Appears in Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), May 2001

Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel

George Candea Armando Fox
Stanford University
{candea,fox}@cs.stanford.edu

Abstract

Even after decades of software engineering research, complex computer systems still fail, primarily due to nondeterministic bugs that are typically resolved by rebooting. Conceding that Heisenbugs will remain a fact of life, we propose a systematic investigation of restarts as "high availability medicine." In this paper we show how recursive restartability (RR) — the ability of a system to gracefully tolerate restarts at multiple levels — improves fault tolerance, reduces time-to-repair, and enables system designers to build flexible, highly available software infrastructures. Using several examples of widely deployed software systems, we identify properties that are required of RR systems and outline an agenda for turning the recursive restartability philosophy into a practical software structuring tool. Finally, we describe infrastructural support for RR systems, along with initial ideas on how to analyze and benchmark such systems.

1 Introduction

Despite decades of research and practice in software engineering, latent and pseudo-nondeterministic bugs in complex software systems persist; as complexity increases, they multiply further, making it difficult to achieve high availability. It is common for such bugs to cause a system to crash, deadlock, spin in an infinite loop, livelock, or to develop such severe state corruption (memory leaks, dangling pointers, damaged heap) that the only high-confidence way of continuing is to restart the process or reboot the system.

The rebooting "technique" has been around as long as computers themselves, and remains a fact of life for substantially all nontrivial systems today. Rebooting can be applied at various levels: Deadlock resolution in commercial database systems is typically implemented by killing and restarting a deadlocked thread in hopes of avoiding a repeat deadlock [15]. Major Internet portals routinely kill and restart their web server processes after waiting for them to quiesce, in order to deal with known memory leaks that build up quickly under heavy load. A major search engine periodically performs rolling reboots of all nodes in their search engine cluster [3]. Although rebooting is often only a crude "sledgehammer" for maintaining system availability, its use is motivated by two common properties:

1. **Restarting works around Heisenbugs.** Most software bugs in production quality software are Heisenbugs [27, 8, 17, 2]. They are difficult to reproduce, or depend on the timing of external events, and often there is no other way to work around them but by rebooting. Even if the source of such bugs can be tracked down, it may be more cost-effective to simply live with them, as long as they occur sufficiently infrequently and rebooting allows the system to work within acceptable parameters. The time to find and deploy a permanent fix can sometimes be intolerably long. For example, the Patriot missile defense system, used during the Gulf War, had a bug in its control software that could be circumvented only by rebooting every 8 hours. Delays in sending a fix or the reboot workaround to the field led to 28 dead and 98 wounded American soldiers [34].

2. **Restarting can reclaim stale resources and clean up corrupt state.** This returns the system to a known, well-tested state, albeit with possible loss of data integrity. Corrupt or stale state, such as a mangled heap, can lead to some of the nastiest bugs, causing extensive periods of downtime. Even if a buggy process cannot be trusted to clean up its own resources, entities with hierarchically higher supervisory roles (e.g., the operating system) can cleanly reclaim any resources used by the process and restart it.

Rebooting is not usually considered a graceful way to keep a system running – most systems are not designed to tolerate unannounced restarts, hence experiencing extensive and costly downtime when rebooted, as well as potential data loss. Case in point: UNIX systems that are abruptly halted without calling `sync()`.

The Gartner Group [31] estimates that 40% of unplanned downtime in business environments is due to application failures; 20% is due to hardware faults, of which 80% are transient [8, 25], hence resolvable through reboot. Starting from this observation, we argue that in an *appropriately designed* system, we can *improve* overall system availability through a combination of reactively restarting failed components (revival) and prophylactically restarting functioning components (rejuvenation) to prevent state degradation that may lead to unscheduled downtime. Correspondingly, we present initial thoughts on how to design for recursive restartability, and outline a research agenda for systematic investigation of this area.

The paper is organized as follows: In section 2, we explain how the property of being recursively restartable can improve a system's overall availability. In section 3, we present examples of existing restartable and non-restartable systems. Section 4 identifies some required properties for recursively restartable systems and proposes an initial design framework. Finally, in section 5, we outline a research agenda for converting our observations into structured design rules and software tools for building and evaluating recursively restartable systems. Many of the basic ideas we leverage have appeared in the literature, but have not been systematically exploited as a collection of guidelines; we will highlight related work in the context of each idea.

2 Recursive Restartability Can Improve Availability

"Recursive restartability" (RR) is the ability of a system to tolerate restarts at multiple levels. An example would be a software infrastructure that can gracefully tolerate full reboots, subsystem restarts, and component restarts. An alternate definition is provided by the following recursive construction: the simplest, base-case RR system is a restartable software component; a general RR system is a composition of RR systems that obeys the guidelines of section 4. In the present section we describe properties of recursively restartable systems that lead to high availability.

RR improves fault tolerance. The unannounced restart of a software component is seen by all other components as a temporary failure; systems that are designed to tolerate such restarts are inherently tolerant to all transient non-Byzantine failures. Since most manifest software bugs and hardware problems are short lived [25, 27, 8], a strategy of failure-triggered, reactive component restarts will mask most faults from the outside world, thus making the system as a whole more fault tolerant.

RR can make restarts cheap. The fine granularity of recursive restartability allows for a bounded portion of the system to be restarted upon failure, hence reducing the impact on other components. This way, the system's global time-to-repair is minimized (e.g., full reboots are replaced with partial restarts), which increases availability. Similarly, RR allows for components and subsystems to be independently rejuvenated on a rolling basis; such incremental rejuvenation, unlike full application reboots, makes software rejuvenation [21] affordable for a wide range of 24×7 systems.

RR provides a confidence continuum for restarts. The components of a recursively restartable system are tied together in an abstract "restartability tree," in which (a) siblings are well isolated from each other by the use of simple, high-confidence machinery, and (b) a parent can unilaterally start, stop, or reclaim the resources of any of its children, using the same kind of machinery. For example, in a cluster-based network service, the root of the tree would be an administrator, each child of the root would be a node's OS, each grandchild a process on a node, and each great-grandchild a kernel-level process thread. This tree captures the tradeoff that, the closer to the root a restart occurs, the more expensive the ensuing downtime, but the higher the confidence that transient failures will be resolved. In the above example, processes are fault-isolated from each other by the hardware-supported virtual memory system, which is generally a high-confidence field-tested mechanism. The same mechanism also allows parents to reclaim process resources cleanly. Nodes are fault-isolated by virtue of their independent hardware. When a bug manifests, we can use a cost-of-downtime/benefit-of-certainty tradeoff to decide whether to restart threads, processes, nodes, or the entire cluster.

RR enables flexible availability tradeoffs. The proposed rejuvenation/revival regimen can conveniently be tailored to best suit the application and administrators: it can be simple (reboot periodically) or sophisticated (differentiated restart treatment for each subsystem/component). Identical systems can have different revival and rejuvenation policies, depending on the application's requirements and the environment they are in. Scheduled non-uniform rejuvenation can transform unplanned downtime into planned, shorter downtime, and it gives the ability to more often rejuvenate those components that are critical or more prone to failure. For example, a recent history of revival restarts and load characteristics can be used to automatically decide how often each component requires rejuvenation. Simpler, coarse-grained solutions have already been proposed by Huang et al. [21] and are used by IBM's xSeries servers [22].

3 Existing Systems

Very few systems today can be classified as being recursively restartable. Many systems do not tolerate restarts at all, and we provide some examples in this section. Others, though not necessarily designed by following an existing set of RR principles, fortuitously exhibit RR-friendly properties. Our long term goal is to derive a canon of design rules, including tradeoffs and programming model semantics, so that future efforts will be more systematic and deliberate.

3.1 Poorly Restartable Systems

In software systems not designed for restartability, the transient failure of one or more components often ends up being treated as a permanent failure. Depending on the system's design, the results can be anywhere from inconvenient to catastrophic. NFS [30] exhibits a flavor of this problem in its implementation of locking: a crash in the lock subsystem can result in an inconsistent lock state between a client and the server, which sometimes requires manual intervention by an administrator to repair. The result is that many applications requiring file locks test whether they are running on top of NFS and, if so, perform their own locking using the local filesystem, thereby defeating the NFS lock daemon's purpose.

As a more serious example, in July 1998, the USS Yorktown battleship lost control of its propulsion system due to a string of events started by a data overflow. Had the overall system been recursively restartable, its components could have been independently restored, avoiding the need to have the entire missile cruiser towed back to port [10].

Many UNIX applications use the `/tmp` directory for temporary files. Should `/tmp` become unavailable (e.g., due to a disk upgrade), programs will typically hang in the I/O system calls. Consequently, these monolithic, tightly coupled applications become crippled and cannot be restarted without losing all the work in progress.

Tightly coupled operating systems belong in this category as well. For example, Windows NT depends on the presence of certain system libraries (DLLs); accidentally deleting one of them can cause the entire system to hang, requiring a full reboot and the loss of all applications' work in progress. In the ideal case, an administrator would be able to replace the DLL and restart the dependent component, allowing the system to continue running. If the failed component was, say, the user interface on a machine running a web server, RR would allow availability of the web service to be unaffected. The ability to treat operating system services as separate components can avoid these failures, as evidenced by true microkernels [1, 24].

3.2 Restartability Winners

The classic replicated Internet server configuration has n instances of a server for a population of u users, with each server being able to handle in excess of u/n users. In such systems, node reboots result simply in a transient $1/n$ throughput loss. Moreover, read-only databases can be striped across these instances such that each node contributes a fixed fraction of DQ (data/query $\times$ queries/unit time) [4]. Independent node reboots or transient node failures result solely in decreased data/query, while keeping overall queries/unit time constant. Such a design makes "rolling rejuvenation" very affordable [3].

At major Internet portals, it is not uncommon for newly hired engineers to write production code for the system after little more than one week on the job. Simplicity is stressed above all else, and code is often written under the explicit assumption that it will necessarily be killed and restarted frequently. This affords programmers such luxuries as never calling `free()` in their C code, thereby avoiding an entire class of pernicious bugs.

Finally, NASA's Mars Pathfinder illustrates the value of coarse-grained reactive restarts. Shortly after landing on Mars, the spacecraft identified that one of its processes failed to complete execution on time, so the control software decided to restart all the hardware and software [28]. Despite the fact that the software was imperfect — it was later found that the hang had been caused by a hard-to-reproduce priority-inversion deadlock — the watchdog timers and restartable control system saved the mission and helped it exceed its intended lifetime by a factor of three.

4 The Restart Scalpel: Toward Structured Recursive Restartability

In proposing RR, we are inspired by the effect of introducing ACID (atomic, consistent, isolated, durable) transactions [16] as a building block many years ago. Not only did transactions greatly simplify the design of data management systems, but they also provided a clean framework within which to reason about the error behavior of such systems. Our goal is for recursive restartability to offer the same class of benefits for systems where ACID semantics are not required or are expensive to engineer, given the system's availability or performance goals. In particular, we address systems in which weaker-than-ACID requirements can be exploited for tradeoffs that improve availability or simplicity of construction.

In this section we make some observations about the properties of RR-friendly systems, and propose guidelines for how RR subsystems can be assembled into more complex RR systems. The overarching theme is that of designing applications as loosely coupled distributed systems, even if they are not distributed in nature.

Accepting *No* for an answer. Software components should be designed such that they can deny service for any request or call. Then, if an underlying component can say *No*, applications must be designed to take *No* for an answer and decide how to proceed: give up, wait and retry, reduce fidelity, etc. Such components can then gracefully tolerate the temporary unavailability of their peer, as evidenced in the cluster-based distributed hash table described by Gribble et al. [19]. Dealing with *No* answers in the callers, as opposed to trying to cope with them in the server, closely follows the end-to-end argument [29]. Moreover, Lampson observes that such error handling is absolutely necessary for a reliable system anyway [23].

> Subsystems should make their interface guarantees sufficiently weak, so they can occasionally restart with no advance warning, yet not cause their callers to hang/crash.

Using reconstructable soft state with announce/listen protocols. Soft state and announce/listen have been extensively used at the network level [37, 9] as well as the application level [12]. Announce/listen makes the default assumption that a component is unavailable unless it says otherwise; soft state can provide information that will carry a system through a transient failure of the authoritative data source for that state. The use of announce/listen with soft state allows restarts and "cold starts" to be treated as one and the same, using the same code path. Moreover, complex recovery code is no longer required, thus reducing the potential for latent bugs and speeding up recovery.

Unfortunately, sometimes soft state systems cannot react quickly enough to deliver service within their specified time frame. Use of soft state implies tolerance of some state inconsistency, and sometimes the state may never stabilize. For exam-

ple, in a soft-state load balancer for a prototype scalable network server [14], the instability manifested as alternating saturation and idleness of workers. This was due to load balancing decisions based on worker load data that was too old. Mitzenmacher [26] derives a quantitative analytical model to capture the costs and benefits of using such stale information, and his model's predictions coincide with behavior observed in practice. This type of problem can be addressed by increasing refresh frequency, albeit with additional bandwidth and processing overhead.

> State shared among subsystems should be mostly soft. The extent of soft state depends on (a) the application's convergence and response-latency requirements and (b) the refresh frequency supported by the inter-component communication substrate (which is a function not only of "raw" bandwidth and latency but also of "goodput").

Automatically trading precision or consistency for availability. Online aggregation [20], harvest/yield tradeoffs [13], and distributed databases such as Bayou [33] are examples of dynamic or adaptive trading of some property, usually either consistency or precision, for availability. Recently, TACT [36] showed how such tradeoffs could be brought to bear on systems employing replication for high availability, by using a framework in which consistency degradation is measured in application-specific units. The ability to make such tradeoffs dynamically and automatically during transient failures makes a system much more amenable to RR.

> Inter-component "glue" protocols should allow components to make dynamic decisions on trading consistency/precision for availability, based on both application-specific consistency/precision measures, and a consistency/precision utility function (e.g., "a perfectly consistent answer is twice as good as one missing the last two updates," or "a 100% precise answer is twice as good as a 90% precise answer").

Structuring applications around fine grain workloads. A primary example of fine grain workload requirements comes from HTTP: the Web's architecture has challenged application architects to design mechanisms for state maintenance and session identification, some more elegant than others. The result is that the Web as a whole exhibits the desirable property that individual server processes can be quiesced rapidly, since HTTP connections are typically short-lived, and servers are extremely loosely bound to their clients, given that the protocol itself is stateless. This makes them highly restartable and leads directly to the simple replication and failover techniques found in large cluster-based Internet services.

> "Glue" protocols should enforce fine grain interactions between subsytems. They should provide hooks for computing the cost of a subsystem's restart based on the expected duration of its current task and its children's tasks.

Using orthogonal composition axes. Independent subsystems that do not require an understanding of each other's functionality are said to be mutually orthogonal. Compositions of orthogonal subsystems exhibit high tolerance to component restarts, allowing the system as a whole to continue functioning (perhaps with reduced utility) in spite of temporary failures. There is a strong connection between good modular structure and the ability to exploit orthogonal mechanisms; systems that exploit them well seem to go even further: their control flows are completely decoupled, influencing each other only indirectly through explicit message passing. Examples of orthogonal mechanisms include deadlock resolution in databases [15], software-based fault isolation [35], as well as heartbeats and watchdogs used by process peers that monitor each others' liveness [14, 7].

> Split functionality along orthogonal axes. Each corresponding subsystem should be centered around an independent locus of control, and interact with other subsystems via events posted using an asynchronous mechanism.

5 Research Agenda and Evaluation

After refining the above design guidelines, evaluation of a RR research agenda will consist of answering at least three major categories of questions:

- What classes of applications are amenable to RR? What model would capture the behavior of these applications and allow them to be compared directly?
- How do we quantify the improvements in availability and the possible losses in performance, consistency or other functionality that may result from the application of RR?
- What software infrastructure and tools are necessary to execute the proposed automatic revival/rejuvenation policy?

5.1 Building RR Systems

Some existing applications, most notably Internet services, are already incorporating a subset of these techniques (usually in an ad hoc fashion) and are primary candidates for systematic RR. Similarly, many geographically dispersed systems can benefit if they tolerate weakened consistency, due to the potential lack of reliability in their communication medium. We suspect the spectrum of applications that are amenable to RR is much wider, but still needs to be explored.

Loosely coupled architectures often exhibit emergent properties that can lead to instability (e.g., noticed in Internet rout-

ing [11]) and investigating them is important for RR. There is also a natural tension between the cost of restructuring a system for RR and the cost (in downtime) of restarting it. Fine module granularity improves the system's ability to tolerate partial restarts, but requires the implementation of a larger number of internal, asynchronous interfaces. The paradigm shift required of system developers could make RR too expensive in practice and, when affordable, may lead to buggier software. In some cases RR is simply not feasible, such as for systems with inherent tight coupling (e.g., real-time closed-loop feedback control systems).

Finally, the key to wide adoption of recursive restartability are tools that can aid the software architect in deciding when to use a RR structure and how to apply the RR guidelines.

5.2 Quantifying Availability and the Effects of Recursive Restartability

A major contribution of the transaction concept was the emergence of a model, TP systems, that allowed different implementations of data management systems to be directly compared (e.g., using TPC benchmarks [18]). We are seeking an analogous model that characterizes applications possessing RR properties, and that can serve in quantifying availability.

Availability benchmarking has been of interest only for the past decade [32, 5]. It is considerably more difficult than performance benchmarking, because a fault model is required in addition to a workload, and certain aspects, such as software aging, cannot even be captured reliably. Performance benchmark results that ignore availability measurements, such as "our system obtained 300,000 tpmC", are dishonest — a fast system that is hung or crashed is simply an infinitely slow system. The converse holds for avalability benchmarks as well, so we seek a unified approach to the measurement of RR systems.

Given an application amenable to RR, a model, and a suitable benchmark, we must quantify the improvement in availability and the decrease in functionality (reduced precision, weaker consistency, etc.) when specific RR rules are applied. We expect that work such as TACT [36] and Mitzenmacher's models for usefulness of stale information [26] will provide a starting point for quantitative validation of RR.

We will identify application classes that, compared to their current implementations, are more tolerant of our guidelines (e.g., trading precision for availability). We will restructure the applications incrementally, while maintaining their semantics largely intact. Availability will be evaluated at different stages: (1) initial application; (2) recursively restartable version of the application; (3) RR version using our execution infrastructure (described below), with revival restarts; (4) RR version using the execution infrastructure with both revival and rejuvenation restarts.

5.3 RR Infrastructure Support

Recursively restartable systems rely on a generic execution infrastructure (EI) which is charged with instantiating the restartability tree mentioned in section 2, monitoring each individual component and/or subsystem, and prompting restarts when necessary. In existing restartable systems, the EI homologue is usually application-specific and built into the system itself.

The execution infrastructure relies on a combination of periodic application-specific probes and end-to-end checks (such as verifying the response to a well-known query) to determine whether a component is making progress or not. In most cases, application-specific probes are implemented by the components themselves via callbacks. When the EI detects an anomaly, it advises the faulty component that it should clean up any pending state because it is about to be restarted by its immediate ancestor in the restartability tree. An analogy would be UNIX daemons that understand the "`kill -TERM; sleep 5; kill -9`" idiom. If restarting does not eliminate the anomaly, a restart at a higher level of the hierarchy is attempted, similar to the return up a recursive call structure.

Note how the availability problem itself becomes recursive: we now need a highly available infrastructure that cares for the RR system. Medusa [6], our EI prototype, is functionally much simpler than most applications, making it possible to design and implement it with care. Medusa is built out of simple, highly restartable segments that run on different hosts, use multicast heartbeats to keep track of each other and their activity, and self-reinstantiate to replace dead segments.

6 Conclusion

In this paper we took the view that transient failures will continue plaguing the software infrastructures we depend on, and thus reboots are here to stay. We proposed turning the reboot from a demonic concept into a reliable partner in the fight against system downtime, given that it is a time-tested, effective technique for circumventing Heisenbugs.

We defined recursively restartable (RR) systems as being those systems that tolerate successive restarts at multiple levels. Such systems possess a number of valuable properties that by themselves improve availability. For instance, a RR system's fine granularity permits partial restarts to be used as a form of bounded healing, reducing the overall time-to-repair, and hence increasing availability. On top of these desirable intrinsic properties, we can employ an automated, recursive policy of component revival/rejuvenation to further reduce downtime.

Building RR systems in a systematic way requires a framework consisting of well-understood design rules. A first attempt at formulating such a framework was presented here, advocating the paradigm of building applications as distributed systems, even if they are not distributed in nature. We set forth a research agenda aimed at validating these proposals and verifying that re-

cursive restartability can be an effective supplement to existing high availability mechanisms. With recursive restartability, we hope to add a useful item to every system architect's toolbox.

7 Acknowledgments

We thank Peter Chen, David Cheriton, Jim Gray, Steve Gribble, Butler Lampson, David Lowell, Udi Manber, Dejan Milojicic, Milyn Moy, and Stanford's Mosquitonet and SWIG groups for helpful and stimulating comments on the ideas presented here.

References

[1] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. R. A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, pages 93–113, 1986.

[2] E. Adams. Optimizing preventative service of software products. *IBM J. Res. Dev.*, 28(1):2–14, 1984.

[3] E. Brewer. Personal communication. 2000.

[4] E. Brewer. Lessons from giant-scale services (draft). Submitted for publication, 2001.

[5] A. Brown and D. A. Patterson. Towards availability benchmarks: A case study of software RAID systems. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[6] G. Candea. Medusa: A platform for highly available execution. CS244C (Distributed Systems) course project, Stanford University, `http://stanford.edu/~candea/papers/medusa`, June 2000.

[7] Y. Chawathe and E. A. Brewer. System support for scalable and fault tolerant internet service. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, Lake District, UK, Sep 1998.

[8] T. C. Chou. Beyond fault tolerance. *IEEE Computer*, 30(4):31–36, 1997.

[9] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei, P. Sharma, and A. Helmy. Protocol independent multicast (PIM), sparse mode protocol: Specification, March 1996. Internet Draft.

[10] A. DiGiorgio. The smart ship is not enough. *Naval Institute Proceedings*, 124(6), June 1998.

[11] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Transactions on Networking*, 2(2):122–136, Apr. 1994.

[12] S. Floyd, V. Jacobson, C. Liu, and S. McCanne. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. In *ACM SIGCOMM '95*, pages 342–356, Boston, MA, Aug 1995.

[13] A. Fox and E. A. Brewer. ACID confronts its discontents: Harvest, yield, and scalable tolerant systems. In *Seventh Workshop on Hot Topics In Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[14] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, St.-Malo, France, October 1997.

[15] J. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, J. H. Saltzer, and G. Seegmüller, editors, *Operating Systems, An Advanced Course*, volume 60, pages 393–481. Springer, 1978.

[16] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of VLDB*, Cannes, France, September 1981.

[17] J. Gray. Why do computers stop and what can be done about it? In *Proc. Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[18] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufman, 2 edition, 1993.

[19] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, October 2000.

[20] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *ACM–SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

[21] Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *International Symposium on Fault-Tolerant Computing*, pages 381–390, 1995.

[22] International Business Machines. IBM director software rejuvenation. White paper, Jan. 2001.

[23] B. W. Lampson. Hints for computer systems design. *ACM Operating Systems Review*, 15(5):33–48, 1983.

[24] J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[25] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing relevance of memory hardware errors. a case for recoverable programming models. In *ACM SIGOPS European Workshop "Beyond the PC: New Challenges for the Operating System"*, Kolding, Denmark, Sept. 2000.

[26] M. Mitzenmacher. How useful is old information? In *Principles of Distributed Computing (PODC) 97*, pages 83–91, 1997.

[27] B. Murphy and N. Davies. System reliability and availability drivers of Tru64 UNIX. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, Madison, WI, February 1999. IEEE Computer Society. Tutorial.

[28] G. Reeves. What really happened on Mars? RISKS-19.49, Jan. 1998.

[29] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[30] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the USENIX Summer Conference*, pages 119–130, Portland, OR, 1985.

[31] D. Scott. Making smart investments to reduce unplanned downtime. Tactical Guidelines Research Note TG-07-4033, Gartner Group, Stamford, CT, 1999.

[32] D. P. Siewiorek, J. J. Hudak, B.-H. Suh, and Z. Segall. Development of a benchmark to measure system robustness. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 88–97, 1993.

[33] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, TX, Sept. 1994.

[34] U.S. General Accounting Office. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Technical Report GAO/IMTEC-92-26, 1992.

[35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP-14)*, 1993.

[36] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, Oct. 2000.

[37] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource Reservation Protocol. *IEEE Network*, 7(5), Sept. 1993.