Reinitializing to a known operating state before a failure occurs or reconfiguring after a failure such that the service the software provides remains operational.

The authors present two complementary ways of dealing with software aging: reinitializing to a known operating state before a failure occurs or reconfiguring after a failure such that the service the software provides remains operational.

Achieving Fault-Tolerant Software with Rejuvenation and Reconfiguration

William Yurcik and David Doss, Illinois State University

Requirements for constantly functioning software have increased dramatically with commercialization of the Internet. Application service providers and service-level agreements specify contractual software performance in terms of guaranteed availability and error thresholds (failed connection attempts, transaction failures, and fulfillment failures). These requirements are difficult to satisfy, particularly as applications grow in complexity, but the alternative of letting systems unpredictably crash is becoming less of an option. Such crashes are becoming increasingly expensive to business and potentially life-threatening to those who depend on essential services built on networked software systems.

As the makeup of systems is increasingly composed of software relative to hardware, system crashes are more likely to be the result of a software fault than a hardware fault. Although enormous efforts go into developing defect-free software, it isn’t always possible to find and eliminate every software bug. Software engineers develop software that works in the best of all possible worlds, but the real world includes environmental disruptions, transient faults, human errors, and malicious attacks. Building constantly functioning software systems in such a highly dynamic and unbounded environment is a challenge.

Even if individual software could be certifiably “assured” as bug-free, this assured software would likely have to execute on systems with “nonassured” software that could potentially introduce new faults into the system. Developing systems through software integration and reuse (rather than customized design) has become a cornerstone of modern software engineering. Thus, when considering software systems as a whole, it is prudent to assume that bugs are inherent and software should be fault tolerant.

Furthermore, when specific software continuously executes, software aging occurs: The software ages due to error conditions that accumulate with time and use. Causes include memory leaks, memory fragmentation, memory bloating, missing scheduling deadlines, broken pointers, poor register use, and build-up of numerical round-off errors. This aging manifests itself
Software decay is a proposed phenomenon, and it refers to the changing behavior of software. Specifically, software decay refers to software that degrades through time as it becomes increasingly difficult and expensive to maintain. If software remains in the same supporting environment, it is possible for it to constantly function without changing its behavior, but this is unrealistic. Hardware- and software-supporting environments change over time, and new features are added to change or enhance functionality. The original software architects can’t anticipate all possible changes, so unanticipated changes sometimes violate design principles or fail to follow the intent of imprecise requirements. The result is software that has decayed in function and would be more efficient if completely rewritten.

While software decay occurs in incremental steps through change processes that humans initiate, software aging occurs through underlying operating system resource management in response to dynamic events and a varying load over time.

Software rejuvenation

Most software theory has focused on static behavior by analyzing software listings. Little work was performed on longitudinal dynamic behavior and performance under varying loads until Yennun Huang and colleagues introduced the concept of software rejuvenation in 1995. Software rejuvenation is a proactive approach that involves stopping executing software periodically, cleaning internal states, and then restarting the software. Rejuvenation may involve all or some of the following: garbage collection, memory defragmentation, flushing operating system kernel tables, and reinitializing internal data structures.

Software rejuvenation does not remove bugs resulting from software aging but rather prevents them from manifesting themselves as unpredictable whole system failures. Periodic rejuvenation limits the state space in the execution domain and transforms a nonstationary random process into a stationary process that can be predicted and avoided.

To all computer users, rejuvenation is as intuitive as occasionally rebooting your computer. Of course, Murphy’s Law holds that this reboot will occur when irreplaceable data will be lost. Examples of using large-scale software rejuvenation include:

- on-board preventative maintenance for long-life deep space missions (1998).
- Rejuvenation is similar to preventive maintenance for hardware systems. While rejuvenation incurs immediate overhead in terms of some services being temporarily unavailable, the idea is to prevent more lengthy unexpected failures from occurring in the future.
- Cluster computing also provides a similar fault tolerance by using planned outages. When we detect a failure in one computer in a cluster, we can “fail over” the executing process to another computer within the cluster. Similar to rejuvenation, computers can be removed from the cluster as if under failure, serviced, and upgraded, and then restored back to the cluster. This ability to handle unexpected failures and scheduled maintenance makes clusters the only information systems that can attain 100 percent availability.
- The critical factor in making scheduled downtime preferable to unscheduled downtime is determining how often a system must be rejuvenated. If unexpected failures are catastrophic, then a more aggressive rejuvenation schedule might be justified in terms of cost and availability. If unexpected failures are equivalent to scheduled downtime in terms of cost and availability, then a reactive approach is more appropriate. Currently, the two techniques used to determine an optimal rejuvenation schedule are a

Reference

N-Version Programming

NVP, first proposed by Algirdas Avizienis in 1977, refers to multiple (N > 2) functionally equivalent program versions based on the same specification. To provide fault tolerance, each version must employ design diversity (different algorithms and programming languages) to maximize the probability that any error results are distinguishable. This is based on the conjecture that the probability of a random, independent fault producing the same error results in two or more versions is less when the versions are diverse.

A consistent set of inputs is supplied to all N versions and all N versions are executed in parallel. Similar to majority-voting hardware units, a consensus software decision mechanism then examines the results from all N versions to determine the accurate result and mask error results. NVP is increasingly feasible because asymmetrical multiprocessing now allows different processors running different operating systems for applications requiring reliability. Research continues into the feasibility of NVP for different problems and whether the NVP assumption of independent failures from functionally equivalent but independently developed versions holds (or whether failures remain correlated).

Reference


measurement-based technique (which estimates rejuvenation timing based on system resource metrics) and a modeling-based technique (which uses mathematical models and simulation to estimate rejuvenation timing based on predicted performance). IBM is pioneering software rejuvenation technology, in conjunction with Duke University, and products are beginning to appear. Software rejuvenation has been incorporated in IBM’s Netfinity Director for Microsoft Windows-based IBM and non-IBM servers, desktops, workstations, and notebook systems, and an extension has been created for Microsoft Cluster Service.

Software reconfiguration

In contrast to proactive rejuvenation, fault-tolerance techniques have traditionally been reactive. The reactive approach to achieving fault-tolerant software is to reconfigure the system after detecting a failure—and redundancy is the primary tool used. For hardware, the reconfiguration approach to providing fault tolerance is redundancy in terms of backup processors, power supplies, disk drives, and circuits.

For software, the reconfiguration approach uses redundancy in three different dimensions:

- software redundancy expressed as independently-written programs performing the same task executing in parallel (see the “N-Version Programming” sidebar) and comparing outputs;
- time redundancy expressed as repetitively executing the same program to check consistent outputs; and
- information redundancy expressed as redundancy bits that help detect and correct errors in messages and outputs.

Redundancy in these three dimensions provides flexible and efficient recovery, independent of knowledge about the underlying failure (such as fault identification or causal events). While robust software can be built with enough redundancy to handle almost any arbitrary failure, the challenge is to provide fault tolerance by minimizing redundancy—which reduces cost and complexity.

However, reactive techniques don’t have to mean that a system must crash before it can be gracefully recovered. Software reconfiguration can use redundant resources for real-time recovery while dynamically considering a large number of factors (operating system services, processor load, and memory variables among others)—a human-in-the-loop might not be necessary.

To PC users, however, reactive reconfiguration means recovery after a system crash. When your PC freezes, reconfiguration can take place by listing executing processes (<ctrl> <alt> <delete>) and attempting to identify and terminate the process responsible for the problem, often using a trial-and-error approach. If something catastrophic has occurred, a reboot from tape backup or original system disks might be necessary. Realistically, most users do not keep current backups, so a market for automatic software reconfiguration products has taken off (see the “Software Reconfiguration Products” sidebar).

Reconfiguration techniques have been pioneered for fault-tolerant networks and include,

- preplanned reconfiguration with disjoint working and backup circuits, such that after detecting a failure in a working circuit, the traffic can be automatically rerouted to its dedicated backup circuit;
- dynamic reconfiguration, such that after detecting a failure, signaling messages search the network for spare circuits on which to reroute traffic;
multilayer reconfiguration, in which recovery from a failure at one layer might take place at higher layers either independently or in coordination with each layer having different characteristics; and
- priority reconfiguration, which might involve re-optimizing an entire network to reconnect disrupted high-priority circuits over currently established lower-priority circuits.

Each of these reconfiguration techniques requires the provisioning of spare resources for redundancy that can be used when a failure occurs. The redundant resources can be dedicated, to guarantee reconfiguration, or shared, in which case recovery might not be possible (spare resources might not be available at the time of a fault). On the other hand, sharing redundant resources is more efficient in environments of low fault probability or when reconfiguration need not be guaranteed.

Reconfiguration can be provided at different layers and implemented with different algorithms at each layer. In fact, if all restoration mechanisms are similar at each layer, there is increased whole system vulnerability. For example, if all layers used preplanned mechanisms, then each layer—and the system as a whole—will not be able to handle unexpected fault events. If all layers used real-time search algorithms, then system behavior would be hard to predict. Instead, it is better to use complementary reconfiguration algorithms at different layers and draw on the benefits of each. For example, a preplanned algorithm at a lower layer (for speed), followed by a real-time search mechanism at a higher layer, can handle unexpected faults that lower layers have been unable to handle.

In general, reconfiguration of successfully executing software for recovery from a failure in another part of a system should only be performed if it can be accomplished transparently such that it is imperceptible to users. However, there are cases when high-priority software fails and requires resources for recovery. In this scenario, lower-priority software should be delayed or terminated and its resources reassigned to aid this recovery. Intentional system degradation to maintain essential processing is the most extreme type of reconfiguration.

Fault-tolerant software requires a whole system approach. We have attempted to outline the use of contrasting proactive and reactive approaches to achieve fault-tolerant software, but it’s not easy to say which approach is better. Both approaches are nonexclusive and complementary, such that they work well together in an integrated system (see Figure 1).

The high cost of redundancy required for reactive reconfiguration suggests it is better suited for software in which a rejuvenation schedule appears unrealistic due to imminent faults or where an outage’s effect could be catastrophic. Proactive rejuvenation is the preferred solution when faults can be efficiently avoided using a realistic rejuvenation schedule or where the risk an outage presents is low. Because both approaches are relatively new and under study, we direct readers to our references for more details.

Software Reconfiguration Products

Given the maturation of software reconfiguration techniques, products have begun to appear, particularly in the PC operating system market. These products do not protect from hardware failures, such as CPU malfunction or disk failure, but they can be useful tools against buggy software and human error. In general, these products track software changes (system, application, data file, and registry setting), use a hard disk to make redundant copies, and let the user restore (reconfigure) a system to a previous “snapshot.” Note that there are trade-offs for providing this reconfiguration capability versus system performance and hard disk space requirements.

For more information, here is a representative sampling of current products:

- ConfigSafe v 4.0, by imagine LAN (www.configsafe.com),
- GoBack v 2.21, by Roxio (www.roxio.com),
- Rewind, by Power On Software (www.poweronsoftware.com), and
- System Restore Utility, included in Microsoft Windows ME (www.microsoft.com).

![Figure 1. A model depicting the complementary nature of rejuvenation and reconfiguration.](image-url)
An analogy can be made between these fault-tolerant software approaches and CPU communications in a computer system. Re-active reconfiguration is equivalent to event-driven interrupts, and proactive rejuvenation is equivalent to polling resources. Preventing failures before they occur might be the best approach when finding all software bugs is possible, just as polling is preferable when CPU communication can be anticipated. However, when finding all bugs is improbable (or maybe testing is not even attempted), then having the flexibility to react to multi-priority interrupts with robust service-handing routines might be the critical last line of defense against software faults.

Acknowledgments
The authors thank Katerina Goseva-Popstojanova, Duke University, who provided an outstanding introduction to the concept of rejuvenation; David Tipper, University of Pittsburgh, and Deep Medhi, University of Missouri-Kansas City, for making significant contributions to the field of fault-tolerant networking using reconfiguration; and Kishor S. Trivedi, Duke University, for making seminal contributions in the development of software rejuvenation. Lastly, we thank past reviewers for their specific feedback that has significantly improved this article.

References

For further information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.

About the Authors

William Yurcik is an assistant professor in the Department of Applied Computer Science at Illinois State University. Prior to his academic career, he worked for organizations such as the Naval Research Laboratory, MITRE, and NASA. Contact him at wyurci@ilstu.edu.

David Doss is an associate professor and the graduate program coordinator in the Department of Applied Computer Science at Illinois State University. He is also a retired Lt. Commander US Navy (SSN). Contact him at ddoss@ilstu.edu.