SOFTWARE FAULT TOLERANCE FORESTALLS CRASHES: TO ERR IS HUMAN; TO FORGIVE IS FAULT TOLERANT

From: Advances in Computers Volume 58, Highly Dependable Software, edited by Marvin Zelkowitz, Academic Press, ISBN 0-012-012158-1, 2003, pp. 240- 285.

Abstract

Software Fault Tolerance prevents ever-present defects in the software from hanging or crashing a system. The problem of preventing latent software faults from becoming system failures is the subject of this chapter. Software architectures, design techniques, static checks, dynamic tests, special libraries, and run-time routines help software engineers create fault tolerant software. The nature of software execution is chaotic because there are few ways to find singularities, and even those are rarely practiced. This leads to complex and untrustworthy software products.

The study of software fault tolerance starts with the goal of making software product available to users in the face of software errors. Availability is a mathematical concept that is the Mean Time-To-Failure divided by the Mean Time-To-Failure plus the Mean Time-To-Repair. The idea is to make the Mean Time-To-Failure as large as possible and the Mean Time-To-Repair as small as possible. Continuing with Reliability Theory we can express the Mean Time-To-Failure as the reciprocal of the failure rate. Assuming an exponential reliability model the failure rate is the expected value of the reliability of the system. This chapter shows how these quantitative concepts can be used to make software engineering tradeoffs.

First, there is a historical perspective on and problems with the study of software fault tolerance. Then new approaches are presented with a theme of making it possible to trade-off software execution time, complexity, staff effort, and the effectives of the staff to achieve desired system availability.

SOFTWARE FAULT TOLERANCE FORESTALLS CRASHES: TO ERR IS HUMAN; TO FORGIVE IS FAULT TOLERANT

Lawrence Bernstein
Industry Research Professor
Stevens Institute of Technology
Castle Point, Hoboken NJ 07030

Table of Contents

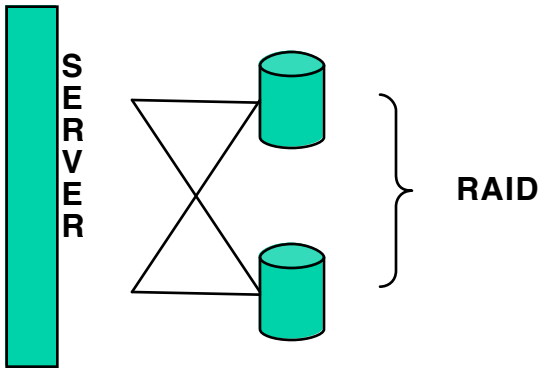
1 Background

1.1 Fault Tolerant Computers

The 1990s were to be the decade of fault tolerant computing. Fault tolerant hardware was in the works and software fault tolerance was imminent. But it didn't happen. Hardware suppliers successfully convinced customers that highly reliable configurations with duplicate disks in RAID configurations as shown in Figure 1 were good enough. The most popular computer suppliers did not have fault tolerant configurations to offer their customers. To keep their market share they convinced their customers that the incremental value of a hardware fault tolerant configuration did not warrant the extra costs. In addition, they radically reduced the costs of their high reliability configurations. Fault tolerant computers became twice as expensive as high reliable ones. Since the application software was not fault tolerant the incumbent supplies argued the new computers were not cost effective. Except in some special cases their arguments carried the day. The incumbent suppliers eventually absorbed the new companies that made fault tolerant computers. In the 1990s, the Web Wave surged. Highly reliable server hardware configurations became the solution of choice. Software failures were not addressed. Software developers lost interest in fault tolerance

until a rash of server failures, denial of service episodes, web outages and the September 11th attacks.

Figure 1 High Reliability STORAGE SERVER



Storage Server:
- RAID: Redundant Array of Independent Disks
6-gigabytes/sec transfer
20 gigabytes/drive

- Disk files are mirrored.

- When a sector error occurs the RAID device driver falls over to the backup copy.

Software fault tolerance methods are often extrapolated from hardware fault tolerance concepts. This approach misses the mark because hardware fault tolerance is aimed at conquering manufacturing faults. Environmental and other faults are rarely treated. Redundant hardware subsystems solved many single errors with extended operating systems programmed to recognize the hardware failure and launch the application on the working hardware. Design diversity was not a concept usually applied hardware fault tolerance. Software fault tolerant designers, however, adopt this approach by using N-version (also called multi-version) programming.

1.2 Why Software is Different from Hardware

The N-version concept attempts to parallel in software the hardware fault tolerance concept of N-way redundant hardware. In an N-version software system, each module is made with up to N different implementations. Each version accomplishes the same task but in a different way. Each version then submits its answer to a decider that determines the correct answer and returns that as the result of the module. In practice, this means that more than one person must work a module in order to have different approaches. If each version but one succumbed to the same conceptual error, the one genius version would be cancelled out even though it alone was correct. This approach works only when it is possible to create uncorrelated yet equivalent designs and that the resulting programs do not share similar failure modes. Design diversity with independent failure modes are hard to achieve. Nancy Leveson points out, " …every experiment with [multi-version programming] that has checked for dependencies between software failures has found that independently written software routines do not fail in a statistically independent way [Leve95]. The techniques of N-version programming are well treated by Michael Lyu [Lyu00].

Lui Sha shows that "…the key to improving reliability is not the degree of diversity, per se. Rather the existence of simple and reliable components ensures the system's critical functions despite the failure of non-core software components." He continues to say, "…diversity with N-version programming does not usually improve reliability and points out that FAA DO 178B discourages its use" [Sha00].

An alternative to N-version is the use of recovery blocks. Transactions are closely monitored so that if there is a failure during the execution of any transaction, the software can be rolled back to a previously sound point. Then special recovery software is executed based on the needs of the application. The failed transaction can be dropped allowing the system to execute other transactions, or the transaction may be retried and, if it is not successful within some number of attempts, the entire system may be halted. If the loss of any transaction is unacceptable, the database may be secured and the system halted. The database might be rebuilt by special recovery software and then the system could be restarted. Older recovery blocks executed several alternative paths serially until an acceptable solution emerged. Newer recovery block methods may allow concurrent execution of various alternatives. The N-

version method was designed to be implemented using N-way hardware concurrently. In a serial retry system; the cost in time of trying multiple alternatives may be too expensive, especially for a real-time system. Conversely, concurrent systems require the expense of N-way hardware and a communications network to connect them. Software uses a software module to decide that may consider more than the absolute results to determine the action to take. Hardware uses strict comparison of results. The recovery block method requires that each module build a specific decider. This requires a lot of development work. The recovery block method creates a system that makes it is difficult to enter into an incorrect state, if the programmer can design simple and bug free decision code. In comparison the N-version method may use a single decider.

Hardware faults are predominantly caused when component performance degrades. Design faults are found and fixed early. In a few cases hardware design failures appear in the final system. When they do, it is often left to the software to compensate for them. Software faults are different; they result from design shortcomings [Storey96]. Techniques for dealing with such shortcomings are detailed later in this chapter. Software manufacturing is the process of configuration management, trouble tracking and reproduction of software [Bern89]. Errors in this process, unlike the hardware-manufacturing situation can be entirely prevented by appropriate tools, technician education and standards. Errors in software manufacturing are not part of software fault tolerance studies. For the purposes of this chapter, the software is assumed to be accurately reproduced and installed on the target host computer.

Software is characterized by branching, executing alternative series of commands based on input. This quality creates complexity. Even short programs can be very difficult to fully understand. Software branching can contain latent faults which only come to light long after a software product has been introduced into the marketplace.

Typically, testing alone cannot fully verify that software is complete and correct. In addition to testing, other verification techniques and a structured and documented development process must be combined to assure a comprehensive validation approach.

In their work on “Quantitative Analysis of Faults and Failures in a Complex Software System” N.E. Fenton and N. Ohlsson describe a number of results from a quantitative study of faults and failures in two releases of a major commercial system. They tested a range of basic software engineering hypotheses relating to: the Pareto principle of distribution of faults and failures; the use of early fault data to predict later fault and failure data; metrics for fault prediction; and benchmarking fault data. They found very strong evidence that a small number of modules contain most of the faults discovered in pre-release testing, and that a very small number of modules contain most of the faults discovered in operation. However, in neither case is this explained by the size or complexity of the modules. They found no evidence relating module size to fault density, nor did they find evidence that popular complexity metrics are good predictors of either fault-prone or failure-prone modules. They

confirmed that the number of faults discovered in pre-release testing is an order of magnitude greater than the number discovered in one year of operational use. They also discovered stable numbers of faults discovered at corresponding testing phases. Their most surprising and important result was strong evidence of a counter-intuitive relationship between pre and post release faults: those modules which are the most fault-prone pre-release are among the least fault-prone post-release, while conversely the modules which are most fault-prone post release are among the least fault-prone pre-release. This observation has serious ramifications for the commonly used *fault density* metric. Not only is it misleading to use it as a surrogate quality measure, but also its previous extensive use in metrics studies is flawed.

Another related characteristic of software is the speed and ease with which it can be changed. This factor can lead to the false impression that software faults can be easily corrected. Combined with a lack of understanding of software, it can lead engineering managers to believe that tightly controlled engineering is not as much needed for software as for hardware. In fact, the opposite is true. Because of its complexity, the development process for software should be tightly controlled. The goal is to prevent problems that will be hard to find later.

Software may improve with age, as latent defects are discovered and removed. Repairs made to correct software defects, in fact, establish a new design. Seemingly insignificant changes in software code can create unexpected and very significant problems elsewhere. Musa's execution time model takes advantage of data gathered during software testing [Musa 87]. His approach extends the fundamental reliability exponential model that can be used at all stages of the software development process. Musa's approach is best used during a carefully designed set of longevity and stress tests. Test cases typically reflect samples of user operations. The key idea is to find how long a system will operate before it fails and the operational impact when it does fail. One way to estimate the intrinsic reliability of the software is to examine charts showing tests passed as a function of time. The more reliable the software the more this plot following a typical human skills acquisition graph, with plateaus during which competence is acquired followed by sudden jumps in capability [Bern93].

Software validation is a critical tool in assuring product quality for device software and for software automated operations. Software validation can increase the usability and reliability of the device, resulting in decreased failure rates, fewer recalls and corrective actions, less risk to patients and users, and reduced liability to manufacturers. Software validation can also reduce long term costs by making it easier and less costly to reliably modify software and revalidate software changes. Software maintenance represents as much as 50% of the total cost of software. An established comprehensive software validation process helps to reduce the long term software cost by reducing the cost of each subsequent software validation.

1.3 Software Errors (Bugs)

In contrast to hardware, software faults are most often caused by design shortcomings that occur when a software engineer either misunderstands a specification or simply makes a mistake. Design for reliability is rarely taught to Computer Science majors. Software faults are common for the simple reason that the complexity in modern systems is often pushed into the software part of the system. Then the software is pushed to and beyond its limits. It is estimated that 60-90% of current computer errors are from software faults. [Gray91] Software faults may also be triggered from hardware; these faults are usually transitory in nature, and can be masked using a combination of current software and hardware fault tolerance techniques. In the 1960s, I managed a software project that tracked missiles. As we prepared for the first missile test, managers argued about the wisdom of using special fault tolerant hardware and software subsystems called 'Mission Mode'. The problem was that these subsystems were not fully tested. One manager argued using the alternative the 'stop on error' approach there would be no chance to recover from an error. Another argued, "Untested software will not work." Which would you pick? The executive in charge chose the logically correct "Mission Mode' approach. For safety reasons the missile was equipped with a countdown timer that needed to be reset periodically. If the timer fell to zero a special onboard explosive would fragment the missile so that it would not fall in populated areas. After 30 seconds of a planned 90 flight, the clock was not properly reset. The missile blew up. The computer skipped the instruction that updated the position of the electronically steered phase array radar. There was no inertia to keep the radar pointing in the general direction of the missile. A six-week investigated showed that the timing changes had negative timing margins when run in certain configurations peculiar to Mission Mode with a particular combination of data. Even though there had been hundreds of tests run before the mission the hardware and software were run in that configuration for the first time 30 seconds into the mission. During the investigation, the problem could not be reproduced and all missile testing stopped. Some wanted to proceed with more missions before the problem was solved, claiming that the problem was a random occurrence. Some twenty-five years later AT&T experienced a massive network failure caused by a similar problem in the fault recovery subsystem they were upgrading. This time there was a latent software fault in the update. It became a failure. There was not enough testing of the recovery subsystem before it was deployed. In both cases, the system failed because there was no limits placed on the results the software could produce. There were no boundary conditions set. Designers built with a point solution in mind and without bounding the domain of software execution. Testers were rushed to meet schedules and the planned fault recovery mechanisms did not work.

While software cannot be designed without bugs, it does not have to be as buggy as it is. For example, as early as 1977, a software based store and forward message switching was in its fourth year of operation and it handled all administrative messages for Indiana Bell without a single failure. This record was achieved after a very buggy start followed by a substantial investment in failure prevention and bug fixes. One particularly error-prone software subsystem was the pointers used to account for clashes in the hash function that indexed a message data file. The messages could remain in the file system for up to thirty days. There were many hash clashes due to the volume of messages and the similarity of their names.

Once the obvious bugs were fixed the residual ones were hard to find. This led to unexpected behavior and system crashes. A firm requirement was not to lose any messages. Failures exhibited by latent faults can appear to be random and transient. But they are predictable if only we can get the initial conditions and transaction load to trigger them. They are sometimes called Heisenbugs. It was just too costly to find and fix all the Heisenbugs in the file index code, so the hash tables were rebuilt daily in the early hours of the morning when there was no message traffic. With fresh hash tables daily, the chances of triggering a fault was small especially after the bugs that were sensitive to the traffic mix were found and fixed. This experience shows that it is not necessary for software to be inherently buggy.

Software runs as a finite state machine. Software manipulates variables that have states. Unfortunately flaws in the software that permit the variables to take on values outside of their intended operating limits often cause software failures. Software also fails when coded correctly, but the design is in error. Software can fail when the hardware or operating systems are changed in ways unanticipated by the designer. Also, software often fails when users overload it. In the final analysis, most failures are due to some human error. While human error is difficult to predict, it is predictable. The work of John Musa for example, which for years has dealt with the predictability of software faults, shows that the behavior of many software failures fit a mathematical model.

1.4 Application Concerns

When some service is especially critical or subject to hardware/network failure, the application designer needs to build software fault tolerance into the application. These are typical issues facing application designers:

- *Consistency:* In distributed environments, applications sometimes become inconsistent when code in a host is modified unilaterally. For example, the code in one server software component may be updated and this change may require sending out new versions of the client application code. In turn, all dependent procedures must be re-compiled. In situations where a single transaction runs across several servers a two-phase commit approach may be used to keep the distributed databases consistent. If the clients and servers are out of step there is a potential for a failure even though they have been designed and tested to work together. The software in the hosts need to exchanged configuration data to make sure they are in lock step before every session.

- *Robust Security:* Distributed application designers need to ensure that users cannot inadvertently or deliberately violate any security privileges.

- *Software Component Fail Over:* The use of several machines and networks in distributed applications increases the probability that one or more could be broken.

> The designer must provide for automatic application recovery to bypass the outage and then to restore the complex of systems to its original configuration. This approach contains the failure and minimizes the execution states of the complex of systems.

BEA pioneered industrial strength two-phase commit middleware in their Tuxedo product and applications developed using Oracle databases address many of the complex reliability and security considerations that affect partitioned, distributed applications. BEA's Application Infrastructure platform [BEA02] implements each layer of the application infrastructure as a single, well-architect solution. The platform simplifies software engineering by providing an integrated framework for developing, debugging, testing, deploying and managing applications. The platform is

a. Reliable – ensuring that applications never "break," even under the most demanding circumstances
b. Available – enabling applications that can run continuously 24x7x365
c. Scalable – allowing companies to plan cost-effectively for any level of usage
d. Trusted – because in today's world of heightened security, an enterprise must maintain complete control of its data

In information technology – communications no less than computing – software often is the machine. It's been said that it takes 100 million lines of code to make a typical phone call. Some analysts estimate that the global telecommunications industry has roughly 100 billion lines of software code in products or in development. Any of these may be a bug. Across the board, there will be increasing demands for reliability on a level seldom-encountered outside telecommunications, defense and aerospace. Customers want future Internet services to be as reliable and predictable as services on yesterday' voice networks.

Software fault tolerance is at the heart of the building trustworthy software. Today Microsoft is embarking on a major Trustworthy Computing initiative. Bill Gates sent a memo to his entire workforce demanding, " … company wide emphasis on developing high-quality code that is available, reliable and secure- even if it comes at the expense of adding new features." Information Week, Jan. 21 2002,issue 873, p.28. Trustworthy software is stable. It is sufficiently fault-tolerant that it does not crash at minor flaws and will shut down in an orderly way in the face of major trauma. Trustworthy software does what it is supposed to do and can repeat that action time after time, always producing the same kind of output from the same kind of input. The National Institute of Standards and Technology (NIST) defines trustworthiness as "software that can and must be trusted to work dependably in some critical function, and failure to do so may have catastrophic results, such as serious injury, lost of life or property, business failure or breach of security. Some examples include software used in safety systems of nuclear power plants, transportation systems, medical devices, electronic banking, automatic manufacturing, and military systems." [Wall 94]

1.5 Origins of Software Engineering

NATO convened a meeting in 1968 to confront a crisis that didn't make the headlines – "the software crisis." Experts from a dozen countries, representing industrial labs as well as universities, met in Garmisch, Germany, to grapple with two basic questions that are still with us: Why is it so hard to produce functionally correct and **reliable** software that meets users' needs and performance requirements and comes in on time and within a projected budget? And where should software producers be looking for solutions? The answers revolved around "software engineering," a term coined by the organizers of the Garmisch conference – somewhat controversial at the time, and wholly inspirational – to focus on a missing discipline with the potential to resolve the crisis. Software production should be "industrialized." Systems should be built from reliable components.

There are significant advances in Software Engineering. Programmers are better now, most code is written in high level languages, better tools exist, development is done on-line, better design models exist, and standards have been established in some key areas. A number of recently developed or recently successful innovations include libraries for fault tolerant computing, object-oriented programming, remote procedure calls that are the foundation for client-server applications, application programming interfaces, graphical user interfaces and development tools, prototyping, and source-level debugging. Components exist and are widely used. This belies the common wisdom that software components are rare. The C libraries with all their richness of fundamental or atomic functions provide 20% reuse in most industrial strength UNIX based applications. Software that makes a library of graphical elements, or a text-processing tool, available to multiple applications – with a major reservation is also in wide use. IBM and others are using software libraries providing generic fault avoidance and recovery.

2 Fault Tolerance is related to Reliability Theory

A fault is an erroneous state of software and fault tolerance is the ability of the software system to avoid execution the fault in a way that causes the system to fail. "The reliability of a system as a function of time R(t), is the conditional probability that the system has (not failed) in the interval [0,t], given that it was operational at time t=0 [Siew82]. " Therefore it is essential to examine software reliability to understand software fault tolerance.

The most common reliability model is:

$$R(t) = e^{-\lambda t},$$

where λ is the failure rate. It is reasonable to assure that the failure rate is constant even though faults tend to be clustered in a few software components. The software execution is very sensitive to initial conditions and external data driving the software. What appear to be random failures are actually repeatable. The problem in finding and fixing these problems is

the difficulty of doing the detective work needed to discover the particular initial conditions and data sequences can trigger the fault so that it becomes a failure. [Musa87].

In a two-state continuous-time Markov chain the parameters to be estimated are failure rate λ and repair rate μ.

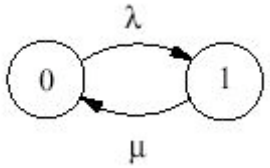


Figure 2. Two-state Reliability Model

The Mean Time Between Failures (MTTF) = $1/\lambda$.

The Mean Time To Repair (MTTR)= $1/\mu$.

The steady-state availability is:

Availability = MTTF/(MTTF +MTTR) = $1/[1 + \lambda/\mu]$

The goal of Software Fault Tolerance is to make Availability = 1.

This can be approached by making λ very small and/or by making μ very large. The rest of this chapter describes how to accomplish both using a unified equation tying the practices, processes, algorithms and methods of software engineering together. With this unifying equation software engineering tradeoffs may be quantitatively made.

Software Fault Tolerance in the large focuses on failures of an entire system, whereas Software Fault Tolerance in the small, usually called transaction recovery, deals with recovery of an individual transaction or program thread. The system MTTF is usually greater for the system than for any transaction as individual transactions may fail without compromising other transactions. Telephone switching systems employ this strategy by aborting a specific call in favor of keeping the remaining calls up. Sometimes the integrity of the data is so important that the system tires to rerun a transaction or recover lost data as part of an error recovery strategy. This recovery software operates for one transaction while the rest of the transactions operate normally. If there are too many transaction failures the system may need to be restarted to clear a data problem or reinitialize control flows. This is a fault becoming a failure. The MTTR usually addresses the recovery of a transaction but with the software rejuvenation technique described in this paper the re-initialization of the software execution states can be considered the MTTR. With this function built in the system there is no system failure, but transaction executions may be delayed.

2.1 Sha's Reliability Model

Professor Lui Sha's of University of Illinois at Urbana-Champaign created a model of reliability based on these postulates:

1. Complexity begets faults. For a given execution time software reliability decreases as complexity increases.
2. Faults are not equal, some are easy to find and fix and others are Hisenbugs. Faults are not random.
3. All budgets have limits so that there is not unlimited time or money to pay for exhaustive testing

Sha chooses the MTTF = E/kC and the reliability of the system is $R(t) = e^{-k\,Ct/E}$, where

k is a scaling constant,

C is the complexity, where Sha defines complexity as the effort needed to verify the reliability of a software system made up of both new and reused components,

t is the continuous execution time for the program,

E is the development effort that can be estimated by such tools as Checkpoint, COCOMO or Putnam's approach. Development effort is a function of the complexity of the software so the numerator and denominator of the exponent must be factored and

R(0) = 1 because all the startup failures are assumed to be removed through classical unit, block and system testing.

Unfortunately we cannot use these equations to compute the reliability of a system from its components because failure modes may depend on a particular combination of the software components. Sha's approach works for a system as a whole. Until software components can be isolated and their execution characteristics made independent of one another the reliability of an entire system is **not** the product of the reliability of its component parts.

2.2 Effectiveness Extension of Reliability Model

An extension of the model adds an effectiveness factor to the denominator. This reflects the investment in software engineering tools, processes and code expansion that makes the work of one programmer more effective. Let α be the expansion factor that expresses the ability to solve a program with fewer instructions with a new tool such as a complier. It may be the ratio of the number of lines of code a complier may expand into machine code. This idea of the improvement in productivity due to investment in the software tools has been explained in an earlier article [Berns97]. Then the effectiveness equals the conciseness factor and the reliability equation becomes:

$R = e^{-k\,Ct/E\alpha}$

This equation expresses reliability of a software system in a unified form as related to software engineering parameters. The longer the software system runs the lower the reliability and the more likely a fault will be executed to become a failure. Reliability can be improved by investing in tools (), simplifying the design (C), or increasing the effort in development to do more inspections or testing than required by software effort estimation techniques. The estimation techniques provide a lower bound on the effort and time required for a successful software development program. These techniques are based on using historical project data to calibrate a model of the form:

Effort = a + b (NCSLOC)

Where a and b are calibration constants,

NCSLOC is the number of new or changed source lines of code needed to develop the software system

_ is an exponent expressing the diseconomies of scale for software projects and is greater than 1 [Boehm00].

An elaboration of the Sha model provides the foundation for a general theory of reliability-based software engineering. Various software engineering processes are combined to understand the reliability of the software; by increasing reliability, the software becomes more fault tolerant. Consider the reliability equation term by term: C, t, and E.

2.3 Complexity factors (C)

2.3.1 Complexity

Sha states that the primary component of complexity is the effort needed to verify the reliability of a software system. Typically reused software has less complexity than newly developed software because it has been tested in the crucible of live operation [Lim94]. But, this is just one of many aspects of software complexity. Among other aspects of software engineering complexity a function of [Stoyen97]:

a. The nature of the application characterized as
    1. real-time, where key tasks must be executed by a hard deadline or the system will become unstable or software that must be aware of the details of the hardware operation. Operating systems, communication and other drivers are typical of this software. Embedded software must deal with all the states of the hardware. The hardest ones for the software engineer to cope with is the ‘don’t care states.’
    2. on-line transactions, where multiple transactions are run concurrently interfacing to people or to other hardware. Large database systems are typical of these applications.
    3. Report generation and script programming
b. the nature of the computations including the precision of the calculations,

c. the size of the component,
d. the steps needed to assure correctness of a component,
e. the length of the program,
f. and the program flow.

Sha's view is that the effort to verify the correctness of the component factor dominates the length factor. He assumes that reasonable design steps have been taken and the there are no singularities in the execution of the component. Cyclomatic metrics are used effectively to find components that are orders of magnitude more complex than others in the system. Trustworthy system design demands that these components be redesigned. In any event, by reducing complexity or simplifying the software, the reliability increases.

2.3.2 Trustworthy Software is Reliable

First there must not be any degenerate situations in the implementation of the software. A few causes for such degeneration are:

1. computational lags in controlling external equipment,
2. round-off errors in computing control commands,
3. truncation errors induced when equations are approximated,
4. memory leaks that prevent other processes from executing memory,
5. thrashing, and
6. buffer overflows.

Reliable software is trustworthy software. It is easier to make simple software reliable. Trustworthiness is the ideal. It is confounded by what insurance companies call "acts of God" in the environment, by human beings who misunderstand or ignore or circumvent system warnings, by chaotic software conditions that arise from untested areas, by the pressures of the marketplace to sell new versions of software that invalidate earlier versions, and by malevolent attacks from people. The interplay among these dimensions of instability is complicated.

Software system development is frequently focused solely on performance and functional technical requirements and does not adequately address the need for reliability or trustworthiness in the system. Not only must software designers consider how the software will perform they must account for consequences of failures. Trustworthiness encompasses this concern.

"Trustworthiness is a holistic property, encompassing security including confidentiality, integrity, and availability, correctness, reliability, privacy, safety and survivability. It is not sufficient to address only some of these diverse, dimensions, nor is it sufficient to simply assemble components that are themselves trustworthy. Integrating the components and understanding how the trustworthiness dimensions interact is a central challenge in building a

trustworthy Networked Information System." This definition appeared in a fine article by Fred Schneider, Steven Bellovin and Alan S. Inouye in the November/December issue of IEEE Internet Computing, page 64[http://computer.org/internet/]. The article discusses many aspects of network operation but takes for granted the trustworthiness of the underlying software. Because of the increasing complexity and scope of software, its trustworthiness will become a dominant issue.

Modern society depends on large-scale software systems of astonishing complexity. Because the consequences of failure in such systems are so high, it is vital that they exhibit trustworthy behavior. Much effort has been expended in methods for reliability, safety and security analysis, as well as in methods to design, implement, test and evaluate these systems.

Yet the "best practice" results of this work are often not used in system development. A process is needed to integrate these methods within a trustworthiness framework, and to understand how best to ensure that they are applied in critical system development. It is critical that we focus attention on critical systems and to understand the societal and economic implications of potential failures.

Trustworthiness is already an issue in many vital systems, including those found in transportation, telecommunications, utilities, health care and financial services. Any lack of trustworthiness in such systems can adversely impact large segments of society, as shown by software-caused outages of telephone and Internet systems. It is difficult to estimate the considerable extent of loses experienced by individuals and companies that depend on these systems. This issue of system trustworthiness is not well understood by the public. One measure of the Trustworthiness of a system is its stability.

2.3.3 Software Stability is key to Simplicity

Internal software stability means that the software will respond with small outputs to small inputs. If the software response grows without bound, the system will usually crash of hang. These are the most egregious failures. All systems have latent faults that can cause system failures. The trick is to use feedback control to keep system execution away from these latent faults or singularities so that the faults do not become failures. Most instabilities internal to the software are caused by:

- buffer usage that increases to eventually dominate system performance,
- computations that cannot be completed before new data arrive,
- round-off errors that build
- an algorithm that is embodied in the software is inherently flawed.
- memory leaks
- register overflow
- thrashing access to files on secondary storage

- thrashing of routes in a network
- broadcast storms

There is only a weak theoretical model for software behavior. Chaotic conditions can arise over time outside the windows of testing parameters. There is little theory on dynamic analysis and performance under load. The question is how to compensate for destabilizing factors and catch problems early in the design process. The Federal Food and Drug Administration has issued a paper on general principles of software validation that remarks, "Software verification includes both static (paper review and automatic programs that process the source code such as Lint) and dynamic techniques. Dynamic analysis (i.e., testing) is concerned with demonstrating the software's run-time behavior in response to selected inputs and conditions. Due to the complexity of software, both static and dynamic analysis is needed to show that the software is correct, fully functional and free of avoidable defects.

2.3.4 Buffer Overflows

A significant issue facing software engineers is keeping buffers from overflowing. Buffers are used to allow computer processors to multithread processes and by TCP to exchange data between computers. When control programs are fast buffer areas can be small, otherwise large buffers must be available so that they do not overflow and cause the software to stop exchanging data or to transfer to undefined locations causing the system to hang or crash and thus fail.

The exploitation of buffer overflow bugs in process stacks cause many security attacks. To deal with the problems there are new libraries that work with any existing pre-compiled executable and can be used system-wide.

The Microsoft®.NET Framework eliminates security risks due to buffer overflows; and shifts the burden from having to make critical security decisions—such as whether or not to run a particular application or what resources that application should be able to access—from end users to developers . With the ever-increasing complexity and functionality of software applications comes assaults. The managed code architecture of the .NET Framework provides one solution to the problem of software application security. It transparently controls the behavior of code even in the most adverse circumstances, so that the risks inherent in all types of applications— client- and server-side—are greatly reduced. A *common language runtime* (CLR) is the engine that runs and "manages" executing code enforcing .NET Framework's restrictions and prevents executing code from behaving unexpectedly. The CLR performs "just-in-time" compilation and inserts control code in the execution module. The .NET *Framework class libraries* are a collection of reusable classes,

or types, that developers use to write programs that will execute in the common language runtime. An *assembly* is an executable compiled using one of the .NET Framework's many language compilers. Assemblies contain *metadata*, which the CLR uses to locate and load classes, lay out instances in memory, resolve method invocations and generate native code. It sets runtime context boundaries. The *verification process* ensures the runtime safety of managed code. During JIT compilation, the CLR verifies all managed code to ensure memory type safety. This eliminates the risk of code executing or provoking “unexpected” actions that could bypass the common application flow and circumvent security checks. The verification process prevents common errors from occurring. It does not allow integer pointers to access arbitrary memory locations. It does not allow access to memory outside the object boundary or accessing method outside its class. It does not allow access to newly created objects before they have been initialized. It prevents buffer overflows. These common programming faults no longer pose a threat within the type safe, managed environment provided by the .NET Framework. .NET cannot be used to solve all problems and many applications cannot honor the .NET restrictions.

Another approach is to use special library processes to intercepts all calls to other library functions known to be vulnerable. A safe version of the buffer management code then ensures that any buffer overflows are contained within the invoked stack frame. A second approach is to force verification of critical elements of stacks before use. Performance overhead of fault tolerant libraries implementing these safe approaches range from negligible to 15%. This method does not require any modification to the operating system and works with existing binary programs. It does not require access to the source code of defective programs, nor does it require recompilation or off-line processing of binaries. It can be implemented on a system-wide basis transparently. It is based on a middleware software layer that intercepts all function calls made to library functions that are known to be vulnerable. A substitute version of the corresponding function implements the original functionality, but in a manner that ensures that any buffer overflows are contained within the current stack frame, thus, preventing attackers from 'smashing' (overwriting) the return address and hijacking the control flow of a running program. It is a Linux dynamically loadable library called *libsafe*. Figure 3 is a description of the library [Barat00].

Figure 3: Libsafe: Protecting Critical Elements of Stacks

Overview

It is generally accepted that the best solution to buffer overflow and format string attacks is to fix the defective programs. However, fixing defective programs requires knowing that a particular program is defective. The true benefit of using libsafe and other alternative security measures is protection against future attacks on programs that are not yet known to be vulnerable. That is libsafe version 2.0 source code is under the GNU Lesser General Public License.

In contrast to most other solutions, libsafe is extremely easy to install and use. No source code, recompilation or special expertise is needed. And, the installation only takes a few minutes.

Libsafe does not support programs linked with libc5. If you find that a process protected by libsafe experienced a segmentation fault, use the *ldd* utility to determine if the process is linked with libc5. If that is the case, then you will either need to recompile/relink the application with libc6 (i.e., glibc) or to download a newer version that has been linked with libc6 although most applications are offered with a libc6 version.

NOTE: The latest release of libsafe is version 2.0-11, released on 02-28-02

Buffer overflow bugs can cause a large transfer of data to a buffer, overflowing it, and then overwriting the memory. A hacker can inject additional code into an unsuspecting process and hijack control of that process by overwriting return addresses on the process stack or by overwriting function pointers in the process memory. There are many error-prone functions in the Standard C Library. Here are a few examples of these functions:

Function	Potential problem
strcpy(char *dest, const char *src)	May overflow the destination buffer.
gets(char *s)	May overflow the s buffer.
realpath(char *path, char resolved path[])	May overflow the path buffer
scanf(const char *format, ...)	May overflow its arguments

Nicolas Wirth in the family of intermediate level languages defines C in his paper PL/360. These languages differ from Higher Level Languages in that they give programmers access to the machine registers and architecture when they need it. The compilers for these languages differ from assembly languages in that they provide compactness of expression. This ability along with the C libraries made C industrial strength. It is widely used today and laid the foundation for C++. One can even see the influence of C on Java. [Bell84]

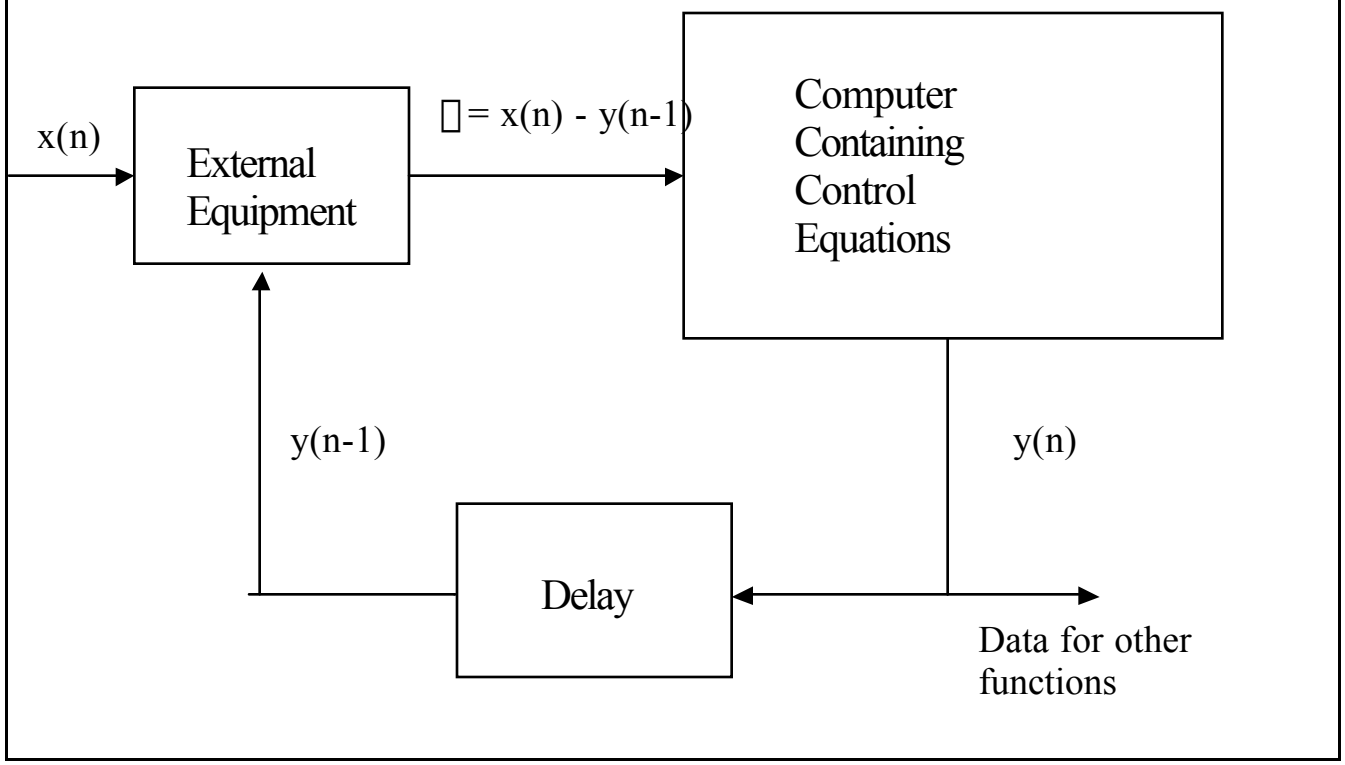
The importance of C is that it led to remarkable improvement in programmer productivity by a factor of 3:1 over assembler language development without giving up the key ability to solve complex software problems or meet harsh performance requirements in industrial strength production systems. Because of its inherent flexibility, C is vulnerable to misuse. Higher-level languages protect programmers are from the architecture of the machine. If they

need to gain significant performance improvement they must drop down to the machine level. This leads to a discontinuity in the development environment and makes it likely that more code than desired will be low-level code. It is very hard for skilled software engineers to switch between development paradigms. Furthermore, tools are need to manage two and sometimes more languages and the issue of assembler vs. complier permeates many design designs within the project. Control of software changes becomes more difficult too. Today's Java-hype as a total C++ replacement still is unproven. Java is a fine language, but when industrial strength programs are needed C++ with its ability to naturally drop down to C is the industrial software engineer's choice.

2.3.5 Small and bounded Time Lags are Critical

Sometimes software components embedded in a system must respond to the environment within some time interval. If the system fails because the time constraints are not satisfied the system is a real-time one. If the response time becomes unacceptably long the system is an on-line one. In either case successful performance of the software demands that the computations are completed in the required time. The feedback characteristics of these systems often dominate, as computation results must be available in sufficient time to affect some external process. Feedback operation and meeting deadlines are two key attributes of embedded software. Systems containing a computer as one of their elements belong to the class of sampled data systems. A typical feedback control system is shown in figure 4,

Figure 4. Feedback control system



where

x(n) is the physical input to the external equipment that is sampled and encoded with a certain number of bits or precision.

y(n) is the computer output from the computers control equations using the error from the previous sampled time as the input.

The objectives of the control equations are to:

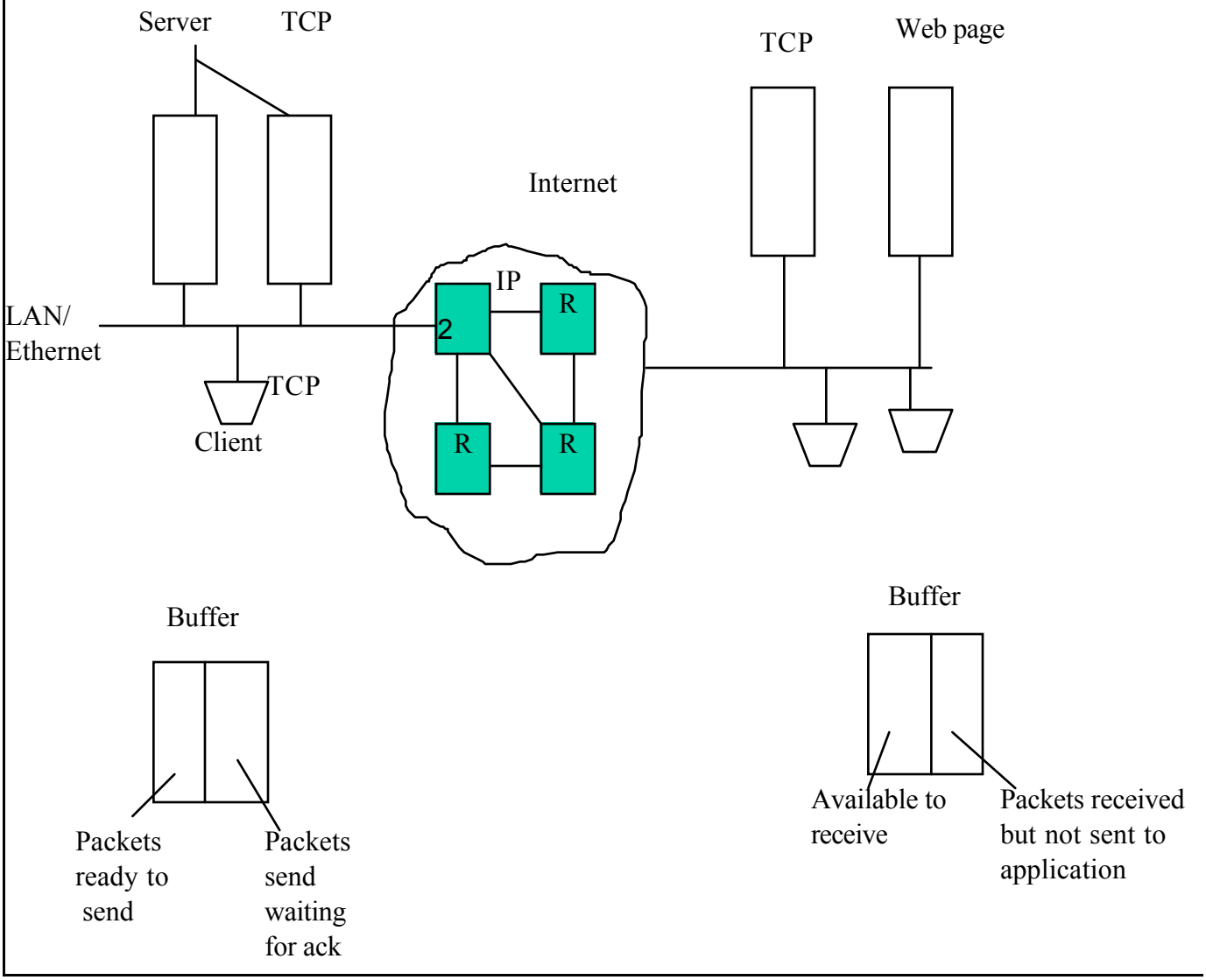
- achieve a satisfactory level of reliability in system operation. The outputs of the system need to be bounded to match physical properties of the equipment being controlled,
- be easy to initialize and that the initial conditions lead to stable software execution,
- calculations should be easy to implement and quick to process,
- occupy a minimum amount of memory that suggests using recursive filters such as those use in the Transaction Control Protocol (TCP) computation of roundtrip time used for flow control in Internet applications,
- avoid duration-related memory leaks and fragmentation can degrade execution time, often causing reduced throughput and eventual system failure.

A case history of a mechanism all web users employ daily illustrates these points.

2.3.6 Case study: TCP Timer for Resend

TCP uses an Automatic Request Response window with selective repeat to control the flow of packets between the sender and the receiver. The buffer size of the receiver and the bandwidth-delay product of the network typically limit the window size. Buffers may overflow. The lowest capacity link on the route becomes the bottleneck. The goal is to make the window as large as possible to gain the best network throughput consistent with not losing packets or driving the network into congestion that can lead to application failures. Figure 5 shows a client accessing a web page across the internet using TCP.

Figure 5. TCP Timer for resend



If one packet is lost TCP re-sends everything.

The problem is when do the senders resend packets. Resending too soon and too often causes congestion and resending too late causes inefficient network throughput. Therefore, the engineering compromise is to average the last 10 measurements of the round trip-time

(RTT). Every TCP message is time stamped, the receiver measures the difference between the time the acknowledgement is received, and the time the message is sent.

$\overline{R}TT(k) = 1/k \sum_{i=1}^{k} RTT(i),$ Where $\overline{R}TT(k)$ is the average roundtrip time.

$$\overline{R}TT(k+1) = 1/(k+1) \sum_{i=1}^{k+1} RTT(i),$$

$$= 1/(k+1)[RTT(k+1) + k/k \sum_{i=1}^{k} RTT(i),$$

$$= k/(k+1)\overline{R}TT(k) + 1/(k+1)RTT(k+1),$$

Now using exponential filter smoothing

$$\hat{R}TT(k+1) = \alpha \hat{R}TT(k) + (1-\alpha)RTT(k+1),$$

α = 7/8 for smoothing over the last 10 observations

But with wild swings in RTT, TCP had too many retransmissions clogging up the links.

Designers wanted to use the standard deviation of the average RTT but they soon saw that they would be unable to complete the computations within the required time because of the need to take a square root. The time needed would lead to a computation lag destabilizing the system. So a measure of the variance is used, the mean variance that avoids square roots.

$$\hat{D}EV(k+1) = \alpha \quad \hat{D}EV(k) + (1-\alpha)|\text{ RTT (k)} - \overline{R}TT(k)|$$
$$\text{RTO (k+1)} = \hat{R}TT(k+1) + f\hat{D}EV(k+1)$$

By trial and error it was found that the expected value of f =2 that reflected on deviation for each direction was too tight a bound and many retransmissions occurred. The pragmatic use of f = 4 was used to keep the calculations simple.

If a timeout still occurs a binary exponential backoff is used for each timeout

RTO (j) = 2 RTO (j-1) where j-1 is the number of timeouts in a row up to 16.

Here is a summary of fault tolerant design algorithms:

- Approximate an averaging process by a polynomial filter of length 10.
- Change the filter length from .9 to 7/8 to simplify the implementation and take advantaging of binary arithmetic in the computer
- Move from standard deviation to mean deviation to eliminate square roots

- Use 4, a binary number, for computing RTO settings

Simplify the design by reducing complexity of equations, eliminating redundant functions and sections of code, and reducing the fan out of the modules. The fan out is the 'used by' view, which is the inverse of the 'uses.' A general approach is to use a concordance of the components 'make files.' Start with a visualization of the makefile calling trees to see the complexity.

- Eliminate 'gold plated' functions.
- Use Commercial Off-the-Shelf packages to replace custom modules.
- Renegotiate requirements with hard cost/value analysis. Costs may be development or equipment costs in the target machine.
- Use architectural, design and implementation patterns.
- Use well-known algorithms.

2.3.7 Refactoring to Simpler Software

Martin Fowler writes, "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written." [Fow00]

Refactoring is a powerful and very effective way to reduce complexity. The notion of 'if works, don't fix it' is a poor approach to software design. Experts, the inventors of Unix, C and C++, practiced a 'make it work, make it work right and then make it work better philosophy.' One big obstacle is that software revision is very difficult without the originator's help because most code is obscure. Designers must work hard to get the logical organization right at every level. It is even harder with object-oriented code because the long-reaching effects of early decisions in bottom-up design demand greater insight than top-down design. Managers don't tout their product's internal simplicity and clarity. Efficiency, features, production schedule, all comes in for praise, but clarity--never! Yet only clear code can be modified. Preserving clarity through cycles of modification is even harder. During Norman Wilson's five-year tenure as the primary maintainer of research UNIX™, he wrote a negative amount of code. The system became more capable, more maintainable and more portable. Imagine a major software project subtracting code in the course of adding a feature! Allocating as much as twenty percent of the effort on a new release to improving the maintenance of the system pays large dividends by making the system perform better,

avoiding failures induced by undesired interactions between modules and reducing the time and space constraints on new feature designs. The goal is to reduce the amount of processor time old modules use and the amount of memory they occupy or I/O they trigger while holding their interfaces fixed. Other modules may be modified or new ones added to provide new features. This strategy naturally leads to more reliable systems. This approach is best demonstrated in the story of the development of 'diff,' one of the most used and least understood C function that can take the difference of arbitrary files. It is the backbone of most software change control and build systems.

2.3.8 The Tale of "diff": Real-world Refactoring

Once upon a time, there was a mathematical problem of finding the longest subsequence of lines common to two files[1]*. "No sweat," thought the developer. A dynamic programming technique that takes time mn and space mn to compare an m-line file to an n-line file would do the trick. But space mn was unacceptable on the small machines of yesteryear. "OK, we'll fly seat of the pants," thought our hero. So he read both files until he found a line that disagreed, then figured he would somehow search back and forth in both until he got back in sync. 'Somehow' was the killer. Suppose the second line in one file agreed with the fourth line ahead in the other and vice versa. How to choose?*

Then news came from afar in Princeton that the Wizard Hirschberger had seen a way to reduce space mn by a mathematical method to space m, while only doubling the time. "Good deal!" thought our guy. "Now we can afford to run it. It was slow, but it did work and gave an explainable 'right' answer in a clearly defined way."

But the people complained. When they moved a paragraph, it showed up as two changes, a deletion here and an addition there. So our hero made a "diff" that found moves. It was again seat of the pants, but it ran pretty well. Yet, sometimes, an evil occurred. If the people ran it on stuff where the same line occurred in many places, like assembly language or text processing, it discovered lots of deletions and additions that could be explained as moves. Our hero was filled with consternation.

Then along came a shining knight, Harold Stone, with a dynamic programming technique that reduced the running time from the product to the sum of the file lengths, except in unnatural cases. Now here was something fast enough to use on big files, efficient in space and time, mathematically justifiable as giving a good answer, and experimentally shown to be physiologically useful.

But then the people tinkered. Three times they altered output. They added features. They added stars! And the tinkering caused the code to increase and the manual to swell to half again its size. "Well," said our guy. "It is important to know when to stop."

[1] With the permission of Doug McIlroy, inventor of diff.

2.3.9 Reuse 'as is'

Data collected on the reuse of 2,954 modules of NASA programs [Selb88] clearly demands the shocking conclusion that to reap the benefits of the extra original effort to make a module reusable, it must be reused essentially unchanged. No change costs five percent; the slightest change drives the cost up to sixty percent. The issues of who pays the differential and who pays for ongoing support remain serious barriers to reuse. Within an organization, however, success is possible. Before middleware platforms were available, most products contained only ten percent reused modules and none contained a hundred percent reused modules. After platform became available some product was made entirely of reused modules.

In the category of currently intractable problems, it has been impossible to systematically reuse software across application domains. There is ongoing work in modeling application domains to capture the relationship between requirements and object types to reuse software architectures [Goma95]. Also, reuse even in the same application domain is successful only when throughput and response time are not overriding concerns. Finally, it is not yet possible to maintain an asset base of software modules except when they are in packaged libraries and when they are utility functions.

Where reuse is successful, there is high level of management attention to detail and a willingness to invest in design for reusability. Software configuration management assumes that there is an existing base of software components from which the components of a specific system are chosen, assembled, tested and distributed to a user [Prac95]. Even then, exhaustive re-testing is still required to root out what Jackson called "undesired interactions."

2.3.10 Boundary and Self Checking Software

One of the fundamental challenges to those building fault tolerant software is bonding the results so that errors cannot propagate and become failures. In one case an electronically steered radar was tracking a missile when it told by a computer to suddenly steer the radar beam down to the horizon when the missile was actually at 100,00 feet. The system lost track of the missile. For safety reasons the missile was destroyed when an onboard countdown timer timed out and triggered an explosion. The countdown timer was reset whenever a radar tracking pulse reflected off the missile. This was a pioneering use of electronically steered radars and the software developers could not imagine the radar beam shifting so wildly in so quickly. After all, mechanically steered radars had inertia working for them. There were no bounds placed on the output commands for radar beam steering from the computer. Software engineers understand the need to bound outputs but they are often at a loss for just what bounds to use. Checking on the outputs and other internal states of the software during its execution is referred to as self-checking software.

Self-checking software is not a rigorously described method in the literature, but rather a more ad hoc method used in some important systems [Lyu95]. Self-checking software has been

implemented in some extremely reliable and safety-critical systems already deployed in our society, including the Lucent ESS-5 phone switch and the Airbus A-340 airplanes. [Lyu95]

Self-checking software are the extra checks, often including some amount check pointing and rollback recovery methods added into fault-tolerant or safety critical systems. Other methods including separate tasks that "walk" the heap finding and correcting data defects and the options of using degraded performance algorithms. While self-checking may not be a rigorous methodology, it has shown to be surprisingly effective.

The obvious problem with self-checking software is its lack of rigor. Code coverage for a fault tolerant system is unknown. Furthermore, just how reliable a system made with self-checking software? Without the proper rigor and experiments comparing and improving self-checking software cannot effectively be done.

A breakthrough idea by Sha [Sha01] uses well-tested high reliable components to bound the outputs of newer high performance replacements. He reports, “Once we ensure that the system states will remain admissible, we can safely conduct statistical performance evaluation of the high-performance controller…the high-assurance subsystem (used to bound the states)…protects...against latent faults in the high-performance control software that tests and evaluations fail to catch.” I call this the Sha Tandem High- Assurance Paradigm.

When systems share a common architecture, they are the same and can form the base for use of the Sha Tandem paradigm. Architecture is the body of instructions, written in a specific coding language, which controls the structure and interactions of the system modules. The properties of reliability, capacity, throughput, consistency and module compatibility are fixed at the architectural level.

The processing architecture is code that governs how the processing modules work together to do the system functions. The communication architecture is code that governs the interactions of the processing modules with data and with other systems. The data architecture is code that controls how the data files are structured, filled with data, and accessed.

Once the architecture is established, functions may be assigned to processing modules, and the system may be built. Processing modules can vary greatly in size and scope, depending on the function each performs, and the same module may differ across installations. In every case, however, the processing architecture, communication architecture and data architecture constitute the software architecture that is the system’s unchanging “fingerprint.”

When several sites use software systems with a common architecture, they are considered to be using the same software system even though they may do somewhat different things. Alternatively, two systems with differing architectures can perform the same function although they do not do it the same way. They would be different systems.

For example, in the late 1980s, Bell Laboratories needed to develop a system to control a very critical congestion situation. The system was called NEMOS and had distributed database architecture. It soon became apparent that the design problem was extremely complex and broke new theoretical ground. Since there was no history of similar development for a guide and the need was urgent, Bell Laboratories decided, for insurance, to develop a second system in parallel. It was also called NEMOS, but used instead integrated database architecture. The result was two systems with the same name, performing the same function. The system with the distributed database architecture failed, and the system with the integrated database architecture succeeded. They were two different systems.

No two iterations of a software system are the same, despite their shared architecture. The high assurance one can be put in Tandem with the newer high performance one to bound the outputs. When a system is installed at two or more sites, localization is always required. Tables are populated with data to configure the software to meet the needs of specific customer sites. The customer may have special needs that require more than minor table adjustments. Customization of some modules may be required. New modules may be added. Ongoing management of this kaleidoscope of systems is a major effort. The use of the Sha Tandem paradigm once again can keep all sites at a high assurance level.

2.3.11 Trustworthiness in the Large

Some software practitioners[Nat97] broaden the definition of trustworthiness to include confidentiality, authentication of users and authentication of the source and integrity of data. Technologies exist to further these ends. Cryptography is little used in commercial, personal computer and network environments because, if it is to be widely used, it must be inexpensive and easy to use, and must not impose significant performance burdens. Firewalls are a mechanism that is deployed at the boundary between a secure enclave and an insecure network, an effective and relatively inexpensive approach. Specifications for system functionality can constrain access to system resources and require authentication of users and their access requests. An interesting idea is that it may be possible to build networked computer systems that protect themselves with the same kind of herding or schooling behaviors exhibited in the natural world. The aggregate behavior of the system, not the functioning of specific single components according to their requirements, would achieve trustworthiness. For example, a system that relies on a consensus decision to change a routing table may be more resilient than one that does not, because an attacker would need to subvert not just an individual router but the entire consensus group. Finally, the best protection is increasing the overall quality of commercial software through formal engineering methods: high-level languages, coding standards, object-oriented design and testing based on various coverage metrics.

***First constraint*:** Control-free interfaces

Large distributed real-time systems can be built effectively by integrating a set of nearly autonomous components that communicate via stable control-free interfaces, called temporal firewalls. A temporal firewall provides an understandable abstraction of the subsystem behind the firewall, confines the impact of most changes to the encapsulated subsystem, and limits the potential of error propagation. "Reusing software components in mission-critical applications cannot succeed if the components do not export clearly stated service guarantees." [Bieg99]

Second Constraint: Software Error Recovery

If failure is unavoidable, then the software design must be constrained so that the system can recover in an orderly way. This is called exception handling. Each software process or object class should provide special code that recovers when triggered. A software fault-tolerant library with a watchdog daemon can be built into the system. When the watchdog detects a problem, it launches the recovery code peculiar to the application software. In call processing systems this usually means dropping the call but not crashing the system. In administrative applications where keeping the database is key, the recovery system may recover a transaction from a backup data file or log the event and rebuild the database from the last checkpoint. Designers are constrained to explicitly define the recovery method for each process and object class using a standard library.

Fault tolerance differs from exception handling. Fault tolerance attempts to provide services compliant with the specification after detecting a fault [Lyu95]. Exception handling contains a problem and eliminates it. Reliable software accomplishes its task under adverse conditions while robust software finds and isolates a problem. Both approaches are needed in trustworthy software.

Peter Weinberger of AWK fame[2] pointed out that there are process recovery features in UNIX: "A process can choose to catch signals. This mechanism gives the process a chance to react to certain kinds of internal and external events. A data filtering process can catch arithmetic errors (like overflow or divide by zero) …and by using *longjump()*to re-initialize itself and continue." A parent process can restart a damaged process and avoid complicated recovery code.

The software architecture for the Safeguard anti-missile system included restarts . The operating system provided a 'mission mode' capability. It allowed the software engineer to tailor specific error recovery to a process and exit without crashing or hanging the computer. For example, the software that tracked Intercontinental Ballistic Missiles had error recovery code that dropped track and reinitialized the tracking data area if there was a 'divide by zero' trap reported to the operating system. The operating system executed the special 'on

[2] Private communication

interrupt' code and then returned to normal processing. Since the fuel tank of an ICBM flies on a lower trajectory than its re-entry vehicle (RV) and breaks into pieces during atmospheric reentry, could the system track the reentry vehicle through tank breakup? Would the RV be masked by the tank pieces? Would the system lose track of the reentry vehicle? Once the tank pieces reenter the atmosphere they slow down. Being heavier and specially designed the reentry vehicle continues to fly at high speed. During one test flight, the software was tracking the tank and the reentry vehicle as two separate objects. Once the tank hit the atmosphere it broke up and slowed. Additional software tracking channels were assigned and tank pieces were tracked. In many cases the software computed zero velocity. A design flaw, later corrected, did not validate the computed velocity for an object in a track channel. More than 1000 'divide by zero' traps occurred in the tracking channels assigned to the tank but the system continued operating. The software continued to track the RV. Since this tank breakup was not expected, the software was not tested for the high rate of 'divide by zero' traps before the test flight.

Third Constraint: Recovery Blocks

The recovery block method is a simple method developed by Randell from what was observed as somewhat current practice at the time [Lyu95]. The recovery block operates with a program that confirms the results of various implementations of the same algorithm. In a system with recovery blocks, the system view is broken down into fault recoverable blocks. The entire system is constructed of these fault tolerant blocks. Each block contains at least a primary, secondary and exceptional case code along with an adjudicator. (It is important to note that this definition can be recursive, and that any component may be composed of another fault tolerant block composed of primary, secondary, exceptional case, and adjudicator components.) The adjudicator is the component that determines the correctness of the various blocks to try. The adjudicator should be kept somewhat simple in order to maintain execution speed and aide in correctness. Upon first entering a unit, the adjudicator first executes the primary alternate. (There may be N alternates in a unit which the adjudicator may try.) If the adjudicator determines that the primary block failed, it then tries to roll back the state of the system and tries the secondary alternate. If the adjudicator does not accept the results of any of the alternates, it then invokes the exception handler, which then indicates the fact that the software could not perform the requested operation. The recovery block system is also complicated by the fact that it requires the ability to roll back the state of the system from trying an alternate. This may be accomplished in a variety of ways, including hardware support for these operations. This try and rollback ability has the effect of making the software to appear extremely transactional, in which only after a transaction is accepted is it committed to the system. There are advantages to a system built with a transactional nature, the largest of which is the difficult nature of getting such a system into an incorrect or unstable state. This property, in combination with check pointing and recovery may aide in constructing a distributed hardware fault tolerant system.

Fourth Constraint: Limit the Language Features Used and inspect the code

Most communications software is developed in the C or C++ programming languages. Hatton's Safer C [Hatt97] describes the best way to use C and C++ in mission-critical applications. Hatton advocates constraining the use of the language features to achieve reliable software performance and then goes on to specify instruction by instruction how to do it. He says, "The use of C in safety-related or high-integrity systems is not recommended without severe and automatically enforceable constraints. However, if these are present using the formidable tool support (including the extensive C library), the best available evidence suggests that it is then possible to write software of *at least* as high intrinsic quality and consistency as with other commonly used languages."

C is an intermediate language, between high level and machine level. The power of C can be harnessed to assure that source code is well structured. One important constraint is to use function prototypes or special object classes for interfaces.

Once you have the code it is important to read it. While formal code inspections have proven valuable in finding faults and improving the reliability of the software it can lead to an exodus of the very best developers. This happens when code inspections become perfunctory. A good practice is to inspect the code of programmers when the first join the project and then inspect if they produce buggy code. Programming standards should at least cover:

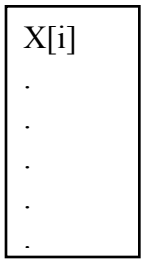
1. Defining Variable to make the code self –documenting
2. Commentary: Too many comments are not best as they could mask the code and be hard to keep current. The comments must why you choose a particular pattern of instructions rather than what they do.

Code inspections. Inspections need to see

1. If wild transfers are possible by checking every entry and exit point.
2. boundary conditions are very important
3. buffer overflows are still a serious problem even with Java. Unconstrained pointers can result from poor array bounds leading to memory leaks.
4. Do comments agree with code?
5. Are variables, pointers and arrays initialized.
6. Does every loop terminate?
7. Examine subscripts to see if we are within bounds.

Example:

Array



i is defined as $16 \leq i \leq 37$

How many elements? It is (h-l +1), but very often people forget to take into account the "1".

Best way to define i is $16 \leq i < 38$

Now the equation becomes (h-l) i.e. (38-16) = 22.

Check if semantics and syntax of boundary conditions are specified properly.

Code Reading: Everybody on the team is involved in this process. Everyone hands in some code. The code is re-distributed until everyone has somebody else's code. Code is read and then group meet to discuss what they have found. This process improves the coding ability of the group.

Code Review: Person who wrote the code will find the most bugs. Test the code for syntax error, logic errors, and incompleteness error. Check the code against user requirements. Check the code against coding standards[3]. Don O'Neill, an excellent expert in software process writes," Analysis of the issues raised in the experiment to date has revealed common problems that reoccur from session to session. Organizations that want to reduce their software fault rates need to prevent these defects:

1. Software product source code components not traced to requirements. As a result, the software product is not under intellectual control, verification procedures are imprecise, and changes cannot be managed.
2. Software engineering practices for systematic design and structured programming is applied without sufficient rigor and discipline. As a result, high defect rates are experienced in logic, data, interfaces and functionality.
3. Software product designs and source code are recorded in an ad hoc style. As a result, the understandability, adaptability and maintainability of the software product is directly impacted.
4. The rules of construction for the application domain are not clearly stated, understood, and applied. As a result, common patterns and templates are not exploited in preparation for later reuse.
5. The code and upload development paradigm is becoming predominant in emerging e-commerce applications. As a result, the enterprise code base services only the short term planning horizon where code rules and heroes flourish, but it mortgages the future where traceable baseline requirements, specification, and design artifacts are necessary foundations."

Read the following code segment. Note how it is self-documenting but it is not fault tolerant. Even though Qname and Qvalue are validated on an input screen on a web client, there is the possibility that an unanticipated data value can be passed to the server. The last 'else' clause should have another else that reports an error condition and reinitializes the process so that it

[3] See http://hometown.aol.com/ONeillDon/nsqe-results.html where there is an explanation of the code review process.

is ready for the next user. This code was taken from an undergraduate in-class formal code review at Stevens Institute of Technology:

Code Extract:

```
while(values.hasMoreElements())
                {
                        Qname = new String((String)values.nextElement());
                        Qvalue = new String(req.getParameterValues(Qname)[0]);

if (("day".equals(Qname)) || ("month".equals(Qname)) || ("year2".equals(Qname)))
                                {
                                        date.addElement(Qvalue);
                                }

                        else if (("death".equals(Qname)) || ("road_func".equals(Qname)) ||
("atmos_cond".equals(Qname)))
                                {
                                        afields.addElement(Qname); // accident category
                                        avals.addElement(Qvalue);
                                }

                        else if (("restraint".equals(Qname)) || ("drug_invl".equals(Qname)) ||
("injury_severity".equals(Qname)) || ("police_report_alco".equals(Qname)) ||
("sex".equals(Qname)) || ("ejection".equals(Qname)))
                                {
                                        pfields.addElement(Qname);  // person category
                                        pvals.addElement(Qvalue);
                                }

                        else if (("make".equals(Qname)) || ("model".equals(Qname)) ||
("year".equals(Qname)) || ("rollover".equals(Qname)) || ("no_of_occup".equals(Qname)) ||
("death".equals(Qname)) || ("reg_state".equals(Qname)) || ("impact1".equals(Qname)) ||
("fire".equals(Qname)))
                                {
                                        vfields.addElement(Qname); // vehicle category
                                        vvals.addElement(Qvalue);
                                }

                        else
                                {
                                        dfields.addElement(Qname); // driver category
                                        dvals.addElement(Qvalue);
```

```
                    }
            }
```

Fifth Constraint: Limit Module Size and Initialize Memory

The optimum module size for the fewest defects is between 300 to 500 instructions. Smaller modules lead to too many interfaces and larger ones are too big for the designer to handle. Structural problems creep into large modules.

All memory should be explicitly initialized before it is used. Memory leak detection tools should be used to make sure that a software process does not grab all available memory for itself, leaving none for other processes. This creates gridlock as the system hangs in a wait state because it cannot process any new data.

Sixth Constraint: Reuse Modules Without Change

A study of 3000 reused modules showed that changes of as little as 10 percent led to substantial rework--as much as 60 percent--in the reused module. It is difficult for anyone unfamiliar with a module to alter it, and this often leads to redoing the software rather than reusing it. For that reason, it is best to reuse tested, error-free modules as they are with no changes.

Formal methods specify or model the requirements mathematically, even though not all ambiguity can be eliminated with this method. Prototyping, simulation and modeling can also be used to complement mathematical requirements. Component isolation separates safety critical components; this modularity ensures that changes are contained. Information hiding similarly prevents one component's actions from affecting another's. Redundancy is used to prevent or recover from failure. Human factors design during the design phase is critical.

Use of high-level languages lessens programming errors by eliminating problematic programming practices. Reverse engineering recreates documentation for preexisting software and provides a basis for reuse. There are also software engineering practices that apply to the software assurance processes. Use of cost-modeling and risk assessment techniques can aid the project management process. Inspections, reviews and audits can be applied to all software processes under the software quality assurance process. Software error, measurement, statistical, algorithm, database, technical, control and data flow, and timing and sizing analysis techniques are useful in the software verification and validation process. Test strategies based on coverage metrics are efficient and thorough.

2.4 Time factors (t)

2.4.1 Program execution time- Software Rejuvenation

Reliability is improved by limiting the execution domain state space. Today's software runs non-periodically, which allows internal states to develop chaotically without bound. Software rejuvenation is a concept that seeks to contain the execution domain by making it periodic. An application is gracefully terminated and immediately restarted at a known, clean, internal state. Failure is anticipated and avoided. Non-stationary, random processes are transformed into stationary ones. The software states would be re-initialized each day, process by process, while the system continued to operate. Increasing the rejuvenation period reduces the cost of downtime but increases overhead. Rejuvenation does not remove bugs; it merely avoids them with incredibly good effect. Chandra Kintala of Bell Labs defines three software fault tolerance components. They may be used with any UNIX or NT application to let the application withstand faults. They are watchd, libft and repl[4].

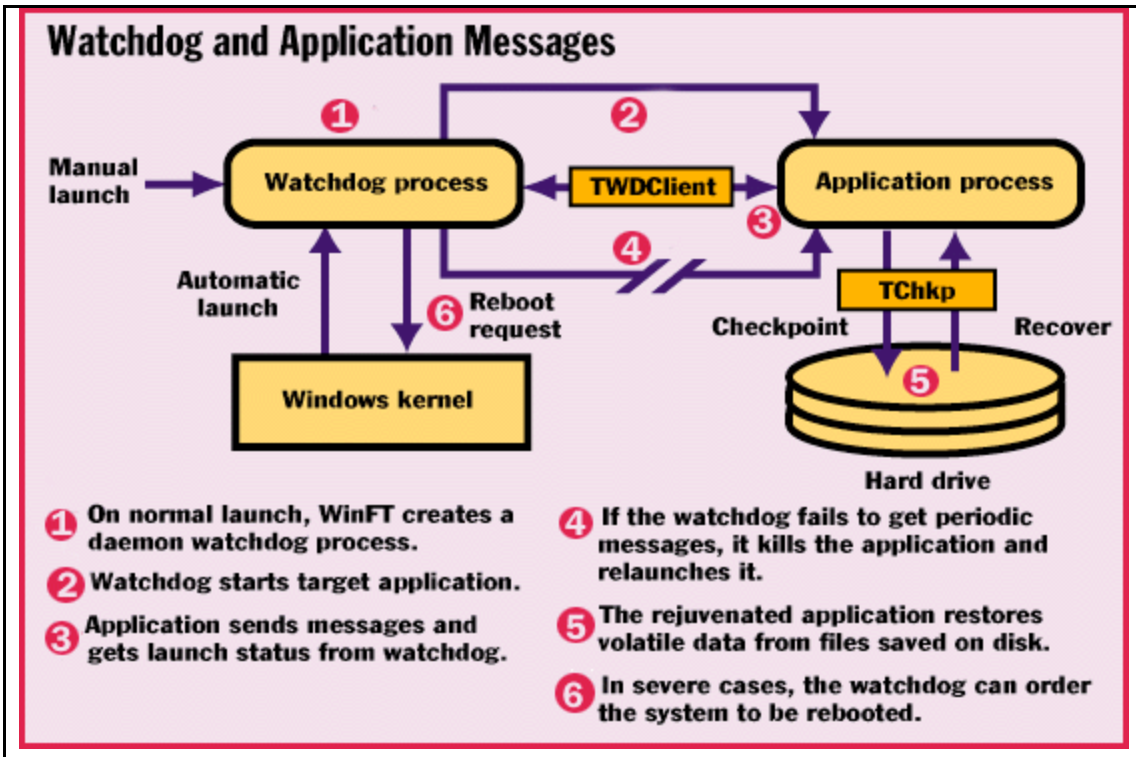
Watchd is a watchdog daemon process for detecting UNIX process failures (crashes and hangs) and restarting those processes. The fault tolerance mechanism is based on a cyclic protocol and the recovery mechanism is based on the primary copy approach.
Libft is a C library for check pointing the internal state of an application process periodically on a backup node. It also provides recovery routines to restore the state of a process at the backup node and a location-dependant connection mechanism between server and client processes. With these checkpoint and recovery mechanisms, a server process can be dynamically migrated to a different node for load balancing and fault tolerance. To tolerate design and program faults, it provides fault tolerance programming constructs, such as, recovery blocks, N-version programming, and exception handling. It also provides fault tolerant I/O functions for safe I/O.

REPL is a file replication mechanism that replicates files located on one physical file system onto another physical file system at run time. It provides features for both synchronous as well as asynchronous run-time replication of file updates [Huang93].

Windows 95 has a special library WinFT that provides automatic detection and restarting of failed processes; diagnosing and rebooting of a malfunctioning or strangled OS; checking pointing and recovery of critical volatile data; and preventive actions, such as software rejuvenation[Carr97].

Figure 6. Watchdog and application messages (from

[4] Note that *watchd, libft* and *REPL* are registered trademarks of AT&T Corporation.



By using a fixed or upper bound on the execution time and then restarting the reliability equation becomes:

$R(t) = e^{-ktC/E}$, where $0 < t < T$ and T is the upper bound of the rejuvenation interval. This limits the reliability to be no less than $e^{-kTC/E}$ for a fixed C and E.

Software rejuvenation was initially, developed by Bell Laboratories in the late 1970s for its billing systems and perfected by NASA.

The execution of a software process can show signs of wear after it executes for a long period. This process aging can be the effects of buffer overflow's, memory leaks, unreleased file locks, data corruption or round-off errors. Process aging degrades the execution of the process and can often cause it to fail. This effect is different then the software aging problem identified by Parnas. He points out that application programs become less reliable and often fail due to a changing extended machine environment, new requirements and maintenance. In contrast process aging is related to application processes degrading after days and weeks of execution. Software fault tolerance techniques need to deal with both aging mechanisms.

With process aging the software works perfectly for a period with no risk of failure. It then enters a jeopardy period where it is vulnerable to executing the fault that now becomes a failure. As an example a process with a memory leak problem will not fail until the process memory request exceeds all allocated memory. For the time that the memory footprint for

the process is growing the software is executing with no problem. Sometimes slower response times are noticed before the failure when the process enters the jeopardy state. Kintala calls the period that the software is working fine the 'base longevity interval.'

Here is a case story in progress. The NASA mission to explore Pluto has a very long mission life of 12 years. A fault-tolerant environment incorporating on-board preventive maintenance is critical to maximize the reliability of a spacecraft in a deep-space mission. This is based on the inherent system redundancy (the dual processor strings that perform spacecraft and scientific functions during encounter time). The two processor strings are scheduled to be on/off duty periodically, in order to reduce the likelihood of system failure due to radiation damage and other reversible aging processes.

Since the software is reinitialized when a string is powered on, switching between strings results in software rejuvenation. This avoids failures caused by potential error conditions accrued in the system environment such as memory leakage, unreleased file locks and data corruption. The implementation of this idea involves deliberately stopping the running program and cleaning its internal state through flushing buffers, garbage collection, reinitializing the internal kernel tables or, more thoroughly, rebooting the computer.

Such preventive maintenance procedures may result in appreciable system downtime. However, by exploiting the inherent hardware redundancy in this Pluto mission example, the performance cost is minimal. One of the strings is always performing and starting it before the current active string is turned off can mask the overhead for a string's initialization. An essential issue in preventive maintenance is to determine the optimal interval between successive maintenance activities to balance the risk of system failure due to component fatigue or aging against that due to unsuccessful maintenance itself [Tai97].

Continuing experimentation is being done, notably at the University of Southern California, to refine this technique. It has, however, been in the literature for more than twenty years and its use in the industry is negligible. Most software practitioners are unaware of it.

2.5 Effort factors (E)

2.5.1 Effort Estimates

Barry Bohem, Capers Jones and Larry Putnam have developed software estimation theory and models. The fundamental equation in Barry Boehm's COCOMO model is

$$PM = (2.94)(Size)^{E} [\_ EM(n)],$$

where

PM is the expected number of staff months required to build the system,

Size is thousands of new or changed source lines of code excluding commentary

[EM(n)] is the product of effort multipliers, one of the multipliers is complexity. The complexity multiplier rates a component based on the factors of Control Operations, Computational Operations, Device dependent operations, Data Management Operations and User Interface Management Operations. This effort multiplier varies from 0.73 for very low complexity and 1.74 for extra high complexity.

The effort term E in the Sha equation is equal to or greater than PM. For a given component once the average effort is estimated, reliability can be improved if the invested effort exceeds the nominal effort. If the invested development effort is less very unreliable software can be expected. The programmers can be made more effective by investing in tools and training. These factors are integrated into the COCOMO model.

2.5.2 Hire Good People and Keep Them

This is key. Every shop software claims to employ the 'best and the brightest.' Few really do. The book Peopleware is necessary read for how to improve the software staff. Guru Programmers who masters at their art are twenty to thirty times more productive than average programmers. Hiring the best available tends to raise the level of the entire organization as methods and reasoning are taught to colleagues. It is vital to recognize the difference between vocational training and education. An educated staff can quickly adapt to new technologies and processes.

Practitioners are poorly trained in known good methods. The New York Times has commented on the trend in the computer industry towards younger and less well educated practitioners. They are supposedly valued for their intuitive skills and aversion to structure. High school dropouts are hobbyists who learn skills not in a classroom but in pursuit of computer games, digital music, video editing, computer animation and film. On the job training consists of random mucking about, romantically excused by saying the industry moves too quickly for textbooks and knows precious few rules. Immaturity is prized. Some started as toddlers with parents' home computers and are admittedly deficient in human socialization and interactive skills due to that early and prolonged isolation. That a major software corporation offers quasi-certification of such people diminishes the value of genuine engineering. A Cisco Certified Internet work Expert certificate holder's starting salary is $75K, quite remarkable even in an inflated technology atmosphere [Wall00]. By 2002 these same hot shots were looking for work.

2.5.3 Effectiveness of Programming Staff

Any software shop can make their people more effective as they set about improving the quality of their people. They can recognize that very large changes (more than 100 instructions) and very small changes (fewer than 5 instructions) are more error-prone than medium sized changes. This may have some relationship to the average size of human working memory in the large case, and limited ability to attend to detail in the small case.

Even when excellent methods are developed it is difficult to have the methodology promulgated and incorporated into curricula for widespread study. An effective software shop invests in keeping their people current.

Do new methods ignore some dynamic of commercial software development? Successful corporations act in their own best interests. If basic engineering principles are ignored and trustworthiness is imperiled, there must be some stronger motivation in operation. Microsoft Windows is an example of planned obsolescence. Each new version is deliberately incompatible with the previous version in commands, menus, formats and responses. Each new version requires extensive human re-learning and reconfiguration of interfacing systems. It creates a market for services and pressure to buy the most current version to remain compatible, which is profitable for Microsoft, but it is a chronic drain of time and money.

Sometimes it might be expedient to not invest in expensive security measures because the customer had not demanded it. It might be that customers are slowly becoming better-educated consumers, however. The lessons learned from a recent crippling of commercial web servers are that everyone is super-empowered, individuals and vandals alike, in a networked world. Many little encroachments on privacy add up to large losses of personal control that people find distressing. Nations may come to understand that government still matters because it is the only entity with the resources and the laws to ensure that personal rights are protected [Fried00]. When there is such demand from people through their governments, basic engineering principles will probably be valued by software corporations.

Do practitioners accurately identify key problems? There is no cultural basis for sound engineering practice in the software industry. Even NASA, the very best software shop in the world and renown for its early successes has fallen into the trap of taking engineering shortcuts that have come back to haunt them.

On June 4, 1997, the Mars Pathfinder landed and for the next 3 months Sojourner surveyed the terrain. This very successful mission began, however, with Pathfinder experiencing mysterious complete system resets that took ingenious work on the part of the software design team at NASA's Jet Propulsion Laboratory to reproduce and fix the defect. It turned out that the priorities assigned to tasks in the multi-tasking real-time system fell into a priority inversion because the load volume was heavier than the maximum that had been tested as shown in the next case study

2.5.4 Case study: the Mars Explorer

Prepared by Thomas Krusinski

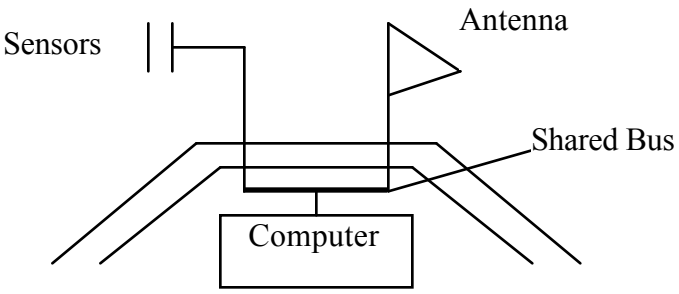
Two failures on successive Mars satellites:

- 1st: communication problem between 2 teams, one using the metric system and the other using yards
- 2nd: a software bug caused the probe not to decelerate fast enough when entering the Mars atmosphere

Another bug:

- the probe stops sending images occasionally, then increasingly often and for longer periods, and then reboots.

Explanation



There is a conflict for accessing the resource: Bus.

Priorities should be:

- Reboot
- Send images
- Gather data

But due to a faulty use of preemptive multithreading, they were:

- Reboot
- Gather data
- Send images

Priorities designed for the hardware, the application software and the driver that controlled the bus were inconsistent. A fail-safe watchdog counter rebooted the system after it had been inactive for some time. This is why the probe was rebooting after being silent for too long. This is a **fail-safe** system.

This problem is a typical deadlock that can happen when access to resources are not properly designed. The deadlock is similar to those experienced in multi-thread database systems.

Some solutions

- Design defensively: leave in the debug code, build fail-safe systems
- Stress test: test the system beyond its limits established in the requirements. It is best to test to the breaking point. The difference between the breaking point and the maximum design load is the system margin. The stress tests for the Mars Explorer were actually thought to be the worst case. But because the data gathering went better than expected there was more data than expected. This resulted in the shared bus not being released.
- Explain all anomalies that show up during system test. Sometimes you must release the software for its intended use even though you have not been able to discover the cause of the anomaly. In these cases treat the release as "provisional" and continue to work on understanding the anomaly. In the case of the Mars Explorer, the anomaly has been detected during the tests; however, it was believed to be a hardware problem.
- Hold architecture reviews and pay special attention to performance issues. Process scheduling algorithms need detailed analysis. All shared resources must be understood and interactions must be analyzed. Simulate use cases.

Fortunately, the software on the Pathfinder had debugging trace features left in place, so an exact replica of the software could be used on Earth with the confidence that the timing nature of the problem would not be contaminated. A fix was implemented using a priority inheritance protocol that was built into the operating system. As an aside, an early anti-missile test failed when the configuration used in the missile firing differed slightly from the one used in testing. Software developers insisted that the firing be done with the debug software in the configuration. The mission planners over ruled them since the debug software might halt the computer. Logic in this case failed because of the fragile nature of the software. A small difference in the real data from that used in all the tests caused mission failure.

Returning to the Mars mission.

The actual data rates experienced during the mission were better than the test "worst case." NASA's test engineers did report one or two system resets during their testing, however they never successfully reproduced or explained these failures. To their dismay their observations were dismissed as aberrations.

There were two missed opportunities to catch this priority inversion. First, classically trained engineers stress a system until it breaks, then guarantee it for considerably less than the break point. A mere guess at what the break point might be, especially in a very new environment, is unacceptable. Had the software been sufficiently loaded to reveal the priority inversion, it could have been fixed without heroics. Over-engineering and over-testing are not sound practices that can take the place of mathematical definitions and designs.

Human factors specialists are familiar with the pitfalls of dreaming and are vigilant against it. Unexplained system failures cannot be ignored because of management pressures to pass the software to meet a date. Failures cannot be brushed away with the hope that they were so infrequent, maybe they will not happen again [March99]. The Apollo missions owed their successes in large part to the excellent human factors work that was done to rehearse and anticipate every possible contingency in entirely new circumstances. Apparently, with a new generation of designers, that knowledge has been lost because it is rarely incorporated in university curricula. Moreover, these efforts are expensive. They can consume as much as 20% of resources and there are few financial consequences to failure. To get the contract be the low-cost bidder. The low-cost bidder often minimizes reliability testing.

2.5.5 Object-Oriented Design improves effectiveness

Object oriented technology can help to fulfill early computer industry aspirations and lead to predictable system developments with high reliable software, fast time to market and solid performance

The case history of one modern telephone software system makes the point. The project was to support the use of new, very fast broadband networks in the telephone company access network. Since this was clearly a large-scale development effort, the designers adopted the use of objects very early. The size of the project in its first release was 12,600 function points, 22 software modules with 47 interfaces, and 12 databases. This complexity was organized into 278 object classes and 1200 objects. The developers adhered to five overarching principles in making their design decisions:

1. System synthesis, the melding of methods and business objects, began from the customer's, not the developers', viewpoint.
2. Modular architecture separated data from applications
3. Effective data stewards were appointed with responsibility and authority for the object classes. They were charged with authority and responsibility of the reliability of the methods in the object class.

4. Object oriented analysis included extensive domain analysis, rigorous requirements, business usage scenarios worked out with the user, formal external and internal interface agreements, and an integrated data model.
5. Object oriented design used client/server architecture and industry wide standards.
6. Code inspections were performed on selected modules. Cylomatic metrics were computed for each module to find those that needed formal code inspections.
7. System testers were granted the 'right of refusal.' They could reject any object library or application processes that in their judgment were not reliable. Designers had to do the redesign and still meet their schedule commitments. These redesigned modules were expected to be formally code inspected. Management resolved conflicts.

The most serious problem this project faced was the need to keep data consistent. Consistency drove accuracy. All former designs used convoluted error paths; these were error-prone and required more code and execution time than straightforward designs that included consistency checks in the object class methods.

Legacy system provided and accepted data from the system. Data normalization techniques, with robust error processing isolated the new system from the legacy system. Here object oriented technology was a powerful tool for allowing quick system updates to accommodate new features and changes in business practices. It made reuse natural. System designers were able to accurately reflect business objectives in the object classes.

Until such robust object oriented design became a habit, an enforced object encapsulation strategy with centralized object libraries is vital. Skilled project managers must insist that all subsystems and modules use the same Operation, Administration and Management (OA&M) software. This achieves meaningful reuse and results in huge system cost savings in operation of the system itself.

2.5.6 Corroborating Object Experiences

That these results can be attributed to a disciplined use of object-oriented technology is corroborated by the experience of others. Swiss Bank Corporation designers told me that they obtained a fifty percent productivity improvement during re-engineering efforts that started in 1991. By 1994, they were installing their new object oriented system and said that reuse was the key to their success. The benefits of prototyping and adherence to clean object class definitions were particularly apparent. They managed risks by adhering to the standard enterprise object classes and linking them together. They anticipated some performance problems and these did occur, but the cost/performance improvement of new computer servers more than compensated for the ten percent performance overrun they saw.

AT&T developed more than fifty object-oriented systems using unique objects *in memory* approach. The objects were locked in memory while the system ran. One such system may be biggest and fastest object oriented network management system in the world. It uses 1 gigabyte of memory for its 15 million objects and thousands of transactions per second on a HP high-end workstation. It has been in production for three years with no significant problems. It replaced a vintage IBM-hosted facility provisioning system. This new approach can become widely used when logical memory is extended to 64 bit addressing and added to the natural structure of object-oriented databases like Versant. This will open virtual memory machines to objects and regain freedom from memory constraints enjoyed by application developers in the earlier transaction systems.

2.5.7 Objects in Large-Scale Projects

Large-scale evolving software presents a special challenge to object architects. Typically, an application consists of a network of objects connected through compatible interfaces. The need to meet new requirements and/or fix defects often results in new interfaces and object versions. When a new version of an object is created, it must be dynamically installed without causing disruption to existing software. Objects must be intelligent enough to handle the problems of dynamic reconfiguration, coordinate inter-module communication, and track the internal states of both the objects and the links. This increases the complexity of objects and can prevent them from being reused in different contexts. One solution is to not allow interface changes. This harsh rule often makes the application difficult to build because application level interfaces are imprecise due to time-outs and repeated transmissions triggered by buffer losses in asynchronous communication. Additionally, the interface specifications are vague and not amenable to analysis.

In this dynamic environment, however, there is a premium for keeping all the modules consistent. It is very difficult for designers, who are focused on the function of each module, to worry about the way all the pieces will fit together. As a result, the issue of interface consistency is often left to test teams, where it is inefficient and time consuming. Experience shows that it is three times more expensive for testers to find and fix problems than developers. So, the interfaces must change but in a controlled way. Object oriented technology opens the door to dynamic checking of interface states and internal consistency because for the first time it is possible for projects to create libraries of interface object classes to do this job. International standards bodies recognized this problem and developed the Common Object Request Broker Architecture (CORBA) standard to do distributed computing. CORBA is in its infancy, but industry cooperation is making CORBA the object middleware standard[Gaudin97]. One problem is that CORBA locks the sender until the receiver receives and acknowledges the message, and CORBA does not support multi-cycle transactions. CORBA's object module is evolving and may become the standard of choice. The hope is that the object oriented CORBA will provide the fabric to let architects connect independently designed components together. Versant Object Technology has a powerful

way of combining Orbix from Iona Technologies and their object database. The Versant/Orbix mediator adds the dimension of a database storing the objects used in the communication interface. Network Programs, Inc. provided a multiple transaction capability for mapping applications to one another. Their adapter/collector technology is a robust way of connecting systems while avoiding undesired interactions. Meanwhile, Microsoft offers its own brand-specific object approach, Distributed Common Object Model (DCOM), which allows clients to access servers across a network and provides a binary interface for packaging and linking libraries or other applications. Since the situation is still fluid, most organizations are using both approaches in combination with in-house controls for their interface designs.

Java applets and CORBA are well suited to building distributed web applications. Browsers give access to network management data, and they allow networks to be managed remotely. To overcome delays due to network latency in the time it takes for a command to get to a network element, the *management by delegation* approach has become popular [Yem91].[i] Data are stored in management information databases close to the network elements. They may be sent to the network elements as remote agents with their programs or they may be mapped to CORBA objects which can be located anywhere but must be statically mapped to the network element. Using Java applets, designers can overcome this limitation and dynamically reconfigure programs and their data objects. Java has a remote method invocation feature that is similar to CORBA but is restricted to only Java objects. Since CORBA can be used for many languages, it is the best choice for distributing the network management data.

3 Summary

Software Fault Tolerance can be aimed at either preventing a transaction failure to keep the system operating or at recovering the database for an entire system. In both cases the goal is to prevent a fault from becoming a failure.

Software fault tolerance can be measured in terms of system availability that is a function of reliability. The exponential reliability equation using an extension to Sha's Mean Time To Failure can be used to quantitatively analyze the utility various processes, tools, libraries, test methods, management controls and other quality assurances technologies. Collectively these technologies comprise the field of software engineering. The extended reliability equation provides a unifying equation to reliability-based software engineering. Now, it is possible to define the software fault tolerance requirements for a system and then make engineering tradeoffs to invest in the software engineering technology best able to achieve the required availability.

Citigal Labs measures software fault tolerance and software safety. Their web page summarizes the state-of-the art of software fault tolerance in 2002 [Cit02]:

> "Traditionally, fault-tolerance has referred to building subsystems from redundant components that are placed in parallel [Marc00]. A prime example is the computer system for the space shuttle. On page 20 of his book, Peter Neuman states that [Neu95]: "the on-board shuttle software runs on two pairs of primary computers, with one pair being in control as long as the simultaneous computations on both agree with each other, with control passing to the other pair in the case of a mismatch. All four primary computers run identical programs. To prevent catastrophic failures in which both pairs fail to perform (for example, if the software were wrong), the shuttle has a fifth computer that is programmed with different code by different programmers from a different company, but using the same specifications and the same compiler (HAL/S). Cutover to the backup computer would have to be done manually by the astronauts."

On the shuttle, we see a combination of redundant computers and redundant software versions; redundant software versions from the same specification are typically referred to as N-version programming. *N-version programming* is a fault-tolerance improvement paradigm that executes multiple versions (that were independently designed/written and implement the same software function) in parallel and then takes a vote among the results as to which output value is most frequent.

We use the term fault-tolerance in our research and commercial products at Citigal slightly differently. For us, software is deemed as fault-tolerant if and only if:

1. the program is able to compute an *acceptable* result even if the program itself suffers from incorrect logic; *and*,
2. the program, whether correct or incorrect, is able to compute an *acceptable* result even if the program itself receives *corrupted* incoming data during execution.

The key to our definition is what is considered "acceptable." This can include characteristics such as correctness and/or safety, and is based on the system. The interpretation for what constitutes software fault-tolerance according to our definition results from a combination of the principles of software safety and robust design.

Widely accepted software engineering design practices call for *robustness* and *graceful degradation* whenever a system gets into an undesirable state. Software fault tolerance is a related concept. The distinction between robustness and fault tolerance rests on whether the undesirable state is "expected" or "unexpected." Robustness deals primarily with problems that are expected to occur and must be protected against. By contrast, fault tolerance deals with unexpected problems. These must also be protected against. For example, if we are reading in an integer that will be used in a division operation, a robust design will ensure that the division operation is not applied if the integer is zero. A fault-tolerant design accounts for unanticipated possibilities (*e.g.*, if the integer is corrupted, a fault tolerant design might freeze the state of the program and not compute the division operation --- which is equivalent

to an integer divide-by-1 --- or it might require that the integer be reread.) Here, we are interested in assessing fault-tolerance, which can be a side-benefit of robust design practices.

For critical software, there are three classes of output states that can be produced from a program execution: (1) correct, (2) incorrect, but acceptable, and non-hazardous, and (3) hazardous. Software fault-tolerance refers to the ability of the software to produce "acceptable" outputs regardless of the program states that are encountered during execution. Software safety refers to the ability of the software to produce "non-hazardous" outputs regardless of the program states that are encountered during execution. (What constitutes an output hazard is defined by the system level safety requirements.) Software safety then, according to our perspective on fault-tolerance, is simply a special type of software fault-tolerance.

Fault-tolerance refers to a class of outputs that can be tolerated, and software safety refers to a class of outputs that cannot be tolerated. For example, for an input value of 1 to the software, suppose that the correct output value is 100.0. But suppose that the numerical algorithms we use produce an output of 99.9. If the set of acceptable, non-hazardous values is {99.0, 101.0} and only this range, then this value is acceptable, and hence the software was fault-tolerant with respect to an inaccurate algorithm. Further suppose that the software, for some reason, may not receive the correct input value of 1 when it should, due to some external problem. Say the code instead receives a value of 0.0. Finally suppose that along with its low-accuracy numerical algorithm, the software produces an output value of 102.0, which is hazardous since it is out of the range of {99.0, 101.0}. Then we immediately gain knowledge that there is a potential safety problem."

4 Acknowledgements

My colleague Professor A. David Klappholz at Stevens Institute of Technology helped me to formulate my ideas. His insights guided the structure of this chapter. The Committee on National Software Studies (www.cnsoftware.org) provided data and contacts. Professor Sha's excellent paper triggered the reliability approach to software engineering. Professor Brian Randell's, winner of the 2002 IEEE Emanuel R. Piore Award for seminal contributions and leadership in computer system dependability research, established this filed of study. Les Hatton, Collin Tulley, Sam Keene leads discussions groups on software reliability and quality that were very helpful. Chandra Kintala and H. Yuang worked with me on these ideas for many years and made fundamental contributions to making software more reliable as they reduced rejuvenation to practice. Peter Neuman identified the risks of not attending to software reliability. Will Tracz encouraged me for many years.

Special thanks to C.M. Yuhas of Have Laptop - Will Travel for contributing parts of this chapter.

5. References

[Barat00] Arash Baratloo, Navjot Singh and Timothy Tsai, "Transparent Run-Time Defense AgainstStack Smashing Attacks,", Proceedings of 2000 USENIX Annual Technical Conference, San Diego, California, USA, June 18–23, 2000)

[BEA02] BEA White Paper, "Managing Complexity with Application Infrastructure," BEA Systems, Inc. 2315 North First Street, San Jose, CA 95131 U.S.A. www.bea.com

[Bell84] The origins of C are explained in Section VII. Programming Languages on page 375 of "A History of Engineering and Science in the Bell System," AT&T Bell Laboratories 1984, ISBN 0-9327634-1, Library of Congress 84-72181, pp 379-380

[Bern89] "Software Manufacturing," UNIX Review, Vol. 7, p. 7, July 1989, pp 38-45

[Bern93] Lawrence Bernstein and C.M. Yuhas, " Testing Network Management Software," Journal of Network and Systems Management, Vol. 1, No. 1, Plenum Press, 1993, ISSN 1064-7570, pp 5-15

[Berns97] Lawrence Bernstein, "Software Investment Strategy," Bell Labs Technical Journal, Summer 1997, Volume 2, Number 3, ISSN 1089-7089, pp.233-243.

[Bieg99] Antoine Beugnard, "Making Components Contract Aware," Computer, IEEE Computer Society, July 1999, Volume 12, Number 7, ISSN 0740-7459, pp 38-45

[Boehm00] Barry Boehm, et. al. "Software Estimation with COCOMO II," PTR, ISBN 0-13-026992-2]

[Carr97] Joao Carreira et. al., "Fault Tolerance for Windows Applications," Byte Magazine February 1997, pp 51-52

[Cit02]: http://www.cigitallabs.com/resources/definitions/software_safety.html

[Fow00] Martin Fowler, "Refactoring- Improving the Design of Existing Code," Addison-Wesley, 2000, ISBN 02001485672, pg xvi

[Fried00] Friedman, Thomas L. "The Hackers' Lessons," The New York Times, February 15, 2000, p. A27.

[Gaudin97] Sharon Gaud in, "Object Stamp of Approval," ComputerWorld, March 17, 1997, Vol. 31, No. 11, p. 1.

[Goma95] Hassan Goma, "Reusable Software Requirements and Architecture for Families of Systems," Journal of Systems & Software, Mar 1995 Vol. 28, No. 3, pp. 189-202.

[Hatt97] Les Hatton, Safer C: Developing Software for High-integrity and Safety-critical Systems, The McGraw-Hill International Series in Software Engineering, 1997, ISBN 0-07-707640-0

[Huang93] Y. Huang and C. M. R. Kintala, ``Software Implemented Fault Tolerance: Technologies and Experience", Proceedings of 23rd Intl. Symposium on Fault-Tolerant Computing, Toulouse, France, pp. 2-9, June 1993; Also appeared as a chapter in the book Software Fault Tolerance , M. Lyu (Ed.), John Wiley & Sons, March 1995

[Leve95] Nancy G. Leveson, Safeware-System Safety and Computers, Addison-Wesley, 1995, ISBN 0201119722, p. 436

[Lim94] Wayne C. Lim, "Effects of Reuse on Quality, Productivity and Economics," IEEE Software, IEEE Computer Society, September 1994, ISSN 0740-7459, pp 23-29

[Lyu95] M. R. Lyu, *Software Fault Tolerance* Chichester, England: John Wiley and Sons, Inc., 1995.

[Lyu00] Michael Lyu Handbook of Software Fault Tolerance, ISBN 0-471-93784-3, chapter 2

[March99] Steve March, "Learning from Pathfinder's Bumpy Start," Software Testing & Quality Engineering, Vol. 1, Issue 5, September/October 1999, pp. 10-12.

[Marc02] J. J. Marciniak. Encyclopedia of Software Engineering, John Wiley and Sons, 2002. pp 526-546

[Musa87] Musa, John; Iannino, Anthony; Okumoto, Kazuhira, Software Reliability: Measurement, Prediction, Application, McGraw-Hill, 1987, ISBN 0-07-044093-X, Appendix E

[Nat97] National Research Council. "Information Systems Trustworthiness Interim Report," National Academy Press, Washington, D.C., 1997, pp. 18-29.

[Neu95] P. G. Neuman. Computer Related Risks., Addison-Wesley, 1995, ISBN 0-201-55805-X, pp. 20-21

[Prac95] Practical Reusable UNIX Software, edited by Balachander Krishnamurthy, John Wiley & Sons, Inc., New York, 1995, pp. 5-8.

[Selb88] Selby, R. "Empirically Analyzing Software Reuse in a Production Environment," Software Reuse: Emerging Technology" W. Tracz (Ed.), IEEE Computer Society Press, 1988, pp. 176-189.

[Sha00] Lui Sha, "Using Simplicity to Control complexity," IEEE Software, July/August 2001, Volume 18, Number 4, IEEE 0740-7459/01, software@computer.org, page 27

[Siew82] Daniel Siweiorek and Robert Swarz, The Theory and Practice of Reliable System Design, Digital Press, Bedford Massachusetts, 1982, ISBN 0-932376-13-4 pp 7

[Stoyen97] Alexander D Stoyen, "Fighting Complexity in Computer Systems," Computer, IEEE Computer Society, August 1997,http://computer.org/pubs/computer/computer.htm, Volume 30, Number 8, pp. 47-48.

[Tai97] Tai, A.T., Chau, S.N., Alkalaj, L. and Hecht, H. "On-Board Preventive Maintenance: Analysis of Effectiveness and Optimal Duty Period," Proceedings of the 3rd International Workshop on Object-Oriented Real-time Dependable Systems (WORDS'97), February 1997, Newport Beach, CA, pp. 40-47.

[Wall94] Dolores R Wallace, and Laura M Ippolito, "A Framework for the Development and Assurance of High Integrity Software," National Institute of Standards and Technology (NIST) Special Publication 500-223, U.S. Dept. of Commerce, Dec. 1994, p. ix.

[Wall00] Mark Wallace, "Who Needs a Diploma? Why the high-tech industry wants dropouts," The New York Times Magazine, March 5, 2000, pp. 76-78.

[Yem91] Yemini, Yechiam, Goldszmidt, German and Yemini, Shaula. "Network Management by Delegation," Integrated Network Management, II, ed. Krishnan, Iyengar and Zimmer, Wolfgang, Elsevier Publishing Co., Inc., New York, 1991, pp. 95-107, ISBN 0-444-89028-9.
