure manifestation rate increases with the time of execution. Such faults have been observed in many software systems and have been called aging-related bugs [4], [5], [6]. Memory leaks and round-off errors are examples of aging-related bugs. There are effective approaches to dealing with residual Bohrbugs after a software product has been released. If a failure due to a Bohrbug is detected in production, it can be reproduced in the original testing environment, and a patch correcting the bug or a workaround can be issued. Mandelbugs, however, often cannot be easily fixed, thus techniques to recover from Mandelbugs at run-time are needed. Broadly applicable cost- and time-effective

run-time techniques also exist to address aging-related bugs. We focus on Mandelbugs and aging-related bugs in this tutorial.

In general, there are two ways to improve availability: increase time-to-failure (TTF) and reduce time-to-recovery (TTR). To increase TTF, proactive failure avoidance techniques known as rejuvenation can be used for aging-related bugs. To reduce TTR, we propose instead escalated levels of recovery, so that most failures are fixed by the quickest recovery method and only few by the slowest ones. We are also interested in quantifiable availability assurance.

2 Quantified Availability Assurance

In practice, availability assurance is provided qualitatively by means of verbal arguments or using checklists. Quantitative assurance of availability by means of stochastic availability models constructed based on the structure of the system hardware and software is very much lacking in today's practice [7], [8], [9]. While such analyses are supported by software packages [10], they are not routinely carried out on what are touted as high availability products; there are only islands of such competency even in large companies.

Engineers commonly use reliability block diagrams or fault trees to formulate and solve availability models because of their simplicity and efficiency [10], [11]. But such combinatorial models cannot easily incorporate realistic system behavior such as imperfect coverage, multiple failure modes, or hot swap [7], [9]. In contrast, such dependencies and multiple failure modes can be easily captured by state-space models [11] such as Markov chains, semi-Markov processes, and Markov regenerative processes. However, the construction, storage, and solution of these models can become prohibitive for real systems. The problem of large model construction can be alleviated by using some variation of stochastic Petri nets, but a more practical alternative is to use a hierarchical approach using a judicious combination of state space models and combinatorial models [10]. Such hierarchical models have been successfully used on practical problems including hardware availability prediction [12], OS failures [7], [8] and application software failures [9]. Furthermore, user and service-oriented measures can be computed in addition to system availability. Computational methods for such user-perceived measures are just beginning to be explored [9], [13], [14]. Subsequently, parameter values are needed to solve the models and predict system availability and related measures. Model input parameters can be divided into failure rates of hardware or software components; detection, failover, restart, reboot and repair delays and coverages; and parameters defining the user behavior. Hardware failure rates (actually MTTFs) are generally available from vendors, but software component failure rates are much harder to obtain. One approach is to carry out controlled experiments and estimate software component failure rates. In fact, we are currently performing such experiments for the IBM WAS SIP implementation. Fault-injection experiments can be used to estimate detection, restart, reboot, and repair delays, as in the IBM SIP reliability model [9]. Due to many simplifying assumptions made about the system, its components, and their interactions and due to unavailability of accurate parameter values, the results of the models cannot be taken as a true availability assurance. Monitoring and statistically

inferring the observed availability is much more satisfactory assurance of availability. Off-line [16] and on-line [17] monitoring of deployed system availability and related metrics can be carried out. The major difficulty is the amount of time needed to get enough data to obtain statistically significant estimates of availability.

3 Recovery from Failures Caused by Mandelbugs

Reactive recovery from failures caused by Mandelbugs has been used for some time in the context of operating system failures, where reboot is the mitigation method [8]. Restart, failover to a replica, and further escalated levels of recovery such as node reboot and repair are being successfully employed for application failures. Avaya's NT-SwiFT and DOORS systems [18], JPL REE system [19], Alcatel Lucent [15], IBM x-series models [20], and IBM SIP/SLEE cluster [9] are examples where applications or middleware processes are recovered using one or more of these techniques. To support recovery from Mandelbug-caused failures, multiple run-time failure detectors are employed to ensure that detection takes place within a short duration of the failure occurrence. In all but the rarest cases, manual detection is required. As, by definition, failures caused by non-aging-related Mandelbugs cannot be anticipated and must be reacted to, current research is aimed at providing design guidelines as to how fast recovery can be accomplished and obtaining quantitative assurance on the availability of an application.

Stochastic models discussed in the previous section are beginning to be used to provide quantitative availability assurance [9], [18], [19], [20]. Besides system availability, models to compute user-perceived measures such as dropped calls in a switch due to failures are also beginning to be used [9]. Such models can capture the details of user behavior [14] or the details of the call flow [9] and their interactions with failure and recovery behavior of hardware and software resources. Difficulties we encounter in availability modeling are model size and obtaining credible input parameters. To deal with the large size of availability models for real systems, we typically employ a hierarchical approach where the top-level model is combinatorial, such as a fault tree [7], [9], [12] or a reliability block diagram [8]. Lower-level stochastic models for each subsystem in the fault tree model are then built. These submodels are usually continuous-time Markov chains, but if necessary non-Markov models can be employed. Weak interactions between submodels can be dealt with using fixed-point iteration [21]. The key advantage of such an hierarchical approach is that closed-form solution now appears feasible [7], [13] as the Markov submodels are typically small enough to be solved by Mathematica and the fault tree can be solved in closed-form using tools like our own SHARPE software package [10]. Once the closed-form solution is obtained, we can also carry out formal sensitivity analysis to determine bottlenecks and provide feedback for improvement to the designers [13]. We are currently working on interfacing SHARPE with Mathematica to facilitate such closed-form solutions. Errors in these approximate hierarchical models can be studied by comparison with discrete-event simulation or exact stochastic Petri net models solved numerically.

4 Proactive Recovery and Aging-Related Bugs

Aging-related bugs in a system are such that their probability of causing a failure increases with the length of time the system is up and running. For such bugs, besides reactive recovery, proactive recovery to clean the system internal state can effectively reduce the failure rate. This kind of preventive maintenance is known as “software rejuvenation” [2], [22]. Many types of software systems, such as telecommunication software [22], web servers [5], [23], and military systems [6], are known to experience aging. Rejuvenation has been implemented in several kinds of software systems, including telecommunication billing data collection systems [22], spacecraft flight systems [24], and cluster servers [25].

The main advantage of planned preemptive procedures such as rejuvenation is that the consequences of sudden failures (like loss of data and unavailability of the entire system) are postponed or prevented; moreover, administrative measures can be scheduled to take place when the workload is low. However, for each such preemptive action, costs are incurred in the form of scheduled downtime for at least some part of the system. Rejuvenation can be carried out at different granularities: restart a software module, restart an entire application, restart a specific virtual machine in a VMM, perform garbage collection in a node, or reboot a hardware node [23], [26], [27]. A key design question is finding the optimal rejuvenation schedule and granularity. Rejuvenation scheduling can be time-based or condition-based. In the former, rejuvenation is done at fixed time intervals, while, in the latter, the condition of system resources is monitored and prediction algorithms are used to determine an adaptive rejuvenation schedule [4]. A rejuvenation trigger interval, as computed in time-based rejuvenation, can adapt to changing system conditions, but its adaptation rate is slow as it only responds to failures that are expected to be rare. Condition-based rejuvenation instead does not need time to failure inputs; it computes rejuvenation trigger interval by monitoring system resources and predicting the time to exhaustion of resources for the adaptive scheduling of software rejuvenation [4], [25], [28]. Whatever schedule and granularity of rejuvenation is used, the important question is what improvement this implies on system availability, if any. Published results are based on either analytic models [20] or simulations [29]. Early experimental results [26] are very encouraging, where rejuvenation increased the MTTF by a factor of two.

References

1. Holzmann, G.J.: Conquering complexity. IEEE Computer, Los Alamitos (2007)
2. Grottke, M., Trivedi, K.S.: Fighting bugs: Remove, retry, replicate and rejuvenate. IEEE Comp. 40, 107–109 (2007)
3. Grottke, M., Nikora, A., Trivedi, K.S.: Preliminary results from the NASA/JPL investigation - Classifying Software Faults to Improve Fault Detection Effectiveness (2007)
4. Garg, S., van Moorsel, A., Vaidyanathan, K., Trivedi, K.S.: A methodology for detection and estimation of software aging. In: 9th Int’l Symp. on Software Reliability Engineering, pp. 283–292 (1998)

5. Grottke, M., Li, L., Vaidyanathan, K., Trivedi, K.S.: Analysis of software aging in a web server. *IEEE Transactions on Reliability* 55, 411–420 (2006)
6. Marshall, E.: Fatal error: how Patriot overlooked a Scud. *Science* 255, 1347 (1992)
7. Smith, W.E., Trivedi, K.S., Tomek, L., Ackeret, J.: Availability analysis of multi-component blade server systems. *IBM Systems Journal* (to appear, 2008)
8. Trivedi, K.S., Vasireddy, R., Trindade, D., Nathan, S., Castro, R.: Modeling high availability systems. In: *Pacific Rim Dependability Conference* (2006)
9. Trivedi, K.S., Wang, D., Hunt, J., Rindos, A., Peyravian, M., Pulito, B.: IBM SIP/SLEE cluster reliability model. In: *Globecom 2007, D&D Forum, Washington* (2007)
10. Sahner, R.A., Trivedi, K.S., Puliafito, A.: *Performance and Reliability Analysis of Computer Systems*. Kluwer Academic Press, Dordrecht (1996)
11. Trivedi, K.S.: *Probability & Statistics with Reliability, Queueing and Computer Science Applications*, 2nd edn. John Wiley, New York (2001)
12. Lanus, M., Yin, L., Trivedi, K.S.: Hierarchical composition and aggregation of state-based availability and performability models. *IEEE Transactions on Reliability*, 44–52 (2003)
13. Sato, N., Nakamura, H., Trivedi, K.S.: Detecting performance and reliability bottlenecks of composite web services. In: *ICSOC* (2007)
14. Wang, D., Trivedi, K.S.: Modeling user-perceived service availability. In: Malek, M., Nett, E., Suri, N. (eds.) *ISAS 2005. LNCS*, vol. 3694, Springer, Heidelberg (2005)
15. Mendiratta, V.B., Souza, J.M., Zimmerman, G.: Using software failure data for availability evaluation. In: *GLOBECOM 2007, Washington* (2007)
16. Garzia, M.: Assessing the Reliability of Windows Servers. In: *Int'l Conf. Dependable Systems and Networks* (2003)
17. Haberkorn, M., Trivedi, K.S.: Availability monitor for a software based system. In: *HASE, Dallas* (2007)
18. Garg, S., Huang, Y., Kintala, C.M.R., Trivedi, K.S., Yajnik, S.: Performance and reliability evaluation of passive replication schemes in application level fault tolerance. In: *29th Annual Int'l Symp. on Fault Tolerant Computing, Wisconsin*, pp. 15–18 (1999)
19. Chen, D., et al.: Reliability and availability analysis for the JPL remote exploration and experimentation system. In: *Int'l Conf. Dependable Systems and Networks, Washington* (2002)
20. Vaidyanathan, K., Harper, R.E., Hunter, S.W., Trivedi, K.S.: Analysis and implementation of software rejuvenation in cluster systems. In: *ACM SIGMETRICS* (2001)
21. Mainkar, V., Trivedi, K.S.: Sufficient conditions for existence of a fixed point in stochastic reward net-based iterative methods. *IEEE Transactions on Software Engineering* 22, 640–653 (1996)
22. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.: Software rejuvenation: analysis, module and applications. In: *25th Int'l Symp. on Fault-Tolerant Computing*, pp. 381–390 (1995)
23. Matias Jr., R., Freitas, P.J.F.: An experimental study on software aging and rejuvenation in web servers. In: *30th IEEE Annual Int'l Computer Software and Applications Conference, Chicago*, pp. 189–196 (2006)
24. Tai, A., Chau, S., Alkalaj, L., Hect, H.: On-board preventive maintenance: a design-oriented analytic study for long-life applications. *J. Perf. Evaluation* 35, 215–232 (1999)
25. Castelli, V., Harper, R.E., Heidelberger, P., Hunter, S.W., Trivedi, K.S., Vaidyanathan, K., Zeggert, W.P.: Proactive management of software aging. *IBM Journal of Research and Development* 45, 311–332 (2001)
26. Kourai, K., Chiba, S.: A fast rejuvenation technique for server consolidation with virtual machines. In: *Int'l Conf. on Dependable Systems and Networks*, pp. 245–255 (2007)

27. Xie, W., Hong, Y., Trivedi, K.S.: Analysis of a two-level software rejuvenation policy. *Reliability Engineering and System Safety* 87, 13–22 (2005)
28. Vaidyanathan, K., Trivedi, K.S.: A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing* 2, 124–137 (2005)
29. Dohi, T., Goseva-Popstojanova, K., Trivedi, K.S.: Statistical Non-Parametric Algorithms to Estimate the Optimal Software Rejuvenation Schedule. In: 2000 Pacific Rim Intl. Symp. on Dependable Computing, Los Angeles, pp. 77–84 (2000)