NASA/TM-2008-215108

A Primer on Architectural Level Fault Tolerance

Ricky W. Butler
Langley Research Center, Hampton, Virginia

February 2008

The NASA STI Program Office . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/TM-2008-215108

A Primer on Architectural Level Fault Tolerance

Ricky W. Butler
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

February 2008

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 605-6000

Table of Contents

1 Introduction 3
2 Basics 3
3 Faults and Failures 3
3.1 Faults 3
3.2 Errors 4
3.3 Fault Tolerance Mechanisms 5
3.4 Fault Containment Regions 6
3.5 Design for Minimum Risk 6
4 Watch-dog Timers 6
5 Voting 7
6 Interactive Consistency 8
7 Clock Synchronization 10
7.1 Impact of Asymmetric Failures 11
7.2 Fault-tolerant Clock Synchronization Algorithms 12
7.3 Application-Level Reference Time 13
8 Diagnosing Failed Components 13
8.1 Detection Using Exact-Match Voters 13
8.2 Detection Using Thresholds 14
8.3 Detection Using Built-in-Test (BIT) 14
9 Real time Operating Systems and Fault Tolerance 15
10 Reconfiguration 15
11 Transient Faults 17
11.1 Distinguishing Transient Faults from Permanent Faults 17
11.2 Transient Fault Recovery 17
12 Self-Checking Pairs 18
13 Bus Guardian Units 20
14 Integrated Modular Avionics (IMA) 21
14.1 ARINC 653 21
15 Protecting a System from Common Cause Faults 22
15.1 Types of Common Cause Faults 22
15.2 Software Common Cause 23
15.3 Design Errors in Hardware 24
15.4 Radiation Induced Common Cause Failure 24
15.5 Other Common Cause Faults 25
15.6 Functional-level Dissimilar Backup System 26
15.7 Common Cause Failure and Integrated Modular Avionics 27
16 Re-Usable Fault Tolerance and System Layering 27
16.1 Asynchronous Flight Control Systems 28
16.2 Synchronous Fault-Tolerant Systems 30
16.3 Maintaining Independence between the applications and the fault-tolerant mechanisms 31
16.4 Technology Obsolescence and Refresh 32
17 Reliability Analysis 32
17.1 Markov Models 33

17.2 Solution of a Markov Model 34
17.3 The Impact of Long Mission Times 35
17.4 Beware of the Ambiguous Term “coverage” 36
18 The Synergism Between Formal Verification and Reliability Analysis......39
19 Function Migration..39
20 Vehicle Health Management 40
20.1 Basic Concepts...40
20.2 Failure Modes and Effects Analysis (FMEA) 41
20.3 Sensor Fault Tolerance 42
21 Concluding Remarks...43
22 Glossary...43
23 References...47

1 Introduction

Fault Tolerance is a deep subject with hundreds of sub-topics. It is often difficult to know where to begin the study of this vast subject. The purpose of this paper is to illustrate the key issues in architectural-level fault tolerance by way of example. The main objective is to explain the rationale and identify the trade-offs between the variety of techniques that are used to achieve fault tolerance. The primer focuses on high-level fault tolerance concepts (i.e. architectural) rather than low-level mechanisms such as Hamming codes or protocols used for communication. For information about the latter the reader is referred to [Pradhan86].

2 Basics

Fault Tolerance is founded on redundancy. If we have two or more identical components we can ignore the faulty component or switch to a spare if the primary fails. Of course this assumes that we know when the failure occurs. Some failures are easy to detect, e.g. the device just stops working. Other failures are not, e.g., the device continues to work but produces incorrect results. So immediately we are confronted with one of the reasons that the fault tolerance field is broad—systems are designed to handle different kinds of failures. Some systems are designed to handle fail-stop faults. Others are designed to handle any kind of fault. Still others are designed to handle faults that can be detected via a diagnostic program. Sometimes the faults are always assumed to be the manifestation of a physical disruption, while other system designs seek to survive logical errors as well. There are many possibilities. Many designs seek to survive the class of faults that are assumed to be common and provide little or no capability against what are assumed to be less common failures. Ideally, the set of faults handled by a system is delineated in a well-specified fault model.

3 Faults and Failures

3.1 Faults

A *fault* is a defect in the hardware or software that can lead to an incorrect state. Faults can arrive randomly from the physical failure of hardware (e.g. stuck-at-one bit) or a bit-flip in memory due to electromagnetic upset or they can arrive in a non-random manner if they are due to manufacturing defects or logical mistakes in a design. System *failure* occurs when the delivered service deviates from the correct service.

When developing a fault-tolerant system, the designer makes assumptions about the types of faults that must be handled. This is often referred to as the system fault model. The fault model elaborates all of the assumptions about how components of the system can fail. The following types of faults are often considered:

- Fail stop (or fail silent) -- the component stops producing outputs when it fails
- Fail benign – the component's failure is recognized by all non-faulty components
- Fail symmetric – the fault results in the same erroneous value being sent to all other replicates
- Fail asymmetric (Byzantine) – the fault results in different erroneous values being sent to some of the other replicates

See [Thambidurai88] for more details about this classification.

Also the duration of the fault may be considered:

- Permanent faults – once they occur they do not disappear
- Transient faults – appear for a short time and then disappear (e.g., upset from electromagnetic interference).
- Intermittent faults -- they appear, disappear and then reappear

However, it should be noted that although a fault is transient, it can still produce permanent error in the system state if the system is not designed to handle transients.

Some defects or events can trigger multiple simultaneous errors. This class of fault is referred to as *common cause faults* (CCF) and if not mitigated they may overcome all of the available redundancy and hence cause system failure. Sources of common cause faults are many and varied and require special consideration in the design of a fault tolerant system. The following is a partial list of common cause faults:

- Design flaw (i.e. bug) in software
- Hardware design errors (e.g., logical error in a processor)
- Compiler error
- Manufacturing defects
- Environmental induced effects (e.g. MMOD, Lightning, EMI, Launch shock/vibrations)

Computer systems can also be vulnerable to common-mode failure if they rely on a single source of power or any other needed resource.

3.2 Errors

A rigorous definition of an error is non-trivial. Intuitively it is the manifestation of the fault in some visible state of the system that you actually care about. The standard IEEE definition states that an error impacts a service. Avizienis, Laprie, Randell, Landwehr [Avizienis04] write

> Since a service is a sequence of the system’s external states, a service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is called an error… In most cases, a fault first causes an error in the service state of a component that is a part of the internal state of the system and the external state is not immediately affected. For this reason, the definition of an error is the part of the total state of the system that may lead to its subsequent service failure. It is important to note that many errors do not reach the system’s external state and cause a failure. A fault is active when it causes an error, otherwise it is dormant.

An error is *detected* if its presence is noted by the system via an error message or error signal. Errors that are present but not detected are *latent* errors.

Note: a fault is usually defined fairly broadly as a defect within the system. This definition includes “software bugs” in addition to physical failures such as a memory “stuck-at-1” fault. It also includes the bit flips induced by noise in communications systems. Because the techniques used to detect and remove software bugs (e.g. logical mistakes) are very different from the techniques used to handle physical faults, it is very important to distinguish situations where one is talking about physical faults and when one is talking about “design errors”.

3.3 Fault Tolerance Mechanisms

The primary goal of fault-tolerance is to prevent errors from leading to system failure. The fault tolerance functionality of a system is sometimes decomposed into the following:

1. Fault *masking* – preventing an error from propagating to a system output.
2. Error *detection* -- observing a difference between system state and the expected state.
3. Error *recovery* – an attempt to restore the system state to an error-free state.
4. *Reconfiguration* – removing a faulty component from the system

The techniques used to achieve this functionality is a major focus of this paper.

It is very important to recognize that redundancy alone does not provide fault-tolerance. There must be mechanisms that coordinate and control the redundancy and make decisions and selections concerning the redundant information. These mechanisms may be centralized or distributed, they may be implemented in hardware or software, they may be simple or complex, but it is absolutely essential that these mechanisms be designed correctly. If there is a logical defect in the design of the redundancy management logic, system failure can occur even when there is no physical failure in the system. For example, most of the problems which were discovered in the AFTI F-16 flight tests at Dryden were due to defects in the redundancy management logic [Mackall88].

3.4 Fault Containment Regions

Fault tolerant systems are often built around the concept of fault containment regions (FCRs). The primary goal of a FCR is to limit the effects of a fault and prevent the propagation of errors from one region of the system to another. A FCR is a subsystem that will operate correctly regardless of any arbitrary fault outside the region. FCRs must be physically separated, electrically isolated, and have independent power supplies. Physical dispersion limits the effect of physical phenomena such as the impact of a micrometeoroid. Electrical isolation protects against fault propagation from lightning or other forms of static discharge. Power supply isolation prevents a power failure affecting the entire system. The number of FCRs in a system is a primary factor in determining how many faults a system can tolerate without failure.

3.5 Design for Minimum Risk

Design for Minimum Risk (DFMR) is a process that allows safety-critical mechanisms to claim adequate fault tolerance through rigorous design, analysis, testing, and inspection practices rather than through true physical redundancy. DFMR is intended as an option when physical redundancy is highly impractical or too expensive. DFMR is primarily used on mechanical systems and is not intended for computer systems. Johnson Space Center policy [MA2-00-057] permits the use of “fully compliant simple mechanical systems” without redundancy in safety-critical applications when they meet certain special requirements AND are approved by the Mechanical Systems Working Group and the safety review panel. More information is available at

http://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_8705_002A_&page_name=AppendixB
http://mmptdpublic.jsc.nasa.gov/mswg/Policies.htm#DFMR

and from [Stephans04].

4 Watch-dog Timers

A well known type of system failure is the non-responsive system or locked-up system, which is sometimes referred to as the “blue screen of death” in desktop computers. This is usually the result of a software bug that throws the system into a non-operable state (e.g. an erroneous jump into the data space). When this occurs in a desktop computer, a simple reboot usually suffices to recover the system. But in some safety-critical systems, operator restart is not available or

appropriate (e.g. requires a response time that is beyond normal human capability). In these systems, a watch-dog timer can be helpful.
The simplest approach is to have some watch-dog subsystem observe “I’m alive” messages which are periodically produced by the primary. If these messages disappear then the watch-dog system initiates some recovery action such as reboot or rollback to a previous safe state. It is also possible to have the watch-dog system turn over control to a backup system. There are several issues here, but the most important is how to protect the system from failures in the watch-dog subsystem.

5 Voting

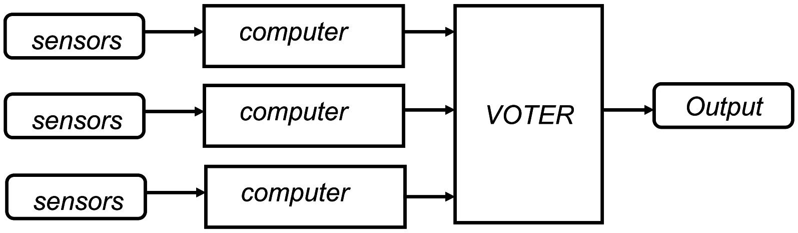
Fault tolerant systems are usually designed to handle more than just fail-stop faults, so inevitably some form of voting must be employed. There are many varieties, but there are roughly three basic categories of voting: exact match, fault-tolerant average, and mechanical. In the exact match case all of the replicates including their internal data are assumed to be identical so any bit in an output that differs may be an indication of failure. One can simply select the majority value and use that value for the system output. In averaging, the good replicates are assumed to be “close” to each other but not identical. So the medial values either come from non-faulty components or they are bound by other good values[1]. Either way the selection of a median or an average of some middle values is enough to mask a faulty component

Mechanical voting can be performed at the actuators of a system. Each replicate provides a fraction of the “force” needed to actuate the physical component. These forces are added mechanically. In this way a failed component can be out-forced by the other good components. Though the force will be different than nominally commanded, this is compensated by the use of feedback control.

Voting can also be used to detect or diagnose which component of the system is faulty, but careful engineering is required to determine when a disagreement in a vote may be used for the diagnosis of faults. This is discussed in Section 8 (Diagnosing Failed Components).

The following diagram illustrates the concept of voting:

[1] By assuming that there is only one faulty component, we have two cases: (1) the faulty component is the middle value, in which case it is between two good values, so the faulty component is still producing an acceptable output, or (2) the faulty component is one of the extreme values, in which case the middle value came from a good component.



However, it should be noted that voting is often implemented as a distributed algorithm that executes on the redundant computers themselves. Also voting is not just employed at the outputs of fault tolerant systems. System inputs (i.e. sensor inputs), and intermediate results can be voted as well. It is not uncommon for the voting to occur in software when the underlying hardware has sufficient interconnect. When software voting is done, information is passed between processors and then voting algorithms are used to select a result. If not designed properly, the logic for exchanging and voting can be another source of faults, namely design defects. Some fault-tolerant systems are designed to maintain exactly the same data in the redundant lanes, while other systems seek to maintain close but not exact agreement in the lanes. See Section 16 (Re-Usable Fault Tolerance and System Layering) for a discussion of the pros and cons of these alternatives.

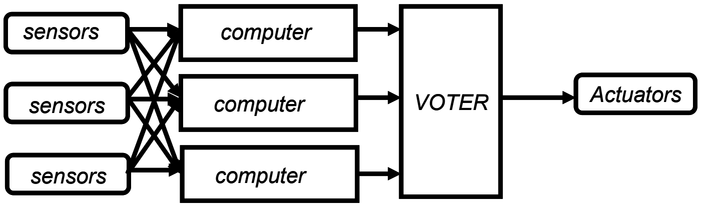
6 Interactive Consistency

A fundamental issue in a fault-tolerant system is how to distribute input data to the replicates in a manner that preserves consistency. Because inputs begin as single source data, we must be sure that the replicate consistency is not destroyed by a failure in the distribution system.

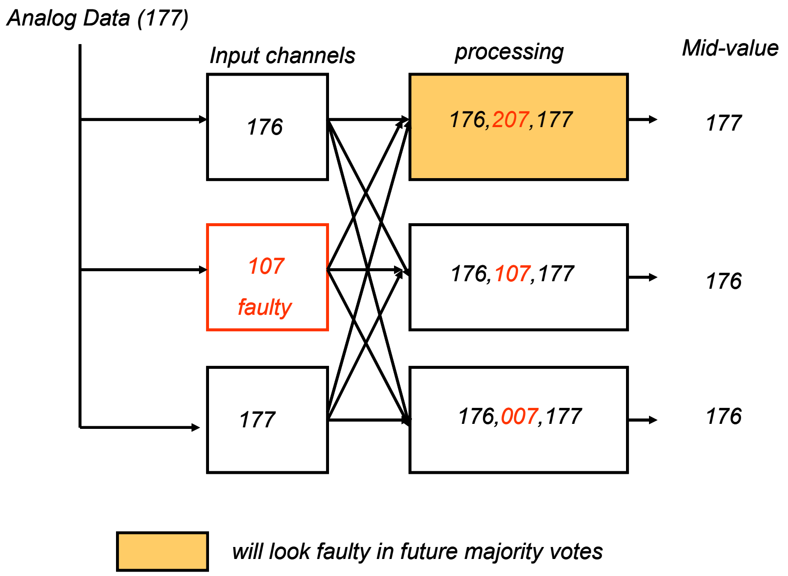
Different methods can be used to distribute sensor values to the replicates

- Single sensor sampled by all processors
- Single sensor sampled by a processor that distributes the value to all other processors
- Redundant sensors each sampled by all processors
- Redundant sensors each of which is sampled by a single processor that distributes its value to all other processors

Even if there are multiple sensors the individual sensor values are distributed to each of the computers:



The following diagram illustrates the problem with asymmetric failure

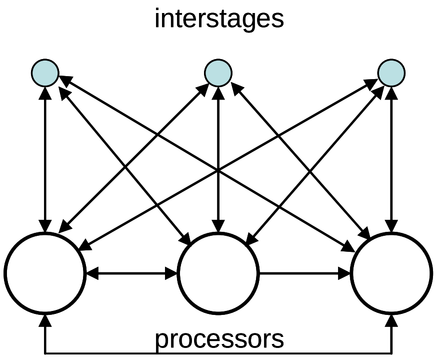


So after the failure one good channel will be operating on a value of 177 and another on a value of 176. If exact-match voting is used on the outputs, then the wrong processor will be identified as failed. If mid-value select voting is used on the outputs, then there will be no immediate problem. However, if the asymmetric fault persists over many iterations, the deviation between the two good channels can continue to get larger and larger and eventually exceed the threshold set for fault diagnosis. Once again the wrong processor can be reconfigured out of the system. Recent results at NASA Langley (Paul Miner) have shown that a two-stage mid-value select can be designed to achieve *Byzantine Agreement*. However most asynchronous systems today have not been designed using this particular strategy.

As shown above some types of faults are more insidious and require more FCRs to properly mask them. For example, it has been shown [Driscoll03] that tolerating *f* Byzantine faults requires:

- 3*f*+1 Fault Containment Regions (FCRs)
- FCRs must be interconnected through 2*f*+1 disjoint paths
- Inputs must be exchanged *f*+1 times between FCRs
- FCRs must be synchronized to a bounded skew

Consequently a simple triple modular redundant (TMR) system is not *Byzantine Resilient,* that is, there are some single faults that can defeat it. But it is important to realize that the FCR need not be a complete processor. All that is needed is enough circuitry to perform the interactive consistency algorithm. To withstand a single asymmetric fault, one needs four fault isolation regions, but they need not all be full channels. A very simple circuit can provide the necessary functionality to implement critical Byzantine Agreement algorithms. If the circuit is electrically isolated it can serve as one of the fault isolation regions.   This idea was first exploited in the Draper FTP computer system:



A triplex FTP contains six components, but only three of them are full processors. The three interstages are simple circuits requiring approximately *50* gates***.*** Thus, the overall hardware complexity, and hence the fault-arrival rate, of a triplex FTP is less than a quadruplex. So its cost and reliability are correspondingly better than using six complete processors. Furthermore, a quadruplex FTP provides fault tolerance comparable to a 5-plex, but with only four full processors.

This idea of utilizing special hardware to off-load the redundancy management functions from the computing resources has gained popularity in recent years. This strategy has been used in SafeBUS, TTP/C, FTPP, and SPIDER.

The beauty of modern fault-tolerance is that you don't have to replicate entire "strings" to increase your ability to withstand more types of faults. To withstand a single asymmetric fault, one needs four fault isolation regions, but they need not all be full channels. A very simple circuit called an interstage can provide the necessary functionality to implement critical Byzantine Agreement algorithms. This was first done by Draper Labs in their FTP architecture [Lala]. Since the interstage is electrically isolated it can serve as one of the fault isolation regions. It merely relays messages and does not participate in the voting steps that provide fault-tolerance in FTP. Full protection from a Byzantine fault is achieved with only three processors and three interstages. The Draper FTPP built on this foundation and developed a parallel processing version [Harper91].

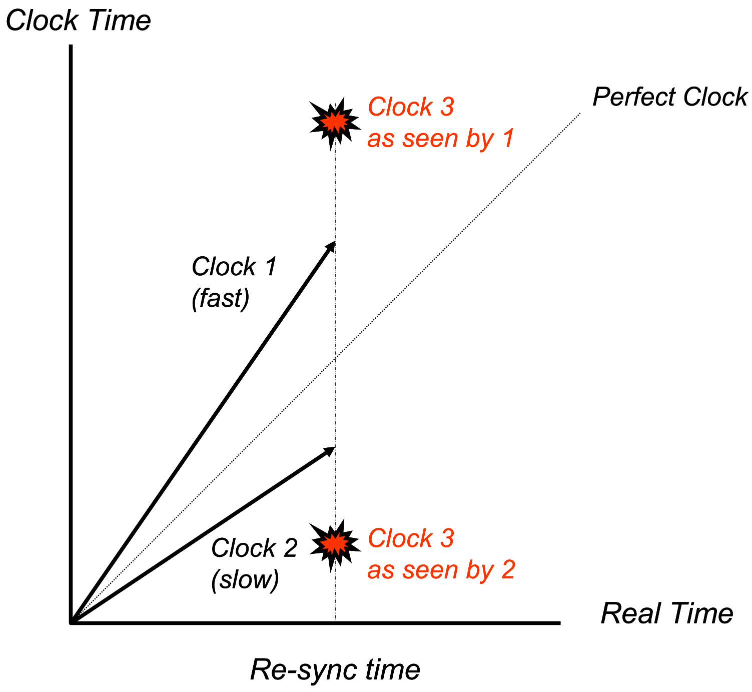
7 Clock Synchronization

It is usually advantageous to provide many layers of voting in a fault-tolerant system and not just rely on a final force-sum voting at the actuators. If the system is designed to perform a vote within the processors itself, it is necessary that there be some mechanism to synchronize the clocks of the system so that this vote can be scheduled and accomplished. Fault tolerant systems that synchronize the clocks on the redundant processors are called *synchronous* fault-tolerant systems. Systems that do not synchronize the clocks are called *asynchronous* fault-tolerant systems.

The algorithms that perform clock synchronization are inherently distributed algorithms because the algorithm would not be fault-tolerant if it were based on some centralized timer. Fault-tolerant clock synchronization algorithms are notoriously difficult to get right. Simple intuitive solutions don't always work.

7.1 Impact of Asymmetric Failures

An intuitive yet vulnerable approach is to build a fault-tolerant clock using three different clocks that are fully connected. At a pre-determined fixed rate, each clock sends its value to the other clock. Each clock then examines the arriving clock values and updates its own clock to the mid value of the three. So if there is only one faulty clock, it is either one of the outer values or it falls between two good values. Either way the selection of the mid value is appropriate. But this analysis overlooks the impact of an asymmetric failure, i.e. where the failed clock sends one value to one of the good clocks and a different value to the other good clock. This is illustrated in the following figure.



Here both clocks 1 and 2 are non-faulty, but clock 1 is faster than real time and clock 2 is slower than real time. When clock 3 fails it sends clock 1 a value bigger than clock 1 and sends clock 2 a value smaller than clock 2. Therefore both clock 1 and clock 2 select themselves as the mid value and continue to drift apart. If clock 3 continues to do this, clocks 1 and 2 can drift arbitrarily far apart even though they continue to execute the synchronization algorithm.

7.2 Fault-tolerant Clock Synchronization Algorithms

Many different algorithms have been developed which achieve fault-tolerant clock synchronization. Because of the subtleties associated with these algorithms they are usually accompanied by formal proofs of correctness [Ramanathan90]. A proof is provided that the algorithm maintains all of the good processor's clocks to within a small skew of each other, often denoted as ε[2]:

$$|C_p(t) - C_q(t)| < \varepsilon$$

Where $C_j(t)$ denotes the clock time of processor *j* at real time *t*. Once you have this implemented in your system, the voting system can be reliably built on top of this as follows:

1. A vote is scheduled at a predetermined time (usually in a static schedule table)
2. Let D = the maximum transport delay in sending a clock value from one processor to another over the communication system[3].
3. The vote is delayed until $D + \varepsilon$ time units after the scheduled vote time. In this way all of the good processors are guaranteed to have good values from all of the good clocks.

Fault-tolerant clock synchronization algorithms are based on the idea of periodic resynchronization. At a predetermined cycle time, all clocks exchange their current values. Then each local clock executes some filtering and/or averaging function to calculate a new clock value. It then updates its own local clock value with the new value. Most of these algorithms are simple to describe but the proof that the algorithm works correctly in the presence of an arbitrary failure can be very difficult [Rushby89].

[2] The bound on the clock skew is determined by the **variance** of the communication times and not the mean. The mean can be subtracted out in the clock sync algorithms.

[3] Note that this means that communication system must have a maximum delay. If a databus is used it must have predictable behavior and bounded communication delays. Therefore fault-tolerant systems are developed using TDMA rather than CSMA/CD communication protocols.

7.3 Application-Level Reference Time

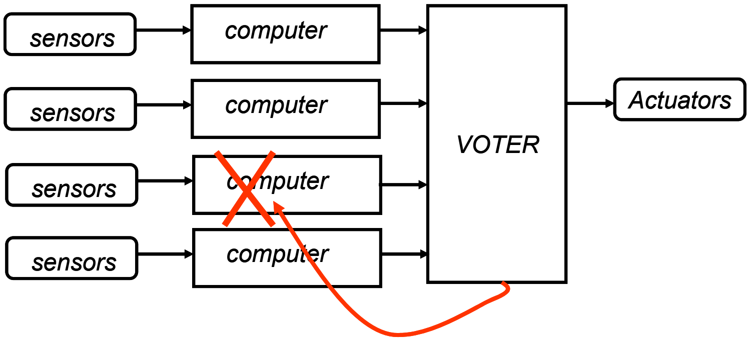
Although we have talked about clock synchronization between replicate computers, we have not addressed the issue of obtaining a clock time that is synchronized to an external time source. This is important if the application level software is event-driven by external events. If the fault-tolerant system must interact with other external systems and this interaction is based upon some external time, then the operating system must provide this time source to the applications.

The NIST Internet Time Service can be used to synchronize the clock of any computer connected to the Internet. However, due to the unreliability of the internet, this is not a suitable candidate for a safety-critical system. For a safety-critical system the Global Positioning System (GPS), which is a used for navigation throughout the world, is more suitable. GPS signals are derived from atomic clocks on the satellites so they can be used as a reference for time synchronization and frequency calibration.

8 Diagnosing Failed Components

There are 3 basic methods for diagnosing when a component has failed:

- Using the discrepancies in the exact-match votes
- Thresholds on the mid-value select voting
- Using built-in test (BIT)



8.1 Detection Using Exact-Match Voters

The most straight-forward mechanism for detecting a failed processor is the exact-match voter. If the lanes of the fault-tolerant system are synchronized, then system can use exact-match, majority voting. Detection of the faulty lane is simple because the correct lanes all agree bit by bit. Any deviation from the majority value is an indication of failure. But it should be noted that this strategy depends critically upon the use of *interactive consistency* algorithms to properly distribute single source data to the replicates. See Section 6 (Interactive Consistency).

8.2 Detection Using Thresholds

If the lanes (i.e. FCRs) are asynchronous, then the inputs will be sampled at slightly different times and so the memory states of the different lanes will not be exactly the same. In this case the system must use a mid-value select or averaging algorithm. Unfortunately, mid-value selection algorithms alone do not provide a way to detect and isolate a faulty lane. To accomplish this, thresholds must also be used. A threshold is the maximum amount of deviation from the average value that is tolerated before a lane is declared to be faulty. The system designer must set thresholds so that most failures are detected, but he must be careful not to make them too tight so as to cause excessive false alarms. Consequently it can be very difficult to determine appropriate levels for the thresholds and it requires extensive testing [Mackall88].

8.3 Detection Using Built-in-Test (BIT)

Built-In-Test (BIT) mechanisms run automatically and seek to isolate faulty components without the use of redundancy. There are two basic types: power-on bit (PBIT) which executes at startup and continuous BIT (CBIT) that runs continuously on a system. CBIT can be periodically scheduled or executed whenever there is any slack time available. Built-In-Test can be implemented in software or directly on an integrated circuit. The latter is sometimes referred to as *circuit-level* Built In Self Test (BIST) and is characterized by well-developed standards (e.g. IEEE 1149).

The ideal BIT would be one that could detect every possible fault, but 100% *coverage* is rarely achievable in practice. It is not uncommon to see requirements on the order of 98% coverage, that is, the BIT can detect 98% of all possible faults. Also most BIT mechanisms also produce some level of false alarms. Because of the imperfection of BIT, it is not usually used as the first line of defense in most safety-critical systems. Rather it used to augment the functionality of the redundancy management system and aid in identifying a faulty component. BIT is also extremely important for off-line maintenance and repair.

9 Real time Operating Systems and Fault Tolerance

Traditional real time operating systems (RTOS) do not provide direct system services that manage redundant processes. Nevertheless, custom software can be developed that accomplishes this task. A primary goal of this custom software is to hide the details of the process management and voting. The application software should be designed as if there were only simplex versions of the software. All of the details about replication and voting should be hidden from the applications. Nevertheless, the applications will have visibility into whether the simplex abstractions are preemptible or non-preemptible. If the tasks are preemptible, then an additional challenge must be met, namely how can I be assured that all of the critical tasks meet their hard real-time deadlines. While this is the classic problem that RTOS's solve, it should be noted that in this situation the RTOS will be scheduling redundant tasks with a strict need to vote their outputs at task completion. In the non-preemptible case, this problem is usually solved by using a preplanned static table. This is what was done in SafeBUS and TTP/C [Rushby01].

10 Reconfiguration

Once a component has been identified as faulty, it is sometimes advantageous to remove the faulty component from the system through a reconfiguration process. Reconfiguration is built around the idea of "ignoring the outputs" of components that have been diagnosed as faulty by the community of good processors (i.e. the *working set*). There are diagnosis algorithms that work in the presence of even malicious asymmetric failures yet guarantee that all of the good processors have a consistent view of who is faulty as long as a majority of the processors are working. It is usually not a good idea to design the system in a manner that allows a component to shut down (e.g. power-down) another component. Failures in the shut-down circuitry can lead to the removal of good components. Of course it is often appropriate to provide manual means for the human operators to accomplish this task.

A system that can survive a single fault is sometimes referred to as one fault-tolerant. A system that can survive two faults is referred to as two fault-tolerant[4]. A two fault tolerant system is usually constructed using reconfiguration, however a five-plex with five-way voting can mask two simultaneous faults.[5] A four plex that reconfigures to a three-plex is an example of a two fault-tolerant system.

[4] The N fault-tolerant characterization is very crude. The use of actual reliability numbers is much to be preferred. A one fault-tolerant system can be more reliable than a two fault-tolerant system if the latter has a higher processor failure rate.

[5] This assumes that there are at least two additional fault containment regions for solving the interactive consistency problem in order to be sure that we are guaranteed to have replicate consistency in the 5-plex.

Because the reconfiguration process relies on fault-detection, it is usually desirable to augment the detection by voting with built-in test. If the system reliability requirements do not warrant reconfiguration, then built-in test may not be needed, since the voters can provide the needed fault masking capability. However, in a reconfigurable system it is desirable to minimize fault-latency (i.e. the time period from fault arrival to its manifestation as an error) because while the first fault is latent, a second fault may arrive in another component. The simultaneous presence of two faulty components will defeat the typical 3-way voter. The time that it takes for the system to detect, isolate and reconfigure a faulty component directly impacts its reliability. Interestingly, a two-fault tolerant system can actually have a lower reliability than a one-fault tolerant system, if its reconfiguration time is poor or if its components have much higher failure rates. Therefore it is desirable to specify the system fault tolerance requirements in terms of a probability of system failure rather than as simply one or two-fault tolerant.

It is important to note that the view of the working set must be consistent throughout the system. This is sometimes referred to as the *distributed diagnosis* problem. It is essential that one verifies that all good components agree who is in the working set. In system initialization or in recovery from a massive transient upset, establishing this *working set* is the fundamental challenge. If the system employs *smart actuators*, it is necessary that the computers systems in these smart actuators know what is in the “working set” so that the physical force-sum voters can know which inputs to ignore.

Reconfiguration can be accomplished by (1) removing a faulty processor and hence reducing the level of redundancy or (2) by replacing the faulty processor with a spare processor. These spares can be either hot or cold. A *hot spare* is one which executes continuously in parallel with the processors which are currently in the working set. The use of hot spares minimizes the reconfiguration time because the state of the spare is already loaded when it is needed. However, hot spares increase the amount of power consumed. Also, the spare is subject to the same operating conditions as the working set processors and can fail just as the active ones. A *cold spare* is powered off until it reconfigured into the working set. Cold spares do not use power and are usually assumed to have a zero (or low) failure rate while inactive.

Reconfiguration is used to increase the reliability of the system without adding more redundancy. The effectiveness of this strategy depends upon how fast a faulty FCR can be removed from the system. See section 17 (Reliability Analysis) for more details about this. A more crude method of characterizing the level of fault-tolerance is the “N fault tolerant” terminology. A “N fault tolerant” system is a system that is still operational after N consecutive faults (not simultaneous). For example, a two fault tolerant (2 FT) system is a system that is "fail operational, fail operational", i.e. after two sequential faults, the system is still a functioning system. This definition implicitly assumes that there is adequate time for the system to reconfigure from the first fault before the arrival of the second fault. Without reconfiguration in a quad system, the arrival of the second fault can create a 2-2 dilemma case that can lead to directly to system failure. For example, if the two good processors report that the Boolean variable `release_parachute` is `true`, while the two faulty processors report that the value is `false`, the voting logic could easily pick the wrong value.

11 Transient Faults

The reliability of a fault-tolerant system depends upon a reasonably fast detection of faults to ensure that two faults are not active at the same time. The detection and removal of faults is complicated by the presence of transient faults. Although a transient fault can corrupt the internal state of the system and must be handled, it does not permanently damage the hardware. Therefore the recovery mechanism is different. You want to restore the state of the memory but you do not want to remove hardware. So the redundancy management system must make a judgment about whether a fault is permanent or transient.

11.1 Distinguishing Transient Faults from Permanent Faults

There are two basic techniques:

- If error persists, declare the fault to be permanent
- Reconfigure a faulty component immediately upon indication of failure, run built in test (BIT) diagnostics, and then re-integrate if it passes the BIT diagnostics.

When using the first technique, the designer has to decide on an appropriate time period (or an error count level) before declaring a fault to be permanent rather than transient. Surprisingly the reliability of a typical N-modular redundant (NMR) system is not very sensitive to this parameter. Even a delay of a few minutes does not degrade the system reliability [DiVito90].

11.2 Transient Fault Recovery

Even though a fault is transient, it can still corrupt memory. The process of correcting memory errors is sometimes called *scrubbing.*

Several approaches can be used to recover the state of memory in a transiently affected digital processor. The most common method is through the use of error-correcting codes (ECC) which take advantage of extra bits in the memory.[6] Another approach is to rely on the reading of new inputs to replace corrupted memory. Of course, this does not give 100% coverage over the space of potential memory upsets, but it is much more effective than one might expect at first glance. Since control-law implementations produce outputs as a function of periodic inputs and a relatively small internal state, a large fraction of the memory upsets can be recovered in this manner. This accounts for the fact that although many systems in service are not designed to accommodate transient faults, they do actually exhibit some ability to tolerate such faults.

Another important technique is the use of a watchdog timer. Since a transient fault can (and frequently does) affect the program counter (PC), a processor can end up executing in an entirely inappropriate place -- even in the data space. If this happens, then the previous technique becomes totally inoperative. The only hope in this situation is to recognize that the PC is corrupted. A watchdog timer is a countdown register that sets the PC to a pre-determined “restart" location if the timer ever counts down all the way to 0. In a non-transiently affected processor, the watchdog timer is periodically reset by the operating system. Once a fault has been detected by a watchdog timer, the entire system may be “rolled-back" to a previous state by use of a checkpoint -- a previous dump of the dynamic memory state to a secondary storage device of some kind. This technique has not been used very often in flight control systems because of the unacceptable overhead of this type of operation.

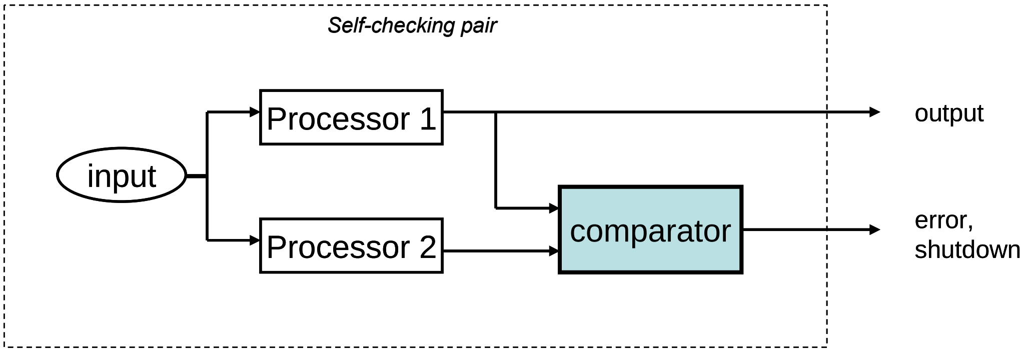
A more appropriate technique is the use of majority-voting to replace the internal state of a processor. It is important to note that this is done continuously rather than just after a transient fault is detected. Of course, such voting can be expensive if the dynamic state is not small. The use of hardware-based memory-management units (MMUs) can also mitigate the impact of transient faults by containing the propagation of an error to within a single memory-mapped region.

12 Self-Checking Pairs

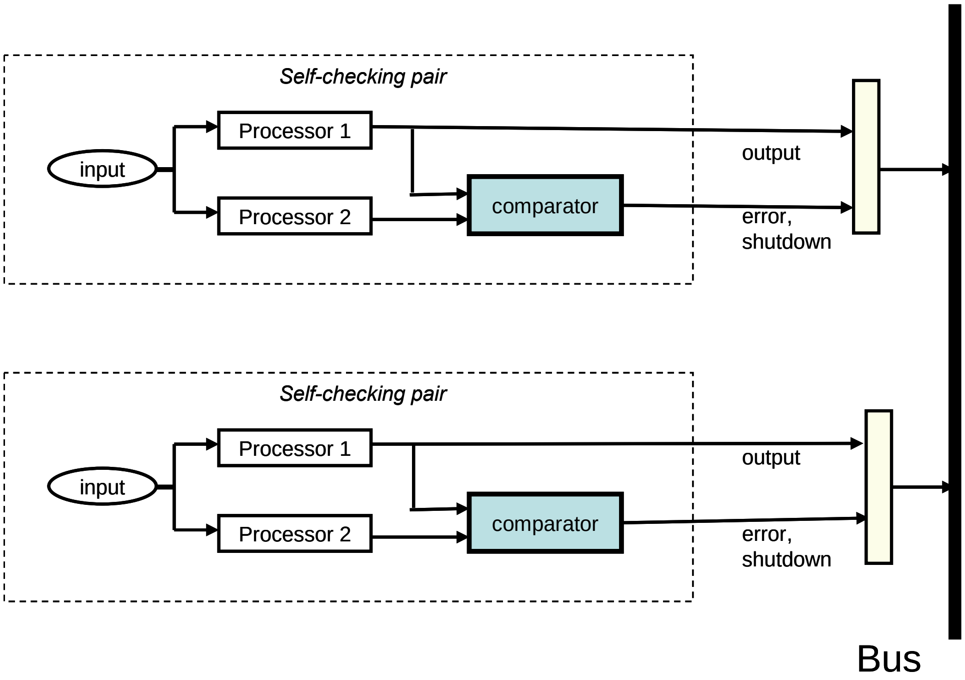
A fault-tolerant system can be built on the foundation of self-checking pairs. The use of self-checking pairs brings several key advantages, but this is not without some additional cost. The key benefit is that self-checking pairs can greatly simplify the design of certain aspects of the system by providing a high probability that faults will manifest themselves as “fail-stop”. However, this comes at the price of a more inefficient reconfiguration process which wastes more good hardware than the more traditional NMR approach.

[6] Single Error Correction Double Error Code (SECDEC) is commonly used for spacecraft systems. However, the processor’s internal caches often do not have this level of correction. Periodic cache flush is sometimes used or a flush on error detect is employed.

The basic idea of a self-checking pair (SCP) is to combine two identical processing elements which are given identical inputs and their results are compared. If there is a mis-compare of their outputs, then the comparator circuit shuts-down the SCP and prevents the output from leaving the SCP. This can be a temporary shutdown (e.g. the current output only) or a permanent shutdown. Because it is usually not desirable to permanently remove a SCP due to a transient fault, a persistence counter can be used which delays permanent shutdown until a number of consecutive faults have occurred.



To achieve system-level fault-tolerance, self-checking pairs must be organized together so that when a self-checking pair shuts itself down, its function can be taken over by another self-checking pair. Conceptually, this can be pictured as follows:



This type of architecture is often referred to as Dual-Dual. The key advantage of this strategy is that it provides low level, autonomous processor fault detection and isolation. There is no need to design voting mechanisms and high-level strategies for redundancy management.

Of course there must be some mechanism to handle the transition from one self-checking pair to another. Under the assumption that a self-checking pair fail stops, the bus can just accept the first broadcast from either one of the pairs or search for a valid output in some pre-determined order. Alternatively one of the pairs can be active while the other one “shadows” the active one. Either way you are employing four-fold redundancy to avoid the problem of loading the data state after the primary fails. But of course this only takes care of faults in the processing element. What about failures in the connection from the processor to the bus? To deal with these failures we need something more sophisticated – bus guardian units (see next section).

Although systems designed with self-checking pairs require more redundancy than a corresponding NMR approach, the software that manages the redundancy is much simpler. Significant cost savings may accrue from not having to design detection and isolation mechanisms and from the reduced software verification and validation effort.

13 Bus Guardian Units

A bus guardian unit protects the inter-processor communication channels (e.g. bus) from transmission faults. A bus guardian can also serve a “repeater function” to strengthen the signal from a processor and thereby protect the bus from signals that are degraded and potentially ambiguous. The bus guardian unit protects the system from the “babbling idiot” phenomena where a failed transmitter floods the communication subsystem with illegal transmissions. When the communication mechanism is a time-division multiplex bus, the bus guardian protects the bus by preventing a transmission outside of its pre-determined time slot and thus providing fail-silence of the self-checking pair and its connection to the bus.

A fundamental challenge in designing a system using bus guardian units and time-division multiplex communication is to provide a consistent time-source to all of the bus guardian units so that they can provide properly enforce the time-slot for its self-checking pair. This time must be synchronized with all of the rest of the other bus guardian units and so in many modern fault-tolerant systems, a fault-tolerant clock signal is generated in the communications subsystem and provided to the bus-guardian units and processor pairs. Clearly the bus guardian unit cannot be triggered off of its own local processors clock. A single failure in this clock would enable the processing pair and bus guardian unit to access the bus at an illegal time.

The bus guardian unit must protect the bus from any transmission outside of its allocated time slot, while not keeping the self-checking pair from transmitting at correct times. Because the fault-tolerant clock signal cannot be assumed to be perfect, the bus guardian must open the window slightly earlier than the start of the time slot, and close it slightly later than the expected end of its time slot. The system designer must be careful to make sure that these small excesses do not overlap with the excesses of other units. So there has to be some wasted bandwidth of the bus to achieve fault-tolerance this way. The selection of these time intervals must be analyzed carefully to insure that the system functions properly.

14 Integrated Modular Avionics (IMA)

Physical separation of critical avionics functions from less critical functions has been the primary strategy used by the designers of civil aircraft to produce safe avionic systems. Traditional avionics systems are built around federated architectures in which each processing site contains a single application such as an autopilot, flight management system, or display. Typically, each application is assigned a single level of criticality and all software and hardware at that processing site is verified and certified to that level of criticality. Critical functions are protected from non-critical tasks by physical isolation. NASA has used this approach on interplanetary spacecraft, where functions critical to the survival of the spacecraft are handled by an attitude and articulation control system that is separate from the systems that control the science experiments.

Physical separation of criticality in avionics, however, is a costly approach. Each piece of hardware must be separately supported and maintained. Furthermore, each of these functions (hardware/software) is certified in isolation; there is no notion of a common hardware platform that has been certified and can thus be used for multiple functions. The concept of a certified reusable computer (probably multi-processor) that could execute multiple applications with different levels of criticality is commonly referred to as Integrated Modular Avionics (IMA). Because of the utilization of IMA in the Civil Aviation Industry, information is available about IMA application in safety-critical systems. IMA may also have some special advantages for space applications, where power, weight, and volume are of particular concern. By hosting many applications on the same platform, some of which run at different times than others, the total amount of hardware needed can be reduced and consequently there will be weight and volume savings and perhaps some power savings as well.

14.1 ARINC 653

The API (ARINC 653) defines an APplication EXecutive (APEX) for space and time partitioning that is gaining support in the commercial avionics markets.

Several vendors currently offer ARINC 653 compliant real-time operating systems including the LynxOS®-178 RTOS, Green Hills Integrity-178, VxWorks653, and the BAE Systems CsLEOS, the first two of which have been used and certified under the FAA's DO-178B guidelines. Each partition in an ARINC 653 system represents a separate application and makes use of memory space that is dedicated to it. Similarly, the APEX allots a dedicated time slice to each, thus creating time partitioning. Each ARINC 653 partition supports multitasking. The advantages are:

- Uses a simple approach to memory and time partitioning and provides support for common I/O through a well-defined application program interface (API).
- Simplified maintenance through the ability to modify or add hardware without impacting application software.
- Enhanced safety assurance through software fault isolation. For example, software faults in one partition cannot corrupt the memory space in another partition or impact the timing of another partition.
- Better use of computing resources (as compare to federated system) results in reduction of mass and power
- Scalable: the centralized processing function can be distributed in multiple computers (cabinets)

It should also be noted that a fault-tolerant operating system can provide different levels of fault tolerance on the same platform. The tasks can be dispatched at different levels of redundancy. Some tasks can run as simplex tasks while others may be triplex or higher.

15 Protecting a System from Common Cause Faults

There are some failure mechanisms than can impact multiple fault containment regions all at once and can thus create multiple errors in these separate regions at the same time. These kinds of faults are often referred to as common cause faults (CCF) or generic faults and the associated phenomena is sometimes referred to as common mode failure. We will use the term common cause fault exclusively to prevent confusion. Because common cause faults can completely undermine a fault-tolerant system, system designers give considerable attention to this class of faults.

15.1 Types of Common Cause Faults

Sources of common cause faults are many and varied and require different mitigation strategies. Examples that can lead to common cause failure are:

- Logic errors in software (i.e. programming bugs, requirements oversights)
- Hardware logic errors (i.e. bug in the instruction set of a processor)

- Compiler/loader programming error
- Manufacturing defects
- Environmental induced effects (i.e. MMOD, Lightning, EMI, total dose radiation)
- A Byzantine asymmetric fault in a system not designed to handle such faults

A valuable resource for learning more about CCF is the SAE ARP 4761, which outlines guidelines for conducting safety assessment for civil airborne systems. This standard enumerates a total of eight common mode types, twenty two common mode subtypes with between two and nine example sources for each subtype. Due to the breadth of this topic, it cannot possibly be covered completely in this report, but some brief mitigation strategies are given as examples.

15.2 Software Common Cause

Software common cause can be addressed through rigorous verification and test methods which seek to discover and remove potential errors before the system is placed in operation. Software dissimilarity is another technique that has been advocated to address software CCF. The use of dissimilarity at the code level has been discredited (Knight-Leveson)[7]. However, at the requirements level where a completely different function or approach can be specified, this is deemed by most experts generally to be a good idea. Another method that can provide some mitigation of software common cause errors is the use of restarts and retries though the latter can sometimes be complicated by the need for some rollback mechanism. The hope is that after the restart, the software will not traverse through exactly the same set of inputs that triggered the software bug, because the system will be processing new sensor values. Software errors can also result from compiler design errors. However, these errors are less common due to the heritage of the compilers and a large user base which tends to weed out the bugs in the field. Probably the most promising of all approaches is formal

[7] It has been demonstrated that even independently-developed software versions can fail on the same input. In fact the probability that this occurs was shown to be much greater than one would expect (i.e. the independence assumption is false) [Leveson]. It is believed that low-level design diversity (e.g. often called software fault tolerance) is more vulnerable to this problem than high-level design diversity. . Because design diversity does not provide a strong guarantee against common mode failure, a combination of all the approaches is highly recommended. In the most critical systems and subsystems the use of formal verification is prudent, even though it is may impose higher early life cycle costs.

Both N-version programming techniques and recovery block schemes have been proposed as code-level mechanisms. For an excellent tutorial on software fault tolerance see [Pomales00]. Recovery blocks are based around the idea of an acceptance test. If the output of a module fails the acceptance test, a backup software routine is used to produce the output. N-version programming relies on multiple versions of a program created by different programming teams. All versions are created from the same specification so that the outputs can be voted.

methods. Formal methods are able to mathematically establish the absence of hazardous behavior over all possible inputs using formal proof (using theorem provers) or exhaustive search (using model checkers).

15.3 Design Errors in Hardware

Design errors in hardware (i.e processors, ASICs, FPGAs) are also possible. These types of errors generally have lower failure rates than software errors because of the lower complexity and more rigorous verification culture present in the hardware industry. Adding hardware dissimilarity is a method that mitigates this class of error, but this can easily be “straining at a gnat”. The author knows of no loss of a safety-critical system due to a design error in the hardware. Whereas the loss of safety-critical systems due to errors in the application software are quite numerous. Is should also be noted that adding dissimilarity to a redundant computing system can actually increase the probability that it fails due to CCF. Whenever redundancy is added to a system, additional logic (in hardware/software) must be created to manage that redundancy. Mistakes in that logic can be catastrophic –- creating a new single point of failure. So dissimilarity is no panacea. It should be used with the utmost of care and probably only where there is a commitment to formally prove the managing logic.

15.4 Radiation Induced Common Cause Failure

The impact of radiation on spacecraft electronics can be dramatic. For example, in September 2005 a solar flare upset the ROSETTA spacecraft electronics leaving the primary star tracker function in INIT mode and the secondary Star Tracker in standby mode.

Radiation can cause single-event upsets (SEUs) such as memory bit-flips or transients on clock or data lines. SEUs do not cause permanent damage but can cause data and/or code corruption that can lead to functional upsets. Permanent damage can result from single event latch-up (SEL) event. A SEL event can activate a parasitic low-impedance path between the power supply rails of an electronic component (which acts as a short circuit) and cause displacement effects (i.e. the cumulative effect of displacing atoms out of their lattice, which can destroy the device). A third radiation effect is accumulated total incident dose (TID). As TID accumulates in electronic components (over months or years) transistor parameters can shift and eventually lead to device failure . Pritchard et.al. [Pritchard02] list the following NASA missions which were adversely impacted by radiation effects including (1) TOPEX-Poseidon (permanent failure of optocouplers), (2) Cassini (solid-state recorder errors), (3) Deep Space 1 (Latchup in stellar reference unit, upset in solar panel control electronics), (4) QuikScat (GPS receiver failure, 1553 bus lockups), (5) Mars Odyssey (entered safe mode due to processor reset caused by latch upset in DRAM) and (6) GRACE (Resets, reboots, double-bit errors in MMU-A, some GPS errors).

Although redundancy is helpful to some extent, a serious radiation event can impact a majority of the FCRs and defeat the architectural-level fault-tolerance. Therefore classic architectural-level fault-tolerance must be augmented with other strategies for spacecraft. The primary methods are

- Use of radiation hardened parts and materials. (This reduces TID, can effectively eliminate SEL, and improves SEU susceptibility)
- Latch-up detection and device reset
- Low-level Error Detection and Correction (EDAC) mechanisms such as memory ECC (See Section 11.2 “Transient Fault Recovery”).
- Use of shielding (This is primarily used to reduce TID effects, but can have some impact on low energy particles that can result in SEUs).
- Low-level rewriting of the control state in hardware (e.g. rewriting of mode registers in SDRAMS.)
- On-chip redundancy. (TMR is common in space-qualified FPGAs)
- Device-level redundancy (e.g, multiple processors configured as a TMR within a single board computer.)
- Use of self-stabilization algorithms. (A distributed algorithm is self-stabilizing if, starting from an arbitrary state, it is guaranteed to converge to an operational state within a bounded amount of time.)

15.5 Other Common Cause Faults

Common cause manufacturing defects can be partially mitigated through redundancy and fault tolerant techniques depending on their manifestation. Screening and inspection can also identify manufacturing defects before system deployment. Part defects can be mitigated through dissimilarity. Sometimes manufacturing defect manifestation is due to the environment (physical stress, temperature, radiation etc.). Therefore testing in appropriate environments helps to identify some defects.

Common cause power related problems are worthy of special mention. Because there are often significant constraints on mass and cable length, fault containment regions share the redundant power lanes. Therefore if a power fault occurs, multiple FCRs (from a defect or data transfer perspective) can be affected. Also NASA has a history of power related faults, making it appropriate for extra scrutiny. Parts screening, derating, testing and careful fault containment analysis and design help mitigate this class of fault.

A final example that can lead to common cause failure is due to the vibration and shock loads of a launch or reentry. The harshness of these environments needs to be carefully analyzed and designed for with ample margin for uncertainty. Fault containment through physical separation and isolation are mitigation strategies. Also, environmental testing can identify problems that may not be identified by system modeling.

15.6 Functional-level Dissimilar Backup System

Sometimes safety-critical systems are designed with a completely independent backup system. The Space Shuttle for example has a primary system that is a quadraplex and a fifth dissimilar computer that serves as a backup. This backup system has never been used, but has provided a level of comfort to the astronauts and the operators. Typically a backup system is an extremely simple system that provides only basic functionality. . It is usually programmed with different software from the primary system and provides just enough functionality to get the crew home safely. Simplicity in the hardware (i.e. reduced part count) reduces the fault arrive rate and hence makes it more reliable. Simple software reduces the software defect rate and potentially enables formal verification.

The process that governs the switchover to the backup system is of critical importance. You cannot have the backup unilaterally takeover, because it may do so improperly when it has failed. And the failure rate of the simplex backup system is much higher than the primary fault-tolerant system. Of course it is reasonable to allow the primary system to give control to the backup, but this cannot be relied upon for all situations. Therefore inevitably, the switchover process must involve the human operator. But it should be remembered that the effectiveness of this strategy depends upon the ability of the astronauts to recognize the need for and accomplish the switchover to the backup in adequate time.

Adding hardware dissimilarity is one method that mitigates common cause failure. But as noted previously, additional redundancy should always be added judiciously and complexity must be carefully managed. For example, adding a dissimilar backup to a primary redundant computing system can help mitigate common cause failure, but system designers must carefully understand how control is passed from primary to backup. Unintended results could occur if a fault in the backup system enables it to take control from a perfectly good primary system. Design error in the system level redundancy management logic is one class of common cause failure that can only be mitigated through rigorous design assurance techniques.

Before a backup system is given control, the backup has to be initialized with all of the relevant program state. Sometimes the backup system is run as a hot spare and performs all of the same calculations as the primary system, only its outputs are not sent to the system actuators. If the backup is “cold” it has to be loaded with appropriate code and data prior to being given control.

The cost of a backup system can be considerable given that different software has to be written for it. Also the backup system must be connected to the input/output communications systems or have its own set of sensors and actuators.

15.7 Common Cause Failure and Integrated Modular Avionics

The impact of IMA on the problem of common cause failure is a complicated issue:

- The use of IMA creates a higher potential for common mode failure due to the extensive use of common hardware modules and a common operating system, but
- The ready availability of partitions provides a tool to the application designer to isolate the impact of insidious software bugs to smaller regions. A divide-by-zero error may temporarily halt one function but will not impact all of the others.

It is the author's opinion that IMA can provide a net gain in the area of common mode failure.   The following factors mitigate the unique risks associated with IMA:

- Historically processor design errors have not resulted in catastrophic failure.
- Several IMA operating systems are being designed in accordance with the FAA's DO-178B software standard.
- More rigorous verification techniques such as formal methods can be deployed because the cost is amortized over many different applications and vehicles.
- The IMA operating systems will become increasingly reliable as they are deployed in aerospace applications and millions of operational hours flush out the bugs with very low fault arrival rates.

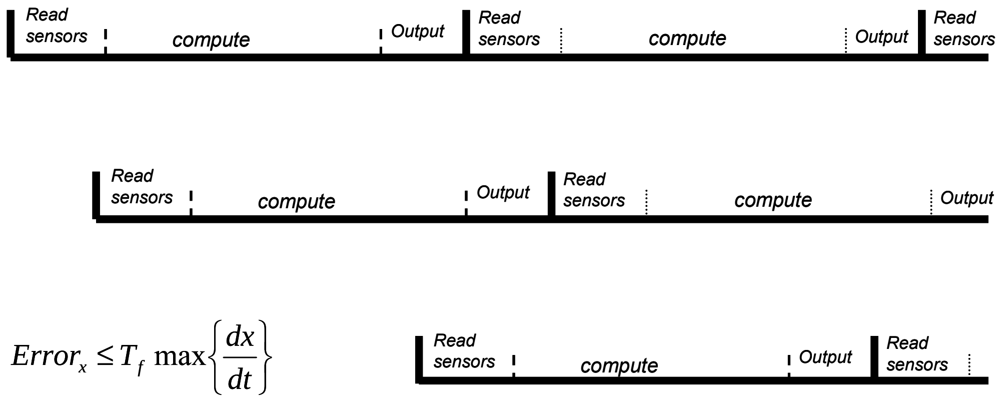
16 Re-Usable Fault Tolerance and System Layering

Many fault-tolerant computer systems have been built with voting strategies that utilize information about the applications. Although at first this may seem like a good thing to do, i.e. we might as well use the maximum information available to detect faults, it turns out that this is really very counter-productive. The problem is that one cannot verify and validate the fault-tolerant aspects of the system without the applications.   There is no divide-and-conquer approach that can simplify the verification and validation.   The aviation industry has frequently used application-level fault-tolerance in the design of their flight control systems. Interestingly they have largely avoided this approach in the design of fault-tolerant flight management systems. Here architectures based upon self-checking pairs are frequently employed.

16.1 Asynchronous Flight Control Systems

Asynchronous digital fault-tolerant flight control systems arose in the 1980s. They exploit the fact that the control laws that run on them are periodic in nature and that they sample inputs and produce actuator outputs. Therefore, if the channels are allowed to drift apart (i.e. no clock synchronization), then they will never really drift more than the sample period apart (e.g. 40 ms), because once they get a full frame apart the channels are once again sampling the sensors at the same time.

Voting in an asynchronous architecture is built around the idea that the rate of change of output value is bounded and that the sensor data can be separated in time by no more the period of the sample rate. This is illustrated below



$$Error_x \le T_f \max\left\{\frac{dx}{dt}\right\}$$

Therefore the error between the outputs is:

T_f max (df/dt)

where $T_f =$ sensor sample period, and max (df/dt) = the maximum rate of change of the output function. If the control laws are stable then the output differences will be bounded if the input differences are bounded. Using these bounds, thresholds can be set at the voters. The mid-value from the channels is used to drive the actuators and if the difference between an output and the mid-value exceeds the threshold then the channel is declared as failed.

But eventually the designers have to deal with the fact that the control laws have state variables associated with them (e.g. integrator variables). So although the input variables are re-aligned once a channel drifts a full period apart, the integrator variables are one iteration step apart! So inevitably in these architectures the designers end up using cross-channel strapping and data synchronization techniques. So this is typically handled by performing data synchronization at the application level. For example Y.C. Yeh of the Boeing Commercial Aircraft Company describes how this is done for the Boeing 777:

> *"The potential for functional asymmetry, which can lead to disagreement among digital computing systems, is dealt with via* ***frame and data synchronization*** *and median value selection of the PFC's output commands" []*

So although the original design was built around the concept of no clock synchronization between the channels, they end up synchronizing anyway. But, instead of solving this problem at a lower level of the system in an application-independent way, the problem is solved at the application level while dealing with a lot of other issues. In other words, there is no separation of concerns. It is possible to defer the issue of synchronizing the channels, but eventually this issue must be at solved at some higher level of the system.

The whole strategy for diagnosing failed channels is based upon bounded rates of change of the output values. But what happens when there is a discrete change or a mode change? Well this could lead to an erroneous diagnosis of a channel failure. So another ad-hoc solution is patched onto the architecture--the channel outputs are passed through an output filter which ramps up and ramps down the actuator outputs in a way that bounds the rate of change. These ramps also serve to smooth out any rapid change of an output to an actuator.

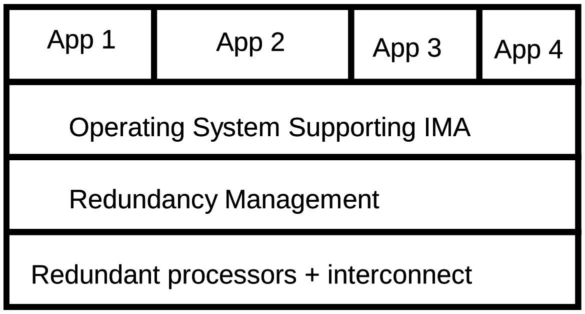
So the designers are essentially dealing with the lack of clock synchronization using a suite of ad-hoc patches. But there is no separation of concerns, no divide and conquer and thus one ends up with an extremely complex architecture that is difficult to validate and test. Dale Mackell wrote about this problem after Dryden had flight tested the asynchronous F-16 DFCS, "The fault-tolerant design should also be transparent to the control law functions. The control laws should not have to be tailored to the system's redundancy level." [Mackall88]

There are some additional concerns associated with mid-value select algorithms. Because the channel outputs are not identical even in fault-free conditions, exact-match voting cannot be used. A mid value select algorithm must be used, but mid value select algorithms cannot be used for diagnosis because they do not decide which of the other two channels are faulty. Therefore, thresholds based upon dynamics must also be employed for fault diagnosis. But this is not as good a fault detector as an exact match voter. A permanently faulty processor may remain undiagnosed for a long time, e.g. if it "flat lines" between two good channels. So fault latency increases and this impacts the reliability analysis. Also where to set the thresholds is fundamentally a trial-and-error process. These thresholds change as the control laws change. It is not unusual for the control laws to be modified at integration testing when the vehicle dynamics are better understood. System designers have to wait until they have a full-scale simulation (e.g. iron-bird) to debug the fault-tolerance of the system. But this means that the basic fault-tolerance design is being modified at integration time as well. What should be an early-life cycle, low level activity is deferred until late in the life-cycle when repair of errors are notoriously expensive to repair.

16.2 Synchronous Fault-Tolerant Systems

A synchronous fault-tolerant system is built on the foundation of a fault-tolerant clock synchronization algorithm which can keep all of the good processor's clocks within a small skew of each other, often denoted as ε. This foundation enables all of the good processors or processes to be exact replicas of each other. A simple timing protocol suffices. Because all non-faulty processors or processes are exact replicas, exact match voting is used to both mask faults and detect failed components. Interactive consistency algorithms (see Section 6 "Interactive Consistency") are used to distribute single source data (such as sensor data) to the replicas in a way that guarantees that all good processors receive the exact same value even in the presence of faults. As long as 3*f*+1 FCRs (processors and/or interstages) are non-faulty, *f* faults can be tolerated. Finally a distributed diagnosis algorithm is used to identify faulty components.

Because the management of redundancy can be done independently of the applications (using exact-match voting), a layered approach to the system design is possible as illustrated below:

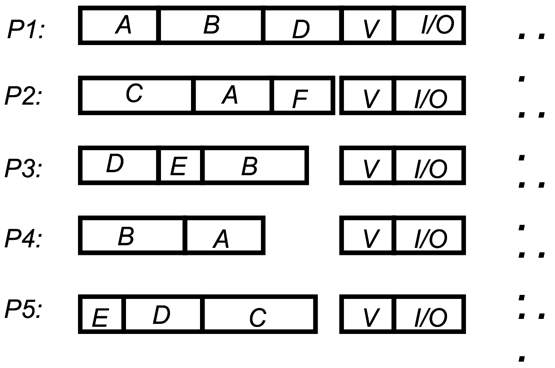

App 1
App 2
App 3
App 4
Operating System Supporting IMA
Redundancy Management
Redundant processors + interconnect

Many cost-saving and safety benefits accrue from this approach. First, the application software can be designed by a different vendor than the one building the fault-tolerant computing platform. This prevents the government from getting locked into a single large vendor. Second, the software can be developed as if it were running on a single ultra-reliable operating system. It can be tested and verified independently from the fault-tolerant system. Third, the fault-tolerant computing platform can be reused over many different applications. The fault-tolerant computing platform can be highly configurable and scalable supporting different safety and reliability goals. Fourth, the redundancy management algorithms can be designed in a processor-independent manner which enables the use of COTS processors that won't lock you into an antiquated hardware technology. The following characteristics of the redundancy management layer are highly desirable:

- Fault-masking, fault detection, and reconfiguration are independent of the applications software

- The redundancy management should be handled in a processor-independent way (either via software or via small custom hardware that interfaces to COTS processors)
- The redundancy management should be highly configurable allowing some processes to run in triplex, some as simplex, some as dual-dual etc.
- The redundancy management should support low power modes, allowing systems to be turned on and off without interruption of critical applications and processes.
- The redundancy management should provide a standard interface so that multiple IMA operating systems could be supported.

Sometimes it is argued that a synchronous system is more vulnerable to electromagnetic interference (EMI) or single-event upset (SEU) than an asynchronous system. But a strategy that can be helpful is to schedule the redundant tasks so that they run at different times:



16.3 Maintaining Independence between the applications and the fault-tolerant mechanisms

Many fault-tolerant systems that are deployed today are not reusable because the fault-tolerance mechanisms used in these systems are intimately connected to the specific applications that run on them.   If the applications are changed, then the fault-tolerance mechanisms must be changed as well.

To enable reuse, the fault-tolerant system’s methods for detecting and reconfiguring in the presence of physical faults should **not** be dependent upon the application software. In particular, the system should not be design using threshold voting. Rationale for this is:

- Threshold voting based on control software characteristics defers validation of the fundamental fault tolerance until integration testing (late life cycle).
- The thresholds have to be changed as the controls are changed and so we have a serious maintenance and configuration management problem.

- Threshold voting based on application software characteristics leads to false alarms because of uncertainty in the cause of a threshold being exceeded (e.g. did vehicle dynamics change or was it a processor fault).
- Exact match voting detects errors without regard to the nature of the applications.
- Exact match voting can be validated early in the life cycle.
- Exact match voting can be used in conjunction with BIT to provide 100% fault masking and high probability of a timely reconfiguration.
- You do not have to artificially ramp-up/ramp-down outputs to make sure that voting thresholds won't be exceeded during non-linear actions such as mode switching.
- You do not have to introduce cross-channel synchronization of integrator values because the interactive consistency algorithms used will maintain a globally consistent state on all good processors.
- Diagnosis of faulty processors, memories, and I/O resources is not confounded by uncertainty over whether the fault is due to computational resource failure or due to failure in an external vehicle system or actuator that is affecting the thresholds.

The fault-tolerant system should be developed in a layered systems approach with well-defined APIs between the layers. It should reside in a layer below the applications and create an interface that provides an abstraction that **hides the redundancy and voting from the higher layers** to the maximum extent possible.

16.4 Technology Obsolescence and Refresh

The fault-tolerance used in most avionics systems today is not easily updated by new technology. Because much of the timing and voting in the system is handled by the processors themselves and depends upon specific aspects of these processors, they cannot be easily updated with more capable processors. The fault-tolerant system must be designed with this goal in mind and carefully configured to avoid these dependencies. The SPIDER is a modern architecture that has been designed with this as a primary goal [Geser02]. See http://shemesh.larc.nasa.gov/fm/spider/ for details about this fault-tolerant system.

17 Reliability Analysis

Since fault tolerance seeks to increase the reliability of a system through redundancy it is important to be able to compute the reliability of a fault-tolerant system as a function of measurable parameters such as processor failure rates and system recovery rates.

17.1 Markov Models

Markov analysis is a powerful mathematical approach that accomplishes this goal. The Markov model shown in Figure 1 describes the behavior of a simple degradable quadraplex system.

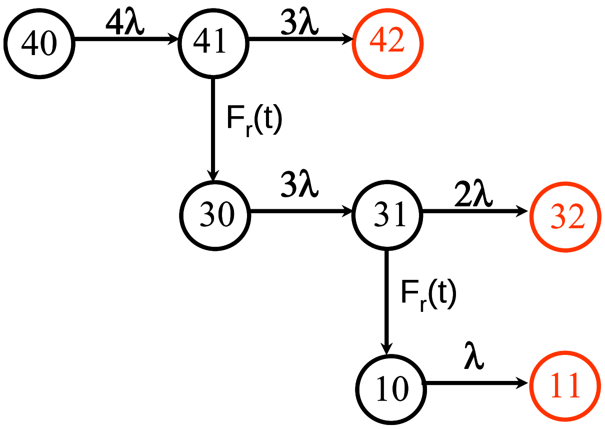


Figure 1. Reliability model for a degradable quadraplex system.

The states of the system are labeled with two digits. The first digit denotes the number of processors that are currently being voted. The second digit denotes the number of faulty processors in the system. The system starts in state 40, where all four active processors participate in the vote and none of them are faulty. Anyone of these processors can fail, which takes the system to state 41 at rate 4λ, where λ represents the single processor failure rate. Because the processors are identical, the failure of each processor is not represented with a separate transition. For example, at state (41), the system has one failed processor but there is no delineation as to which processor has failed and so the total rate of reaching this state is 4λ. Here, the system analyzes the errors from the voter and diagnoses the problem. The transition from state (41) to state (30) represents the removal (reconfiguration) of the faulty processor. The reconfiguration transitions are labeled with a distribution function ($F_r(t)$) rather than a rate. The reason for this labeling is that experimental measurement of the reconfiguration process has revealed that the distribution of recovery time is usually not exponential. Consequently, the transition is not described by a constant rate. This label indicates that the probability that the transition time from state (41) to state (30) will be less than t is F_r (t). The presence of a non-exponential transition generalizes the mathematical model to the class of semi-Markov models. At state (30), the system is operational with three good processors. The recovery transition from state (41) to state (30) occurs as long as a second processor does not fail before the diagnosis is complete. Otherwise, the voter could not distinguish the good results from the bad. Thus, a second transition exists from state (41) to state (42), which represents the situation where two out of the four processors have failed and are still participating in the vote. The rate of this transition is 3 λ, because any of the remaining three processors could fail. State (42) is a death state (an absorbing state) that represents failure of the system due to coincident faults. It is labeled in red to indicate system failure. Of course, this is a conservative assumption. Although two out of the four processors have failed, the failed processors may not produce errors at the same time nor in the same place in memory. In this case, the voting mechanism may effectively mask both faults and the reliability of the system would be better than the model predicts.

At state (30), the system is operational with three good processors and no faulty processors in the active configuration. Either one of these processors may fail and take the system to state (31). At state (31), once again, a race occurs between the reconfiguration process that ends in state (10) and the failure of a second processor that ends in state (32). The recovery distribution from state (31) could easily be different from the recovery distribution from state (41) to state (30). However, for simplicity it is assumed to be the same. State (32) is thus another death state and state (10) is the operational state where one good processor remains. In this case the system was not designed to operate as a dual so the system degrades to a simplex here. The transition from state (10) to state (11) represents the failure of the last processor. At state (11) no good processors remain, and the probability of reaching this death state is often referred to as failure by exhaustion of parts.

17.2 Solution of a Markov Model

The mathematical solution of a Markov model involves the solution of a system of linear differential equations often referred to as the Chapman Kolmogorov equations. Because the analytic solution becomes unwieldy as the number of states is increased, numerical approaches for solving Markov models has been pursued. Several programs have been developed at NASA Langley to solve Markov and semi-Markov models (a generalization) numerically: SURE, PAWS, STEM and ASSIST and their Windows versions WinSURE, WinSTEM, and WinASSIST[8]. These programs are approved for public release and available for free. See http://shemesh.larc.nasa.gov/people/rwb/rel.html for information about how to obtain them.

The SURE program is a program for solving semi-Markov models useful in describing the fault-handling behavior of fault-tolerant computer systems [Butler88, Butler92, Butler95]. The only modeling restriction imposed by the program is that the nonexponential recovery transitions must be fast in comparison to the mission time—a desirable attribute of all fault-tolerant

[8] The SURE and WinSURE programs solve semi-Markov models. A semi-Markov model is more general than a pure Markov model in that it allows non-exponential transitions. The SURE program requires the mean and standard deviation for all of the non-exponential distributions. The PAWS and STEM programs accept the same input language as SURE, but they assume that all transitions are exponentially distributed. The exponential rate is derived from the specified mean.

systems. The PAWS and STEM programs accept the model input in exactly the same format as the SURE program. They assume that all of the recovery transitions are exponential because they are pure Markov solvers. The ASSIST program is a tool that generates large models for SURE, STEM, or PAWS [Johnson95].

The model shown in figure 1 can be described in the SURE input language as follows:

```
LAMBDA = 3E-6;          (* processor failure rate per hour *)
REC_TIME = 1/3600;      (* 1 second recovery time *)

40,41 = 4*LAMBDA;
41,42 = 3*LAMBDA;
41,30 = <REC_TIME, REC_TIME>;
30,31 = 3*LAMBDA;
31,32 = 2*LAMBDA;
31,10 = <REC_TIME, REC_TIME>;
10,11 = LAMBDA;
TIME = 1;                (* Mission Time of 1 hour *)
```

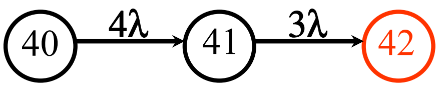
The SURE program produces the following output:

```
                        LOWERBOUND        UPPERBOUND
  --------------     --------------    --------------
                     2.865386E-0014    3.005409E-0014
```

The upper and lower bounds are due to numerical inaccuracy.

17.3 The Impact of Long Mission Times

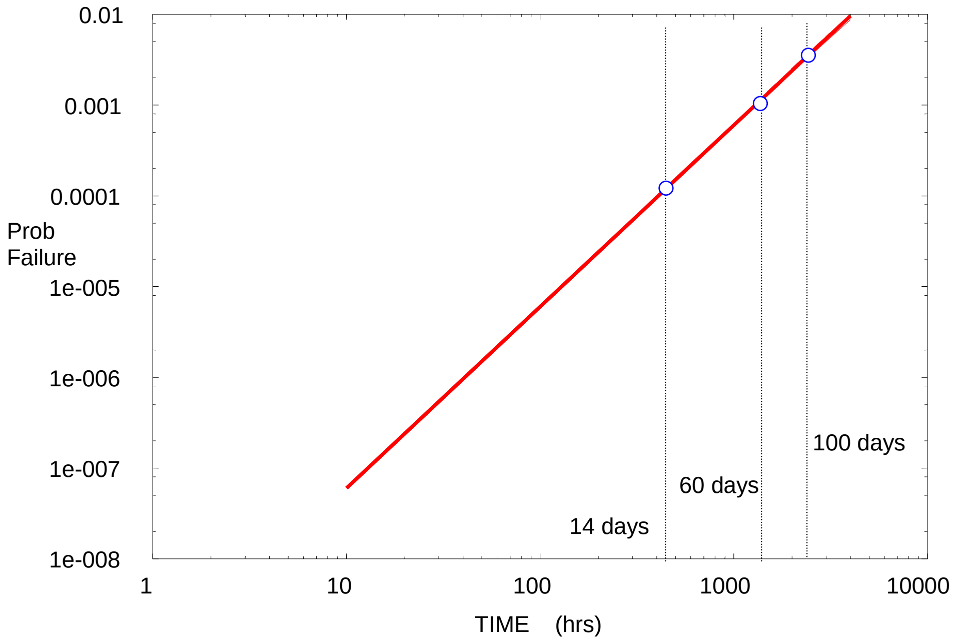
Redundancy loses its effectiveness as the mission time increases. As the mission time approaches the mean time between failure (MTBF), there is an increasing probability that multiple processors will have failed before the end of the mission. For example suppose you have a non-reconfigurable quadraplex. This can be modeled as follows:



The SURE input is simply

```
LAMBDA = 1E-5;              (* processor failure rate *)
TIME = 10 TO+ 4000;         (* mission time *)
1,2 = 4*LAMBDA;
2,3 = 3*LAMBDA;
```

The following logarithmic plot shows the dramatic impact of a large mission time.



The WinSTEM output is

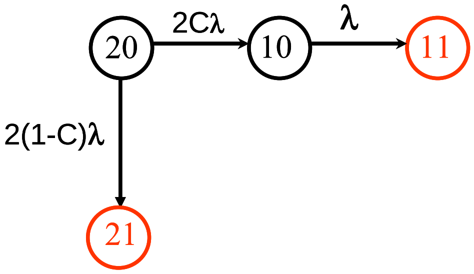
```
    TIME              PROBABILITY          ACCURACY
-----------      -------------------     --------
1.000E+0001      5.99860018498E-0008     14 DIGITS
1.000E+0002      5.98601848251E-0006     14 DIGITS
1.000E+0003      5.86183262937E-0004     13 DIGITS
```

From this graph it is clear why the fault tolerance used in deep space missions is very different from that used in commercial aircraft. In long mission scenarios, the designers focus on reducing the processor failure rate λ and sometimes use cold spares.

17.4 Beware of the Ambiguous Term "coverage"

If you hang around reliability people for a while, you will inevitably encounter the term "coverage". Unfortunately the term is terribly ambiguous. It is used in many different ways in different contexts. In this section, four of the most common uses will be explained. It is the author's opinion that this term should be used with caution or totally avoided, because it is easily mis-interpreted.

First fault coverage: Some systems are not fully capable of surviving all first faults. The percentage of first faults that they can recover from is often called *coverage*. For example, suppose you have a dual system which runs a built-in test diagnostic with coverage C. The following model describes this system:



The system starts in state 20 with two good processors. State 21 represents the system after the arrival of a fault which the system cannot detect and hence mask so it is a death state. State 10 represents the system after the arrival of a fault that is covered. Here the system successfully removes the faulty processor and continues operation with one good processor. The following table shows the dramatic impact of first fault coverage with a fixed $\lambda = 1.0 \times 10^{-5}$/hour:

C	P_f
0.9999	2.1×10^{-9}
0.999	2.0×10^{-8}
0.99	2.0×10^{-7}
0.9	2.0×10^{-6}
0	2.0×10^{-5}

Detection coverage: Some systems are fully capable of *masking* all first faults but can only *detect* a fraction of the first faults. Suppose we have an asynchronous quadraplex system that relies on threshold voting for detection. Hence there can be some faults that remain latent because they do not propagate errors that violate the thresholds. Suppose C represents the fraction of faults that are detectable by the threshold voters and the built-in test (BIT). This means that C percent of all faults will be detected and reconfigured out. The following Markov model describes this quadraplex:

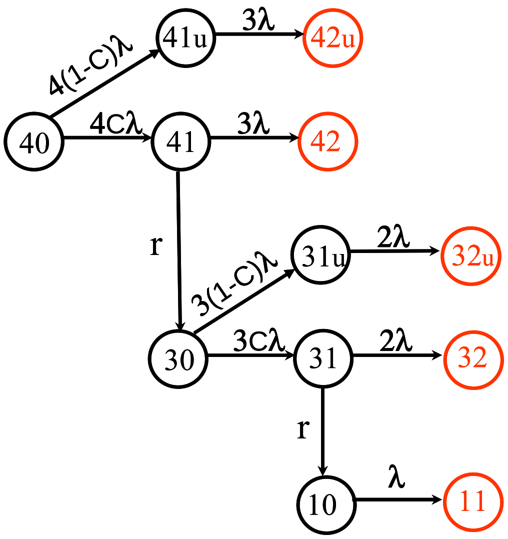
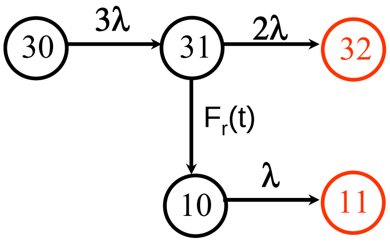


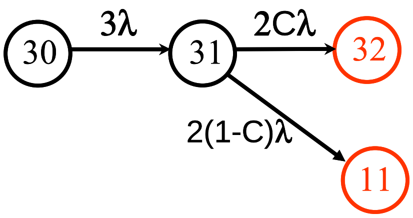
Figure 2. Model of a degradable quadraplex voting system with C% of detectable faults.

Notice that here the "*uncovered*" faults do not lead to system failure because they are outvoted (i.e. masked), but they do take the system to a state (i.e. 41u) where no reconfiguration transition is found.

Second Fault coverage: Because some analysis tools are combinatorial in nature, they cannot solve a Markov model directly. In these tools, the Markov reconfiguration process is modeled as a coverage. For example, the following triplex system model:



is replaced by



where C represents the effectiveness of the reconfiguration process. The advantage of course is that this model can now be solved combinatorially without the need to solve differential equations. But unfortunately the C is not directly measurable. It is a simply an artifact of the model reduction and combines many concepts into one number. It is the author’s opinion that this type of coverage should be avoided.

BIT coverage: the fraction of the faults that can be detected by a Built-In-Test technique. (See Section 8.3 “Detection Using Built-in-Test (BIT)”).

18 The Synergism Between Formal Verification and Reliability Analysis

The question is often raised how do you prove that the reliability of the system is 10^{-9}? The answer is that you decompose the problem into two components, one of which is answered by formal methods and the other which is answered by a numerical reliability analysis:

Using formal methods one seeks to prove formulas of the form

Enough Good Hardware → SAFETY-PROPERTIES

where the → indicates logical implication. The symbol “SAFETY-PROPERTIES” represents all the properties that must be true of the system if it is to avoid reaching some hazardous system state. Using reliability analysis one calculates the probability

Prob [Enough Good Hardware]

Together these provide a basis for assuring the safety of the system.

Formal methods also offers an approach to overcoming a serious dilemma for the reliability analyst, namely, how can I assure myself that the reliability model itself is a valid representation of the implemented system? The key assumptions of the Markov model can be formally verified. For example, a formal proof of the system’s fault tolerant operating system can be used to establish the absence of any direct transition from the fault-free state to a death state.

19 Function Migration

Though a fault-tolerant system is inherently a distributed processor system, systems designers need not identically schedule all of the applications on the available processors. Spare processing power can be allocated as needed. In fact it is possible to move one function (including all replicates) from one set of processors to another. This is often referred to as *function migration* or *dynamic task allocation*. Function migration requires that a flexible communication scheme be employed that provides connections between all of the sensors and actuators and the processors. The application program code must be available to all of the processors in the system. This can be accomplished by providing access to a mass memory where all of the software codes are stored or by making copies of the codes on all of the local stores. Often fault-tolerant systems rely on time-division multiplex buses that are driven by static schedule tables. In these systems, function migration requires that there be some mechanism for updating these tables in a fault-tolerant and safe manner.

Sometimes the distinction is made between

- Asymmetric multiprocessing: where specific processors can execute only certain task types, and
- Symmetric multiprocessing: where any processor can execute any task.

Symmetric multiprocessing enables each processor to be utilized to the fullest. It is usually achieved through identical processors and full interconnection and a flexible operating system interface.

20 Vehicle Health Management

It is useful to distinguish faults that occur in the computing resources themselves from faults that occur in the subsystems that are external to the computing resources. This primer has concentrated on the mechanisms that handle failures in the computing resources. The process of detecting, isolating and recovering from faults and failures in the external subsystems is referred to as *Vehicle Health Management (VHM)*.

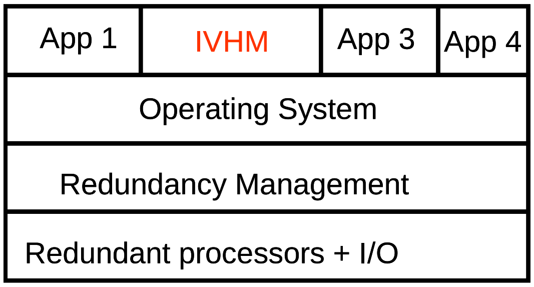
20.1 Basic Concepts

This is a huge topic in itself but a few observations about this topic will be provided here:

1. The mechanisms that are used for *FDIR* (fault detection, isolation, and recovery) are very different from those used in the computing resources and are often based upon concepts from artificial intelligence.
2. The VHM system is usually implemented as a software application that runs on the fault-tolerant computing system.
3. The VHM system seeks to diagnose which external components have failed on the basis of observables.
4. The scope of a VHM system can be huge including sensors, actuators, power systems, displays, thermal systems, landing gear, hydraulics, etc.
5. The use of a single, unified approach to diagnose many different kinds of subsystem failures is usually referred to as *Integrated Vehicle Health Management (IVHM).*

In some systems, the fault-tolerance mechanisms of the computing platform are included in the IVHM functionality. It is the author's opinion that this is generally a bad idea. Since the IVHM system is a software application that inevitably executes on the computing resources, a layered approach is desirable and the

IVHM system should remain independent of lower-level redundancy management:



IVHM systems fall roughly into two categories:

- Rule based
- Model based

In the more traditional *rule-based* approach, rules which associate symptoms with underlying faults are used to diagnose the system. The *model-based* approach uses a model of the subsystem components which are given that same inputs as the system. The outputs of the model are compared with the actual system outputs in real time. If the output of the real system component differs significantly from its model, a failure of that component is indicated. See [Davis88] for an excellent introduction to this topic.

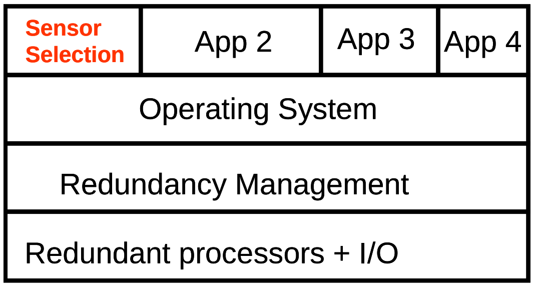
20.2 Failure Modes and Effects Analysis (FMEA)

A number of tools and techniques have been developed to aid the system designer in the identification of hazards in safety-critical systems. Some examples are (1) Failure Modes and Effects Analysis (FMEA), (2) Hazard and Operability Studies (HAZOP), and Deviation Analysis. While performing an FMEA, the analyst creates a list of component failure modes and tries to deduce the effects of those failure modes on the system. Then an assessment is made about the severity, the likelihood of these failures and the ability of the system to detect them. Sometimes these factors are rolled up into a single risk priority number that is assigned to each identified failure mode. This enables the designer to focus his attention on the most critical failure modes. While this type of analysis can be applied to the low-level design of the computing platform components, it is most useful when applied to the vast array of components of the subsystems external to computing resources. This type of analysis aids the designer of an IVHM system or the designer of application software which performs some local diagnosis and recovery from subsystem component failures. This analysis aids the designer in identifying the critical component failures and helps him develop mechanisms to handle these failures. For more information about FMEA, the reader is referred to [Dailey04].

20.3 Sensor Fault Tolerance

Fault-tolerant control systems periodically sample inputs from the environment and produce outputs which are sent to the actuators. Although it is possible for each application task to directly sample sensors and seek to mask sensor failures appropriately, many fault-tolerant systems separate this functionality into a separate application task. In aerospace application this function has often been handled in a separate computer called the *flight data computer*. In an integrated modular avionics system, this function is typically allocated to a separate partition. In either case, for every measurement needed, a single value must be extracted from a set of redundant sensor values, which can then be made available to the application software for processing. Sensor failures must be detected and factored into this selection process. Sensor failure can be detected with local algorithms and/or by a separate IVHM system.

Due to the criticality of the sensor selection software it must also run as a redundant task on the fault-tolerant computing platform:



Each sampled sensor value is initially a non-replicated data item, so it must go through an interactive consistency algorithm before it is processed by the sensor selection logic. This insures that every redundant task executing the sensor selection logic is operating on the same set of values and that faults in the I/O system do not corrupt this selection process. In a properly layered system, this is accomplished by the lower redundancy management layer. In this approach, the designer of the selection logic can focus on the algorithm for selecting a value from multiple distinct sensors.

For discrete inputs, the sensor selection process involves selection of a value from a set of redundant 1 or 0 values. The input selection process for sensors with a range of values is significantly more complicated. Usually some form of mid-value selection is used on all inputs that did not come from sensors detected as failed is employed. In systems that do not have a separate IVHM capability, thresholds can be used to detect sensor failure locally. If a value of a sensor deviates significantly from the mid-value, it is declared as failed.

Sometimes there are dynamic relationships between different kinds of sensors, e.g. speed and acceleration. These dynamic relationships can be leveraged to synthesize more accurate values for the required measurements. This can be especially useful when critical sensors have failed. This approach is called *analytic redundancy.* These techniques tend to be very application specific because they depend upon the dynamic characteristics of a particular vehicle.

21 Concluding Remarks

The design of a fault-tolerant computing system is an extremely challenging endeavor. The algorithms and techniques that are at the center of fault-tolerance are among the most subtle and difficult to prove in Computer Science. Fortunately these algorithms have been vigorously studied and analyzed by the academic world. Many of these have been formally verified and mechanically checked using theorem prover technology [Miner04]. It would be foolish to design and implement a fault-tolerant computer today without taking advantage of this storehouse of results.

The study of fault tolerance cannot be divorced from reliability analysis. A basic understanding of Markov modeling and analysis is essential to understanding the tradeoffs that must be made in the design of a fault-tolerant computer. Fortunately this is not a difficult thing to obtain [Butler95]. The solution of these models is straight-forward using freely available programs such as SURE or STEM [Butler88].

The selection the appropriate fault-tolerance for a system is a complex process that depends upon the specified reliability, mission duration, maintenance processes, cost, expected lifetime and many other factors. In this short primer the key concepts and techniques available to the fault-tolerant system designer have been introduced.

22 Glossary

Application Programming Interface (API): A language processed by an operating system (or lower layer in the system), which is used to provide services to the applications.

Application software: the software that implements the primary functions of the system. This software executes in an environment providing by system software (e.g. the operating system) which is distinct from the application software.

Asynchronous: Systems that do not synchronize the clocks of the redundant processors are called *asynchronous* systems

Built-In-Test (BIT): diagnostics which run automatically and seek to isolate faulty components.

Byzantine fault: A fault with arbitrary failure manifestation including asymmetric ones where different good components get different values.

Clock Synchronization: Clock synchronization is a fundamental issue in fault tolerance which overcomes the problem that the internal clocks of different processors drift apart. Clock synchronization is based upon distributed algorithms that periodically adjust the local clocks in a way that is not vulnerable to a failure in any single clock.

Coincident faults: The presence of two or more faults in a redundant system at the same time.

Common Cause Fault (CCF): A fault that can trigger multiple simultaneous errors in different fault containment regions.

Commercial Off-The-Shelf (COTS): software or hardware that is available for sale to the general public.

Coverage: An ambiguous term that is the fraction of faults “covered”. See section 17.4 for several different definitions.

Design Error: A design error is a difference between the system requirement and the specified design. The failure mechanism is in the human mind. Design errors range from syntax errors in the program code to fundamental mistakes including the use of wrong algorithms, inconsistent interfaces, and software architecture mistakes.

Design for Minimum Risk (DFMR): is a process that allows safety-critical mechanisms to claim adequate fault tolerance through rigorous design, analysis, testing, and inspection practices rather than through true physical redundancy.

Distributed Diagnosis: A key capability in a fault-tolerant system: each component of the system must maintain correct information about which other components in the system have failed. It is important that the view of all working components be the same.

Error: The manifestation of a fault -- an incorrect state of hardware or software due to a defect in a component, physical interference from the environment, or from an incorrect design.

Error detection: The process of detecting that a component has failed by observing a difference between system state and the expected state.

Error recovery: the process of restoring the system state to an error-free state after the occurrence of a fault (usually transient).

Exact-match voting: The process of determining a “voted” value in a system where any value that is not bit-for-bit identical to the majority value is known to be faulty.

Fail stop: A component stops producing outputs when it fails.

Failure: The result of a system deviating from the specified function of that system because of an error.

Failure Modes and Effects Analysis (FMEA): A methodology that helps the system designer to identify and handle hazards in safety-critical systems.

Fault: A defect in the hardware, software or system component that can lead to an incorrect state (i.e. error).

Fault Containment Region (FCR): a subsystem that will operate correctly regardless of any arbitrary fault outside the region.

Fault masking: A method for preventing an error from propagating to a system output and hence insuring that only correct values are propagated.

Formal Methods: Formal Methods refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. The phrase "mathematically rigorous" means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are rigorous deductions in that logic (i.e. each step follows from a rule of inference and hence can be checked by a mechanical process.) The value of formal methods is that they provide a means to symbolically examine the entire state space of a digital design (whether hardware or software) and establish a correctness or safety property that is true for all possible inputs.

Integrated Modular Avionics (IMA): An architectural concept wherein functions that have historically been separated and often of different criticality are hosted on a common computing resource.

Integration testing: A late life-cycle phase of software development where all of the software modules of the system are combined and tested together.

Interactive Consistency: A property of a system which states that the input values to the system are distributed in a manner that guarantees that all redundant tasks get exactly the same value even in the presence of faults.

Intermittent fault: A fault that appears, disappears and then reappears.

Mean Time Between Failure (MTBF): is the average time between failures of a system. When the failure rate of the system is constant, it is the reciprocal of the failure rate.

N fault tolerant: a system that is still operational after N consecutive faults (not simultaneous). For example, a two fault tolerant (2FT) system is a system that is "fail operational, fail operational", i.e. after two sequential faults, the system is still a functioning system.

N-plex: a fault-tolerant computer composed of N voting lanes (i.e. N FCRs). Also referred to as a N-modular redundant (NMR) system.

Permanent fault: a fault that continues to produce errors.

Quadraplex: a fault-tolerant computer composed of 4 voting lanes (i.e. 4 FCRs).

Real Time Operating System (RTOS): is a multitasking operating system intended for applications with strict deadline requirements.

Reconfiguration: the process of removing a faulty component from the system.

Self-Checking Pair: Built out of two identical processing elements. The two outputs are compared and if there is a miscompare, then the output is inhibited making the SCP fail-stop.

Synchronous: Systems that synchronize the clocks of the redundant processors are called *synchronous* systems

Threshold Voting: A threshold is the maximum amount of deviation from the average value that is tolerated before a component is declared to be faulty. Threshold voting refers to the use of thresholds when voting in a fault-tolerant system. It is usually used in conjunction with mid-value select.

Transient fault: a fault that appears for a short time and then disappears

Triplex: a fault-tolerant computer composed of 3 voting lanes (i.e. 3 FCRs). Also referred to as a triple-modular redundant (TMR) system.

Triple Modular Redundancy (TMR): A fault-tolerant architecture that uses three processors and voting to produce the output.

Voting: the process of selecting a final result from the outputs of redundant channels or redundant computations.

23 References

[Avizienis04] Algirdas Avizienis, Fellow, IEEE, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions On Dependable And Secure Computing, Vol. 1, No. 1, January-March 2004

[Butler88] Butler, Ricky W.; and White, Allan L.: SURE Reliability Analysis: Program and Mathematics. NASA Technical Paper 2764, Mar. 1988.

[Butler92] Butler, Ricky W.: The SURE Approach to Reliability Analysis. IEEE Transactions on Reliability, vol. 41, no. 2, June 1992, pp. 210--218.

[Butler95] Ricky W. Butler and Sally C. Johnson, Techniques for Modeling the Reliability of Fault-Tolerant Systems With the Markov State-Space Approach , NASA RP-1348, September 1995, pp. 130.

[Dailey04] Dailey, Kenneth W. The FMEA Pocket Handbook, DW Publishing, 2004.

[Davis88] Davis, Randall and Hamscher, Walter. Model-Based Reasoning: Troubleshooting, *Exploring Artificial Intelligence*, Shrobe, H. E. (ed), Morgan Kaufmann, 1988, Chapter 8.

[DiVito90] Di Vito, Ben L.; Butler, Ricky W.; and Caldwell, James L.: Formal Design and Verification of a Reliable Computing Platform For Real-Time Control (Phase 1 Results). NASA Technical Memorandum 102716, NASA Langley Research Center, Hampton, Virginia, October 1990.

[Driscoll03] Driscoll, Kevin; Hall, Brendan; Sivencronam, Hakan; Zumsteg, Phil: Byzantine Fault Tolerance, from Theory to Reality: Computer Safety, Reliability, and Security, Publisher: Springer-Verlag Heidelberg, ISBN: 3-540-20126-2, Volume 2788 / 2003, October 2003, pp. 235 - 248

[Geser02] Alfons Geser, Paul Miner. A Formal Correctness Proof of the SPIDER Diagnosis Protocol.. Theorem-Proving in Higher-Order Logics (TPHOLs), track B, 2002.

[Harper91] Richard E. Harper and Jaynarayan H. Lala. Fault-tolerant parallel processor, AIAA Journal of Guidance, Control, and Dynamics, 14(3), pp. 554-563, May-June 1991.

[Johnson95] Johnson, Sally C.; and Boerschlein, David P.: ASSIST User Manual. NASA technical memorandum 4592, August 1995.

[Lala86] Jaynarayan H. Lala. A Byzantine resilient fault tolerant computer for nuclear power application. IEEE Fault Tolerant Computing Symposium 16 (FTCS-16), pp. 338-343, Vienna, Austria, July 1986.

[MA2-00-057] http://mmptdpublic.jsc.nasa.gov/mswg/Documents/MA2-00-057.pdf

[Mackall88] Mackall, Dale A. Development and Flight Test Experiences With a Flight-Crucial Digital Control System. Technical Report NASA TP-2857, Research Engineering, NASA Dryden Flight Research Center, 1988.

[Miner04] Paul Miner, Alfons Geser, Lee Pike, and Jeffrey Maddalon: A Unified Fault-Tolerance Protocol. Presented at Formal Modelling and Analysis of Timed Systems - Formal Techniques in Real-Time and Fault Tolerant System (FORMATS-FTRTFT 2004), Grenoble, France, September 22-24, 2004.

[Pomales00] Wilfredo Torres-Pomales. Software Fault Tolerance: A Tutorial, NASA/TM-2000-210616, October 2000.

[Pradhan86] Pradhan, Dhiraj K.   Fault-tolerant computing: theory and techniques. Prentice-Hall, Inc., 1986.

[Pritchard02] Pritchard, Bruce E., Swift, Gary M, Johnston, Allan H. Radiation Effects Predicted, Observed, and Compared for Spacecraft Systems, Radiation Effects Data Workshop, 2002.

[Ramanathan90] Ramanathan, P.; Shin, K. G.; and Butler, Ricky W.: Fault-Tolerant Clock Synchronization in Distributed Systems. IEEE Computer, October 1990, Vol. 23, No. 10, p. 33-42.

[Rushby89] Rushby, John; and von Henke, Friedrich: Formal Verification of a Fault-Tolerant Clock Synchronization Algorithm. NASA Contractor Report 4239, June 1989.

[Rushby01] John Rushby, Bus Architectures for Safety-Critical Embedded Systems, Lecture Notes In Computer Science; Vol. 2211, Proceedings of the First International Workshop on Embedded Software table of contents, 2001, pp 306 – 323.

[Stephans04] Stephans, Richard A: System Safety for the 21st Century,
John Wiley & Sons, Inc., 2004, Print ISBN: 9780471444541, Online ISBN: 9780471662549.

[Thambidurai88] Thambidurai, P., and You-Keun Park, "Interactive Consistency with Multiple Failure Modes", 7th Symposium on Reliable Distributed Systems, Columbus Ohio, Oct 10-121988, pp 93--100.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
01- 02 - 2008

2. REPORT TYPE
Technical Memorandum

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
A Primer on Architectural Level Fault Tolerance

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Butler, Ricky W.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
604746.02.06.08.04

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-19403

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM-2008-215108

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov

14. ABSTRACT

This paper introduces the fundamental concepts of fault tolerant computing. Key topics covered are voting, fault detection, clock synchronization, Byzantine Agreement, diagnosis, and reliability analysis. Low level mechanisms such as Hamming codes or low level communications protocols are not covered. The paper is tutorial in nature and does not cover any topic in detail. The focus is on rationale and approach rather than detailed exposition.

15. SUBJECT TERMS
Fault Tolerance; Redundancy; Reliability Analysis; Safety; Voting

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER ***(Include area code)***
U	U	U	UU	53	(301) 621-0390