Fault Tolerance

Jeffrey Voas

Participants

Joanne Bechta Dugan is a professor of electrical and computer engineering at the University of Virginia. Her research interests include hardware and software reliability engineering, fault-tolerant computing, and mathematical modeling using dynamic fault trees, Markov models, Petri nets, and simulation. Contact her at jbd@virginia.edu.

Les Hatton's biography appears on page 39.

Karama Kanoun's biography appears on page 33.

Jean-Claude Laprie is a research director at CNRS, the French National Organization for Scientific Research. He is Director of LAAS-CNRS, where he founded and previously directed the research group on Fault Tolerance and Dependable Computing. He also founded and previously directed the Laboratory for Dependability Engineering, a joint academia–industry laboratory. His research interests have focused on fault tolerance, dependability evaluation, and the formulation of the basic concepts of dependability. Contact him at laprie@laas.fr.

Mladen A. Vouk is a professor of computer science at the N.C. State University, Raleigh, North Carolina. His research and development interests include software engineering, scientific computing, computer-based education, and high-speed networks. He received his PhD from the King's College, University of London, UK. He is an IEEE Fellow and a member of the IEEE Reliability, Communications, Computer and Education Societies, and of the IEEE TC on Software Engineering, ACM, ASQC, and Sigma Xi. Contact him at vouk@csc.ncsu.edu.

This is a "virtual" roundtable discussion between five respected experts from the fault-tolerant computing field: Joanne Bechta Dugan (UVA), Les Hatton (Oakwood Computing), Karama Kanoun and Jean-Claude Laprie (LAAS-CNRS), and Mladen Vouk (NC State University). The reason for including this piece is to provide a mini tutorial for readers who are unfamiliar with the field of software fault tolerance or who might have had more exposure to hardware fault tolerance. Eleven questions were posed to each member of the panel over email, and here we present their responses.

—*Jeffrey Voas*

What are the basic principles of building fault-tolerant systems?

Joanne Bechta Dugan: To design and build a fault-tolerant system, you must understand how the system should work, how it might fail, and what kinds of errors can occur. Error detection is an essential component of fault tolerance. That is, if you know an error has occurred, you might be able to tolerate it—by replacing the offending component, using an alternative means of computation, or raising an exception. However, you want to avoid adding unnecessary complexity to enable fault tolerance because that complexity could result in a less reliable system.

Les Hatton: The basic principles depend, to a certain extent, on whether you're designing hardware or software. Classic techniques such as independence work in different ways and with differing success for hardware as compared to software.

Karama Kanoun and Jean-Claude Laprie: Fault tolerance is generally implemented by error detection and subsequent system recovery. Recovery consists of *error handling* (to eliminate errors from the system state) and *fault handling* (to prevent located faults from being activated again). Fault handling involves four steps: fault diagnosis, fault isolation, system reconfiguration, and system reinitialization. Using sufficient redundancy might allow recovery without explicit error detection. This form of recovery is called *fault masking*. Fault tolerance can also be implemented preemptively and preventively—for example, in the so-called software rejuvenation, aimed at preventing accrued error conditions to lead to failure.

Mladen Vouk: A principal way of introducing fault tolerance into a system is to provide a method to dynamically determine if the system is behaving as it should—that is, you introduce a self-checking or "oracle" capability. If the method detects unexpected and unwanted behavior, a fault-tolerant system must provide the means to recover or continue operation (preferably, from the user's perspective, in a seamless manner).

What is the difference between hardware and software fault tolerance?

Joanne: The real question is what's the difference between design faults (usually software) and physical faults (usually hardware). We can tolerate physical faults in redundant (spare) copies of a component that are identical to the original, but we can't generally tolerate design faults in this way because the error is likely to recur on the spare component if it is identical to the original. However, the distinction between design and physical faults is not so easily drawn. A large class of errors

0740-7459/01/$10.00 © 2001 IEEE

arise from design faults that do not recur because the system's state might slightly differ when the computation is retried.

Les: From a design point of view, the basic principles of thinking about system behavior as a whole are the same, but again, we need to consider what we're designing—software or hardware. I suspect the major difference is cost. Software fault tolerance is usually a lot more expensive.

Karama and Jean-Claude: A widely used approach to fault tolerance is to perform multiple computations in multiple channels, either sequentially or concurrently. To tolerate hardware physical faults, the channels might be identical, based on the assumption that hardware components fail independently. Such an approach has proven to be adequate for *soft faults*—that is, software or hardware faults whose activation is not systematically reproducible—through rollback. Rollback involves returning the system back to a saved state (a checkpoint) that existed prior to error detection. Tolerating solid design faults (hardware or software) necessitates that the channels implement the same function through separate designs and implementations—that is, through design diversity.

Mladen: Hardware fault tolerance, for the most part, deals with random failures that result from hardware defects occurring during system operation. Software fault tolerance contends with random (and sometimes not-so-random) invocations of software paths (or path and state or environment combinations) that usually activate software design and implementation defects. These defects can then lead to system failures.

What key technologies make software fault-tolerant?

Joanne: Software involves a system's conceptual model, which is easier than a physical model to engineer to test for things that violate basic concepts. To the extent that a software system can evaluate its own performance and correctness, it can be made fault-tolerant—or at least error-aware. What I mean is that, to the extent a software system can check its responses before activating any physical components, a mechanism for improving error detection, fault tolerance, and safety exists. Think of the adage, "Measure twice, cut once." The software analogy might be, "Compute twice, activate once."

Les: I'm not sure there is a key technology. The behavior of the system as a whole and the hardware and software interaction must be thought through very carefully. Efforts to support software fault tolerance, such as exception handling in languages, are rather crude, and education on how to use them is not well established in universities.

Karama and Jean-Claude: We can use three key technologies—design diversity, checkpointing, and exception handling—for software fault tolerance, depending on whether the current task should be continued or can be lost while avoiding error propagation (ensuring error containment and thus avoiding total system failure). Tolerating solid software faults for task continuity requires diversity, while checkpointing tolerates soft software faults for task continuity. Exception handling avoids system failure at the expense of current task loss.

Mladen: Runtime failure detection is often accomplished through either an acceptance test or comparison of results from a combination of "different" but functionally equivalent system alternates, components, versions, or variants. However, other techniques—ranging from mathematical consistency checking to error coding to data diversity—are also useful. There are many options for effective system recovery after a problem has been detected. They range from complete rejuvenation (for example, stopping with a full data and software reload and then restarting) to dynamic forward error correction to partial state rollback and restart.

What is the relationship between software fault tolerance and software safety?

Joanne: Both require good error detection, but the response to errors is what differentiates the two approaches. Fault tolerance implies that the software system can recover from—or in some way tolerate—the error and continue correct operation. Safety implies that the system either continues correct operation or fails in a safe manner. A safe failure is an inability to tolerate the fault. So, we can have low fault tolerance and high safety by safely shutting down a system in response to every detected error.

Les: It's certainly not a simple relationship. Software fault tolerance is related to reliability, and a system can certainly be reliable and unsafe or unreliable and safe as well as the more usual combinations. Safety is intimately associated with the system's capacity to do harm. Fault tolerance is a very different property.

Karama and Jean-Claude: Fault tolerance is—together with fault prevention, fault removal, and fault forecasting—a means for ensuring that the system function is implemented so that the dependability attributes, which include safety and availability, satisfy the users' expectations and requirements. Safety involves the notion of controlled failures: if the system fails, the failure should have no catastrophic consequence—that is, the system should be fail-safe. Controlling failures always include some forms of fault tolerance—from error detection and halting to complete system recovery after component failure. The system function and environment dictate, through the requirements in terms of service continuity, the extent of fault tolerance required.

Mladen: You can have a safe system that has little fault tolerance in it. When the system specifications properly and adequately define safety, then a well-designed fault-tolerant system will also be safe. However, you can also have a system that is highly fault-tolerant but that can fail in an unsafe way. Hence, fault tolerance and safety are not synonymous. Safety is concerned with failures (of any nature) that can harm the user; fault tolerance is primarily concerned with runtime prevention of failures in any shape or

form (including prevention of safety-critical failures). A fault-tolerant and safe system will minimize overall failures and ensure that when a failure occurs, it is a safe failure.

What are the four most seminal papers or books on this topic?

Joanne: Important references on my desk include Michael Lyu's *Software Fault Tolerance* (John Wiley, 1995), Nancy Leveson's *Safeware: System Safety and Computers*—especially for the case studies in the appendix (Addison-Wesley, 1995), Debra Herrmann's *Software Safety and Reliability* (IEEE Press, 2000), and Henry Petroski's *To Engineer is Human: The Role of Failure in Successful Design* (Vintage Books, 1992).

Les: The literature is pretty packed, although some of Nancy Leveson's contributions are right up there. Beyond that, I usually read books on failure tolerance and safety in conventional engineering.

Karama and Jean-Claude: B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, no. 10, 1975, pp. 1220–1232. A. Avizienis and L. Chen, "On the implementation of N-version Programming for Software Fault Tolerance During Execution," *Proc. IEEE COMPSAC 77*, 1977, pp. 149–155. *Software Fault Tolerance*, M.R. Lyu, ed., John Willey, 1995. J.C. Laprie et al., "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures," *Computer*, vol. 23, no. 7, July 1990, pp. 39–51.

Mladen: I agree that M. Lyu's *Software Fault-Tolerance* and B. Randell's "System Structure for Software Fault-Tolerance" are important. Also, A. Avizienis, "The *N*-Version Approach to Fault-Tolerant Software," *IEEE Trans. Software Eng.*, vol. SE-11, no. 12, 1085, pp. 1491–1501, and J.C. Laprie et al., "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures," *Computer*, vol. 23, no. 7, July 1990, pp. 39–51. (Reprinted in *Fault-Tolerant Software Systems: Techniques and Applications*, Hoang Pham, ed., IEEE Computer Soc. Press, 1992, pp. 5–17.)

Much research has been done in this area, but it seems little has made it into practice. Is this true?

Joanne: I think it's a psychological issue. In general, I don't think that software engineers are trained to consider failures as well as other engineers. Software designers like to think about neat ways to make a system work—they aren't trained to think about errors and failures. Remember that most software faults are design faults, traceable to human error. If I have a finite amount of resources to expend on a project, it's hard to make a case for fault tolerance (which usually implies redundancy) rather than just spending more time to get the one version right.

Les: Yes, I agree that little has made it into practice. Most industries are in too much of a rush to do it properly—if at all—with reduced time-to-market so prevalent, and software engineers are not normally trained to do it well. There is also a widespread and wholly mistaken belief that software is either right or wrong, and testing proves that it is "right."

Karama and Jean-Claude: This could be true for design diversity, as it has been used mainly for safety-critical systems only. Exception handling is used in most systems: telecommunications, commercial servers, and so forth.

Mladen: I think we've made a lot of progress—we now know how to make systems that are quite fault-tolerant. Most of the time, it is not an issue of technology (although many research issues exist) but an issue of cost and schedules. Economic and business models dictate cost-effective and timely solutions. Available fault-tolerant technology, depending on the level of confidence you desire, might offer unacceptably costly or time-consuming solutions. However, most of our daily critical reliability expectations are being met by fault-tolerant systems—for example, 911 services and aircraft flight systems—so it is possible to strike a successful balance between economic and technical models.

How can a company deciding whether to add fault tolerance to a system determine whether the return on investment is sufficient for the additional costs?

Joanne: I think it all hinges on the cost of failure. The more expensive the failure (in terms of actual cost, reputation, human life, environmental issues, and other less tangible measures), the more it's worth the effort to prevent it.

Les: This is very difficult. Determining the cost of software failure in actuarial terms given the basically chaotic nature of such failure is just about impossible at the current level of knowledge—the variance on the estimates is usually ridiculous. ROI must include a knowledge of risk. Risk is when you are not sure what will happen but you know the odds. Uncertainty is when you don't know either. We generally have uncertainty. On the other hand, I am inclined to believe that the cost of failure in consumer embedded systems is so high that all such systems should incorporate fault-tolerant techniques.

Karama and Jean-Claude: It is misleading to consider the ROI as the only criterion for deciding whether to add fault tolerance. For some application domains, the cost of a system outage is a driver that is more important than the additional development cost due to fault tolerance. Indeed, the cost of system outage is usually the determining factor, because it includes all sources of loss of profit and income to the company.

Mladen: There are several fault-tolerance cost models we can use to answer this question. The first step is to create a risk-based operational profile for the system and decide which system elements need protection. Risk analysis will yield the problem occurrence to cost-to-benefit results, which we can then use to make appropriate decisions.

What key standards, government agencies, or standards bodies require fault-tolerant systems?

Joanne: I'd be surprised if there are any standards that require fault tolerance (at least for design faults)

per se. I would expect that correct operation is required, which might imply fault tolerance as an internal means to achieve correct operation.

Les: Quite a few standards have something to say about this—for example, IEC 61508. However, IEC 61508 also has a lot to say about a lot of other things. I do not find the proliferation of complex standards particularly helpful, although they are usually well meaning. There is too much opinion and not enough experimentation.

Karama and Jean-Claude: Several standards for safety-critical applications recommend fault tolerance—for hardware as well as for software. For example, the IEC 61508 standard (which is generic and application sector independent) recommends among other techniques: "failure assertion programming, safety bag technique, diverse programming, backward and forward recovery." Also, the Defense standard (MOD 00-55), the avionics standard (DO-178B), and the standard for space projects (ECSS-Q-40-A) list design diversity as possible means for improving safety.

Mladen: Usually, the requirement is not so much for fault tolerance (by itself) as it is for high availability, reliability, and safety. Hence, IEEE, FAA, FCC, DOE, and other standards and regulations appropriate for reliable computer-based systems apply. We can achieve high availability, reliability, and safety in different ways. They involve a proper reliable and safe design, proper safeguards, and proper implementation. Fault tolerance is just one of the techniques that assures that a system's quality of service (in a broader sense) meets user needs (such as high safety).

How do you demonstrate that fault tolerance is achieved?

Joanne: This is a difficult question to answer. If we can precisely describe the classes of faults we must tolerate, then a collection of techniques to demonstrate fault tolerance (including simulation, modeling, testing, fault injection, formal analysis, and so forth) can be used to build a credible case for fault tolerance.

Les: I honestly don't know. Demonstrating that something has been achieved in classic terms means comparing the behavior with and without the additional tolerance-inducing techniques. We don't do experiments like this in software engineering, and our prediction systems are often very crude. There are some things you can do, but comparing software technologies in general is about as easy as comparing supermarkets.

Karama and Jean-Claude: As for any system, analysis and testing are mandatory. However, fault injection constitutes a very efficient technique for testing fault-tolerance mechanisms: well-selected sets of faults are injected in the system and the reaction of fault-tolerance mechanisms is observed. They can thus be improved. The main problem remains the representativeness of the injected faults. For design diversity, the implementation of the specifications by different teams has proven to be efficient for detecting specifications faults. Back-to-back testing has also proven to be efficient in detecting some difficult faults that no other method can detect.

Mladen: You know you've achieved fault tolerance through a combination of good requirements specifications, realistic quantitative availability, reliability and safety parameters, thorough design analysis, formal methods, and actual testing.

What well-known projects contain fault-tolerant software?

Les: Certainly some avionics systems and some automobile systems, but the techniques are not usually widely publicized so I'm not sure that many would be well known. Ariane 5 is a good, well-documented example of a system with hardware tolerance but no software tolerance.

Karama and Jean-Claude: Airbus A-320 and successors use design diversity for software fault tolerance in the flight control system. Boeing B-777 uses hardware diversity and compiler diversity. The Elektra Austrian railway signaling system uses design diversity for hardware and software fault tolerance. Tandem ServerNet and predecessors (NonStop) tolerate soft software faults through checkpointing.

Mladen: Well-known projects include the French train systems, Boeing and Airbus fly-by-wire aircraft, military fly-by-wire, and NASA space shuttles. Several proceedings and books, sponsored by IFIP WG 10.4 and the IEEE CS Technical Committee on Fault Tolerant Computing, describe older fault-tolerance projects (see www.dependability.org).

Is much progress being made into fault-tolerant software?

Les: I don't think there is much new in the theory. Endless and largely pointless technological turnover in software engineering just makes it harder to do the right things in practice. With a new operating system environment emerging about every two years on average, a different programming language emerging about every three years, and a new paradigm about every five years, it's a wonder we get anywhere at all.

Karama and Jean-Claude: To the best of our knowledge, the main concepts and means for software fault tolerance have been defined for a few years. However, some refinements of these concepts are still being done and implemented in practice, mainly for distributed systems such as fault-tolerant communication protocols.

Mladen: The basic ideas of how to provide fault tolerance have been known for decades. Specifics on how to do that cost effectively in modern systems with current and future technologies (wireless, personal devices, nanodevices, and so forth) require considerable research as well as ingenious engineering solutions. For example, our current expectations for 911 services require telephone switch availabilities that are in the range of 5 to 6 nines (0.999999). These expectations are likely to start extending to more mundane things such as personal computing and networking applications, car automation (such as GIS locators), and, perhaps, space tourism. Almost none of these would measure up today, and many will require new fault-tolerance approaches that are far from standard textbook solutions.