Big iron lessons, Part 2: Reliability and availability: What's the difference?

Reliability, availability, and serviceability design considerations for the system architect

Sam Siewert (Sam.Siewert@Colorado.edu), Adjunct Professor, University of Colorado

Summary: How do you design a computing system to provide continuous service and to ensure that any failures interrupting service do not result in customer safety issues or loss of customers due to dissatisfaction? Historically, system architects have taken two approaches to answer this question: building highly reliable, fail-safe systems with low probability of failure, or building mostly reliable systems with quick automated recovery. The RAS (Reliability, Availability, Serviceability) concept for system design integrates concepts of design for reliability and for availability along with methods to quickly service systems that can't be recovered automatically. This approach is fundamental to systems where the concern is quality of service, customer retention, and due diligence for customer safety.

Date: 08 Mar 2005 (Published 08 Mar 2005)
Level: Introductory
Activity: 453 views
Comments: 0 (Add comments)


Average rating (based on 18 votes)

From the end-user standpoint, computing systems provide services, and any outage in that service can mean lost revenue, lost customers due to dissatisfaction, and, in extreme cases, loss of life and possible legal repercussions. For example, with cell phone services, if it's only in rare cases that I can't get a signal or make a connection, I'll stick with my service provider. But if this occurs too often or in critical locations like my office or home, I'll most likely switch providers. The end result is loss of revenue and loss of a customer. Embedded systems not only provide value added services such as communications, but they also provide critical services for human safety. For example, my anti-lock braking system is provided by a digital control service activated by my ignition. My expectation is that this service will work without failure once ignition is completed. Any system fault that might interrupt service should prevent me from using the vehicle before I start to drive. A failure during operation could result in loss of life and product liability issues.

Ideally, service outages would not be an issue at all, but experienced system architects know they must analyze, predict, and design systems for handling failure modes in advance. For safety-critical systems, this is the due diligence required to avoid product liability nightmares. Historically, system architects have taken two approaches to this problem: building highly reliable, fail-safe systems with low probability of failure, or building mostly reliable systems with quick automated recovery. Both approaches are judged with probability measures. Reliability is evaluated by mean time to failure, and availability is measured by service uptime over a year of continuous operation. The RAS (Reliability, Availability, Serviceability) concept for system design integrates concepts of design for reliability and for availability along with methods to quickly service failures that can't be designed for automatic recovery.

Building systems for very high reliability can be cost prohibitive, so RAS offers an approach to balance reliability with recovery and servicing features to control cost and ensure safety and quality of service.

This approach is fundamental to systems where the concern is affordable quality of service, customer retention, and due diligence for customer safety.

You can learn a lot from the experience built into systems like the IBM® mainframes that have evolved from a rich heritage of design for reliability, availability, and serviceability. This article is the second in the *Big iron lessons* series, which explores elements of IBM mainframe architecture to assist those developing new architectures by examining the design decisions made in the big iron mainframes. This article gives background on the evolution of RAS features developed for IBM mainframes and summarizes significant design decisions.

To best understand the evolution of RAS in IBM mainframe architecture, it is useful to step back in time to 1964 and examine RAS features in the IBM System/360™ (see Resources) and consider how architects have balanced the issues of cost, reliability, safety, availability, and servicing, and improved upon this over time. Early systems were most often centralized rather than in the hands of end users, and may have been less cost-sensitive than today's mainframes, but the concepts of availability and reliability emerged early and have evolved over time into the well-proven RAS features now found in the z990.

Often, system and application requirements will determine if availability is stressed over reliability or vice versa. For example, the concept of availability has also been fundamental to telecommunication systems, where most often quality of service is more of an issue than safety. In contrast, reliability has been fundamental to systems such as commercial flight control systems where failure means significant loss of life and assets. The balance of availability and reliability features should fit the system -- building FAA (Federal Aviation Administration) levels of reliability into cell phones would make the system much less affordable and therefore not usable by many customers. Likewise, safety-critical systems can't simply quote uptime to convince customers that the systems are not too risky to trust -- fail-safe operation, reliable parts, triple redundancy, and the extra cost that goes along with these design features is expected and will be paid for to mitigate risk.

The difference between availability and reliability

Availability is simply defined as the percentage of time over a well-defined period that a system or service is available for users. So, for example, if a system is said to have 99.999%, or *five nines,* availability, this system must not be unavailable more than five minutes over the course of a year. Quick recovery and restoration of service after a fault greatly increases availability. The quicker the recovery, the more often the system or service can go down and still meet the five nines criteria. Five nines is often called *high availability,* or HA.

In contrast, high reliability (HR) is perhaps best described by the old adage that a chain is only as strong as its weakest link. Building a system from components that have very low probability of failure leads to maximal system reliability. The overall expected system reliability simply is the product of all subsystem reliabilities, and the subsystem reliability is a product of all component reliabilities. Based upon this mathematical fact, components are required to have very low probability of failure if the subsystems and system are to also have reasonably low probability of failure. For example, a system composed of 10 components, each with 99.999% reliability, is $(0.99999)^{10}$, or 99.99%, reliable. Any decrease in the reliability of a single component in this type of single-string design can greatly reduce overall reliability -- for example adding just one 95% reliable component would drop the overall reliability to 94.99%.

The z990 includes features that allow it to be serviced and upgraded without service interruption. The

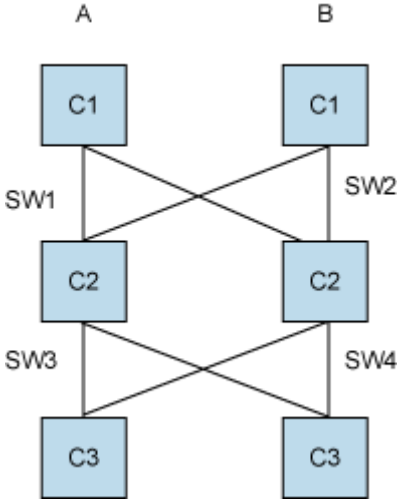
subsystem level of redundancy is the "book," which is an independently powered, multi-chip module, with cooling, memory cards, and IO cards. There are also redundant components within a book including CPU spares (2 per book) and redundant interconnection to level-2 cache. Furthermore, the z990 provides jumpering to preserve the interconnection of MP (multi-processor) books while other books are being serviced or replaced in case of non-recoverable errors.

Service or system outages can be caused by routine servicing, upgrades, and failures on most traditional computing systems. Probability of a system or service outage on the z990 is limited to scenarios where failures are non-recoverable by switching redundant components or subsystems into operation. In most component or subsystem failure modes, the z990 is able to isolate the non-recoverable book or component and continue to operate with some performance degradation. Redundant books and components allow the z990 to operate without service interruption until the degraded book is replaced. As you will see later on in this article, designing interconnection networks for isolation and switching of modules like the z990 books is complex. While this complexity adds to cost, it does significantly decrease probability of service interruption. You can find more detailed information on processor book management in the "Reliability, availability, and serviceability (RAS) of the IBM eServer z990" paper (see Resources).

How high reliability helps

It is theoretically possible to build a system with low-quality, not-so-reliable components and subsystems, and still achieve HA. This type of system would have to include massive redundancy and complex switching logic to isolate frequently failing components and to bring spares online very quickly in place of those components that failed to prevent interruption to service. Most often, it is better to strike a balance and invest in more reliable components to minimize the interconnection and switching requirements. If you take a very simple example of a system designed with redundant components that can be isolated or activated, it becomes clear that the interconnection and switching logic does not scale well to high levels of redundancy and sparing. Consider the simple, single-spare dual-string system shown in Figure 1.

Figure 1. Dual-string, cross-strapped subsystem interconnection



The example system in Figure 1 has eight possible configurations that can be formed by activating and

isolating components C1, C2, or C3 from side A or side B. The system must be able to positively detect failed components and track these failures to reconfigure with an operable switch state and new interconnection of activated components. Table 1 describes the eight possible configurations for this small-scale HA system example.

Table 1. Enumeration of configurations for three subsystem dual-string cross-strapped system

Configuration	Component C1	Component C2	Component C3
1	A	A	A
2	A	A	B
3	A	B	A
4	A	B	B
5	B	A	A
6	B	A	B
7	B	B	A
8	B	B	B

From the simple example described by Figure 1, you can see that a trade-off can be made between the complexity of interconnecting components and redundancy management with the cost of including highly reliable components. The cost of hardware components with high reliability is fairly well known and can be estimated based upon component testing, expected failure rates, MTBF (mean time between failures), operational characteristics, packaging, and the overall physical features of the component.

System architects should also consider three simple parameters before investing heavily in HA or HR for a system component or subsystem:

- Likelihood of unit failure
- Impact of failure on the system
- Cost of recovery versus cost of fail-safe isolation

How cost and safety factor in

HA design may not always ensure that a design will be safe. Much depends on how long service outages will be during recovery scenarios. For safety-critical systems such as flight control or anti-lock braking, it is possible that even very brief outages could lead to loss of system stability and total system failure. Thus, for safety-critical systems, the balance between HA and HR must often favor HR in order to avoid risky recovery scenarios.

Achieving HA/HR with redundancy as the primary method

One approach to HA is to use redundancy and switching to not only increase availability, but so lower reliability (and most often lower cost) components can be used, which for some systems yields overall

target HA at the lowest overall system cost. The best example of this approach is RAID (Redundant Array of Inexpensive Disks); see Resources for examples and links. In fact, numerous RAID configurations have been designed which make trade-offs between HA/HR and serviceability by using larger numbers of disk drives of various types including SCSI, Fiber Channel, Serial-Attached SCSI, and Serial-Attached ATA drives. RAID also can provide improved storage performance by striping writes/reads over multiple drives as well as using drives for parity or more sophisticated error encoding methods. Typical RAID systems with volume protection allow for a drive failure and replacement with automatic recovery of the volume and no downtime given a single drive failure. Protection from double faults or failures while a RAID system is recovering has become of interest more recently and has led to development of RAID 6. One of the more interesting aspects of the RAID approach is that it not only relies upon specialized HA hardware, but on fairly complex software.

RAS designs should span hardware, firmware, and software layers

Perhaps much harder to estimate is the cost of highly reliable software. Clearly, reliable hardware running unreliable software will result in failure modes that are likely to cause service interruption. It is well accepted that complex software is often less reliable, and that the best way to increase reliability is with testing. Testing takes time and ultimately adds to cost and time to market.

Early on, system architects focused on designing HA and HR hardware with firmware to manage redundancy and to automate recovery. So, for example, firmware would reconfigure the components in the example in Table 1 to recover from a component failure. Traditionally, rigorous testing and verification have ensured that firmware has no flaws, but history has shown that defects can still wind up in the field and emerge due to subtle differences in timing or execution of data-driven algorithms.

High reliability software often comes at a high cost

Designing firmware and software for HR can be costly. The FAA requires rigorous documentation and testing to ensure that flight software on commercial aircraft is highly reliable. The DO-178B class A standard requires software developers to maintain copious design, test, and process documentation. Furthermore, testing must include formal proof that code has been well tested with criteria such as multiple condition decision coverage (MCDC). This criteria ensures that all paths and all statements in the code have been exercised and shown to work. It is very laborious and therefore greatly increases the cost of software components.

A number of different operating systems can run can run on the z990, but for those interested in RAS, z/OS® (see Resources) is perhaps most interesting, since it has been designed for HA and verified for HR. Furthermore, z/OS has been designed to take full advantage of the z990 hardware features for HA. Based upon success in IBM mainframe HA design, IBM engineers realized that it was important to put equal effort into the firmware and software to achieve the highest levels of RAS. The FEDC (First Error Data Capture) system integrated into z/OS is a great example of coordinated hardware/firmware design for RAS serviceability. The designers of the FEDC feature on the z990 for z/OS ("The z990 First error data capture concept;" see Resources) note that, "due to the increased reliability of the hardware, firmware failures represented an increasingly larger percentage of overall system failures in the field." The FEDC provides detailed trace and state data for users and for IBM through a call home service, so that when automatic recovery occurs (or, in rare cases, when it is unsuccessful), human-assisted recovery is accelerated by very precise failure data.

Cost trade-offs in the hardware layer

Designing for HR alone can be cost prohibitive, so most often a balance of design for HA and HR is better. HA at the hardware level is most often achieved through redundancy (sparing) and switching, which is the case for the z990 and has been fundamental to IBM mainframe design since the System/360. A trade-off is made between the cost of duplication and simply engineering higher reliability into components to reduce the MTBF. Over time, hardware designers have found balance between HR and HA features to optimize cost and availability. Fundamental to duplication schemes is the recovery latency. For example, the z990 has a dynamic CPU sparing (DCS) feature that can cover a failure so that firmware is unaffected by reconfiguration to isolate the errant CPU and to switch in the spare.

When considering component or subsystem duplication for HA, architects must carefully consider the complexity and latency of the recovery scheme and how this will affect firmware and software layers. Trade-offs between working to simply increase reliability instead of increasing availability through sparing should be analyzed. A simple well-proven methodology that is often employed by systems engineers is to consider the trade-off of probability of failure, impact of failure, and cost to mitigate impact or reduce likelihood of failure. This method is most often referred to as FMEA (Failure Modes and Effects Analysis); see Resources. Another, less formal process that system engineers often use is referred to as the "low hanging fruit" process. This process simply involves ranking system design features under consideration by cost to implement, reliability improvement, availability improvement, and complexity. The point of low hanging fruit analysis is to pick features that improve HA/HR the most for least cost and with least risk. Without existing products and field testing, the hardest part of FMEA or low hanging fruit analysis is estimating the probability of failure for components and estimating improvement to HA/HR for specific features. For hardware, the tried and true method for estimating reliability is based upon component testing, system testing, environmental testing, accelerated testing, and field testing. The trade-offs between engineering reliability and availability into hardware are fairly obvious, but how does this work with firmware and software?

Cost trade-offs in the firmware and software layers

Designing and implementing HR firmware and software can be very costly. The main approach for ensuring that firmware/software is highly reliable is verification with formal coverage criteria along with unit tests, integration tests, system tests, and regression testing. Test coverage criteria include feature points; but for HR systems, much more rigorous criteria are necessary, including statement, path, and, in extreme cases, multiple condition decision coverage (MCDC). The IBM z/OS testing has followed this rigorous approach to ensure HR. (This testing is described in the article "Testing z/OS; see Resources.) The FEDC system in z/OS provides support for HA, recognizing that despite rigorous testing, software/firmware testing cost and time spent must be limited at some point, and the design should include support for quick operator assisted recovery. Finally, the FEDC is also very useful for application developers who most likely would also like to strike a balance between rigorous testing of their application and provision for recovery.

Why is MCDC required for full coverage in software testing?

One might wonder why simply designing test cases for path and statement coverage is not sufficient for HR software. The simple snippet of C code in Listing 1 shows why MCDC is required (also see the sidefile for code without line numbers). The `if` test in `main()` has two expressions logically ordered together. Most C compilers will generate code such that the second expression is short-circuited if the first evaluates to true. As a result, both paths in `main()` for the `if` blocks can be driven by a test without

ever executing (testing) the `OutsideLimits` code. So, a test driver must drive the same path with the condition where `recoveryRequired` is false and where `recoveryRequired` is true and `OutsideLimits` is either true or false in combination with this. So, for simple path coverage in main there are only two paths noted by coverage of code on the line numbers: Path-A) 28, 30, 32 and Path-B) 28, 30, 13, 20, 36. However, by MCDC you must ensure that main Path-A is covered in combination with the paths of the function `OutsideLimits`, which defines Path-C) 28, 30, 13, 15, 16, 32.

Listing 1. Simple C code with two paths and MCDC testing requirements

```
01: #define UPPER_LIMIT 100
02: #define TRUE 1
03: #define FALSE 0
04:
05: extern void logMessage(char *msg);
06: extern unsigned int MMIORead(unsigned int addr);
07: extern void StartRecovery(void);
08: extern void ContinueOperation(void);
09: extern int RecoveryRequired;
10:
11: int LimitTest(unsigned int val)
12: {
13:     if(val > UPPER_LIMIT)
14:     {
15:         logMessage("Limit Exceeded\n");
16:         return TRUE;
17:     }
18:     else
19:     {
20:         return FALSE;
21:     }
22: }
23:
24: main()
25: {
26:     unsigned int IOValue = 0;
27:
28:     IOValue = MMIORead(0xF0000100);
29:
30:     if(RecoveryRequired || OutsideLimits(IOValue))
31:     {
32:         StartRecovery();
33:     }
34:     else
35:     {
36:         ContinueOperation();
37:     }
38: }
```

Component-level error detection and correction

Ideally, all system, subsystem, and component errors can be detected and corrected in a hierarchy so that component errors are detected and corrected without any action required by the containing subsystem. This hierarchical approach for fault detection and fault protection/correction can greatly simplify verification of a RAS design. The z990 ECC memory component provides for single-bit error detection and automatic correction. The incorporation of ECC memory provides a component level of RAS, which

can increase RAS performance and reduce the complexity of supporting RAS at higher levels.

Redundancy management at the subsystem level

The z990 includes a number of redundancy management features that provide online replacement/upgrade, automatic recovery with spares, and continuous operation in degraded modes for double and triple faults that require servicing. The z990 organizes processing, memory, and I/O resources into logical units called *books,* which are interconnected so that they can be switched into or out of operation without interruption to the overall system. This scheme includes the ability to add and activate processors, memory, and I/O connections while the system continues to run. The fault-tolerant book interconnection and cross-book CPU sparing provides excellent automatic recovery as well for most fault scenarios.

Finally, support subsystems such as cooling also include redundancy so that the system is not endangered by thermal control system faults. Redundancy and management of that redundancy with automated fault detection, isolation, and automatic recovery is fundamental to the z990 RAS design.

Fail-safe design

HR systems often include design elements that ensure that non-recoverable failures result in the system going out of service, along with safing to reduce risk of losing the asset, damaging property, or causing loss of life. The z990 may or may not be used in applications that are considered safety critical -- arguably, even a database error could result in significant loss of assets (for stock market applications) or even loss of life (in certain health care applications). The z990 does incorporate fail-safe modes when recovery is impossible or too costly to incorporate (for example, double processor faults in a single book) and the likelihood of a failure is low. In the case of double or triple faults, the z990 isolates the failed subsystem (a book) and requires operator assistance -- for most users. This is likely a good cost-versus-HA/HR trade-off, given the built-in support for serviceability in the z990 (on-line replacement).

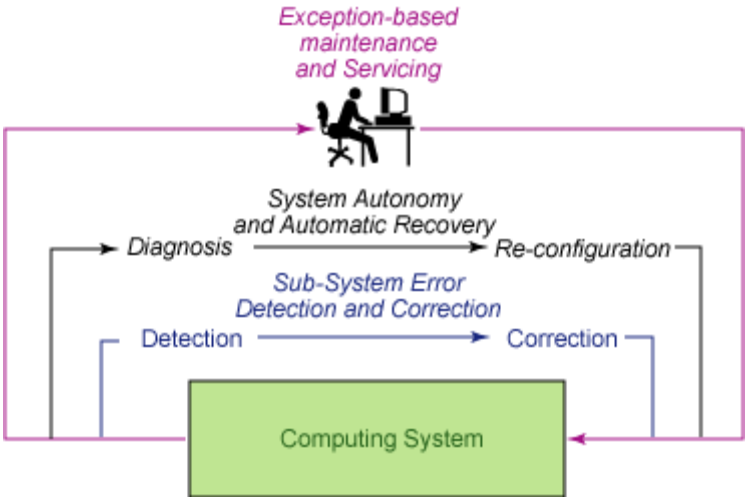
Serviceability concepts

The z990 strikes a nice balance between HA, HR, system safety (permanent loss of data could have high related risk), and simplicity of operation and servicing. In most cases, this tracking of errors, data logging and upload to IBM with RETAIN and configuration tracking of FRUs (field-replaceable units) simplifies service calls, sometimes (or often) allowing a technician to handle a non-recoverable failure or upgrade without causing service interruption. The z990 organization of books as field-replaceable units with support to assist the operator in recovery greatly increases the z990 serviceability for faults that occur despite the HR/HA design.

Recovery concepts

Conceptually, architects should consider how levels of recovery are handled with varying degrees of automation, as depicted in Figure 2.

Figure 2. Supporting multiple levels of recovery autonomy



The z990 approach includes all levels of recovery automation and management between fully automatic, operator assisted, and fully manual. Manual recovery requirements are minimized, and manual recovery still includes isolation features so that FRUs can be replaced easily to restore full performance without service interruption and without impact to other processor books.

Putting it all together

The z990 is an excellent example of good RAS design because RAS is considered at all levels of hardware, from components and subsystems to the system level, and because firmware/software design for RAS is considered in addition to hardware. Perhaps most important, though, is the fact that the z990 RAS design spans all of these levels and layers so that overall RAS is a system feature. The well-integrated RAS features of the z990 no doubt increase the cost of the system considerably. However, the z990 provides customers with an HA/HR computing platform with low risk of losses due to downtime. The cost trade-off will vary for each system design based upon the risk and cost associated with downtime compared to the cost of decreasing the overall probability of downtime. The system architect has to find the right balance of cost, complexity, recovery automation, reliability, and time to market for each, given the system's intended usage. There is no simple formula for HA/HR analysis, but one place to start is with careful consideration of what the risk and cost is of being out of service -- if this can be measured in dollars lost per hour or day, potential loss of life, or loss of customers, this is a good starting point. If my business will potentially lose thousands of dollars per hour while my enterprise system is out of service, then I will be more willing to pay much more for HA/HR.

Is autonomic architecture the future for RAS design?

Strategies for RAS like the z990 have been refined and improved significantly since the concepts of continuous availability were introduced in early mainframes like the System/360. Clearly, RAS requires balance between safety and uninterrupted service on the one hand and the cost to provide these features on the other. The cost of additional RAS performance can be high, and this cost must be balanced with the risk and cost of occasional service failures and safety. How much is enough? How can the additional cost of better RAS be financed?

One concept is that systems that require little to no monitoring are not only more cost efficient, but necessary, if automation is to scale up significantly beyond present day systems. I served as a consultant once for a company that was considering the launch and operation of hundreds of satellites for a low earth orbit global coverage data communications system. Presently, such satellites require a half-dozen operators apiece to monitor and control them. In our proposed system, this would have scaled to approximately 5,000 operators, each with an engineering degree and a salary of US$100,000 a year, plus benefits -- a total of US$500 million per year for operations alone. Clearly, for emergent applications like this, much more significant RAS and autonomous operation is needed, and would warrant significant investment in unprecedented levels of RAS and system operational autonomy. This global datacom system was never deployed; however, the scale of many emergent enterprise systems rivals the proposed system in terms of operational complexity.

Enterprise systems include processing, storage, and I/O, with thousands of interconnections, hundreds of processors, and terabytes of data. Quick access to widely distributed information for decision support, global operations, and commerce is required, and the volume and speed of information flow is scaling beyond the point where traditional monitoring and service methods can be applied. Autonomic architecture is an alternative to traditional systems administration that uses RAS as a starting point to further reduce the human attention required to maintain enterprise systems. A full description of autonomic architecture is beyond the scope of this review of RAS strategies; however, you can find more information in the Resources section below.

Resources

- See the C source code referenced in Listing 1.
- Part 1 of this series, "FPU architecture, now and then" gives an overview of two floating point formats used in the z990 architecture and discusses key FPU issues that system architects should consider in new designs (developerWorks, February 2005).
- Reliability, Availability, and Serviceability (RAS) of the IBM eServer z990 provides an overview of IBM's current state-of-the-art RAS strategy for scalable mainframes.
- RAS Design for the IBM eServer z900 describes RAS strategies further refined in the z990.
- Even earlier, RAS concepts were being developed for IBM mainframes. The RAS Strategy for IBM S/390 G5 and G6 provides a good early picture of the IBM mainframe RAS strategy that has evolved into the current z990 strategy.
- Finally, the concept of designing for availability in IBM servers can be found as early as 1964. The Structure of System/360: Part V - Multisystem Organization describes how the System/360 incorporated numerous features that today would qualify as RAS features.
- Fundamental to the z990 Serviceability design is The z990 First Error Data Capture Concept, which provides diagnostic data crucial to exception maintenance for Serviceability in the z990.
- This RAID tutorial provides an excellent introduction to understanding RAID levels and concepts. RAID emphasizes HA over reliability and in fact touts the ability to use low cost (and therefore less reliable) disks in the system while still achieving high overall system availability and reliability.

- High reliability in software/hardware systems comes at a high cost. The FAA DO-178B class A standard for software quality is complex and laborious. FAA reports on acceptable off-the-shelf software and verification procedures provide an idea of just how involved this level of reliability assurance can be. The FAA DO-178B class A standard is available for purchase from RTCA.

- Environmental and stress testing requires sophisticated facilities such as the Environmental Testing Facilities found at Ball Aerospace. The aerospace satellite industry has historically had some of the most rigorous system environmental testing requirements including: EMI/EMC (Electromagnetic Interference/Compatibility), vibration and shock, thermal vacuum cycling, and accelerated lifetime testing.

- Many of the large aerospace corporations lease testing facilities and provide consulting to amortize the cost of these facilities. For example, Boeing, like Ball Aerospace in Boulder Colorado, provides similar consulting services and access to their facilities by contract.

- This Web page on Failure Modes and Effects Analysis is a good resource for systems engineers who need to make decisions on how to handle trade-offs between cost, reliability, and availability.

- This NASA tutorial on Multiple Condition Decision Coverage, a testing requirement for FAA DO-178B class A certification, demonstrates the rigor involved in developing highly reliable software systems.

- An alternative to the high cost of high reliability is high availability. As long as brief service outages are acceptable, then an approach for rapid recovery and minimal downtime can be taken as described in this overview of MS Windows Server 2003.

- Testing z/OS describes the IBM approach for verification of the z/OS software to ensure high reliability similar to FAA standards.

- Beyond RAS, Autonomic Architecture provides goals for self-knowing, self-configuring, self-optimizing, self-healing, self-protecting (securing), and self-managing systems. Eight goals are outlined for the Autonomic architecture overall, and many of them relate to and significantly extend RAS strategies found in the z990.

- Read more about autonomic architecture at the IBM developerWorks Autonomic computing zone.

- Have experience you'd be willing to share with Power Architecture zone readers? Article submissions on all aspects of Power Architecture technology from authors inside and outside IBM are welcomed. Check out the Power Architecture author FAQ to learn more.

- Have a question or comment on this story, or on Power Architecture technology in general? Post it in the Power Architecture technical forum or send in a letter to the editors.

- The Power Architecture Community Newsletter includes full-length articles as well as recent news about members of the Power Architecture community and upcoming events of interest. Learn more about the Power Architecture Community Newsletter and how to contribute to it. Subscription is free.

- All things Power are chronicled in the developerWorks Power Architecture editors' blog, which is just one of many developerWorks blogs.

- Find more articles and resources on Power Architecture technology and all things related in the developerWorks Power Architecture technology content area.
- Download a IBM PowerPC 405 Evaluation Kit to demo a SoC in a simulated environment, or just to explore the fully licensed version of Power Architecture technology. This and other fine Power Architecture-related downloads are listed in the developerWorks Power Architecture technology content area's downloads section.

About the author

Dr. Sam Siewert is an embedded system design and firmware engineer who has worked in the aerospace, telecommunications, and storage industries. He also teaches at the University of Colorado at Boulder part-time in the Embedded Systems Certification Program, which he co-founded. His research interests include autonomic computing, firmware/hardware co-design, microprocessor/SoC architecture, and embedded real-time systems.

Trademarks