Software Rejuvenation

Lawrence Bernstein and Dr. Chandra M. R. Kintala
Stevens Institute of Technology

Here is a design approach that makes software trustworthier, called software rejuvenation. It is a periodic, pre-emptive restart of a running system at a clean internal state that prevents latent faults from becoming future failures. It was used in systems ranging from a Lucent billing unit to NASA's long-duration space mission to Pluto, and is implemented in IBM's Netfinity resource manager. It is easy to apply, uses a little central processing unit, increases software reliability by two orders of magnitude, and is recommended for all software-intensive systems.

Software modules comprise a large part of life- and mission-critical systems. Software crashes are more likely to be the result of a fault in the software than in the hardware. In spite of our best efforts at removing the errors/faults (*bugs*[1]) before deploying those systems, it is wise to assume that bugs remain in the system and those bugs often lead to failures (*crashes*).

Software fault tolerance is aimed at tolerating those residual faults by building mechanisms to watch for failures and recover from them [1, 2]. Fault tolerance is a reactive approach: Failures usually happen at unexpected times, and the built-in mechanisms to recover from those failures will kick-in to restart the system and the service. However, these unscheduled interruptions in service are expensive and can be life-threatening. This article describes a proactive, preventive technique called *software rejuvenation* that prevents faults from becoming failures.

Lawrence Bernstein observed in 1990 that faults/bugs, when triggered in software, do not always cause failures/crashes immediately but take the system into a state where it begins to *decay*[2]. This decay has symptoms of memory leakage, broken pointers, unreleased file locks, numerical error accumulation, etc., causing gradual degradation in availability of service and data quality and eventually leading to a failure/crash.

Based on this observation, a new method to enhance the dependability of a software system, called *software rejuvenation*, was introduced in 1995 by Kintala and his colleagues in Bell Labs[3] [1, 3]. Software rejuvenation is a proactive approach that involves stopping an executing process periodically or when a failure is imminent, cleaning up the internal state of the system, and then restarting it at a known healthy state to prevent a predicted future failure.

Software rejuvenation is as intuitive as occasionally rebooting your PC, except that it was never defined, implemented, modeled, and analyzed for software systems before 1995[3] [3]. Shari Pfleeger used the term *software rejuvenation* to mean, "…looking back at software work products to try to derive additional information …" in her seminal software engineering book [4]. Her use differs from ours as we focus on the execution of the software during its mission, and she focuses on the software development process.

Use

Since the 1960s, data communication designers knew to have software modules restart a communication line when it hung. Communication line handlers often include retry logic to restart a line if it hangs. IBM implemented these techniques in its data communication systems. Their system network architecture software was especially robust to communication line hangs and restarted lines several times once a hang was detected.

"A billing data collector system, originally built by AT&T and used in most of the U.S. regional telephone companies, was the first system that used software rejuvenation for the entire system and whose use was modeled and analyzed."

An early implementation of this technique was part of the Safeguard Antimissile Missile System software implemented in the 1960s. Software designers noted that hangs could occur once error reporting buffers were full. Rather than clearing the buffers, a simple fix was implemented to restart the lines for the remote launch sites periodically when the system was in a peacetime surveillance mode. This avoided extraneous error reporting and improved the availability of the system. Separate maintenance software monitored the quality of the communication lines.

Software rejuvenation technology became the modern realization of this early design that restarts a line before the hang to avoid potential secondary problems. It is a low-cost, easy-to-implement technology that makes systems trustworthier in telecommunication systems.

A billing data collector system, originally built by AT&T and used in most of the U.S. regional telephone companies, was the first system that used software rejuvenation for the entire system and whose use was modeled and analyzed[3] [3]. Since then it has been used in many telecommunication applications, transaction processing systems, and Web servers [5]. Billing system failures and the use of software rejuvenation to prevent those failures, as described in[3] [3], are quite similar to the failures and the fix that Nick van der Zweep described recently in *Computer World*[4].

Software rejuvenation is also implemented in IBM's Director Resource Manager [6] for use in applications built on Netfinity cluster systems. Netfinity Director provides an interface to rejuvenate an application using a time interval as well as a prediction based on a number of operating system resource values.

The X2000 computing system for NASA's 15-year long Pluto-Kuiper Express mission has stringent constraints in both performance and dependability. The mission itself has three phases: initial *Cruise* phase of 12 years, *Encountering* phase

of four months, and *Exploration* phase of three years. The X2000 system has several processor strings, and all their computing power is needed during the critical Encountering phase while only a subset of the strings is required to be in service during Cruise and Exploration phases. This aspect is made use in the X2000 by rotating the individual processor strings to an on-duty and off-duty cycle and rejuvenating the software [7] to increase system reliability.

Recent experiments at Stevens Institute of Technology showed that datalink protocols suffering memory leak failures could be made reliable using rejuvenation libraries without having to fix the memory leak bug [8]. In essence, rejuvenation bounds the execution space for the working software so that latent failure modes are not executed. Had this technology been used in the Patriot Missile system (see the next section) during the first Iraq war, the counter overflow problem causing the anti-scud system to fail would not have occurred.

Patriot Missile Case History

> On Feb. 11, 1991, the Patriot Project Office received Israeli data identifying a 20 percent shift in the Patriot system's radar range gate after the system had been running for eight consecutive hours. This shift was significant because it meant that the target (in this case, the Scud) was no longer in the center of the range gate. The target needs to be in the center of the range gate to ensure the highest probability of tracking the target.

Figure 1: *Probabilistic State Transition Model for A Without Rejuvenation*

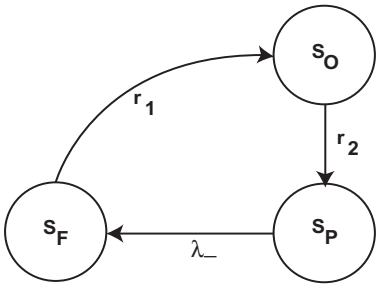
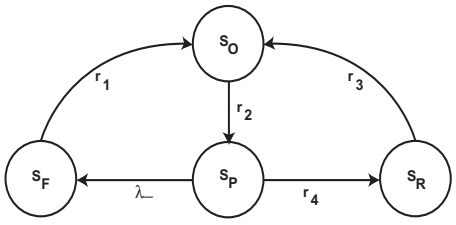


Figure 2: *Probabilistic State Transition Model for A With Rejuvenation*



The range gate algorithm determines if the Scud is in the Patriot's firing range. If it is, the Patriot fires its missiles.

Patriot Project Office officials said that the Patriot system would not track a Scud when there is a range gate shift of 50 percent or more. Because the shift is directly proportional to time, extrapolating the Israeli data (which indicated a 20 percent shift after eight hours) determined that the range gate would shift 50 percent after about 20 hours of continuous use. Specifically, after about 20 hours, the inaccurate time calculation becomes sufficiently large to cause the radar to look in the wrong place for the target. Consequently, the system fails to track and intercept the Scud.

The range gate's prediction of where the Scud will next appear is a function of the Scud's known velocity and the time of the last radar detection. Velocity is a real number that can be expressed as a whole number and a decimal (e.g., 3750.2563 miles per hour). Time is kept continuously by the system's internal clock in tenths of seconds but is expressed as an integer or whole number (e.g., 32, 33, 34, etc.). The longer the system has been running, the larger the number representing time. To predict where the Scud will next appear, both time and velocity must be expressed as real numbers. Because of the way the Patriot computer performed its calculations and the fact that its registers are only 24 bits long, the conversion of time from an integer to a real number cannot be any more precise than 24 bits. This conversion results in a loss of precision causing a less accurate time calculation. The effect of this inaccuracy on the range gate's calculation is directly proportional to the target's velocity and the length of time the system has been running. Consequently, performing the conversion after the Patriot has been running continuously for extended periods causes the range gate to shift away from the center of the target, making it less likely that the target will be successfully intercepted.

By automatically restoring the registers to a safe initial state every eight hours when there are no targets in track the system can avoid making the fault into a failure. The problem need not be fixed in the algorithm itself. This is precisely the effect of software rejuvenation.

This was not the first time this type of problem caused an ABM [antiballistic missile] system to fail. During the Safeguard Antimissile Test Program conducted at Meck Island in the Kwajalien Atoll, a similar problem occurred in the early 1970s. The test site was in an extended hold due to a range problem. The computers and radars scanned the sky for the target that was still on the launch pad in California. After several hours of idling, the antimissile system computer crashed. A timing register overflowed. The system was not tested in this configuration. The problem was found and fixed and well documented in the Mission Test Reports. Further study led to the innovative idea to restart the computer periodically when it was scanning the sky so that it returned to a known tested state. This design was included in the tactical system design. The design was later applied to avoiding hash table problems in a telephone data switch, and collecting billing data from telephone switches, but unfortunately not in the follow-on Patriot antimissile system. [9]

Modeling and Analysis

Software rejuvenation incurs overhead and should be done at a time when the cost due to service interruption is minimal. Hence modeling the system to find optimal rejuvenation times is crucial. A simple and useful model based on continuous-time Markov chains was first introduced in[3] [3] to analyze software rejuvenation.

Figure 1 shows the model for system A without rejuvenation and Figure 2 is the model for system A with rejuvenation. S_0 is the initial robust state of system A, S_p is the failure probable state, and S_f is the failure state. The transition time from the failed state S_f to robust state S_0 is exponentially distributed with rate r_1 (the repair rate), the transition rate from robust state S_0 to failure probable state S_p is r_2, and λ is that rate for transition from a failure probable state to a failed state. If the system performs rejuvenation, it will go from S_p to S_r at rate r_4 and will transition to robust state at rate r_3.

From this model you can compute the

expected downtime due to rejuvenation over period L to be $(\lambda/r_1+r_4/r_3)/(1+\lambda/r_1+r_4/r_3+(\lambda+r_4)/r_2)$ x L. For example, suppose system A has the following profile:

1. Its mean time between failures (MTBF) is three months; hence, its failure distribution rate λ is 1/MTBF=1/(3x30x24).
2. Its expected repair time is two hours after an unexpected failure, so its repair distribution rate r_1 is (1/2)=0.5.
3. Its expected time to go from robust state to a failure probable state is 10 days; hence, its r_2 is 1/(10x24).
4. Its expected repair time after a scheduled failure is 10 minutes, so its r_3 is $(1/_{(1/6)})=6$.

The expected downtime of A over a period of one year will then be 7.19 hours without rejuvenation (r_4=0) and 6.36 hours with a rejuvenation frequency of two weeks (r_4=1/(14x24)).

This model was extended using Stochastic Petri Nets to study rejuvenation using the cluster-based fail-over mechanisms in IBM's Netfinity systems [6]. Using this model, it has been shown, for example, that in a two-node cluster system running a database application with one node acting as a spare, the reduction in downtime due to a software rejuvenation interval of 100 hours is 0.74. In the X2000 for the Pluto-Kupier mission, analysis of reliability due to software rejuvenation showed two orders of magnitude improvement and the optimal interval was found to be 31.2 weeks in the 12-year long *Cruise* phase [7].

A number of other modeling techniques were developed to study software rejuvenation in other application scenarios, including the Markov regenerative process model for transaction-based systems, the Weibull distribution model to combine check pointing and rejuvenation, and several others [10].

The Future

Software rejuvenation is ready for industry-wide deployment. It can make software systems trustworthier. Good designers will use it and move from the state of the art to the state of the practice. It is a *good design practice* for individual systems.

Software rejuvenation is one aspect of self-healing that has gained research interest recently There are some interesting new problems for software rejuvenation in large-scale, networked, self-healing systems. We describe some of those problems here and make some suggestions:

1. For networked applications, we need to monitor and gather the availability and quality of all the required resources for the application across the network, and then synthesize that gathered data and make a prediction about possible failure of the application or a component in the application. Network application monitoring might be hard to do in such a generalized fashion. You can perhaps do it in a limited domain such as a Voice over Internet Protocol (VoIP) application in an enterprise network.
2. Self-healing systems on a network need alternate paths for communication between components to avoid an impending failure. This may be hard to do in a generalized fashion. But in much the same way as in clustered systems providing redundancy for centralized applications, you can perhaps provide alternate communication paths for some self-healing applications (for example, VoIP) using alternate service provider networks.
3. Modeling and implementation have several problems due to their large-scale nature. What is a state in a large-scale system when *state* is across several products and systems in a network? Perhaps, you need to model the system in a hierarchical, tree-structured fashion decomposing the state into smaller units as you need it for analysis. Failure symptoms are at a system/network (macro) level but rejuvenation actions are at a component (micro) level; how do you correlate the two? This topic is perhaps related to event correlation in network management. How do you do rejuvenation efficiently in very large systems? Perhaps gradual load shedding can be used. What is a safe (clean internal) state to back up to? How do you back up to that state?

"Recent experiments at Stevens Institute of Technology showed that data link protocols suffering memory leak failures could be made reliable using rejuvenation libraries without having to fix the memory leak bug."

Conclusion

Software rejuvenation is a periodic, preemptive restart of a running system at a clean, internal state to prevent future failures. It was used in systems ranging from a Lucent billing unit to NASA's long-duration space mission to Pluto, and is implemented in IBM's Netfinity resource manager. It is one aspect of self-healing systems. Interesting future research directions for software rejuvenation and self-healing are in large-scale networked systems built with commercial off-the-shelf components and open interfaces.◆

References

1. Bernstein, L. "Software Fault Tolerance Forestalls Crashes: To Err is Human, to Forgive Is Fault Tolerant" in Advances in Computers 58. Highly Dependable Software. Ed. M. Zelkowitz. Academic Press, 2003: 240-285.
2. Lyu, M., Ed. Software Fault Tolerance. New York: John Wiley, 1995.
3. Huang, Y., C. Kintala, N. Kolettis, and N.D. Fulton. Software Rejuvenation: Analysis, Module and Applications. Proc. of 25th Symposium on Fault Tolerant Computing FTCS-25, Pasadena, CA, June 1995: 381-390 <www.ece.stevens-tech.edu/~ckintala/Papers/RejuvFTCS25.pdf>.
4. Pfleeger, S.L. Software Engineering Theory and Practice. 2nd ed. Prentice Hall, 2001: 496-502.
5. Li, L., K. Vaidyanathan, and K.S. Trivedi. "An Approach for Estimation of Software Aging in a Web Server." International Symposium on Empirical Software Engineering, Nara, Japan, Oct. 2002.
6. Vaidyanathan, K., R.E. Harper, S.W. Hunter, and K.S. Trivedi. Analysis and Implementation of Software Rejuvenation in Cluster Systems. Proc. of the Joint Intl. Conference on Measurement and Modeling of Computer Systems, ACM SIGMETRICS 2001/ Performance 2001, Cambridge, MA, June 2001.
7. Tai, A.T., L. Alkalai, and S.N. Chau. "On-Board Preventive Maintenance: A Design-Oriented Analytic Study for Long-Life Applications." Performance Evaluation 35.3-4 (June 1999): 215-232.
8. Bernstein, L., Y.D. Yao, and K. Yao. "Software Rejuvenation: Avoiding Failures Even When There Are Faults." The DoD SoftwareTECH

News 6.2 (Oct. 2003): 8-11 <www.softwaretechnews.com>.
9. General Accounting Office. "B-24709, Report to the House of Representatives." Washington, D.C.: GAO, Information Management and Technology Division, 4 Feb. 1992 <www.fas.org/spp/starwars/gao/im92026.html>.
10. Bao, Y., X. Sun, and K. Trivedi. Adaptive Software Rejuvenation: Degradation Models and Rejuvenation Schemes. Proc. of The International Conference on Dependable Systems and Networks, San Francisco, CA, June 2003.

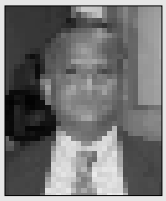
Notes

1. We use the terms *errors*, *faults*, and *bugs* interchangeably for software systems in this article, even though there are some subtle differences in academic literature.
2. Software decay, sometimes called aging, is not same as software obsolescence due to changing requirements from the system.
3. The Web site <www.software-rejuvenation.com>, maintained by professor Trivedi at Duke University, has a collection of follow-up research papers on the topic.
4. Go to <www.computerworld.com> and enter 43636 in QuickLink box, or click on <www.computerworld.com/softwaretopics/software/story/0,10801,88872,00.html>.

About the Authors

Lawrence Bernstein is a professor of Software Engineering at Stevens Institute of Technology. He is a member of the board of the Center for National Software Studies and director of the New Jersey Center for Software Engineering. Bernstein is an expert witness in arbitration cases where he assesses the quality and origins of a large software system. He spent 35 years at Bell Laboratories as chief technical officer managing large software projects. Bernstein holds eight software patents, has given 24 talks, published one book, and has written 58 articles on software engineering. He conceived of the notion of software rejuvenation, encouraged work on studying the dynamic behavior of software, applied and extended software management techniques, and led the work on adopting intermediate level languages in support of military software development. Recently, he has pioneered the use of live-through case histories with David Klappholz in the successful education of undergraduates to the principles of software engineering.

Stevens Institute of Technology
Computer Science Department
Hoboken, N.J. 07030



Chandra M.R. Kintala, Ph.D., is a distinguished service professor at Stevens Institute of Technology. Prior to that, he served as vice president of the Network Software Research and Realization Center in Avaya Labs, a spin off from Bell Labs. Previously, he was director of Distributed Software Research in Bell Labs. Kintala has done pioneering research on _____ (SwiFT) for software-implemented fault tolerance and software rejuvenation. He received a *ComputerWorld*-sponsored Smithsonian medal for SwiFT in Lucent in 1998. Under his management, his groups created ExpertNet for enterprise network Voice over Internet Protocol assessment, and Gryphon for network layers 4-7 switch, etc. He has over 40 research publications and five software patents..

Stevens Institute of Technology
Electrical and Computer Engineering Department
Hoboken, N.J. 07030

WEB SITES

INCOSE

www.incose.org

The International Council on Systems Engineering (INCOSE) was formed to develop, nurture, and enhance the interdisciplinary approach and means to enable the realization of successful systems. INCOSE works with industry, academia, and government to disseminate systems engineering knowledge, promote collaboration in systems engineering, establish integrity in systems engineering standards, and encourage research and educational support for systems engineering processes and practices.

Where in Federal Contracting?

www.wifcon.com

Where in Federal Contracting? is a free, non-commercial site that serves the federal and state acquisition and the federal assistance community, including public and private organizations. It provides quick access to acquisition and assistance information such as contract laws and pending legislation, current and proposed regulations, courts and boards of contract appeals, bid protest decisions, contracting newsletters, selected analysis of federal acquisition issues, federal assistance policy, daily listings of grants and cooperative agreements, archived listings of grants and cooperative agreements, and federal assistance sites.

Practical Software and Systems Measurement

www.psmsc.com

Practical Software and Systems Measurement (PSM): A Foundation for Objective Project Management was developed to meet today's software and system technical and management challenges. The Department of Defense and the U.S. Army sponsor PSM. The goal of the project is to provide project managers with the objective information needed to successfully meet cost, schedule, and technical objectives on programs. The PSM is based on actual measurement experience on DoD, government, and industry programs. The PSM supports current software and system acquisition and measurement policy.