

Challenges facing Software Fault-tolerance

Ram Chillarege

Center for Software Engineering, IBM Watson Research
30 Saw Mill River Road, Hawthorne, NY 10532
ramchill@watson.ibm.com, (914) 784 7375

Abstract

As software dominates most discussions in the information technology business, one needs to carefully examine where we are headed in software dependability. This paper re-examines some of the basic premises upon which the area of software fault-tolerance is built and critiques some current practices and beliefs. A few of the thoughts and contributions are:

- The definition of a software failure needs to change from a *specification* based thought to one of *customer expectation* and *ability to do productive work*. This will cause a significant shift on what we build fault-tolerance for. However, it would also help narrow the gap between today's theory, practice and customer need.
- Data on customer problems illustrates that 90% of the problems reported are what we have traditionally considered as non-defect – implying no need for a programming change. However, with the new definition of failure, we will need to address this more seriously as a part of fault-tolerance. This change could level the playing field and help achieve greater customer satisfaction.
- A rationale for determining the amount of fault-tolerance based on the concept of *the threshold of pain*, is suggested. It helps guide the prioritization of fault-tolerance amongst competing forces, by platform and market segment.

In conclusion, the paper reflects on a few of the development world “realities” to temper what can be achieved and what we as a community need to be aware of.

1 What is a software failure?

Fundamental to developing methods for software fault-tolerance is a need for clear understanding of what faults to tolerate. This is usually a question that is assumed to be well understood. However, when one examines closely, it is startling how little is truly understood in terms of the faults that need to be tolerated. This problem is significantly better understood in the hardware than it is in software. Failures in hardware components, sub-systems or systems, have been better tracked over the years and there is a larger body of experience in failure

modes and effects analysis. However, the counterpart in software is far less understood. It is further complicated by a lack of clarity as to what is considered a software failure.

The concept of a failure is defined as “deviation of the delivered service from compliance with the specification” [7]. This definition is very applicable in the area of hardware where there tends to be a reasonable level of specifications either on the product or the system. However, in the world of software, it causes a major source of confusion since there aren’t well-defined specifications for most products. Having an unclear definition for failure makes the task of designing fault-tolerance even harder. Therefore, it is necessary for us to carefully examine what might be an apt definition of failure in the area of software.

A good place to start is the customer service centers. Apart from the very obvious failures when a product ceases to work, there are a myriad of problem conditions where the distinction between failure and correct operation is not very clear. If there exists well-defined specifications, it could be argued whether the product worked as designed or not, distinguishing failure from correct operation. However, as the software business explodes and reaches new customer groups, the tradition of a specification has changed. Often products do not have detailed specifications, nor would the customer expect them. Determining a failure by strict definition of the word loses relevance. What is important to the customer is that the product be able to do the work and that they are satisfied. Customer expectation largely determines whether a failure has occurred or not.

It is far more useful, in the modern software business, to define a failure when customers expectation has not been met and are unable to do useful work with the product.

This change in the definition of what is considered a failure has major repercussions throughout the software industry. It changes the focus of how products are planned and designed. For the fault tolerant community, this would be a major change that needs to be woven into the fabric of research, tools, and the development process.

Let us examine some of the immediate implications. Firstly, there is no fixed fault model. Since we drive the definition of a failure primarily by customer expectation and their ability to work, it will be a moving target. As technology evolves, product concepts evolve and knowledge in the customer base grows; the types of faults to be tolerated will change. To plan the fault-tolerance one has to be aware of the window of opportunity and aggressively conduct the studies to gain the necessary insights. It also puts a variety of disparate problems on the same platter. Since the focus is on customer expectations, it does not distinguish sources of faults as much as the effect. Some would argue that this has always been the case. However, with the new definition of failure there is no concept of a specification to hide under.

To gain a perspective of how important the different kinds of failures and faults are it is instructive to study some real data. Figure 1 shows a pareto of the different causes of outage in a commercial data processing center. These data are from an application that uses host computers and host software. This view of failure is much more traditional, since failure was associated with a hard outage and loss of availability. The message is that the proportion of software related outages is far more significant than other sources of outage. These data are consistent with similar trends identified by [5].

However, these cases belong to the class of well-known failure modes and scenarios that fault-tolerance has traditionally addressed. To gain a broader perspective one has to look at the new sources of problems and complaints that are raised by the users. Once again, a good place to gain such a perspective is to study customer call center experience.

The data captured in the customer call centers has traditionally been split into defect oriented

problems and non-defect oriented problems. The word defect refers to the fact that there is a change in software, which is necessitated due to a programming fault. A non-defect has been associated in the past with issues that do not require a programming change. It would include difficulties with install, use, configurations, inter-operability, etc. The pareto in Figure 2 shows a 90/10 split between the non-defect oriented problems and defect oriented problems. The tradition view would have been that 90% of the problems are not associated with what would be classified as a software failure. However, since we have redefined software failures as events where customer expectation is not met or that a customer is unable to do their work it changes the set of problems that need to be also considered for fault-tolerance. Beneath the set of failure are a set of faults. When we have a new set of failures to address, fault-tolerance has to reexamine the methods, techniques and issues that we focus on. This is a major shift in the perception of what fault-tolerance needs to provide and what directions research needs to take.

2 How much fault-tolerance?

A major point of discussion and contention during product planning is how much fault-tolerance is needed. This is a difficult question to answer and there are no formulae to arrive at the right amount. The areas where there are clearer guidelines are hardware and semiconductor technology where there is a better history of field data to base ones judgement. In the area of software, even measuring a mean time between failure is extremely difficult. Only few reports exist from the Telcom industry [4, 6] and the few known data points for commercial software [3, 5]. There are some guidelines set by federal agencies for safety critical applications such as the air traffic control. However, there are no such magic numbers for commercial software. A recent discussion that explores the limits of computer systems dependability from several perspectives can be found in [8].

A relevant question to ask is how much would a customer pay for the fault-tolerance? This is also a difficult question to answer. One of the calculations which could makes sense is to estimate the loss of work incurred due to failures and argue a business case for the value add of fault-tolerance. In most cases, these make better arguments for availability than for fault-tolerance alone. These computations are commonly done for hardware redundancy. They are not as effectively done for software primarily because software fault-tolerance has yet to make a significant and credible impact.

Given the discussion that we had under the section of software failures it is further unclear as to what kind of failures are really worth tolerating. The classical model of tolerating outage and creating fault-tolerance to avoid a total outage or a loss of availability is only one part of the puzzle. The other part of the puzzle that is growing in importance is the usability and non-defect oriented problems that cause disruption of work and loss of productivity. Attacking these will need a new set of fault models that identify human related faults. What is increasingly becoming evident is that one needs a rational basis to guide these trade-offs and help the decision making. The mere increase of mean time between hard failures and decrease in total outages alone may not answer the overall issues.

Finally, fault-tolerance is governed by the level of investment that one is willing to put into a product. Fault tolerance costs money. The development of known fault-tolerance techniques into a product also require design time and incurs additional test costs and possibly lengthens the development cycle time. Therefore, a judicious decision has to be made on the amount of fault tolerant capability planned for a product. As always, these decisions do not necessarily have mathematical backing to precisely engineer what needs to be done. However, it is useful to

develop a rationale that apply across the disparate issues to help arrive at a reasonable design point.

2.1 Fault-tolerance: Just enough to keep below the threshold of pain

To help develop a conceptual model of tradeoff, one needs to recognize that the goal of fault-tolerance is to insure that customer expectation is met and that their work is productive. The corollary to this is that there will always be failures, resulting in a certain degree of pain to the customer. Critical to the success of a product is to know what level of pain is too much, and engineer the product so that we operate just below that threshold of pain. Engineering implies that we are able to develop a product that just meets a required specification and not overexceed it or underachieve it too much. Fault tolerance should ensure that it operates way below the threshold of pain at a reasonable cost; then a customer may be satisfied and their expectation met.

We need to recognize the sources of pain which changes as a function of time and technology. In the history of fault tolerant computing, one has traditionally focused on those faults that tend to cause total outage of a product attributable directly to programming errors. However, if the sources of pain are largely due to the non-defect oriented problems then the technology does not address the needs of the customer. This does not mean that one loses perspective of the classical total outage and failure models, but one does not focus on those exclusively at the cost of ignoring issues that cause loss of productivity.

Although the concept of the threshold of pain is fairly intuitive, the complexity arises from the fact that the different industry segments and platforms have different issues that are a priority. Traditionally, host based computing has placed a premium on outage and availability, which fall into the classical fault models that fault tolerant computing has addressed. The client software and desktop software has not paid as much attention to the classical fault models. However, it has paid close attention to usability which is the dominant source of pain for the average customer. As each of these markets evolve and mature, the attributes that originally made them successful do not necessarily make them successful in the future. Since each product has a different design point at conception, we have to consciously make a switchover into addressing new aspects of the pain that become important to maintain their competitive position.

3 Traditional software fault-tolerance

The area of software fault-tolerance has probably two broadly known paradigms of implementing fault-tolerance. One of them is the recovery block [9] and the other one is N-version programming [1]. A third that probably has no author and is likely to be the most widely used is retry. All these three methods largely address the fault model of a programming error that causes a malfunction resulting in an error that is identifiable. The error may be identified through an acceptance test, multiple versions, or detected by the system. The fault models assumed have mostly been derived from the hardware counterparts and extended into the programming model. Thus, they are often inside-out views as opposed to outside-in views. This is broad generalization (and, therefore, will not always be true), but helps provide a framework to recognize where the technology is today.

The model of a recovery block is very intuitive and helps formulate a strategy for fault-tolerance. The difficulty is in trying to develop acceptance tests that are reasonable which provide adequate coverage. If the software was trying to recover from a hardware fault, it is a kind of a problem; if from a programming error, it is entirely another kind of problem. Programming errors within a module are only a part of the problem. The larger set of problems

being interactions between components and inheritance conditions which defeat most common tests. The development process tries to weed out most of these errors and the question is how do these methods perform on faults that have escaped to the field?

The idea of N-version programming is appealing from an academic perspective. However, the reality of development is such that developing one version with reasonable quality and budget is itself a challenge. Developing three or more is out of the question for most commercial development. It is true that the method has been employed in a few safety critical environments. In the commercial world its popularity is questionable.

The *retry* is an idea that has been used for several years. It is particularly effective in a multi-programming situation where the non-determinism of the system allows for certain programs to terminate without failure when retried. Jim Gray calls defects Heisenbugs that work without failure when the procedure is reexecuted. Clearly, this is a method that several operating systems have implemented and is one that will probably continue to be implemented for lack of better methods. Practitioners believe this is probably the most significant software fault-tolerance technique invented.

There are a whole class of user interface techniques that are evolving without any particular name, icon, or fanfare. These have to do with trying to comprehend the intended action of a customer and provide an interface which reacts as close as possible to what is desired. We are not talking about intelligent agents, but fairly simple methods. They do address a vast majority of the non-defect oriented problems that are arising. These have to do with context sensitive help, error checking on the input, taking natural defaults before quizzing a customer, providing information that is associated with an action and remembering the last set of defaults. Many of the problems that these methods tend to address would not have been considered as faults in the past. However, they do cause the large number of customer calls and addressing them clearly makes a product closer to customer expectation and avoids unproductive waste of time. Therefore, they are avoiding failures that result in customer downtime. These class of methods are not ordinarily fashioned under the fault-tolerance banner, but certainly are methods that need to be better understood and refined.

Reducing the threshold of pain in the client and desktop type of software is a different set of problems than reducing the pain in the host and network center computing environment. One has to understand the fault models that are relevant to the current state of technology in each of the market segments. The techniques have to be refined so that the resultant product achieves a higher productivity level with fault-tolerance than without.

4 Some “realities” in fault-tolerance

As technology advances, there has been a changing panorama in the need for fault-tolerance. We discussed issues in terms of the definition of failure and the kind of faults we need to tolerate. Coupled with this there is a technology element too. Hardware and software and have taken very different turns. From a semiconductor technology perspective the failure rate of components have come down by three orders of magnitude in five years. Whereas in software the failure rate in server software has remained the same [5]. This drives different priorities for product development.

At one extreme there is less of a perceived need for building fault tolerant hardware. While it may be true for certain segments of the market, it cannot be generalized in the server business where 24 by 7 availability is expected. From a software perspective, although the failure rate of software has remained constant, the hue and cry appears to occur only in a few market

segments. This is partly explainable by the discussion in Section 1. The other part of the explanation has to do with very few quantifiable measures in software.

With the result, the focus on software failure tends to be more of qualitative discussion and philosophical positioning. The exponential growth of client or desktop software where the belief that failures are not critical has also blunted the need for fault-tolerance. It has become evident that even the low quality of desktop software seems not to be critical to market success; whereas, its usability features are high priority.

It is, thus, easy to confuse a discussion on fault-tolerance with that of a product quality (defects). There is usually a time when the product quality is high enough that the remaining faults cause failures in a random enough fashion that one needs fault-tolerance methods to protect oneself against. Figure 3 illustrates a curve that is commonly used to describe the failure rate of software as a function of the remaining defects in a product. The shape of this curve has been recently re-validated for widely distributed commercial software. There is probably a level above which it is not realistic to think in terms of fault-tolerance. While below which it is worth the additional expense required to employ fault-tolerance techniques. Where exactly this cutoff occurs is hard to predict, but an easy concept to understand. Usually the cost of improving the product quality increases rapidly as one reaches higher and higher levels of product quality. The cost of fault-tolerance is reasonably high to begin with but becomes competitive as the standard development methods reach the point of diminishing returns.

If there is a fallacy in such an argument, it is that one cannot suddenly switch on fault-tolerance at any arbitrarily chosen time. It usually has to be designed into the product at concept. Unfortunately, this simple truth is usually beyond development organizations. While entering a new market, the focus is on time to market and barely on issues such as products quality or fault-tolerance. However, there does come a time when the product is growing and increasing in function rapidly. At such a time if the concepts of better development processes and fault-tolerance are ignored it often results in very inefficient development and low customer satisfaction.

There are several “realities” which compete for attention in a development program. These make the insertion of concepts in such fault-tolerance secondary and result in them being an afterthought [2]. This list is long. However, its worthwhile reviewing a couple of them.

Schedule pressure is probably the single largest driving force in most product development efforts. The pressure translates to a careful selection of features which need to be included into the current release. When a product is competing aggressively in a growing market, fault-tolerance does not tend to be the differentiating feature, always. In fact, it is rarely the differentiating feature in the client market. In the server market, it has a very respected standing and does compete for attention since it is a part of customer expectation.

What is often ignored but vital to the customer is service. Service is a collection of things that help the customer install, maintain and migrate applications to and from the software product. Often change management or maintenance is also included under the umbrella of service. Providing good serviceability to a product is vital. The fault tolerant community has not necessarily addressed issues relating to service or maintenance as much as it could. Firstly, diagnosing a software problem has always been a challenge. Today, in the network environment, it is worse. It is never clear where the source of trouble is and diagnosis requires skilled people and a lot of time. This does not help provide any customer satisfaction and is expensive on the part of the customer and the vendor. Getting good *first failure data capture* to diagnose and isolate is a challenge. The area of fault tolerant computing does not adequately focus on some

of these issues.

Understanding some of the issues and increasing awareness of broader academic communities is critical. There are no easy methods or convenient channels to do so. Often, the problems that dominate a development community are perceived as unattractive to the academic and vice versa. Yet, when one overcomes the initial barriers and looks deeply into these issues, they have very challenging problems that interest both. The past decade has seen a significant greater degree of interactions in several areas and cross pollination of ideas between research and industry. The fault-tolerant community, particularly in software, could benefit a lot more from such sharing.

5 Summary

This paper has attempted to address some of the basic issues that face software and reflect on what that means to fault-tolerance. Fundamentally, we need to recognize that our traditional views of failure need to be significantly revised. The paper proposes a re-definition to the concept of failure for software, based on customer expectation and ability to work productively. This is a significant departure from the classical view of a failure based on a specification and the delivery of service. The change has a significant implication on what the area of software fault-tolerance needs to address in the future.

The paper then illustrates some of the trends in software failures as reported by customers. It shows that 90% of the problems are from the traditional non-defect category, which usually did not come under the domain of being treated by fault-tolerance. In addition, the paper tries to develop a rationale for how much fault-tolerance is necessary in a market segment. We review where the major thought processes have been on software fault tolerance methods and critique where we ought to go. Finally, a few real life issues are raised, to sensitize us to the fact that bringing these strategies into a real product plan are not easy.

References

- [1] A. Avizienis, The N-version approach to fault-tolerant software, *IEEE Trans. Software Engineering*, vol SE-11, pp. 1491-1501, December 1985.
- [2] R. Chillarege, Top 5 challenges facing the practice of fault-tolerant computing, *Lecture Notes in Computer Science*, 774, Hardware and Software Architectures for Fault Tolerance, Springer-Verlag, 1994.
- [3] R. Chillarege, S. Biyani, J. Rosenthal, Measurements of failure rate in widely distributed software, *FTCS-25, Proceedings International Symposium on Fault-tolerant Computing*, June 1995.
- [4] R. Cramp, M. A. Vouk, and W. Jones, On Operational Availability of a Large Software-Based Telecommunication System, *Third International Symposium on Software Reliability Engineering*, pp. 358-366, 1992.
- [5] J. Gray, A Census of Tandem System Availability between 1985 and 1990, *IEEE Transactions on Reliability*, vol 39, no. 4, pp. 409-418, Oct, 1990.
- [6] K. Kanoun and T. Sabourin, Software Dependability of a Telephone Switching System. *Digest of Papers of The 17th International Symposium on Fault Tolerant Computing*, pp. 236-241, 1987.

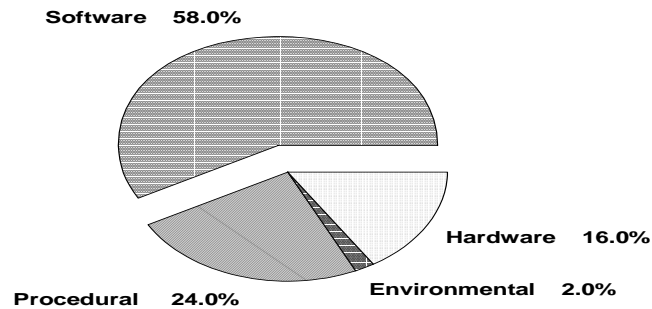


Figure 1: **Distribution of Outages.** Taken from several host server installations to show the preponderance of software related outages. A traditional view of failures.

- [7] J. C. Laprie, et.al., Dependability: Basic Concepts and Terminology, *Dependable Computing and Fault-Tolerant Systems*, Springer-Verlag, Vol 5, 1992.
- [8] J. C. Laprie, Dependability of Computer Systems: Concepts, Limits, Improvements, *ISSRE-6, International Symposium on Software Reliability Engineering*, pp 2-11, 1995.
- [9] B. Randell, System structure for software fault-tolerance, *IEEE Trans. Software Engineering*, SE-1, pp. 220-232, June 1975.

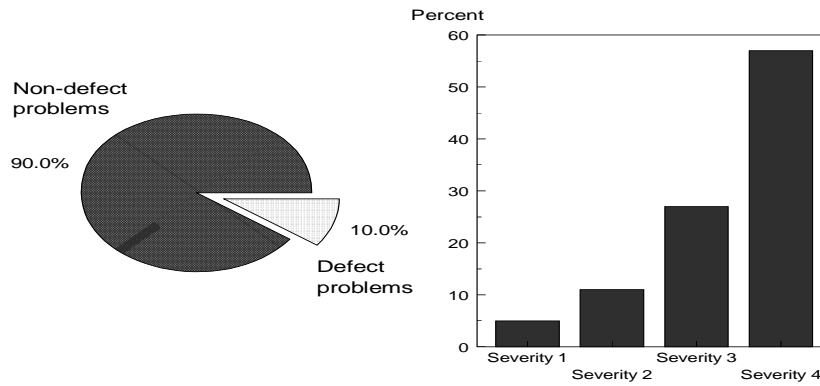


Figure 2: **Distribution of Problems.** The non-defect dominates the customer problems raising the need to focus on these kinds of failures. The traditional high severity (1) problems are less than 5%.

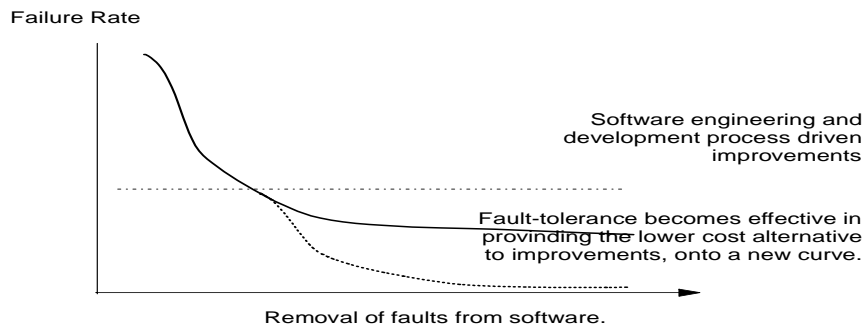


Figure 3: **Tradeoffs in reducing failure rate.** A conceptual model to illustrate when the strategy of fault-tolerance should provide a lower cost alternative. Additionally it may operate on a competitive curve while the software engineering and development process methods reach a points of diminishing returns.