Crash-Only Software

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

Abstract

Crash-only programs crash safely and recover quickly. There is only one way to stop such software—by crashing it—and only one way to bring it up—by initiating recovery. Crash-only systems are built from crash-only components, and the use of transparent component-level retries hides intra-system component crashes from end users. In this paper we advocate a crash-only design for Internet systems, showing that it can lead to more reliable code, easier failure prevention, and faster, more effective recovery. We present ideas on how to build such crash-only Internet services, taking successful techniques to their logical extreme.

1. Occam's Razor and the Restart Potpourri

There are many reasons to restart software, and many ways to do it. Studies have shown that a main source of downtime in large scale software systems is caused by intermittent or transient bugs [14, 22, 21, 1]. Most non-embedded systems have a variety of ways to stop; for example, an operating system can shut down cleanly, panic, hang, crash, lose power, etc.

When shutting down programs cleanly, unavailability consists of the time to shut down and the time to come back up; when crash-rebooting, unavailability consists only of the time to recover. Ironically, shutting down and reinitializing can sometimes take longer than recovering from a crash. Table 1 illustrates a casual comparison of reboot times; no important data was lost in either of the experiments.

System	Clean reboot	Crash reboot
RedHat 8 (w/ ext3fs)	104 sec	75 sec
JBoss 3.0 application server	47 sec	39 sec
Windows XP	61 sec	48 sec

Table 1. Duration of clean vs. crash reboots.

It is impractical to build a system that is guaranteed to never crash, even in the case of carrier class phone switches or high end mainframe systems. Since crashes are unavoidable, software must be at least as well prepared for a crash as it is for a clean shutdown. But then—in the spirit of Occam's Razor—if software is crash-safe, why support additional, non-crash mechanisms for shutting down? A frequent reason is the desire for higher performance.

For example, to avoid slow synchronous disk writes, many UNIX file systems cache metadata updates in memory. As a result, when a UNIX workstation crashes, the file system reaches an inconsistent state that takes a lengthy `fsck` to repair, an inconvenience that could have been avoided by shutting down cleanly. This captures the design tradeoff that improves steady state performance at the expense of shutdown and recovery performance. In the face of inevitable crashes, such a file system turns out to be brittle: a crash can lose data and, in some cases, the post-crash inconsistency cannot even be repaired. Not only do such performance tradeoffs impact robustness, but they also lead to complexity by introducing multiple ways to manipulate state, more code, and more APIs. The code becomes harder to maintain and offers the potential for more bugs—a fine tradeoff, if the goal is to build fast systems, but a bad idea if the goal is to build highly available systems. If the cost of such performance enhancements is dependability, perhaps it's time to reevaluate our design strategy.

In earlier work, we used recursive micro-reboots to improve the availability of a soft-state system that was trivially crash-safe [4]. In this paper we advocate a *crash-only design* (i.e., crash safety + fast recovery) for Internet systems, a class distinguished by the following properties: large scale, stringent high availability requirements, built from many heterogenous components, accessed over standard request-reply protocols such as HTTP, serving workloads that consist of large numbers of relatively short tasks that frame state updates, and subjected to rapid and perpetual evolution. We restrict our attention to single installations that reside inside one data center, that don't span administrative domains and don't communicate over the WAN.

In high level terms, a crash-only system is defined by the equations *stop=crash* and *start=recover*. In the rest of the paper, we describe the benefits of the crash-only design approach by analogy to physics, describe the internal properties of components in a crash-only system, the architectural properties governing the interaction of components, and a restart/retry architecture that exploits crash-only design, including our work to date on a prototype using J2EE.

2. Why Crash-Only Design ?

Mature engineering disciplines rely on macroscopic *descriptive* physical laws to build and understand the behavior of physical systems. These sets of laws, such as Newtonian mechanics, capture in simple form an observed physi-

cal invariant. Software, however, is an abstraction with no physical embodiment, so it obeys no physical laws. Computer scientists have tried to use *prescriptive* rules, such as formal models and invariant proofs, to reason about software. These rules, however, are often formulated relative to an abstract model of the software that does not completely describe the behavior of the running system (which includes hardware, an operating system, runtime libraries, etc.). As a result, the prescriptive models do not provide a complete description of how the implementation behaves in practice, because many physically possible states of the complete system do not correspond to any state in the abstract model.

With the crash-only property, we are trying to impose, from outside the software system, macroscopic behavior that coerces the system into a simpler, more predictable universe with fewer states and simpler invariants. Each crash-only component has a single idempotent "power-off switch" and a single idempotent "power-on switch"; the switches for larger systems are built by wiring together their subsystems' switches in ways described by section 3. A component's power-off switch implementation is entirely external to the component, thus not invoking any of the component's code and not relying on correct internal behavior of the component. Examples of such switches include `kill -9` sent to a UNIX process, or turning off the virtual, or physical, machine that is running some software inside it.

Keeping the power-off switch mechanism external to components makes it a high confidence "component crasher." Consequently, every component in the system must be prepared to suddenly be deactivated. Power-off and power-on switches must provide a small repertoire of high-confidence, simple behaviors, leading to a small state space. Of course, the "virtual shutdown" of a virtual machine, even if invoked with `kill -9`, has a much larger state space than the physical power switch on the workstation, but it is still vastly simpler than the state space of a typical program hosted in the VM, and it does not vary for different hosted programs. Indeed, the fact that virtual machines are relatively small and "simple" compared to the applications they host has been successfully invoked as an argument for using VMs for inter-application isolation [28].

2.1. Crash-Only and Transactions

Many of the benefits resulting from a crash-only design have been previously obtained in the data storage/retrieval world with the introduction of transactions. Our approach aims for a similar effect on the properties of Internet systems; while being less specific than the transaction model, crash-only design is also more general.

Internet applications do not have to use transactions in order to be crash-only; in fact, ACID semantics can sometimes lead to overkill. For example, session data accumulates information at the server over a series of user service requests, for use in subsequent operations. It is mostly single-reader/single-writer, thus not requiring ordering and concurrency control. Such state usually does not persist beyond a few minutes; the richness of a query language like SQL is unnecessary. These observations are leveraged by SSM [20], a crash-only hashtable-like session state store.

2.2. Crash-Only and Fault Model Enforcement

A crash-only system makes it affordable to coerce every detected failure into component-level crash(es); this leads to a simple fault model in that components only need to know how to recover from one type of failure. Fault model enforcement uses such an approach to turn unknown faults into crashes, effectively coercing reality into a well-understood, simple fault model. By performing recovery based on this fault model, [23] managed to improve availability in a cluster system. Much existing literature assumes unrealistic fault models (e.g., that failures are uncorrelated and occur according to well-behaved tractable distributions) for analysis of system behavior; fault model enforcement can increase the impact of such work.

Similarly, a system built from components that tolerate crashes at low cost makes it affordable to use software rejuvenation [18] to prevent failure. Rejuvenation can be triggered by fail-stutter behavior [3], a workload trough, or based on mathematical models of software aging [12].

2.3. Crash-Only and Recovery-Oriented Design

Recovery code deals with exceptional situations, and must run flawlessly. Unfortunately, exceptional situations are difficult to handle, occur seldom, and are not trivial to simulate during development; this often leads to unreliable recovery code. In crash-only systems, however, recovery code is exercised every time the system starts up, which should ultimately improve its reliability. This is particularly relevant given that the rate at which we reduce the number of bugs per Klines of code lags behind the rate at which the number of Klines per system increases, with the net result being that the number of bugs in an evolving system increases with time [9]. More bugs mean more failures, and systems that fail more often will need to recover more often.

3. Properties of Crash-Only Software

In this section we describe a set of properties that we deeem sufficient for a system to be crash-only. In some systems, some of these properties may not be necessary for crash-only behavior. In section 5 we to capture this fact by defining multiple levels of crash-only ("CO levels").

To make components crash-only, we require that all persistent state be kept in dedicated state stores, that state stores provide applications with the right abstractions, and that state stores be crash-only. To make a system of interconnected components crash-only, it must be designed so that components can tolerate their peers' crashes. This means

we require strong modularity with relatively impermeable component boundaries, timeout-based communication and lease-based resource allocation, and self-describing requests that carry a time-to-live and information on whether they are idempotent. Many Internet systems today have some subset of these properties, but we do not know of any that combines all properties into a true crash-only system.

In section 4 we will show how crash-only components can be glued together into a robust Internet system based on a restart/retry architecture; in the rest of this section we describe in more detail the seven properties of crash-only systems. The first three relate to intra-component state management, while the remaining four relate to inter-component interactions. While we recognize that many of these sacrifice performance, we strongly believe the time has come for robustness to reclaim its status as a first-class citizen.

3.1. Intra-Component Properties

In today's Internet applications there are a small number of types of state: transactional persistent state, single-reader/single-writer persistent state (e.g., user profiles, that almost never see concurrent access), expendable persistent state (important server-side information that could be sacrificed for the sake of correctness or performance, such as clickstream data and access logs), session state (e.g., shopping cart contents), soft state (state that can be reconstructed at any time based on other data sources), and volatile state. While differentiated mostly by guaranteed lifetime, the requirements for these categories of state lead to qualitatively different implementations.

Non-volatile state is managed by dedicated state stores, leaving applications with just program logic. Specialized state stores (e.g., relational and object-oriented databases, file system appliances, distributed data structures [16], non-transactional hashtables [17], session state stores [20], etc.) are much better suited to manage state than code written by developers with minimal training in systems programming. Applications become stateless clients of the state stores, which allow them to have simpler and faster recovery routines. A popular example of such separation can be found in three-tier Internet architectures, where the middle tier is largely stateless and relies on backend databases to store data.

State stores are crash-only, otherwise the problem has just moved down one level. Many commercial off-the-shelf state stores available today are crash-safe (i.e., they can be crashed without fearing the loss of data), such as databases and the various network-attached storage devices, but most are not crash-only, because they recover slowly. A large group of products, however, offer tuning knobs that permit the administrator to trade performance for improved recovery time, such as making checkpoints more often in the Oracle DBMS [19]. An example of a pure crash-only state store is the Postgres database system [27], which avoids write-ahead logging and maintains all data in one append-only log. Recovery is practically instantaneous, because it only needs to mark the transactions that were aborted because of being uncommitted at the time of the crash.

Abstractions and guarantees provided by the state store are congruent with application requirements. This means that the state abstraction exported by the state store is not too weak (e.g., offering a file interface for storing customer records) and not too powerful either (e.g., offering a SQL interface with ACID semantics for storing and retrieving simple key-value tuples). First, this property enables applications to operate at their "natural" semantic level. Second, by ensuring a close match between the offered and the required abstraction, we can exploit application semantics to build simpler, faster, more reliable state stores.

For example, Berkeley DB [24] is a storage system supporting B+tree, hash, queue, and record abstractions. It can be accessed through four different interfaces, ranging from no concurrency control/no transactions/no disaster recovery to a multi-user, transactional API with logging, fine-grained locking, and support for data replication. Applications can use the abstraction that is right for their purposes and the underlying state store optimizes its operation to fit those requirements. Workload characteristics can also be leveraged by state stores; e.g., expecting a read-mostly workload allows a state store to utilize write-through caching, which can significantly improve recovery time and performance.

We believe Internet systems will standardize on a small set of state store types: ACID stores (e.g., databases for customer and transaction data), non-transactional persistent stores (e.g., DStore [17], a crash-only system specialized in handling non-transactional persistent data, like user profiles), session state stores (e.g., SSM for shopping carts), simple read-only stores (e.g., file system appliances for static HTML and images), and soft state stores (e.g., web caches). If we think carefully about the state abstractions required by each application component and use suitable state stores, we can make these components crash-only.

3.2. Extra-Component Properties

For a crash-only system to gracefully tolerate subsystem crashes, which temporarily make them unavailable to serve requests, components and their relationships must follow a few guidelines.

Components have externally enforced boundaries that provide strong fault containment. The desired isolation can be achieved using virtual machines such as VMware, isolation kernels [28], task-based intra-JVM isolation [26, 10], OS processes, etc. Indeed, the Denali isolation kernel is designed for such lightweight encapsulation. For example, web service hosting providers often use multiple virtual machines on one physical machine to offer their clients individual web servers they can administer at will, without affecting other customers. The boundaries between components

delineate distinct, individually recoverable stages in the processing of requests.

All interactions have a timeout. This includes communication as well as timeout-based RPC: if no response is received to a call within the alloted timeframe, the caller assumes the callee has failed and reports it to a recovery agent [6], which can crash-restart the callee if appropriate. Crash-restarting helps ensure the called component is in a known state; this is safe because the component is crash-safe and crash-restart is idempotent. Timeouts provide an orthogonal mechanism for turning all non-Byzantine failures, both at the component level and at the network level, into fail-stop events (i.e., the failed entity either provides results or is stopped), even though the components are not necessarily fail-stop. Such behavior is easier to accomodate, and containment of faults is improved.

All resources are leased, rather than permanently allocated. This includes many types of persistent state, such as account profiles for a free e-mail provider: every time the user logs in, a 6-month lease is renewed; when the lease expires, all associated data can be purged from the system. It also includes CPU resources: if a computation lasts longer than can be afforded by the system, it is terminated (e.g., PHP, a server-side scripting language used for writing dynamic web pages, offers a function that limits the maximum execution time of a script; when this limit is reached, the script is killed and an error is returned). Leases [13] give us the ability to reason about the conditions that hold true after a lease expires: certain resources become available, and the component is in a known state. Infinite timeouts or leases are not acceptable; the maximum-allowed timeout and lease are specified in an application-global policy. This way it is less likely that the system will hang or become blocked.

Requests are self-describing continuations that carry information on whether they are idempotent, as well as a time-to-live. Continuations [11] force the state and context needed by the request to be made explicit, which makes state management simpler. The idempotency and TTL information can be set by the web front ends: timeouts as a function of load or service level agreements, and idempotency flags based on application-specific information (which can be derived, for instance, from URL substrings that determine the type of request). Many interesting operations in an Internet service are idempotent, or can easily be made idempotent by keeping track of sequence numbers or by wrapping requests in transactions; some large Internet services have already found it practical to do so [25]. Over the course of its lifetime, a request will split into multiple sub-operations, which may rejoin, in much the same way nested transactions do. Recovering from a failed idempotent sub-operation entails simply reissuing it; for non-idempotent operations, the system can either roll them back, apply compensating operations, or tolerate the inconsistency resulting from a retry. Such transparent recovery of the request stream can hide intra-system component failures from the end user.

4. A Restart/Retry Architecture

A component infers failure of a peer component either based on a raised exception or a timeout. When a component is reported failed, a recovery agent may crash-reboots it; the idempotency of crash-shutdown makes this an inexpensive way to ensure the component is indeed turned off before attempting recovery. Components waiting for an answer from the restarted component receive a *RetryAfter(n)* exception, indicating that the in-flight requests can be resubmitted after n msec (the estimated time to recover); this exception is purely an optimization because, in its absence, components would have timeouts as a fallback mechanism. If the request is idempotent and its time-to-live allows it to be resubmitted, then the requesting component does so. Otherwise, a failure exception is propagated up the request chain until either a previous component decides to resubmit, or the client web browser needs to be notified of the failure. The web front-end issues an HTTP/1.1 `Retry-After` directive to the web browser with an estimate of the time to recover, and retry-capable clients can resubmit the original HTTP request.

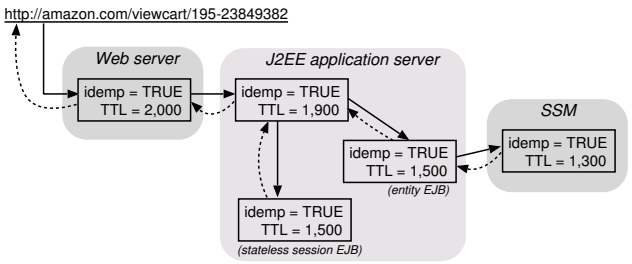

Figure 1. A simple restart/retry architecture.

In Figure 1 we show a simple restart/retry example, in which a request to view a shopping cart splits inside the application server into one subrequest to an entity Enterprise JavaBean (EJB) that communicates with a session state store and another subrequest to a stateless session EJB. Should the state store become unavailable, the application server either receives a *RetryAfter* exception or times out, at which time it can decide whether to resubmit the SSM-bound request or not. Within each of the subsystems shown in Figure 1, we can imagine each subrequest further splitting into finer grain subrequests submitted to the respective subsystems' components. We have implemented crash-restarting of EJBs, which we call micro-reboots; rebooting one EJB takes less than a second.

Timeout-based failure detection is supplemented with traditional heartbeats and progress counters. The counters—compact representations of a component's processing progress—are usually placed at state stores and in messaging facilities, where they can map state access and messaging activity into per-component progress. Many existing performance monitors can be transformed into progress monitors by augmenting them with request origin

information. Components themselves can also implement progress counters that more accurately reflect application semantics, but they are less trustworthy, because they are inside the components.

The dynamics of loosely coupled systems can sometimes be surprising. For example, resubmitting requests to a component that is recovering can overload it and make it fail again; for this reason, the *RetryAfter* exceptions provide an estimated time-to-recover. This estimated value can be used to spread out request resubmissions, by varying the reported time-to-recover estimate across different requestors. A maximum limit on the number of retries is specified in the application-global policy, along with the lease durations and communication timeouts. These numbers can be dynamically estimated based on historical information collected by a recovery manager [6], or simply captured in a static description of each component, similar to deployment descriptors for EJBs. In the absence of such hints, a simple load balancing algorithm or exponential backoff is used.

5. Levels of Crash-Only

Section 3 described properties that are sufficient for a system to be crash-only; however, they are not necessary in all cases. This suggests that crash-only is not an absolute property, but rather there may be various levels of "crash-onlyness." As was the case for RAID [8], defining such levels and their implications on RAID users is useful not only for classifying systems, but for building them as well.

	CO Level		
Crash-Only Property	Low	Med	High
Use dedicated state stores	√	√	√
State stores are crash-only			√
Congruent state abstractions		√	√
Strong isolation		√	√
Timeout-based interactions			√
Leased resources			√
Self-describing requests			√

Table 2. Levels of crash-only (CO) systems.

Table 2 shows what such a classification might look like. For example, a webmail application implemented using cgi-bin scripts may fall in the "low" CO category: it uses a file server as a dedicated state store, but does not meet any of the other requirements. To get to a "medium" CO level, we could implement the same web application on a J2EE platform that uses an object-oriented database (adds the property of congruent state abstractions) and the EJB component model (adds strong isolation within the application). Finally, to achieve a "high" level of CO, this application would have to be implemented on the ideal application server incorporating all the crash-only mechanisms—this may or may not be desirable.

6. Discussion

Building crash-only systems is not easy (for example, making databases crash-safe took a lot of careful engineering). The key to widespread adoption is to employ the right architectural models and to have the right tools. With the recent success of component-based architectures (e.g., J2EE and .Net), and the emergence of the application server as an operating system for Internet applications, it is possible to provide many of the crash-only properties in the platform itself. This would allow all applications running on that platform to take advantage of the effort and become crash-only.

We are applying the principles described here to an open-source Java 2 Enterprise Edition (J2EE) application server. We are separating the individual J2EE services (naming, directory lookup, messaging, etc.) into well-isolated components, implementing requests as self-describing continuations, modifying the RMI layer to allow for timeout-based operation, modifying the EJB containers to implement lease-based resource allocation, and integrating non-transactional state stores like DStore and SSM. A first step in this direction is described in [6].

We are focusing initially on applications whose workloads can be characterized as relatively short-running tasks that frame state updates. Substantially all Internet services fit this description—in part because the evolution of the tools for building such systems forced designers into the "three-tier application" mold. As enterprise services and applications (e.g., workflow, customer management) become web-enabled, they adopt similar architectures. We expect there are many applications outside this domain, such as interactive desktop applications, that could not easily be cast this way, and for which deriving a crash-only design would be impractical or infeasible. We also restricted the domain of Internet systems to those interacting based on HTTP, although Internet services might use additional protocols.

The restart/retry architecture has "execute at least once" semantics; in order to be highly available and correct, most requests it serves must be idempotent. This requirement might be inappropriate for some applications. Our proposal does not handle Byzantine failures or data errors, but such behavior can be turned into fail-stop behavior using well-known orthogonal mechanisms, such as triple modular redundancy [15] or clever state replication [7].

In today's Internet systems, fast recovery is obtained by overprovisioning and using rapid failure detection to trigger failover. Such failover can sometimes mask hours-long recovery times. Crash-only software is complementary to this approach and can help alleviate some of the complex and expensive management requirements for highly redundant hardware, because faster recovering software means less redundancy is required. In addition, a crash-only system can reintegrate recovered components faster, as well as better accomodate removed, added, or upgraded components [2].

We expect throughput to suffer in crash-only systems, but we consider this concern secondary to the high availability

and robustness we expect in exchange. Although much of the software industry still cares about the last ounce of performance, dependability started gaining ground on development agendas in the past few years.

7. Conclusion

By using crash-only designs, we expect to obtain better reliability and higher availability in Internet systems. By using an externally-enforced *stop=crash, start=recover* approach, fault models that applications need to account for can be simplified, thus encouraging simpler recovery routines which have higher chances of being correct. Writing crash-only components may be harder, but their simple failure behavior can make the assembly of such components into large systems easier.

Once we surround a crash-only system with a suitable infrastructure, we obtain a recursively restartable system [5]. Transparent recovery based on component-level micro-reboots enables restart/retry architectures to hide intra-system failure from the end users, thus improving the perceived reliability of the service. We find it encouraging that our initial prototype [6] was able to complete 78% more client requests under faultload than a non-crash-only version of the system that did not employ micro-reboots for recovery.

References

[1] T. Adams, R. Igou, R. Silliman, A. M. Neela, and E. Rocco. Sustainable infrastructures: How IT services can address the realities of unplanned downtime. Research Brief 97843a, Gartner Research, May 2001. Strategy, Trends & Tactics Series.

[2] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: How to upgrade distributed systems. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, 2003.

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, 2001.

[4] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive micro-reboots: A soft-state system case study. *Performance Evaluation Journal*, Summer 2003. To appear.

[5] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, 2001.

[6] G. Candea, P. Keyani, E. Kiciman, S. Zhang, and A. Fox. JAGR: An autonomous self-recovering application server. In *Proc. 5th International Workshop on Active Middleware Services*, Seattle, WA, June 2003.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, LA, 1999.

[8] P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[9] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. 18th ACM Symposium on Operating Systems Principles*, Lake Louise, Canada, 2001.

[10] G. Czajkowski and L. Daynés. Multitasking without compromise: A virtual machine evolution. In *Proc. Conference on Object Oriented Programming Systems Languages and Applications*, Tampa Bay, FL, 2001.

[11] R. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proc. 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1991.

[12] S. Garg, A. V. Moorsel, K. Vaidyanathan, and K. S. Trivedi. A methodology for detection and estimation of software aging. In *Proc. 9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998.

[13] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, AZ, 1989.

[14] J. Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, 1986.

[15] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, San Francisco, CA, 1993.

[16] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000.

[17] A. C. Huang and A. Fox. Decoupled storage: State with stateless-like properties. Submitted to the 22nd Symposium on Reliable Distributed Systems, 2003.

[18] Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. 25th International Symposium on Fault-Tolerant Computing*, Pasadena, CA, 1995.

[19] T. Lahiri, A. Ganesh, R. Weiss, and A. Joshi. Fast-Start: Quick fault recovery in Oracle. In *Proc. ACM International Conference on Management of Data*, Santa Barbara, CA, 2001.

[20] B. Ling and A. Fox. A self-tuning, self-protecting, self-healing session state management layer. In *Proc. 5th International Workshop on Active Middleware Services*, Seattle, WA, 2003.

[21] B. Murphy and N. Davies. System reliability and availability drivers of Tru64 UNIX. In *Proc. 29th International Symposium on Fault-Tolerant Computing*, Madison, WI, 1999. Tutorial.

[22] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International*, 11:341–353, 1995.

[23] K. Nagaraja, R. Bianchini, R. P. Martin, and T. D. Nguyen. Using fault model enforcement to improve availability. In *Proc. 2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, 2002.

[24] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the 1999 Summer USENIX Technical Conference*, Monterey, CA, June 1999.

[25] A. Pal. Personal communication. Yahoo!, Inc., 2002.

[26] P. Soper. Application isolation api specification. Java Specification Request No. 121, `http://jcp.org/en/jsr/detail?id=121`, 2002.

[27] M. Stonebraker. The design of the Postgres storage system. In *Proc. 13th Conference on Very Large Databases*, Brighton, England, 1987.

[28] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.