Chapter 7

Fault-Tolerance Techniques

[1]

7.1 Introduction

The failure rates of real-time computers must be extraordinarily small: indeed, they must be smaller than the failure rates of the components out of which they are built. Such computers must therefore be *fault-tolerant*, i.e., be able to continue operating despite the failure of a limited subset of their hardware or software. They must also be *gracefully degradable*; as the size of the faulty set increases, the system must not suddenly collapse, but should still be capable of executing part of its workload. Figure 7.1 shows how a properly designed fault-tolerant system tends to behave as the failures increase in number and scope. Initially, the performance is kept from degrading despite a limited number of failures. This is done by such tactics as switching in spares, and using up slack computational capacity. As the extent of the failures increases, performance starts to degrade. The system runs out of slack capacity, and the operating system must begin shedding computational load. The less critical tasks are shed, and the system is still able to carry out the critical core of tasks which are vital to the survival of the controlled process. Finally, when the failures are so great in extent that the computer can no longer meet even these critical computational requirements, we have *system failure*, which may have catastrophic consequences

[1]

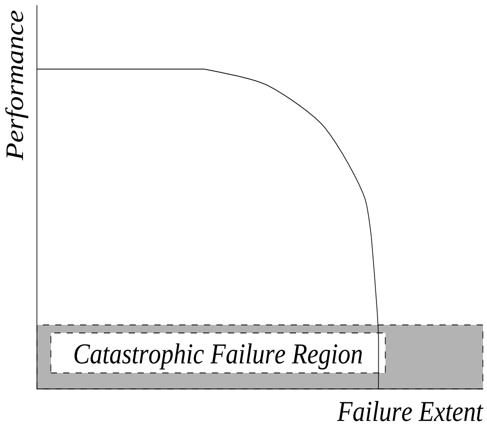


Figure 7.1: Performance degradation of a fault-tolerant system

for the application. The goal of the system designer is to ensure that the probability of entering the system failure state is acceptably small.

The first step in making a system robust is to reduce the number of faults that the system will encounter. This means reducing the number of design and manufacturing faults, and to reduce its vulnerability to disturbances in the operating environment.

Reducing the incidence of design faults requires careful specification and design, followed by extensive design reviews. The system must be built of high-grade components to reduce the probability of manufacturing defects. For example, the VLSI chips out of which it is built must each be tested carefully to reduce the probability of using a bad component. Extensive system-wide testing must also be conducted before the system is released for use. Indeed, the system must be *designed for testability*, i.e., it should be designed to facilitate thorough testing. Testing is not a trivial process: most systems are so complex, with so many inputs and operating conditions that it is impossible to test the system under every condition and every combination of inputs. We can never be sure that a complex system leaves the factory with no faults in it; all that can be done is to minimize the probability of faults. Designing for testability is an entire subject in its own right, and we do not cover it here.

Making the system robust against interference from the operating environment requires shielding and the use of appropriate (radiation-hardened) components. In such applications, using the latest integrated-circuit technology is not necessarily a good idea. The more mature the technology, the more likely it is to produce robust circuits.

No matter how robustly the system is designed, there will likely always be faults in it, either at the time it is released, or later while it is in use. The system must thus be able to tolerate a certain number of failures and still function.

Since real-time systems must meet deadlines, there must be both a short-term and a long-term response to failure. The short-term response consists of quickly correcting for a failure to allow immediate deadlines to be met. The long-term response consists of locating the failure, determining the best response to it, and initiating a recovery and reconfiguration procedure.

In this chapter, we will survey techniques for fault-tolerance applied at the system level. There is a large body of research in fault-tolerance at the circuit level, which will be outside the scope of our discussion. Such fault-tolerance is used to make individual components or chips more reliable (using self-checking circuits, interstitial redundancy, complementary logic, etc.), but is more within the province of the designer of integrated circuits than that of the system designer. We will not cover fault-tolerant synchronization here: that topic is so important that we have an entire chapter devoted to it.

7.2 Some Definitions

A hardware *fault* is some physical defect which can cause a component to malfunction. A broken wire, or the output of a logic gate that is perpetually stuck at some logic value (0 or 1), are hardware faults. A software fault is a "bug" which can cause the program to fail for a given set of inputs.

An *error* is a manifestation of a fault. For example, a broken wire will cause an error if the system tries to propagate a signal through it. A program that has a fault which induces incorrect outputs for some set I of inputs will generate errors if that set of inputs is applied.

There is usually a duration, called the *fault latency*, between the onset of a fault and its manifestation as an error. This duration can be considerable. Since faults by themselves are invisible to the outside world, and only show themselves when they cause errors, such latency can, as we shall see, impact the reliability of the overall system. When an error is produced, there is usually a duration – called the *error latency* – before it is either recognised as an error, or causes failure of the system. Like fault-latencies, error latencies

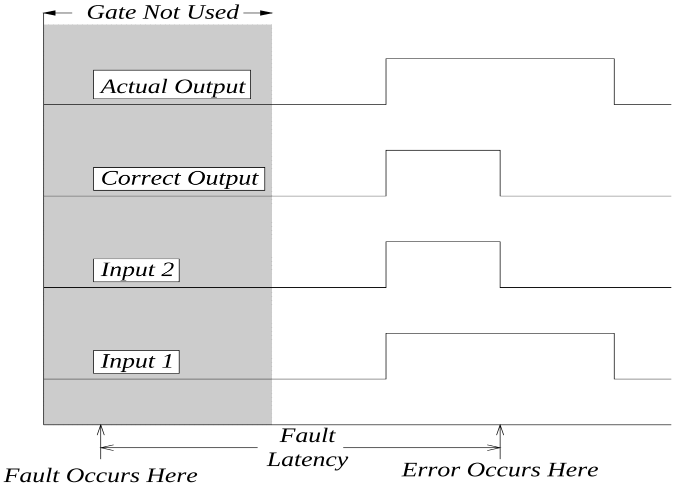


Figure 7.2: Illustrating fault latency

can have a considerable impact on the overall system reliability.

As an example of fault-latency, consider an AND gate, which has one of its inputs (say, input line 2) stuck at 0. That is, the gate behaves as if the input on that wire were 0, regardless of what the actual input happens to be. Figure 7.2 illustrates the concept of fault and error latency.

Error recovery is the process by which the system attempts to recover from the effects of an error. There are two forms of error recovery: *forward* and *backward*. In forward error recovery, the error is masked without any computations having to be redone. In backward error recovery, the system is "rolled back" to a moment in time before the error is believed to have occurred (this roll-back involves the restoration of system state to the state at that moment), and computation is carried out again. Backward error recovery uses time redundancy, since it consumes additional time to mask the effects of failure. We shall encounter examples of both forward and backward recovery in this chapter.

7.3 What Causes Failures?

There are three causes of failure: *(a)* errors in specification or design, *(b)* defects in components, and *(c)* environmental effects.

Mistakes in specification and design are very difficult to guard against.

Many hardware failures and all software failures occur due to such mistakes. Specification can be regarded as a mapping from the "real world" into an artificial specification space. The real-world requirements are expressed in formal or informal terms, in terms that can be understood by the computer designer. The person who writes the specification must thoroughly understand the application and the environment in which that application will operate. The specification is, strictly speaking, the *only* link in the design process between the real world of the application and the more cloistered world of the computer designer. If the specification is wrong, everything that proceeds from it – the design and implementation – is likely to be unsatisfactory. A specification must be unambiguous, i.e., it must not admit to more than one interpretation. It must be complete, i.e., it must not require additions to define a whole system. And yet, it should not be so restrictive that it unduly takes away the designer's initiative. There is no mechanical way of writing a specification. It is possible to develop languages in which specifications can be conveniently written. However, specification is intrinsically an art, which, because it provides an interface with the undefinable "real world," is itself incapable of being completely mechanized.

It is very difficult to ensure that one has got the specification completely right. Common-sense checks can be made. Specifications can be reviewed by persons unconnected with writing them to increase the chances of mistakes being caught. Detailed "defenses" of a specification can be carried out, where each line of the specification is defended before a committee playing devil's advocate. "Be careful" is the best advice one can give to a specification writer.

By contrast, the design that is developed from a formal specification can, theoretically, be checked formally. Indeed, any design is just one realization of the specification, and theoretically speaking, can therefore be formally checked against it. Reality is, alas, quite different, with formal techniques being too primitive to be of much value in large-scale systems. However, the large number of researchers working in this area makes it possible that the situation could change in the medium-term future.

The second cause of faults is defects in components. Only hardware components have defects in them: these may arise from manufacturing defects, or from defects caused by the wear and tear of use. For example, a MOSFET may fail due to electromigration, which is the drifting away over time of metal atoms towards the cathode.

The third cause of faults is the operating environment. Devices can be subjected to a whole array of stresses, depending on the application. Poor ventilation or excessively high ambient temperatures can melt components or otherwise damage them[2]. If the computer is in a missile, it can undergo high G-forces and vibrational stress.

Sometimes, especially with aerospace or robotics applications, there can be considerable electromagnetic or elementary-particle (such as α-particle) radiation, which can cause spurious changes in the state of flip-flops. For example, there is an area between 600 and 6000 nautical miles above the South Atlantic, called the South Atlantic Anomaly, which is especially rich in elementary particles. Some spacecraft are exposed to as much as 10^5 rads[3] per hour. In silicon dioxide, ionizing radiation produces about 7.6×10^{12} electron-hole pairs per rad per cubic centimetre. This charge can cause devices to change state.

A rough idea of how the reliability of a component can be affected by some of these factors can be obtained by considering the following expression for failure rate, which was developed by the US Department of Defense (MIL-HDBK-217E):

$$\lambda = \pi_L \pi_Q (C_1 \pi_T \pi_V + C_2 \pi_E) \tag{7.1}$$

where:

π_L Represents the fabrication process (1 if mature technology, 10 otherwise).

π_Q Represents the testing process to discard devices which have manufacturing defects (ranges between 0.25 and 20).

C_1, C_2 Complexity factors expressed as a function of the number of transistors in the device and the number of pins

π_T Represents the effects of temperature and is a function of the type of device (ranges between 0.1 and 1000).

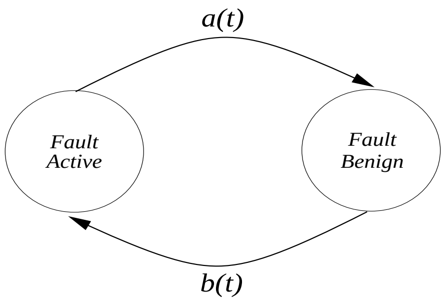
π_V Represents voltage stress for CMOS devices (is 1 if the device is not CMOS, and ranges from 1 to 10 if it is CMOS).

π_E Represents other stresses in the operating environment (ranges between 0.38 and 220).

A glance at the range of these parameters shows how potent is the impact of the operating environment. Voltage (for CMOS) and temperature stresses

[2] In general, the hotter a component, the higher its failure rate.

[3] A rad is a unit of radiation, that measures how much energy is absorbed per unit of mass. It stands for *radiation absorbed dose*. Each rad corresponds to the absorption of 100 ergs per gram.



Fault Type	Condition
Permanent	a(t)=0; b(t)>0
Transient	a(t)>0; b(t)=0
Intermittent	a(t)>0; b(t)>0

Figure 7.3: State diagram of the fault classes

are multiplicative, and their product ranges across four orders of magnitude. The same component, used in a benign environment can be up to 10,000 times more reliable than if used in a harsh environment. While the actual numbers mentioned above are likely to change with technology, the importance Equation (7.1) ascribes to the operating environment is unlikely to change in the near future.

7.4 Fault Types

Faults are classified according to their *(a)* temporal behaviour, and *(b)* output behaviour. A fault is said to be *active* when it is physically capable of generating errors, and is *benign* when it is not.

7.4.1 Temporal Behaviour Classification

There are three fault types: *permanent*, *intermittent*, and *transient*. A permanent fault, as the name implies, does not die away with time, but remains until it is repaired or the affected unit is replaced. An intermittent fault cycles between the *fault-active* and *fault-benign* states. A *transient* fault dies away after some time. See Figure 7.3. In the figure, $a(t)$ and $b(t)$ are the rates at which the fault switches states, and t is the age of the fault.

Intermittent faults can be caused, for example, by loose wires. Transient faults can be caused by environmental effects: for instance, if there is a burst of electromagnetic radiation, and the memory is not properly shielded, the contents of the memory can be altered, without the memory chips themselves suffering any structural damage. When the memory is rewritten, the fault will go away. Experiments suggest that the vast majority of hardware faults are transient, and that only a minority are permanent. Transient failures are hard to catch, since quite often, by the time the system has recognised that such a failure has occurred, they have disappeared, leaving behind no permanent defect that can be located.

It is likely that as devices become faster, their vulnerability to environmental effects which lead to transient failure will increase. This is because the principal way of making a device run faster is to make it smaller, so that signal propagation times as well as the volume of charge required to switch a flip-flop from one state to the other is reduced. It therefore becomes more likely that a charged particle passing through a device or electromagnetic induction will cause a spurious change in the device state. This problem is especially serious in real-time systems which operate in hazardous environments, such as space, which are replete with such radiation.

7.4.2 Output Behaviour Classification

A fault is also characterized by the nature of the errors that it generates. There are two categories of output behaviour: *non-malicious* and *malicious*.

To understand the difference, consider a unit A which is providing output to units $B_1, \cdots, B_n$. If A has a non-malicious fault, any errors it generates will be interpreted in a consistent way by all of the units B_i. For example, if A has an output line that is stuck at logic 0, this is a non-malicious fault: all the units that receive input from this line will read the line as producing a logic 0.

It is possible, however, for A to fail in such a way that the B_i see non-identical values of output for the same physical signal. For example, let an output line of A float rather than be stuck at 0. Now, logic circuits are designed to interpret a certain range of voltages (say $[0_\ell, 0_h]$) as corresponding to a logical 0 and another range (say $[1_\ell, 1_h]$) as corresponding to a logical 1. So long as the output voltage maintained by that faulty output line is in one of these ranges, it will be interpreted consistently by all the B_i. However, if the voltage is outside these ranges, e.g., between the values that would cause it to be interpreted as a 0 or as a 1, consistency can break down. It is then possible for the same voltage signal to be interpreted as representing a different logic value by different receivers (see Figure 7.4).

Failures of this type, which correspond to an inconsistent output, are much harder to neutralise than the non-malicious type. Indeed, one may

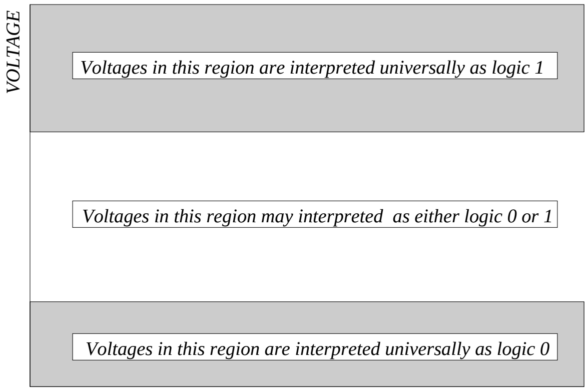

VOLTAGE
Voltages in this region are interpreted universally as logic 1
Voltages in this region may interpreted as either logic 0 or 1
Voltages in this region are interpreted universally as logic 0

Figure 7.4: Interpretation of voltages as logic values

think of them – somewhat whimsically – as the output of some malicious intelligence that has captured the device, and is putting forth errors in such a way as to cause maximum disruption to the functioning of the system. Such failures are known as *malicious* or *Byzantine* failures. In short, a malicious fault is one which is assumed to be able to behave arbitrarily.

In this context the terms *fail-safe* and *fail-stop* are defined. A unit is said to be *fail-stop* if it responds to up to a certain maximum number of failures by simply stopping, rather than putting out incorrect output. Fail-stop units typically consist of multiple processors, running the same tasks, and comparing results. Faults are detected by comparing outputs: if the outputs are different, the whole unit turns itself off. A simple example is to have a system with two processors, each running the same tasks on the same inputs, and comparing its output to that of the other processor. If either processor detects a discrepancy between these outputs, it has the ability to turn off the interface to the network, thus isolating itself from the rest of the system.

A system is said to be *fail-safe* if its failure mode is biased so that the application process does not suffer catastrophe upon failure. The classic example of a fail-safe system is a traffic light controller. If, when it fails, it sets all the lights to green, catastrophe can ensue. On the other hand, if it is designed in such a way that failure of the controller results in the lights all turning to red, or flashing a warning sign, it is fail-safe. Some applications may not admit of a fail-safe mode.

7.5 Independence and Correlation

Component failures may be *independent* of one another or *correlated.* A failure is said to be independent if it does not directly or indirectly cause another failure. Failures are said to be *correlated* if they are related in some way. For example, they may be triggered by the same cause, or one of them might cause the others to occur. Failures can be correlated due to a physical or electrical coupling of units, or because the same external event (e.g., a lightning strike) affects both units. Correlated failures are far more difficult to deal with than independent failures, and means must be found to avoid them if at all possible. Designers might consider shielding the electronics so that the probability of a transient upset due to the environment is reduced. They may want to ensure that the sources of power for the processors are disparate. Sometimes, the hardware is physically separated, so that the probability of multiple processors being affected by the same environmental event is reduced: for example, the processors of an aircraft-control computer might be distributed amongst multiple instrument bays.

7.6 Fault Detection

There are essentially two ways to determine that a processor is malfunctioning: online and offline.

Online detection goes on in parallel with normal system operation. One way of doing this is to check for any behaviour that is inconsistent with correct operation.

The following actions are indicative of a faulty processor.

- Branch to an invalid destination.
- Fetching an opcode from a location containing data.
- Writing into a portion of memory to which the process has no write access.
- Fetching an illegal opcode.
- Inactive for more than a prescribed period.

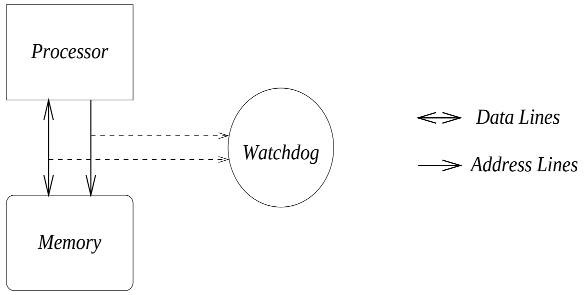


Figure 7.5: Using a watchdog processor

One can have a monitor (called a *watchdog processor*) associated with each processor, looking for these signs that the processor is faulty. The watchdog processor watches the data and address lines, as shown in Figure 7.5.

A second approach is to have multiple processors which are supposed to put out the same result, and compare these results. A discrepancy indicates the existence of a fault.

Offline detection consists of running diagnostic tests. When a processor is running such a test, it obviously cannot be executing the applications software. Diagnostic tests can be scheduled just like ordinary tasks. The greater the failure rate, the greater must be the frequency with which these tests are run.

7.7 Fault and Error Containment

When a fault or error occurs in one part of the system, it can, if unchecked, spread through the system like an infectious disease. A fault in one part of the system might, for example, cause large voltage swings in another; a fault-free processor can put out erroneous results as a result of using erroneous input from a faulty unit. Faults and errors must therefore be prevented from spreading through the system. This is called *containment*. The concept is old, and has been widely used in other fields. For example, ships have long been divided into separate watertight compartments, so that damage to a small number of the compartments does not sink the ship. Persons suffering from highly infectious diseases, like the plague or tuberculosis, have traditionally been placed in hospital isolation wards.

The system is divided into *fault* and *error containment zones* (FCZ and

ECZ, respectively). An FCZ is a subset of the system that operates correctly despite arbitrary logical or electrical faults outside the subset. That is, the failure of some part of the computer outside an FCZ cannot cause any element inside that FCZ to fail. Hardware inside an FCZ must be isolated from hardware outside it. The isolation should be sufficiently robust to withstand either a short-circuit or the application of the maximum voltage imposed on the lines connecting an FCZ to the outside world. Each FCZ should have an independent power supply and its own clocks. Of course, these clocks can be synchronized with the clocks in other FCZ's provided that malfunction in these outside clocks will not cause those inside the FCZ to fail. Typically, an FCZ is consists of a whole computer (including processors, memory, input/output and control interfaces).

The function of an error containment zone is to prevent errors from propagating across zone boundaries. This is typically achieved by *voting* redundant outputs, as we shall see in the next few pages.

7.8 Redundancy

Fault-tolerance consists of using and properly managing *redundancy.* In other words, if the system is to be kept running despite the failure of some of its parts, it must have spare capacity to begin with. There are four types of redundancy, and we shall consider each in some detail:

1. *Hardware Redundancy:* The system is provided with far more hardware than it would need if all the components were perfectly reliable. Typically, one uses between two and three times as much.

2. *Software Redundancy:* The system is provided with different software versions of tasks, preferably written independently by different teams of programmers, so that when one version of a task fails under certain inputs, another version can be used.

3. *Time Redundancy:* The task schedule has some slack in it, so that some tasks can be rerun if necessary and still meet critical deadines.

4. *Information Redundancy:* The data are coded in such a way that a certain number of bit errors can be detected and/or corrected.

The concept of using redundancy to enhance reliability is not new. The following quotation is from an article by D. Lardner in the *Edinburgh Review* of 1824:

> The most certain and effectual check upon errors which arise in the process of computation is to cause the same computations to be made by separate and independent computers[4]; and this check is rendered still more decisive if their computations are carried out by different methods.

7.9 Hardware Redundancy

Hardware redundancy is the use of additional hardware to compensate for failures. It can be used in two ways. The first is its use for fault detection, correction and masking. Multiple hardware units may be assigned to do the same task in parallel, and their results compared. If one or more of these units is faulty, we can expect that to show up in a disagreement between the results. This is fault detection. If only a minority of the units are faulty, and a majority of the units produce the same output, we can use this majority result and thus mask the effects of failure. If more than a minority of the units disagree, at least we have detected the failure, and can use other methods such as repeating the computation on other processors, thus correcting for the fault(s). Correction and masking are short-term measures. They neutralise the effects of the observed failure.

The second – and long-term – use of hardware redundancy is to replace malfunctioning units. It is possible for systems to be designed so that spares can be switched in to replace faulty units.

These two ways complement each other. The extent to which they are employed depends on the nature of the application. If the computer is used aboard an unmanned deep-space probe which must function unattended and unrepairable for over a decade, it must include sufficient spare modules for that duration, and ways to automatially switch them in. If, on the other hand, it is used in a chemical plant application, where the computer is accessible to repair, we are primarily interested in providing short-term measures to respond to failure, since the malfunctioning module can be manually replaced soon after the malfunction is detected.

[4] Back in 1824, "computers" meant "people who compute."

Redundancy is expensive. Duplicating or triplicating hardware is a luxury that is only justified in the most critical applications. Traditionally, such use has been largely confined to aerospace applications. However, as computers increase their presence in more cost-sensitive areas such as automobiles, only the most critical functions can be allocated such levels of redundancy. In aerospace and other applications, redundancy may be limited by how much power consumption, heat dissipation, or volume can be accommodated.

There is an infinity of ways in which to structure hardware redundancy, and it would be an instructive exercise for the reader to conjure up a few. Below is a description of some popular structures for hardware redundancy, each of which forms an error-containment zone. First, however, we turn to the important issue of voting and consensus.

7.9.1 Voting and Consensus

One way of using redundancy is to have multiple units execute the same task, and compare their outputs. If at least three units are involved, this comparison can choose the majority value (a process called *voting*), and thus mask the effects of some failures. If two units are used, the comparison can detect (but not correct) an error.

The designer must decide whether *exact* or *approximate* agreement is expected between functioning units. For example, if one processor produces an output of 1.400984 and another 1.400978, are the two to be considered in agreement or not? They are certainly not in exact agreement, by which is meant bit-by-bit equality. However, they are close. How close should the values be to be regarded as practically the same? One approach with approximate agreement is to accept the median value (if $N \geq 3$) as the true one. This does not, however, solve the problem of deciding if a fault exists or not. If there are three units A, B, C, of which both A and B produce the value x and C the value $x \pm \alpha$, for what values of α is C to be considered as faulty?

This problem does not arise if all the processors are exactly the same model, receive exactly the same external interrupts at the same time, and are running the same software to the same input: under such conditions, they should produce the same output if they are functioning properly. However, to provide resistance to design faults, we sometimes use diverse processor types, with different software. Roundoff errors can cause divergence in the less significant bits of the output. If the designer sets the value of α too high, the probability increases that C will be faulty without being treated as such by the system. On the other hand, if α is too small, there is the chance of false alarms, with a functional C being wrongly treated as faulty. There is thus a tradeoff here, but its resolution depends so much on the numerical properties of the algorithms being executed, and the nature of the hardware, that very little general guidance can be given.

Approximate agreement may also be used in cases where sensors are measuring the physical environment. Consider, for example, temperature sensors placed in a boiler. Even if they are highly accurate, it is impractical to expect them to produce exactly the same temperature reading. For one thing, they will be mounted slightly apart, and the temperature at one point may be slightly different from the temperature at another. Secondly, they will always have certain tolerances. To get a good output, the voter must then take the median: this ensures that if the faulty units are in a minority, the chosen output is either the output of a functioning sensor, or the output of a failed sensor whose value is sandwiched between the values of good sensors. Either way, the output is acceptable. The question of how to distribute the output of a sensor to processors so that they see consistent values is important, and we will return to it later in this chapter.

We now introduce three types of voters, which can function in cases where approximate agreement is required. We will require in each case a distance metric, which measures how far apart the outputs from redundant units are. Let $d(x_1, x_2)$ denote the distance between outputs x_1 and x_2. If x_1, x_2 are real numbers, we may simply define $d(x_1, x_2) = |x_1 - x_2|$. If they are vectors of real numbers, other suitable metrics (e.g., Cartesian distance) may be chosen. In each case, assume that there are N outputs to be voted on, and that N is an odd number.

The *formalized majority voter* works as follows. From an analysis of the software, suppose that it is reasonable to assume that if $d(x_1, x_2) \leq \epsilon$, x_1 and x_2 are sufficiently equal for all practical purposes[5]. Note that "sufficiently equal" does not satisfy the transitivity relations of true equality. That is, if x_1 is sufficiently equal to x_2 and x_2 is sufficiently equal to x_3, that does not necessarily imply that x_1 is sufficiently equal to x_3.

Then, the voter constructs a set of classes, $P_1, \cdots, P_n$ such that

- $x, y \in P_i$ iff $d(x, y) \leq \epsilon$

[5] Note that by setting $\epsilon = 0$, sufficiently equal becomes truly equal. Thus, sufficient equality is a weaker condition than true equality.

- P_i is maximal, i.e., if $z \notin P_i$, then there exists some $w \in P_i$ such that $d(w, z) > \epsilon$.

The P_is may share some elements. Take the largest of the P_is thus generated. If it has more than $\lceil N/2 \rceil$ elements in it, choose any of its elements as the output of the voter.

Example 1 *Let* $\epsilon = 0.001$ *for some five-unit system. Let the five outputs be 1.0000, 1.0010, 0.9990, 1.0005, and 0.9970. The classes will be* $P_1 = \{1.0000, 1.0010, 1.0005\}$*,* $P_2 = \{1.0000, 0.9990\}$*, and* $P_3 = \{0.9970\}$*. Note that 1.0000 is in both* P_1 *and* P_2*.* P_1 *is the largest class, and has* $3 > \lceil N/2 \rceil$ *elements. Any element in* P_1 *could be picked as the output of the voter.*

The *generalized k-plurality voter* works along the same lines as the generalized majority voter, except that it simply chooses any output from the largest partition, P_i, so long as P_i contains at least k elements. k is selected by the system designer.

The *generalized median voter* works by selecting the middle value (since N is assumed to be odd, such a middle value always exists). The middle value is selected by successively throwing away outlying values until only the middle value is left. The algorithm is as follows. Let the outputs being voted on be the set $S = \{x_1, \cdots x_N\}$.

1. Compute $d_{ij} = d(x_i, x_j)$ for all x_i, $x_j \in S$ for $i \neq j$.
2. Let $d_{k\ell}$ be the maximum such d_{ij}. (Break any ties arbitrarily.) Define $S = S - \{x_k, x_\ell\}$. If S contains only one element, that is the output of the voter; else, go back to Step 1.

Table 7.1 contains a comparison of these voters for various cases.

7.9.2 Static Pairing

One of the simplest schemes of all is to hardwire processors in pairs, and to discard the entire pair when one of the processors fails. Figure 7.6 illustrates a system which uses this approach. The pair runs identical software using identical inputs, and compares the output of each task. If the outputs are identical, the pair is functional. If either processor in the pair detects non-identical outputs, that is an indication that at least one of the processors in the pair is faulty. The processor which detects this discrepancy switches

Case	Majority Voter	k-Plurality Voter	Median Voter
All outputs correct and SE	Correct	Correct	Correct
Majority correct and SE	Correct	Correct	Correct
k correct and SE	No Output	Correct	Possibly Correct
All outputs correct but none SE	No Output	No Output	Correct
All outputs incorrect and none SE	No Output	No Output	Incorrect
Majory incorrect and SE	Incorrect	Incorrect	Incorrect

SE = Sufficiently equal

Table 7.1: Comparison of voter types (Adapted from P. R. Larczak, A. K. Caglayan, D. E. Eckhardt, "A Theoretical Investigation of Generalized Voters for Redundant Systems," *Proc. Fault-Tolerant Computing Symposium*, 1989.)

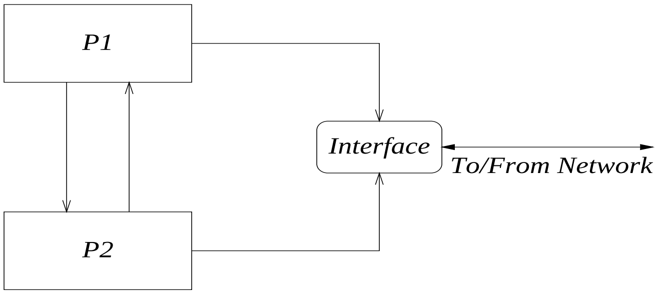


Figure 7.6: Illustration of static pairing

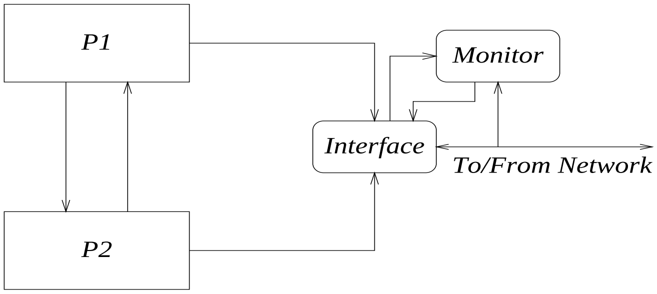


Figure 7.7: Use of a monitor

off the interface to the rest of the system, thus isolating this pair. This approach will work well so long as *(a)* the interface does not fail, and *(b)* both processors do not fail identically and around the same time. *(b)* is an acceptable assumption; *(a)* is not necessarily acceptable. To get around the interface being a critical point of failure, we can introduce an interface monitor (see Figure 7.7. This checks to ensure that the interface correctly transmits/receives messages, and switches it off whenever the interface is seen to be faulty. Likewise, the interface can check the output of the monitor, and turn itself off if it detects a fault in the monitor. This will work correctly unless both the monitor and the interface fail at the same time.

The probability of the pair being allowed to continue functioning despite failure is the probability that either *(a)* both processors fail in exactly the same way and produce exactly the same wrong result, or *(b)* both the monitor and the interface fail simultaneously. The actual expressions for this depend on the design of these components; however, the probability is usually sufficiently low that it can be assumed to be zero. On the other hand, the probability that the pair will be available for use is – if the simultaneous failures mentioned above are can be regarded as a second-order effect – the probability that there is no failure in any of the four components: the two processors, the monitor, and the interface.

Just because a fault has caused a static pair to be taken offline does not mean that it will never again be used. As mentioned above, the majority of failures tend to be transient, and if it has suffered a transient failure, the pair can be reinitialized and brought back on line after the failure has died. Checking to see if a previously faulty pair has recovered can be done by running test programs.

An analogy of this type of redundancy at the circuit level is the *self-*

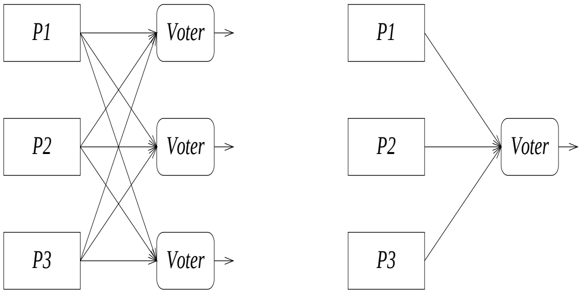


Figure 7.8: Structure of an NMR cluster

checking circuit. The simplest example of a self-checking circuit is the *two-rail checker*. Here, logic is duplicated into two modules and the output of the duplicates compared. A fault which generates an error in one module will cause a discrepancy at the outputs which can be detected.

7.9.3 N-Modular Redundancy

N-modular redundancy (NMR) is a scheme for forward error recovery. It works by using N processors instead of one, and *voting* on their output. N is usually odd. Figure 7.8 illustrates the concept for $N = 3$. One of two approaches are possible. In the first design, there are N voters and the entire cluster produces N outputs. In the second, there is just one voter.

To sustain up to m failed units, the NMR system requires $2m + 1$ units in all. The most popular is the *triplex*, which consists of a total of 3 units, and can can mask the effects of up to one failure.

Usually, the NMR clusters are designed to allow the purging of malfunctioning units. That is, when a failure is detected, the failed unit is checked to see whether or not the failure is transient. If it is not, it must be electrically isolated from the rest of the cluster, and a replacement switched in. The faster the replacement, the more reliable the cluster.

For example, consider a triplex. If a processor fails, that failure will be detected the next time there is a vote. Suppose the system determines that the failure is not transient. (This can be done by waiting for some time and then checking if the fault has gone away. If it has not, it is labelled a permanent fault. The actual waiting time depends on how long we expect the transient failures to be.) This processor is then isolated so that the cluster becomes, for the time being, only a duplex. During this time, if another

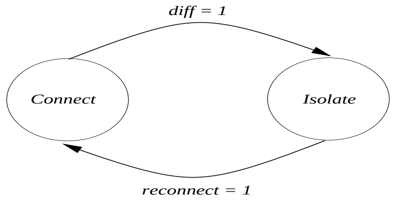


Figure 7.9: Finite-state machine description of monitor

processor fails, it will be possible to detect, but impossible to mask, it. A spare processor (assuming one is available) will be inducted into the cluster. Before it can start executing, it must first be *aligned*. That is, its memory must be made consistent with that of the other members of the cluster, and its clock synchronized with theirs. Once this is completed, the cluster has been restored to health.

Purging can be done either by hardware or by the operating system. Self-purging consists of a monitor at each unit comparing its output against the voted output. If there is a difference, the monitor disconnects the unit from the system. The monitor (see Figure 7.9) can be described as a finite-state machine with two states: *connect* and *isolate*. There are two signals: *diff*, which is set to 1 whenever the module output disagrees with the voter output, and *reconnect*, which is a command from the system to reconnect the module (after it has been tested by the system diagnostic circuitry, and found to be no longer faulty). The reconnect signal is very important because most faults are transient. To permanently disconnect a processor if it undergoes failure therefore wastes hardware and increase the probability of running out of functioning processors.

Another way to remove faulty units is through *sift-out* redundancy. Figure 7.10 shows the structure of a sift-out cluster of N processors. The comparator produces a total of $\left(\begin{array}{c} N \\ 2 \end{array}\right)$ outputs: one output for each pair of processors: if the pair disagrees, the corresponding line is, say a 1, while it is 0 if the pair produces a coincident output.

The detector is a circuit which disconnects a module which disagrees with the majority. It does so by analyzing the controller outputs. As with self-purging redundancy, a disconnected module can be reconnected on system command. Finally, the collector – which is a simple OR/AND circuit – produces output by sifting out the processors which have been disconnected

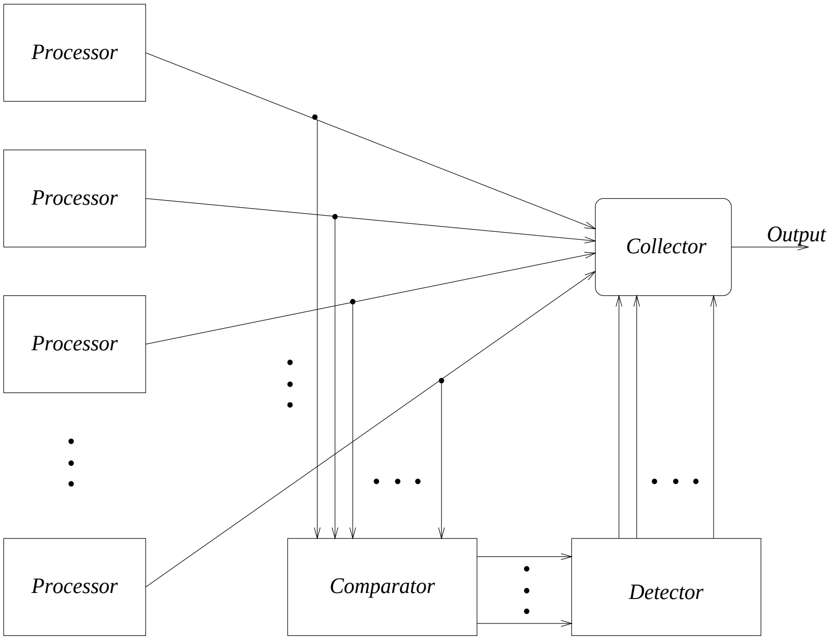


Figure 7.10: Sift-out redudancy

by the detector. The circuitry for the detector, collector, and comparator are very simple and left to the reader.

A cluster of $N = 2m + 1$ processors is sufficient to guard against up to m failures if the processor failures are not malicious. If they are malicious, a more sophisticated mechanism must be used. To see why, consider three sensors connected as in in Figure 7.11. Suppose that Sensor 1 in Figure 7.11 is maliciously faulty, and Sensors 2 and 3 are good (and provide identical inputs to the three voters). Let the values reported by the sensors to the various voters be as in Table 7.2. The median value put out by the voters will be 14, 15, and 16, respectively due to the inconsistent outputs of faulty sensor 1. We will show later in this chapter that for the system to be proof against up to m malicious failures, we require $N \geq 3m + 1$ processors.

Whether one voter or N are used depends on the application. Most applications require a single output. For example, if some valve is to be set to a particular position, what is needed is one output signal to that valve. On the other hand, some applications are specifically designed to take multiple outputs from the computer. For example, a control surface in an aircraft may be driven by the total force generated by N actuators, each obtaining

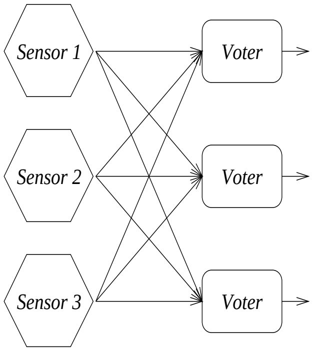


Figure 7.11: Voting on sensor values

Sensor No.	Output To		
	Voter 1	Voter 2	Voter 3
1	11	15	35
2	14	14	14
3	16	16	16

Table 7.2: Illustrating the effects of malicious failure

input from a separate voter. The system can be designed so that even if a minority of the actuators outputs are wrong (either because they receive incorrect inputs or the actuators themselves are faulty), the remainder have sufficient force to be able to set the control surface correctly.

A cluster with just one voter is clearly vulnerable to the failure of that one voter: fortunately, since the voter circuitry is so much simpler than that of the processor, voters can be expected to fail much less often than processors. Clusters with N voters are sometimes called *restoring organs*, because they produce N correct results so long as a majority of the processors (and all voters) are non-faulty.

A voter should not wait until all the inputs have been received. (If it did that, a failed processor which never produces an output would cause the voter to wait indefinitely, and thus cause deadlines to be missed). If enough coincident signals are received to make up a majority of the number in the cluster, that is sufficient. This approach is especially useful when *hardware diversity* is used. To avoid the effects of hardware design faults, the N processors in the cluster are not always identical, but might be different (diverse) models, the hope being that a design fault which causes a particular make of processor to fail on a given set of inputs will not be present in the others. Since the diverse processors have different speeds and perhaps different instruction sets, they can be expected to complete execution of the same task at quite different times. Waiting for only a majority of coincident signals would mean not having to wait for the laggards. There are two options regarding processors which have not completed execution before a majority of their functioning colleagues. The first is to terminate their execution, since a correct output has been calculated already. The second is to let them run to completion, and check their results against the consensus to detect any faults in these lagging processors.

If a voter waits until all functioning processors have produced output, it must be provided with a *watchdog timer*. This is a device which indicates when it has waited enough: when the timer goes off, any processors which have not yet delivered output are labelled as faulty. The setting of the watchdog timer depends on the run times of the cluster workload.

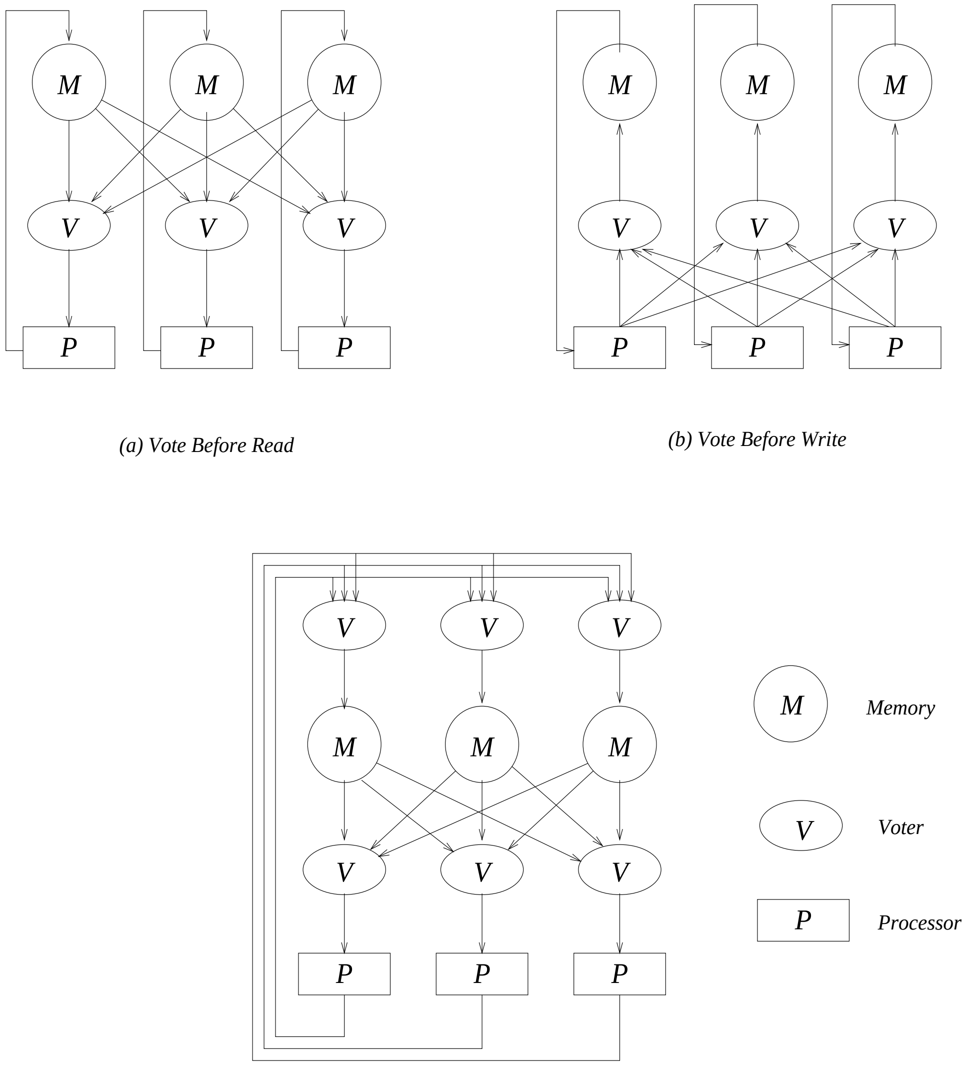
The designer has to decide at what level NMR is to be applied. An NMR cluster is a error-containment zone in that errors that are generated within it are masked (so long as a majority of the units are functioning), and never leak into the rest of the system. Intuitively, it should be clear that the smaller the error-containment zone, the better the reliability. However, there are problems with an excessive use of the NMR concept. The most obvious reason is that it is expensive. The second is that voting takes time, and having a very large number of voters between input and output can add an appreciable delay to the entire system.

For example, one can migrate the concept all the way to the gate level, and have N gates replicating in parallel what one gate would otherwise do. This approach is very expensive, however, and the authors are not aware of anyone using such an approach. NMR is used in practice at the board or module level. The architecture, speed, and reliability of the system are determined substantially by the level at which redundancy is used. Let us consider a few alternatives to illustrate this point, of redundancy at the module level.

The simplest – and the most common – scheme is to allocate each processor its own private memory, and to treat the processor-memory combination as an integral unit, i.e., a module. When the processor finishes a task, its output is applied to a voter, along with those of its colleagues in the cluster.

An alternative to this is to treat each processor and each memory as a separate module. There are several choices, of which we will consider three. Suppose we have a system of 3 processors, and 3 memory modules. Figure 7.12 illustrates three ways to arrange them, depending on when votes are taken. Keep in mind that the three processors are running the same workload, since they are part of a single triplex cluster. In *(a)*, the memory modules are arranged so that a read operation by a processor is replicated and goes to all three modules. The output of these memories is voted on, and the processor receives the result of the read. However, if a processor wishes to write, it may do so into one of the memory modules without affecting the others. In *(b)*, the reverse is the case. To write into a memory module, a vote of the corresponding write operations of the three processors is carried out, while a read can be done on just one processor. Finally, in *(c)*, both reads and writes are voted on.

There are advantages and disadvantages to each of these schemes. Every time there is a vote, there is the opportunity to mask out an error. For example, memories sometimes undergo transient failure due to charged particles passing through them and changing 0s to 1s or vice versa. A vote-before-read means that up to one such failure can be masked. Similarly, if a processor undergoes failure, that can be masked with respect to the memory by a vote-before-write policy. However, voting is expensive. Not only is there a delay caused by the voting process itself (i.e., propagation time through the voter), but the voter can only commence computation after a majority of the voter



(a) Vote Before Read

(b) Vote Before Write

(c) Vote after Read and Write

Figure 7.12: Some design alternatives

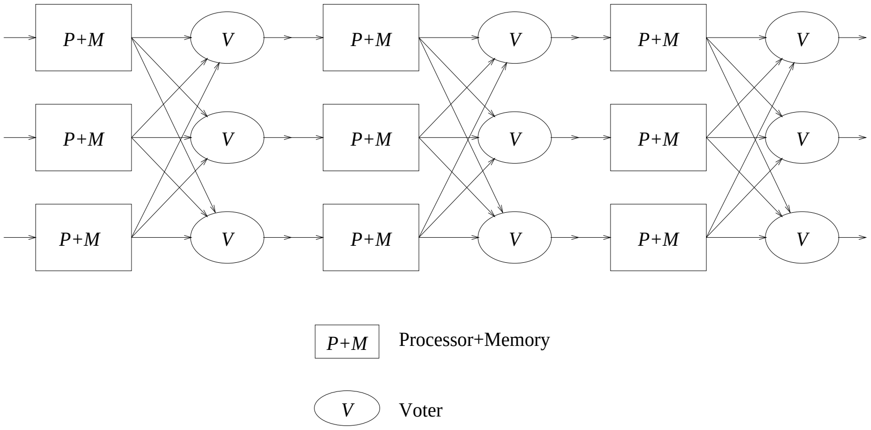


Figure 7.13: Cascaded triplex system

inputs are available. Thus, this scheme will result in complications unless the processors are identical, and produce outputs or read requests at very nearly the same time. Even then, there will be an excessive demand on memory bandwidth in the vote-before-read case. In this case, each memory module will see the sum total of all the read requests generated by all the processors. One way of getting around this is to vote on the read requests themselves, and instead of receiving a read request from each processor, amalgamate coincident read requests for the same location into one.

Yet another alternative is to chop up the task into portions and vote on the output of each portion. This is equivalent to cascading NMR clusters, as shown in Figure 7.13. Each of the stages in this cascade acts as a fault-containment unit. Deciding how the task should be divided is a tradeoff. If it is divided into many portions, there will be many votes and resistance to failure will be high. The price for this is that the voting overhead will be high.

7.10 Software Redundancy

Unlike hardware fault-tolerance, software fault-tolerance is a new and emerging field, and the state of the art is primitive.

As every programmer knows, it is practically impossible to write any large piece of software without introducing faults in it. Software faults are quite unlike hardware faults, however. Software never wears out, so that faults are never spontaneously generated during system operation. Software faults can be regarded instead as faults in design.

As systems become more complex, the ratio of software to hardware failures keeps increasing. Anecdotal evidence suggests that the software failure rate of modern systems is many times that of the hardware failure rate.

To provide reliability in the face of software faults, we must use redundancy. However, simply replicating the same software N times will not work: all N copies will fail for the same inputs. Instead, the N versions of the software must be diverse so that the probability that they fail on the same input is acceptably small. Such diversity in software can be introduced by having independent teams of programmers, with no contact between the teams, generating software for the same task. However, even then, there can be common-mode failures (i.e., multiple versions failing on the same input). For example, if the teams all work off the same specification, they may interpret an ambiguity in the specification in the same (incorrect) way. If they use similar numerical algorithms, the system is vulnerable to inputs for which these algorithms exhibit numerical instabilities. In every experiment that we are aware of, common-mode software failures have been significant. There is also the issue of cost: single-version software is already more expensive than the hardware in most large systems. Demanding N versions of software for even small N can be very expensive.

The designer needs answers to the following questions.

1. How many versions are required at a minimum to ensure acceptable levels of software reliability?

2. What procedures should be established to minimize common-mode failures amongst the various independently-generated versions?

3. How should the software be tested?

There are, as yet, no good answers to these questions. We do not have good models of software reliability. That is, given a program, how do we determine how many faults it has? Software reliability models are much less reliable than their hardware counterparts since hardware failure occurs due to physical reasons which can be modelled fairly well, while software failure occurs due to a design fault. Design faults – both hardware and software – are difficult to model since the processes that give rise to them are poorly understood[6]. Clearly, the number of faults is likely to be a function of the size of the software, its complexity, the skills of the programming team, the programming language used, the clarity and correctness of the specifications, and so on. All of these parameters, with the exception of program size, are difficult to measure – or even to define – precisely. Indeed, as we shall see in the chapter on reliability evaluation, arguments can be made to the effect that it is impractical to model the reliability of real-time software.

The number of versions will depend on the extent of the common-mode failures. How to measure this in an acceptably short time is another unsolved (and perhaps unsolvable) problem. The testing of the multiple versions of the software can be done back-to-back. That is, since these versions must produce approximately the same output for the same input (approximately since the versions may generate different roundoff and other numerical errors), any mismatch in outputs for the same input indicates an error. Of course, this will not catch those faults that are common to all the versions.

There are two approaches to handling multiple versions. *N-version programming* involves running all N versions in parallel and voting on the output. By contrast, in the *recovery-block* approach, only one version is run at any one time. The output of this version is put through an *acceptance test*, which checks to see if it is in an acceptable range. If so, the output is passed as correct. If not, another version is executed with its output being checked by the acceptance test, and so on. Figure 7.14 illustrates these concepts.

7.10.1 N-Version Programming

N-version programming owes whatever reliability it can provide to the diversity of the software. Such a system will fail whenever a majority of the versions fails on some input, and so the probability of common-mode failures must be minimized. To ensure this, one can pick teams of programmers who never communicate with one another, and work to develop independent versions of the required software.

Let us consider a few factors which affect the diversity of the multiple versions.

[6] However, since hardware is typically many times simpler than the software for most systems, the number of hardware design faults tends to be small, while there is usually a large number of software design faults. Thus, the problem of design faults is much more pressing in software than in hardware.

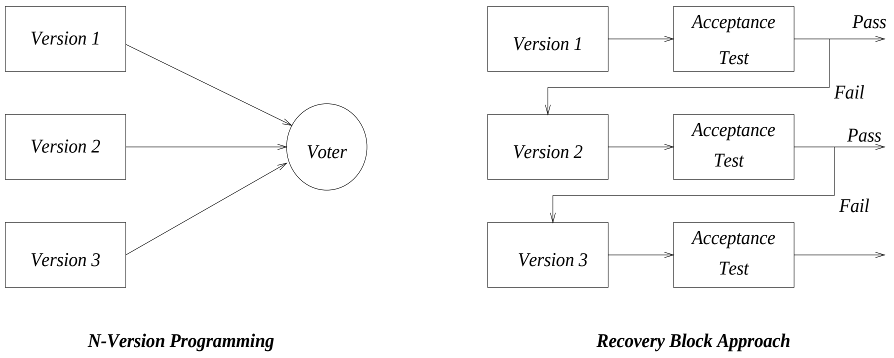


Figure 7.14: Software fault-tolerant structures

The first factor is the requirements specification. Errors and ambiguities in the specification are a major cause limiting diversity. A mistake in the specification, which causes a wrong output to be delivered in response to a given set of inputs will be translated into software versions which jointly fail on that set. If there is an ambiguity, programmers might interpret the specifications in a way that reflects their biases. Multiple teams of programmers may have similar biases as a result of their professional training or other cultural factors. Such similarity translates into the possibility of common-mode failures.

One response to this is to write the specifications in very formal terms, and subject them to a rigorous process of checking. However, being very formal in specification can render the specification difficult to understand. If a programmer, who most likely has a baccalaureate degree in computer science or even less, comes across a specification written in some strange formalism that he cannot understand fully, what is he to do? Typically, he will try to get the specifications translated to him in natural language, or base his understanding of what is required on some examples provided with the specifications. Both approaches are fraught with possibilities of error: the former vitiates the formality of the specification and may introduce errors in the translation; the latter may not cover all possible cases. So, it is not sufficient for the specifications to be thoroughly checked and written out formally: they must also be lucid. It is not easy to be formal and lucid at the same time.

A second factor affecting diversity is the programming language. The nature of the language affects greatly the programming style: a program written in FORTRAN is likely to be structured very differently from one in C. It is not very unlikely that two programmers working independently to the same set of specifications in the same language will come up with similar software. One example will suffice to drive home this point. When they were developing the Unix system, Thompson and Ritchie of the Bell Telephone Laboratories would parcel out the job of developing various subroutines and procedures amongst themselves. Once, as a result of a mistake, each person was assigned the same procedure to write. When they compared these independently-written versions of the same procedure, they were identical, line by line! Writing programs in the same language increases the probability of common-mode failures. If the same compiler is used, that is another possible source of common-mode failures.

A third factor is the numerical algorithms that are used. Algorithms implemented to a finite precision can behave quite differently for certain sets of inputs than theoretical algorithms which assume infinite precision. If the teams are implementing the same algorithm, they will duplicate the same numerical instabilities.

A fourth factor is the nature of the tools that are being used. If the same tools (e.g., static and dynamic analyzers, expert systems to aid debugging, etc.) are being used, the probability of common-mode failure might increase.

A fifth factor is the training and quality of the programmers and the management structure. If the programmers have identical or similar educational backgrounds, they can be expected to make similar mistakes. If they have a low level of skill, that can translate not only into an increased incidence of errors, but also into more common-mode failures due to similar incorrect interpretations of what is required. If they have a great deal of experience of similar projects, they may well carry over in their minds, some aspects of the specifications of those other projects. The management style can also affect the quality of the code. Some managers might insist that the programmers who develop the code not be the ones to test it, and indeed that the authors of the code not be allowed to directly contact the testing team. Other management approaches may consist of allowing the programmers to work together in informal structures, where tasks are not well-defined and tests are carried out informally.

There does not yet exist any reliable way of quantifying diversity, except to subject the programs to testing and checking whether they fail on common inputs. This is, however, an area of continuing research. For example, the ESPRIT software project supported by the European Union, is developing a diversity assessment technique based on fuzzy logic.

There is very little available data on the incidence of common-mode failures in multiple versions. The major difficulty in arranging large-scale experiments is cost: software is labour-intensive. A small number of experiments have been carried out for this purpose. However, they have all been done in the academic environment, and it is open to debate as to how representative the results might be to software developed under industrial conditions. Let us consider one such experiment, carried out at the University of Virginia (UVA) and the University of California, Irvine (UCI), by Knight and Leveson.

The application was a simple antimissile system. The software was meant to receive radar input and judge whether the signals indicated any incoming missiles. The specifications were written in English and pseudocode. The programmers were a mix of undergraduate and graduate students with varying levels of computer experience. The software was written in Pascal. The compilers used at the two universities were different, although all programmers at the same university used the same compiler. Contact amongst the teams was discouraged, and questions about the specifications were to be submitted and answered by electronic mail. This allowed a record to be kept of all interactions. Any specification errors that were uncovered were announced to all the programmers, also by electronic mail.

In all 27 versions were written. They were tested to indicate their reliability. To do this, the versions were tested against each other for the same input sets, and against a program written to the same specifications as part of an earlier NASA experiment. This last program had been extensively tested and was believed to be highly reliable. The only errors that would escape notice would be those which caused identical errors in all 28 versions (the 27 student-written plus the NASA version). Table 7.3 indicates the extent of the correlated failures between the versions written at UVA and UCI. For example, there were 323 instances where Versions 8 and 20 failed on the same input.

7.10.2 Recovery Blocks

As in N-version programming, multiple versions are used in the recovery block approach. There is a primary version and one or more alternatives. Unlike N-version programming, however, only one version is run at any one time. This is a backward error recovery scheme.

		UVA Versions								
		1	2	3	4	5	6	7	8	9
	10	0	0	0	0	0	0	0	0	0
	11	0	0	58	0	0	2	1	58	0
	13	0	0	1	0	0	0	71	1	0
	14	0	0	28	0	0	3	71	26	0
	15	0	0	0	0	0	0	0	0	0
	16	0	0	0	0	0	1	0	0	0
	17	2	0	95	0	0	0	1	29	0
UCI	18	0	0	2	0	0	1	0	0	0
Versions	19	0	0	1	0	0	0	0	1	0
	20	0	0	325	0	0	3	2	323	0
	21	0	0	0	0	0	0	0	0	0
	22	0	0	52	0	0	15	0	36	2
	23	0	0	72	0	0	0	0	71	0
	24	0	0	0	0	0	0	0	0	0
	25	0	0	94	0	0	0	1	94	0
	26	0	0	115	0	0	5	0	110	0
	27	0	0	0	0	0	0	0	0	0

Table 7.3: Correlated failures between UCI and UVA Versions (From J. Knight and N. Leveson, “An Experimental Evaluation of Independence in Multiversion Programming,” *IEEE Trans. Software Engineering*, Vol. SE-12, No. 1, January 1986. ©IEEE 1986. Reprinted with permission.)

We have seen a schematic diagram of the recovery block approach in Figure 7.14. The primary software is run in the first instance. Its output is passed through an acceptance test. This test is supposed to indicate whether the output is acceptable or not, and is the weakest point in the entire design. For, the acceptance test has no prior way of knowing what the correct output should be (if it did, there would be no need to run the primary). It makes "sanity" checks: these generally consist of making sure the output is within a certain acceptable range, or that the output does not change at more than the allowed maximum rate. For example, if the task is one of position calculation in an ship, any output that claims that the ship is 10,000 miles away from where it was computed to be a few milliseconds ago is clearly wrong. These ranges and rates are functions of the application, and must be specified by the designer.

An alternative need not always use the same inputs as the primary: it may use other approaches to carry out the computation[7]. As we know, much of the load in real-time systems consists of the same tasks being repeatedly executed. If the ith iteration has failed to produce acceptable output, or has not terminated within some prespecified time, the alternative is invoked. If this also fails, there may be yet another version that can be invoked. The system can keep trying until one of the following happens:

- One of the versions passes the acceptance test.
- The system runs out of versions.
- The deadline is missed.

Let us now turn to an example of how acceptance tests may be determined.

Example 2 *Ships compute their position by timing inputs from the Global Positioning System, which consists of a constellation of satellites in earth orbit. We know the position of the ship a few milliseconds (or seconds) ago as a result of a prior iteration of the position-calculation algorithm. We also know the ship's speed and its heading. Based on that, we can clearly estimate the current position (i.e., by dead reckoning) within some range. To pass the acceptance test, the output of a version must be within this range.*

[7] Indeed, on the theory that increased complexity gives rise to an increased probability of faults, we may want the alternative modules to be simpler than the primary so that they are less prone to produce errors. We may have to pay for this by allowing the alternatives to produce outputs of lower quality than the primary (e.g., outputs which are not quite so accurate, but still acceptable).

Setting the acceptance tests places the designer in the following dilemma. If the allowed ranges are set too strictly, the acceptance tests will generate a lot of false alarms (by labelling as bad output which happens to be correct). If they are set too loosely, the probability will be high that incorrect outputs are accepted as good. Setting the acceptance tests is an art, and the state of that art is very unsatisfactory at the moment.

When the acceptance test fails one version and invokes another, all global state changes made by the failed version must be reversed. This can be done using a *recovery cache*, and is further discussed in Section 7.11.

There is, of course, a time overhead incurred by using recovery blocks. This overhead occurs when the primary has failed, and consists of reversing the global state changes and running one or more alternatives.

7.11 Implementing Backward Error Recovery

Backward error recovery can take multiple forms. The simplest is *retry*, where the failed instruction is repeated. Other options include rolling the affected computation back to a previous checkpoint and continuing from there, or restarting the computation all the way from its beginning.

Critical to a succesful implementation of backward error recovery is the restoration of the state of the affected processor or system to what it was before the error occurred[8]. By doing so, we are, in effect, wiping out all traces of the faulty program or process. Corrective action (e.g., assigning another processor to carry on with the exection beyond this point, or retrying on the same processor with the corrected state information) can then be taken.

7.11.1 Recovery Points

One way of implementing backward error recovery is to store the process state at prespecified moments in time. Such snapshots are called *checkpoints*. Figure 7.15 provides a simple example. There are three checkpoints, taken at *recovery points* $R1, R2, R3$, i.e., points to which we want to be able to roll

[8] By “state” we mean all the information that enables us to restart the process from that point on. Typically, it includes the values of all the registers and the working set in memory.

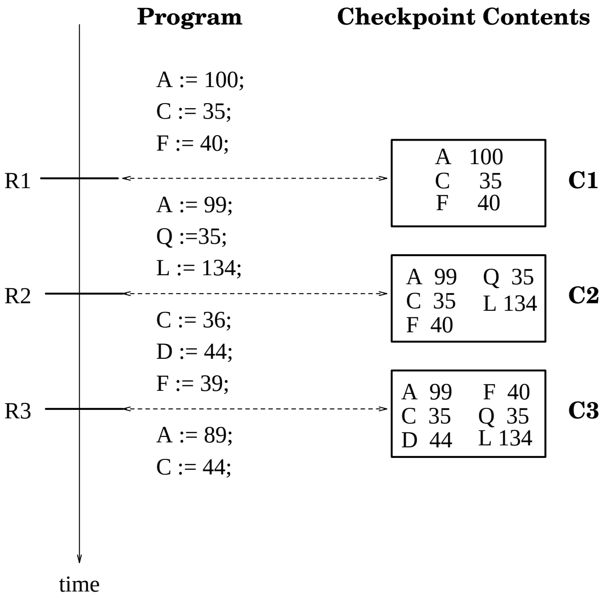
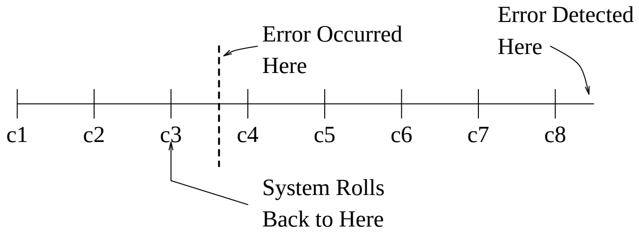


Figure 7.15: Example of checkpointing a program

back the process. If an error is identified, state restoration is done by simply reading the latest checkpoint before the error is known to have occurred.

In some instances, if we don't know exactly when the error occurred, we might have to roll back all the way to the earliest checkpoint that we have. See Figure 7.16 for an example. An error in a processor occurs between checkpoints c_3 and c_4, and is detected after c_8. *If we know at that time that the error occurred between* c_3 *and* c_4, the proper action would be to roll the processor back to the latest checkpoint preceding the onset of the error, i.e., to c_3. This presupposes two things: first, that we have enough memory available to keep checkpoints c_3 to c_8; second, that we know when the error occurred. Most often, neither condition is satisfied. Due to a shortage of memory, typically only the last one or two checkpoints are kept. Also, in general, there is no way to know when the error occurred, unless we have some information – from the nature of the error, or any acceptance tests that may have been passed, or some other means – to guide us. In that case, we can roll back to the oldest checkpoint that is stored, and hope that the error did not occur prior to that time. Otherwise, we will have to roll all the way back to the starting point of the computation, i.e., we will have to *restart* the computation.



c_i are the checkpoints.

Figure 7.16: Use of checkpoints

The designer must decide when checkpoints are to be taken, by balancing the cost against the benefits. The cost is in terms of the quantity of memory that is required to store the state (this depends on the system), and the time taken to store the checkpoints. The benefit in having many checkpoints taken at short intervals is that the extent of the rollback can be limited. Sometimes there are places in the code that are natural points for checkpointing. For example, if it is possible to have an acceptance test of certain outputs, one can checkpoint just after these outputs have been produced, after they have passed the acceptance test. Another way is to use redundancy and voting to check that the output is correct before the state is saved to form the checkpoint. This way, we have some confidence in the quality of the checkpoint.

Since each time we checkpoint, we must make a copy of the entire process state, checkpointing is expensive in memory and time. It is wasteful because unnecessary copies of the same variable value might be made. This expense can be reduced by taking *incremental* checkpoints. The idea is to only record a variable value when it is going to be changed. If this is done, however, we cannot simply discard old checkpoints, since some of the information they contain may still be relevant. A mechanism to incrementally checkpoint is the *recovery cache*[9].

Figure 7.17 illustrates how this works. Associated with each recovery point is a recovery cache. The recovery cache contains the value, at the associated recovery point, of the variables that are changed before the next recovery point. For example, the only variable that is changed between $R1$ and $R2$ is A, and so the recovery cache for $R1$ only contains the old value of

[9] The term “cache” as used here is unfortunate, because it has nothing to do with the standard computer-architecture meaning of cache. The word was apparently chosen to convey its original dictionary meaning of a safe place (to put the recovery information).

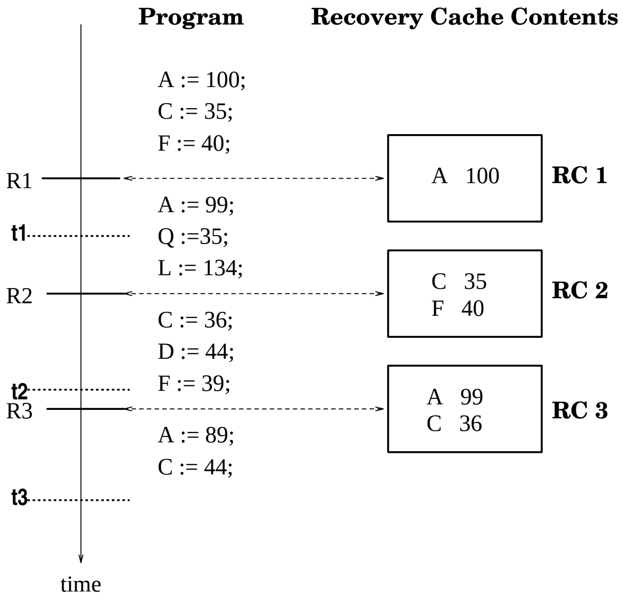


Figure 7.17: Use of recovery caches

A.

Let us consider how the recovery cache contents can be used to restore the state to the recovery points. For example, consider that a failure is detected at $t2$, and the system decides to roll back to $R1$. To restore the state to what it was at $R1$, (with values 100 in A, 35 in C, and 40 in F; the values of Q, L, D are irrelevant), we set C to 35 and F to 40 from RC 2. Then, we use RC 1 to set A to 100. The other variables, Q and L are not relevant since they are not defined at $R1$.

Suppose instead that the failure was detected at $t1$, and the system decides to roll back to $R1$. RC 1 is used to set A to 100. Nothing need be done to C and F since they still have the same values as they did at $R1$.

It is clear that the recovery cache saves memory. Does it necessarily also save time? We do save time in not having to save as many things as in the standard checkpointing scheme. However, whenever the program changes a variable value, it will have to check if this is the first change in that variable since the last recovery point. If so, the old variable value (i.e., the one current at the last recovery point) needs to be saved in the appropriate recovery cache. This check can be done in two ways. First, the system can

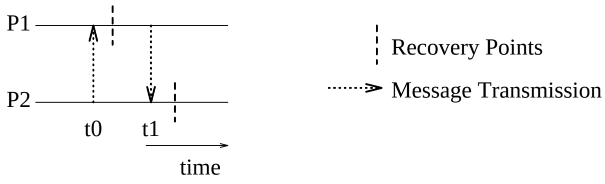


Figure 7.18: The domino effect

check the recovery cache to see if a value has been saved for that variable: if so nothing needs to be done; if not, we need to save it. This takes a great deal of time, involving as it does an additional read (or two, depending on the memory access system) for each variable update. The second way is to associate a flag with each variable, which is set whenever that variable has been saved in the current recovery cache. Note that these flags will all have to be reset every time a recovery point is encountered.

Discarding a recovery point is also more complicated than in checkpointing[10]. Discarding a checkpoint only requires us to throw away its contents. We cannot *always* do that when using recovery caches since each recovery block may contain unique information. For example, suppose we decide at $t3$ to get rid of recovery point $R2$, keeping $R1$ and $R3$. Before discarding RC 2, we will have to store the values of 35 for C and 40 for F in RC 1. On the other hand, if we decide to get rid of $R1$, i.e., the oldest extant recovery cache, RC 1 can simply be deleted.

Thus far, we have assumed that each process is independent, i.e., it does not work in cooperation with any other process. If this is not the case, and processes interact, things become more complex. Consider what happens in Figure 7.18. There are two processes, $P1$ and $P2$, which communicate as shown. An error is discovered after $t1$, which forces $P1$ to roll back to its previous recovery point. However, to undo this, the effects of the message from $P1$ to $P2$ must also be undone. The only way to do this is to roll back $P2$ to its previous checkpoint, which happens to be, in this example, the beginning. If $P2$ is to be rolled back, everything it did must be undone. This includes its message to $P1$ at time $t0$. But this causes everything done by $P1$ after $t0$ to be undone, and $P1$ must roll back to the latest checkpoint preceding $t0$, which happens in this example to be the beginning of the execution. So, both $P1$ and $P2$ are rolled back to the beginning. This is an

[10] We have to regularly discard recovery points, otherwise we will run out of memory.

example of how rollbacks can propagate, and is called the *domino effect*.

The domino effect can be countered by insisting that all the processes have their recovery points or checkpoints at the same time. The proof of this is left as an exercise to the reader.

7.11.2 Audit Trails

A second way of enabling backward error recovery is through *audit trails*. These are especially popular in databases. An audit trail consists of a record of all the actions that have been taken by the system, together with a timestamp indicating when each action was taken. Backward recovery to some time t is effected by undoing actions taken after time t and then restarting from that point.

7.11.3 Irrecoverable Actions

We have hitherto assumed that all actions are reversible. This is the basis of backward error recovery. However, some actions are not reversible since they are out of the control of the computer system. Two examples will suffice. Suppose a process has had something printed on a line printer. The computer has no way of unprinting it. (While it may be possible to print a correction message, the computer has no way of ensuring that the correction has been put out in time for the outside world to cancel any actions taken as a result of the printout). Or, consider an anti-ballistic missile system which, on the mistaken belief that it is under missile attack, launches an anti-missile missile. Even if it later discovers the error, it will be impossible for the computer to unlaunch the missile.

7.12 Data Diversity

Data diversity is an approach which can be used in association with any of the redundancy techniques considered above. The idea behind it is as follows. Sometimes, hardware or software may fail for certain inputs, but not for other inputs which are very close to them. So, instead of applying the same input data to the redundant processors, we apply slightly different input data to them, we have – in some cases – another line of defence against failure.

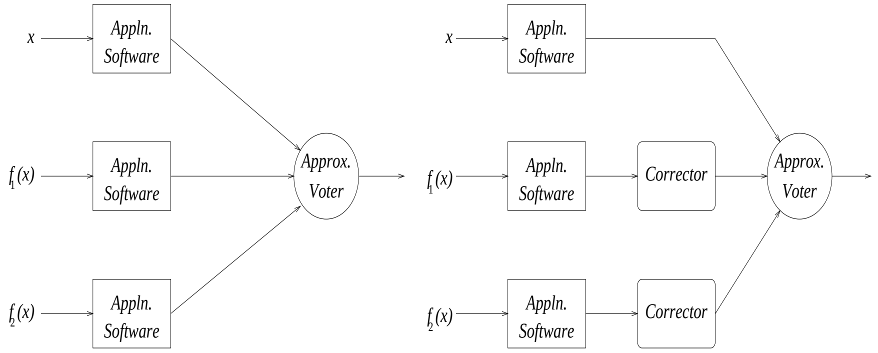


Figure 7.19: Use of data diversity with voting

This approach will only work if the sensitivity of the output is either very small with respect to small changes in the input, or if perturbing the output can be corrected analytically. Let us look at each case separately.

If the output changes very little as a result of changing the input slightly, then we can perturb the input and carry out approximate voting on the results, to obtain an approximate, but still acceptable, output. For example, instead of applying the same input x to all members of an NMR cluster, we apply $x, f_1(x), \cdots, f_N(x)$. Let $P(y)$ be the output of the program when the input is y. Then, the functions f_i must be chosen in such a way that $P(x) \approx P(f_i(x))$.

The second case is when perturbing the input causes a change in the output which can be forecast analytically. This change can then be corrected. That is, suppose we can obtain a function $Q(x, y) = P(x) - P(y)$ for x, y sufficiently close to each other. Then, $P(x) = P(f_i(x)) - Q(f_i(x), x)$. Figures 7.19 illustrates these concepts where voting is used.

Data diversity can also be used with recovery blocks: instead of running alternative software when the primary fails the acceptance test, the primary can be reinvoked with slightly different data.

7.13 Reversal Checks

If there is a simple relationship between the inputs and outputs of a system, it may be possible to calculate the inputs given the outputs. This can be compared with the actual inputs as a check. For example, consider a task which finds the square root of a number. To see if this is correct, one can

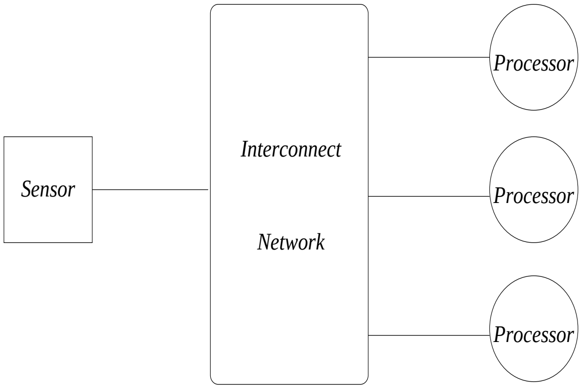


Figure 7.20: Distribution of sensor information

square this output and check it against the original input. Or, let the task consist of writing a block onto disk. The reverse operation would consist of reading this block from the disk after writing, and comparing to make sure that the two are the same. Reversal is a powerful method, but its applicability is limited to tasks where it is relatively easy to carry out the reverse computation.

7.14 Malicious or Byzantine Failures⋆

Whenever a failure can cause a unit to behave arbitrarily, *malicious* or *Byzantine* failure is said to happen. We have already seen an example of malicious failures, where a sensor sends out conflicting information to different processors.

For correct operation, it is often the case that copies of the same data as seen by various processors must be consistent, i.e., the same. When communication is limited to two-party messages, the faulty units must be fewer than a third of the total number of units if consistency is to be guaranteed. This may come as a surprise to those who, conditioned by the majority voting covered in earlier sections, may believe intuitively that a system can maintain consistency so long as only a minority of units are faulty.

Figure 7.20 illustrates a situation where consistency is important. Data originating from a sensor are to be distributed amongst the processors shown. If the sensor is nonfaulty, we would like all the functioning processors to read the sensor output consistently, i.e., all the processors see the same data value.

If the sensor is faulty, we would like the processors also to agree on some input from the sensor (which may be a default value if they realise that the sensor is faulty).

Let us begin with what has become the canonical introductory example for the field of Byzantine failures, by showing that three units are insufficient to maintain consistency in the face of one malicious failure.

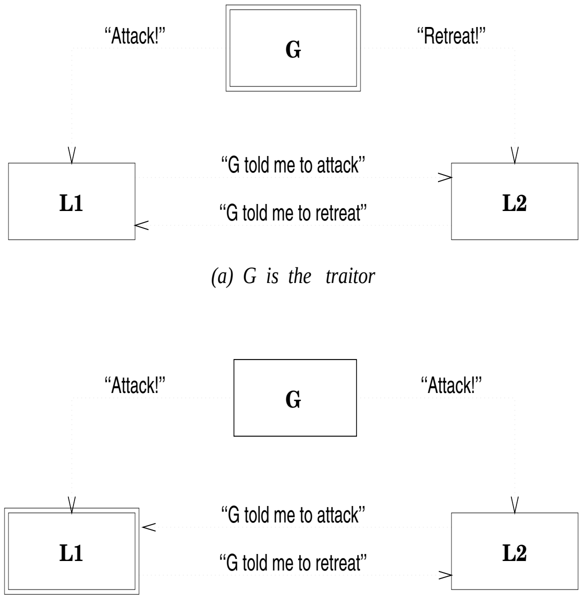
Example 3 *Consider an army of the Empire of Byzantium (here is the genesis of the term "Byzantine" for such faults) engaged in besieging a city. The army consists of two divisions, each with its own divisional commander, L1 and L2. There is a supreme commander of the assault, G, and L1 and L2 function as his lieutenants. The three are physically separated, and can only communicate by means of oral two-party messages, conveyed by messenger. At most one of the three commanders can be a traitor. The messengers are assumed to loyal.* Interactive Consistency *will be maintained if the following two conditions are satisfied:*

- *IC1: All loyal lieutenants obey the same order.*
- *IC2: If G is loyal, then every loyal lieutenant obeys the order he sends.*

Both conditions will obviously be satisfied if everyone is loyal. Let us consider what happens if (a) *G is a traitor, and* (b) *if one of L1 and L2 is a traitor. In each case, we will show that the traitor can force the consistency conditions to be violated. In each case, we assume that G, L1, and L2 have synchronized their watches, and that G must give an order ("attack" or "retreat") by a certain time.*

Suppose G is a traitor, while L1 and L2 are loyal to the Empire. Let him send conflicting orders to L1 and L2: telling L1 to attack and L2 to retreat. If L1 and L2 do not communicate, neither knows of the conflict, and being loyal obey his orders. Since these orders are conflicting, IC1 is violated.

Suppose L1 and L2 communicate with each other by sending to each other the order they received from G. L1 tells L2 that G told him to attack, while L2 tells L1 that G told him to retreat. Now consider what L2 should do. Should he believe L1 or G? One of them is clearly a traitor since there is a conflict. If L1 is telling the truth, then he is accurately conveying to L2 the fact that G has sent conflicting orders and is therefore a traitor. If L1 is a traitor, he could be lying to L2 about the order he received from G. So, in the absence of any further information, L2 cannot decide whether G is loyal.



(a) G is the traitor

(b) L1 is the traitor

Figure 7.21: Effects of treason on consistency

There is no way for him to act in such a way that the consistency conditions are satisfied.

Now, consider the case that G is loyal, but L1 is a traitor. By lying to L2, he can construct – from L2's point of view – a similar conflict that shows L2 that either L1 or G is a traitor, but he does not know which. Figure 7.21 summarizes this. Note that even if L1 and L2 each refer this discrepancy back to G, G can do nothing about it. If G is a traitor, he will then send L1 and L2 each a message, confirming his original orders, and telling L1 that L2 must be the traitor and vice versa. If G is loyal, he will reiterate his prior messages to L1 and L2. Either way, L2 will see exactly the same set of messages, regardless of whether G or L1 is the traitor. This proves that it is impossible to guarantee satisfying the conditions IC1 and IC2 with three generals and one traitor.

Let us now consider the case where the Byzantine army has three *divisions, each with its own commander. These are lieutenants L1, L2, L3, of the commanding general, G. Suppose once more that at most one of them can possibly be a traitor. Let us consider the case where G is the traitor and L1, L2, and L3 are loyal officers, trying to maintain the consistency* conditions IC1 and IC2. Suppose G sends out inconsistent orders: "attack" to L1 and L2, and "retreat" to L3. As before, the messages are two-party oral messages, conveyed by loyal messengers. To describe what ensues, we need some notation. Let X1.X2.X3...Xn(a) denote the event that Xn-1 tells Xn that Xn-2 told Xn-1 that ... that X1 told X2 the value a. For example, G.L1.L2.L3(attack) means that L2 told L3 that L1 told L2 that G told L1 to attack. Consider the messages that can be sent back and forth after G gives his inconsistent orders:

	Messages	
Received by L1	Received by L2	Received by L3
G.L1(attack)	G.L2(attack)	G.L3(retreat)
G.L2.L1(attack)	G.L1.L2(attack)	G.L2.L3(attack)
G.L3.L1(retreat)	G.L3.L2(retreat)	G.L1.L3(attack)

Simply by taking a majority vote on these messages, L1, L2, and L3 can behave consistently (they attack), satisfying IC1 and IC2. In addition, L3 knows G is a traitor since there is at most one traitor, and so L1 and L2 cannot both be traitors (he knows that he himself is loyal). However, L3 has no way of convincing the others of this based on two-party messages (since as far as the others are concerned, L3 could be a traitor himself!).

Consider what happens if L1 is the traitor, and G, L2, and L3 are loyal. As before, the lieutenants compare notes about what they heard from G. Note that L1 can behave arbitrarily, by lying about what he heard from the others. However, despite this, the consistency conditions will continue to be satisfied: every message which does not pass through L1 will be the same. There are enough of such messages to override anything that L1 might do. Consider the messages that flow after G gives his orders. In the following table, m_1 and m_2 refer to the messages emanating from L1: each of these can be either "attack" or "retreat."

	Messages	
Received by L1	Received by L2	Received by L3
G.L1(attack)	G.L2(attack)	G.L3(attack)
G.L2.L1(attack)	G.L1.L2(m_1)	G.L2.L3(attack)
G.L3.L1(attack)	G.L3.L2(attack)	G.L1.L3(m_2)

L2 thus receives the following messages: attack, attack, m_1. L3 receives these messages: attack, attack, m_2. Taking a majority vote results in both L2 and L3 attacking, as required, irrespective of what m_1 and m_2 may be.

Traitors are difficult to handle because they can lie about messages that were received. Suppose, instead, that *message authentication* is possible. For example, each sender can affix an unforgeable signature to a message. Any lieutenant who passes on the order he got to a colleague must do so by copying the message he got, with the signature on that message, and then adding his own signature. Any attempt to alter the message or the signature can be detected by anyone. As a result, nobody can lie without being found out, and so it will be impossible for a traitor to cause the consistency conditions to break down.

Example 4 *Let us return to the case in Figure 7.21. Consider what happens when G is a traitor. He sends out signed messages to L1 and L2, saying "attack" and "retreat" respectively. L1 sends L2 a copy of the message he received from G, including G's signature, and vice versa. L1 knows that L2 cannot be lying, since he sees G's signature on the order to retreat, and so he knows that G did send out contradictory orders, and is therefore the traitor. Similarly, L2 also knows that G is the traitor. In such a case, both L1 and L2 can take some default action, and the conditions of consistency will be satisified. The case where L1 is a traitor is similar.*

We can now state, without proof, the key result concerning malicious failures.

Theorem 1 *If messages cannot be authenticated, then to sustain the conditions of consistency IC1 and IC2 in the face of up to m traitorous generals requires a total of at least $3m + 1$ generals.*

The field of Byzantine generals algorithms has become something of a cottage industry, with many new and improved (in terms of speed and memory requirements) algorithms appearing regularly in the literature. Here, we provide what is perhaps the simplest such algorithm.

The algorithm, Byz(m), is recursive, and is a function of m, which is the specified maximum number of traitors that must be sustained. In the following, when we say that x sends a message to y, we assume either that such a message is received by y, or that y (upon a timeout after receiving no such message) assumes that x transmitted a default value (say "retreat").

Algorithm Byz(0)

1. The commander sends his order to every lieutenant.

2. The lieutenant uses the order he receives from the commander, or the default, say "retreat," if he receives no order.

Algorithm Byz(x)

1. The commander sends his order to every lieutenant. Let v_i be the order he sends to lieutenant Li.

2. For each i, lieutenant Li acts as the commander in an Byz(x-1) algorithm, and sends out the order v_i to each of the $N-2$ other lieutenants. That is, the $N-2$ other lieutenants try to obtain consensus on lieutenant Li's order, for every i.

3. For each i, and $j \neq i$, let $w_{i,j}$ be the order that lieutenant Li received from Lj in step (2) (using the Byz(x-1) algorithm), or the default, say "retreat," if he received no such order. Lieutenant Li follows the order majority$\{v_i, w_{i,j}, j \neq i\}$.

This is the same algorithm used in the 4-general example considered earlier.

Example 5 *Consider Byz(2) run on a 7-general system. In the first round, the commander sends his order to each of his generals. In the second round, each of these generals distributes to his colleagues the value he received from the commander using the Byz(1) algorithm. Once this has been concluded, each general can calculate the order that he is to follow.*

Let us now prove that this algorithm satisfies the interactive consistency conditions.

Theorem 2 *If there are at most m traitors and a total of $N \geq 3m+1$ generals in all, algorithm Byz(m) guarantees interactive consistency.* **Proof:We work by induction on m. If $m = 0$, i.e., there are no traitors, then the algorithm obviously works. This will form the induction basis.**

Assume that the algorithm satisfies conditions IC1 and IC2 for up to $m-1$ traitors. There are two cases:
Case 1: The supreme commander is loyal: **We must show that IC2 is satisfied, i.e., that all the loyal lieutenants follow their commander's order.**

Since the commander is loyal, all the lieutenants receive the same order, say v. Each loyal lieutenant then acts as the commander for a Byz(m-1) algorithm (step 2 of the Byz(m) algorithm for

$m \geq 1$). There are at least $2m$ loyal lieutenants.
Case 1.1: **$m = 1$*:* In such a case, every loyal lieutenant simply sends out the order he received to every other lieutenant.**
Case 1.2: **$m > 1$*:* If $N \geq 3m + 1$, $N - 1 > 3(m - 1) + 1$, so by the induction hypothesis, in Step 2 of Byz(m-1) each loyal lieutenant agrees on the order, v, sent out by every other loyal lieutenant. Now, since the commander is loyal, the the majority vote in Step 3 of Byz(m) results in the order v being carried out by each loyal lieutenant.**

Case 2: The supreme commander is a traitor: **Since the commander is one of the up to m traitors, up to $m - 1$ of the lieutenants can be traitors, and the others must be loyal. There are at least $3m$ lieutenants, of whom up to $m - 1$ are traitors. $3m - 1 > 3(m - 1) + 1$. Hence by the induction hypothesis, we conclude that each invocation, by a loyal lieutenant, of Byz(m-1) in Step 2 of Byz(m) satisfies the interactive consistency conditions. Consequently, each loyal lieutenant sees the same vector of values sent by every other lieutenant in Step 2 of Byz(m). They therefore carry out the same order, and the theorem is proved.**

The analogy between the Byzantine Empire and the world of computing should be apparent: traitorous generals are analogous to processors which have suffered malicious (or Byzantine) failure, and loyal generals are analogous to functioning processors. "Orders" are analogous to some variable value on which consistency is to be obtained. Let us now consider the relevance of all this to real-time computing.

The Byzantine generals algorithm is important whenever we have a single sensor sending out variable values to processors in a redundant cluster which then vote on the result, using the schemes considered earlier in this chapter. (The analogy here is sensor=supreme commander; processor=lieutenant). For majority voting schemes to work, the inputs to the processors should be consistent. If the sensor can suffer Byzantine failure, then a Byzantine generals algorithm is required to obtain consistent values for the processors.

The Byzantine generals algorithm will also work with other kinds of voting, rather than majority voting (in Step 3). The details of this are left as an exercise.

Note that throughout we have assumed that the participants in this algorithm are expecting a message. This assumes either a synchronized clock or a watchdog timer before the expiry of which a message is expected. This is necessary, else a traitor could cause everyone to wait forever by simply not providing any output.

A problem similar to the one we have considered arises when trying to synchronize clocks in the face of some failures. This is a problem of great importance in real-time computing, and is covered in Chapter **??**.

7.15 Information Redundancy

In this section, we consider coding to detect or correct errors. There has been a tremendous amount of research in coding in recent years, and it finds use in both computing and communication networks. Since the coding used in real-time systems is no different from that used in general-purpose computers, we only provide a brief and partial survey.

The basic idea of information redundancy is to provide more information than is strictly necessary, and to use that extra information to check for errors. We use coding all the time ourselves, while correcting for typographical errors. For example, if you encounter the word "startegic," you will most likely unconsciously correct it to "strategic." This was possible because *(a)* there is no such word as "startegic," and *(b)* "strategic" is the closest word that one can think of to "startegic." If, on the other hand, there was a misprint "taut" instead of "taught," you would only be able to discover the error by reading the context, since "taut" is as much a word as "taught."

The conditions *(a)* and *(b)* are at the basis of all coding theory. All computer words are strings of 0's and 1's. Coding ensures that not all strings of 0's and 1's are legal (i.e., are valid). An illegal combination of bits indicates an error. Sometimes there is an unambiguous "nearest" valid word as which the word can be read, thus correcting the error. Here, "nearest" is computed by defining the distance between two words as the number of bit positions in which they differ. This is called the *Hamming distance*, H_d. The greater the Hamming distance between valid combinations (code words), the more errors can be corrected or detected. In general, a code can correct up to c bit errors and detect up to d additional bit errors iff $2c + d + 1 \leq H_d$: we will omit the proof for this.

When assessing a coding scheme, we want to know how many extra bits it adds to the words, and how many bit errors it can detect or correct. We are also interested in how much work it takes to encode and decode.

A code may be *separable* or *nonseparable.* In a separable code, the coded word consists of the original (uncoded) word, concatenated with a number of code bits. The process of decoding to get the original word back is thus limited to stripping away the code bits. A nonseparable code does not have this property of separability, and is more complex to decode.

7.15.1 Duplication

The simplest code of all is *duplication.* Each word is duplicated. An error in a bit position is detected by the discrepancy between the bit and its duplicate. Note that it is not possible to *correct* any errors: if a discrepancy is noted, we have no way to know whether it is the bit or the duplicate that is in error. While this code is simple, it requires 100% redundancy, i.e., the coded word is twice the length of the uncoded word.

7.15.2 Parity Coding

Another simple, and widely used code is *parity.* Parity coding, or variations of it, is widely used in memory chips. It consists of adding an extra bit (called the parity bit) to each word to ensure that the number of 1's in it is always even (even parity) or odd (odd parity). Parity has a Hamming distance of two. For example, if even parity was being used, the word 001101 would have its parity bit set to 1 so that the word would now be 0011011. It is possible to detect all one-bit errors using this code. For instance, if through some faults, the word were to be recorded as 0011010, we would know by counting the 1's, that there was an error in that word. However, we couldn't correct that error, since violation of parity may have been caused by any of the bits being wrong. Also, we cannot always detect errors if two or more bits are wrong. For instance, if the word 0011011 became 0000011, the parity rule would be satisfied, and the errors would not be detected. Parity is widely used because of its simplicity, and the small overhead it imposes.

The basic parity rule can be varied to provide additional protection against errors. Instead of having just one parity bit per word, we can subdivide the word into portions, and associate a parity bit with each portion. Another interesting concept is *interlaced* parity, which has the ability to correct some bit errors, rather than just detect them. Here again, the word is

Bit Error	Leads to Error in Parity Bit			
	P_3	P_2	P_1	P_0
w_0	×	×	×	×
w_1	×	×	×	
w_2	×	×		×
w_3	×		×	×
w_4	×	×		
w_5		×	×	
w_6			×	×
w_7		×		×
P_0				×
P_1			×	
P_2		×		
P_3	×			

P_0 covers bits w_0, w_2, w_3, w_6, w_7, P_0.
P_1 covers bits w_0, w_1, w_3, w_5, w_6, P_1.
P_2 covers bits w_0, w_1, w_2, w_4, w_5, w_7, P_2.
P_3 covers bits w_0, w_1, w_2, w_3, w_4, P_3.

Table 7.4: Example of overlapping parity with 8 information bits

divided into portions and a parity bit is associated with each portion. However, in interlaced parity, the portions are not disjoint, but overlap. Table 7.4 provides an example for the case where there are eight bits, $w_7 \cdots w_0$, in the uncoded word. The idea is for a unique combination of parity bits to be in error for any single bit error. Whenever that combination occurs, we can use the table to determine which bit is in error. How many bits do we need for this scheme? If there are n bits of the uncoded word (called the *information bits*), and c parity bits, we must have the ability to distinguish between errors in $n + c$ positions. We must also be able to determine that there is no single-bit error. This makes a total of $n + c + 1$ cases to distinguish. So, choose c such that $n + c + 1 \leq 2^c$.

The same principle can be extended to obtain a code that will correct multiple bit errors. The details are left to the exercises.

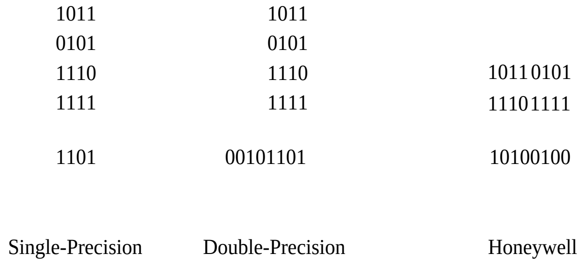


Figure 7.22: Illustrating three types of checksum on 1011, 0101, 1110, and 1111

7.15.3 Checksum Codes

The checksum code is used when blocks of data are being transferred. Suppose that we are transferring a certain number of n-bit words. The sender computes the checksum, by adding together these words, and transmits it along with the words. The receiver computes the checksum of the words that it receives, and compares it with the checksum obtained by the sender. If the two sums do not match, an error is detected. A checksum code can detect, but not correct, errors.

There are variations on the checksum, depending on how the addition is carried out. If the addition is modulo-2^n (with the overflow out of the nth bit being ignored), it is called the *single-precision checksum*. If it is modulo-2^{2n} (with the overflow out of the $2n'$th bit being ignored), it is called a *double-precision checksum*. The *Honeywell checksum* works by concatenating two words to form words of $2n$-bit length, and then carries out a modulo-2^{2n} addition on these to form the $2n$-bit checksum. Figure 7.22 illustrates the various checksum codes for $n = 4$. The double-precision checksum captures some errors that are lost by its single-precision counterpart for obvious reasons. The Honeywell checksum is useful in detecting errors that occur consistently in the same bit position: this will affect at least two bit positions of the checksum.

7.15.4 Cyclic Codes

A cyclic code is so called because any cyclic shift of a valid code word will produce another valid code word. Cyclic coding can be implemented with shift registers and exclusive-or gates.

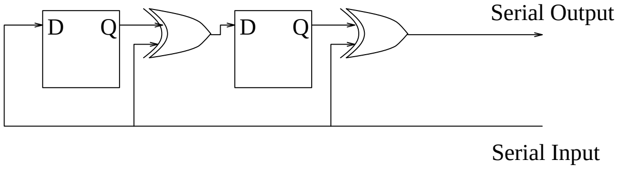


Figure 7.23: Coding with the generator polynomial $1 + X + X^2$.

Fundamental to cyclic coding is the interpretation of bits as the coefficients of a polynomial. This is not as unnatural as it might seem at first: a moment's reflection shows that all the numbers that we encounter are really coefficients of polynomials which represent their value. For example, consider the number 1101. Its value, if the number is in binary, is the polynomial $1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$. If the number is in radix-r, its value is the polynomial $1 \cdot r^0 + 0 \cdot r^1 + 1 \cdot r^2 + 1 \cdot r^3$. When we multiply two numbers, we are really multiplying the underlying polynomials.

The polynomial $G(X) = a_0 + a_1 X + \cdots a_n X^n$ represents the word $a_n \cdots a_1 a_0$. For example, the polynomial $1 + X + X^5$ represents the word 100011 (the 0-coefficients terms are traditionally dropped from the polynomial representation). A polynomial of order n represents an $n + 1$-bit number.

Cyclic coding is carried out by multiplying the word to be coded by a polynomial, called the *generator* polynomial. *All additions in this process are modulo-2.* Multiplication by X^n essentially means shifting by n places. To see this, suppose we multiply the polynomial $1 + X + X^5$ (representing the word 100011) by X^2 (representing the word 100). The product is $X^2 + X^3 + X^7$, which represents the word 10001100: the original number 100011 has been shifted two places.

Let us consider a more complex multiplication. Multiply the word to be coded, $1 + X + X^5$ (representing the number 100011) by the generator polynomial $1 + X + X^2$ (representing 101). We have $1 + (1 + 1)X + (1 + 1)X^2 + 1X^3 + 0X^4 + 1X^5 + 1X^6 + 1X^7$. Doing the additions modulo-2 (which means putting them through exclusive-OR gates) results in $1 + 0X + 0X^2 + 1X^3 + 0X^4 + 1X^5 = 1 + X^3 + X^5 + X^6 + X^7$, representing 11101001. The coded value corresponding to 100011 is therefore 11101001. The circuit in Figure 7.23 will carry out this coding operation. To begin with, all flip-flops have their value set to 0. The flip-flops represent multiplication.

The coding circuit can be written down by inspection of the generator

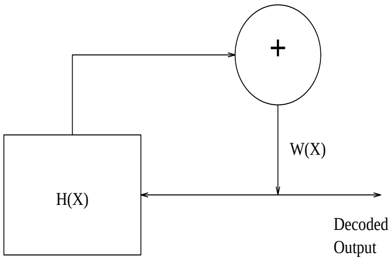


Figure 7.24: Feedback circuit for decoding

polynomial. Let us return to Figure 7.23, and note that the input is fed in serially, bit by bit. When we ask for the multiplication (with modulo-2 addition) by $1 + X + X^2$, we are, in effect saying, "add, modulo-2, the present input bit to the previous one (representing X) to the input before that one (representing X^2)." The circuit follows immediately from that: the flip-flop produces the required delay, and the exclusive-OR gate carries out the modulo-2 addition.

To decode the codeword, we need to reverse the coding process, by dividing by the generator polynomial. This turns out to be easy if we keep in mind that division just means having a multiplier in the feedback loop. Let the initial word to be coded be the polynomial $W(X)$, and the generator polynomial be $G(X) = g_0 + g_1X + \cdots g_nX^n$. Then, the coded value is $C(X) = W(X)G(X)$. Now, division by $G(X)$ should yield a quotient and no remainder. If a remainder does occur, that is an indication of an error.

This division process is implemented as follows. Let $H(X) = G(X) - 1$ (keep in mind that all additions and subtractions are done modulo-2). That is, $H(X) = \overline{g}_0 + g_1X + \cdots g_nX^n$. We have

$$\begin{aligned} C(X) + H(X)W(X) &= W(X)\{g_0 + \overline{g}_0 + (g_1 + g_1)X + \cdots (g_n + g_n)X^n\} \\ &= W(X) \quad (\text{since } g_0 + \overline{g}_0 = 1; g_i + g_i = 0). \end{aligned}$$

So, to obtain the original codeword back, the division process consists of adding $H(X)W(X)$ to the codeword $C(X)$. We can do this by means of the feedback circuit shown in Figure 7.24. For example, if $G(X) = 1 + X + X^2$, we have the circuit in Figure 7.25. If, due to some errors, $C(X)$ is no longer a multiple of the generator polynomial, the codeword is invalid and a non-zero remainder occurs when it is divided by the generator. The remainder is the values held by the flip-flops after the decoding (i.e., the division) is completed, and so if these hold non-zero values after decoding, it is an indication of error.

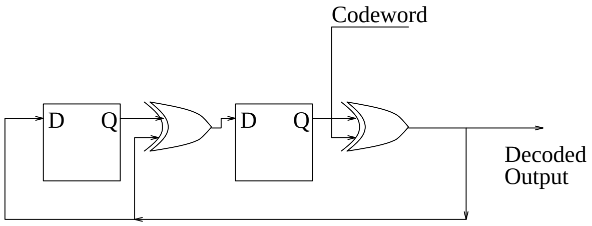


Figure 7.25: Feedback circuit for decoding with $G(X) = 1 + X + X^2$

Number	Encoding
000	000000
001	000101
010	001010
011	001111
100	010100
101	011001
110	011110
111	100101

Table 7.5: AN code with multiplier = 5

7.15.5 Arithmetic Codes

An arithmetic code has the property that given arithmetic operations on a valid codeword will produce another valid codeword. For example, if $C(a)$ and $C(b)$ are the codewords of a and b, respectively, and $A()$ is an arithmetic operation covered by the code, then $A(C(a), C(b)) = C(A(a, b))$. This is very useful for checking the correctness of the arithmetic operation.

The simplest arithmetic code that covers the addition and subtraction operations is the *AN* code, where a number is encoded by multiplying it by some constant. Table 7.5 shows some examples corresponding to the multiplier 5.

7.15.6 Use of Information Redundancy

The most frequent use of coding is to detect and/or correct errors in information transmission and memory. Error-correcting codes used in memory are used during memory *scrubbing.* Periodically, each memory location is accessed, and checked for integrity. If an error is detected, the system corrects it using the error-correcting property of the code, and writes it back. This prevents the accumulation of errors due to transient faults.

7.16 Integrated Failure Handling

When an error is detected, the system must respond swiftly to deal with it. In the short term, the error might be masked by voting. In the long term, the system will have to locate the failure which gave rise to the error, and decide what to do with the failed unit. The following options are usually available.

- *Retry.* Instruction retry simply consists of retrying the failed instruction, in the hope that the failure was caused by a transient, which has since gone away. To be able to do this, we need to be able to detect the error very quickly: if the error is detected after several further instructions have been executed, the question of which instruction caused the error is not easy to answer. To catch errors at this rate, we need fast detection using *signal-level detection* mechanisms. Such mechanisms include error-detection codes, duplicated circuits with matchers, and so on. These mechanisms are distinct from *function-level detection* mechanisms, which operate at the function, or module, level, and take much more time to catch an error.

  If instruction retry is impractical, it might be useful to wait for some time and then run a diagnostic on the affected processor. If the failure was as a result of a transient which has since vanished, the processor will now pass this test. If, after a long time, the processor still exhibits faulty behavior, we can designate it as a permanent failure. How long we should wait for the failure to go away before declaring the failure permanent depends on the lifetime of transient faults. Designers might use their past experience or collected data on transient fault durations to answer this question. Alternatively, the system might gather data on its own transients, and dynamically adjust this waiting time.

- *Disconnect.* If we decide that the processor has suffered permanent failure, it must be disconnected from the rest of the system. This can be done by preventing it communicating on the network, or ignoring its output. If a processor is disconnected, we must find new processors to run the tasks that it was allocated. In Chapter **??**, we discuss a fault-tolerant task assignment and scheduling algorithm which allows this to be done fairly quickly.

- *Replace.* This is another way of responding to permanent failure. If spare processors exist, they can be switched in to take the place of the failed unit. All the duties previously performed by the failed processor will now be transferred to the replacement. The replacement processor must have its memory updated suitably to allow it to undertake these computations, and must have its clock synchronized with the rest of the system.

The following are some of the issues to take into account when deciding which recovery action to follow.

- *Timing information:* The overall aim is to meet the hard deadlines of the critical tasks. This is the ultimate test of success or failure, and everything is subordinated to the need to maximize the probability of meeting such deadlines. Timing information includes the worst-case requirements of the current workload, together with the requirements of any interrupt-issued tasks that may be released. It will also include any impending *mode changes*, which can cause a change in the workload.

- *Recovery time:* This is connected with the probability of meeting hard deadlines. Different recovery mechanisms take differing amounts of time, and this therefore affects the choice of mechanism.

- *Probability of recovery action succeeding:* There is often a tradeoff between low overhead and the probability of success. Instruction retry has perhaps the lowest overhead of all. However, if the failure is a transient of long duration, or a permanent, it will not succeed. At the other end of the spectrum, the entire system (or a subsystem) can be reconfigured to isolate the failure: this has a high probability of success, but takes a long time. A recovery action that fails is worse than no recovery action at all, because it consumes precious time. On the other hand, a recovery action that could have succeeded with little overhead, but was overlooked in favour of another (higher-overhead) action represents a waste of time that can affect reliability and performance.

- *State transition rates:* The rate at which failures occur will play a role in determining what recovery action to take. If, for example, there is a reasonable probability that a second failure will occur while the first is still being dealt with, this will tend to cause us to place an increased premium on short recovery actions.

In order to determine a response to failure, the system must be aware of the time taken by the various error-detection and failure-handling mechanisms. Unfortunately, very little is known about these times in practice. This is an area which requires extensive and systematic experimental research.

7.17 Suggestions for Further Reading

There are several well-known books on fault-tolerant computing. Among these are the books by Anderson and Lee [1], Johnson [7], and Siewiorek and Swarz [20]. Most of the topics covered in this chapter are treated in greater detail in one or more of these books.

A full discussion of the failure rates of solid-state components and the factors that determine them is presented in [21]. The hostile environment that spacecraft operate in is described in detail in [9].

Some concepts related to fail-stop processors are presented in [18]. The concept of sift-out redundancy was introduced by deSouza and Mathur [5]. For a good discussion on voting schemes, see [14]. For papers on the placement of checkpoints in real-time systems, see [11, 19]. The issue of fault-containment is well discussed by Lala, *et al.* [12].

Self-checking and fail-safe circuits are described in [6].

The Byzantine Generals algorithm was introduced by Pease, Shostak and Lamport [15], and named as such in [13].

The recovery block approach to software fault-tolerance was introduced by Randell [17]. For a detailed discussion of N-version programming, see [2, 3, 8, 10].

A good reference for coding theory for fault-tolerance and testing is [4]. There are also several good books on coding. The reader might consult [16] for a thorough – if slightly dated – coverage.

Exercises:

1. Draw the logic diagram of a three-input majority voter, voting on 8-bit inputs, assuming that exact agreement is required. The voter has five

outputs: $\theta_1, \theta_2, w_1, w_2, w_3$. When at least two of the three inputs agree, θ_1 carries the majority value. θ_2 indicates whether or not at least two inputs agree. If a majority exists and input i does not agree with the majority, line w_i is set to 1; it is 0 otherwise.

2. Design the logic of an approximate voter, with the same inputs and outputs as in Exercise 1. Two inputs are said to agree if their values in the 6 most significant bits are identical. The majority value is here defined as the median value.

3. A block diagram of the sift-out redundancy system is shown in Figure 7.10. Draw the detailed logic diagram of such a system for 8-bit processor outputs (each processor has 8 output lines). Assume there are 4 processors in all.

4. Write a subroutine which implements the formalized majority voter. This subroutine accepts as input ϵ, the number of inputs n_i, and a vector of length n_i containing the inputs to be voted on. The output will be the majority output if it exists. If no majority can be found, the subroutine will return with a variable `NOMAJ` set to 1, else this variable value will be 0.

5. Suppose you are asked to design a fault-tolerant system which uses memory scrubbing to get rid of transient errors. The only fault-tolerance scheme used for the memory is an error-correcting code which can correct up to 2 bit errors per word. Failure occurs if more than 2 bit errors occur in a word. Suppose the coded word is 32 bits long, and that memory cells have a probability p of being corrupted in each clock cycle. Assume that these transient cell failures are independent of one another. Calculate the probability of a word suffering failure if the period between consecutive memory scrubs is P clock cycles.

6. What factors govern the optimal placement of checkpoints? Assume that the purpose is to minimize the probability of missing task deadlines.

7. Prove that the domino effect can be prevented by ensuring that recovery points of all the parallel processes are taken at identical times.

8. Consider a Byzantine generals algorithm being run with a total of 7 generals, with two of the lieutenants being traitors, and the remaining officers being loyal. The traitors always convey the message "retreat" no matter what message they receive from their commander. Write out the sequence of messages that results, and show that the interactive consistency conditions are satisfied.

9. Consider a Byzantine generals algorithm being run with a total of 5 generals. The commanding general and one of her lieutenants are traitors; the remaining three lieutenants are loyal. Write out a sequence of messages they would send out to ensure that two of the loyal lieutenants attack and the third retreats.

10. In Theorem 1, we said that the existence of some algorithm whereby the Phoenecian generals achieve agreement for $N \leq 3m$ implies that such algorithm when run with 3 generals would be able to successfully deal with up to one traitor. Prove that this is true.

11. Prove that (if the commander produces a numerical value on which agreement is required), the Byzantine generals algorithm will also work with a voting scheme that picks the median value, rather than majority voting (in Step 3).

12. Which of the following codes are separable: parity, checksum, cyclic? Explain your answer.

13. Design an interlaced parity scheme where the uncoded word has 16 bits, and which can correct up to 2 bit errors.

14. The following is the entire set of code words in some scheme.

A	1000001000
B	1111110000
C	1111110100

What is the Hamming distance for this code? Between which two words does it occur? If the word 0000000000 was received, which codeword would it map to?

15. Draw a circuit for coding with the generator polynomial $1 + X + X^4$. If the serial input is the byte 11110011, what is the output of this generator?

Bibliography

[1] T. Anderson and P. A. Lee, *Fault Tolerance, Principles and Practice,* Englewood Cliffs: Prentice-Hall, 1981.

[2] A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Fault-Tolerance During Execution," *Proceedings of IEEE COMPSAC*, 1977.

[3] A. Avizienis and J. P. J. Kelly, "Fault-Tolerance by Design Diversity," *IEEE Computer*, Vol. 17, 1984.

[4] B. Bose and J. Metzner, "Coding Theory for Fault-Tolerant Systems," in D. K. Pradhan, editor, *Fault-Tolerant Computing*, Englewood Cliffs: Prentice Hall, 1986.

[5] P. T. de Sousa and F. P. Mathur, "Sift-out Modular Redundancy," *IEEE Transactions on Computers*, Vol. C-27, 1978.

[6] M. Diaz, P. Azema, and J. M. Ayache, "Unified Design of Self-Checking and Fail-Safe Combinational Circuits and Sequential Machines," *IEEE Transactions on Computers*, Vol. C-28, 1978.

[7] B. Johnson, *Design and Analysis of Fault Tolerant Digital Systems,* Reading: Addison-Wesley, 1989.

[8] J. P. J. Kelly and S. Murphy, "Dependable Distributed Software," in Y.-H. Lee and C. M. Krishna (editors), *Readings in Real-Time Systems*, Cupertino: IEEE Computer Society Press, 1993.

[9] S. E. Kerns and K. F. Galloway, editors, *Proceedings of the IEEE* special section on space radiation effects, Vol. 76, 1988.

[10] J. Knight and N. Leveson, "An Experimental Evaluation of Independence in Multiversion Programming," *IEEE Transactions on Software Engineering*, Vol. SE-12, 1986.

[11] C. M. Krishna, Y.-H. Lee, and K. G. Shin, "Optimization Criteria for Checkpoint Placement," *Communications of the ACM*, Vol. 27, 1984.

[12] J. H. Lala, R. E. Harper, and L. S. Alger, "A Design Approach for Ultrareliable Real-Time Systems," *IEEE Computer*, Vol. 24, 1991.

[13] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Algorithm," *ACM Transactions on Programming Languages and Systems*, Vol. 4, 1982.

[14] P. R. Larczak, A. K. Caglayan, D. E. Eckhardt, "A Theoretical Investigation of Generalized Voters for Redundant Systems," *Proc. Fault-Tolerant Computing Symposium*, 1989.

[15] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, Vol. 25, 1980.

[16] W. W. Peterson and E. J. Weldon, jr, *Error-Correcting Codes*, Cambridge: MIT Press, 1972.

[17] B. Randell, "System Structure for Software Fault-Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, 1975.

[18] R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computing Systems*, Vol. 1, 1983.

[19] K. G. Shin, T.-H. Lin, and Y.-H. Lee, "Optimal Checkpointing of Real-Time Tasks," *IEEE Transactions on Computers*, Vol. C-36, 1987.

[20] D. P. Siewiorek and R. Swarz, *Reliable Computer Systems: Design and Evaluation,* Maynard: Digital Press, 1992.

[21] U.S. Department of Defense, *Military Standardization Handbook: Reliability Prediction of Electronic Equipment*, MIL-HDBK-217E, 1986.