Johnson, B.W. "Fault Tolerance"
The Electrical Engineering Handbook
Ed. Richard C. Dorf
Boca Raton: CRC Press LLC, 2000

93

Fault Tolerance

Barry W. Johnson
University of Virginia

93.1 Introduction

Fault tolerance is the ability of a system to continue correct performance of its tasks after the occurrence of hardware or software faults. A **fault** is simply any physical defect, imperfection, or flaw that occurs in hardware or software. Applications of fault-tolerant computing can be categorized broadly into four primary areas: long-life, critical computations, maintenance postponement, and high availability. The most common examples of long-life applications are unmanned space flight and satellites. Examples of critical-computation applications include aircraft flight control systems, military systems, and certain types of industrial controllers. Maintenance postponement applications appear most frequently when maintenance operations are extremely costly, inconvenient, or difficult to perform. Remote processing stations and certain space applications are good examples. Banking and other time-shared systems are good examples of high-availability applications. Fault tolerance can be achieved in systems by incorporating various forms of redundancy, including hardware, information, time, and software redundancy [Johnson, 1989].

93.2 Hardware Redundancy

The physical replication of hardware is perhaps the most common form of fault tolerance used in systems. As semiconductor components have become smaller and less expensive, the concept of hardware redundancy has become more common and more practical. There are three basic forms of hardware redundancy. First, *passive* techniques use the concept of fault masking to hide the occurrence of faults and prevent the faults from resulting in **errors**. Passive approaches are designed to achieve fault tolerance without requiring any action on the part of the system or an operator. Passive techniques, in their most basic form, do not provide for the detection of faults but simply mask the faults. An example of a passive approach is triple modular redundancy (TMR), which is illustrated in Fig. 93.1. In the TMR system three identical units perform identical functions, and a majority vote is performed on the output.

The second form of hardware redundancy is the *active* approach, which is sometimes called the *dynamic* method. Active methods achieve fault tolerance by detecting the existence of faults and performing some action to remove the faulty hardware from the system. In other words, active techniques require that the system perform reconfiguration to tolerate faults. Active hardware redundancy uses fault detection, fault location, and fault recovery in an attempt to achieve fault tolerance. An example of an active approach to hardware redundancy is standby sparing, which is illustrated in Fig. 93.2. In standby sparing one or more units operate as spares and replace the primary unit when it fails.

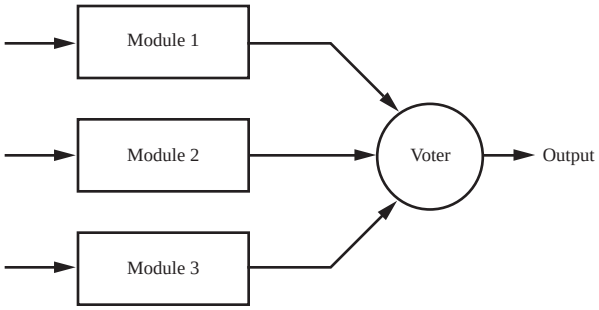

FIGURE 93.1 Fault masking using triple modular redundancy (TMR).

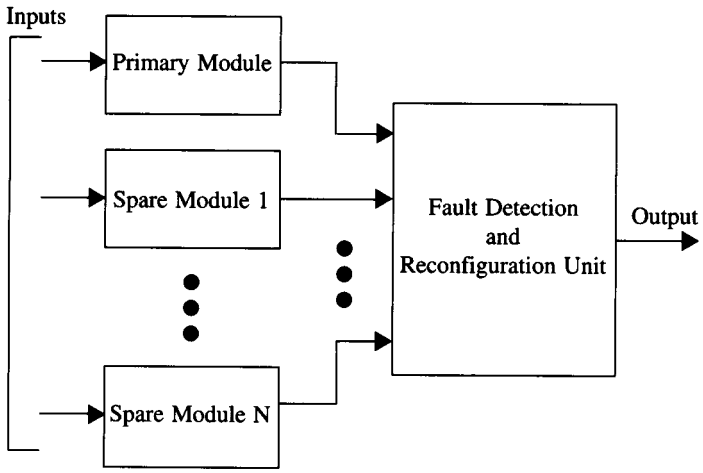

FIGURE 93.2 General concept of standby sparing.

The final form of hardware redundancy is the *hybrid* approach. Hybrid techniques combine the attractive features of both the passive and active approaches. Fault masking is used in hybrid systems to prevent erroneous results from being generated. Fault detection, fault location, and fault recovery are also used in the hybrid approaches to improve fault tolerance by removing faulty hardware and replacing it with spares. Providing spares is one form of providing redundancy in a system. Hybrid methods are most often used in the critical-computation applications where fault masking is required to prevent momentary errors, and high reliability must be achieved. The basic concept of the hybrid approach is illustrated in Fig. 93.3.

93.3 Information Redundancy

Another approach to fault tolerance is to employ redundancy of information. Information redundancy is simply the addition of redundant information to data to allow fault detection, fault masking, or possibly fault tolerance. Good examples of information redundancy are error detecting and error correcting codes, formed by the addition of redundant information to data words or by the mapping of data words into new representations containing redundant information [Lin and Costello, 1983].

In general, a *code* is a means of representing information, or data, using a well-defined set of rules. A *code word* is a collection of symbols, often called digits if the symbols are numbers, used to represent a particular piece of data based upon a specified code. A *binary code* is one in which the symbols forming each code word consist of only the digits 0 and 1. A code word is said to be *valid* if the code word adheres to all of the rules that define the code; otherwise, the code word is said to be *invalid.*

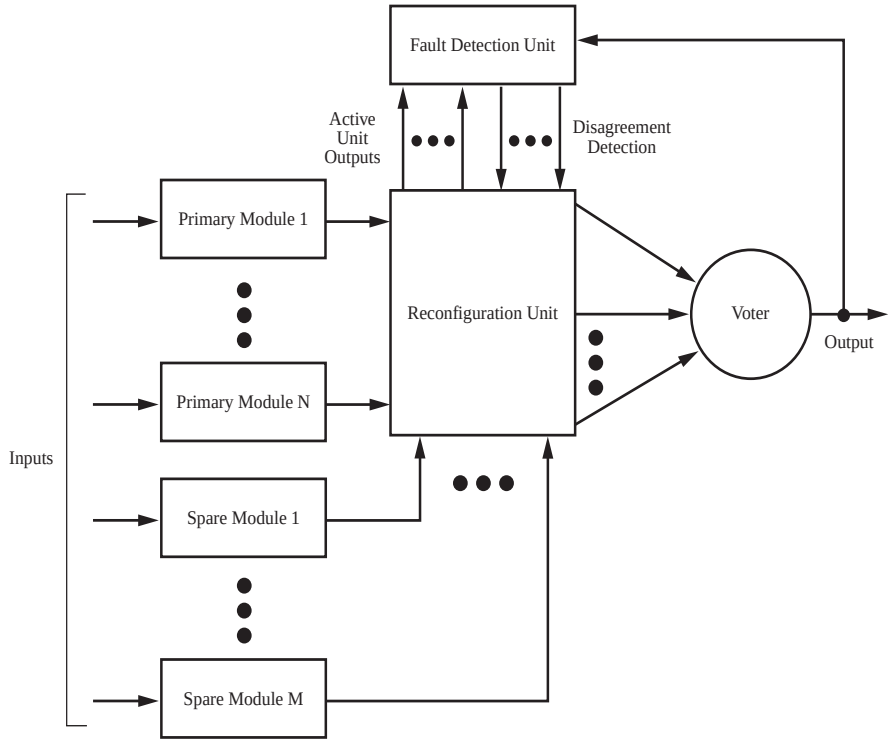

FIGURE 93.3 Hybrid redundancy approach.

The *encoding operation* is the process of determining the corresponding code word for a particular data item. In other words, the encoding process takes an original data item and represents it as a code word using the rules of the code. The *decoding operation* is the process of recovering the original data from the code word. In other words, the decoding process takes a code word and determines the data that it represents.

It is possible to create a binary code for which the valid code words are a subset of the total number of possible combinations of 1s and 0s. If the code words are formed correctly, errors introduced into a code word will force it to lie in the range of illegal, or invalid, code words, and the error can be detected. This is the basic concept of the *error detecting codes.* The basic concept of the *error correcting code* is that the code word is structured such that it is possible to determine the correct code word from the corrupted, or erroneous, code word.

A fundamental concept in the characterization of codes is the *Hamming distance* [Hamming, 1950]. The *Hamming distance* between any two binary words is the number of bit positions in which the two words differ. For example, the binary words 0000 and 0001 differ in only one position and therefore have a Hamming distance of 1. The binary words 0000 and 0101, however, differ in two positions; consequently, their Hamming distance is 2. Clearly, if two words have a Hamming distance of 1, it is possible to change one word into the other simply by modifying one bit in one of the words. If, however, two words differ in two bit positions, it is impossible to transform one word into the other by changing one bit in one of the words.

The Hamming distance gives insight into the requirements of error detecting codes and error correcting codes. We define the *distance* of a code as the minimum Hamming distance between any two valid code words. If a binary code has a distance of two, then any single-bit error introduced into a code word will result in the erroneous word being an invalid code word because all valid code words differ in at least two bit positions. If a code has a distance of 3, then any single-bit error or any double-bit error will result in the erroneous word being an invalid code word because all valid code words differ in at least three positions. However, a code distance of 3 allows any single-bit error to be corrected, if it is desired to do so, because the erroneous word with a single-bit error will be a Hamming distance of 1 from the correct code word and at least a Hamming

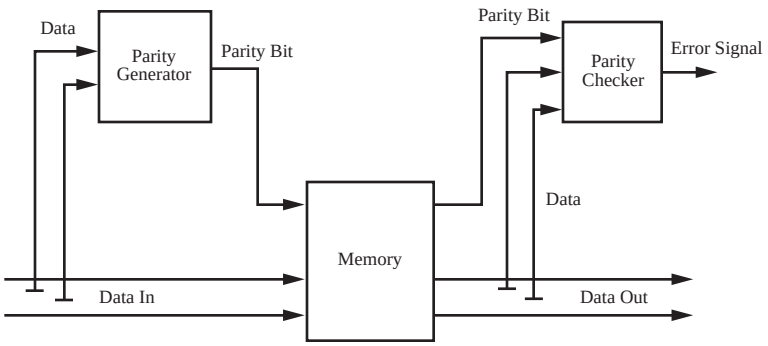

FIGURE 93.4 Use of parity coding in a memory application.

distance of 2 from all others. Consequently, the correct code word can be identified from the corrupted code word.

In general, a binary code can correct up to c bit errors and detect an additional d bit errors if and only if

$$2c - d - 1 \leq H_d$$

where H_d is the distance of the code [Nelson and Carroll, 1986]. For example, a code with a distance of 2 cannot provide any error correction but can detect single-bit errors. Similarly, a code with a distance of 3 can correct single-bit errors or detect a double-bit error.

A second fundamental concept of codes is *separability.* A *separable code* is one in which the original information is appended with new information to form the code word, thus allowing the decoding process to consist of simply removing the additional information and keeping the original data. In other words, the original data is obtained from the code word by stripping away extra bits, called the code bits or check bits, and retaining only those associated with the original information. A *nonseparable code* does not possess the property of separability and, consequently, requires more complicated decoding procedures.

Perhaps the simplest form of a code is the parity code. The basic concept of parity is very straightforward, but there are variations on the fundamental idea. Single-bit parity codes require the addition of an extra bit to a binary word such that the resulting code word has either an even number of 1s or an odd number of 1s. If the extra bit results in the total number of 1s in the code word being odd, the code is referred to as *odd parity.* If the resulting number of 1s in the code word is even, the code is called *even parity.* If a code word with odd parity experiences a change in one of its bits, the parity will become even. Likewise, if a code word with even parity encounters a single-bit change, the parity will become odd. Consequently, a single-bit error can be detected by checking the number of ls in the code words. The single-bit parity code (either odd or even) has a distance of 2, therefore allowing any single-bit error to be detected but not corrected. Figure 93.4 illustrates the use of parity coding in a simple memory application.

Arithmetic codes are very useful when it is desired to check arithmetic operations such as addition, multiplication, and division [Avizienis, 1971]. The basic concept is the same as all coding techniques. The data presented to the arithmetic operation is encoded before the operations are performed. After completing the arithmetic operations, the resulting code words are checked to make sure that they are valid code words. If the resulting code words are not valid, an error condition is signaled. An arithmetic code must be invariant to a set of arithmetic operations. An arithmetic code, A, has the property that $A(b^*c) = A(b)^*A(c)$, where b and c are operands, * is some arithmetic operation, and $A(b)$ and $A(c)$ are the arithmetic code words for the operands b and c, respectively. Stated verbally, the performance of the arithmetic operation on two arithmetic code words will produce the arithmetic code word of the result of the arithmetic operation. To completely define an arithmetic code, the method of encoding and the arithmetic operations for which the code is invariant must be specified. The most common examples of arithmetic codes are the *AN* codes, residue codes, and the inverse residue codes.

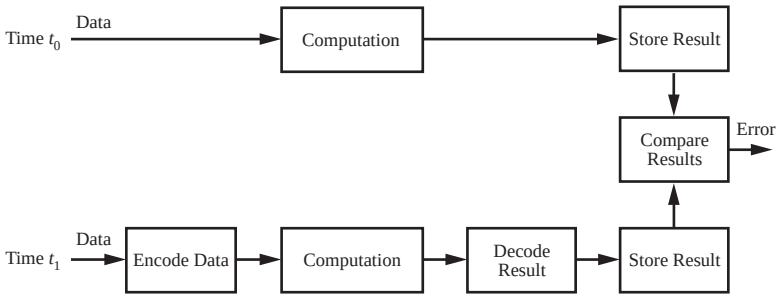

FIGURE 93.5 Time redundancy concept.

93.4 Time Redundancy

Time redundancy methods attempt to reduce the amount of extra hardware at the expense of using additional time. In many applications, the time is of much less importance than the hardware because hardware is a physical entity that impacts weight, size, power consumption, and cost. Time, on the other hand, may be readily available in some applications. The basic concept of time redundancy is the repetition of computations in ways that allow faults to be detected. Time redundancy can function in a system in several ways. The fundamental concept is to perform the same computation two or more times and compare the results to determine if a discrepancy exists. If an error is detected, the computations can be performed again to see if the disagreement remains or disappears. Such approaches are often good for detecting errors resulting from transient faults but cannot provide protection against errors resulting from permanent faults.

The main problem with many time redundancy techniques is assuring that the system has the same data to manipulate each time it redundantly performs a computation. If a transient fault has occurred, a system's data may be completely corrupted, making it difficult to repeat a given computation. Time redundancy has been used primarily to detect transients in systems. One of the biggest potentials of time redundancy, however, now appears to be the ability to detect permanent faults while using a minimum of extra hardware. The fundamental concept is illustrated in Fig. 93.5. During the first computation or transmission, the operands are used as presented and the results are stored in a register. Prior to the second computation or transmission, the operands are encoded in some fashion using an encoding function. After the operations have been performed on the encoded data, the results are then decoded and compared to those obtained during the first operation. The selection of the encoding function is made so as to allow faults in the hardware to be detected. Example encoding functions might include the complementation operator and an arithmetic shift.

93.5 Software Redundancy

Software faults are unusual entities. Software does not break as hardware does, but instead software faults are the result of incorrect software designs or coding mistakes. Therefore, any technique that detects faults in software must detect design flaws. A simple duplication and comparison procedure will not detect software faults if the duplicated software modules are identical, because the design mistakes will appear in both modules.

The concept of N self-checking programming is to first write N unique versions of the program and to develop a set of acceptance tests for each version. The acceptance tests are essentially checks performed on the results produced by the program and may be created using consistency checks and capability checks, for example. Selection logic, which may be a program itself, chooses the results from one of the programs that passes the acceptance tests. This approach is analogous to the hardware technique known as hot standby sparing. Since each program is running simultaneously, the reconfiguration process can be very fast. Provided that the software faults in each version of the program are independent and the faults are detected as they occur by the acceptance tests, then this approach can tolerate $N - 1$ faults. It is important to note that the assumptions of fault independence and perfect fault coverage are very big assumptions to make in almost all applications.

The concept of *N*-version programming was developed to allow certain design flaws in software modules to be tolerated [Chen and Avizienis, 1978]. The basic concept of *N*-version programming is to design and code the software module *N* times and to vote on the *N* results produced by these modules. Each of the *N* modules is designed and coded by a separate group of programmers. Each group designs the software from the same set of specifications such that each of the *N* modules performs the same function. However, it is hoped that by performing the *N* designs independently, the same mistakes will not be made by the different groups. Therefore, when a fault occurs, the fault will either not occur in all modules or it will occur differently in each module, so that the results generated by the modules will differ. Assuming that the faults are independent the approach can tolerate $(N-1)/2$ faults where N is odd.

The recovery block approach to software fault tolerance is analogous to the active approaches to hardware fault tolerance, specifically the cold standby sparing approach. *N* versions of a program are provided, and a single set of acceptance tests is used. One version of the program is designated as the primary version, and the remaining $N-1$ versions are designated as spares, or secondary versions. The primary version of the software is always used unless it fails to pass the acceptance tests. If the acceptance tests are failed by the primary version, then the first secondary version is tried. This process continues until one version passes the acceptance tests or the system fails because none of the versions can pass the tests.

93.6 Dependability Evaluation

Dependability is defined as the quality of service provided by a system [Laprie, 1985]. Perhaps the most important measures of dependability are reliability and availability. Fundamental to reliability calculations is the concept of failure rate. Intuitively, the *failure rate is* the expected number of **failures** of a type of device or system per a given time period [Shooman, 1968]. The failure rate is typically denoted as λ when it is assumed to have a constant value. To more clearly understand the mathematical basis for the concept of a failure rate, first consider the definition of the reliability function. The **reliability** $R(t)$ of a component, or a system, is the conditional probability that the component operates correctly throughout the interval $[t_0, t]$ given that it was operating correctly at the time t_0.

There are a number of different ways in which the failure rate function can be expressed. For example, the failure rate function $z(t)$ can be written strictly in terms of the reliability function $R(t)$ as

$$z(t) = \left(-\frac{dR(t)/dt}{R(t)} \right)$$

Similarly, $z(t)$ can be written in terms of the unreliability $Q(t)$ as

$$z(t) = -\frac{dR(t)/dt}{R(t)} = \frac{dQ(t)/dt}{1-Q(t)}$$

where $Q(t) = 1 - R(t)$. The derivative of the unreliability, $dQ(t)/dt$, is called the *failure density function.*

The failure rate function is clearly dependent upon time; however, experience has shown that the failure rate function for electronic components does have a period where the value of $z(t)$ is approximately constant. The commonly accepted relationship between the failure rate function and time for electronic components is called the bathtub curve and is illustrated in Fig. 93.6. The bathtub curve assumes that during the early life of systems, failures occur frequently due to substandard or weak components. The decreasing part of the bathtub curve is called the early-life or infant mortality region. At the opposite end of the curve is the wear-out region where systems have been functional for a long period of time and are beginning to experience failures due to the physical wearing of electronic or mechanical components. During the intermediate region, the failure rate function is assumed to be a constant. The constant portion of the bathtub curve is called the useful-life phase

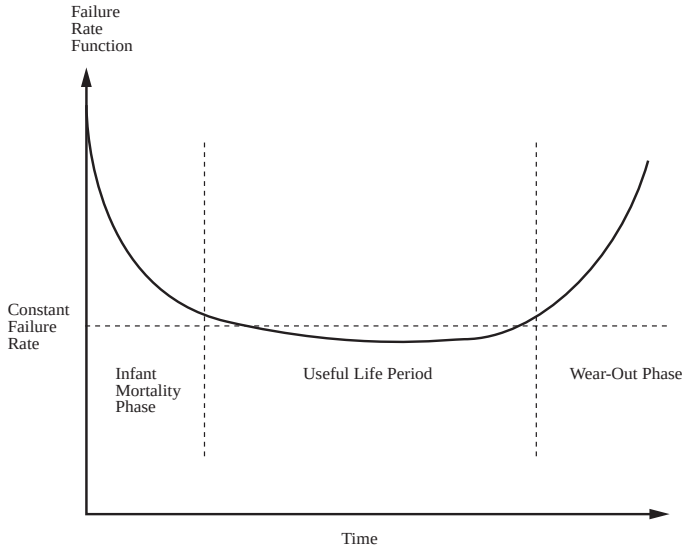

FIGURE 93.6 Bathtub curve relationship between the failure rate function and time.

of the system, and the failure rate function is assumed to have a value of λ during that period. λ is referred to as the failure rate and is normally expressed in units of failures per hour.

The reliability can be expressed in terms of the failure rate function as a differential equation of the form

$$\frac{dR(t)}{dt} = -z(t)R(t)$$

The general solution of this differential equation is given by

$$R(t) = e^{-\int z(t)dt}$$

If we assume that the system is in the useful-life stage where the failure rate function has a constant value of λ, the solution to the differential equation is an exponential function of the parameter λ given by

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

where λ is the constant failure rate. The exponential relationship between the reliability and time is known as the *exponential failure law,* which states that for a constant failure rate function, the reliability varies exponentially as a function of time.

In addition to the failure rate, the mean time to failure (MTTF) is a useful parameter to specify the quality of a system. The MTTF is the expected time that a system will operate before the first failure occurs. The MTTF can be calculated by finding the expected value of the time of failure.

From probability theory, we know that the expected value of a random variable, X, is

$$E[X] = \int_{-\infty}^{\infty} x f(x) dx$$

where $f(x)$ is the probability density function. In reliability analysis we are interested in the expected value of the time of failure (MTTF), so

$$\text{MTTF} = \int_0^\infty t f(t) dt$$

where $f(t)$ is the failure density function, and the integral runs from 0 to ∞ because the failure density function is undefined for times less than 0. We know, however, that the failure density function is

$$f(t) = \frac{dQ(t)}{dt}$$

so, the MTTF can be written as

$$\text{MTTF} = \int_0^\infty t \frac{dQ(t)}{dt} dt$$

Using integration by parts and the fact that $dQ(t)/dt = -dR(t)/dt$ we can show that

$$\text{MTTF} = \int_0^\infty t \frac{dQ(t)}{dt} dt = -\int_0^\infty t \frac{dR(t)}{dt} dt = \left[-tR(t) + \int R(t) dt\right]\Big|_0^\infty = \int_0^\infty R(t) dt$$

Consequently, the MTTF is defined in terms of the reliability function as

$$\text{MTTF} = \int_0^\infty R(t) dt$$

which is valid for any reliability function that satisfies $R(\infty) = 0$.

The mean time to repair (MTTR) is simply the average time required to repair a system. The MTTR is extremely difficult to estimate and is often determined experimentally by injecting a set of faults, one at a time, into a system and measuring the time required to repair the system in each case. The MTTR is normally specified in terms of a repair rate, μ, which is the average number of repairs that occur per time period. The units of the repair rate are normally number of repairs per hour. The MTTR and the rate, μ, are related by

$$\text{MTTR} = \frac{1}{\mu}$$

It is very important to understand the difference between the MTTF and the mean time between failure (MTBF). Unfortunately, these two terms are often used interchangeably. While the numerical difference is small in many cases, the conceptual difference is very important. The MTTF is the average time until the first failure of a system, while the MTBF is the average time between failures of a system. If we assume that all repairs to a system make the system perfect once again just as it was when it was new, the relationship between the MTTF and the MTBF can be determined easily. Once successfully placed into operation, a system will operate, on the average, a time corresponding to the MTTF before encountering the first failure. The system will then require

some time, MTTR, to repair the system and place it back into operation once again. The system will then be perfect once again and will operate for a time corresponding to the MTTF before encountering its next failure. The time between the two failures is the sum of the MTTF and the MTTR and is the MTBF. Thus, the difference between the MTTF and the MTBF is the MTTR. Specifically, the MTBF is given by

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

In most practical applications the MTTR is a small fraction of the MTTF, so the approximation that the MTBF and MTTF are equal is often quite good. Conceptually, however, it is crucial to understand the difference between the MTBF and the MTTF.

An extremely important parameter in the design and analysis of fault-tolerant systems is fault coverage. The fault coverage available in a system can have a tremendous impact on the reliability, safety, and other attributes of the system. Fault coverage is mathematically defined as the conditional probability that, given the existence of a fault, the system recovers [Bouricius et al., 1969]. The fundamental problem with fault coverage is that it is extremely difficult to calculate. Probably the most common approach to estimating fault coverage is to develop a list all of the faults that can occur in a system and to form, from that list, a list of faults from which the system can recover. The fault coverage factor is then calculated appropriately.

Reliability is perhaps one of the most important attributes of systems. The reliability of a system is generally derived in terms of the reliabilities of the individual components of the system. The two models of systems that are most common in practice are the series and the parallel. In a series system, each element of the system is required to operate correctly for the system to operate correctly. In a parallel system, on the other hand, only one of several elements must be operational for the system to perform its functions correctly.

The series system is best thought of as a system that contains no redundancy; that is, each element of the system is needed to make the system function correctly. In general, a system may contain N elements, and in a series system each of the N elements is required for the system to function correctly. The reliability of the series system can be calculated as the probability that none of the elements will fail. Another way to look at this is that the reliability of the series system is the probability that all of the elements are working properly. The reliability of a series system is given by

$$R_{\text{series}}(t) = R_1(t)R_2(t) \ldots R_N(t)$$

or

$$R_{\text{series}}(t) = \prod_{i=1}^{N} R_i(t)$$

An interesting relationship exists in a series system if each individual component satisfies the exponential failure law. Suppose that we have a series system made up of N components, and each component, i, has a constant failure rate of λ_i. Also assume that each component satisfies the exponential failure law. The reliability of the series system is given by

$$R_{\text{series}}(t) = e^{-\lambda_1 t} e^{-\lambda_2 t} \ldots e^{-\lambda_N t}$$

$$R_{\text{series}}(t) = e^{-\sum_{i=1}^{N} \lambda_i t}$$

The distinguishing feature of the basic parallel system is that only one of N identical elements is required for the system to function. The reliability of the parallel system can be written as

$$R_{\text{parallel}}(t) = 1.0 - Q_{\text{parallel}}(t) = 1.0 - \prod_{i=1}^{N} Q_i(t) = 1.0 - \prod_{i=1}^{N} (1.0 - R_i(t))$$

It should be noted that the equations for the parallel system assume that the failures of the individual elements that make up the parallel system are independent.

M-of-*N* systems are a generalization of the ideal parallel system. In the ideal parallel system, only one of *N* modules is required to work for the system to work. In the *M*-of-*N* system, however, *M* of the total of *N* identical modules are required to function for the system to function. A good example is the TMR configuration where two of the three modules must work for the majority voting mechanism to function properly. Therefore, the TMR system is a 2-of-3 system.

In general, if there are *N* identical modules and *M* of those are required for the system to function properly, then the system can tolerate $N - M$ module failures. The expression for the reliability of an *M*-of-*N* system can be written as

$$R_{M\text{-of-}N}(t) = \sum_{i=0}^{N-M} \binom{N}{i} R^{N-i}(t)(1.0 - R(t))^i$$

where

$$\binom{N}{i} = \frac{N!}{(N-i)!\,i!}$$

The **availability**, $A(t)$, of a system is defined as the probability that a system will be available to perform its tasks at the instant of time *t*. Intuitively, we can see that the availability can be approximated as the total time that a system has been operational divided by the total time elapsed since the system was initially placed into operation. In other words, the availability is the percentage of time that the system is available to perform its expected tasks. Suppose that we place a system into operation at time $t = 0$. As time moves along, the system will perform its functions, perhaps fail, and hopefully be repaired. At some time $t = t_{\text{current}}$, suppose that the system has operated correctly for a total of t_{op} hours and has been in the process of repair or waiting for repair to begin for a total of t_{repair} hours. The time t_{current} is then the sum of t_{op} and t_{repair}. The availability can be determined as

$$A(t_{\text{current}}) = \frac{t_{\text{op}}}{t_{\text{op}} + t_{\text{repair}}}$$

where $A(t_{\text{current}})$ is the availability at time t_{current}.

If the average system experiences *N* failures during its lifetime, the total time that the system will be operational is *N*(MTTF) hours. Likewise, the total time that the system is down for repairs is *N*(MTTR) hours. In other words, the operational time, t_{op}, is *N*(MTTF) hours and the downtime, t_{repair}, is *N*(MTTR) hours. The average, or steady-state, availability is

$$A_{SS} = \frac{N(\text{MTTF})}{N(\text{MTTF}) + N(\text{MTTR})}$$

We know, however, that the MTTF and the MTTR are related to the failure rate and the repair rate, respectively, for simplex systems, as

$$\text{MTTF} = \frac{1}{\lambda}$$

$$\text{MTTR} = \frac{1}{\mu}$$

Therefore, the steady-state availability is given by

$$A_{SS} = \frac{1/\lambda}{1/\lambda + 1/\mu} = \frac{1}{1 + \lambda/\mu}$$

Defining Terms

Availability, $A(t)$: The probability that a system is operating correctly and is available to perform its functions at the instant of time t.

Dependability: The quality of service provided by a particular system.

Error: The occurrence of an incorrect value in some unit of information within a system.

Failure: A deviation in the expected performance of a system.

Fault: A physical defect, imperfection, or flaw that occurs in hardware or software.

Fault avoidance: A technique that attempts to prevent the occurrence of faults.

Fault tolerance: The ability to continue the correct performance of functions in the presence of faults.

Maintainability, $M(t)$: The probability that an inoperable system will be restored to an operational state within the time t.

Performability, $P(L,t)$: The probability that a system is performing at or above some level of performance, L, at the instant of time t.

Reliability, $R(t)$: The conditional probability that a system has functioned correctly throughout an interval of time, $[t_0,t]$, given that the system was performing correctly at time t_0.

Safety, $S(t)$: The probability that a system will either perform its functions correctly or will discontinue its functions in a well-defined, safe manner.

Related Topics

98.1 Introduction • 98.4 Relationship between Reliability and Failure Rate

References

A. Avizienis, "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," *IEEE Transactions on Computers,* vol. C-20, no. 11, pp. 1322–1331, November 1971.

W. G. Bouricius, W. C. Carter, and P. R. Schneider, "Reliability modeling techniques for self-repairing computer systems," in *Proceedings of the 24th ACM Annual Conference,* pp. 295–309, 1969.

L. Chen and A. Avizienis, "N-version programming: A fault tolerant approach to reliability of software operation," in *Proceedings of the International Symposium on Fault Tolerant Computing,* pp. 3–9, 1978.

R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal,* vol. 26, no. 2, pp. 147–160, April 1950.

B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems,* Reading, Mass.: Addison-Wesley, 1989.

J-C. Laprie, "Dependable computing and fault tolerance: Concepts and terminology," in *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing,* Ann Arbor, Mich.: pp. 2–11, June 19–21, 1985.

S. Lin and D. J. Costello, Jr., *Error Control Coding: Fundamentals and Applications,* Englewood Cliffs, N.J.: Prentice-Hall, 1983.

V. P. Nelson and B. D. Carroll, *Tutorial: Fault-Tolerant Computing,* Washington, D.C.: IEEE Computer Society Press, 1986.

M. L. Shooman, *Probabilistic Reliability: An Engineering Approach,* New York: McGraw-Hill, 1968.

Further Information

The *IEEE Transactions on Computers, IEEE Computer* magazine, and the *Proceedings of the IEEE* have published numerous special issues dealing exclusively with fault tolerance technology. Also, the IEEE International Symposium on Fault-Tolerant Computing has been held each year since 1971. Finally, the following textbooks are available, in addition to those referenced above:

P. K. Lala, *Fault Tolerant and Fault Testable Hardware,* Englewood Cliffs, N.J.: Prentice-Hall, 1985.
D. K. Pradhan, *Fault-Tolerant Computing: Theory and Techniques,* Englewood Cliffs, N.J.: Prentice-Hall, 1986.
D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable Systems Design,* 2nd ed., Bedford, Mass.: Digital Press, 1992.