Embracing Failure: A Case for Recovery-Oriented Computing (ROC)

Aaron B. Brown and David A. Patterson
Computer Science Division, University of California at Berkeley
{abrown,pattrsn}@cs.berkeley.edu

Abstract

Motivated by the lack of availability demonstrated by current approaches to building servers for the Internet environment, we argue for a new approach to building highly-available systems that better reflects the realities of the modern server environment, namely that failures of hardware, software, and humans are inevitable. Our approach, denoted recovery-oriented computing (ROC)*, recognizes the inevitability of unanticipated failure and thus emphasizes recovery and repair rather than simple fault-tolerance. We define the properties that a ROC system must provide, and briefly consider how they might be achieved.*

1 The importance of availability and why it is lacking

Availability is the most important metric for modern computer systems. It is the lifeblood of corporate survival for e-commerce and e-business enterprises, and it is particularly crucial to those built around delivering services directly to the customer over the Internet. In the words of Kal Raman, CEO of drugstore.com, "availability is as important as breathing in and out [is] to human beings" [4].

High availability used to be a solved problem in the transaction-processing community. The solution was simple: buy an expensive fault-tolerant mainframe from a company like Tandem or IBM, deploy the vendor-supplied high-availability database system, and hide the whole thing in the back office behind a firewall of dumb terminals and human agents. With such a well-designed, stable, and controlled environment, high availability was straightforward to engineer. While some applications can still use this approach, today's most interesting applications do not. The computing world has shifted toward a much more distributed, heterogeneous environment, where transaction processing functionality and data access are exposed directly to customers through a complicated, heterogeneous conglomeration of interconnected systems—databases, application servers, middleware, and web servers—constructed from a multi-vendor mix of off-the-shelf hardware and software. In these systems, perceived availability is defined by the weakest link in the system, and so it is not enough to simply have a robust TP backend.

In this kind of environment, the environment of e-commerce and Internet business, availability is not being delivered as promised. Despite high-profile advertising by all major vendors purporting to offer high-availability solutions, and despite entire communities of researchers focusing on topics such as reliability, maintainability, and fault tolerance, outages and failures remain frequent. A recent survey by InternetWeek revealed that 65% of surveyed sites suffered a customer-visible outage at least once in the previous 6-month period; 25% reported three or more outages during that period [15].

What's wrong with this picture? Why does delivered availability fall so short of what is promised? We claim that the answer is simple: there is a fundamental mismatch between traditional high-availability approaches—fault-tolerant hardware, careful software testing, vendor-supplied technicians—and the realities of modern heterogeneous, distributed server environments, like those backing e-commerce and e-business sites. This mismatch goes as deep as the key assumptions underlying traditional high-availability design: that hardware and software can be built to have negligible failure rates, that failure modes of systems can be predicted and tolerated, and that maintenance and repair are error-free procedures.

As we will show in the next section of this paper, none of these assumptions hold true for modern network service environments. If we want to address availability in these environments, then, we must sub-

scribe to a new set of assumptions and accept the inevitability of unpredictable failures in hardware, software, and human operators. Once we do this, we recognize that, fundamentally, the only way to increase availability is to embrace failure rather than fear it, and build systems with a mentality of failure recovery and repair rather than failure avoidance. We call this approach *Recovery-Oriented Computing*, or *ROC* for short.

2 Failures are inevitable

There are several factors contributing to the inevitability of failures in modern Internet-connected transaction processing environments, but most important are the pressures of rapid innovation and cost reduction, and a fundamental ignorance of the behavior of human operators during maintenance procedures. In this section, we will show how these factors have lead to a method of system design where hardware and software failures are inevitable, where human-induced failures are more likely than not, and where the complexity and coupling of systems has made even the most unlikely failures a normal part of life.

2.1 Hardware and software failures are inevitable

Traditional high-availability systems have been designed to reduce the rate of visible hardware and software failures to nearly zero. While this may be possible in the stable, highly-constrained, cost-insensitive environment of traditional fault-tolerant systems, it is unrealistic given the way modern server systems are constructed.

The world of Internet service delivery is one where functionality is king. Functionality changes weekly as companies compete in an arms race of features; companies operate "in the gold rush to get their applications online first" [10]. As functionality changes, so does software, and thus the traditional high-availability design techniques of careful software engineering and extensive software testing go out the window [6]. Major internet portals are deploying code written by gumshoe engineers with little more than a week of job experience [2]. In the words of Debra Chrapraty, former CIO of E*Trade, a major online brokerage service, "We used to have six months of development on a product and three months of testing. We don't live that way any more. . . . In Internet time, people get sloppy" [10]. When people are sloppy, software bugs are the norm, and so software-induced failures are inevitable.

When we turn our attention to hardware, the situation does not look any better. Companies deploying online services often operate on razor-thin profit margins (and in some cases no profits at all). They cannot afford to spend millions of dollars on high-quality, fault-tolerant hardware when equivalent functionality (although not fault-tolerance) is available at a fraction of the cost from commodity, off-the-shelf PC-based hardware like the ubiquitous 1U PC. Unfortunately, this commodity hardware is failure-prone. The most basic fault-detection and repair mechanisms are often omitted for cost reasons—many PC motherboards are not even available with ECC memory. Less expensive desktop-oriented components (like IDE disks) are often used. These cost-cutting measures result in non-trivial hardware failure rates. For example, the production cluster run by the Google search engine experiences a node failure rate of 2-3% per year, with one-third of those failures attributable to the failure of a DRAM or memory bus, problems that could have been avoided had Google built their systems with ECC memory [7].

The Google example raises another important point: that of scale. While the memory failure described above only occurs in about 1% of the cluster over the course of a year—an almost negligible failure rate for a single system—the Google cluster has approximately 8000 nodes. On such a scale, that 1% failure rate translates to more than one node failure per week. This is a general problem: as the scale of the infrastructure behind Internet-delivered services increases, the typically-ignored failure modes in the tails of the failure distribution will become more frequent.

2.2 Human failures are inevitable

All large systems rely on human beings for maintenance and repair. At the very least, humans must perform the physical actions of repairing, replacing, and expanding hardware. Most systems require human intervention for software configuration and upgrading, and many require human intervention in the performance tuning loop. The task of diagnosing and fixing failures and other aberrant behavior is also a standard task of the human administrator.

However, as we all know, humans make mistakes. Psychologists have shown that human error rates are unquestionably non-zero, and can rise to between 10% and 100% in stressful situations (as one might expect during off-hours emergency system maintenance) [13]. Humans make mistakes even in simple tasks such as digit recognition [8]; we can hardly expect them to do better when dealing with unwieldy maintenance interfaces to complex transaction processing systems.

Unfortunately, modern system designs do not take into account the possibility of human error. Traditional high-end fault-tolerant systems have a partial solution in that their vendors lock up their systems and give the keys only to certified, trained service personnel. But even a highly-trained operator will inevitably make mistakes, so this is hardly a complete solution. Furthermore, it is a solution that does not apply to modern Internet service environments, where systems consist of collections of hardware and software from different vendors deployed in highly varied configurations.

Thus all the conditions are right for human-induced system failures, and field data collected over the past several decades bears out the claim that these failures are inevitable. Data from the late 1970s reveals that operator error accounted for 50-70% of failures in electronic systems, 20-53% of missile system failures, and 60-70% of aircraft failures [3]. In the mid-1980s, a study of failures in fault-tolerant Tandem systems revealed that 42% were due to system administration errors—again human error [5]. Data collected on the causes of failures in VAX systems reveals that in 1993, human operators were responsible for more than 50% of failures, and that the error rate was rising as hardware and software failures become less important [11]. A study of the telephone network (arguably one of the most fault-tolerant systems in deployed use today) between 1992 and 1994 reveals that human errors resulted in 52% of non-overload outages and were responsible for 50% of non-overload outage minutes, with about half of those due to errors made by telco personnel performing maintenance [9]. And more recently, in 1999, a Vice President at Oracle was reported as claiming that one "can have the best technology on the planet, but half the failures we analyze are human error" [10].

2.3 Unanticipated failures are inevitable

Although we have shown that hardware, software, and human failures are inevitable in modern network service environments, one could still argue that with enough care, these failures could be anticipated and avoided through fault-tolerance techniques. We have already explained why this kind of approach is unlikely to succeed in the fast-paced, cost-conscious environment of Internet service delivery, but there is a further argument to make that even with the best fault-tolerance techniques, unanticipated failures will still sneak through and affect the system. This is most easily seen in the case of human error: humans are notoriously good at finding ways to break systems, and often ignore warnings and error messages when they do not match the human's own mental model of how the system should be operating [16].

More generally, we can turn to the sociological theory of risk analysis to discover that even simple hardware and software failures will occur in practice in unanticipated combinations. Large servers are complex, reasonably-tightly-coupled systems that perform a transformational function, consuming user requests, transforming databases, and synthesizing new results, all under the guidance of human maintainers. In the system taxonomy defined by sociologist and risk investigator Charles Perrow, these are exactly the kind of system that is highly susceptible to unexpected interactions [12]. Perrow's theories predict that such systems are by their very nature subject to "normal accidents": accidents (outages or failures in the case of servers) that arise from multiple and unexpected hidden interactions of smaller failures and the recovery systems designed to handle those failures. When viewed individually, normal accidents appear as

very unlikely, rare situations arising from bizarre and improbable combinations of factors. Perrow's claim is that normal accidents (and therefore outages) are inevitable and unpredictable, despite the best attempts to model and compensate for failures.

3 Embracing failure: Recovery-Oriented Computing

> *"If a problem has no solution, it may not be a problem, but a fact, not to be solved, but to be coped with over time" — Shimon Peres [14]*

We claim that the lack of high-availability in Internet-delivered data and transaction-processing services is a result of focusing too much attention on avoiding failures rather than repairing them, on trying to increase mean-time-to-failure while ignoring mean-time-to-repair. From the evidence and discussion in the previous section, we can deduce that failures will always occur despite the best efforts of system design and modeling. Drawing on the wisdom in the quotation by Shimon Peres that starts this section, we thus conclude that further significant gains in availability can only come once failures are accepted as a normal, unavoidable fact of system operation, and systems are designed with fast and effective repair mechanisms that mitigate the long-term impact of those failures. We call this philosophy *Recovery-Oriented Computing (ROC)*.

At its heart, ROC addresses hardware, software, and human failures by providing rapid and effective mechanisms for detecting and recovering from them. Recovery can take many forms—from simple mechanisms like design-for-reboot [2], to more complex schemes such as fail-stop fault containment combined with data redundancy, to full regeneration of system state from backups, checkpoints, and logs. A full discussion is outside the scope of this paper. But in all cases, these mechanisms should be designed to make as few assumptions about failure characteristics as possible, and they should provide means to recover from unanticipated catastrophic failures that make it past any standard fault-tolerance lines of defense.

Furthermore, a recovery-oriented system design has to go beyond simply providing recovery mechanisms. After all, all of the major transaction-processing systems today provide some recovery mechanisms in the form of backups and transaction logging, and the simple existence of these mechanisms has not solved the availability problem. To truly have a recovery-oriented design, recovery mechanisms have to be treated as first-class parts of the system, and integrated into a framework that manages their use and guarantees their effectiveness.

This recovery-oriented framework needs to provide several guarantees. First, it must ensure that problems and failures are detected quickly so that they can be contained or repaired as soon as possible, and before they have propagated through the system. As part of this detection, a recovery-oriented system should strive to expose and repair latent errors in the system before they are activated; the kinds of "normal accidents" analyzed by Perrow and discussed above in Section 2.3 often occur only when many latent errors have accumulated in the system and are all activated simultaneously in a chain-reaction cascade of failures. Furthermore, recovery-oriented systems should provide assistance in diagnosing the root cause of problems once they have been detected, speeding repair of the correct system component.

In addition to fast failure detection, a ROC system must also guarantee that its repair mechanisms are trustworthy, whether they are fully automated or require human intervention; this is a particular problem today, as recovery code is difficult to test and thus is often untested or buggy. The solution is to periodically test recovery code *in situ* as part of normal system operation, allowing automated recovery mechanisms be exercised and verified in the production environment. When repair requires human intervention, those mechanisms should be exercised as well: the human operators should be subjected to realistic "fire-drill" simulations of failures and repair, allowing them to become familiarized with the system's failure modes, maintenance interfaces, and recovery procedures, all in the realistic context of the production environment. Such realistic, on-the-job operator training helps human operators calibrate their mental models of the system and allows them to make mistakes and learn from them, an essential part of gaining familiarity and confidence with system repair tasks.

The last key guarantee for a ROC system is that it must tolerate further errors and failures during recovery and repair. In the large-scale systems that are being built today, the statistical probability of double failures is becoming non-negligible. Furthermore, with human operators involved in recovery and repair procedures, human-induced failures during these procedures are inevitable.

Finally, note that recovery-oriented design is complimentary to traditional fault-tolerance. A ROC system can still incorporate redundant hardware and use well-tested software; these traditional mechanisms will simply reduce the reliance on the recovery mechanisms, while the recovery mechanisms will continue to provide a backstop for those unanticipated and human-induced failures that make it past the traditional fault-tolerance defenses.

3.1 Building ROC systems

How can we build recovery-oriented systems, especially in a transaction-processing environment? A full discussion is well outside the scope of this position paper, but we will identify a few important techniques. First, a ROC system inherently requires redundancy of hardware and data—not necessarily at the level of lockstep CPUs, but at least in the form of a clustered hardware design with replicated state. The system design must be partitionable to support fault containment and to provide the means of safely exercising recovery mechanisms; again, physically-partitioned designs such as clustered or shared-nothing organizations seem appropriate. To achieve the goal of quickly detecting failures, ROC systems should be built to incorporate extensive self-testing and checking at the component and system-wide level. In addition to traditional assertion-checking techniques, system components should verify the proper behavior of dependent components by explicitly injecting realistic test inputs and checking the resulting outputs for both correctness and performance. To aid in diagnosis, ROC systems should automatically track the health of all components, and use techniques such as dependency analysis [1] to automatically pinpoint the root-cause of detected problems. Finally, to carry out the exercising of recovery mechanisms needed to guarantee their proper behavior, ROC systems should have integrated mechanisms for fault-injection to simulate failures and to trigger recovery mechanisms.

4 Conclusion

The transaction-processing world is different today than it used to be. With traditional back-office TP systems being supplanted by multi-tier, Internet-based service delivery, the assumptions behind system design have changed, and traditional techniques for providing high availability have lost much of their power. Since high availability is as important today as it has always been, we need new techniques for improving availability, ones based on the realities of modern server environments. Our recovery-oriented design provides such an approach: it recognizes the inevitability of failures that arises from Internet service environments and devotes system resources to repairing and recovering from failures rather than avoiding them. With failures being a fact of life in today's server environments, only by embracing them can we take new strides toward increasing availability.

References

[1] A. Brown, G. Kar, and A. Keller. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. To appear in *Proceedings of the Seventh IFIP/IEEE International Symposium on Integrated Network Management (IM VII)*, Seattle, WA, May 2001.

[2] G. Candea and A. Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. *Submission to the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*.

[3] J. M. Christensen and J. M. Howard. Field Experience in Maintenance. *Human Detection and Diagnosis of System Failures: Proceedings of the NATO Symposium on Human Detection and Diagnosis of System Failures*, J. Rasmussen and W. Rouse (Eds.). New York: Plenum Press, 1981, 111–133.

[4] S. Fisher. E-business redefines infrastructure needs. *InfoWorld*, 7 January 2000. Available from www.infoworld.com.

[5] J. Gray. Why Do Computers Stop and What Can Be Done About It? *Symposium on Reliability in Distributed Software and Database Systems*, 3–12, 1986.

[6] J. Hamilton. Fault Avoidance vs. Fault Tolerance: Testing Doesn't Scale. *High Performance Transaction Systems (HPTS) Workshop*, Asilomar, CA, 1999.

[7] J. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 3e (beta version)*. San Francisco: Morgan-Kauffmann, 2001.

[8] B. H. Kantowitz and R. D. Sorkin. *Human Factors: Understanding People-System Relationships*. New York: Wiley, 1983.

[9] D. R. Kuhn. Sources of Failure in the Public Switched Telephone Network. *IEEE Computer* 30(4), April 1997.

[10] J. Menn. Prevention of Online Crashes is No Easy Fix. *Los Angeles Times*, 2 December 1999, C-1.

[11] 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.

[12] C. Perrow. *Normal Accidents: Living with High-Risk Technologies*. Princeton, NJ: Princeton Press, 1999.

[13] R. H. Pope. Human Performance: What Improvement from Human Reliability Assessment. *Reliability Data Collection and Use in Risk and Availability Assessment: Proceedings of the 5th EureData Conference*, H.-J. Wingender (Ed.). Berlin: Springer-Verlag, April 1986, 455–465.

[14] D. Rumsfeld. Rumsfeld's Rules: Advice on government, business and life. *The Wall Street Journal Manager's Journal*, 29 January 2001.

[15] T. Sweeney. No Time for DOWNTIME—IT Managers feel the heat to prevent outages that can cost millions of dollars. *InternetWeek*, n. 807, 3 April 2000.

[16] C.D. Wickens and C. Kessel. Failure Detection in Dynamic Systems. *Human Detection and Diagnosis of System Failures: Proceedings of the NATO Symposium on Human Detection and Diagnosis of System Failures*, J. Rasmussen and W. Rouse (Eds.). New York: Plenum Press, 1981, 155–169.