Understanding Fault Tolerance and Reliability

Future systems will be more complex and so more susceptible to failure. Despite many proposals in the past three decades, fault tolerance remains out of the reach of the average computer user. The industry needs techniques that add reliability without adding significant cost.

Arun K. Somani
University of Washington

Nitin H. Vaidya
Texas A&M University

Most people who use computers regularly have encountered a failure, either in the form of a software crash, disk failure, power loss, or bus error. In some instances these failures are no more than annoyances; in others they result in significant losses. The latter result will probably become more common than the former, as society's dependence on automated systems increases.

The ideal system would be perfectly reliable and never fail. This, of course, is impossible to achieve in practice: System builders have finite resources to devote to reliability and consumers will only pay so much for this feature. Over the years, the industry has used various techniques to best approximate the ideal scenario. The discipline of fault-tolerant and reliable computing deals with numerous issues pertaining to different aspects of system development, use, and maintenance.

The expression "disaster waiting to happen" is often used to describe causes of failure that are seemingly well known, but have not been adequately accounted for in the system design. In these cases we need to learn from experience how to avoid failure.

Not all failures are avoidable, but in many cases the system or system operator could have taken corrective action that would have prevented or mitigated the failure. The main reason we don't prevent failures is our inability to learn from our mistakes. It often takes more than one occurrence of the same failure before corrective action is taken.

EXPRESSING FAULT TOLERANCE

The two most common ways the industry expresses a system's ability to tolerate failure are *reliability* and *availability*.

Reliability is the likelihood that a system will remain operational (potentially despite failures) for the duration of a mission. For instance, the requirement might be stated as a 0.999999 availability for a 10-hour mission. In other words, the probability of failure during the mission may be at most 10^{-6}. Very high reliability is most important in critical applications such as the space shuttle or industrial control, in which failure could mean loss of life.

Availability expresses the fraction of time a system is operational. A 0.999999 availability means the system is not operational at most one hour in a million hours. It is important to note that a system with high availability may in fact fail. However, its recovery time and failure frequency must be small enough to achieve the desired availability. High availability is important in many applications, including airline reservations and telephone switching, in which every minute of downtime translates into significant revenue loss.

UNDERSTANDING FAULT TOLERANCE

Systems fail for many reasons. The system might have been specified erroneously, leading to an incorrect design. Or the system might contain a fault that manifests only under certain conditions that weren't tested. The environment may cause a system to fail. Finally, aging components may cease to work properly. It's relatively easy to visualize and understand random failures caused by aging hardware. It's much more difficult to grasp how failures might be caused by incorrect specifications, design flaws, substandard implementation, poor testing, and operator errors. And many times failures are caused by a combination of these conditions.

In addition, a system can fail at several levels. The

A good fault-tolerant system design requires a careful study of design, failures, causes of failures, and system response to failures.

system or subsystem itself may fail, its human operator may fail and introduce errors, or the two factors may interact to cause a failure (when there is a mismatch between system operation and operator understanding). Operator error is the most common cause of failure. But many errors attributed to operators are actually caused by designs that require an operator to choose an appropriate recovery action without much guidance and without any automated help. The operator who chooses correctly is a hero; the one who chooses incorrectly becomes a scapegoat. The bigger question here is whether these catastrophic situations could have been avoided if the system had been designed in an appropriate, safe manner.

Importance of design

Piecemeal design approaches are not desirable. A good fault-tolerant system design requires a careful study of design, failures, causes of failures, and system response to failures. Such a study should be carried out in detail before the design begins and must remain part of the design process.

Planning to avoid failures—*fault avoidance*—is the most important aspect of fault tolerance. Proper design of fault-tolerant systems begins with the requirements specification. A designer must analyze the environment and determine the failures that must be tolerated to achieve the desired level of reliability. Some failures are more probable than others. Some failures are transient and others are permanent. Some occur in hardware, others in software. To optimize fault tolerance, it is important (yet difficult) to estimate actual failure rates for each possible failure.

The next obvious step is to design the system to tolerate faults that occur while the system is in use. The basic principle of fault-tolerant design is *redundancy*, and there are three basic techniques to achieve it: spatial (redundant hardware), informational (redundant data structures), and temporal (redundant computation).

Redundancy costs both money and time. The designers of fault-tolerant systems, therefore, must optimize the design by trading off the amount of redundancy used and the desired level of fault tolerance. Temporal redundancy usually requires recomputation and typically results in a slower recovery from failure compared to spatial redundancy. On the other hand, spatial redundancy increases hardware costs, weight, and power requirements.

Many fault tolerance techniques can be implemented using only special hardware or software, and some techniques require a combination of these. Which approach is used depends on the system requirements: Hardware techniques tend to provide better performance at an increased hardware cost; software techniques increase software design costs. Software techniques, however, are more flexible because software can be changed after the system has been built.

Here we describe the six most widely used hardware and software techniques.

Modular redundancy and *N*-version programming

Modular redundancy uses multiple, identical replicas of hardware modules and a *voter* mechanism. The voter compares the outputs from the replicas and determines the correct output using, for example, majority vote. Modular redundancy is a general technique—it can tolerate most hardware faults in a minority of the hardware modules.

N-version programming can tolerate both hardware and software faults. The basic idea is to write multiple versions of a software module. A voter receives outputs from these versions and determines the correct output. The different versions are written by different teams, with the hope that these versions will not contain the same bugs. N-version programming can therefore tolerate some software bugs that affect a minority of the replicas. However, it does not tolerate correlated faults, which may be catastrophic. In a correlated fault, the reason for failure may be common to two modules. For example, two modules may share a single power supply or a single clock. Failure of either may make both modules fail.

Error-control coding

Replication is effective but expensive. For certain applications, such as RAMs and buses, *error-correcting codes* can be used, and they require much less redundancy than replication. Hamming and Reed-Solomon codes are among those commonly used.

Checkpoints and rollbacks

A *checkpoint* is a copy of an application's state saved in some storage that is immune to the failures under consideration. A *rollback* restarts the execution from a previously saved checkpoint. When a failure occurs, the application's state is rolled back to the previous checkpoint and restarted from there. This technique can be used to recover from transient as well as permanent hardware failures and certain types of software failures. Both uniprocessor and distributed applications can use rollback recovery.

Recovery blocks

The recovery block technique uses multiple alternates to perform the same function. One module is the primary module; the others are secondary modules. When the primary module completes execution, its outcome is checked by an *acceptance test*. If the output is not acceptable, a secondary module executes, and so on, until either an acceptable output is obtained or the alter-

Beyond Fault Tolerance

Timothy C.K. Chou, Oracle Corporation

While the cost of hardware and software drops every year, the cost of downtime only increases. Whether measured by lost productivity, lost revenue, or poor customer satisfaction, downtime is becoming a significant part of the cost of computer-based services. Although many believe the state of the art in fault-tolerant technology is adequate, the following data shows that the industry has a long way to go.

State of the art

Availability is usually expressed in terms of the percentage of time a system is available to users. While this is a valid metric, a much more useful metric is outage minutes. Outage minutes are directly measurable, understandable, and—most important—they can be translated directly to cost. Most people are comfortable with a system that is "99 percent available," but if that means 5,000 outage minutes at a cost of $1,000 each minute, the cost of this 99 percent availability is actually $5 million per year.

There is little publicly available data about the number of outage minutes computer systems experience. One study found that the average mainframe experiences 57,000 outage minutes per year.[1] Another calculated Internet downtime at 15,000 outage minutes per year.[2] The US telecommunications industry requirement is that voice switches be down no more than two hours in 40 years, which is less than five outage minutes per year. Experience has shown that well-managed production systems can run in the range of 500 to 5,000 outage minutes per year. Many production systems do not run even this well.

All applications do not have the same cost of unavailability. A 1995 survey of 450 companies concluded that the average cost per outage minute for business systems is $1,300.[3] The famous nine-hour breakdown of AT&T's long distance network in early 1990 cost AT&T some $60 to $75 million in lost revenues—more than $100,000 per outage minute. The Gartner Group has calculated that an average LAN system has an annual downtime cost of more than $600,000.[4]

Whatever the numbers, the costs are driving both IT and MIS departments to consider availability as one of their most important issues.

7 × 24 Framework

Building systems that are always available (7 × 24 systems) is a complex, multidimensional problem. Developers need a framework or model that lets them discuss these challenges from both a customer and a technology view.

		50,000	5,000	500	50	5	0.5
Server	Hardware		716				
	System software		898				
	Application software		873				
Network	Hardware			170			
	System software			284			
	Application software						
Client	Hardware						
	System software						
	Application software						

Figure A. Customer view of 7 × 24 Framework.

Customer view

Consumer demand for services drives demand for availability, and so end-user availability is the only interesting metric. It is not enough to consider the availability of the hardware, the operating system, or even the database-management system. Today's applications consist of several components: the servers, the network, and the clients.

Figure A shows the customer view of the 7 × 24 Framework I developed. Across the top are the number of outage minutes/year. The left side shows the components seen by the customer's application: server, network, and client. Each of these is further divided into its major components: hardware, system software (operating system, database management, middleware, and networking), and applications software. Using this matrix, developers can plot where actual faults are occurring and their magnitude. For example, the numbers in Figure A are the outage minutes experienced by a very well run production system over a 12-month period. By plotting outage minutes data this way, organizations can invest their R&D funds in the areas that will have the most impact on their customers. Although the industry has tended to focus on the upper-left quadrant (hardware faults), this matrix shows that in fact

		Physical	Design	Operator	Environmental	Reconfiguration
Server	Hardware	2@156 outage minutes			1@60 outage minutes	3@500 outage minutes
	System software		3@565 outage minutes	1@120 outage minutes		4@213 outage minutes
	Application software		1@60 outage minutes	1@120 outage minutes		2@693 outage minutes
Network	Hardware	2@150 outage minutes		1@20 outage minutes		
	System software		3@134 outage minutes			2@150 outage minutes
	Application software					
Client	Hardware					
	System software					
	Application software					

Figure B. Technology view of 7 × 24 Framework.

system and application faults have a greater magnitude in terms of outage minutes.

Technology view

There are two ways to obtain zero outage minutes: decrease the number of faults and decrease the time to recover from a fault. Five classes of faults are relevant: physical, design, operator, environmental, and reconfiguration.

The technical challenge is to eliminate or tolerate all faults in all classes for a range of hardware and software components. Figure B shows the technology view of the 7 × 24 Framework. The notch under physical faults reflects the fact that there is no such thing as a physical software failure. Again, I have plotted actual data into the matrix: the number of faults and the total number of outage minutes attributed to those faults.

Physical faults. High-quality hardware design will continue to be a prerequisite to building reliable 7 × 24 systems. Beyond that, commodity cluster-management software will enable every system to tolerate permanent physical faults. As we move to smaller and smaller VLSI implementations, *transient* hardware faults will predominate; today about 80 percent of all hardware faults are transient. (If your PC fails only to seem okay on rebooting, you just experienced a transient fault.) Transient faults are particularly nasty: They are difficult to diagnose and, worse, can corrupt data. We need research to determine how transient faults affect logic and memory circuits and how VLSI scaling will affect transient fault rates.

Soon systems will be built from devices with more than 100 million transistors. At these densities it will be impossible to prove design correctness or to exhaustively test production parts. Here research must focus on improving the CAD tools to ensure design correctness and on developing new design-for-test methods and online techniques to detect faults before they can corrupt data.

Design faults. Design faults are the primary cause for software failures, although of course there are hardware design faults as well. The Year 2000 problem is a well-known design fault. While the progress in reducing hardware design faults can be largely attributed to the stuck-at fault model, no such model exists for software. What software fault models do exist are inaccurate, because there have been few documented, quantitative experiments to determine why software fails. The Software Engineering Institute has done much to document different maturity levels as they relate to software productivity. Is there experimental data to show that an organization rated at a higher CMM level delivers more reliable software? Research is needed to define a software fault model based on outage profiles.

Bugs greatly contribute to software downtime. Software engineering technology does offer improvements: An analysis of "excellent" software engineering versus "average" groups has shown a 2:1 difference in the number of permanent design faults found in each 1,000 lines of code developed.[6]

Just as in hardware, transient software faults account for an increasingly large percentage of bugs found in the field. The software manufacturing process typically removes deterministic, predictable design faults. Once in production, software suffers from a large number of state-dependent faults. Process-pair technology, which allows a backup process to shadow a primary process, has a demonstrated ability to tolerate transient software faults. One study, using a sample of 2,000 systems representing 10 million system hours, revealed that about 99.3 percent of the faults affecting a primary/backup spooling group affected just one member of the group but left the spooling process running correctly.[5] Advanced technologies such as process pairs are difficult to program. Research is needed to incorporate this technology into modern programming environments.

Operator faults. Some say the best technology is a mean dog, installed between the operator and the console to keep the operator from introducing error. Mean dogs may not be practical, but automating the operator function can dramatically affect speed and accuracy: Production systems have demonstrated a 50 percent reduction of outage minutes using automated recovery.

Automation is a good first step, but there has been very little research into how to build fault-tolerant operator interfaces. Studies in this area done for aviation and military applications have not yet crossed over into the broader computer industry. As in the case of software, there is no operator fault model. The industry needs an operator fault model based on outage profiles. In the absence of this fundamental research, continued investment in automated operations, recovery, and security is important.

Environmental faults. Environmental faults include disasters such as earthquakes as well as simple power failures. The National Power Lab reports that the average computer room experiences 443 power glitches (264 sags, 128 surges, 36 spikes, and 15 outages) each year. Downtime attributed to the environment was believed to be at 30 percent in 1990 and projected to be well over 50 percent by the late 1990s.[7,8]

The telecommunications industry has taken the lead in creating technology to tolerate hardware failures caused by environmental factors. The generic Network Equipment Building Standard (NEBS) specification, for example, defines requirements for office equipment that can tolerate an 8.2 earthquake. Increasingly, many computer systems are deployed in buildings with motor-generator pairs, batteries, universal power supplies, and other ac sources. Computers are powered with dc sources. Rather than have this array of equipment converting ac to dc to ac to dc, NEBS-compliant equipment is designed to be powered directly off a –48V dc outlet.

Environmental fault-tolerant software also requires database technology that can synchronize two servers installed far enough apart to tolerate these faults. Technology to address this problem is driven by three parameters: the degree of synchronization, the time required to resynchronize after failure, and the distance between the databases. A remote database facility, extended mirroring, and other techniques are some solutions requiring more research.

Reconfiguration faults. While the other four classes include unplanned faults, this class of faults includes "planned outages" such as taking a system down to add a disk or change a piece of software. Systems built for 7 × 24 availability have no window for maintenance or upgrades. In order to physically reconfigure a database today, you have to shut down the communications, shut down the middleware, move the data, update the file labels, recompile the query language, write the database to disk, restart the middleware, and restart communications—a series of operations that can take hundreds of minutes. Some technology exists to solve parts of this problem, but much work remains.

While computer technology has made great strides in reliability, current demands are beginning to exceed our ability to deliver solutions. While hardware is getting more reliable and fault tolerant, there remains much to be done in developing systems that can tolerate design, operator, environmental, and reconfiguration faults. ❖

References

1. S. Smith, "Stratus Computer, Just Another 2nd Tier Computer Company," Paine Webber Report, Nov. 30, 1989.
2. J. Gray and D. Siewiorek, "High Availability Computer Systems," *Computer*, Sept. 1991, pp. 39-48.
3. *The Impact of Online Computer Systems Downtime on American Businesses: A Survey of Senior MIS Executives Prepared for Stratus*, Find/SVP Strategic Research Division, New York, 1995.
4. *Review of the Telecommunications Industry*, Gartner Group, Stamford, Conn., July 16, 1993.
5. M. Cusumano, *Japan's Software Factories*, Oxford University Press, Stamford, Conn., 1991.
6. J. Gray, "Why Do Computers Stop and What Can be Done About It?," *Proc. Fifth Symp. Reliability in Distributed Software and Database Systems*, CS Press, Los Alamitos, Calif., 1986, pp. 13-17.
7. "Computersite Engineering," *MIS Week*, June 18, 1994.
8. "Power Glitches Become Critical as World Computerizes," *Wall Street J.*, May 18, 1995.

Timothy Chou is chief operations officer and senior vice president of research & development at Reasoning, Inc. Reasoning provides transformation software that automatically finds and fixes Year 2000 design faults. Previously he was vice president, server products group, at Oracle. Chen received a PhD in computer engineering from the University of Illinois at Urbana-Champaign. He also teaches introductory computer architecture at Stanford University. Contact him at Reasoning Systems, 3260 Hillview Ave., Palo Alto, CA 94304; chou@reasoning.com.

nates are exhausted. Recovery blocks can tolerate software failures because the alternates are usually implemented with different approaches (algorithms).

DEPENDABILITY EVALUATION

Once a fault-tolerant system is designed, it must be evaluated to determine if its architecture meets reliability and dependability objectives. There are two ways to evaluate dependability: using an analytical model or injecting faults.

An analytical model (such as a Markov model) can help developers determine a system's possible states and the rates or probabilities of transitions among them. Such a model can then be used to evaluate different dependability metrics for a system. Unfortunately, it is very complex to analyze models accurately, so this approach is not viable for many systems.

Various types of faults can be injected into a system, again to determine various dependability metrics. Faults can be injected into simulated or real systems, or into a system that is part simulation, part real. Fault injection can be used to evaluate hardware or software. To choose an approach, developers must consider the characteristics of the system under consideration, the desired metrics, and the desired accuracy of the metrics.

The increased complexity of future systems will make them even more susceptible to failure. Yet, despite many proposals in the past three decades, fault tolerance remains out of the reach of the average computer user. The industry needs low-cost techniques that can provide an added measure of fault tolerance and reliability to the computers used by the vast majority of users. For this to happen, the extra costs associated with fault tolerance (especially monitoring and performance) must be minimized.

Next to failures caused by software and computer operators, hardware failures are comparatively easy to understand. Past research has been much more effective in obtaining solutions for tolerating hardware failures, so now the pressure is on to develop solutions for software and operators. The search for better solutions is expected to continue well into the next millennium. ❖

Acknowledgments

We thank Joan Francioni, *Computer*'s former theme issue editor for her help in proposal stages of this theme issue. We thank the authors of 30 submitted papers for their contributions. We also thank the reviewers, whose timely reviews helped us choose the articles that appear in this issue.

For More Information

Sources of information on fault-tolerant and reliable computing include:

- The electronic newsletter of the IEEE Computer Society's Technical Committee on Fault-Tolerance (contact Arun Somani at somani@ee.washington.edu).
- Conferences and symposiums, including the International Symposium on Fault-Tolerant Computing (http://denali.ee.washington.edu/~webroot/ftcs97.html), Symposium on Reliable Distributed Systems, Pacific Rim International Symposium on Fault-Tolerant Systems (flai@cc.ee.ntu. edu.tw), European Dependable Computing Conference, and Dependable Computing for Critical Applications.
- Special issues of *IEEE Transactions on Computers* (January 1998, February 1995, and May 1992) and *Computer* (July 1990).

Arun K. Somani is a professor of electrical engineering and computer science and engineering at the University of Washington. Somani's research interests are in the area of fault-tolerant computing, interconnection networks, parallel computer system architecture, and parallel algorithms. He received a BE with honors in electronics engineering from the Birla Institute of Technology and Science, India; an MTech in computer engineering from the Indian Institute of Technology; and an MSEE and a PhD in electrical engineering from McGill University. He is a senior member of the IEEE and a member of ACM, the IEEE Computer Society, and the IEEE Communications Society.

Nitin H. Vaidya is an assistant professor of computer science at Texas A&M University. His research interests include mobile/wireless computing and fault-tolerant computing. He received a BE in electrical and electronics engineering from the Birla Institute of Technology and Science, India; an ME in computer science and engineering from the Indian Institute of Science; and an MS and a PhD in electrical and computer engineering from the University of Massachusetts. He is editor of the IEEE Technical Committee on Computer Architecture's newsletter. He is a member of ACM and the IEEE Computer Society.

Contact Somani at Dept. of Electrical Engineering, Univ. of Washington, Seattle, WA 98195-2500; somani@ee.washington.edu. Contact Vaidya at the Computer Science Dept., Texas A&M Univ., College Station, TX 77843-3112; vaidya@cs.tamu.edu.