THE EFFECTS OF VOTING ALGORITHMS ON N-VERSION SOFTWARE RELIABILITY

by

Gregory L. Greeley

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1987

Copyright (c) 1987 Gregory L. Greeley

Signature of Author ______________________
Department of Electrical Engineering and Computer Science
June 1, 1987

Certified by ______________________
Peter Elias
Thesis Supervisor

Certified by ______________________
Linda Alger
Company Supervisor

Accepted by ______________________
Arthur C. Smith
Chairman, Committee on Graduate Students

Archives
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 0 8 1987
LIBRARIES

THE EFFECTS OF VOTING ALGORITHMS ON N-VERSION SOFTWARE RELIABILITY

by

Gregory L. Greeley

Submitted to the Department of Electrical Engineering and Computer Science on June 1, 1987 in partial fulfillment of the requirements for the degree of Master of Science.

Abstract

Highly reliable computer systems that can tolerate software faults are needed for applications such as aircraft and spacecraft control. To solve this problem, Draper Laboratory has developed a specialized architecture for running N-version software. This architecture solves several of the technical problems associated with N-version software. Software failures, however, are fundamentally different from their hardware counterparts and a new way of modeling these failures is needed. This thesis presents a Markov model for software failures in multi-version systems that is not a converted hardware failure model, but instead reflects software's underlying failure process. This thesis also investigates the effects of decision algorithms on the reliability of a 4-version, software fault tolerant system. Data from a previous N-version experiment are used to compare a new decision algorithm with a standard majority vote. Given certain assumptions about the data used, this new algorithm demonstrates an decrease in failures rates that ranges up to an order of magnitude. The results of these simulations show that, under the given assumptions, the decision algorithm can take advantage of the past history of software failures to choose a correct answer when a standard majority vote would have to fail safe.

Thesis Supervisor: Peter Elias
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Linda Alger and Jaynarayan Lala for always saying "yes" and making me believe that this work was worthwhile. I would also like to thank Sally Johnson of NASA Langley for always saying "no" and reminding me that it takes a lot to convince some people. A special thanks to Peter Elias whose insights helped to pull me out of several dead ends.

This work was done at the Charles Stark Draper Laboratory with support from NASA under Langley contract NAS1-18061.

Publication of this report does not constitute approval by the Draper Laboratory or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to the Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.

Gregory L. Greeley

Permission is hereby granted by the Charles Stark Draper Laboratory, Inc. to the Massachusetts Institute of Technology to reproduce any or all of this thesis.

To Linda Alger, for her support and guidance.

Table of Contents

Abstract 2
Acknowledgments 3
Table of Contents 5
List of Figures 6

1. Introduction 7
1.1 Previous Work 9
1.2 The FTP/AP Architecture 10
1.3 Modeling Software Failures 11
1.4 Comparing Algorithms 12

2. Current Research on Software Failures 13
2.1 Avizienis and Chen 13
2.2 Kelly 14
2.3 Boeing and the Research Triangle Institute 15
2.4 Knight and Leveson 15
2.5 Implementations in Operation 17

3. The FTP/AP Architecture 20
3.1 The Fault Tolerant Processor 20
3.2 Communication with the Attached Processors 24
3.3 Architecture Solutions to N-version Problems 26
3.3.1 Input Consistency 26
3.3.2 Interversion Communication 27
3.3.3 Protection of the Support Environment 27
3.3.4 Version Synchronization 28
3.3.5 Meeting Real-Time Constraints 28

4. A Markov Model for Software Failures in a Multi-Version System 29
4.1 Eckhardt and Lee's Theory on Software Failures 30
4.2 Modeling Multiple Programs 32
4.3 A Markov Model for FTP/AP Reliability 35

5. Decision Algorithms 41
5.1 N-Version Software Reliability 41
5.2 Analysis of Knight's Data 43
5.3 The Confidence Voter 44
5.4 Simulating the Confidence Voter's Operation 46
5.5 Results of the Simulations 47

6. Conclusions 56

List of Figures

Figure 3-1: The FTP Data Exchange Mechanism 23
Figure 3-2: FTP/AP Architecture 25
Figure 4-1: A Case for which Redundancy Reduces Reliability 31
Figure 4-2: A 2-Dimensional Slice of an Input Space 32
Figure 4-3: Markov Model for Multiple Software Failures 39
Figure 5-1: A 2-Dimensional Slice of an Input Space 42
Figure 5-2: Types of 4-Version Combinations 44
Figure 5-3: Combinations With Only One "Bad" Pair 48
Figure 5-4: Results of the First Simulations 49
Figure 5-5: Learning Curve For the Confidence Voter 50
Figure 5-6: Counter Values for the First Simulations 50
Figure 5-7: Combinations With Several "Bad" Pairs (Difference > 2) 51
Figure 5-8: Results of the Second Two Simulations 52
Figure 5-9: Learning Curve For the Confidence Voter (Case 4) 53
Figure 5-10: Combination With Several "Bad" Pairs (Difference < 2) 54
Figure 5-11: Results of the Last Simulations 55
Figure 5-12: Counter Values for the Last Simulation 55

Chapter 1

Introduction

As our reliance upon computer systems increases, so does our need for reliable systems. In terms of computer hardware, techniques for increasing reliability through fault tolerance have been well developed. Computer software, however, has not received the same attention. In fact, many of the "highly reliable" systems in operation cannot tolerate a software failure [3].

One of the reasons that software fault tolerance has been ignored for so long is that software failures are fundamentally different from their hardware counterparts, which are mostly due to the random failure of individual components. Software failures differ from hardware failures in three ways. First, software failures are due to latent flaws in the design or implementation, not component failures. Second, software failures are a *deterministic* function of the programs' input, they are not random. And, most importantly, failures of different versions of a program are not necessarily independent events [14] [23]; they are linked by a common input space. This input space has regions in which the probability of failure is much higher than average [8].

Past efforts in software reliability have concentrated on techniques for discovering and removing software faults. Rigorous validation and verification has worked well because software reliability does not degrade with time, as hardware reliability does. This technique, however, is not perfect. In past avionics systems 50 percent of the software failures occurred during the operational phase of the system's life cycle [20]. Given the increasing complexity of software for life critical missions, one can assume that these systems will also contain latent software faults and that some kind of software fault tolerance will be necessary to prevent the possible loss of life or mission aborts.

This thesis covers N-version programming, a technique that has the potential to create fault tolerant software. N-version programming uses N versions of a program (where N > 1) which have been independently created to satisfy a common specification. After each iteration of the N versions, a decision algorithm compares the results, chooses a correct answer, and identifies any faulty versions. This thesis compares two different decision algorithms to see if they can substantially affect the reliability of an N-version system. The first algorithm is a standard "majority" vote. If two or more of the 4 versions agree on an answer, then that answer will be considered correct. If there is no majority or if there is a 2:2 split, then the system will fail safe.[1] The second algorithm is called the confidence voter. The confidence voter maintains a record of the failure rates for the different versions and uses that information to choose a correct answer when there is an even split (i.e., two versions agree on one answer while the other two versions agree on a second answer.)

When working with software failures (as opposed to hardware failures), there are some complications that must be taken into account. First is the fact that the software failures in different versions are not necessarily independent. That is, the probability of two versions failing on a single iteration may not be the same as the probability of the first version failing times the probability of the second version failing. If the currently accepted theories of software failures are correct, then the probability of multiple coincident failures could be higher than if the failures were independent. The second complication is that sequential failures are probably not independent. This means that the probability of a second version failing, given that one version has failed on a previous iteration, could be higher than the probability of the first version failing.

We also must assume that we have no prior knowledge of past failure rates. While

---

[1]The term majority is actually a misnomer. The technically correct term for this type of vote would be "plurality."

we may have some information on past failures, each time a fault in the program is fixed, it alters both the failure rate and the correlation of failures among the N versions. We also assume that the failure rate for a pair of versions with different answers is *not* independent of the failure rate of a pair of versions with identical answers. Without this assumption, the ability to use past information would be seriously impaired.

1.1 Previous Work

Chapter 2 outlines the past research in the area of software reliability and N-version programming. Because there has been relatively little work done in this area, there is a severe lack of hard data on how failures interact among the multiple versions. This chapter, however, covers the major discoveries and systems in this area.

In 1975 Chen and Avizienis [6] initiated a project to investigate the feasibility of multi-version software and to investigate its cost-effectiveness. Their research did not attempt to address the issue of correlation among failures, and the number of test cases were insufficient to statistically validate any conclusion on the interrelation of failures in the different versions.

Kelly [13] performed multi-version experiment study on the effects of specification techniques on the interrelation of the different versions of a program. One of Kelly's conclusions is that specification errors are a major difficulty for N-version software because multiple versions coded from the same specification are unlikely to overcome any errors or ambiguities in the specification.

In a Boeing study [22], Nagel and Skrivan recorded the interfailure times for programs after each detected fault has been fixed. Their results indicate that the program error rates exhibited a log-linear trend and that the rate at which different faults generate errors in the same program vary widely.

The Research Triangle Institute [7] evaluated three new versions of the launch interceptor used in Knight's experiment [14]. RTI's results corroborate the results of the Boeing study [22].

Knight and Leveson [14] [16] conducted a major 27-version experiment to study the dependence of failures among different versions of software. They ran the 27-versions of a launch interceptor through 1 million randomly generated test cases. They concluded that the failures were not independent. They also conducted and empirical study of the failure rates of the different combinations of 2 and 3 versions. The second study concluded that multiple versions of unreliable software cannot be put together to produce a highly reliable system.

Both Airbus Industries [11] and Boeing [27] have 2-version implementations in operation. These implementations had to meet very stringent requirements in terms of reliability, and initial operations have been promising.

1.2 The FTP/AP Architecture

Chapter 3 discusses the Fault Tolerant Processor/Attached Processor (FTP/AP) architecture, which has been designed specifically for N-version programming. The FTP/AP architecture consists of a quadruply redundant Fault Tolerant Processor (FTP) and four Attached Processors (AP's). Each of the FTP's channels (or processing units) is connected to a single AP. The FTP provides hardware fault tolerance by using four tightly synchronized, redundant processing units, while the AP's provide the software fault tolerance by running, in parallel, four independently developed application programs that perform critical flight control functions.

In a previous paper on fault tolerant software [5], Avizienis discusses several technical problems with N-version programming. Among the problems noted by Avizienis

were the assurance of input consistency, interversion communication, protection of the support environment, and meeting real time constraints. The FTP/AP architecture provides solutions to each of these problems.

By resolving these technical difficulties, the reliability of N-version software is limited only by the effects of interversion dependencies. Now the question that must be answered is: is there some N such that N (not necessarily independent) versions of a program provide enough increase in reliability to justify the increased cost of the N versions?

1.3 Modeling Software Failures

When attempting to model software failures, one's first thought would be to use the same model as hardware failures. Unfortunately, the two failure mechanisms are different. Hardware models assume that failures are random, statistically independent events with a constant failure rate. Software failures, on the other hand, are deterministic functions of the inputs to a program, and the failures of multiple versions have been shown to be dependent. Chapter 4 discusses a new method of modeling multiple software failures that is the result of investigating the underlying failure mechanism, not just the symptoms of the failures. This Markov model assumes that the dependencies among multiple versions are a result of failure regions that are distributed throughout the input space. When the input to these programs are in the failure regions, then there are some given probabilities that each version will fail.

1.4 Comparing Algorithms

Chapter 5 covers the different algorithms and how they are compared. Comparisons were done using a simulated 4-version system. The simulated failure rates for the four versions are based on Knight's 27-version experiment [14]. Using Knight's data, each failure mode for the 4-version combination is assigned a probability. The simulator starts an iteration by randomly selecting a failure mode. Once its determined which versions (if any) have failed, the correct versions are given identical floating point numbers as their "result." The failed versions are given different numbers as their "result." The simulator then sends these 4 numbers to the voters and evaluates their performance.

Chapter 6 presents the conclusions of this thesis.

Chapter 2

Current Research on Software Failures

The field of software fault tolerance is new. Because of this, there is relatively little work that is readily available or useful. This chapter covers some of the better known research that has been conducted recently. The works listed here are either experimental or operational in nature. Chapter 4 covers the more theoretical research.

2.1 Avizienis and Chen

Probably one of the first experiments in N-versions software was conducted by Chen and Avizienis [6] in 1975 at UCLA. The experiment was performed to investigate the feasibility and cost-effectiveness of N-version software. For the experiment, a graduate seminar course was divided into 18 teams. Each team was given the specification for RATE (Region Approximation and Temperature Estimation), a program for computing dynamic changes of temperatures at discrete point in a particular region of a plain [sic]. The problem specification for RATE also specified one of three possible algorithms to solve the differential equations that governed the temperature changes.

The 16 programs received were tested against six test cases. Based on the results of these tests, four programs were selected for further tests. In addition to these four programs, three programs were developed by the authors, bringing the total to seven. Three of these programs used the first algorithm; two used the second; and two used the third. These programs were grouped into 12 3-version combinations and tested with 32 test cases. Of the 384 test cases:

1. 290 cases contained no bad version
2. 71 cases contained one bad version
3. 18 cases contained two bad versions, and

4. 5 cases contained three bad versions.

The authors were investigating the cost-effectiveness of fault tolerant software and therefore did not attempt to draw any conclusions on the subject of correlated failures. Also the number of test cases were not really large enough to validate any conclusions. The authors conclude that the methodology for implementing N-version programming is relatively simple and can be generalized to other similar applications and that in some cases, 3-version programming has been effective in preventing failure due to defects localized in one version of code.

2.2 Kelly

Kelly also conducted an N-version experiment at UCLA [13] [4]. This experiment studied how different specification techniques effected the reliability of multiple versions. For his experiment, Kelly used an airport scheduler data base. The specifications for this scheduler were written in English, OBJ and PDL.[2] These specifications were given to a group of junior, senior, and graduate students at UCLA.

Of the 18 versions completed, seven were from OBJ specifications; five from PDL; and six from English. These 18 versions were then run through 100 input transactions.[3] On the average, the individual version produce a correct output only 73.1% of the time. When put in three version combinations, 78.5% of the time at least two versions agreed on a correct output. Also, 3.1% of the time, two or more versions agreed on an *incorrect* output.

---

[2]OBJ is a formal language for writing and testing algebraic program specifications developed at UCLA by Goguen from 1976 to 1979. It is also an applicative, nonprocedural programming language. PDL is a nonformal language for writing program specifications and designs. It was developed by Cain, Faber, and Gordon in the early 70's and functions as a production documentation tool.

[3]Eleven versions would abort on invalid input. These versions were fitted with an acceptance test that detected and attempted to recover from these catastrophic faults.

2.3 Boeing and the Research Triangle Institute

In a Boeing study [22], two programmers independently wrote three FORTRAN programs. Each of these programs was run (using randomly-generated inputs) until a failure. The fault that produce the failure was corrected, then the program would be run until the next failure. Nagel and Skrivan recorded the interfailure times and their results indicate that the program error rates exhibited a log-linear trend as each fault is removed and that the failure rates of different faults in the same program vary widely. Using some of the same procedures, Janet Dunham and John Pierce at the Research Triangle Institute [7] studied the Launch Interceptor program used by Knight [14]. Their results agreed with those found in the Boeing study. Both investigations, however, made no conclusions on the impact of dependencies among faults.

2.4 Knight and Leveson

In their well known paper [14] Knight, Leveson, and Saint Jean conclude that "N-version programming must be used with care and that analysis of its reliability must include the effect of dependent errors." This conclusion came out of an experiment that used 27 independently written programs which satisfied a common specification. These 27 programs were run through a million randomly-generated test cases. While the programs were individually fairly reliable (no failure rate was > 1%), the probability of multiple failures occurring was so high that the hypothesis that the failures were independent was easily rejected. This conclusion is, however, conditioned on the application used, and the authors note that their results may or may not extend to other programs.

Knight and Leveson's second paper [16] uses the 27 versions from their previous paper in different two and three version combinations. For the two version systems, the average failure rate (1.19×10^{-2}) was worse than the average failure rate for a single version

(6.98×10^{-4}). However, when there was an error, the average probability that it was undetected is small (3.24×10^{-3}). This gives an average probability of 3.9×10^{-5} for an *undetected* failure. For three version systems, the average probability of a system failure was 3.67×10^{-5} and of these failures, the average probability that a system failure was detected is 0.65. This gives an average probability of 1.28×10^{-5} for an *undetected* failure.

Using the average results, one could say that a 3-version system will give a 19 fold decrease in the failure rate. Unfortunately, the reliability of each different 3-version system varied widely. In come cases, using more reliable versions actually *decreased* the reliability of the system because there were more multiple failures. Regardless of these problems, a 19 fold decrease in the failure rate is much less that one would expect. Under the model where the programs are independent, the failure rate would decrease by several orders of magnitude.

Again, the authors warn that these results and conclusions apply to the specific software used. But they conclude that one *cannot* assume that low quality programs can be combined to form a reliable system, and one cannot assume that very high quality programs can be combined to from an ultra high reliability system.[4]

Several questions are still left unanswered, however. First is the effects of "error bursts." Each of the iterations performed in this experiment were randomly-generated test cases, not successive iterations. Thus, no conclusions can be drawn about bursts of errors occurring as the inputs to the program take on certain "error prone" values. Also, each test case is unusual in that the actual launch-interceptor will encounter relative few cases similar to the test cases in its operation mode. Thus, how the reliability of each version in this experiment would translate to reliability in an operational system is unclear.

---

[4]While the authors are correct in stating that one *cannot* assume low quality programs can be combined to form a reliable system, their statement about combining very high quality programs to form ultra high reliability systems has no basis in their experiment. Their experiment used only low quality programs and it is not clear how their results would apply to a highly reliable system.

The second unanswered question, and the one which is more important to this thesis, is the ratio of identical versus non-identical correlated errors. Knight has published no data on this figure. These data are critical to the performance of the confidence voter. If, for example, the number of non-identical correlated errors is independent of the number of identical correlated errors between two versions, then the confidence voter has no way to gather reliable information on correlated errors. Chapter 5 discusses this problem in more detail.

2.5 Implementations in Operation

While the N-version technique is relatively new, there are already systems in operation that use this technology. When designing the slat and flap controller for their A310 passenger transport, Airbus Industrie decided to use a "digitally implemented" control system [11]. This system has no mechanical linkage between the cockpit controls and their respective actuators. This system offers advantages in terms of weight, maintainability, and flexibility.

There was, however, concern about reliability. The system's requirements were as follows:

- No more than 1 (indicated) failure to complete a commanded movement every 10^5 flight hours.
- No more than 1 (non-indicated) failure to complete a commanded movement every 10^9 flight hours.
- No more than 1 uncommanded movement every 10^9 flight hours.
- No more than 1 asymmetrical surface deployment every 10^9 flight hours.

These strict requirements led to the choice of a 2-version system that uses dissimilar architectures for operation. Care was taken to ensure independence between the two channels that control the flaps. The hardware uses different microprocessors, contains separate clocks, and operates asynchronously. The software was created from different

specifications, written in different assembly languages, and used assemblers on different host computers.

The goal of all this effort is to reduce the number of double failures. If only one of the channels has a failure, the system will notice the disagreement. After a disagreement is caught, the system will fail-safe by locking the flaps and slats in their current position. Double failures, on the other hand, could produce two identical, incorrect answers. In this case, the system will experience a catastrophic failure.

While this relatively straight forward systems will not necessarily work for other flight control applications, it has proved successful in this case. The time to test and certify the controller was reduced from what would usually be expected. As of the writing of the Airbus paper [11] the A310 was in revenue service with two airlines and the current mean time between failures (MBTF) is expected to be 11,000 hours (slightly better than the 10,000 hour specification).

Boeing also uses a dual-dissimilar Sperry SP-300 flight control system in their 737-300 aircraft [27]. As in the previous case, Boeing was attracted to a digital Flight Control Computer (FCC) because of the increased flexibility and maintainability. However, they were faced with the difficult problem of verification. Engineers responsible for analyzing possible failures were faced with an almost limitless quantity of unique failure modes. For certification, it is necessary to prove that for a one-hour exposure time the probability of a hazardous event resulting from a generic processing fault (software or hardware) is less than 10^{-9}.

To overcome these problems, Sperry developed the SP-300, which uses dual dissimilar software as part of its reliability strategy. Critical software, such as approach, landing and go-around, was developed for the dissimilar secondary processor as well as the primary processor. The secondary processor software package remained small, comprising only 10 to 15 percent of the total software package.

Because the 737-300 used a dual-dissimilar processor, the FAA gave the processor certain ''credits'' during the certification plan negotiations. Examples of these credits are:

- No special analysis is required to assure the absence of support software design errors.
- Coding errors (at the detail level) in the primary and secondary CPU's causing the same effect do not have to be considered.
- The criticality of all the primary software is classified ''essential.''
- The secondary software retains the ''critical'' designation; however, module test coverage analysis is not required as part of the overall test coverage analysis.
- Processor failure modes and effects analysis are not required.

The author concludes that the dissimilar software implementation of a single set of aircraft-control algorithms is an effective technique for protecting against software faults. This conclusion however is motivated at least in part by the FAA's relaxed certification requirements, which helped make this system economically viable.

Chapter 3

The FTP/AP Architecture

The Fault Tolerant Processor/Attached Processor Architecture was conceived at Draper Laboratory as an architecture that would make use of the Laboratory's Fault Tolerant Processor [10] in order to solve several of the technical problems associated with N-version software [1]. The core of the FTP/AP is a quadruply redundant Fault Tolerant Processor. Connected to the FTP are four Attached Processors. These processors execute the different versions of software while the FTP coordinates the overall operation of the system.

This unique architecture resolves the technical problems associated with the operation of an N-version system. Because difficulties with input consistency, interversion communication, support environment protection, and real time execution are no longer issues, the inherent reliability of N-version software becomes critical to the success of this type of system. The two major sources of failure in multi-version software are specification errors and programmer errors. This thesis will concentrate on the effects of programmer errors rather than the effects of incorrect or ambiguous specifications.

3.1 The Fault Tolerant Processor

Draper Laboratory has many years of experience in developing various fault tolerant systems [12]. Draper has produced triply and quadruply redundant versions of the FTP based on six different commercial microprocessors. These redundant systems have found their way into both research aircraft [25] and research reactors [2].

The FTP achieves a high level of reliability by using four identical processing elements (channels) that perform identical operations on identical input. The FTP will

continue to operate correctly even after two sequential hardware failures because data from the operational channels will vote out and mask data from the faulty one. The design goal of the FTP is to produce a fault tolerant virtual processor out of these four tightly synchronized channels. Thus, the programmer who writes applications for the FTP does not have to know that his code is running on several processors that continually exchange all input and vote all output. In fact, the applications programmer is not involved in any of the redundancy management aspects of the FTP. In the Draper Fault Tolerant Processor, specialized hardware maintains synchronization and handles communication between processing sites. This solution not only reduces the software overhead, but, in fact, allows the FTP to be treated as a virtual simplex processor.

Synchronization is achieved through the use of the Fault Tolerant Clock (FTC), which all four channels use to synchronize their local system clocks. This FTC and its associated hardware keep the processors' local system clocks in tight synchronization so that all four channels maintain the same *rate* of instruction execution. The main advantage of this close synchronization is that the channels can precisely synchronize their code execution and data exchanges. That is, one channel can send data to the other three channels *without* having to coordinate the exchange in advance. This ease of communication removes the burden of maintaining processor synchronization from the applications software and also removes the processor wait states that would have been used to resynchronize the channels prior to every exchange of data.

Exchanging data, which is necessary both for communication with the other channels and for voting outputs, is accomplished by the hardware data exchange mechanism. Whenever data is sent between channels, it is voted on a bit by bit basis: the hardware compares each set of four bits and masks out any bit that disagrees with the other three. If an error is detected, a hardware error latch is set, noting the type of exchange and the channel at fault. Fault detection is implemented by comparing the voters' input and

outputs; fault isolation uses the pattern of errors latched by the voters. By implementing the fault detection and masking in hardware, the FTP frees the software from this burden and helps provide the virtual processor abstraction. The theoretical basis for this interconnection scheme's protection against Byzantine failures can be found in [18].

Figure 3-1 shows a schematic representation of a triplex FTP's data exchange mechanism. (The data exchange for a 4 channel FTP is a simple extrapolation from this figure.) Note that there are three major elements in the mechanism: the transmitters, the interstages, and the receivers. These elements are connected in several different ways. First, each channel's transmitter has a bi-directional link to the other channels' transmitters. These links are used for immediate access to raw data when replicating data. Second, each transmitter has a link to its interstage. This link is used to send data to be latched by the interstage for further re-transmission. Finally, each interstage has a link to each of the four receivers. These links are used by the interstages to send a copy of their data to each channel.

The data exchange mechanism is the FTP's primary means of correcting for random hardware faults. It has been shown [24] that systems such as the FTP need two basic types of data exchanges: a replication and a direct vote. A replication is used in the case where a single channel has a local value that must be sent to the other channels. Since a direct broadcast is vulnerable to Byzantine failures, interstages are used to replicate the data. A direct vote, on the other hand, is used in the case where three channels have computed outputs, such as actuator commands or terminal output, that must be voted before transmission.

In their well-known paper on the Byzantine General's problem [19] Lamport, *et al* show that a system needs $3m + 1$ fault containment regions to tolerate m faults. The FTP is divided into eight fault containment regions; the four processors and the four interstages. Each of these regions is isolated (physically and electrically) so that a fault in one cannot

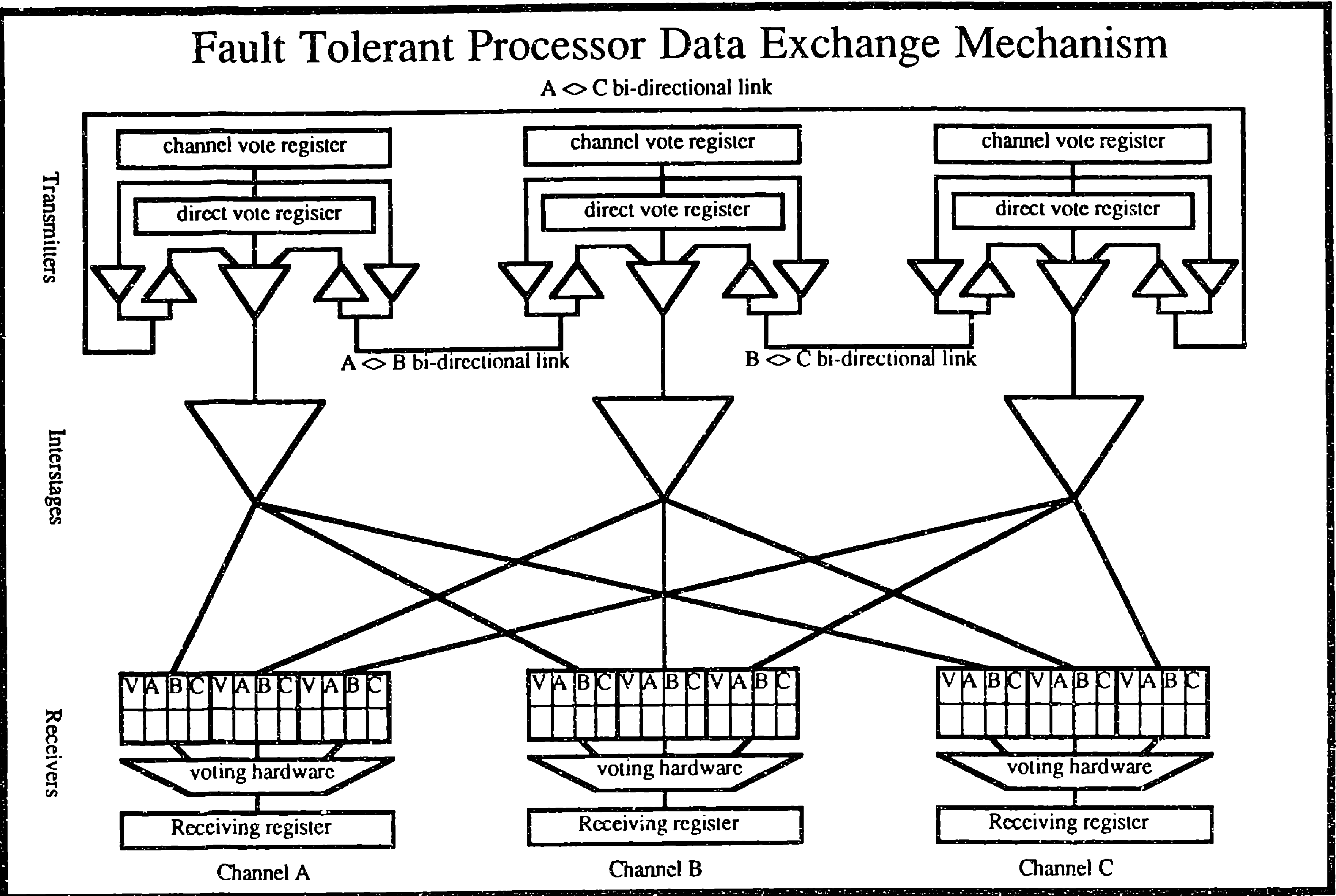


Figure 3-1: The FTP Data Exchange Mechanism

cause a failure in another. Thus, a processor or interstage may transmit bad data due to a single fault, but the bad data will be masked out by the voting process. After the first failure, the FTP still has six fault containment regions (two more than theoretically necessary) and can therefore tolerate a second hardware fault.[5] In this manner, these eight regions will allow the FTP to tolerate as many as two sequential failures.

3.2 Communication with the Attached Processors

In contrast to the specialized nature of the FTP, the Attached Processors have only one unique feature: a dual-port memory that connects with a channel in the FTP. Currently, four VAX 11/750's are being used to emulate Attached Processors. Figure 3-2 shows how the FTP/AP architecture is arranged. The host VAX runs a standard transport aircraft environment simulation. That is, it simulates all the input from sensors and uses the actuator outputs to determine the effects on the aircraft. The FTP then operates as if it were controlling an actual aircraft. It sends the appropriate information to each AP, which runs the programs to control the aircraft. After a set amount of time, the FTP compares the AP's outputs and chooses which answers are correct. These answers are sent back to the host VAX as actuator commands.

While simple in concept, the actual details of the operation are more complex. When the host VAX sends inputs to the FTP through the dual-port memory, the FTP uses its data exchange mechanism to ensure that each channel has the identical data. Without identical input, the multiple versions of the flight code would be useless. Using the simulation data, which is now guaranteed to be identical, each channel instructs its Attached Processor to execute a version of the flight control code. While awaiting the AP's completion of the flight control code, the FTP analyzes the results from the previous iteration and decides which answers are correct. These results are sent back to the host VAX.

---

[5]The reconfiguration algorithm treats each processor and interstage pair as a single entity and the failure of either one results in the pair being reconfigured out of the system.

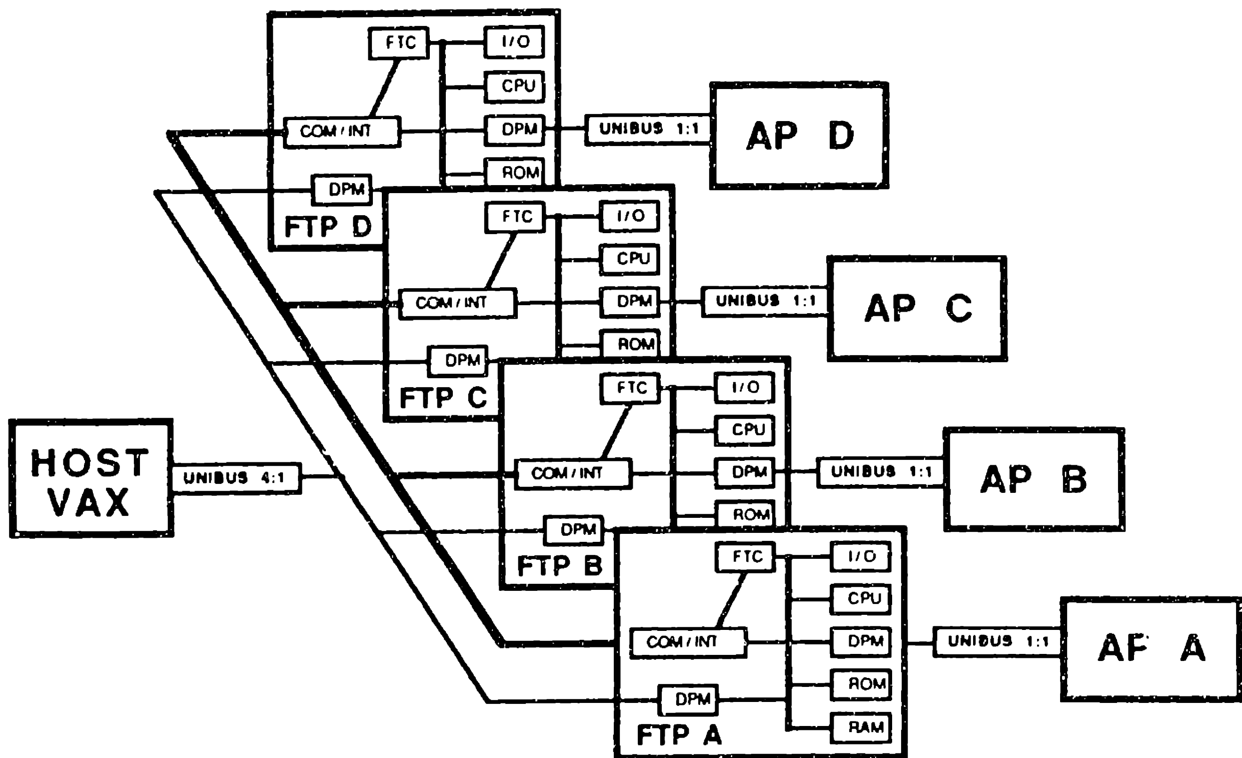


Figure 3-2: FTP/AP Architecture

If each version's output value is identical (within a given tolerance) then that value is sent to the host VAX as an actuator command. If, however, the versions do not agree, the FTP must determine the cause of the error: hardware or software. The suspect version is run in all four AP's with the original input values. If only one AP disagrees, then the error can be assumed to be the result of a hardware failure. If all four AP return the original suspect value, then the software is at fault. Software errors, such as these, are noted by the confidence voter, which maintains records on the failure rates of the versions.

Another issue is how frequently these votes should be performed. One solution is to execute the entire body of the flight code, then vote each actuator output. Another solution is to vote the outputs of the flight code's critical subroutines. The disadvantage of the first solution is that each version has more of an opportunity to diverge from the others. The second solution reduces the opportunity to diverge, but increases the overhead necessary to manage the operation of the four versions.

3.3 Architecture Solutions to N-version Problems

This section discusses, in detail, how the FTP/AP architecture resolves technical problems that have been discovered in past fault tolerant software research. Many of the detractors of N-version software use these difficulties to "prove" that N-version software will not work. By removing these technical difficulties, we can actually concentrate on analyzing the inherent reliability (or unreliability) of the N-version technique.

3.3.1 Input Consistency

One problem with past N-version implementations is that a single point failure in the host computer can cause the different version to receive different inputs, thus causing disagreements in the versions' outputs. The FTP/AP architecture guarantees input consistency by using the FTP data exchange hardware. The FTP reads the sensor and, using the data exchange mechanism, creates four bit-for-bit identical values, one in each channel. Each channel then sends this value, which is guaranteed to be identical with the other three, to its AP.

A single point hardware fault in the FTP will be voted out and have no effect on the versions. A fault in an AP can effect, at most, the input to (or output from) that one AP. Thus, by using the FTP's data exchange mechanism, only the simultaneous failure of two fault containment regions will cause different input values to be sent to the AP's.

3.3.2 Interversion Communication

In past systems, interversion communication suffers from the same problem as input consistency: single point failures. If, for example, version A needed to communicate with version B, they would read and write information in the host computer's memory. As before, a single fault could cause errors in both versions. The solution for this problem is similar to the solution for the previous problem. In the FTP/AP, data from the version in AP 1 would be sent to channel A. Channel A then retransmits that value to the other three channels via the data exchange mechanism, which guarantees that the four values are identical. This data is now available to all four versions.

Again, a single fault can not effect the FTP's data exchange. In fact, the only hardware fault that can effect more than one version is an error in the original communication. If an AP sends a incorrect value, however, all AP's are guaranteed to receive the same incorrect value.

3.3.3 Protection of the Support Environment

Several previous N-version experiments, which were run on a single processor, found that some faulty versions would abort and cause a system failure [4] [6]. The FTP/AP architecture will protect our system from single point failures such as these. Since each version in the FTP/AP architecture is running on an independent processor, both hardware and software faults are contained to the AP. Thus, while an AP may have a catastrophic failure, the system will continue running.

The FTP/AP system is well protected against random hardware faults. If fact, the FTP/AP system is vulnerable only to common-mode hardware faults and faults in the FTP's operating system. A fault in the FTP software could potentially bring down the entire system. To prevent this, the FTP uses a "hardened" kernel, a compact, well tested operating system, that reduces the probability that the FTP/AP system will have a catastrophic failure.

3.3.4 Version Synchronization

The problem of version synchronization is solved by the FTP scheduling each iteration of the N-version application. Because the FTP is synchronized, the versions will also be synchronized. The FTP will then wait a set maximum time for the iteration to complete and reads the outputs from the four versions. If any of the version have not completed at the end of this maximum period, it is declared failed and masked out of the vote.

3.3.5 Meeting Real-Time Constraints

The obvious problem of running four versions of an application in the same amount of time that one version would take is solved by simply using four processors. Since the versions are run in parallel, they take the same amount of time to run as a single version on a single processor (with a small amount of additional overhead).

In summation, the FTP/AP architecture solves the main technical problems associated with N-version software. We are now left with the fundamental common mode software failures: errors in the specification and correlated failures among different versions of software. In the next chapter, we will attempt to model the correlation of failures in multi-version systems by investigating the software's underlying failure process.

Chapter 4

A Markov Model for Software Failures in a Multi-Version System

In the process of studying N-version software, we needed to model the software failures of several different versions running concurrently. While the first thought when modeling software failures might be to use models similar to those used for hardware fault tolerant systems, this is not feasible. Hardware models assume that failures are random, statistically independent events with a constant hazard rate. [26][6] Software failure mechanisms are very different from hardware failures. Software failures are deterministic functions of the program's input and state, and it has been shown that the failures of multiple versions of a program are not statistically independent.

Figure 4-3 shows the Markov model developed at Draper. This model represents a four-version system (such as the FTP/AP) that uses the majority (or "plurality") vote to resolve disagreements. The states in this model correspond to the types of errors the system could encounter. Thus, for example, one of the states is equivalent to the condition where two versions agree on one answer and the two other versions have two different answers. (This case is referred to as a 2:1:1 split.) As you can see, this model contains a large number of transitions rates (λ_1, λ_2, λ_3, λ_4, λ_5, λ_6, λ_7, ρ_1, and ρ_2). All of these rates, however, are given in terms of five parameters: λ_{fr}, λ_{ec}, ρ_{fr}, ρ_{ec}, and π. In this chapter, we will attempt to determine a relationship between these transition rates by examining the underlying failure process, rather than the visible errors, which are only symptoms of the faults within the code.

[6] Many hardware component failures follow the well-known "bathtub" curve, which shows how the failure rate for components decreases as they are burnt-in and how the failure rate increases again as the system reaches the end of its operational lifetime. Other components have an exponentially decaying failure rate which never increases with time. In either case, for the useful life of a system, these failure rates can be approximated by a constant.

4.1 Eckhardt and Lee's Theory on Software Failures

In their papers on software failures [8] [9], Eckhardt and Lee contend that a program has a distribution, $\Theta(x)$, which is a function of the input space of the program. Thus, if a program has 3 inputs, then there exists a 3-dimensional input space, and each point in this input space is assigned a probability $\Theta(x, y, z)$. This probability describes the "propensity of a population of programmers to introduce design errors such that" the software will fail on that input. Alternatively, one can think of the distribution as the probability that a randomly selected program will fail on the given input.

The point of this model is that software failure rates are explicitly linked to the input space of the program. (Current research tends to validate this model [17].) Eckhardt also notes that an N-version system with a pure majority vote[7] will be more reliable than its single-version counterpart only if the intensity distribution is less than 0.5 throughout the operational input space. In other words, *if* there is a region in the input space where the probability that an arbitrarily chosen version will fail is greater than 0.5, then the N-version system will be *less* reliable than the average single version.

Eckhardt gives an example of an N-version system in which the probability of a version failing is equal to 0.6, but only for a small (10^{-5}) subset of the input space. Thus, when in this region of the input space, 60 percent of the versions would produce errors. Otherwise, there were no failures. As versions are added to this system, the failure rate actually increases (See figure 4-1)

Gerald Migneault at NASA Langley Research Center introduced the idea of "dark crystals" [21](which Knight calls error crystals). If one looks at a program's n-dimensional input space, there would be certain regions where the input will cause the program to fail.

---

[7]In this case, "majority" is used in the strictest sense. A 4-versions system would require at least 3 versions to agree on an answer before that answer would be used as the system's output.

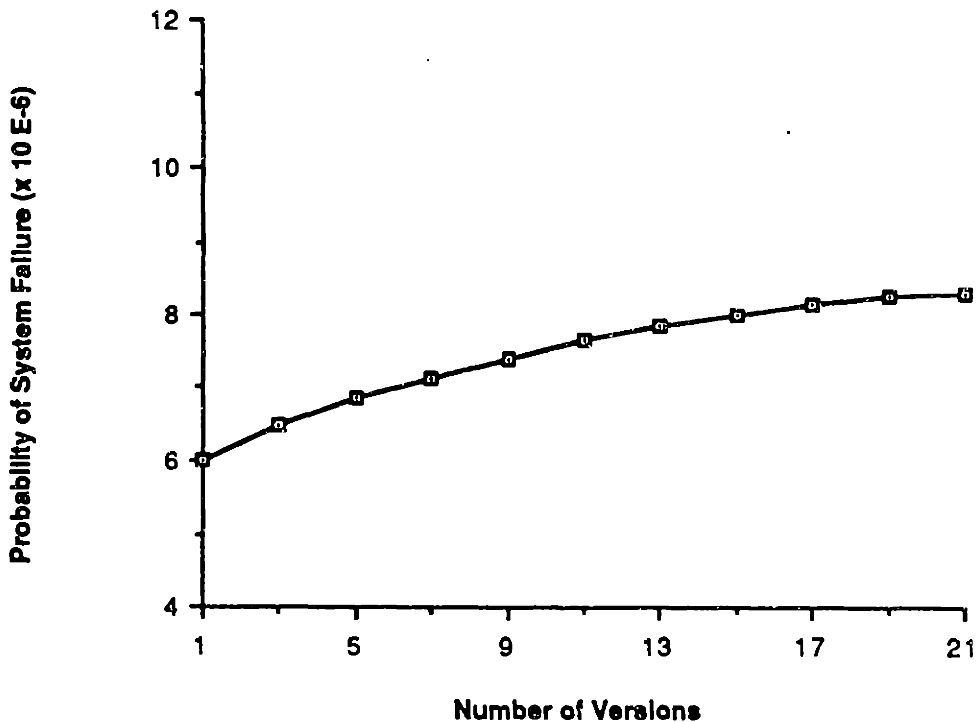


Figure 4-1: A Case for which Redundancy Reduces Reliability

These regions are the program's error crystals. According to this theory, which applies Eckhardt and Lee's model to an individual program, there is nothing random about software failures. If the input to a program is inside an error crystal, then the program will produce erroneous output. Otherwise, the code will produce the correct output. Recent work by Knight [17] has mapped some two dimensional slices of error crystals in versions of his launch interceptor application.

4.2 Modeling Multiple Programs

To model the operation of a program, we can think of a point traveling some path through an n-dimensional input space. During its travel, the point will occasionally enter and leave these error crystals. Figure 4-2 shows a 2-dimensional slice of an input space. Within this space there are two crystals, A and B, that overlap. Because these regions share a large common boundary, an input following the path shown would experience a simultaneous double failure, then two sequential recoveries. Unfortunately, there are some complications with using this model.

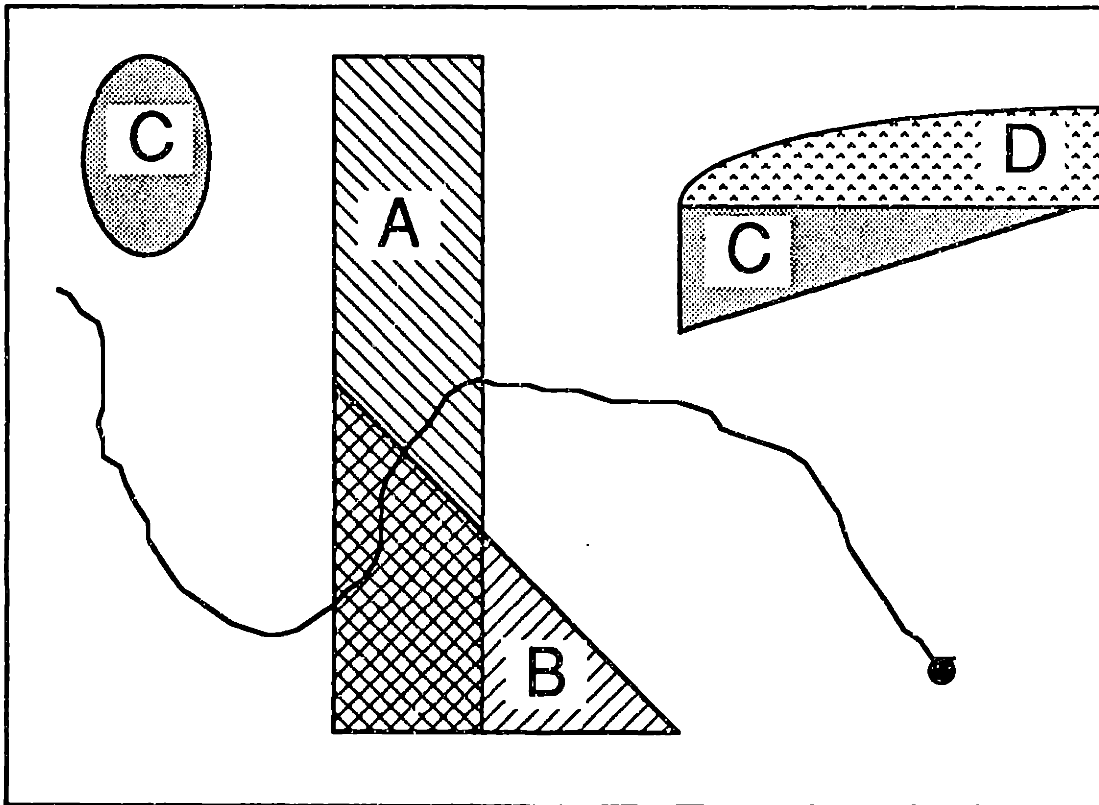


Figure 4-2: A 2-Dimensional Slice of an Input Space

The first complication is that we cannot assume versions will fail independently.

Thus, if the probability of a single failure on a given iteration is a multiple of λ, the probability of two versions failing simultaneously is not necessarily a multiple of λ^2.

The second complication is that sequential failures of different versions are probably not independent. That is, when one version is in a failed state, the probability that the other versions will fail is likely to be higher. This view follows along with the theory put forth by Eckhardt and Lee: versions tend to fail together. Given this theory, we would like to take the more conservative view and say that the probability of a second version encountering an error crystal, while the input point is already inside the first version's error crystal, is higher than the probability of the first version encountering the original error crystal.

Using Eckhardt and Lee's model, one can easily understand, then, the concept of failure regions. The distribution of each version's error crystals may be uniform throughout the input space, but the distribution of the error crystals from four versions would definitely nct be uniform. Thus, failure regions are areas of the input space where these error crystals from the four versions collect. These are the inputs which test or exercise special cases in programs, and it has been shown that programmers tend to make mistakes when then encounter special cases [15]. Because we believe that error crystals collect in certain areas, we will model the probability of failure for each version as increasing while any version is inside an error crystal.

By using this concept of failure regions, we are able to model the various transition rates with only a few parameters. First, however, we must make a critical assumption: the failures of different versions inside a failure region are independent. By making this assumption, we are saying that the dependencies among versions are due to the failure regions. In fact, one can think of a failure region as the area of an input where the specification for the program requires some type code which is more difficult to write. An example of this would be an *if* or *switch* statement. The probability that one programmer

will write a faulty *if* statement is not affected by the other programmers' *if* statements. It is the fact that all the programmers have to write an *if* statement that increases the probability of each version failing on that input. Another example is a program that takes three points as its input. There may be a special case when all three points rest on the same line. Once the input is in a special case, which programs actually make the mistake can be treated as being independent; the dependency comes from the fact that all programs are in a special case simultaneously.

For our model, then, we use two parameters to describe the transition rates into and out of failure regions: λ_{fr} and ρ_{fr}. λ_{fr} is the probability that the input point will enter a failure region on a given iteration. ρ_{fr} is the probability that the input point, which is in a failure region, will exit that region. In an intuitive sense, λ_{fr} controls the density of failure regions within the input space while ρ_{fr} controls the average size of the regions.

After the input point has entered a failure region, we use the parameters λ_{ec} and ρ_{ec} to describe the transition rates into and out of error crystals. As with failure regions, λ_{ec} could be thought of as controlling the density of error crystals within the failure regions while ρ_{ec} controls the average size of each crystal. For continuous processes, such as an aircraft's heading, we would expect the input point to travel through error crystals in a linear fashion. For non-continuous processes, the input point will tend to "jump" through the input space. The net result is that ρ_{ec} reflects the average number of iterations the input point is expected to stay inside an error crystal.

In the model, these parameters are combined to form the transition rates. For example, the probability that the input point will enter at least one error crystal on its next iteration is

Probability(enter failure region) x
Probability(enter error crystal | inside failure region) x
number of versions

or

$$\lambda_{fr} \times \lambda_{ec} \times 4$$

Note that using this model, the rate at which failures occur (given that you are not in a failure region) is $4\lambda_{ec}\lambda_{fr}$, while the rate for double coincident failure is $6\lambda_{ec}^2\lambda_{fr}$. The values of $(4\lambda_{ec}\lambda_{fr})^2$ and $6\lambda_{ec}^2\lambda_{fr}$ are not necessarily equal, implying that the failures are not independent. Also, the rate of additional failures (given that there is already a single failure) is $3\lambda_{ec}$. The fact that the input point has entered the failure region has changed the probability that a single version will fail.

We also need to model the ratio of identical incorrect outputs versus non-identical incorrect outputs for multiple software failures. To do this, we assume that once a version has failed, it has a nearly infinite choice of different incorrect algorithms, each of which returns a unique value as its output. We also assume that there is one algorithm that is much more likely to be selected than any of the others. The probability of choosing this one "popular" incorrect algorithm, given that the version has failed, is π. The probability of choosing one of the nearly infinite number of other incorrect algorithm is $(1 - \pi)/n$ (where n is the number of other incorrect algorithms). If both versions choose the popular incorrect algorithm, then they will have identical incorrect answers. (This happens with probability π^2.) If either version (or both) choose any algorithm other than the popular incorrect algorithm, they will never agree on an incorrect answer. Each individual "unpopular" algorithm is considered so unlikely to be chosen that we assume that it will never be chosen more than once.

4.3 A Markov Model for FTP/AP Reliability

Using the information from Eckhardt and Lee's paper, we will try to model the software failures of a 4-version system. As the four versions are running, we can imagine their input point wandering in the programs' input space. Evenly distributed throughout this space are failure regions. The input point enters these failure regions at a rate of λ_{fr}

and, once in, exits at a rate of ρ_{fr}. Once the input point enters a failure region, error crystals are evenly distributed throughout the region. The input point enters these crystals at a rate of λ_{ec} and, once in, exits at a rate of ρ_{fr}.

Figure 4-3 shows the Markov model for software failures. All the transition rates for this model are based on the five parameters previously discussed, λ_{fr}, λ_{ec}, ρ_{fr}, ρ_{ec}, and π. Thus, λ_1, the failure rate for a single software module, is simply λ_{ec}. (Both rates assume that the input point is already inside a failure region.) The rate of dual coincident failures, is λ_{ec}^2. These failures, however, must be divided into identical failures, which occur at the rate λ_4, and unique failures, which occur at the rate λ_2. Of all the double failures, then, we assume that π^2 will be identical and the remaining $(1 - \pi^2)$ will be unique.

Triple failures occur at the rate λ_{ec}^3. These failure, however, must also be divided into those failures with unique outputs (λ_3) and those failures where two or more outputs agree (λ_5). The probability of all three outputs failing identically is π^3. The probability of any 2 (but only 2) outputs failing identically is $3\pi^2(1 - \pi)$. Thus, the probability that two or more outputs agree on an incorrect output is $3\pi^2 - 2\pi^3$. As a result, the probability the the three incorrect outputs are unique is simply $1 - (3\pi^2 - 2\pi^3)$.

Quadruple failures occur at the rate λ_{ec}^4. These failures are, again, divided into two types: those with all unique outputs (λ_6), and those with 2 or more identical outputs (λ_7).[8] The probability that all 4 answers are identical is π^4; any 3 (but only 3) is $4\pi^3(1 - \pi)$; any 2 (but only 2) is $6\pi^2(1 - \pi)^2$. Combining all these figures, the probability that two or more incorrect answers will agree is $6\pi^2 - 8\pi^3 + 3\pi^4$. The probability that all answers will be unique is just 1 minus that number.

These rates, λ_1 through λ_7, are used to create the transition rates between the states of

[8]Note, however, that this model does not include a 2:2 split as a possible outcome. This is because the identical incorrect answers are modeled by the one "popular" incorrect answer. A quadruple failure that has two pairs of incorrect answers implies that there are two "popular" incorrect answers.

the model. Of the 6 states, only the state labeled "No Failures" corresponds to the point being outside of a failure region. All transitions *into* this state, then, occur at the rate ρ_{fr}, and the sum of all the transition rates *out* of this state must equal λ_{fr}.

Once the input point is inside a failure region, there are 5 different states it could be in. Two of these are trapping states: the "Fail Safe" state occurs when there is no majority of versions that agree on a correct answer; the "Catastrophic Failure" state occurs when a majority of versions agree on an incorrect answer. The remaining 3 states (which are not trapping states) occur when a plurality of versions (2, 3, or 4) agree on a correct answer.

The expressions for the transitions from the "No Failures" state are the most complex because they must divide a single transition rate, λ_{fr}, among five different destinations. The "Fail Safe" state receives all the triple and quadruple failures that have unique outputs as well as the double failures that have identical outputs. (The coefficients in the equation come from the fact that there are 3 combinations of versions that could produce triple failures, 6 combinations that could produce double failures, and one combination that can produce quadruple failures.) The "Catastrophic Failure" state receives all triple and quadruple failures where 2 or more incorrect outputs agree. (Again, the coefficients represent the three combinations of versions that could produce triple failures and the one combination that could produce a quadruple failure.) The "3:1 Split" state is only produced by a single failure, λ_1, and there are four possibilities, hence the coefficient of four. The "2:1:1 or 2:1 Split" state is produced by a double failure where the incorrect outputs are unique, λ_2, and there are six possible combinations of versions that could produce a double failure. Finally, any transition from the "No Failures" state that does not go to the previous four states will go to the "No Error Crystals" state. Thus, the transition rate is λ_{fr} times 1 minus the sum of all the other transition rates.

All transitions from the "No Error Crystals" state to any of the four failure region states follows the same pattern as above except the λ_{fr} term is no longer needed. This term

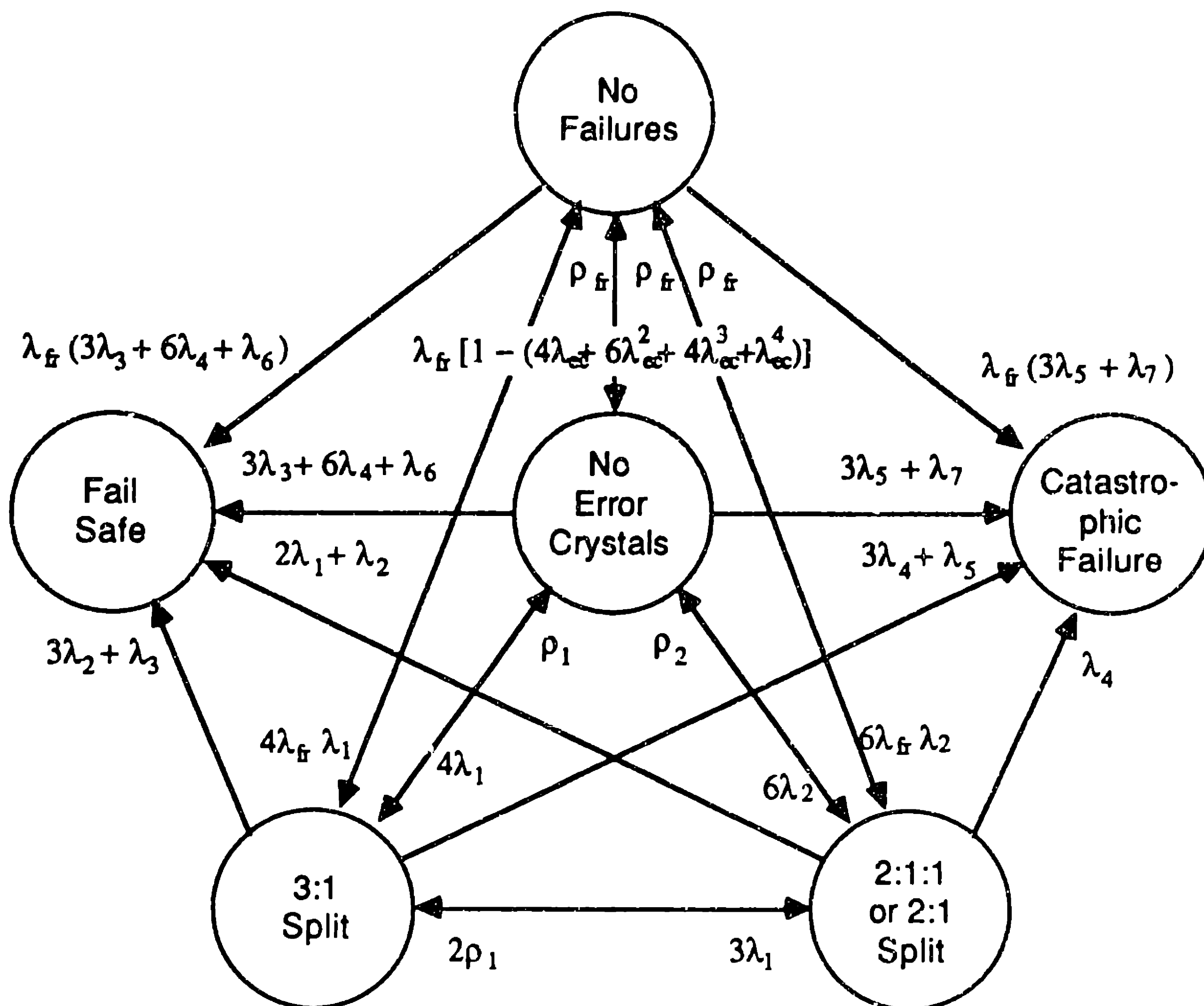
is no longer used because the input point is already inside a failure region. Transitions from the "No Error Crystals" state to the "No Failures" state occurs at the rate ρ_{fr}.

There are five possible transitions from the "3:1 Split" state. A single failure (rate: $3\lambda_1$) leads to the "2:1:1 or 2:1 Split" state. Both a double failure with unique outputs ($3\lambda_2$) and a triple failure with unique outputs (λ_3) lead to a fail safe, while a double failure with identical outputs ($3\lambda_4$) and a triple failure with two or more identical outputs (λ_5) lead to a catastrophic failure. The failed version could also exit its error crystal (rate ρ_{ec}) or the input point could exit the failure region (rate ρ_{fr}).

The state that represents double failures also has five possible transitions. A single failure ($2\lambda_1$) and a double failure with unique outputs (λ_2) both lead to a fail safe, while a double failure with identical outputs (λ_4) leads to a catastrophic failure. A single recovery ($2\rho_1$) leads to a 3:1 split and a double recovery (ρ_2) leads to the "No Error Crystals" state. Finally, the input point could exit the failure region (rate ρ_{fr}).

By investigating the underlying process that causes what we are trying to model, we are able to create a Markov model that reflects the software failure mechanism, which is significantly different from the hardware failure mechanism. While hardware failures are random, statistically independent events, software failures are not, and they should be modeled in a different fashion. This model treats software failures as deterministic functions of the software's input and it links failures between versions via common failure regions.

Another advantage to this model is that it uses a small number of unrelated parameters to define the transition rates between states. These parameters directly represent those quantities we are interested in: the size and density of failure regions, the size and density of error crystals, and the probability that two failed versions will choose an identical incorrect answer. This model also allows a wide range of parameters to be used. For example, we could represent the entire input space as a failure region by setting $\lambda_{fr} = 1$ and



Explanation of Symbols

$\lambda_1 = \lambda_{ec}$	Failure Rate for Software Modules
$\lambda_2 = \lambda_{ec}^2(1 - \pi^2)$	Dual Coincident Error Rate (Unique Outputs)
$\lambda_4 = \lambda_{ec}^2\pi^2$	Dual Coincident Error Rate (2 Identical Outputs)
$\lambda_3 = \lambda_{ec}^3(1 - 3\pi^2 + 2\pi^3)$	Triple Coincident Error Rate (Unique Outputs)
$\lambda_5 = \lambda_{ec}^3(3\pi^2 - 2\pi^3)$	Triple Coincident Error Rate (2 or 3 Identical Outputs)
$\lambda_6 = \lambda_{ec}^4(1 - 6\pi^2 + 8\pi^3 - 3\pi^4)$	Quadruple Coincident Error Rate (1:1:1:1 or 2:2 splits)
$\lambda_7 = \lambda_{ec}^4(6\pi^2 - 8\pi^3 + 3\pi^4)$	Quadruple Coincident Error Rate (>2 Identical Outputs)
$\rho_1 = \rho_{ec}$	Recovery Rate for Software Modules
$\rho_2 = \rho_{ec}^2$	Dual Coincident Recovery Rate

Figure 4-3: Markov Model for Multiple Software Failures

$\rho_{fr} = 0$. We could also treat all multiple failures as identical by setting $\pi = 1$. This flexibility, combined with the ability to directly control the important parameters are additional advantages of this model.

Chapter 5

Decision Algorithms

This chapter covers the development of the confidence voter. The confidence voter is a decision algorithm that takes advantage of the deterministic nature of multiple software failures to increase the reliability of N-version software. First the actual algorithm is described then the results of some simulations are discussed.

5.1 N-Version Software Reliability

From the information in the previous chapters, it is safe to state that the lack of independence between versions is the main impediment to highly reliable, N-version software. The thought, then, is can we use these dependencies to increase our reliability for N-version software? If so, what kind of impact will this have? For example, in figure 5-1 (which is not set to any scale), the area marked A is the subset of the input space where version A will fail. Area B is where version B will fail. Every time B fails, A will fail as well. If our voter could somehow take advantage of these deterministic software failures, we could improve the reliability of our system.

Unfortunately, there are several problems with this scheme. The first problem is we cannot assume that the probability (A fails **and** B fails) equals the probability (A fails) **times** the probability (B fails). In fact, using the latter figure implicitly assumes that the failures are independent. Thus, for our four version system, we need to keep track of the probability of each *pair* failing. Of course, this leads into the next problem: determining *when* a version has failed.

We can, of course use the 3:1 splits to determine single failures with a high degree of confidence, but that information really has a dubious utility. For double failures, we make

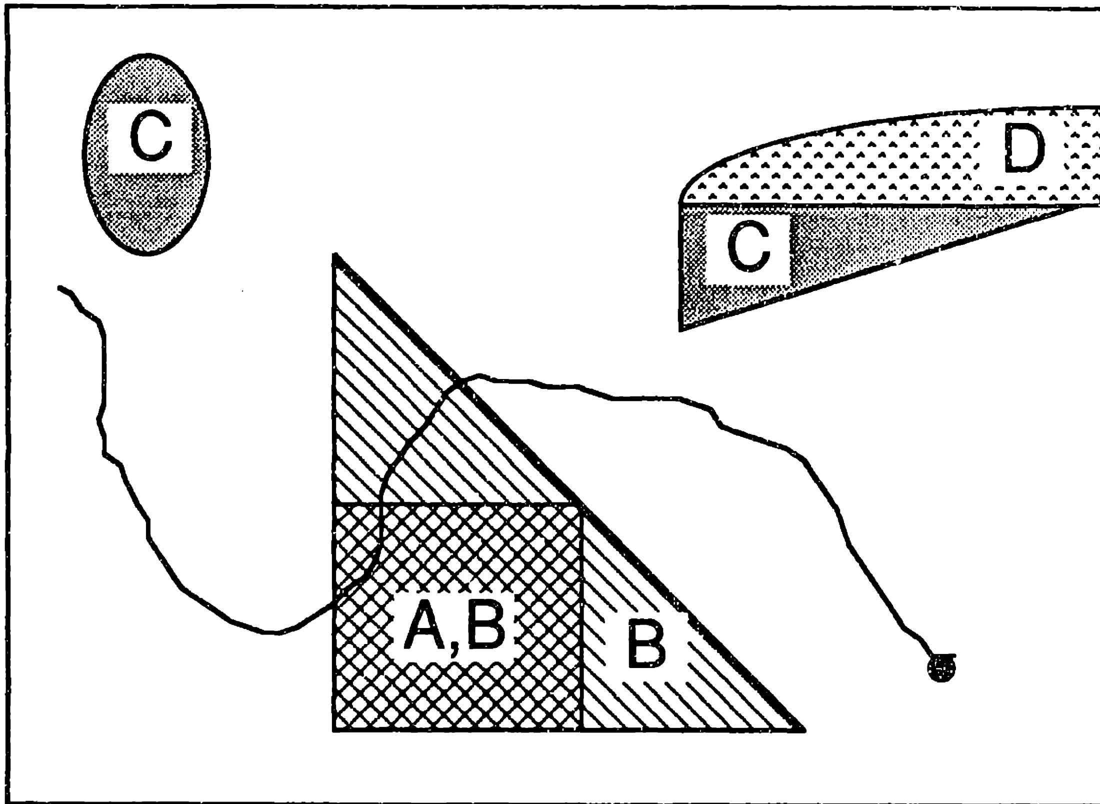


Figure 5-1: A 2-Dimensional Slice of an Input Space

the assumption that the probability that A and B fail with *identical* answers is proportional to the probability that A and B fail with *non-identical* answers. This allows us to use 2:1:1 splits (that is, iterations where the two incorrect values are not identical) to gather information on double failures. We then use this information to decide 2:2 splits (iterations where the two incorrect values are identical).[9]

By picking out which pair of versions fail together, we might be able to increase the

[9]If we do not make the above assumption and if we do not assume failures of different versions are independent, then this particular method cannot gather useful information on correlated failures. An alternative method of detecting failures would be to use a "gold version" that has been more rigorously tested. This version would then help determine when a pair of versions has failed.

reliability of N-version software. Now, however, we would like to know how the double failures are distributed. Does each pair have approximately the same number of double failures, or are there some pairs that are "bad" and fail together often?

5.2 Analysis of Knight's Data

The 27 version experiment done by Knight and Leveson [14] is one of the few N-version experiments done and by far the best known. We used this data to analyze the distribution of double failures. As a first step, we performed a simple analysis on the pair-wise failures in these versions and divided each of the 17550 possible 4-version combinations[10] into one of 5 categories:

1. Combinations with no double failures
2. Combinations with only one pair of versions that fail together
3. Combinations where several pairs of versions fail together, but one pair fails > 10 times more than any other pair
4. Combinations where several pairs of versions fail together, but one pair fails 2 to 10 times more than any other pair
5. All other combinations (difference < 2)

Figure 5-2 shows the distribution of the 4-version combinations among the 5 different cases. Notice that a quarter of the combinations had *no* double failures (Case 1), and a third had only one pair cause all the double failures (Case 2). The combinations with more than an order of magnitude difference in the number of failures caused by each pair also account for a large percentage (Case 3).

I expect that when only one pair causes most or all of the double failures, the confidence voter will be able to learn which pair is responsible and improve the reliability of the system. Even in the case where the frequency difference is between 2 and 10 (Case 4), I expect the confidence voter will be able to pick the pair that produces the majority of the double failures as long as the voter is not too conservative in deciding 2:2 splits.

[10]There are 17550 ways to choose 4 versions from a pool of 27

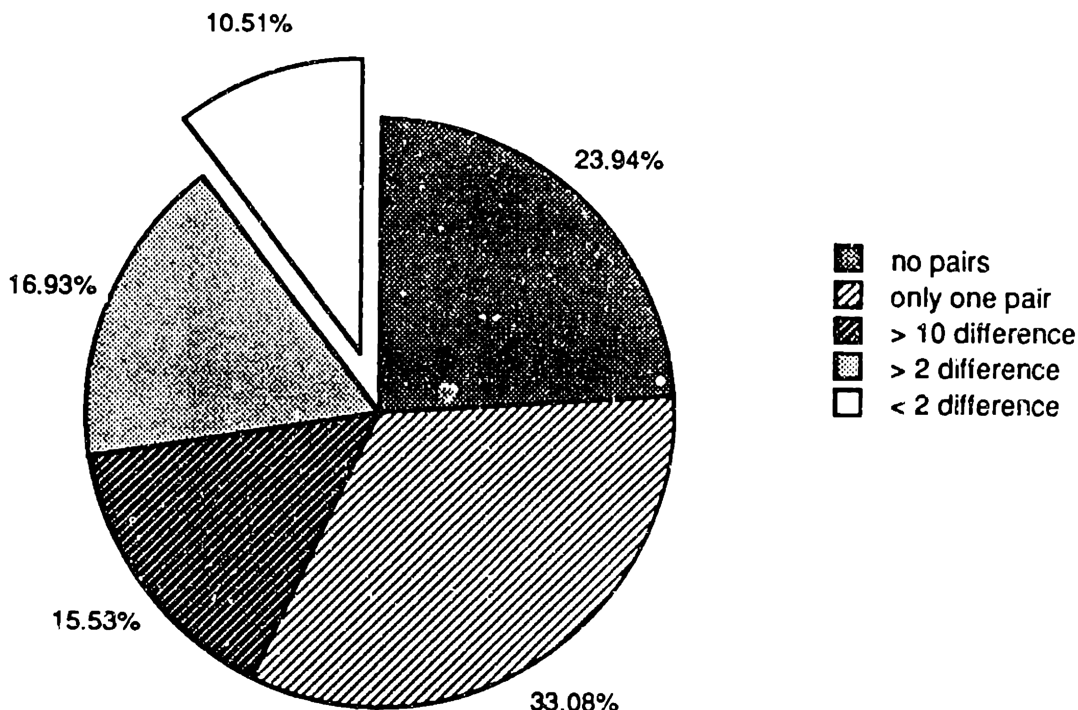


Figure 5-2: Types of 4-Version Combinations

5.3 The Confidence Voter

The actual operation of the confidence voter is fairly straight-forward. Each time a pair of versions fail, the failure counter for that pair is incremented. If there is a fail-safe, the counters for every pair are incremented. This way, the counters work under a worst-case assumption. When there is a 2:2 split, the confidence voter compares the failure rates for the two pairs. If the failure rate for one pair is some threshold larger than the failure rate of the other pair, then the more reliable pair is chosen. If the difference between the two pairs' failure rates is not large enough, the confidence voter will fail-safe.

In operation, then, the confidence voter will behave exactly like the majority (or "plurality") voter until it has gained enough knowledge to cross the threshold. This learning curve varies with the size of the threshold, the proportion of failures that are identical, and the difference in frequency of failures for the different pairs.

For example, if A and B fail together 100 time more often than C and D, then the voter should "learn" faster than if A and B only failed 5 times more often than C and D. Also, if the failures tend to have unique values, the rate that the confidence voter has to make decisions is lower, giving it more time to "learn."

More important, however, is the threshold. The threshold is the *only* parameter affecting the learning curve that can be controlled. But, before setting a threshold, one must first consider the application that the confidence voter will be used with. If, in a given application, a fail-safe is relatively harmless compared to a catastrophic failure, then the threshold should be set high. This causes the confidence voter to be conservative and not decide a 2:2 split until it has a large amount of evidence in favor of one pair. On the other hand, if a fail-safe is almost as bad as a catastrophic failure, then one would set a lower threshold, making the confidence voter less conservative.

An important caveat, however, is that there should always be some threshold. Without a threshold, it is possible for the decision process to influence the earliest stages of the learning process. This could push the confidence voter down the wrong path and ruin the system's reliability. If both a fail-safe and a catastrophic failure are equally bad, then there should be some mechanism used to prevent the voter's decisions from influencing the learning process too early.

The point is that care should be taken in setting the confidence voter's threshold. A careful analysis of the costs of a fail-safe versus the costs of a catastrophic failure needs to be done before any threshold can be substantiated.

5.4 Simulating the Confidence Voter's Operation

Now that the analysis of Knight's data has shown conditions that are favorable to the confidence voter,[11] we would like to simulate the confidence voter in operation. To do this, we created an N-version simulator which mimics the operation (and failure) of four versions of a program. The simulator is divided into two sections: a failure generator and a result generator. The failure generator determines which of the four versions will fail, and the result generator assigns both correct and incorrect floating point numbers as each version's results.

The failure patterns for the four versions are based on Knight's 27-version experiment [14]. Since there are four versions, and on each iteration each version can either fail or not fail, there are 16 possible failure patterns. Using Knight's data, each of the 16 failure patterns is assigned a probability. The simulator starts an "iteration" by randomly selecting a failure pattern. If there are no failures, each version is given an identical floating point number as its result. In the four patterns where only one version has failed, that version will be assigned a result that is different from the other three versions' results.

When two or more versions fail, then the result generator must decide whether the failures are unique or identical. To do this, the result generator uses the parameter π, which is set to some value. (A more complete description of π is in Chapter 4.) The probability that two failed versions have the same incorrect value is the probability that both versions choose the popular incorrect answer (this probability is π^2). If either version (or both) choose an unpopular incorrect answer, then the two answers will be different (this happens

---

[11] Again, we are working under the assumption that the number of double failures with non-identical results is proportional to the number of double failures with identical results. Unfortunately, Knight has not published a complete set of data from his experiment and therefore, we have no way to check the validity of this assumption.

with probability $1 - \pi^2$). After everything is computed, four floating point numbers, one from each version, are given to the voting routines.

5.5 Results of the Simulations

To test how the confidence voter would perform against the majority (or "plurality") voter, one combination of four versions was randomly picked from each of the four types of combinations that experienced double failures. Each of these cases was run through a million simulated iterations two times. For the first million iterations π was equal to 0.5, for the second millions iterations π was equal to 0.7.

Figure 5-3 shows the pattern of failures used for the first simulations. The numbers above each column refer to the version numbers assigned by Knight [14]. The X's in each box indicate that that version is failed for the given failure pattern. The numbers in the first column refer to the number of times each failure pattern should occur during a million iterations. Each of these first two combinations has only one pair of versions that create all the double failures. (There are two combinations for this case because the original combination of versions chosen, case 2a, had a low number of double failures.)

During these simulations, there were no catastrophic failures. This is reasonable because both voters will fail catastrophicly when there are three or four multiple failures, and the confidence voter will also fail catastrophicly when it makes a bad decision on a 2:2 split. Since there were no possible triple or quadruple failures in these patterns, and, for these simulations, the confidence voter never made a bad decision, there were no catastrophic failures. Figure 5-4 shows the number of fail safes that occurred during the first two simulations for both case 2a (versions 2, 9, 11, and 23) and case 2b (versions 2, 9, 11, and 8). The numbers in parentheses after the case refers to the value for π for that simulation (either 0.5 or 0.7). In all simulations, the confidence voter shows an obvious improvement over the majority voter.

Combination Used for the First Case 2 Simulation

	2	9	11	23
0				
53				
545				
71				
0				
0				
0				
0				
0				
9				
0				
0				
0				
0				
0				
999322				

Combination Used for Second Case 2 Simulation

	2	9	11	8
0				
53				
549				
285				
0				
0				
0				
0				
0				
58				
0				
0				
0				
0				
0				
999075				

Figure 5-3: Combinations With Only One "Bad" Pair

Another area of interest is how the performance of the confidence voter changes with time. Figure 5-5 shows how the total number of fail safes changes as the number of iterations increases. Note that somewhere between iteration 25,000 and iteration 375,000 the confidence voter crossed the threshold and "decided" that it had enough information to choose one pair in a 2:2 split. From that point on, the confidence voter had no more fail safes because it had marked which pair causes all the double failures.

Figure 5-6 shows the values of the confidence voter's counters versus the actual values. In this figure, the number in row x, column x refers to the total number of time version x has failed, while row x, column y refers to the number of times version x and version y failed together. The confidence voter has a very accurate idea of which versions had failed. The only real discrepancies are due to the first fail safe that the confidence voter made before it had crossed the threshold. As further iterations are simulated, the counter values would only become more accurate.

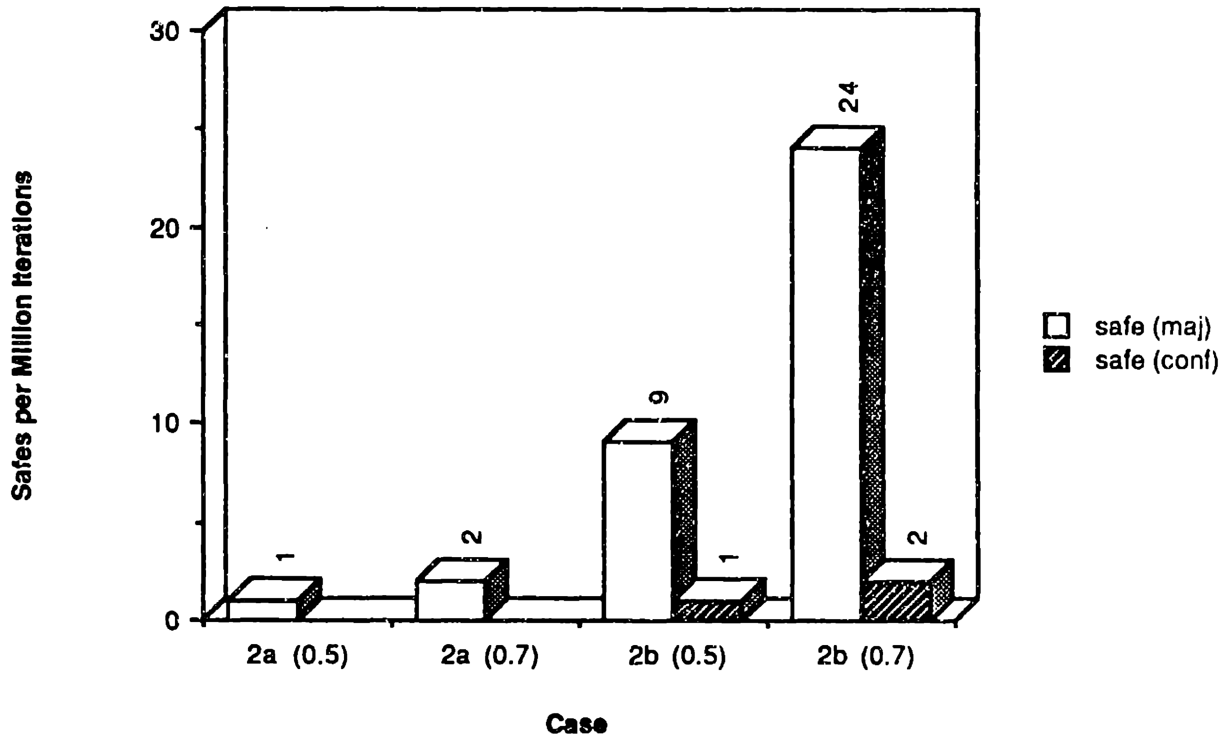


Figure 5-4: Results of the First Simulations

Figure 5-7 shows the pattern of failures used for the second two simulations. These combinations had several pairs of versions cause double failures, but one pair of versions was responsible for many more of the double failures than any other pair. The case 3 combination chosen (which must have at least a factor of 10 difference between the number of double failures produced by the two most failure prone pairs) used versions 1, 14, 19, and 20. There was, in fact, a factor of 33 difference. The case 4 combination chosen (which must have at least a factor of 2 difference) used versions 16, 19, 25, and 26. For this combination, there was a factor of 4 difference.

During these simulations, there was only one catastrophic failure. That was due to

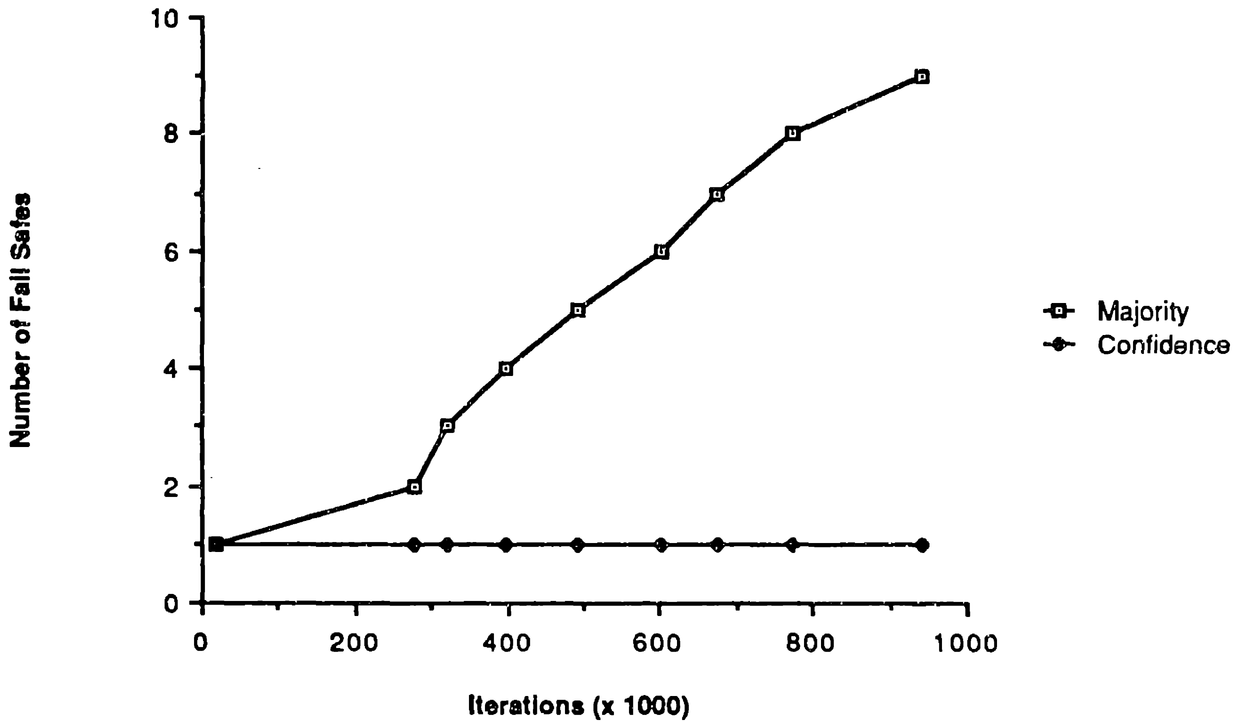


Figure 5-5: Learning Curve For the Confidence Voter

Confidence Voter

	1	2	3	4
1:	1	1	1	1
2:		43	1	1
3:			509	55
4:				292

Actual Values

	1	2	3	4
1:	0	0	0	0
2:		42	0	0
3:			509	55
4:				292

Figure 5-6: Counter Values for the First Simulations

the one triple failure in the pattern of failures for case 3. Figure 5-8 shows the number of fail safes that occurred during the simulations for the case 3 and the case 4 combinations. In both of these cases, the confidence voter still shows an improvement over the majority voter. In fact, as the difference between failure rates for the worst pairs in a combination approaches the threshold of the confidence voter, I would expect two things to happen.

Combination Used for Case 3 Simulation

	1	14	19	20
2				
1334				
262				
901				
0				
0				
0				
0				
33				
1				
0				
0				
0				
1				
0				
997466				

Combination Used for Case 4 Simulation

	16	19	25	26
58				
263				
75				
855				
0				
0				
5				
0				
1				
22				
0				
0				
0				
0				
0				
998721				

Figure 5-7: Combinations With Several "Bad" Pairs (Difference > 2)

First, the length of the learning curve will increase. Second, after the threshold is crossed, the confidence voter will experience more fail safes due to the other bad pairs failing.

Figure 5-10 shows the pattern of failures used for the last simulation. The last combination of versions used had a small difference (a factor of 1.5 in this case), and had double failures from 3 different pairs. This combination included versions 16, 21, 22, and 25.

During these simulations there were a number of catastrophic failures and fail safes. Because the confidence voter never passed its threshold, it continued to behave like the majority voter. The results in figure 5-11 show that there was absolutely no difference between the two voters. These results, again, are exactly what we would expect for a case where the difference between pairs of versions (in terms of frequency of double failures) is less than the confidence voter's threshold.

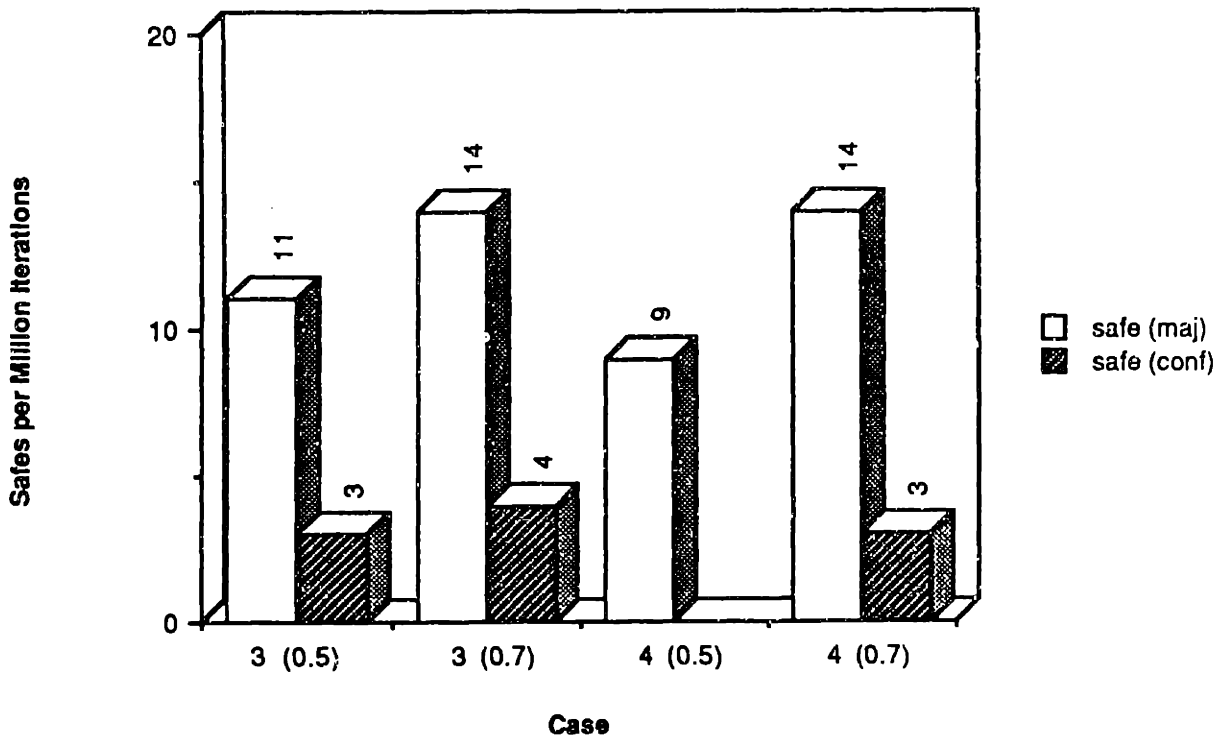


Figure 5-8: Results of the Second Two Simulations

The fact that the confidence voter never crossed its threshold is reflected in the values of the confidence voter's counters. Figure 5-12 shows that the counters for each pair of versions have values greater than the actual number of double failures. This is because the confidence voter always failed safe on 2:2 splits and incremented the counters for every pair.

The results of these simulations show that the confidence voter can improve a 4-version system's reliability. We cannot say that this will always be the case. However, if we believe that Knight's data is somewhat indicative of multiple version software in general, then we can expect that the confidence voter will improve the reliability in a large

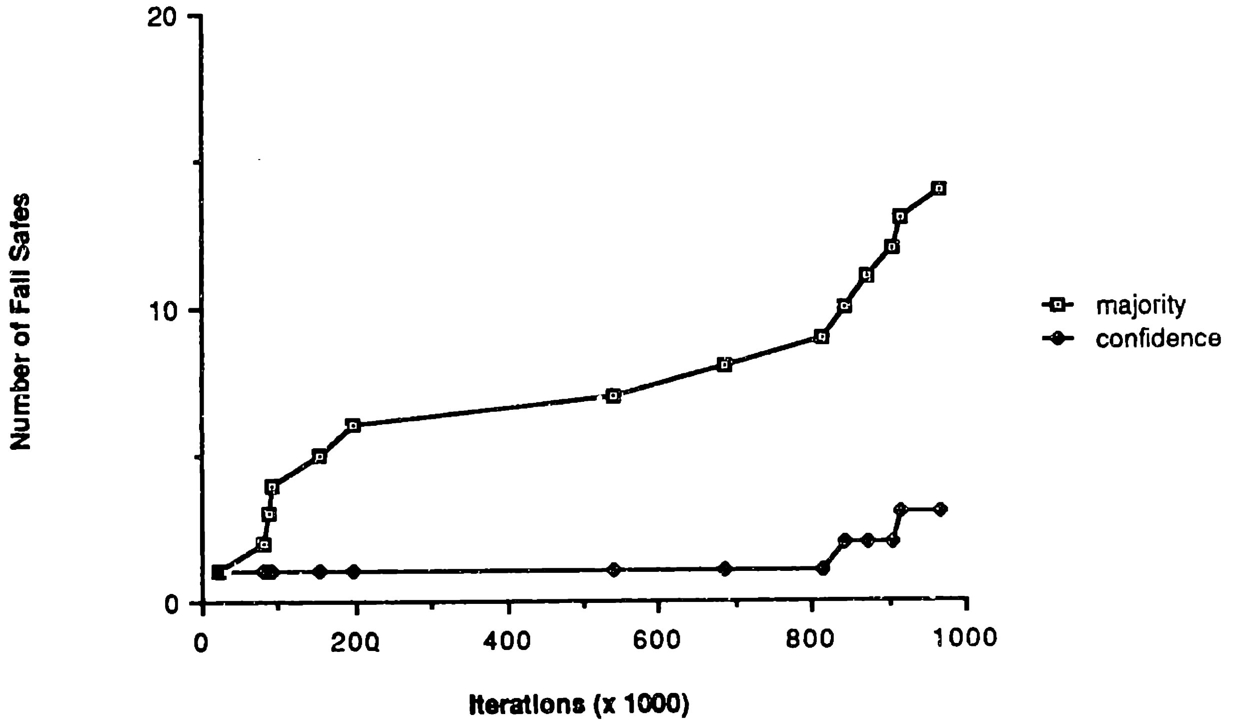


Figure 5-9: Learning Curve For the Confidence Voter (Case 4)

number of cases. Our hope then, is that in the cases where the confidence voter will not improve reliability, then it will not decrease reliability. This hope, however, relies on the assumption that the number of 2:1:1 splits a pair of versions have tells us how many 2:2 splits to expect.

Combination Used for Case 5 Simulation

	16	21	22	25
50				
84				
9642				
89				
6				
4				
0				
0				
0				
8				
1				
0				
0				
0				
0				
990116				

Figure 5-10: Combination With Several "Bad" Pairs (Difference < 2)

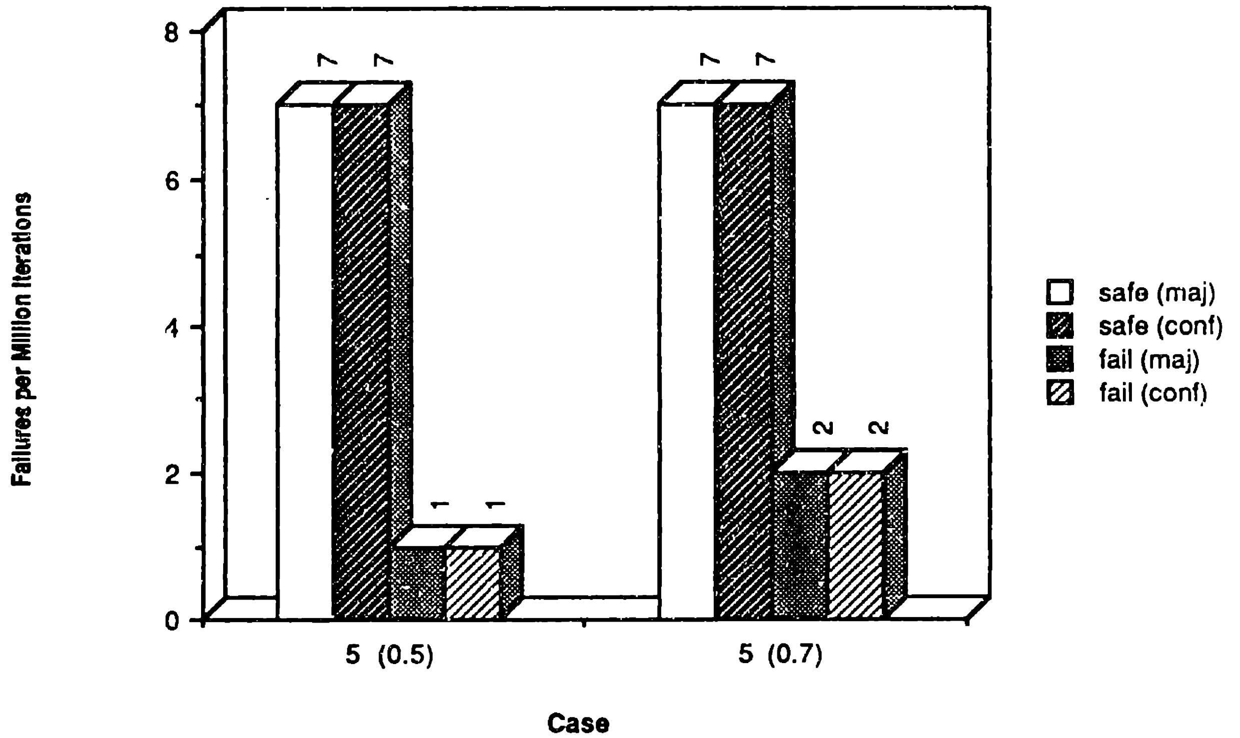


Figure 5-11: Results of the Last Simulations

Confidence Voter	1	2	3	4	Actual Values	1	2	3	4
1:	57	9	8	9	1:	53	4	6	0
2:		83	7	7	2:		78	2	0
3:			9007	13	3:			9009	10
4:				107	4:				102

Figure 5-12: Counter Values for the Last Simulation

Chapter 6

Conclusions

It is clear that the failure mechanism for software is fundamentally different from the failure mechanism for hardware. If the results of current research are correct, then the lack of independence among multiple versions of software severely reduces the potential reliability of N-version software.

In this thesis, we have presented some new ideas on the subject of N-version software. First, the Markov model presented in chapter 4 models failures of multiple versions of software. This model is not the result of "converting" a model for hardware failures to a model for software failures. It was created, instead, by investigating the underlying mechanism for software failures. Second, the confidence voter presented in Chapter 5 has shown, that it is possible, given certain assumptions, to use the deterministic nature of software failures to increase the reliability of an N-version system. The reliability of N-version software suffers from the cases where version A always fails when version B fails. It is this information that the confidence voter uses to increase reliability.

In some systems, the confidence voter has the potential to decrease failure rates by several orders of magnitude. For example, the simulation where only two versions exhibited correlated failures is a prime candidate. As long as our assumptions are met, the confidence voter will be able (after a short learning period) to easily determine which pair is correct in a 2:2 split. Since these failures are the only correlated failures, all others would be single failures, which can already be tolerated. In this system, the confidence voter decreases the N-version system's failure rate by several orders of magnitude.

Even in the simulation where several versions were responsible for the correlated failures, and there was no significant difference in the number of failures caused by each

pair, the confidence voter defaults to the same choices that the majority (or "plurality") voter would have made. There are no more failures in this system, using the confidence voter, than there would be if the system were using the majority voter.

As a decision algorithm for N-version software, the confidence voter clearly has the potential to increase the reliability of these systems. However, we currently do not know enough about software failures to be able to conclude how well the confidence voter will perform in an operational environment. The assumption that the number of identical failures for a pair of versions is proportional to the number of non-identical failures is critical to our analysis of the voter's performance.

Take, for example, a system where one pair of versions have correlated failures that *always* produce identical answers, while another pair *always* produces non-identical answers. In this system, the confidence voter would actually decrease reliability because it would end up picking the wrong pair in a 2:2 split. If we cannot assume that the number of identical failures is proportional to the number of unique failures then a confidence voter with no *a priori* knowledge of failure rates would have to use some other method to gather the information critical to its performance. These methods could use data from the testing phase of software development or perhaps one version could be more extensively tested and used as a gold version.

It is easy to see that the confidence voter's performance is inexorably linked to *how* multiple versions of software fail. Thus, before we draw any definite conclusions about the voter's performance in an operational N-version systems, we must first learn about the interactions of failures in multiple versions of software.

Recommended future work would certainly include experiments to give us more information on how multiple versions of software fail. This information would allow us to test the assumptions that the confidence voter was based upon. Several aspects of the confidence voter itself also need to be investigated further. Little is known about the

factors that influence the length of the voter's learning curve. Also, the voter itself may be used to prevent other types of failures. For example, if a catastrophic failure was much more expensive than a fail-safe, one might want to use the confidence voter's counters to decide when to fail-safe on a 2:1:1 split. This would reduce the number of triple failures that cause catastrophic errors.

References

[1] Linda S. Alger and Gregory L. Greeley.
A Highly Reliable Architecture for Executing N-Version Software.
In *Proceedings of the Eighteenth Joint Services Data Exchange for Inertial Systems*. Joint Services Data Exchange, 1986.

[2] Linda S. Alger and Jaynarayan H. Lala.
A Real Time Operating System for a Nuclear Power Plant Computer.
In *Proceedings of the IEEE Computer Society Real-Time Systems Symposium*. IEEE, 1986.

[3] T. Anderson and P. A. Lee.
Fault Tolerance: Principles and Practice.
Prentice Hall International, 1981.

[4] Algirdas Avizienis and John P. J. Kelly.
Fault Tolerance by Design Diversity: Concepts and Experiments.
IEEE Computer 17(8), August, 1984.
Special issue on fault-tolerant computing.

[5] Algirdas Avizienis.
The N-Version Approach to Fault-Tolerant Software.
IEEE Transactions on Software Engineering SE-11(12), December, 1985.

[6] L. Chen and A. Avizienis.
N-Version Programming: A Fault Tolerant Approach to Reliability of Software Operation.
In *Digest of Papers FTCS-8: The 8th Annual International Conference on Fault Tolerant Computing*. IEEE, 1978.

[7] J. R. Dunham and J.L. Pierce.
An Experiment in Software Reliability.
Contractor Report 172553, National Aeronautics and Space Administration, 1985.

[8] Dave Eckhardt Jr. and Larry D. Lee.
An Analysis of the Effects of Coincident Errors on Multi-Version Software.
In *Computers in Aerospace V Conference*. AIAA, October, 1985.

[9] Dave E. Eckhardt Jr. and Larry D. Lee.
A theoretical Basis for the Analysis of Redundant Software Subject to Coincident Errors.
Technical Memorandum 86369, National Aeronautics and Space Administration, January, 1985.

[10] Gregory L. Greeley.
An Ada Implementation for Fault Detection, Isolation and Reconfiguration Using a Fault-Tolerant Processor.
In *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. NASA, 1986.

[11] Andy D. Hills.
A310 Slat and Flap Control System Management & Experience.
Technical Report, Flight Controls Division, Marconi Avionics Ltd, 1984.

[12] A. L. Hopkins Jr., J. H. Lala, and T. B. Smith III.
The Evolution of Fault Tolerant Computing at the Charles Stark Draper Laboratory, 1955-1985.
In *Proceedings of the Symposium on the Evolution of Fault Tolerant Computing.* Springer Verlag, Baden, Austria, June, 1986.

[13] John P.J. Kelly.
Specification of Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach.
PhD thesis, University of California, Los Angeles, 1982.

[14] John Knight, Nancy Leveson, and Louis D. St. Jean.
A Large-Scale Experiment in N-Version Programming.
In *Digest of Papers FTCS-15: The 15th Annual International Conference on Fault Tolerant Computing.* IEEE, 1985.

[15] John Knight and Paul Ammann.
An Experimental Evaluation of Simple Methods for Seeding Program Errors.
IEEE Transactions on Reliability SE-11(12), December, 1985.

[16] John Knight and Nancy Leveson.
An Empirical Study of Failure Probabilities in Multi-Version Software.
In *Digest of Papers FTCS-16: The 16th Annual International Conference on Fault Tolerant Computing.* IEEE, 1986.

[17] J. C. Knight.
Detection of Faults and Software Reliability Analysis.
Annual Progress Report UVA/528243/CS87/101, University of Virginia, School of Engineering and Applied Science, Department of Computer Science, August, 1986.

[18] Jaynarayan H. Lala.
A Byzantine Resilient Fault Tolerant Cpmputer for Nuclear Power Plant Applications.
In *Digest of Papers FTCS-16: The 16th Annual International Conference on Fault Tolerant Computing.* IEEE, 1986.

[19] Leslie Lamport, *et al.*
The Byzantine Generals Problem.
ACM Transactions on Programming Languages and Systems 4(3):382-401, July, 1982.

[20] M. Lipow.
Prediction of Software Failures.
The Journal of Systems and Software 1:71-75, 1979.

[21] Gerald E. Migneault.
personal communication.

[22] P. M. Nagle and J. A. Skrivan.
Software Reliability: Repetitive Run Experimantation and Modeling.
Contractor Report, National Aeronautics and Space Administration, 1982.

[23] Louis Diane St. Jean.
Testing Version Independence in Multi-Version Programming.
Master's thesis, University of Virginia, January, 1985.

[24] T. Basil Smith III.
Generic Data Manipulative Primitives of Synchronous Fault-Tolerant Computer Systems.
Technical Report, Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts, 1980.

[25] Kenneth J. Szalai, *et al.*
Digital Fly-by-Wire Flight Control Validation Experience.
Technical Memorandum 72860, National Aeronautics and Space Administration, December, 1978.

[26] Walker, Bruce, *et a'.*
Fault Tolerant Control Systems.
December, 1984.
future textbook for MIT course 16.321, Fault Tolerant Control Systems.

[27] Larry James Yount.
Architectural Solutions to Safety Problems of Digital Flight Critical Systems for Commercial Transports.
In *Proceedings of the AIAA/IEEE Digital Avionics Systems Conference and Technical Display.* IEEE/AIAA, 1984.