Principles of Computer System Design

An Introduction

Chapter 8
Fault Tolerance: Reliable Systems from Unreliable Components

Jerome H. Saltzer

M. Frans Kaashoek

Massachusetts Institute of Technology

Version 5.0

Suggestions, Comments, Corrections, and Requests to waive license restrictions: Please send correspondence by electronic mail to:

Saltzer@mit.edu

and

kaashoek@mit.edu

CHAPTER

Fault Tolerance: Reliable Systems from Unreliable Components

8

CHAPTER CONTENTS

Overview 8–2
8.1 Faults, Failures, and Fault Tolerant Design 8–3
8.1.1 Faults, Failures, and Modules 8–3
8.1.2 The Fault-Tolerance Design Process 8–6
8.2 Measures of Reliability and Failure Tolerance 8–8
8.2.1 Availability and Mean Time to Failure 8–8
8.2.2 Reliability Functions 8–13
8.2.3 Measuring Fault Tolerance 8–16
8.3 Tolerating Active Faults 8–16
8.3.1 Responding to Active Faults 8–16
8.3.2 Fault Tolerance Models 8–18
8.4 Systematically Applying Redundancy 8–20
8.4.1 Coding: Incremental Redundancy 8–21
8.4.2 Replication: Massive Redundancy 8–25
8.4.3 Voting 8–26
8.4.4 Repair 8–31
8.5 Applying Redundancy to Software and Data 8–36
8.5.1 Tolerating Software Faults 8–36
8.5.2 Tolerating Software (and other) Faults by Separating State 8–37
8.5.3 Durability and Durable Storage 8–39
8.5.4 Magnetic Disk Fault Tolerance 8–40
8.5.4.1 Magnetic Disk Fault Modes 8–41
8.5.4.2 System Faults 8–42
8.5.4.3 Raw Disk Storage 8–43
8.5.4.4 Fail-Fast Disk Storage 8–43
8.5.4.5 Careful Disk Storage 8–45
8.5.4.6 Durable Storage: RAID 1 8–46
8.5.4.7 Improving on RAID 1 8–47
8.5.4.8 Detecting Errors Caused by System Crashes 8–49
8.5.4.9 Still More Threats to Durability 8–49
8.6 Wrapping up Reliability 8–51

8.6.1 Design Strategies and Design Principles 8–51
8.6.2 How about the End-to-End Argument? 8–52
8.6.3 A Caution on the Use of Reliability Calculations 8–53
8.6.4 Where to Learn More about Reliable Systems 8–53
8.7 Application: A Fault Tolerance Model for CMOS RAM 8–55
8.8 War Stories: Fault Tolerant Systems that Failed.....8–57
8.8.1 Adventures with Error Correction 8–57
8.8.2 Risks of Rarely-Used Procedures: The National Archives 8–59
8.8.3 Non-independent Replicas and Backhoe Fade 8–60
8.8.4 Human Error May Be the Biggest Risk 8–61
8.8.5 Introducing a Single Point of Failure 8–63
8.8.6 Multiple Failures: The SOHO Mission Interruption 8–63
Exercises.....8–64
Glossary for Chapter 8 8–69
Index of Chapter 8 8–75
Last chapter page 8–77

Overview

Construction of reliable systems from unreliable components is one of the most important applications of modularity. There are, in principle, three basic steps to building reliable systems:

1. *Error detection*: discovering that there is an error in a data value or control signal. Error detection is accomplished with the help of *redundancy*, extra information that can verify correctness.
2. *Error containment*: limiting how far the effects of an error propagate. Error containment comes from careful application of modularity. When discussing reliability, a *module* is usually taken to be the unit that fails independently of other such units. It is also usually the unit of repair and replacement.
3. *Error masking*: ensuring correct operation despite the error. Error masking is accomplished by providing enough additional redundancy that it is possible to discover correct, or at least acceptably close, values of the erroneous data or control signal. When masking involves changing incorrect values to correct ones, it is usually called *error correction*.

Since these three steps can overlap in practice, one sometimes finds a single error-handling mechanism that merges two or even all three of the steps.

In earlier chapters each of these ideas has already appeared in specialized forms:

- A primary purpose of enforced modularity, as provided by client/server architecture, virtual memory, and threads, is error containment.

- Network links typically use error detection to identify and discard damaged frames.
- Some end-to-end protocols time out and resend lost data segments, thus masking the loss.
- Routing algorithms find their way around links that fail, masking those failures.
- Some real-time applications fill in missing data by interpolation or repetition, thus masking loss.

and, as we will see in Chapter 11[on-line], secure systems use a technique called *defense in depth* both to contain and to mask errors in individual protection mechanisms. In this chapter we explore systematic application of these techniques to more general problems, as well as learn about both their power and their limitations.

8.1 Faults, Failures, and Fault Tolerant Design

8.1.1 Faults, Failures, and Modules

Before getting into the techniques of constructing reliable systems, let us distinguish between concepts and give them separate labels. In ordinary English discourse, the three words "fault," "failure," and "error" are used more or less interchangeably or at least with strongly overlapping meanings. In discussing reliable systems, we assign these terms to distinct formal concepts. The distinction involves modularity. Although common English usage occasionally intrudes, the distinctions are worth maintaining in technical settings.

A *fault* is an underlying defect, imperfection, or flaw that has the potential to cause problems, whether it actually has, has not, or ever will. A weak area in the casing of a tire is an example of a fault. Even though the casing has not actually cracked yet, the fault is lurking. If the casing cracks, the tire blows out, and the car careens off a cliff, the resulting crash is a *failure*. (That definition of the term "failure" by example is too informal; we will give a more careful definition in a moment.) One fault that underlies the failure is the weak spot in the tire casing. Other faults, such as an inattentive driver and lack of a guard rail, may also contribute to the failure.

Experience suggests that faults are commonplace in computer systems. Faults come from many different sources: software, hardware, design, implementation, operations, and the environment of the system. Here are some typical examples:

- Software fault: A programming mistake, such as placing a less-than sign where there should be a less-than-or-equal sign. This fault may never have caused any trouble because the combination of events that requires the equality case to be handled correctly has not yet occurred. Or, perhaps it is the reason that the system crashes twice a day. If so, those crashes are failures.

- Hardware fault: A gate whose output is stuck at the value ZERO. Until something depends on the gate correctly producing the output value ONE, nothing goes wrong. If you publish a paper with an incorrect sum that was calculated by this gate, a failure has occurred. Furthermore, the paper now contains a fault that may lead some reader to do something that causes a failure elsewhere.
- Design fault: A miscalculation that has led to installing too little memory in a telephone switch. It may be months or years until the first time that the presented load is great enough that the switch actually begins failing to accept calls that its specification says it should be able to handle.
- Implementation fault: Installing less memory than the design called for. In this case the failure may be identical to the one in the previous example of a design fault, but the fault itself is different.
- Operations fault: The operator responsible for running the weekly payroll ran the payroll program twice last Friday. Even though the operator shredded the extra checks, this fault has probably filled the payroll database with errors such as wrong values for year-to-date tax payments.
- Environment fault: Lightning strikes a power line, causing a voltage surge. The computer is still running, but a register that was being updated at that instant now has several bits in error. Environment faults come in all sizes, from bacteria contaminating ink-jet printer cartridges to a storm surge washing an entire building out to sea.

Some of these examples suggest that a fault may either be *latent*, meaning that it isn't affecting anything right now, or *active*. When a fault is active, wrong results appear in data values or control signals. These wrong results are *errors*. If one has a formal specification for the design of a module, an error would show up as a violation of some assertion or invariant of the specification. The violation means that either the formal specification is wrong (for example, someone didn't articulate all of the assumptions) or a module that this component depends on did not meet its own specification. Unfortunately, formal specifications are rare in practice, so discovery of errors is more likely to be somewhat *ad hoc*.

If an error is not detected and masked, the module probably does not perform to its specification. Not producing the intended result at an interface is the formal definition of a *failure*. Thus, the distinction between fault and failure is closely tied to modularity and the building of systems out of well-defined subsystems. In a system built of subsystems, the failure of a subsystem is a fault from the point of view of the larger subsystem that contains it. That fault may cause an error that leads to the failure of the larger subsystem, unless the larger subsystem anticipates the possibility of the first one failing, detects the resulting error, and masks it. Thus, if you notice that you have a flat tire, you have detected an error caused by failure of a subsystem you depend on. If you miss an appointment because of the flat tire, the person you intended to meet notices a failure of

a larger subsystem. If you change to a spare tire in time to get to the appointment, you have masked the error within your subsystem. *Fault tolerance* thus consists of noticing active faults and component subsystem failures and doing something helpful in response.

One such helpful response is *error containment*, which is another close relative of modularity and the building of systems out of subsystems. When an active fault causes an error in a subsystem, it may be difficult to confine the effects of that error to just a portion of the subsystem. On the other hand, one should expect that, as seen from outside that subsystem, the only effects will be at the specified interfaces of the subsystem. In consequence, the boundary adopted for error containment is usually the boundary of the smallest subsystem inside which the error occurred. From the point of view of the next higher-level subsystem, the subsystem with the error may contain the error in one of four ways:

1. Mask the error, so the higher-level subsystem does not realize that anything went wrong. One can think of failure as falling off a cliff and masking as a way of providing some separation from the edge.
2. Detect and report the error at its interface, producing what is called a *fail-fast* design. Fail-fast subsystems simplify the job of detection and masking for the next higher-level subsystem. If a fail-fast module correctly reports that its output is questionable, it has actually met its specification, so it has not failed. (Fail-fast modules can still fail, for example by not noticing their own errors.)
3. Immediately stop dead, thereby hoping to limit propagation of bad values, a technique known as *fail-stop*. Fail-stop subsystems require that the higher-level subsystem take some additional measure to discover the failure, for example by setting a timer and responding to its expiration. A problem with fail-stop design is that it can be difficult to distinguish a stopped subsystem from one that is merely running more slowly than expected. This problem is particularly acute in asynchronous systems.
4. Do nothing, simply failing without warning. At the interface, the error may have contaminated any or all output values. (Informally called a "crash" or perhaps "fail-thud".)

Another useful distinction is that of transient versus persistent faults. A *transient* fault, also known as a *single-event upset*, is temporary, triggered by some passing external event such as lightning striking a power line or a cosmic ray passing through a chip. It is usually possible to mask an error caused by a transient fault by trying the operation again. An error that is successfully masked by retry is known as a *soft error*. A *persistent* fault continues to produce errors, no matter how many times one retries, and the corresponding errors are called *hard errors*. An *intermittent* fault is a persistent fault that is active only occasionally, for example, when the noise level is higher than usual but still within specifications. Finally, it is sometimes useful to talk about *latency*, which in reliability terminology is the time between when a fault causes an error and when the error is

detected or causes the module to fail. Latency can be an important parameter because some error-detection and error-masking mechanisms depend on there being at most a small fixed number of errors—often just one—at a time. If the error latency is large, there may be time for a second error to occur before the first one is detected and masked, in which case masking of the first error may not succeed. Also, a large error latency gives time for the error to propagate and may thus complicate containment.

Using this terminology, an improperly fabricated stuck-at-ZERO bit in a memory chip is a persistent fault: whenever the bit should contain a ONE the fault is active and the value of the bit is in error; at times when the bit is supposed to contain a ZERO, the fault is latent. If the chip is a component of a fault tolerant memory module, the module design probably includes an error-correction code that prevents that error from turning into a failure of the module. If a passing cosmic ray flips another bit in the same chip, a transient fault has caused that bit also to be in error, but the same error-correction code may still be able to prevent this error from turning into a module failure. On the other hand, if the error-correction code can handle only single-bit errors, the combination of the persistent and the transient fault might lead the module to produce wrong data across its interface, a failure of the module. If someone were then to test the module by storing new data in it and reading it back, the test would probably not reveal a failure because the transient fault does not affect the new data. Because simple input/output testing does not reveal successfully masked errors, a fault tolerant module design should always include some way to report that the module masked an error. If it does not, the user of the module may not realize that persistent errors are accumulating but hidden.

8.1.2 The Fault-Tolerance Design Process

One way to design a reliable system would be to build it entirely of components that are individually so reliable that their chance of failure can be neglected. This technique is known as *fault avoidance*. Unfortunately, it is hard to apply this technique to every component of a large system. In addition, the sheer number of components may defeat the strategy. If all N of the components of a system must work, the probability of any one component failing is p, and component failures are independent of one another, then the probability that the system works is $(1 - p)^N$. No matter how small p may be, there is some value of N beyond which this probability becomes too small for the system to be useful.

The alternative is to apply various techniques that are known collectively by the name *fault tolerance*. The remainder of this chapter describes several such techniques that are the elements of an overall design process for building reliable systems from unreliable components. Here is an overview of the *fault-tolerance design process*:

1. Begin to develop a fault-tolerance model, as described in Section 8.3:

- Identify every potential fault.
- Estimate the risk of each fault, as described in Section 8.2.
- Where the risk is too high, design methods to detect the resulting errors.

2. Apply modularity to contain the damage from the high-risk errors.
3. Design and implement procedures that can mask the detected errors, using the techniques described in Section 8.4:

- Temporal redundancy. Retry the operation, using the same components.
- Spatial redundancy. Have different components do the operation.

4. Update the fault-tolerance model to account for those improvements.
5. Iterate the design and the model until the probability of untolerated faults is low enough that it is acceptable.
6. Observe the system in the field:

- Check logs of how many errors the system is successfully masking. (Always keep track of the distance to the edge of the cliff.)
- Perform postmortems on failures and identify *all* of the reasons for each failure.

7. Use the logs of masked faults and the postmortem reports about failures to revise and improve the fault-tolerance model and reiterate the design.

The fault-tolerance design process includes some subjective steps, for example, deciding that a risk of failure is "unacceptably high" or that the "probability of an untolerated fault is low enough that it is acceptable." It is at these points that different application requirements can lead to radically different approaches to achieving reliability. A personal computer may be designed with no redundant components, the computer system for a small business is likely to make periodic backup copies of most of its data and store the backup copies at another site, and some space-flight guidance systems use five completely redundant computers designed by at least two independent vendors. The decisions required involve trade-offs between the cost of failure and the cost of implementing fault tolerance. These decisions can blend into decisions involving business models and risk management. In some cases it may be appropriate to opt for a nontechnical solution, for example, deliberately accepting an increased risk of failure and covering that risk with insurance.

The fault-tolerance design process can be described as a *safety-net* approach to system design. The safety-net approach involves application of some familiar design principles and also some not previously encountered. It starts with a new design principle:

Be explicit

Get all of the assumptions out on the table.

The primary purpose of creating a fault-tolerance model is to expose and document the assumptions and articulate them explicitly. The designer needs to have these assumptions not only for the initial design, but also in order to respond to field reports of

unexpected failures. Unexpected failures represent omissions or violations of the assumptions.

Assuming that you won't get it right the first time, the second design principle of the safety-net approach is the familiar *design for iteration.* It is difficult or impossible to anticipate all of the ways that things can go wrong. Moreover, when working with a fast-changing technology it can be hard to estimate probabilities of failure in components and in their organization, especially when the organization is controlled by software. For these reasons, a fault tolerant design must include feedback about actual error rates, evaluation of that feedback, and update of the design as field experience is gained. These two principles interact: to act on the feedback requires having a fault tolerance model that is explicit about reliability assumptions.

The third design principle of the safety-net approach is also familiar: the *safety margin principle*, described near the end of Section 1.3.2. An essential part of a fault tolerant design is to monitor how often errors are masked. When fault tolerant systems fail, it is usually not because they had inadequate fault tolerance, but because the number of failures grew unnoticed until the fault tolerance of the design was exceeded. The key requirement is that the system log all failures and that someone pay attention to the logs. The biggest difficulty to overcome in applying this principle is that it is hard to motivate people to expend effort checking something that seems to be working.

The fourth design principle of the safety-net approach came up in the introduction to the study of systems; it shows up here in the instruction to identify all of the causes of each failure: *keep digging*. Complex systems fail for complex reasons. When a failure of a system that is supposed to be reliable does occur, always look beyond the first, obvious cause. It is nearly always the case that there are actually several contributing causes and that there was something about the mind set of the designer that allowed each of those causes to creep in to the design.

Finally, complexity increases the chances of mistakes, so it is an enemy of reliability. The fifth design principle embodied in the safety-net approach is to *adopt sweeping simplifications*. This principle does not show up explicitly in the description of the fault-tolerance design process, but it will appear several times as we go into more detail.

The safety-net approach is applicable not just to fault tolerant design. Chapter 11[online] will show that the safety-net approach is used in an even more rigorous form in designing systems that must protect information from malicious actions.

8.2 Measures of Reliability and Failure Tolerance

8.2.1 Availability and Mean Time to Failure

A useful model of a system or a system component, from a reliability point of view, is that it operates correctly for some period of time and then it fails. The time to failure (TTF) is thus a measure of interest, and it is something that we would like to be able to predict. If a higher-level module does not mask the failure and the failure is persistent,

the system cannot be used until it is repaired, perhaps by replacing the failed component, so we are equally interested in the time to repair (TTR). If we observe a system through N run–fail–repair cycles and observe in each cycle i the values of TTF_i and TTR_i, we can calculate the fraction of time it operated properly, a useful measure known as *availability:*

$$Availability = \frac{\text{time system was running}}{\text{time system should have been running}}$$

$$= \frac{\sum_{i=1}^{N} TTF_i}{\sum_{i=1}^{N} (TTF_i + TTR_i)} \qquad \text{Eq. 8–1}$$

By separating the denominator of the availability expression into two sums and dividing each by N (the number of observed failures) we obtain two time averages that are frequently reported as operational statistics: the *mean time to failure* (MTTF) and the *mean time to repair* (MTTR):

$$MTTF = \frac{1}{N}\sum_{i=1}^{N} TTF_i \qquad MTTR = \frac{1}{N}\sum_{i=1}^{N} TTR_i \qquad \text{Eq. 8–2}$$

The sum of these two statistics is usually called the *mean time between failures* (MTBF). Thus availability can be variously described as

$$Availability = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR} = \frac{MTBF - MTTR}{MTBF} \qquad \text{Eq. 8–3}$$

In some situations, it is more useful to measure the fraction of time that the system is not working, known as its *down time*:

$$\text{Down time} = (1 - Availability) = \frac{MTTR}{MTBF} \qquad \text{Eq. 8–4}$$

One thing that the definition of down time makes clear is that MTTR and MTBF are in some sense equally important. One can reduce down time either by reducing MTTR or by increasing MTBF.

Components are often repaired by simply replacing them with new ones. When failed components are discarded rather than fixed and returned to service, it is common to use a slightly different method to measure MTTF. The method is to place a batch of N components in service in different systems (or in what is hoped to be an equivalent test environment), run them until they have all failed, and use the set of failure times as the TTF_i in equation 8–2. This procedure substitutes an ensemble average for the time average. We could use this same procedure on components that are not usually discarded when they fail, in the hope of determining their MTTF more quickly, but we might obtain a different value for the MTTF. Some failure processes do have the property that the ensemble average is the same as the time average (processes with this property are

called *ergodic*), but other failure processes do not. For example, the repair itself may cause wear, tear, and disruption to other parts of the system, in which case each successive system failure might on average occur sooner than did the previous one. If that is the case, an MTTF calculated from an ensemble-average measurement might be too optimistic.

As we have defined them, availability, MTTF, MTTR, and MTBF are backward-looking measures. They are used for two distinct purposes: (1) for evaluating how the system is doing (compared, for example, with predictions made when the system was designed) and (2) for predicting how the system will behave in the future. The first purpose is concrete and well defined. The second requires that one take on faith that samples from the past provide an adequate predictor of the future, which can be a risky assumption. There are other problems associated with these measures. While MTTR can usually be measured in the field, the more reliable a component or system the longer it takes to evaluate its MTTF, so that measure is often not directly available. Instead, it is common to use and measure proxies to estimate its value. The quality of the resulting estimate of availability then depends on the quality of the proxy.

A typical 3.5-inch magnetic disk comes with a reliability specification of 300,000 hours "MTTF", which is about 34 years. Since the company quoting this number has probably not been in business that long, it is apparent that whatever they are calling "MTTF" is not the same as either the time-average or the ensemble-average MTTF that we just defined. It is actually a quite different statistic, which is why we put quotes around its name. Sometimes this "MTTF" is a theoretical prediction obtained by modeling the ways that the components of the disk might be expected to fail and calculating an expected time to failure.

A more likely possibility is that the manufacturer measured this "MTTF" by running an array of disks simultaneously for a much shorter time and counting the number of failures. For example, suppose the manufacturer ran 1,000 disks for 3,000 hours (about four months) each, and during that time 10 of the disks failed. The observed failure rate of this sample is 1 failure for every 300,000 hours of operation. The next step is to invert the failure rate to obtain 300,000 hours of operation per failure and then quote this number as the "MTTF". But the relation between this sample observation of failure rate and the real MTTF is problematic. If the failure process were memoryless (meaning that the failure rate is independent of time; Section 8.2.2, below, explores this idea more thoroughly), we would have the special case in which the MTTF really is the inverse of the failure rate. A good clue that the disk failure process is not memoryless is that the disk specification may also mention an "expected operational lifetime" of only 5 years. That statistic is probably the real MTTF—though even that may be a prediction based on modeling rather than a measured ensemble average. An appropriate re-interpretation of the 34-year "MTTF" statistic is to invert it and identify the result as a *short-term* failure rate that applies only within the expected operational lifetime. The paragraph discussing equation 8–9 on page 8–13 describes a fallacy that sometimes leads to miscalculation of statistics such as the MTTF.

Magnetic disks, light bulbs, and many other components exhibit a time-varying statistical failure rate known as a *bathtub curve*, illustrated in Figure 8.1 and defined more

carefully in Section 8.2.2, below. When components come off the production line, a certain fraction fail almost immediately because of gross manufacturing defects. Those components that survive this initial period usually run for a long time with a relatively uniform failure rate. Eventually, accumulated wear and tear cause the failure rate to increase again, often quite rapidly, producing a failure rate plot that resembles the shape of a bathtub.

Several other suggestive and colorful terms describe these phenomena. Components that fail early are said to be subject to *infant mortality,* and those that fail near the end of their expected lifetimes are said to *burn out*. Manufacturers sometimes *burn in* such components by running them for a while before shipping, with the intent of identifying and discarding the ones that would otherwise fail immediately upon being placed in service. When a vendor quotes an "expected operational lifetime," it is probably the mean time to failure of those components that survive burn in, while the much larger "MTTF" number is probably the inverse of the observed failure rate at the lowest point of the bathtub. (The published numbers also sometimes depend on the outcome of a debate between the legal department and the marketing department, but that gets us into a different topic.) A chip manufacturer describes the fraction of components that survive the burn-in period as the *yield* of the production line. Component manufacturers usually exhibit a phenomenon known informally as a *learning curve,* which simply means that the first components coming out of a new production line tend to have more failures than later ones. The reason is that manufacturers *design for iteration*: upon seeing and analyzing failures in the early production batches, the production line designer figures out how to refine the manufacturing process to reduce the infant mortality rate.

One job of the system designer is to exploit the nonuniform failure rates predicted by the bathtub and learning curves. For example, a conservative designer exploits the learning curve by avoiding the latest generation of hard disks in favor of slightly older designs that have accumulated more field experience. One can usually rely on other designers who may be concerned more about cost or performance than availability to shake out the bugs in the newest generation of disks.

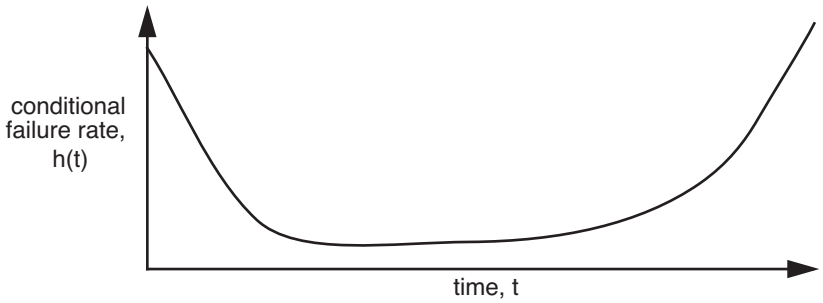


FIGURE 8.1

A bathtub curve, showing how the conditional failure rate of a component changes with time.

The 34-year "MTTF" disk drive specification may seem like public relations puffery in the face of the specification of a 5-year expected operational lifetime, but these two numbers actually are useful as a measure of the nonuniformity of the failure rate. This nonuniformity is also susceptible to exploitation, depending on the operation plan. If the operation plan puts the component in a system such as a satellite, in which it will run until it fails, the designer would base system availability and reliability estimates on the 5-year figure. On the other hand, the designer of a ground-based storage system, mindful that the 5-year operational lifetime identifies the point where the conditional failure rate starts to climb rapidly at the far end of the bathtub curve, might include a plan to replace perfectly good hard disks before burn-out begins to dominate the failure rate—in this case, perhaps every 3 years. Since one can arrange to do scheduled replacement at convenient times, for example, when the system is down for another reason, or perhaps even without bringing the system down, the designer can minimize the effect on system availability. The manufacturer's 34-year "MTTF", which is probably the inverse of the observed failure rate at the lowest point of the bathtub curve, then can be used as an estimate of the expected rate of unplanned replacements, although experience suggests that this specification may be a bit optimistic. Scheduled replacements are an example of *preventive maintenance*, which is active intervention intended to increase the mean time to failure of a module or system and thus improve availability.

For some components, observed failure rates are so low that MTTF is estimated by *accelerated aging*. This technique involves making an educated guess about what the dominant underlying cause of failure will be and then amplifying that cause. For example, it is conjectured that failures in recordable Compact Disks are heat-related. A typical test scenario is to store batches of recorded CDs at various elevated temperatures for several months, periodically bringing them out to test them and count how many have failed. One then plots these failure rates versus temperature and extrapolates to estimate what the failure rate would have been at room temperature. Again making the assumption that the failure process is memoryless, that failure rate is then inverted to produce an MTTF. Published MTTFs of 100 years or more have been obtained this way. If the dominant fault mechanism turns out to be something else (such as bacteria munching on the plastic coating) or if after 50 years the failure process turns out not to be memoryless after all, an estimate from an accelerated aging study may be far wide of the mark. A designer must use such estimates with caution and understanding of the assumptions that went into them.

Availability is sometimes discussed by counting the number of nines in the numerical representation of the availability measure. Thus a system that is up and running 99.9% of the time is said to have 3-nines availability. Measuring by nines is often used in marketing because it sounds impressive. A more meaningful number is usually obtained by calculating the corresponding down time. A 3-nines system can be down nearly 1.5 minutes per day or 8 hours per year, a 5-nines system 5 minutes per year, and a 7-nines system only 3 seconds per year. Another problem with measuring by nines is that it tells only about availability, without any information about MTTF. One 3-nines system may have a brief failure every day, while a different 3-nines system may have a single eight

hour outage once a year. Depending on the application, the difference between those two systems could be important. Any single measure should always be suspect.

Finally, availability can be a more fine-grained concept. Some systems are designed so that when they fail, some functions (for example, the ability to read data) remain available, while others (the ability to make changes to the data) are not. Systems that continue to provide partial service in the face of failure are called *fail-soft*, a concept defined more carefully in Section 8.3.

8.2.2 Reliability Functions

The bathtub curve expresses the conditional failure rate $h(t)$ of a module, defined to be the probability that the module fails between time t and time $t + dt$, given that the component is still working at time t. The conditional failure rate is only one of several closely related ways of describing the failure characteristics of a component, module, or system. The *reliability*, R, of a module is defined to be

$$R(t) = Pr\left(\begin{array}{c}\text{the module has not yet failed at time t, given that}\\ \text{the module was operating at time 0}\end{array}\right) \qquad \text{Eq. 8–5}$$

and the unconditional failure rate $f(t)$ is defined to be

$$f(t) = Pr(\text{module fails between t and t + dt}) \qquad \text{Eq. 8–6}$$

(The bathtub curve and these two reliability functions are three ways of presenting the same information. If you are rusty on probability, a brief reminder of how they are related appears in Sidebar 8.1.) Once $f(t)$ is at hand, one can directly calculate the MTTF:

$$MTTF = \int_0^\infty t \cdot f(t)\,dt \qquad \text{Eq. 8–7}$$

One must keep in mind that this MTTF is predicted from the failure rate function $f(t)$, in contrast to the MTTF of eq. 8–2, which is the result of a field measurement. The two MTTFs will be the same only if the failure model embodied in $f(t)$ is accurate.

Some components exhibit relatively uniform failure rates, at least for the lifetime of the system of which they are a part. For these components the conditional failure rate, rather than resembling a bathtub, is a straight horizontal line, and the reliability function becomes a simple declining exponential:

$$R(t) = e^{-\left(\frac{t}{MTTF}\right)} \qquad \text{Eq. 8–8}$$

This reliability function is said to be *memoryless*, which simply means that the conditional failure rate is independent of how long the component has been operating. Memoryless failure processes have the nice property that the conditional failure rate is the inverse of the MTTF.

Unfortunately, as we saw in the case of the disks with the 34-year "MTTF", this property is sometimes misappropriated to quote an MTTF for a component whose

Sidebar 8.1: Reliability functions The failure rate function, the reliability function, and the bathtub curve (which in probability texts is called the *conditional failure rate function*, and which in operations research texts is called the *hazard function*) are actually three mathematically related ways of describing the same information. The failure rate function, *f(t)* as defined in equation 8–6, is a *probability density function*, which is everywhere non-negative and whose integral over all time is 1. Integrating the failure rate function from the time the component was created (conventionally taken to be $t = 0$) to the present time yields

$$F(t) = \int_0^t f(t)dt$$

F(t) is the cumulative probability that the component has failed by time *t*. The cumulative probability that the component has *not* failed is the probability that it is still operating at time *t* given that it was operating at time 0, which is exactly the definition of the reliability function, *R(t)*. That is,

$$R(t) = 1 - F(t)$$

The bathtub curve of Figure 8.1 reports the conditional probability *h(t)* that a failure occurs between t and t + dt, given that the component was operating at time t. By the definition of conditional probability, the conditional failure rate function is thus

$$h(t) = \frac{f(t)}{R(t)}$$

conditional failure rate does change with time. This misappropriation starts with a fallacy: an assumption that the MTTF, as defined in eq. 8–7, can be calculated by inverting the measured failure rate. The fallacy arises because in general,

$$E(1/t) \neq 1/E(t) \qquad \text{Eq. 8–9}$$

That is, the expected value of the inverse is *not* equal to the inverse of the expected value, except in certain special cases. The important special case in which they *are* equal is the memoryless distribution of eq. 8–8. When a random process is memoryless, calculations and measurements are so much simpler that designers sometimes forget that the same simplicity does not apply everywhere.

Just as availability is sometimes expressed in an oversimplified way by counting the number of nines in its numerical representation, reliability in component manufacturing is sometimes expressed in an oversimplified way by counting standard deviations in the observed distribution of some component parameter, such as the maximum propagation time of a gate. The usual symbol for standard deviation is the Greek letter σ (sigma), and a normal distribution has a standard deviation of 1.0, so saying that a component has "4.5 σ reliability" is a shorthand way of saying that the production line controls variations in that parameter well enough that the specified tolerance is 4.5 standard deviations away from the mean value, as illustrated in Figure 8.2. Suppose, for example, that a pro-

duction line is manufacturing gates that are specified to have a mean propagation time of 10 nanoseconds and a maximum propagation time of 11.8 nanoseconds with 4.5 σ reliability. The difference between the mean and the maximum, 1.8 nanoseconds, is the tolerance. For that tolerance to be 4.5 σ, σ would have to be no more than 0.4 nanoseconds. To meet the specification, the production line designer would measure the actual propagation times of production line samples and, if the observed variance is greater than 0.4 ns, look for ways to reduce the variance to that level.

Another way of interpreting "4.5 σ reliability" is to calculate the expected fraction of components that are outside the specified tolerance. That fraction is the integral of one tail of the normal distribution from 4.5 to ∞, which is about 3.4×10^{-6}, so in our example no more than 3.4 out of each million gates manufactured would have delays greater than 11.8 nanoseconds. Unfortunately, this measure describes only the failure rate of the production line, it does not say anything about the failure rate of the component after it is installed in a system.

A currently popular quality control method, known as "Six Sigma", is an application of two of our design principles to the manufacturing process. The idea is to use measurement, feedback, and iteration (*design for iteration*: "you won't get it right the first time") to reduce the variance (*the robustness principle*: "be strict on outputs") of production-line manufacturing. The "Six Sigma" label is somewhat misleading because in the application of the method, the number 6 is allocated to deal with two quite different effects. The method sets a target of controlling the production line variance to the level of 4.5 σ, just as in the gate example of Figure 8.2. The remaining 1.5 σ is the amount that the mean output value is allowed to drift away from its original specification over the life of the

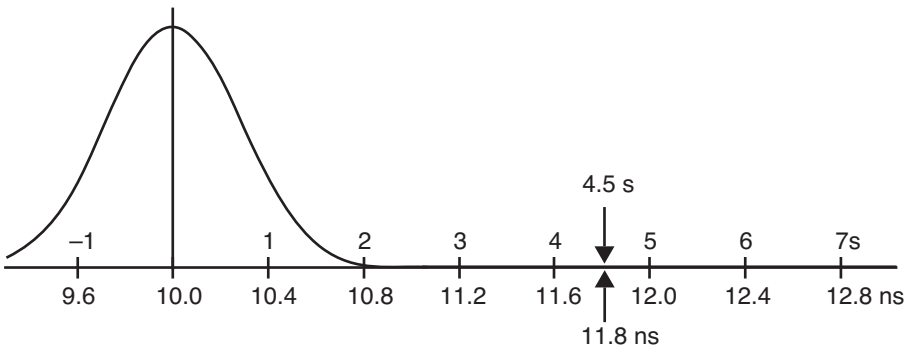


FIGURE 8.2

The normal probability density function applied to production of gates that are specified to have mean propagation time of 10 nanoseconds and maximum propagation time of 11.8 nanoseconds. The upper numbers on the horizontal axis measure the distance from the mean in units of the standard deviation, σ. The lower numbers depict the corresponding propagation times. The integral of the tail from 4.5 σ to ∞ is so small that it is not visible in this figure.

production line. So even though the production line may start 6 σ away from the tolerance limit, after it has been operating for a while one may find that the failure rate has drifted upward to the same 3.4 in a million calculated for the 4.5 σ case.

In manufacturing quality control literature, these applications of the two design principles are known as *Taguchi methods*, after their popularizer, Genichi Taguchi.

8.2.3 Measuring Fault Tolerance

It is sometimes useful to have a quantitative measure of the fault tolerance of a system. One common measure, sometimes called the *failure tolerance*, is the number of failures of its components that a system can tolerate without itself failing. Although this label could be ambiguous, it is usually clear from context that a measure is being discussed. Thus a memory system that includes single-error correction (Section 8.4 describes how error correction works) has a failure tolerance of one bit.

When a failure occurs, the remaining failure tolerance of the system goes down. The remaining failure tolerance is an important thing to monitor during operation of the system because it shows how close the system as a whole is to failure. One of the most common system design mistakes is to add fault tolerance but not include any monitoring to see how much of the fault tolerance has been used up, thus ignoring the *safety margin principle*. When systems that are nominally fault tolerant do fail, later analysis invariably discloses that there were several failures that the system successfully masked but that somehow were never reported and thus were never repaired. Eventually, the total number of failures exceeded the designed failure tolerance of the system.

Failure tolerance is actually a single number in only the simplest situations. Sometimes it is better described as a vector, or even as a matrix showing the specific combinations of different kinds of failures that the system is designed to tolerate. For example, an electric power company might say that it can tolerate the failure of up to 15% of its generating capacity, at the same time as the downing of up to two of its main transmission lines.

8.3 Tolerating Active Faults

8.3.1 Responding to Active Faults

In dealing with active faults, the designer of a module can provide one of several responses:

- *Do nothing.* The error becomes a failure of the module, and the larger system or subsystem of which it is a component inherits the responsibilities both of discovering and of handling the problem. The designer of the larger subsystem then must choose which of these responses to provide. In a system with several layers of modules, failures may be passed up through more than one layer before

being discovered and handled. As the number of do-nothing layers increases, containment generally becomes more and more difficult.

- *Be fail-fast.* The module reports at its interface that something has gone wrong. This response also turns the problem over to the designer of the next higher-level system, but in a more graceful way. Example: when an Ethernet transceiver detects a collision on a frame it is sending, it stops sending as quickly as possible, broadcasts a brief jamming signal to ensure that all network participants quickly realize that there was a collision, and it reports the collision to the next higher level, usually a hardware module of which the transceiver is a component, so that the higher level can consider resending that frame.
- *Be fail-safe.* The module transforms any value or values that are incorrect to values that are known to be acceptable, even if not right or optimal. An example is a digital traffic light controller that, when it detects a failure in its sequencer, switches to a blinking red light in all directions. Chapter 11[on-line] discusses systems that provide security. In the event of a failure in a secure system, the safest thing to do is usually to block all access. A fail-safe module designed to do that is said to be *fail-secure*.
- *Be fail-soft.* The system continues to operate correctly with respect to some predictably degraded subset of its specifications, perhaps with some features missing or with lower performance. For example, an airplane with three engines can continue to fly safely, albeit more slowly and with less maneuverability, if one engine fails. A file system that is partitioned into five parts, stored on five different small hard disks, can continue to provide access to 80% of the data when one of the disks fails, in contrast to a file system that employs a single disk five times as large.
- *Mask the error.* Any value or values that are incorrect are made right and the module meets it specification as if the error had not occurred.

We will concentrate on masking errors because the techniques used for that purpose can be applied, often in simpler form, to achieving a fail-fast, fail-safe, or fail-soft system.

As a general rule, one can design algorithms and procedures to cope only with specific, anticipated faults. Further, an algorithm or procedure can be expected to cope only with faults that are actually detected. In most cases, the only workable way to detect a fault is by noticing an incorrect value or control signal; that is, by detecting an error. Thus when trying to determine if a system design has adequate fault tolerance, it is helpful to classify errors as follows:

- A *detectable error* is one that can be detected reliably. If a detection procedure is in place and the error occurs, the system discovers it with near certainty and it becomes a *detected* error.

- A *maskable error* is one for which it is possible to devise a procedure to recover correctness. If a masking procedure is in place and the error occurs, is detected, and is masked, the error is said to be *tolerated*.
- Conversely, an *untolerated* error is one that is undetectable, undetected, unmaskable, or unmasked.

An untolerated error usually leads to a failure of the system. ("Usually," because we could get lucky and still produce a correct output, either because the error values didn't actually matter under the current conditions, or some measure intended to mask a different error incidentally masks this one, too.) This classification of errors is illustrated in Figure 8.3.

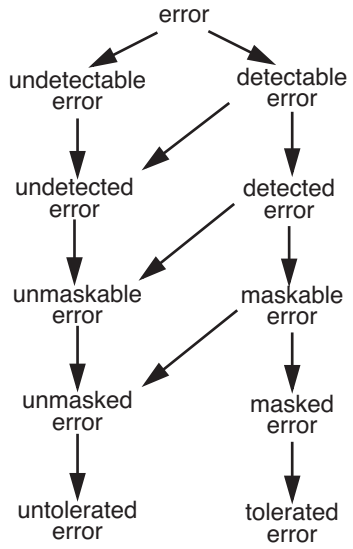


FIGURE 8.3

Classification of errors. Arrows lead from a category to mutually exclusive subcategories. For example, unmasked errors include both unmaskable errors and maskable errors that the designer decides not to mask.

A subtle consequence of the concept of a maskable error is that there must be a well-defined boundary around that part of the system state that might be in error. The masking procedure must restore all of that erroneous state to correctness, using information that has not been corrupted by the error. The real meaning of detectable, then, is that the error is discovered before its consequences have propagated beyond some specified boundary. The designer usually chooses this boundary to coincide with that of some module and designs that module to be fail-fast (that is, it detects and reports its own errors). The system of which the module is a component then becomes responsible for masking the failure of the module.

8.3.2 Fault Tolerance Models

The distinctions among detectable, detected, maskable, and tolerated errors allow us to specify for a system a *fault tolerance model*, one of the components of the fault tolerance design process described in Section 8.1.2, as follows:

1. Analyze the system and categorize possible error events into those that can be reliably detected and those that cannot. At this stage, detectable or not, all errors are untolerated.

2. For each undetectable error, evaluate the probability of its occurrence. If that probability is not negligible, modify the system design in whatever way necessary to make the error reliably detectable.
3. For each detectable error, implement a detection procedure and reclassify the module in which it is detected as fail-fast.
4. For each detectable error try to devise a way of masking it. If there is a way, reclassify this error as a maskable error.
5. For each maskable error, evaluate its probability of occurrence, the cost of failure, and the cost of the masking method devised in the previous step. If the evaluation indicates it is worthwhile, implement the masking method and reclassify this error as a tolerated error.

When finished developing such a model, the designer should have a useful fault tolerance specification for the system. Some errors, which have negligible probability of occurrence or for which a masking measure would be too expensive, are identified as untolerated. When those errors occur the system fails, leaving its users to cope with the result. Other errors have specified recovery algorithms, and when those occur the system should continue to run correctly. A review of the system recovery strategy can now focus separately on two distinct questions:

- Is the designer's list of potential error events complete, and is the assessment of the probability of each error realistic?
- Is the designer's set of algorithms, procedures, and implementations that are supposed to detect and mask the anticipated errors complete and correct?

These two questions are different. The first is a question of models of the real world. It addresses an issue of experience and judgment about real-world probabilities and whether all real-world modes of failure have been discovered or some have gone unnoticed. Two different engineers, with different real-world experiences, may reasonably disagree on such judgments—they may have different models of the real world. The evaluation of modes of failure and of probabilities is a point at which a designer may easily go astray because such judgments must be based not on theory but on experience in the field, either personally acquired by the designer or learned from the experience of others. A new technology, or an old technology placed in a new environment, is likely to create surprises. A wrong judgment can lead to wasted effort devising detection and masking algorithms that will rarely be invoked rather than the ones that are really needed. On the other hand, if the needed experience is not available, all is not lost: the iteration part of the design process is explicitly intended to provide that experience.

The second question is more abstract and also more absolutely answerable, in that an argument for correctness (unless it is hopelessly complicated) or a counterexample to that argument should be something that everyone can agree on. In system design, it is helpful to follow design procedures that distinctly separate these classes of questions. When someone questions a reliability feature, the designer can first ask, "Are you questioning

the correctness of my recovery algorithm or are you questioning my model of what may fail?" and thereby properly focus the discussion or argument.

Creating a fault tolerance model also lays the groundwork for the iteration part of the fault tolerance design process. If a system in the field begins to fail more often than expected, or completely unexpected failures occur, analysis of those failures can be compared with the fault tolerance model to discover what has gone wrong. By again asking the two questions marked with bullets above, the model allows the designer to distinguish between, on the one hand, failure probability predictions being proven wrong by field experience, and on the other, inadequate or misimplemented masking procedures. With this information the designer can work out appropriate adjustments to the model and the corresponding changes needed for the system.

Iteration and review of fault tolerance models is also important to keep them up to date in the light of technology changes. For example, the Network File System described in Section 4.4 was first deployed using a local area network, where packet loss errors are rare and may even be masked by the link layer. When later users deployed it on larger networks, where lost packets are more common, it became necessary to revise its fault tolerance model and add additional error detection in the form of end-to-end checksums. The processor time required to calculate and check those checksums caused some performance loss, which is why its designers did not originally include checksums. But loss of data integrity outweighed loss of performance and the designers reversed the trade-off.

To illustrate, an example of a fault tolerance model applied to a popular kind of memory devices, RAM, appears in Section 8.7. This fault tolerance model employs error detection and masking techniques that are described below in Section 8.4 of this chapter, so the reader may prefer to delay detailed study of that section until completing Section 8.4.

8.4 Systematically Applying Redundancy

The designer of an analog system typically masks small errors by specifying design tolerances known as *margins*, which are amounts by which the specification is better than necessary for correct operation under normal conditions. In contrast, the designer of a digital system both detects and masks errors of all kinds by adding redundancy, either in time or in space. When an error is thought to be transient, as when a packet is lost in a data communication network, one method of masking is to resend it, an example of redundancy in time. When an error is likely to be persistent, as in a failure in reading bits from the surface of a disk, the usual method of masking is with spatial redundancy, having another component provide another copy of the information or control signal. Redundancy can be applied either in cleverly small quantities or by brute force, and both techniques may be used in different parts of the same system.

8.4.1 Coding: Incremental Redundancy

The most common form of incremental redundancy, known as *forward error correction*, consists of clever coding of data values. With data that has not been encoded to tolerate errors, a change in the value of one bit may transform one legitimate data value into another legitimate data value. Encoding for errors involves choosing as the representation of legitimate data values only some of the total number of possible bit patterns, being careful that the patterns chosen for legitimate data values all have the property that to transform any one of them to any other, more than one bit must change. The smallest number of bits that must change to transform one legitimate pattern into another is known as the *Hamming distance* between those two patterns. The Hamming distance is named after Richard Hamming, who first investigated this class of codes. Thus the patterns

```
100101
000111
```

have a Hamming distance of 2 because the upper pattern can be transformed into the lower pattern by flipping the values of two bits, the first bit and the fifth bit. Data fields that have not been coded for errors might have a Hamming distance as small as 1. Codes that can detect or correct errors have a minimum Hamming distance between any two legitimate data patterns of 2 or more. The Hamming distance of a code is the minimum Hamming distance between any pair of legitimate patterns of the code. One can calculate the Hamming distance between two patterns, *A* and *B*, by counting the number of ONEs in $A \oplus B$, where $\oplus$ is the exclusive OR (XOR) operator.

Suppose we create an encoding in which the Hamming distance between *every* pair of legitimate data patterns is 2. Then, if one bit changes accidentally, since no legitimate data item can have that pattern, we can detect that something went wrong, but it is not possible to figure out what the original data pattern was. Thus, if the two patterns above were two members of the code and the first bit of the upper pattern were flipped from ONE to ZERO, there is no way to tell that the result, `000101`, is not the result of flipping the fifth bit of the lower pattern.

Next, suppose that we instead create an encoding in which the Hamming distance of the code is 3 or more. Here are two patterns from such a code; bits 1, 2, and 5 are different:

```
100101
010111
```

Now, a one-bit change will always transform a legitimate data pattern into an incorrect data pattern that is still at least 2 bits distant from any other legitimate pattern but only 1 bit distant from the original pattern. A decoder that receives a pattern with a one-bit error can inspect the Hamming distances between the received pattern and nearby legitimate patterns and by choosing the nearest legitimate pattern correct the error. If 2 bits change, this error-correction procedure will still identify a corrected data value, but it will choose the wrong one. If we expect 2-bit errors to happen often, we could choose the code patterns so that the Hamming distance is 4, in which case the code can correct

1-bit errors and detect 2-bit errors. But a 3-bit error would look just like a 1-bit error in some other code pattern, so it would decode to a wrong value. More generally, if the Hamming distance of a code is *d*, a little analysis reveals that one can detect $d - 1$ errors and correct $\lfloor (d-1)/2 \rfloor$ errors. The reason that this form of redundancy is named "forward" error correction is that the creator of the data performs the coding before storing or transmitting it, and anyone can later decode the data without appealing to the creator. (Chapter 7[on-line] described the technique of asking the sender of a lost frame, packet, or message to retransmit it. That technique goes by the name of *backward error correction*.)

The systematic construction of forward error-detection and error-correction codes is a large field of study, which we do not intend to explore. However, two specific examples of commonly encountered codes are worth examining.

The first example is a simple parity check on a 2-bit value, in which the parity bit is the XOR of the 2 data bits. The coded pattern is 3 bits long, so there are $2^3 = 8$ possible patterns for this 3-bit quantity, only 4 of which represent legitimate data. As illustrated in Figure 8.4, the 4 "correct" patterns have the property that changing any single bit transforms the word into one of the 4 illegal patterns. To transform the coded quantity into another legal pattern, at least 2 bits must change (in other words, the Hamming distance of this code is 2). The conclusion is that a simple parity check can detect any single error, but it doesn't have enough information to correct errors.

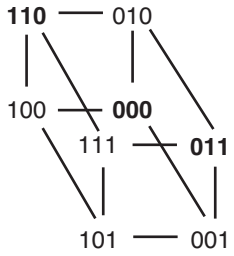


FIGURE 8.4

Patterns for a simple parity-check code. Each line connects patterns that differ in only one bit; **bold-face** patterns are the legitimate ones.

The second example, in Figure 8.5, shows a forward error-correction code that can correct 1-bit errors in a 4-bit data value, by encoding the 4 bits into 7-bit words. In this code, bits P_7, P_6, P_5, and P_3 carry the data, while bits P_4, P_2, and P_1 are calculated from the data bits. (This out-of-order numbering scheme creates a multidimensional binary coordinate system with a use that will be evident in a moment.) We could analyze this code to determine its Hamming distance, but we can also observe that three extra bits can carry exactly enough information to distinguish 8 cases: no error, an error in bit 1, an error in bit 2, … or an error in bit 7. Thus, it is not surprising that an error-correction code can be created. This code calculates bits P_1, P_2, and P_4 as follows:

$$P_1 = P_7 \oplus P_5 \oplus P_3$$
$$P_2 = P_7 \oplus P_6 \oplus P_3$$
$$P_4 = P_7 \oplus P_6 \oplus P_5$$

Now, suppose that the array of bits P_1 through P_7 is sent across a network and noise causes bit P_5 to flip. If the recipient recalculates P_1, P_2, and P_4, the recalculated values of P_1 and P_4 will be different from the received bits P_1 and P_4. The recipient then writes $P_4\ P_2\ P_1$ in order, representing the troubled bits as ONEs and untroubled bits as ZEROs, and notices that their binary value is $101_2 = 5$, the position of the flipped bit. In this code, whenever there is a one-bit error, the troubled parity bits directly identify the bit to correct. (That was the reason for the out-of-order bit-numbering scheme, which created a 3-dimensional coordinate system for locating an erroneous bit.)

The use of 3 check bits for 4 data bits suggests that an error-correction code may not be efficient, but in fact the apparent inefficiency of this example is only because it is so small. Extending the same reasoning, one can, for example, provide single-error correction for 56 data bits using 7 check bits in a 63-bit code word.

In both of these examples of coding, the assumed threat to integrity is that an unidentified bit out of a group may be in error. Forward error correction can also be effective against other threats. A different threat, called *erasure,* is also common in digital systems. An erasure occurs when the value of a particular, identified bit of a group is unintelligible or perhaps even completely missing. Since we know which bit is in question, the simple parity-check code, in which the parity bit is the XOR of the other bits, becomes a forward error correction code. The unavailable bit can be reconstructed simply by calculating the XOR of the unerased bits. Returning to the example of Figure 8.4, if we find a pattern in which the first and last bits have values 0 and 1 respectively, but the middle bit is illegible, the only possibilities are 001 and **011**. Since 001 is not a legitimate code pattern, the original pattern must have been **011**. The simple parity check allows correction of only a single erasure. If there is a threat of multiple erasures, a more complex coding scheme is needed. Suppose, for example, we have 4 bits to protect, and they are coded as in Figure 8.5. In that case, if as many as 3 bits are erased, the remaining 4 bits are sufficient to reconstruct the values of the 3 that are missing.

Since erasure, in the form of lost packets, is a threat in a best-effort packet network, this same scheme of forward error correction is applicable. One might, for example, send four numbered, identical-length packets of data followed by a parity packet that contains

bit	P_7	P_6	P_5	P_4	P_3	P_2	P_1
Choose P_1 so XOR of every other bit ($P_7 \oplus P_5 \oplus P_3 \oplus P_1$) is 0	⊕		⊕		⊕		⊕
Choose P_2 so XOR of every other pair ($P_7 \oplus P_6 \oplus P_3 \oplus P_2$) is 0	⊕	⊕			⊕	⊕	
Choose P_4 so XOR of every other four ($P_7 \oplus P_6 \oplus P_5 \oplus P_4$) is 0	⊕	⊕	⊕	⊕			

FIGURE 8.5

A single-error-correction code. In the table, the symbol ⊕ marks the bits that participate in the calculation of one of the redundant bits. The payload bits are P_7, P_6, P_5, and P_3, and the redundant bits are P_4, P_2, and P_1. The "every other" notes describe a 3-dimensional coordinate system that can locate an erroneous bit.

as its payload the bit-by-bit XOR of the payloads of the previous four. (That is, the first bit of the parity packet is the XOR of the first bit of each of the other four packets; the second bits are treated similarly, etc.) Although the parity packet adds 25% to the network load, as long as any four of the five packets make it through, the receiving side can reconstruct all of the payload data perfectly without having to ask for a retransmission. If the network is so unreliable that more than one packet out of five typically gets lost, then one might send seven packets, of which four contain useful data and the remaining three are calculated using the formulas of Figure 8.5. (Using the numbering scheme of that figure, the payload of packet 4, for example, would consist of the XOR of the payloads of packets 7, 6, and 5.) Now, if any four of the seven packets make it through, the receiving end can reconstruct the data.

Forward error correction is especially useful in broadcast protocols, where the existence of a large number of recipients, each of which may miss different frames, packets, or stream segments, makes the alternative of backward error correction by requesting retransmission unattractive. Forward error correction is also useful when controlling jitter in stream transmission because it eliminates the round-trip delay that would be required in requesting retransmission of missing stream segments. Finally, forward error correction is usually the only way to control errors when communication is one-way or round-trip delays are so long that requesting retransmission is impractical, for example, when communicating with a deep-space probe. On the other hand, using forward error correction to replace lost packets may have the side effect of interfering with congestion control techniques in which an overloaded packet forwarder tries to signal the sender to slow down by discarding an occasional packet.

Another application of forward error correction to counter erasure is in storing data on magnetic disks. The threat in this case is that an entire disk drive may fail, for example because of a disk head crash. Assuming that the failure occurs long after the data was originally written, this example illustrates one-way communication in which backward error correction (asking the original writer to write the data again) is not usually an option. One response is to use a RAID array (see Section 2.1.1.4) in a configuration known as RAID 4. In this configuration, one might use an array of five disks, with four of the disks containing application data and each sector of the fifth disk containing the bit-by-bit XOR of the corresponding sectors of the first four. If any of the five disks fails, its identity will quickly be discovered because disks are usually designed to be fail-fast and report failures at their interface. After replacing the failed disk, one can restore its contents by reading the other four disks and calculating, sector by sector, the XOR of their data (see exercise 8.9). To maintain this strategy, whenever anyone updates a data sector, the RAID 4 system must also update the corresponding sector of the parity disk, as shown in Figure 8.6. That figure makes it apparent that, in RAID 4, forward error correction has an identifiable read and write performance cost, in addition to the obvious increase in the amount of disk space used. Since loss of data can be devastating, there is considerable interest in RAID, and much ingenuity has been devoted to devising ways of minimizing the performance penalty.

Although it is an important and widely used technique, successfully applying incremental redundancy to achieve error detection and correction is harder than one might expect. The first case study of Section 8.8 provides several useful lessons on this point.

In addition, there are some situations where incremental redundancy does not seem to be applicable. For example, there have been efforts to devise error-correction codes for numerical values with the property that the coding is preserved when the values are processed by an adder or a multiplier. While it is not too hard to invent schemes that allow a limited form of error detection (for example, one can verify that residues are consistent, using analogues of casting out nines, which school children use to check their arithmetic), these efforts have not yet led to any generally applicable techniques. The only scheme that has been found to systematically protect data during arithmetic processing is massive redundancy, which is our next topic.

8.4.2 Replication: Massive Redundancy

In designing a bridge or a skyscraper, a civil engineer masks uncertainties in the strength of materials and other parameters by specifying components that are 5 or 10 times as strong as minimally required. The method is heavy-handed, but simple and effective.

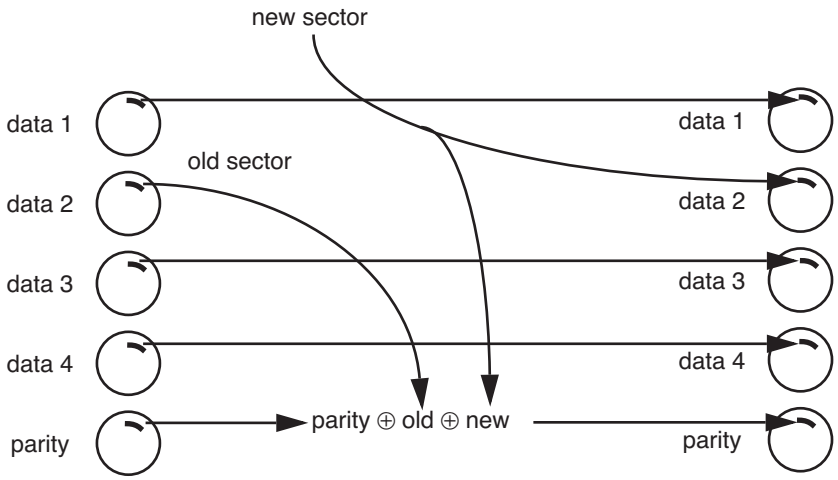


FIGURE 8.6

Update of a sector on disk 2 of a five-disk RAID 4 system. The old parity sector contains parity ← data 1 ⊕ data 2 ⊕ data 3 ⊕ data 4. To construct a new parity sector that includes the new data 2, one could read the corresponding sectors of data 1, data 3, and data 4 and perform three more XORs. But a faster way is to read just the old parity sector and the old data 2 sector and compute the new parity sector as

new parity ← old parity ⊕ old data 2 ⊕ new data 2

The corresponding way of building a reliable system out of unreliable discrete components is to acquire multiple copies of each component. Identical multiple copies are called *replicas*, and the technique is called *replication*. There is more to it than just making copies: one must also devise a plan to arrange or interconnect the replicas so that a failure in one replica is automatically masked with the help of the ones that don't fail. For example, if one is concerned about the possibility that a diode may fail by either shorting out or creating an open circuit, one can set up a network of four diodes as in Figure 8.7, creating what we might call a "superdiode". This interconnection scheme, known as a *quad component*, was developed by Claude E. Shannon and Edward F. Moore in the 1950s as a way of increasing the reliability of relays in telephone systems. It can also be used with resistors and capacitors in circuits that can tolerate a modest range of component values. This particular superdiode can tolerate a single short circuit *and* a single open circuit in any two component diodes, and it can also tolerate certain other multiple failures, such as open circuits in both upper diodes plus a short circuit in one of the lower diodes. If the bridging connection of the figure is added, the superdiode can tolerate additional multiple open-circuit failures (such as one upper diode and one lower diode), but it will be less tolerant of certain short-circuit failures (such as one left diode and one right diode).

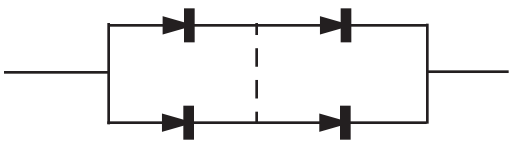
A serious problem with this superdiode is that it masks failures silently. There is no easy way to determine how much failure tolerance remains in the system.

8.4.3 Voting

Although there have been attempts to extend quad-component methods to digital logic, the intricacy of the required interconnections grows much too rapidly. Fortunately, there is a systematic alternative that takes advantage of the static discipline and level regeneration that are inherent properties of digital logic. In addition, it has the nice feature that it can be applied at any level of module, from a single gate on up to an entire computer. The technique is to substitute in place of a single module a set of replicas of that same module, all operating in parallel with the same inputs, and compare their outputs with a device known as a *voter*. This basic strategy is called *N-modular redundancy*, or NMR. When N has the value 3 the strategy is called *triple-modular redundancy*, abbreviated TMR. When other values are used for N the strategy is named by replacing the N of NMR with the number, as in 5MR. The combination of N replicas of some module and

FIGURE 8.7

A quad-component superdiode. The dotted line represents an optional bridging connection, which allows the superdiode to tolerate a different set of failures, as described in the text.



the voting system is sometimes called a *supermodule*. Several different schemes exist for interconnection and voting, only a few of which we explore here.

The simplest scheme, called *fail-vote,* consists of NMR with a majority voter. One assembles N replicas of the module and a voter that consists of an N-way comparator and some counting logic. If a majority of the replicas agree on the result, the voter accepts that result and passes it along to the next system component. If any replicas disagree with the majority, the voter may in addition raise an alert, calling for repair of the replicas that were in the minority. If there is no majority, the voter signals that the supermodule has failed. In failure-tolerance terms, a triply-redundant fail-vote supermodule can mask the failure of any one replica, and it is fail-fast if any two replicas fail in different ways.

If the reliability, as was defined in Section 8.2.2, of a single replica module is R and the underlying fault mechanisms are independent, a TMR fail-vote supermodule will operate correctly if all 3 modules are working (with reliability R^3) or if 1 module has failed and the other 2 are working (with reliability $R^2(1-R)$). Since a single-module failure can happen in 3 different ways, the reliability of the supermodule is the sum,

$$R_{supermodule} = R^3 + 3R^2(1-R) = 3R^2 - 2R^3 \qquad \text{Eq. 8–10}$$

but the supermodule is *not* always fail-fast. If two replicas fail in exactly the same way, the voter will accept the erroneous result and, unfortunately, call for repair of the one correctly operating replica. This outcome is not as unlikely as it sounds because several replicas that went through the same design and production process may have exactly the same set of design or manufacturing faults. This problem can arise despite the independence assumption used in calculating the probability of correct operation. That calculation assumes only that the probability that different replicas produce correct answers be independent; it assumes nothing about the probability of producing specific wrong answers. Without more information about the probability of specific errors and their correlations the only thing we can say about the probability that an incorrect result will be accepted by the voter is that it is not more than

$$(1-R_{supermodule}) = (1 - 3R^2 + 2R^3)$$

These calculations assume that the voter is perfectly reliable. Rather than trying to create perfect voters, the obvious thing to do is replicate them, too. In fact, everything—modules, inputs, outputs, sensors, actuators, etc.—should be replicated, and the final vote should be taken by the client of the system. Thus, three-engine airplanes vote with their propellers: when one engine fails, the two that continue to operate overpower the inoperative one. On the input side, the pilot's hand presses forward on three separate throttle levers. A fully replicated TMR supermodule is shown in Figure 8.8. With this interconnection arrangement, any measurement or estimate of the reliability, R, of a component module should include the corresponding voter. It is actually customary (and more logical) to consider a voter to be a component of the next module in the chain rather than, as the diagram suggests, the previous module. This fully replicated design is sometimes described as *recursive.*

The numerical effect of fail-vote TMR is impressive. If the reliability of a single module at time T is 0.999, equation 8–10 says that the reliability of a fail-vote TMR supermodule at that same time is 0.999997. TMR has reduced the probability of failure from one in a thousand to three in a million. This analysis explains why airplanes intended to fly across the ocean have more than one engine. Suppose that the rate of engine failures is such that a single-engine plane would fail to complete one out of a thousand trans-Atlantic flights. Suppose also that a 3-engine plane can continue flying as long as any 2 engines are operating, but it is too heavy to fly with only 1 engine. In 3 flights out of a thousand, one of the three engines will fail, but if engine failures are independent, in 999 out of each thousand first-engine failures, the remaining 2 engines allow the plane to limp home successfully.

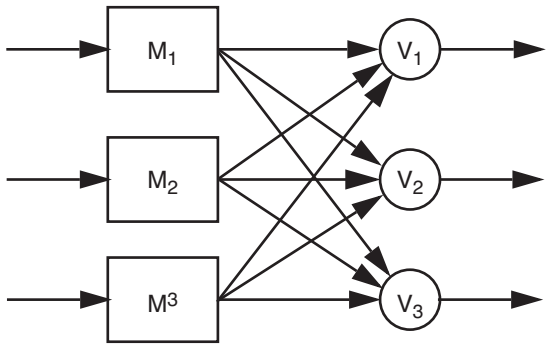
Although TMR has greatly improved reliability, it has not made a comparable impact on MTTF. In fact, the MTTF of a fail-vote TMR supermodule can be *smaller* than the MTTF of the original, single-replica module. The exact effect depends on the failure process of the replicas, so for illustration consider a memoryless failure process, not because it is realistic but because it is mathematically tractable. Suppose that airplane engines have an MTTF of 6,000 hours, they fail independently, the mechanism of engine failure is memoryless, and (since this is a fail-vote design) we need at least 2 operating engines to get home. When flying with three engines, the plane accumulates 6,000 hours of engine running time in only 2,000 hours of flying time, so from the point of view of the airplane as a whole, 2,000 hours is the expected time to the first engine failure. While flying with the remaining two engines, it will take another 3,000 flying hours to accumulate 6,000 more engine hours. Because the failure process is memoryless we can calculate the MTTF of the 3-engine plane by adding:

Mean time to first failure	2000 hours (three engines)
Mean time from first to second failure	3000 hours (two engines)
Total mean time to system failure	5000 hours

Thus the mean time to system failure is less than the 6,000 hour MTTF of a single engine. What is going on here is that we have actually sacrificed long-term reliability in order to enhance short-term reliability. Figure 8.9 illustrates the reliability of our hypo-

FIGURE 8.8

Triple-modular redundant supermodule, with three inputs, three voters, and three outputs.



thetical airplane during its 6 hours of flight, which amounts to only 0.001 of the single-engine MTTF—the mission time is very short compared with the MTTF and the reliability is far higher. Figure 8.10 shows the same curve, but for flight times that are comparable with the MTTF. In this region, if the plane tried to keep flying for 8000 hours (about 1.4 times the single-engine MTTF), a single-engine plane would fail to complete the flight in 3 out of 4 tries, but the 3-engine plane would fail to complete the flight in 5 out of 6 tries. (One should be wary of these calculations because the assumptions of independence and memoryless operation may not be met in practice. Sidebar 8.2 elaborates.)

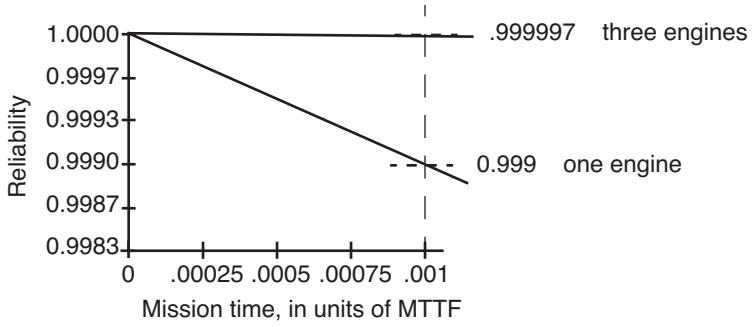


FIGURE 8.9

Reliability with triple modular redundancy, for mission times much less than the MTTF of 6,000 hours. The vertical dotted line represents a six-hour flight.

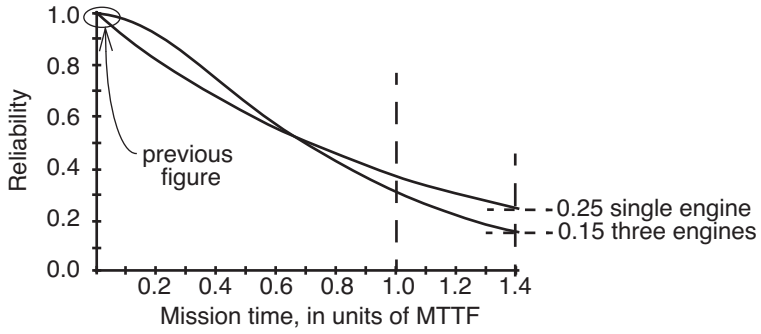


FIGURE 8.10

Reliability with triple modular redundancy, for mission times comparable to the MTTF of 6,000 hours. The two vertical dotted lines represent mission times of 6,000 hours (left) and 8,400 hours (right).

Sidebar 8.2: Risks of manipulating MTTFs The apparently casual manipulation of MTTFs in Sections 8.4.3 and 8.4.4 is justified by assumptions of independence of failures and memoryless processes. But one can trip up by blindly applying this approach without understanding its limitations. To see how, consider a computer system that has been observed for several years to have a hardware crash an average of every 2 weeks and a software crash an average of every 6 weeks. The operator does not repair the system, but simply restarts it and hopes for the best. The composite MTTF is 1.5 weeks, determined most easily by considering what happens if we run the system for, say, 60 weeks. During that time we expect to see

10 software failures
30 hardware failures

40 system failures in 60 weeks —> 1.5 weeks between failure

New hardware is installed, identical to the old except that it never fails. The MTTF should jump to 6 weeks because the only remaining failures are software, right?

Perhaps—but *only* if the software failure process is independent of the hardware failure process.

Suppose the software failure occurs because there is a bug (fault) in a clock-updating procedure: The bug always crashes the system exactly 420 hours (2 1/2 weeks) after it is started—if it gets a chance to run that long. The old hardware was causing crashes so often that the software bug only occasionally had a chance to do its thing—only about once every 6 weeks. Most of the time, the recovery from a hardware failure, which requires restarting the system, had the side effect of resetting the process that triggered the software bug. So, when the new hardware is installed, the system has an MTTF of only 2.5 weeks, much less than hoped.

MTTF's are useful, but one must be careful to understand what assumptions go into their measurement and use.

If we had assumed that the plane could limp home with just one engine, the MTTF would have increased, rather than decreased, but only modestly. Replication provides a dramatic improvement in reliability for missions of duration short compared with the MTTF, but the MTTF itself changes much less. We can verify this claim with a little more analysis, again assuming memoryless failure processes to make the mathematics tractable. Suppose we have an NMR system with the property that it somehow continues to be useful as long as at least one replica is still working. (This system requires using fail-fast replicas and a cleverer voter, as described in Section 8.4.4 below.) If a single replica has an $MTTF_{replica} = 1$, there are N independent replicas, and the failure process is memoryless, the expected time until the first failure is $MTTF_{replica}/N$, the expected time from then until the second failure is $MTTF_{replica}/(N-1)$, etc., and the expected time until the system of N replicas fails is the sum of these times,

$$MTTF_{system} = 1 + 1/2 + 1/3 + \ldots(1/N) \qquad \text{Eq. 8–11}$$

which for large N is approximately $ln(N)$. As we add to the cost by adding more replicas, $MTTF_{system}$ grows disappointingly slowly—proportional to the logarithm of the cost. To multiply the $MTTF_{system}$ by K, the number of replicas required is e^K —the cost grows exponentially. The significant conclusion is that *in systems for which the mission time is long compared with $MTTF_{replica}$, simple replication escalates the cost while providing little benefit.* On the other hand, there is a way of making replication effective for long missions, too. The method is to enhance replication by adding *repair*.

8.4.4 Repair

Let us return now to a fail-vote TMR supermodule (that is, it requires that at least two replicas be working) in which the voter has just noticed that one of the three replicas is producing results that disagree with the other two. Since the voter is in a position to report which replica has failed, suppose that it passes such a report along to a repair person who immediately examines the failing replica and either fixes or replaces it. For this approach, the mean time to repair (MTTR) measure becomes of interest. The supermodule fails if either the second or third replica fails before the repair to the first one can be completed. Our intuition is that if the MTTR is small compared with the combined MTTF of the other two replicas, the chance that the supermodule fails will be similarly small.

The exact effect on chances of supermodule failure depends on the shape of the reliability function of the replicas. In the case where the failure and repair processes are both memoryless, the effect is easy to calculate. Since the rate of failure of 1 replica is $1/MTTF$, the rate of failure of 2 replicas is $2/MTTF$. If the repair time is short compared with $MTTF$ the probability of a failure of 1 of the 2 remaining replicas while waiting a time T for repair of the one that failed is approximately $2T/MTTF$. Since the mean time to repair is MTTR, we have

$$Pr(\text{supermodule fails while waiting for repair}) = \frac{2 \times MTTR}{MTTF} \qquad \text{Eq. 8–12}$$

Continuing our airplane example and temporarily suspending disbelief, suppose that during a long flight we send a mechanic out on the airplane's wing to replace a failed engine. If the replacement takes 1 hour, the chance that one of the other two engines fails during that hour is approximately 1/3000. Moreover, once the replacement is complete, we expect to fly another 2000 hours until the next engine failure. Assuming further that the mechanic is carrying an unlimited supply of replacement engines, completing a 10,000 hour flight—or even a longer one—becomes plausible. The general formula for the MTTF of a fail-vote TMR supermodule with memoryless failure and repair processes is (this formula comes out of the analysis of continuous-transition birth-and-death Markov processes, an advanced probability technique that is beyond our scope):

$$MTTF_{\text{supermodule}} = \frac{MTTF_{replica}}{3} \times \frac{MTTF_{replica}}{2 \times MTTR_{replica}} = \frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} \qquad \text{Eq. 8–13}$$

Thus, our 3-engine plane with hypothetical in-flight repair has an MTTF of 6 million hours, an enormous improvement over the 6000 hours of a single-engine plane. This equation can be interpreted as saying that, compared with an unreplicated module, the MTTF has been reduced by the usual factor of 3 because there are 3 replicas, but at the same time the availability of repair has increased the MTTF by a factor equal to the ratio of the MTTF of the remaining 2 engines to the MTTR.

Replacing an airplane engine in flight may be a fanciful idea, but replacing a magnetic disk in a computer system on the ground is quite reasonable. Suppose that we store 3 replicas of a set of data on 3 independent hard disks, each of which has an MTTF of 5 years (using as the MTTF the expected operational lifetime, not the "MTTF" derived from the short-term failure rate). Suppose also, that if a disk fails, we can locate, install, and copy the data to a replacement disk in an average of 10 hours. In that case, by eq. 8–13, the MTTF of the data is

$$\frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} = \frac{(5 \text{ years})^2}{6 \cdot (10 \text{ hours})/(8760 \text{ hours/year})} = 3650 \text{ years} \qquad \text{Eq. 8–14}$$

In effect, redundancy plus repair has reduced the probability of failure of this supermodule to such a small value that for all practical purposes, failure can be neglected and the supermodule can operate indefinitely.

Before running out to start a company that sells superbly reliable disk-storage systems, it would be wise to review some of the overly optimistic assumptions we made in getting that estimate of the MTTF, most of which are not likely to be true in the real world:

- *Disks fail independently.* A batch of real world disks may all come from the same vendor, where they acquired the same set of design and manufacturing faults. Or, they may all be in the same machine room, where a single earthquake—which probably has an MTTF of less than 3,650 years—may damage all three.
- *Disk failures are memoryless.* Real-world disks follow a bathtub curve. If, when disk #1 fails, disk #2 has already been in service for three years, disk #2 no longer has an expected operational lifetime of 5 years, so the chance of a second failure while waiting for repair is higher than the formula assumes. Furthermore, when disk #1 is replaced, its chances of failing are probably higher than usual for the first few weeks.
- *Repair is also a memoryless process.* In the real world, if we stock enough spares that we run out only once every 10 years and have to wait for a shipment from the factory, but doing a replacement happens to run us out of stock today, we will probably still be out of stock tomorrow and the next day.
- *Repair is done flawlessly.* A repair person may replace the wrong disk, forget to copy the data to the new disk, or install a disk that hasn't passed burn-in and fails in the first hour.

Each of these concerns acts to reduce the reliability below what might be expected from our overly simple analysis. Nevertheless, NMR with repair remains a useful technique, and in Chapter 10[on-line] we will see ways in which it can be applied to disk storage.

One of the most powerful applications of NMR is in the masking of transient errors. When a transient error occurs in one replica, the NMR voter immediately masks it. Because the error is transient, the subsequent behavior of the supermodule is as if repair happened by the next operation cycle. The numerical result is little short of extraordinary. For example, consider a processor arithmetic logic unit (ALU) with a 1 gigahertz clock and which is triply replicated with voters checking its output at the end of each clock cycle. In equation 8–13 we have $MTTR_{replica} = 1$ (in this application, equation 8–13 is only an approximation because the time to repair is a constant rather than the result of a memoryless process), and $MTTF_{supermodule} = (MTTF_{replica})^2/6$ cycles. If $MTTF_{replica}$ is 10^{10} cycles (1 error in 10 billion cycles, which at this clock speed means one error every 10 seconds), $MTTF_{supermodule}$ is $10^{20}/6$ cycles, about 500 years. TMR has taken three ALUs that were for practical use nearly worthless and created a super-ALU that is almost infallible.

The reason things seem so good is that we are evaluating the chance that two transient errors occur in the same operation cycle. If transient errors really are independent, that chance is small. This effect is powerful, but the leverage works in both directions, thereby creating a potential hazard: it is especially important to keep track of the rate at which transient errors actually occur. If they are happening, say, 20 times as often as hoped, $MTTF_{supermodule}$ will be 1/400 of the original prediction—the super-ALU is likely to fail once per year. That may still be acceptable for some applications, but it is a big change. Also, as usual, the assumption of independence is absolutely critical. If all the ALUs came from the same production line, it seems likely that they will have at least some faults in common, in which case the super-ALU may be just as worthless as the individual ALUs.

Several variations on the simple fail-vote structure appear in practice:

- *Purging.* In an NMR design with a voter, whenever the voter detects that one replica disagrees with the majority, the voter calls for its repair and in addition marks that replica DOWN and ignores its output until hearing that it has been repaired. This technique doesn't add anything to a TMR design, but with higher levels of replication, as long as replicas fail one at a time and any two replicas continue to operate correctly, the supermodule works.

- *Pair-and-compare.* Create a fail-fast module by taking two replicas, giving them the same inputs, and connecting a simple comparator to their outputs. As long as the comparator reports that the two replicas of a pair agree, the next stage of the system accepts the output. If the comparator detects a disagreement, it reports that the module has failed. The major attraction of pair-and-compare is that it can be used to create fail-fast modules starting with easily available commercial, off-the-shelf components, rather than commissioning specialized fail-fast versions. Special high-reliability components typically have a cost that is much higher than off-the-shelf designs, for two reasons. First, since they take more time to design and test,

the ones that are available are typically of an older, more expensive technology. Second, they are usually low-volume products that cannot take advantage of economies of large-scale production. These considerations also conspire to produce long delivery cycles, making it harder to keep spares in stock. An important aspect of using standard, high-volume, low-cost components is that one can afford to keep a stock of spares, which in turn means that MTTR can be made small: just replace a failing replica with a spare (the popular term for this approach is *pair-and-spare*) and do the actual diagnosis and repair at leisure.

- *NMR with fail-fast replicas.* If each of the replicas is itself a fail-fast design (perhaps using pair-and-compare internally), then a voter can restrict its attention to the outputs of only those replicas that claim to be producing good results and ignore those that are reporting that their outputs are questionable. With this organization, a TMR system can continue to operate even if 2 of its 3 replicas have failed, since the 1 remaining replica is presumably checking its own results. An NMR system with repair and constructed of fail-fast replicas is so robust that it is unusual to find examples for which N is greater than 2.

Figure 8.11 compares the ability to continue operating until repair arrives of 5MR designs that use fail-vote, purging, and fail-fast replicas. The observant reader will note that this chart can be deemed guilty of a misleading comparison, since it claims that the 5MR system continues working when only one fail-fast replica is still running. But if that fail-fast replica is actually a pair-and-compare module, it might be more accurate to say that there are two still-working replicas at that point.

Another technique that takes advantage of repair, can improve availability, and can degrade gracefully (in other words, it can be fail-soft) is called *partition*. If there is a choice of purchasing a system that has either one fast processor or two slower processors, the two-processor system has the virtue that when one of its processors fails, the system

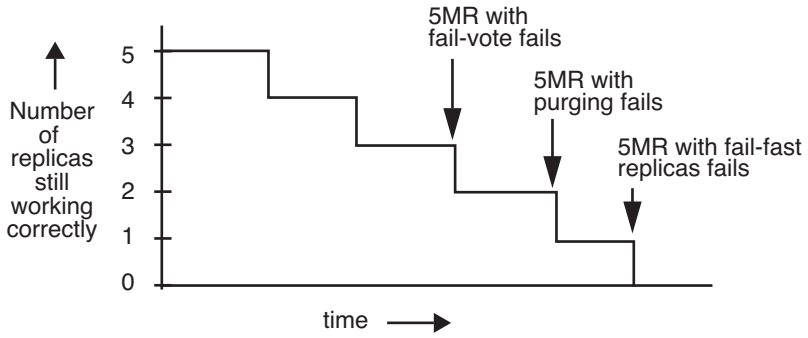


FIGURE 8.11

Failure points of three different 5MR supermodule designs, if repair does not happen in time.

can continue to operate with half of its usual capacity until someone can repair the failed processor. An electric power company, rather than installing a single generator of capacity K megawatts, may install N generators of capacity K/N megawatts each.

When equivalent modules can easily share a load, partition can extend to what is called $N + 1$ *redundancy*. Suppose a system has a load that would require the capacity of N equivalent modules. The designer partitions the load across $N + 1$ or more modules. Then, if any one of the modules fails, the system can carry on at full capacity until the failed module can be repaired.

$N + 1$ redundancy is most applicable to modules that are completely interchangeable, can be dynamically allocated, and are not used as storage devices. Examples are processors, dial-up modems, airplanes, and electric generators. Thus, one extra airplane located at a busy hub can mask the failure of any single plane in an airline's fleet. When modules are not completely equivalent (for example, electric generators come in a range of capacities, but can still be interconnected to share load), the design must ensure that the spare capacity is greater than the capacity of the largest individual module. For devices that provide storage, such as a hard disk, it is also possible to apply partition and $N + 1$ redundancy with the same goals, but it requires a greater level of organization to preserve the stored contents when a failure occurs, for example by using RAID, as was described in Section 8.4.1, or some more general replica management system such as those discussed in Section 10.3.7.

For some applications an occasional interruption of availability is acceptable, while in others every interruption causes a major problem. When repair is part of the fault tolerance plan, it is sometimes possible, with extra care and added complexity, to design a system to provide *continuous operation.* Adding this feature requires that when failures occur, one can quickly identify the failing component, remove it from the system, repair it, and reinstall it (or a replacement part) all without halting operation of the system. The design required for continuous operation of computer hardware involves connecting and disconnecting cables and turning off power to some components but not others, without damaging anything. When hardware is designed to allow connection and disconnection from a system that continues to operate, it is said to allow *hot swap*.

In a computer system, continuous operation also has significant implications for the software. Configuration management software must anticipate hot swap so that it can stop using hardware components that are about to be disconnected, as well as discover newly attached components and put them to work. In addition, maintaining state is a challenge. If there are periodic consistency checks on data, those checks (and repairs to data when the checks reveal inconsistencies) must be designed to work correctly even though the system is in operation and the data is perhaps being read and updated by other users at the same time.

Overall, continuous operation is not a feature that should be casually added to a list of system requirements. When someone suggests it, it may be helpful to point out that it is much like trying to keep an airplane flying indefinitely. Many large systems that appear to provide continuous operation are actually designed to stop occasionally for maintenance.

8.5 Applying Redundancy to Software and Data

The examples of redundancy and replication in the previous sections all involve hardware. A seemingly obvious next step is to apply the same techniques to software and to data. In the case of software the goal is to reduce the impact of programming errors, while in the case of data the goal is to reduce the impact of any kind of hardware, software, or operational error that might affect its integrity. This section begins the exploration of several applicable techniques: N-version programming, valid construction, and building a firewall to separate stored state into two categories: state whose integrity must be preserved and state that can casually be abandoned because it is easy to reconstruct.

8.5.1 Tolerating Software Faults

Simply running three copies of the same buggy program is likely to produce three identical incorrect results. NMR requires independence among the replicas, so the designer needs a way of introducing that independence. An example of a way of introducing independence is found in the replication strategy for the root name servers of the Internet Domain Name System (DNS, described in Section 4.4). Over the years, slightly different implementations of the DNS software have evolved for different operating systems, so the root name server replicas intentionally employ these different implementations to reduce the risk of replicated errors.

To try to harness this idea more systematically, one can commission several teams of programmers and ask each team to write a complete version of an application according to a single set of specifications. Then, run these several versions in parallel and compare their outputs. The hope is that the inevitable programming errors in the different versions will be independent and voting will produce a reliable system. Experiments with this technique, known as *N-version programming*, suggest that the necessary independence is hard to achieve. Different programmers may be trained in similar enough ways that they make the same mistakes. Use of the same implementation language may encourage the same errors. Ambiguities in the specification may be misinterpreted in the same way by more than one team and the specification itself may contain errors. Finally, it is hard to write a specification in enough detail that the outputs of different implementations can be expected to be bit-for-bit identical. The result is that after much effort, the technique may still mask only a certain class of bugs and leave others unmasked. Nevertheless, there are reports that *N*-version programming has been used, apparently with success, in at least two safety-critical aerospace systems, the flight control system of the Boeing 777 aircraft (with $N = 3$) and the on-board control system for the Space Shuttle (with $N = 2$).

Incidentally, the strategy of employing multiple design teams can also be applied to hardware replicas, with a goal of increasing the independence of the replicas by reducing the chance of replicated design errors and systematic manufacturing defects.

Much of software engineering is devoted to a different approach: devising specification and programming techniques that avoid faults in the first place and test techniques

that systematically root out faults so that they can be repaired once and for all before deploying the software. This approach, sometimes called *valid construction*, can dramatically reduce the number of software faults in a delivered system, but because it is difficult both to completely specify and to completely test a system, some faults inevitably remain. Valid construction is based on the observation that software, unlike hardware, is not subject to wear and tear, so if it is once made correct, it should stay that way. Unfortunately, this observation can turn out to be wishful thinking, first because it is hard to make software correct, and second because it is nearly always necessary to make changes after installing a program because the requirements, the environment surrounding the program, or both, have changed. There is thus a potential for tension between valid construction and the principle that one should *design for iteration*.

Worse, later maintainers and reworkers often do not have a complete understanding of the ground rules that went into the original design, so their work is likely to introduce new faults for which the original designers did not anticipate providing tests. Even if the original design is completely understood, when a system is modified to add features that were not originally planned, the original ground rules may be subjected to some violence. Software faults more easily creep into areas that lack systematic design.

8.5.2 Tolerating Software (and other) Faults by Separating State

Designers of reliable systems usually assume that, despite the best efforts of programmers there will always be a residue of software faults, just as there is also always a residue of hardware, operation, and environment faults. The response is to develop a strategy for tolerating all of them. Software adds the complication that the current state of a running program tends to be widely distributed. Parts of that state may be in non-volatile storage, while other parts are in temporary variables held in volatile memory locations, processor registers, and kernel tables. This wide distribution of state makes containment of errors problematic. As a result, when an error occurs, any strategy that involves stopping some collection of running threads, tinkering to repair the current state (perhaps at the same time replacing a buggy program module), and then resuming the stopped threads is usually unrealistic.

In the face of these observations, a programming discipline has proven to be effective: systematically divide the current state of a running program into two mutually exclusive categories and separate the two categories with a firewall. The two categories are:

- State that the system can safely abandon in the event of a failure.
- State whose integrity the system should preserve despite failure.

Upon detecting a failure, the plan becomes to abandon all state in the first category and instead concentrate just on maintaining the integrity of the data in the second category. An important part of the strategy is an important *sweeping simplification*: classify the state of running threads (that is, the thread table, stacks, and registers) as abandonable. When a failure occurs, the system abandons the thread or threads that were running at the time and instead expects a restart procedure, the system operator, or the individual

user to start a new set of threads with a clean slate. The new thread or threads can then, working with only the data found in the second category, verify the integrity of that data and return to normal operation. The primary challenge then becomes to build a firewall that can protect the integrity of the second category of data despite the failure.

The designer can base a natural firewall on the common implementations of volatile (e.g., CMOS memory) and non-volatile (e.g., magnetic disk) storage. As it happens, writing to non-volatile storage usually involves mechanical movement such as rotation of a disk platter, so most transfers move large blocks of data to a limited region of addresses, using a GET/PUT interface. On the other hand, volatile storage technologies typically provide a READ/WRITE interface that allows rapid-fire writes to memory addresses chosen at random, so failures that originate in or propagate to software tend to quickly and untraceably corrupt random-access data. By the time an error is detected the software may thus have already damaged a large and unidentifiable part of the data in volatile memory. The GET/PUT interface instead acts as a bottleneck on the rate of spread of data corruption. The goal can be succinctly stated: to detect failures and stop the system before it reaches the next PUT operation, thus making the volatile storage medium the error containment boundary. It is only incidental that volatile storage usually has a READ/WRITE interface, while non-volatile storage usually has a GET/PUT interface, but because that is usually true it becomes a convenient way to implement and describe the firewall.

This technique is widely used in systems whose primary purpose is to manage long-lived data. In those systems, two aspects are involved:

- Prepare for failure by recognizing that all state in volatile memory devices can vanish at any instant, without warning. When it does vanish, automatically launch new threads that start by restoring the data in non-volatile storage to a consistent, easily described state. The techniques to do this restoration are called *recovery*. Doing recovery systematically involves atomicity, which is explored in Chapter 9[on-line].
- Protect the data in non-volatile storage using replication, thus creating the class of storage known as *durable* storage. Replicating data can be a straightforward application of redundancy, so we will begin the topic in this chapter. However, there are more effective designs that make use of atomicity and geographical separation of replicas, so we will revisit durability in Chapter 10[on-line].

When the volatile storage medium is CMOS RAM and the non-volatile storage medium is magnetic disk, following this programming discipline is relatively straightforward because the distinctively different interfaces make it easy to remember where to place data. But when a one-level store is in use, giving the appearance of random access to all storage, or the non-volatile medium is flash memory, which allows fast random access, it may be necessary for the designer to explicitly specify both the firewall mechanism and which data items are to reside on each side of the firewall.

A good example of the firewall strategy can be found in most implementations of Internet Domain Name System servers. In a typical implementation the server stores the authoritative name records for its domain on magnetic disk, and copies those records into volatile CMOS memory either at system startup or the first time it needs a particular record. If the server fails for any reason, it simply abandons the volatile memory and restarts. In some implementations, the firewall is reinforced by not having any PUT operations in the running name server. Instead, the service updates the authoritative name records using a separate program that runs when the name server is off-line.

In addition to employing independent software implementations and a firewall between categories of data, DNS also protects against environmental faults by employing geographical separation of its replicas, a topic that is explored more deeply in Section 10.3[on-line]. The three techniques taken together make DNS quite fault tolerant.

8.5.3 Durability and Durable Storage

For the discipline just described to work, we need to make the result of a PUT operation durable. But first we must understand just what "durable" means. *Durability* is a specification of how long the result of an action must be preserved after the action completes. One must be realistic in specifying durability because there is no such thing as perfectly durable storage in which the data will be remembered forever. However, by choosing enough genuinely independent replicas, and with enough care in management, one can meet any reasonable requirement.

Durability specifications can be roughly divided into four categories, according to the length of time that the application requires that data survive. Although there are no bright dividing lines, as one moves from one category to the next the techniques used to achieve durability tend to change.

- *Durability no longer than the lifetime of the thread that created the data.* For this case, it is usually adequate to place the data in volatile memory.

For example, an action such as moving the gearshift may require changing the operating parameters of an automobile engine. The result must be reliably remembered, but only until the next shift of gears or the driver switches the engine off.

The operations performed by calls to the kernel of an operating system provide another example. The CHDIR procedure of the UNIX kernel (see Table 2.1 in Section 2.5.1) changes the working directory of the currently running process. The kernel state variable that holds the name of the current working directory is a value in volatile RAM that does not need to survive longer than this process.

For a third example, the registers and cache of a hardware processor usually provide just the first category of durability. If there is a failure, the plan is to abandon those values along with the contents of volatile memory, so there is no need for a higher level of durability.

- *Durability for times short compared with the expected operational lifetime of non-volatile storage media such as magnetic disk or flash memory.* A designer typically

implements this category of durability by writing one copy of the data in the non-volatile storage medium.

Returning to the automotive example, there may be operating parameters such as engine timing that, once calibrated, should be durable at least until the next tune-up, not just for the life of one engine use session. Data stored in a cache that writes through to a non-volatile medium has about this level of durability. As a third example, a remote procedure call protocol that identifies duplicate messages by recording nonces might write old nonce values (see Section 7.5.3) to a non-volatile storage medium, knowing that the real goal is not to remember the nonces forever, but rather to make sure that the nonce record outlasts the longest retry timer of any client. Finally, text editors and word-processing systems typically write temporary copies on magnetic disk of the material currently being edited so that if there is a system crash or power failure the user does not have to repeat the entire editing session. These temporary copies need to survive only until the end of the current editing session.

- *Durability for times comparable to the expected operational lifetime of non-volatile storage media.* Because actual non-volatile media lifetimes vary quite a bit around the expected lifetime, implementation generally involves placing replicas of the data on independent instances of the non-volatile media.

This category of durability is the one that is usually called *durable storage* and it is the category for which the next section of this chapter develops techniques for implementation. Users typically expect files stored in their file systems and data managed by a database management system to have this level of durability. Section 10.3[on-line] revisits the problem of creating durable storage when replicas are geographically separated.

- *Durability for many multiples of the expected operational lifetime of non-volatile storage media.*

This highest level of durability is known as *preservation*, and is the specialty of archivists. In addition to making replicas and keeping careful records, it involves copying data from one non-volatile medium to another before the first one deteriorates or becomes obsolete. Preservation also involves (sometimes heroic) measures to preserve the ability to correctly interpret idiosyncratic formats created by software that has long since become obsolete. Although important, it is a separate topic, so preservation is not discussed any further here.

8.5.4 Magnetic Disk Fault Tolerance

In principle, durable storage can be constructed starting with almost any storage medium, but it is most straightforward to use non-volatile devices. Magnetic disks (see Sidebar 2.8) are widely used as the basis for durable storage because of their low cost, large capacity and non-volatility—they retain their memory when power is turned off or is accidentally disconnected. Even if power is lost during a write operation, at most a small block of data surrounding the physical location that was being written is lost, and

disks can be designed with enough internal power storage and data buffering to avoid even that loss. In its raw form, a magnetic disk is remarkably reliable, but it can still fail in various ways and much of the complexity in the design of disk systems consists of masking these failures.

Conventionally, magnetic disk systems are designed in three nested layers. The innermost layer is the spinning disk itself, which provides what we will call *raw storage*. The next layer is a combination of hardware and firmware of the disk controller that provides for detecting the failures in the raw storage layer; it creates *fail-fast storage*. Finally, the hard disk firmware adds a third layer that takes advantage of the detection features of the second layer to create a substantially more reliable storage system, known as *careful storage*. Most disk systems stop there, but high-availability systems add a fourth layer to create *durable storage*. This section develops a disk failure model and explores error masking techniques for all four layers.

In early disk designs, the disk controller presented more or less the raw disk interface, and the fail-fast and careful layers were implemented in a software component of the operating system called the disk driver. Over the decades, first the fail-fast layer and more recently part or all of the careful layer of disk storage have migrated into the firmware of the disk controller to create what is known in the trade as a "hard drive". A hard drive usually includes a RAM buffer to hold a copy of the data going to and from the disk, both to avoid the need to match the data rate to and from the disk head with the data rate to and from the system memory and also to simplify retries when errors occur. RAID systems, which provide a form of durable storage, generally are implemented as an additional hardware layer that incorporates mass-market hard drives. One reason for this move of error masking from the operating system into the disk controller is that as computational power has gotten cheaper, the incremental cost of a more elaborate firmware design has dropped. A second reason may explain the obvious contrast with the lack of enthusiasm for memory parity checking hardware that is mentioned in Section 8.8.1. A transient memory error is all but indistinguishable from a program error, so the hardware vendor is not likely to be blamed for it. On the other hand, most disk errors have an obvious source, and hard errors are not transient. Because blame is easy to place, disk vendors have a strong motivation to include error masking in their designs.

8.5.4.1 Magnetic Disk Fault Modes

Sidebar 2.8 described the physical design of the magnetic disk, including platters, magnetic material, read/write heads, seek arms, tracks, cylinders, and sectors, but it did not make any mention of disk reliability. There are several considerations:

- Disks are high precision devices made to close tolerances. Defects in manufacturing a recording surface typically show up in the field as a sector that does not reliably record data. Such defects are a source of hard errors. Deterioration of the surface of a platter with age can cause a previously good sector to fail. Such loss is known as *decay* and, since any data previously recorded there is lost forever, decay is another example of hard error.

- Since a disk is mechanical, it is subject to wear and tear. Although a modern disk is a sealed unit, deterioration of its component materials as they age can create dust. The dust particles can settle on a magnetic surface, where they may interfere either with reading or writing. If interference is detected, then re-reading or re-writing that area of the surface, perhaps after jiggling the seek arm back and forth, may succeed in getting past the interference, so the fault may be transient. Another source of transient faults is electrical noise spikes. Because disk errors caused by transient faults can be masked by retry, they fall in the category of soft errors.
- If a running disk is bumped, the shock may cause a head to hit the surface of a spinning platter, causing what is known as a head crash. A head crash not only may damage the head and destroy the data at the location of impact, it also creates a cloud of dust that interferes with the operation of heads on other platters. A head crash generally results in several sectors decaying simultaneously. A set of sectors that tend to all fail together is known as a *decay set*. A decay set may be quite large, for example all the sectors on one drive or on one disk platter.
- As electronic components in the disk controller age, clock timing and signal detection circuits can go out of tolerance, causing previously good data to become unreadable, or bad data to be written, either intermittently or permanently. In consequence, electronic component tolerance problems can appear either as soft or hard errors.
- The mechanical positioning systems that move the seek arm and that keep track of the rotational position of the disk platter can fail in such a way that the heads read or write the wrong track or sector within a track. This kind of fault is known as a *seek error*.

8.5.4.2 System Faults

In addition to failures within the disk subsystem, there are at least two threats to the integrity of the data on a disk that arise from outside the disk subsystem:

- If the power fails in the middle of a disk write, the sector being written may end up being only partly updated. After the power is restored and the system restarts, the next reader of that sector may find that the sector begins with the new data, but ends with the previous data.
- If the operating system fails during the time that the disk is writing, the data being written could be affected, even if the disk is perfect and the rest of the system is fail-fast. The reason is that all the contents of volatile memory, including the disk buffer, are inside the fail-fast error containment boundary and thus at risk of damage when the system fails. As a result, the disk channel may correctly write on the disk what it reads out of the disk buffer in memory, but the faltering operating system may have accidentally corrupted the contents of that buffer after the

application called PUT. In such cases, the data that ends up on the disk will be corrupted, but there is no sure way in which the disk subsystem can detect the problem.

8.5.4.3 Raw Disk Storage

Our goal is to devise systematic procedures to mask as many of these different faults as possible. We start with a model of disk operation from a programmer's point of view. The raw disk has, at least conceptually, a relatively simple interface: There is an operation to seek to a (numbered) track, an operation that writes data on the track and an operation that reads data from the track. The failure model is simple: all errors arising from the failures just described are untolerated. (In the procedure descriptions, arguments are call-by-reference, and GET operations read from the disk into the argument named *data*.)
The raw disk layer implements these storage access procedures and failure tolerance model:

```
RAW_SEEK (track)    // Move read/write head into position.
RAW_PUT (data)      // Write entire track.
RAW_GET (data)      // Read entire track.
```

- error-free operation: RAW_SEEK moves the seek arm to position *track*. RAW_GET returns whatever was most recently written by RAW_PUT at position *track*.
- untolerated error: On any given attempt to read from or write to a disk, dust particles on the surface of the disk or a temporarily high noise level may cause data to be read or written incorrectly. (soft error)
- untolerated error: A spot on the disk may be defective, so all attempts to write to any track that crosses that spot will be written incorrectly. (hard error)
- untolerated error: Information previously written correctly may decay, so RAW_GET returns incorrect data. (hard error)
- untolerated error: When asked to read data from or write data to a specified track, a disk may correctly read or write the data, but on the wrong track. (seek error)
- untolerated error: The power fails during a RAW_PUT with the result that only the first part of *data ends* up being written on *track*. The remainder of *track* may contain older data.
- untolerated error: The operating system crashes during a RAW_PUT and scribbles over the disk buffer in volatile storage, so RAW_PUT writes corrupted data on one track of the disk.

8.5.4.4 Fail-Fast Disk Storage

The fail-fast layer is the place where the electronics and microcode of the disk controller divide the raw disk track into sectors. Each sector is relatively small, individually protected with an error-detection code, and includes in addition to a fixed-sized space for data a sector and track number. The error-detection code enables the disk controller to

return a status code on FAIL_FAST_GET that tells whether a sector read correctly or incorrectly, and the sector and track numbers enable the disk controller to verify that the seek ended up on the correct track. The FAIL_FAST_PUT procedure not only writes the data, but it verifies that the write was successful by reading the newly written sector on the next rotation and comparing it with the data still in the write buffer. The sector thus becomes the minimum unit of reading and writing, and the disk address becomes the pair {*track*, *sector_number*}. For performance enhancement, some systems allow the caller to bypass the verification step of FAIL_FAST_PUT. When the client chooses this bypass, write failures become indistinguishable from decay events.

There is always a possibility that the data on a sector is corrupted in such a way that the error-detection code accidentally verifies. For completeness, we will identify that case as an untolerated error, but point out that the error-detection code should be powerful enough that the probability of this outcome is negligible.

The fail-fast layer implements these storage access procedures and failure tolerance model:

status ← FAIL_FAST_SEEK (*track*)
status ← FAIL_FAST_PUT (*data*, *sector_number*)
status ← FAIL_FAST_GET (*data*, *sector_number*)

- error-free operation: FAIL_FAST_SEEK moves the seek arm to *track*. FAIL_FAST_GET returns whatever was most recently written by FAIL_FAST_PUT at *sector_number* on *track* and returns *status* = OK.
- detected error: FAIL_FAST_GET reads the data, checks the error-detection code and finds that it does not verify. The cause may a soft error, a hard error due to decay, or a hard error because there is a bad spot on the disk and the invoker of a previous FAIL_FAST_PUT chose to bypass verification. FAIL_FAST_GET does not attempt to distinguish these cases; it simply reports the error by returning *status* = BAD.
- detected error: FAIL_FAST_PUT writes the data, on the next rotation reads it back, checks the error-detection code, finds that it does not verify, and reports the error by returning *status* = BAD.
- detected error: FAIL_FAST_SEEK moves the seek arm, reads the permanent track number in the first sector that comes by, discovers that it does not match the requested track number (or that the sector checksum does not verify), and reports the error by returning *status* = BAD.
- detected error: The caller of FAIL_FAST_PUT tells it to bypass the verification step, so FAIL_FAST_PUT always reports *status* = OK even if the sector was not written correctly. But a later caller of FAIL_FAST_GET that requests that sector should detect any such error.
- detected error: The power fails during a FAIL_FAST_PUT with the result that only the first part of *data ends* up being written on *sector*. The remainder of *sector* may contain older data. Any later call of FAIL_FAST_GET for that sector should discover that the sector checksum fails to verify and will thus return *status* = BAD.

Many (but not all) disks are designed to mask this class of failure by maintaining a reserve of power that is sufficient to complete any current sector write, in which case loss of power would be a tolerated failure.

- untolerated error: The operating system crashes during a FAIL_FAST_PUT and scribbles over the disk buffer in volatile storage, so FAIL_FAST_PUT writes corrupted data on one sector of the disk.
- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (Probability should be negligible.)

8.5.4.5 Careful Disk Storage

The fail-fast disk layer detects but does not mask errors. It leaves masking to the careful disk layer, which is also usually implemented in the firmware of the disk controller. The careful layer checks the value of *status* following each disk SEEK, GET and PUT operation, retrying the operation several times if necessary, a procedure that usually recovers from seek errors and soft errors caused by dust particles or a temporarily elevated noise level. Some disk controllers seek to a different track and back in an effort to dislodge the dust. The careful storage layer implements these storage procedures and failure tolerance model:

status ← CAREFUL_SEEK (*track*)
status ← CAREFUL_PUT (*data*, *sector_number*)
status ← CAREFUL_GET (*data*, *sector_number*)

- error-free operation: CAREFUL_SEEK moves the seek arm to *track*. CAREFUL_GET returns whatever was most recently written by CAREFUL_PUT at *sector_number* on *track*. All three return *status* = OK.
- tolerated error: *Soft read, write, or seek error.* CAREFUL_SEEK, CAREFUL_GET and CAREFUL_PUT mask these errors by repeatedly retrying the operation until the fail-fast layer stops detecting an error, returning with *status* = OK. The careful storage layer counts the retries, and if the retry count exceeds some limit, it gives up and declares the problem to be a hard error.
- detected error: *Hard error.* The careful storage layer distinguishes hard from soft errors by their persistence through several attempts to read, write, or seek, and reports them to the caller by setting *status* = BAD. (But also see the note on *revectoring* below.)
- detected error: The power fails during a CAREFUL_PUT with the result that only the first part of *data* ends up being written on *sector*. The remainder of *sector* may contain older data. Any later call of CAREFUL_GET for that sector should discover that the sector checksum fails to verify and will thus return *status* = BAD. (Assuming that the fail-fast layer does not tolerate power failures.)
- untolerated error: *Crash corrupts data.* The system crashes during CAREFUL_PUT and corrupts the disk buffer in volatile memory, so CAREFUL_PUT correctly writes to the

disk sector the corrupted data in that buffer. The sector checksum of the fail-fast layer cannot detect this case.

- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (Probability should be negligible)

Figure 8.12 exhibits algorithms for CAREFUL_GET and CAREFUL_PUT. The procedure CAREFUL_GET, by repeatedly reading any data with *status* = BAD, masks soft read errors. Similarly, CAREFUL_PUT retries repeatedly if the verification done by FAIL_FAST_PUT fails, thereby masking soft write errors, whatever their source.

The careful layer of most disk controller designs includes one more feature: if CAREFUL_PUT detects a hard error while writing a sector, it may instead write the data on a spare sector elsewhere on the same disk and add an entry to an internal disk mapping table so that future GETs and PUTs that specify that sector instead use the spare. This mechanism is called *revectoring*, and most disk designs allocate a batch of spare sectors for this purpose. The spares are not usually counted in the advertised disk capacity, but the manufacturer's advertising department does not usually ignore the resulting increase in the expected operational lifetime of the disk. For clarity of the discussion we omit that feature.

As indicated in the failure tolerance analysis, there are still two modes of failure that remain unmasked: a crash during CAREFUL_PUT may undetectably corrupt one disk sector, and a hard error arising from a bad spot on the disk or a decay event may detectably corrupt any number of disk sectors.

8.5.4.6 Durable Storage: RAID 1

For durability, the additional requirement is to mask decay events, which the careful storage layer only detects. The primary technique is that the PUT procedure should write several replicas of the data, taking care to place the replicas on different physical devices with the hope that the probability of disk decay in one replica is independent of the prob-

```
1  procedure CAREFUL_GET (data, sector_number)
2      for i from 1 to NTRIES do
3          if FAIL_FAST_GET (data, sector_number) = OK then
4              return OK
5      return BAD

6  procedure CAREFUL_PUT (data, sector_number)
7      for i from 1 to NTRIES do
8          if FAIL_FAST_PUT (data, sector_number) = OK then
9              return OK
10     return BAD
```

FIGURE 8.12

Procedures that implement careful disk storage.

ability of disk decay in the next one, and the number of replicas is large enough that when a disk fails there is enough time to replace it before all the other replicas fail. Disk system designers call these replicas *mirrors*. A carefully designed replica strategy can create storage that guards against premature disk failure and that is durable enough to substantially exceed the expected operational lifetime of any single physical disk. Errors on reading are detected by the fail-fast layer, so it is not usually necessary to read more than one copy unless that copy turns out to be bad. Since disk operations may involve more than one replica, the track and sector numbers are sometimes encoded into a virtual sector number and the durable storage layer automatically performs any needed seeks.
The durable storage layer implements these storage access procedures and failure tolerance model:

status ← DURABLE_PUT (*data*, *virtual_sector_number*)
status ← DURABLE_GET (*data*, *virtual_sector_number*)

- error-free operation: DURABLE_GET returns whatever was most recently written by DURABLE_PUT at *virtual_sector_number* with *status* = OK.
- tolerated error: Hard errors reported by the careful storage layer are masked by reading from one of the other replicas. The result is that the operation completes with *status* = OK.
- untolerated error: A decay event occurs on the same sector of all the replicas, and the operation completes with *status* = BAD.
- untolerated error: The operating system crashes during a DURABLE_PUT and scribbles over the disk buffer in volatile storage, so DURABLE_PUT writes corrupted data on all mirror copies of that sector.
- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (Probability should be negligible)

In this accounting there is no mention of soft errors or of positioning errors because they were all masked by a lower layer.

One configuration of RAID (see Section 2.1.1.4), known as "RAID 1", implements exactly this form of durable storage. RAID 1 consists of a tightly-managed array of identical replica disks in which DURABLE_PUT (*data*, *sector_number*) writes *data* at the same *sector_number* of each disk and DURABLE_GET reads from whichever replica copy has the smallest expected latency, which includes queuing time, seek time, and rotation time. With RAID, the decay set is usually taken to be an entire hard disk. If one of the disks fails, the next DURABLE_GET that tries to read from that disk will detect the failure, mask it by reading from another replica, and put out a call for repair. Repair consists of first replacing the disk that failed and then copying all of the disk sectors from one of the other replica disks.

8.5.4.7 Improving on RAID 1

Even with RAID 1, an untolerated error can occur if a rarely-used sector decays, and before that decay is noticed all other copies of that same sector also decay. When there is

finally a call for that sector, all fail to read and the data is lost. A closely related scenario is that a sector decays and is eventually noticed, but the other copies of that same sector decay before repair of the first one is completed. One way to reduce the chances of these outcomes is to implement a clerk that periodically reads all replicas of every sector, to check for decay. If CAREFUL_GET reports that a replica of a sector is unreadable at one of these periodic checks, the clerk immediately rewrites that replica from a good one. If the rewrite fails, the clerk calls for immediate revectoring of that sector or, if the number of revectorings is rapidly growing, replacement of the decay set to which the sector belongs. The period between these checks should be short enough that the probability that *all* replicas have decayed since the previous check is negligible. By analyzing the statistics of experience for similar disk systems, the designer chooses such a period, T_d. This approach leads to the following failure tolerance model:

status ← MORE_DURABLE_PUT (*data*, *virtual_sector_number*)
status ← MORE_DURABLE_GET (*data*, *virtual_sector_number*)

- error-free operation: MORE_DURABLE_GET returns whatever was most recently written by MORE_DURABLE_PUT at *virtual_sector_number* with *status* = OK
- tolerated error: Hard errors reported by the careful storage layer are masked by reading from one of the other replicas. The result is that the operation completes with *status* = OK.
- tolerated error: data of a single decay set decays, is discovered by the clerk, and is repaired, all within T_d seconds of the decay event.
- untolerated error: The operating system crashes during a DURABLE_PUT and scribbles over the disk buffer in volatile storage, so DURABLE_PUT writes corrupted data on all mirror copies of that sector.
- untolerated error: all decay sets fail within T_d seconds. (With a conservative choice of T_d, the probability of this event should be negligible.)
- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (With a good quality checksum, the probability of this event should be negligible.)

A somewhat less effective alternative to running a clerk that periodically verifies integrity of the data is to notice that the bathtub curve of Figure 8.1 applies to magnetic disks, and simply adopt a policy of systematically replacing the individual disks of the RAID array well before they reach the point where their conditional failure rate is predicted to start climbing. This alternative is not as effective for two reasons: First, it does not catch and repair random decay events, which instead accumulate. Second, it provides no warning if the actual operational lifetime is shorter than predicted (for example, if one happens to have acquired a bad batch of disks).

8.5.4.8 Detecting Errors Caused by System Crashes

With the addition of a clerk to watch for decay, there is now just one remaining untolerated error that has a significant probability: the hard error created by an operating system crash during CAREFUL_PUT. Since that scenario corrupts the data before the disk subsystem sees it, the disk subsystem has no way of either detecting or masking this error. Help is needed from outside the disk subsystem—either the operating system or the application. The usual approach is that either the system or, even better, the application program, calculates and includes an end-to-end checksum with the data before initiating the disk write. Any program that later reads the data verifies that the stored checksum matches the recalculated checksum of the data. The end-to-end checksum thus monitors the integrity of the data as it passes through the operating system buffers and also while it resides in the disk subsystem.

Sidebar 8.3: Are disk system checksums a wasted effort? From the adjacent paragraph, an *end-to-end argument* suggests that an end-to-end checksum is always needed to protect data on its way to and from the disk subsystem, and that the fail-fast checksum performed inside the disk system thus may not be essential.

However, the disk system checksum cleanly subcontracts one rather specialized job: correcting burst errors of the storage medium. In addition, the disk system checksum provides a handle for disk-layer erasure code implementations such as RAID, as was described in Section 8.4.1. Thus the disk system checksum, though superficially redundant, actually turns out to be quite useful.

The end-to-end checksum allows only detecting this class of error. Masking is another matter—it involves a technique called *recovery*, which is one of the topics of the next chapter.

Table 8.1 summarizes where failure tolerance is implemented in the several disk layers. The hope is that the remaining untolerated failures are so rare that they can be neglected. If they are not, the number of replicas could be increased until the probability of untolerated failures is negligible.

8.5.4.9 Still More Threats to Durability

The various procedures described above create storage that is durable in the face of individual disk decay but not in the face of other threats to data integrity. For example, if the power fails in the middle of a MORE_DURABLE_PUT, some replicas may contain old versions of the data, some may contain new versions, and some may contain corrupted data, so it is not at all obvious how MORE_DURABLE_GET should go about meeting its specification. The solution is to make MORE_DURABLE_PUT atomic, which is one of the topics of Chapter 9[on-line].

RAID systems usually specify that a successful return from a PUT confirms that writing of all of the mirror replicas was successful. That specification in turn usually requires that the multiple disks be physically co-located, which in turn creates a threat that a single

physical disaster—fire, earthquake, flood, civil disturbance, etc.—might damage or destroy all of the replicas.

Since magnetic disks are quite reliable in the short term, a different strategy is to write only one replica at the time that MORE_DURABLE_PUT is invoked and write the remaining replicas at a later time. Assuming there are no inopportune failures in the short run, the results gradually become more durable as more replicas are written. Replica writes that are separated in time are less likely to have replicated failures because they can be separated in physical location, use different disk driver software, or be written to completely different media such as magnetic tape. On the other hand, separating replica writes in time increases the risk of inconsistency among the replicas. Implementing storage that has durability that is substantially beyond that of RAID 1 and MORE_DURABLE_PUT/GET generally involves use of geographically separated replicas and systematic mechanisms to keep those replicas coordinated, a challenge that Chapter 10[on-line] discusses in depth.

Perhaps the most serious threat to durability is that although different storage systems have employed each of the failure detection and masking techniques discussed in this section, it is all too common to discover that a typical off-the-shelf personal computer file

	raw layer	fail-fast layer	careful layer	durable layer	more durable layer
soft read, write, or seek error	failure	detected	masked		
hard read, write error	failure	detected	detected	masked	
power failure interrupts a write	failure	detected	detected	masked	
single data decay	failure	detected	detected	masked	
multiple data decay spaced in time	failure	detected	detected	detected	masked
multiple data decay within T_d	failure	detected	detected	detected	failure*
undetectable decay	failure	failure	failure	failure	failure*
system crash corrupts write buffer	failure	failure	failure	failure	detected

Table 8.1: Summary of disk failure tolerance models. Each entry shows the effect of this error at the interface between the named layer and the next higher layer. With careful design, the probability of the two failures marked with an asterisk should be negligible. Masking of corruption caused by system crashes is discussed in Chapter 9[on-line]

system has been designed using an overly simple disk failure model and thus misses some—or even many—straightforward failure masking opportunities.

8.6 Wrapping up Reliability

8.6.1 Design Strategies and Design Principles

Standing back from the maze of detail about redundancy, we can identify and abstract three particularly effective design strategies:

- *N-modular redundancy* is a simple but powerful tool for masking failures and increasing availability, and it can be used at any convenient level of granularity.
- *Fail-fast modules* provide a *sweeping simplification* of the problem of containing errors. When containment can be described simply, reasoning about fault tolerance becomes easier.
- *Pair-and-compare* allows fail-fast modules to be constructed from commercial, off-the-shelf components.

Standing back still further, it is apparent that several general design principles are directly applicable to fault tolerance. In the formulation of the fault-tolerance design process in Section 8.1.2, we invoked *be explicit*, *design for iteration*. *keep digging*, and the *safety margin principle*, and in exploring different fault tolerance techniques we have seen several examples of *adopt sweeping simplifications*. One additional design principle that applies to fault tolerance (and also, as we will see in Chapter 11[on-line], to security) comes from experience, as documented in the case studies of Section 8.8:

Avoid rarely used components

Deterioration and corruption accumulate unnoticed—until the next use.

Whereas redundancy can provide masking of errors, redundant components that are used only when failures occur are much more likely to cause trouble than redundant components that are regularly exercised in normal operation. The reason is that failures in regularly exercised components are likely to be immediately noticed and fixed. Failures in unused components may not be noticed until a failure somewhere else happens. But then there are two failures, which may violate the design assumptions of the masking plan. This observation is especially true for software, where rarely-used recovery procedures often accumulate unnoticed bugs and incompatibilities as other parts of the system evolve. The alternative of periodic testing of rarely-used components to lower their failure latency is a band-aid that rarely works well.

In applying these design principles, it is important to consider the threats, the consequences, the environment, and the application. Some faults are more likely than others,

some failures are more disruptive than others, and different techniques may be appropriate in different environments. A computer-controlled radiation therapy machine, a deep-space probe, a telephone switch, and an airline reservation system all need fault tolerance, but in quite different forms. The radiation therapy machine should emphasize fault detection and fail-fast design, to avoid injuring patients. Masking faults may actually be a mistake. It is likely to be safer to stop, find their cause, and fix them before continuing operation. The deep-space probe, once the mission begins, needs to concentrate on failure masking to ensure mission success. The telephone switch needs many nines of availability because customers expect to always receive a dial tone, but if it occasionally disconnects one ongoing call, that customer will simply redial without thinking much about it. Users of the airline reservation system might tolerate short gaps in availability, but the durability of its storage system is vital. At the other extreme, most people find that a digital watch has an MTTF that is long compared with the time until the watch is misplaced, becomes obsolete, goes out of style, or is discarded. Consequently, no provision for either error masking or repair is really needed. Some applications have built-in redundancy that a designer can exploit. In a video stream, it is usually possible to mask the loss of a single video frame by just repeating the previous frame.

8.6.2 How about the End-to-End Argument?

There is a potential tension between error masking and an *end-to-end argument*. An end-to-end argument suggests that a subsystem need not do anything about errors and should not do anything that might compromise other goals such as low latency, high throughput, or low cost. The subsystem should instead let the higher layer system of which it is a component take care of the problem because only the higher layer knows whether or not the error matters and what is the best course of action to take.

There are two counter arguments to that line of reasoning:

- Ignoring an error allows it to propagate, thus contradicting the modularity goal of error containment. This observation points out an important distinction between error detection and error masking. Error detection and containment must be performed where the error happens, so that the error does not propagate wildly. Error masking, in contrast, presents a design choice: masking can be done locally or the error can be handled by reporting it at the interface (that is, by making the module design fail-fast) and allowing the next higher layer to decide what masking action—if any—to take.
- The lower layer may know the nature of the error well enough that it can mask it far more efficiently than the upper layer. The specialized burst error correction codes used on DVDs come to mind. They are designed specifically to mask errors caused by scratches and dust particles, rather than random bit-flips. So we have a trade-off between the cost of masking the fault locally and the cost of letting the error propagate and handling it in a higher layer.

These two points interact: When an error propagates it can contaminate otherwise correct data, which can increase the cost of masking and perhaps even render masking impossible. The result is that when the cost is small, error masking is usually done locally. (That is assuming that masking is done at all. Many personal computer designs omit memory error masking. Section 8.8.1 discusses some of the reasons for this design decision.)

A closely related observation is that when a lower layer masks a fault it is important that it also report the event to a higher layer, so that the higher layer can keep track of how much masking is going on and thus how much failure tolerance there remains. Reporting to a higher layer is a key aspect of *the safety margin principle*.

8.6.3 A Caution on the Use of Reliability Calculations

Reliability calculations seem to be exceptionally vulnerable to the garbage-in, garbage-out syndrome. It is all too common that calculations of mean time to failure are undermined because the probabilistic models are not supported by good statistics on the failure rate of the components, by measures of the actual load on the system or its components, or by accurate assessment of independence between components.

For computer systems, back-of-the-envelope calculations are often more than sufficient because they are usually at least as accurate as the available input data, which tends to be rendered obsolete by rapid technology change. Numbers predicted by formula can generate a false sense of confidence. This argument is much weaker for technologies that tend to be stable (for example, production lines that manufacture glass bottles). So reliability analysis is not a waste of time, but one must be cautious in applying its methods to computer systems.

8.6.4 Where to Learn More about Reliable Systems

Our treatment of fault tolerance has explored only the first layer of fundamental concepts. There is much more to the subject. For example, we have not considered another class of fault that combines the considerations of fault tolerance with those of security: faults caused by inconsistent, perhaps even malevolent, behavior. These faults have the characteristic they generate inconsistent error values, possibly error values that are specifically designed by an attacker to confuse or confound fault tolerance measures. These faults are called *Byzantine faults*, recalling the reputation of ancient Byzantium for malicious politics. Here is a typical Byzantine fault: suppose that an evil spirit occupies one of the three replicas of a TMR system, waits for one of the other replicas to fail, and then adjusts its own output to be identical to the incorrect output of the failed replica. A voter accepts this incorrect result and the error propagates beyond the intended containment boundary. In another kind of Byzantine fault, a faulty replica in an NMR system sends different result values to each of the voters that are monitoring its output. Malevolence is not required—any fault that is not anticipated by a fault detection mechanism can produce Byzantine behavior. There has recently been considerable attention to techniques

that can tolerate Byzantine faults. Because the tolerance algorithms can be quite complex, we defer the topic to advanced study.

We also have not explored the full range of reliability techniques that one might encounter in practice. For an example that has not yet been mentioned, Sidebar 8.4 describes the *heartbeat*, a popular technique for detecting failures of active processes.

This chapter has oversimplified some ideas. For example, the definition of availability proposed in Section 8.2 of this chapter is too simple to adequately characterize many large systems. If a bank has hundreds of automatic teller machines, there will probably always be a few teller machines that are not working at any instant. For this case, an availability measure based on the percentage of transactions completed within a specified response time would probably be more appropriate.

A rapidly moving but in-depth discussion of fault tolerance can be found in Chapter 3 of the book *Transaction Processing: Concepts and Techniques*, by Jim Gray and Andreas Reuter. A broader treatment, with case studies, can be found in the book *Reliable Computer Systems: Design and Evaluation*, by Daniel P. Siewiorek and Robert S. Swarz. Byzantine faults are an area of ongoing research and development, and the best source is current professional literature.

This chapter has concentrated on general techniques for achieving reliability that are applicable to hardware, software, and complete systems. Looking ahead, Chapters 9[on-line] and 10[on-line] revisit reliability in the context of specific software techniques that permit reconstruction of stored state following a failure when there are several concurrent activities. Chapter 11[on-line], on securing systems against malicious attack, introduces a redundancy scheme known as *defense in depth* that can help both to contain and to mask errors in the design or implementation of individual security mechanisms.

Sidebar 8.4: Detecting failures with heartbeats. An activity such as a Web server is usually intended to keep running indefinitely. If it fails (perhaps by crashing) its clients may notice that it has stopped responding, but clients are not typically in a position to restart the server. Something more systematic is needed to detect the failure and initiate recovery. One helpful technique is to program the thread that should be performing the activity to send a periodic signal to another thread (or a message to a monitoring service) that says, in effect, "I'm still OK". The periodic signal is known as a *heartbeat* and the observing thread or service is known as a *watchdog*.

The watchdog service sets a timer, and on receipt of a heartbeat message it restarts the timer. If the timer ever expires, the watchdog assumes that the monitored service has gotten into trouble and it initiates recovery. One limitation of this technique is that if the monitored service fails in such a way that the only thing it does is send heartbeat signals, the failure will go undetected.

As with all fixed timers, choosing a good heartbeat interval is an engineering challenge. Setting the interval too short wastes resources sending and responding to heartbeat signals. Setting the interval too long delays detection of failures. Since detection is a prerequisite to repair, a long heartbeat interval increases MTTR and thus reduces availability.

8.7 Application: A Fault Tolerance Model for CMOS RAM

This section develops a fault tolerance model for words of CMOS random access memory, first without and then with a simple error-correction code, comparing the probability of error in the two cases.

CMOS RAM is both low in cost and extraordinarily reliable, so much so that error masking is often not implemented in mass production systems such as television sets and personal computers. But some systems, for example life-support, air traffic control, or banking systems, cannot afford to take unnecessary risks. Such systems usually employ the same low-cost memory technology but add incremental redundancy.

A common failure of CMOS RAM is that noise intermittently causes a single bit to read or write incorrectly. If intermittent noise affected only reads, then it might be sufficient to detect the error and retry the read. But the possibility of errors on writes suggests using a forward error-correction code.

We start with a fault tolerance model that applies when reading a word from memory without error correction. The model assumes that errors in different bits are independent and it assigns p as the (presumably small) probability that any individual bit is in error. The notation $O(p^n)$ means terms involving p^n and higher, presumably negligible, powers. Here are the possibilities and their associated probabilities:

Fault tolerance model for raw CMOS random access memory

		probability
error-free case:	all 32 bits are correct	$(1-p)^{32} = 1 - O(p)$
errors:		
untolerated:	one bit is in error:	$32p(1-p)^{31} = O(p)$
untolerated:	two bits are in error:	$(31 \cdot 32/2)p^2(1-p)^{30} = O(p^2)$
untolerated:	three or more bits are in error:	$(30 \cdot 31 \cdot 32/3 \cdot 2)p^3(1-p)^{29} + \ldots + p^{32} = O(p^3)$

The coefficients 32, $(31 \cdot 32)/2$, etc., arise by counting the number of ways that one, two, etc., bits could be in error.

Suppose now that the 32-bit block of memory is encoded using a code of Hamming distance 3, as described in Section 8.4.1. Such a code allows any single-bit error to be

corrected and any double-bit error to be detected. After applying the decoding algorithm, the fault tolerance model changes to:

Fault tolerance model for CMOS memory with error correction

		probability
error-free case:	all 32 bits are correct	$(1-p)^{32} = 1 - O(p)$
errors:		
tolerated:	one bit corrected:	$32p(1-p)^{31} = O(p)$
detected:	two bits are in error:	$(31 \cdot 32/2)p^2(1-p)^{30} = O(p^2)$
untolerated:	three or more bits are in error:	$(30 \cdot 31 \cdot 32/3 \cdot 2)p^3(1-p)^{29} + \ldots + p^{32} = O(p^3)$

The interesting change is in the probability that the decoded value is correct. That probability is the sum of the probabilities that there were no errors and that there was one, tolerated error:

$$\begin{aligned} Prob(\text{decoded value is correct}) &= (1-p)^{32} + 32p(1-p)^{31} \\ &= (1 - 32p + (31 \cdot 32/2)p^2 + \ldots) + (32p + 31 \cdot \\ &= (1 - O(p^2)) \end{aligned}$$

The decoding algorithm has thus eliminated the errors that have probability of order p. It has not eliminated the two-bit errors, which have probability of order p^2, but for two-bit errors the algorithm is fail-fast, so a higher-level procedure has an opportunity to recover, perhaps by requesting retransmission of the data. The code is not helpful if there are errors in three or more bits, which situation has probability of order p^3, but presumably the designer has determined that probabilities of that order are negligible. If they are not, the designer should adopt a more powerful error-correction code.

With this model in mind, one can review the two design questions suggested on page 8–19. The first question is whether the estimate of bit error probability is realistic and if it is realistic to suppose that multiple bit errors are statistically independent of one another. (Error independence appeared in the analysis in the claim that the probability of an n-bit error has the order of the nth power of the probability of a one-bit error.) Those questions concern the real world and the accuracy of the designer's model of it. For example, this failure model doesn't consider power failures, which might take all the bits out at once, or a driver logic error that might take out all of the even-numbered bits.

It also ignores the possibility of faults that lead to errors in the logic of the error-correction circuitry itself.

The second question is whether the coding algorithm actually corrects all one-bit errors and detects all two-bit errors. That question is explored by examining the mathematical structure of the error-correction code and is quite independent of anybody's estimate or measurement of real-world failure types and rates. There are many off-the-shelf coding algorithms that have been thoroughly analyzed and for which the answer is yes.

8.8 War Stories: Fault Tolerant Systems that Failed

8.8.1 Adventures with Error Correction*

The designers of the computer systems at the Xerox Palo Alto Research Center in the early 1970s encountered a series of experiences with error-detecting and error-correcting memory systems. From these experiences follow several lessons, some of which are far from intuitive, and all of which still apply several decades later.

MAXC. One of the first projects undertaken in the newly-created Computer Systems Laboratory was to build a time-sharing computer system, named MAXC. A brand new 1024-bit memory chip, the Intel 1103, had just appeared on the market, and it promised to be a compact and economical choice for the main memory of the computer. But since the new chip had unknown reliability characteristics, the MAXC designers implemented the memory system using a few extra bits for each 36-bit word, in the form of a single-error-correction, double-error-detection code.

Experience with the memory in MAXC was favorable. The memory was solidly reliable—so solid that no errors in the memory system were ever reported.

The Alto. When the time came to design the Alto personal workstation, the same Intel memory chips still appeared to be the preferred component. Because these chips had performed so reliably in MAXC, the designers of the Alto memory decided to relax a little, omitting error correction. But, they were still conservative enough to provide error detection, in the form of one parity bit for each 16-bit word of memory.

This design choice seemed to be a winner because the Alto memory systems also performed flawlessly, at least for the first several months. Then, mysteriously, the operating system began to report frequent memory-parity failures.

Some background: the Alto started life with an operating system and applications that used a simple typewriter-style interface. The display was managed with a character-by-character teletype emulator. But the purpose of the Alto was to experiment with better

* These experiences were reported by Butler Lampson, one of the designers of the MAXC computer and the Alto personal workstations at Xerox Palo Alto Research Center.

things. One of the first steps in that direction was to implement the first what-you-see-is-what-you-get editor, named Bravo. Bravo took full advantage of the bit-map display, filling it not only with text, but also with lines, buttons, and icons. About half the memory system was devoted to display memory. Curiously, the installation of Bravo coincided with the onset of memory parity errors.

It turned out that the Intel 1103 chips were pattern-sensitive—certain read/write sequences of particular bit patterns could cause trouble, probably because those pattern sequences created noise levels somewhere on the chip that systematically exceeded some critical threshold. The Bravo editor's display management was the first application that generated enough different patterns to have an appreciable probability of causing a parity error. It did so, frequently.

Lesson 8.8.1a: There is no such thing as a small change in a large system. A new piece of software can bring down a piece of hardware that is thought to be working perfectly. You are never quite sure just how close to the edge of the cliff you are standing.

Lesson 8.8.1b: Experience is a primary source of information about failures. It is nearly impossible, without specific prior experience, to predict what kinds of failures you will encounter in the field.

Back to MAXC. This circumstance led to a more careful review of the situation on MAXC. MAXC, being a heavily used server, would be expected to encounter at least some of this pattern sensitivity. It was discovered that although the error-correction circuits had been designed to report both corrected errors and uncorrectable errors, the software logged only uncorrectable errors and corrected errors were being ignored. When logging of corrected errors was implemented, it turned out that the MAXC's Intel 1103's were actually failing occasionally, and the error-correction circuitry was busily setting things right.

Lesson 8.8.1c: Whenever systems implement automatic error masking, it is important to follow the safety margin principle, by tracking how often errors are successfully masked. Without this information, one has no way of knowing whether the system is operating with a large or small safety margin for additional errors. Otherwise, despite the attempt to put some guaranteed space between yourself and the edge of the cliff, you may be standing on the edge again.

The Alto 2. In 1975, it was time to design a follow-on workstation, the Alto 2. A new generation of memory chips, this time with 4096 bits, was now available. Since it took up much less space and promised to be cheaper, this new chip looked attractive, but again there was no experience with its reliability. The Alto 2 designers, having been made wary by the pattern sensitivity of the previous generation chips, again resorted to a single-error-correction, double-error-detection code in the memory system.

Once again, the memory system performed flawlessly. The cards passed their acceptance tests and went into service. In service, not only were no double-bit errors detected, only rarely were single-bit errors being corrected. The initial conclusion was that the chip vendors had worked the bugs out and these chips were really good.

About two years later, someone discovered an implementation mistake. In one quadrant of each memory card, neither error correction nor error detection was actually working. All computations done using memory in the misimplemented quadrant were completely unprotected from memory errors.

Lesson 8.8.1d: Never assume that the hardware actually does what it says in the specifications.
Lesson 8.8.1e: It is harder than it looks to test the fault tolerance features of a fault tolerant system.

One might conclude that the intrinsic memory chip reliability had improved substantially—so much that it was no longer necessary to take heroic measures to achieve system reliability. Certainly the chips were better, but they weren't perfect. The other effect here is that errors often don't lead to failures. In particular, a wrong bit retrieved from memory does not necessarily lead to an observed failure. In many cases a wrong bit doesn't matter; in other cases it does but no one notices; in still other cases, the failure is blamed on something else.

Lesson 8.8.1f: Just because it seems to be working doesn't mean that it actually is.

The bottom line. One of the designers of MAXC and the Altos, Butler Lampson, suggests that the possibility that a failure is blamed on something else can be viewed as an opportunity, and it may be one of the reasons that PC manufacturers often do not provide memory parity checking hardware. First, the chips are good enough that errors are rare. Second, if you provide parity checks, consider who will be blamed when the parity circuits report trouble: the hardware vendor. Omitting the parity checks probably leads to occasional random behavior, but occasional random behavior is indistinguishable from software error and is usually blamed on the software.

Lesson 8.8.1g (in Lampson's words): "Beauty is in the eye of the beholder. The various parties involved in the decisions about how much failure detection and recovery to implement do not always have the same interests."

8.8.2 Risks of Rarely-Used Procedures: The National Archives

The National Archives and Record Administration of the United States government has the responsibility, among other things, of advising the rest of the government how to preserve electronic records such as e-mail messages for posterity. Quite separate from that responsibility, the organization also operates an e-mail system at its Washington, D.C. headquarters for a staff of about 125 people and about 10,000 messages a month pass through this system. To ensure that no messages are lost, it arranged with an outside contractor to perform daily incremental backups and to make periodic complete backups of its e-mail files. On the chance that something may go wrong, the system has audit logs that track actions regarding incoming and outgoing mail as well as maintenance on files.

Over the weekend of June 18–21, 1999, the e-mail records for the previous four months (an estimated 43,000 messages) disappeared. No one has any idea what went wrong—the files may have been deleted by a disgruntled employee or a runaway house-

cleaning program, or the loss may have been caused by a wayward system bug. In any case, on Monday morning when people came to work, they found that the files were missing.

On investigation, the system managers reported that the audit logs had been turned off because they were reducing system performance, so there were no clues available to diagnose what went wrong. Moreover, since the contractor's employees had never gotten around to actually performing the backup part of the contract, there were no backup copies. It had not occurred to the staff of the Archives to verify the existence of the backup copies, much less to test them to see if they could actually be restored. They assumed that since the contract required it, the work was being done.

The contractor's project manager and the employee responsible for making backups were immediately replaced. The Assistant Archivist reports that backup systems have now been beefed up to guard against another mishap, but he added that the safest way to save important messages is to print them out.*

Lesson 8.8.2: Avoid rarely used components. *Rarely used failure-tolerance mechanisms, such as restoration from backup copies, must be tested periodically. If they are not, there is not much chance that they will work when an emergency arises. Fire drills (in this case performing a restoration of all files from a backup copy) seem disruptive and expensive, but they are not nearly as disruptive and expensive as the discovery, too late, that the backup system isn't really operating. Even better, design the system so that all the components are exposed to day-to-day use, so that failures can be noticed before they cause real trouble.*

8.8.3 Non-independent Replicas and Backhoe Fade

In Eagan, Minnesota, Northwest airlines operated a computer system, named World-Flight, that managed the Northwest flight dispatching database, provided weight-and-balance calculations for pilots, and managed e-mail communications between the dispatch center and all Northwest airplanes. It also provided data to other systems that managed passenger check-in and the airline's Web site. Since many of these functions involved communications, Northwest contracted with U.S. West, the local telephone company at that time, to provide these communications in the form of fiber-optic links to airports that Northwest serves, to government agencies such as the Weather Bureau and the Federal Aviation Administration, and to the Internet. Because these links were vital, Northwest paid U.S. West extra to provide each primary link with a backup secondary link. If a primary link to a site failed, the network control computers automatically switched over to the secondary link to that site.

At 2:05 p.m. on March 23, 2000, all communications to and from WorldFlight dropped out simultaneously. A contractor who was boring a tunnel (for fiber optic lines for a different telephone company) at the nearby intersection of Lone Oak and Pilot Knob roads accidentally bored through a conduit containing six cables carrying the U.S.

* George Lardner Jr. "Archives Loses 43,000 E-Mails; officials can't explain summer erasure; backup system failed." *The Washington Post,* Thursday, January 6, 2000, page A17.

West fiber-optic and copper lines. In a tongue-in-cheek analogy to the fading in and out of long-distance radio signals, this kind of communications disruption is known in the trade as "backhoe fade." WorldFlight immediately switched from the primary links to the secondary links, only to find that they were not working, either. It seems that the primary and secondary links were routed through the same conduit, and both were severed.

Pilots resorted to manual procedures for calculating weight and balance, and radio links were used by flight dispatchers in place of the electronic message system, but about 125 of Northwest's 1700 flights had to be cancelled because of the disruption, about the same number that are cancelled when a major snowstorm hits one of Northwest's hubs. Much of the ensuing media coverage concentrated on whether or not the contractor had followed "dig-safe" procedures that are intended to prevent such mistakes. But a news release from Northwest at 5:15 p.m. blamed the problem entirely on U.S. West. "For such contingencies, U.S. West provides to Northwest a complete redundancy plan. The U.S. West redundancy plan also failed."*

In a similar incident, the ARPAnet, a predecessor to the Internet, had seven separate trunk lines connecting routers in New England to routers elsewhere in the United States. All the trunk lines were purchased from a single long-distance carrier, AT&T. On December 12, 1986, all seven trunk lines went down simultaneously when a contractor accidentally severed a single fiber-optic cable running from White Plains, New York to Newark, New Jersey.†

A complication for communications customers who recognize this problem and request information about the physical location of their communication links is that, in the name of security, communications companies sometimes refuse to reveal it.

Lesson 8.8.3: The calculation of mean time to failure of a redundant system depends critically on the assumption that failures of the replicas are independent. If they aren't independent, then the replication may be a waste of effort and money, while producing a false complacency. This incident also illustrates why it can be difficult to test fault tolerance measures properly. What appears to be redundancy at one level of abstraction turns out not to be redundant at a lower level of abstraction.

8.8.4 Human Error May Be the Biggest Risk

Telehouse was an East London "telecommunications hotel", a seven story building housing communications equipment for about 100 customers, including most British Internet companies, many British and international telephone companies, and dozens of financial institutions. It was designed to be one of the most secure buildings in Europe, safe against "fire, flooding, bombs, and sabotage". Accordingly, Telehouse had extensive protection against power failure, including two independent connections to the national

* Tony Kennedy. "Cut cable causes cancellations, delays for Northwest Airlines." *Minneapolis Star Tribune*, March 22, 2000.

† Peter G. Neumann. *Computer Related Risks* (Addison-Wesley, New York, 1995), page 14.

electric power grid, a room full of batteries, and two diesel generators, along with systems to detect failures in supply and automatically cut over from one backup system to the next, as needed.

On May 8, 1997, all the computer systems went off line for lack of power. According to Robert Bannington, financial director of Telehouse, "It was due to human error." That is, someone pulled the wrong switch. The automatic power supply cutover procedures did not trigger because they were designed to deploy on failure of the outside power supply, and the sensors correctly observed that the outside power supply was intact.*

Lesson 8.8.4a: The first step in designing a fault tolerant system is to identify each potential fault and evaluate the risk that it will happen. People are part of the system, and mistakes made by authorized operators are typically a bigger threat to reliability than trees falling on power lines.

Anecdotes concerning failures of backup power supply systems seem to be common. Here is a typical report of an experience in a Newark, New Jersey, hospital operating room that was equipped with three backup generators: "On August 14, 2003, at 4:10pm EST, a widespread power grid failure caused our hospital to suffer a total OR power loss, regaining partial power in 4 hours and total restoration 12 hours later... When the backup generators initially came on-line, all ORs were running as usual. Within 20 minutes, one parallel-linked generator caught fire from an oil leak. After being subjected to twice its rated load, the second in-line generator quickly shut down... Hospital engineering, attempting load-reduction to the single surviving generator, switched many hospital circuit breakers off. Main power was interrupted to the OR."†

Lesson 8.8.4b: A backup generator is another example of a rarely used component that may not have been maintained properly. The last two sentences of that report reemphasize Lesson 8.8.4a.

For yet another example, the M.I.T. Information Services and Technology staff posted the following system services notice on April 2, 2004: "We suffered a power failure in W92 shortly before 11AM this morning. Most services should be restored now, but some are still being recovered. Please check back here for more information as it becomes available." A later posting reported: "Shortly after 10AM Friday morning the routine test of the W92 backup generator was started. Unknown to us was that the transition of the computer room load from commercial power to the backup generator resulted in a power surge within the computer room's Uninterruptable [sic] Power Supply (UPS). This destroyed an internal surge protector, which started to smolder. Shortly before 11AM the smoldering protector triggered the VESDA® smoke sensing system

---

* Robert Uhlig. "Engineer pulls plug on secure bunker." *Electronic Telegraph,* (9 May 1997).

† Ian E. Kirk, M.D. and Peter L. Fine, M.D. "Operating by Flashlight: Power Failure and Safety Lessons from the August, 2003 Blackout." *Abstracts of the Annual Meeting of the American Society of Anesthesiologists*, October 2005.

within the computer room. This sensor triggered the fire alarm, and as a safety precaution forced an emergency power down of the entire computer room."*

Lesson 8.8.4c: A failure masking system not only can fail, it can cause a bigger failure than the one it is intended to mask.

8.8.5 Introducing a Single Point of Failure

"[Rabbi Israel Meir HaCohen Kagan described] a real-life situation in his town of Radin, Poland. He lived at the time when the town first purchased an electrical generator and wired all the houses and courtyards with electric lighting. One evening something broke within the machine, and darkness descended upon all of the houses and streets, and even in the synagogue.

"So he pointed out that before they had electricity, every house had a kerosene light—and if in one particular house the kerosene ran out, or the wick burnt away, or the glass broke, that only that one house would be dark. But when everyone is dependent upon one machine, darkness spreads over the entire city if it breaks for any reason."†

Lesson 8.8.5: Centralization may provide economies of scale, but it can also reduce robustness—a single failure can interfere with many unrelated activities. This phenomenon is commonly known as introducing a single point of failure. By carefully adding redundancy to a centralized design one may be able to restore some of the lost robustness but it takes planning and adds to the cost.

8.8.6 Multiple Failures: The SOHO Mission Interruption

"Contact with the SOlar Heliospheric Observatory (SOHO) spacecraft was lost in the early morning hours of June 25, 1998, Eastern Daylight Time (EDT), during a planned period of calibrations, maneuvers, and spacecraft reconfigurations. Prior to this the SOHO operations team had concluded two years of extremely successful science operations.

"…The Board finds that the loss of the SOHO spacecraft was a direct result of operational errors, a failure to adequately monitor spacecraft status, and an erroneous decision which disabled part of the on-board autonomous failure detection. Further, following the occurrence of the emergency situation, the Board finds that insufficient time was taken by the operations team to fully assess the spacecraft status prior to initiating recovery operations. The Board discovered that a number of factors contributed to the circumstances that allowed the direct causes to occur."‡

---

* Private internal communication.

† Chofetz Chaim (the Rabbi Israel Meir HaCohen Kagan of Radin), paraphrased by Rabbi Yaakov Menken, in a discussion of lessons from the Torah in *Project Genesis Lifeline*. <http://www.torah.org/learning/lifeline/5758/reeh.html>. Suggested by David Karger.

In a tour-de-force of the *keep digging principle*, the report of the investigating board quoted above identified five distinct direct causes of the loss: two software errors, a design feature that unintentionally amplified the effect of one of the software errors, an incorrect diagnosis by the ground staff, and a violated design assumption. It then goes on to identify three indirect causes in the spacecraft design process: lack of change control, missing risk analysis for changes, and insufficient communication of changes, and then three indirect causes in operations procedures: failure to follow planned procedures, to evaluate secondary telemetry data, and to question telemetry discrepancies.

Lesson 8.8.6: Complex systems fail for complex reasons. In systems engineered for reliability, it usually takes several component failures to cause a system failure. Unfortunately, when some of the components are people, multiple failures are all too common.

Exercises

8.1 Failures are

A. Faults that are latent.
B. Errors that are contained within a module.
C. Errors that propagate out of a module.
D. Faults that turn into errors.

1999–3–01

8.2 Ben Bitdiddle has been asked to perform a deterministic computation to calculate the orbit of a near-Earth asteroid for the next 500 years, to find out whether or not the asteroid will hit the Earth. The calculation will take roughly two years to complete, and Ben wants be be sure that the result will be correct. He buys 30 identical computers and runs the same program with the same inputs on all of them. Once each hour the software pauses long enough to write all intermediate results to a hard disk on that computer. When the computers return their results at the end

‡ Massimo Trella and Michael Greenfield. *Final Report of the SOHO Mission Interruption Joint NASA/ESA Investigation Board* (August 31, 1998). National Aeronautics and Space Administration and European Space Agency.
<http://sohowww.nascom.nasa.gov/whatsnew/SOHO_final_report.html>

of the two years, a voter selects the majority answer. Which of the following failures can this scheme tolerate, assuming the voter works correctly?

A. The software carrying out the deterministic computation has a bug in it, causing the program to compute the wrong answer for certain inputs.
B. Over the course of the two years, cosmic rays corrupt data stored in memory at twelve of the computers, causing them to return incorrect results.
C. Over the course of the two years, on 24 different days the power fails in the computer room. When the power comes back on, each computer reboots and then continues its computation, starting with the state it finds on its hard disk.

2006–2–3

8.3 Ben Bitdiddle has seven smoke detectors installed in various places in his house. Since the fire department charges $100 for responding to a false alarm, Ben has connected the outputs of the smoke detectors to a simple majority voter, which in turn can activate an automatic dialer that calls the fire department. Ben returns home one day to find his house on fire, and the fire department has not been called. There is smoke at every smoke detector. What did Ben do wrong?

A. He should have used fail-fast smoke detectors.
B. He should have used a voter that ignores failed inputs from fail-fast sources.
C. He should have used a voter that ignores non-active inputs.
D. He should have done both A and B.
E. He should have done both Aand C.

1997–0–01

8.4 You will be flying home from a job interview in Silicon Valley. Your travel agent gives you the following choice of flights:

A. Flight A uses a plane whose mean time to failure (MTTF) is believed to be 6,000 hours. With this plane, the flight is scheduled to take 6 hours.
B. Flight B uses a plane whose MTTF is believed to be 5,000 hours. With this plane, the flight takes 5 hours.

The agent assures you that both planes' failures occur according to memoryless random processes (not a "bathtub" curve). Assuming that model, which flight should you choose to minimize the chance of your plane failing during the flight?

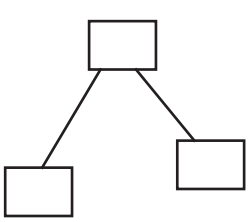
2005–2–5

8.5 (Note: solving this problem is best done with use of probability through the level of Markov chains.) You are designing a computer system to control the power grid for the Northeastern United States. If your system goes down, the lights go out and civil disorder—riots, looting, fires, etc.—will ensue. Thus, you have set a goal of having a system MTTF of at least 100 years (about 10^6 hours). For hardware you are constrained to use a building block computer that has a MTTF of 1000 hours

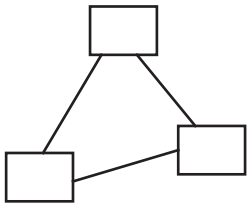
and a MTTR of 1 hour. Assuming that the building blocks are fail-fast, memoryless, and fail independently of one another, how can you arrange to meet your goal?

1995–3–1a

8.6 The town council wants to implement a municipal network to connect the local area networks in the library, the town hall, and the school. They want to minimize the chance that any building is completely disconnected from the others. They are considering two network topologies:



1. "Daisy Chain"



2. "Fully connected"

Each link in the network has a failure probability of p.

8.6a. What is the probability that the daisy chain network is connecting all the buildings?

8.6b. What is the probability that the fully connected network is connecting all the buildings?

8.6c. The town council has a limited budget, with which it can buy either a daisy chain network with two high reliability links (p = .000001), or a fully connected network with three low-reliability links (p = .0001). Which should they purchase?

1985–0–1

8.7 Figure 8.11 shows the failure points of three different 5MR supermodule designs, if repair does not happen in time. Draw the corresponding figure for the same three different TMR supermodule designs.

2001–3–05

8.8 An astronomer calculating the trajectory of Pluto has a program that requires the execution of 10^{13} machine operations. The fastest processor available in the lab runs only 10^{9} operations per second and, unfortunately, has a probability of failing on any one operation of 10^{-12}. (The failure process is memoryless.) The good news is that the processor is fail-fast, so when a failure occurs it stops dead in its tracks and starts ringing a bell. The bad news is that when it fails, it loses all state, so whatever it was doing is lost, and has to be started over from the beginning.

Seeing that in practical terms, the program needs to run for about 3 hours, and the machine has an MTTF of only 1/10 of that time, Louis Reasoner and Ben Bitdiddle have proposed two ways to organize the computation:

- Louis says run it from the beginning and hope for the best. If the machine fails, just try again; keep trying till the calculation successfully completes.
- Ben suggests dividing the calculation into ten equal-length segments; if the calculation gets to the end of a segment, it writes its state out to the disk. When a failure occurs, restart from the last state saved on the disk.

Saving state and restart both take zero time. What is the ratio of the expected time to complete the calculation under the two strategies?

Warning: A straightforward solution to this problem involves advanced probability techniques.

1976–0–3

8.9 Draw a figure, similar to that of Figure 8.6, that shows the recovery procedure for one sector of a 5-disk RAID 4 system when disk 2 fails and is replaced.

2005–0–1

8.10 Louis Reasoner has just read an advertisement for a RAID controller that provides a choice of two configurations. According to the advertisement, the first configuration is exactly the RAID 4 system described in Section 8.4.1. The advertisement goes on to say that the configuration called RAID 5 has just one difference: in an *N*-disk configuration, the parity block, rather than being written on disk *N*, is written on the disk number (1 + *sector_address* modulo *N*). Thus, for example, in a five-disk system, the parity block for sector 18 would be on disk 4 (because 1 + (18 modulo 5) = 4), while the parity block for sector 19 would be on

disk 5 (because 1 + (19 modulo 5) = 5). Louis is hoping you can help him understand why this idea might be a good one.

8.10a. RAID 5 has the advantage over RAID 4 that

A. It tolerates single-drive failures.
B. Read performance in the absence of errors is enhanced.
C. Write performance in the absence of errors is enhanced.
D. Locating data on the drives is easier.
E. Allocating space on the drives is easier.
F. It requires less disk space.
G. There's no real advantage, its just another advertising gimmick.

1997–3–01

8.10b. Is there any workload for which RAID 4 has better write performance than RAID 5?

2000–3–01

8.10c. Louis is also wondering about whether he might be better off using a RAID 1 system (see Section 8.5.4.6). How does the number of disks required compare between RAID 1 and RAID 5?

1998–3–01

8.10d. Which of RAID 1 and RAID 5 has better performance for a workload consisting of small reads and small writes?

2000–3–01

8.11 A system administrator notices that a file service disk is failing for two unrelated reasons. Once every 30 days, on average, vibration due to nearby construction breaks the disk's arm. Once every 60 days, on average, a power surge destroys the disk's electronics. The system administrator fixes the disk instantly each time it fails. The two failure modes are independent of each other, and independent of the age of the disk. What is the mean time to failure of the disk?

2002–3–01

Additional exercises relating to Chapter 8 can be found in problem sets *26* through *28*.

Glossary for Chapter 8

active fault—A fault that is currently causing an error. Compare with *latent fault*. [Ch. 8]

availability—A measure of the time that a system was actually usable, as a fraction of the time that it was intended to be usable. Compare with its complement, *down time*. [Ch. 8]

backward error correction—A technique for correcting errors in which the source of the data or control signal applies enough redundancy to allow errors to be detected and, if an error does occur, that source is asked to redo the calculation or repeat the transmission. Compare with *forward error correction*. [Ch. 8]

Byzantine fault—A fault that generates inconsistent errors (perhaps maliciously) that can confuse or disrupt fault tolerance or security mechanisms. [Ch. 8]

close-to-open consistency—A consistency model for file operations. When a thread opens a file and performs several write operations, all of the modifications weill be visible to concurrent threads only after the first thread closes the file. [Ch. 4]

coheerence—See *read/write coherence* or *cache coherence*.

continuous operation—An availability goal, that a system be capable of running indefinitely. The primary requirement of continuous operation is that it must be possible to perform repair and maintenance without stopping the system. [Ch. 8]

decay set—A set of storage blocks, words, tracks, or other physical groupings, in which all members of the set may spontaneously fail together, but independently of any other decay set. [Ch. 8]

detectable error—An error or class of errors for which a reliable detection plan can be devised. An error that is not detectable usually leads to a failure, unless some mechanism that is intended to mask some other error accidentally happens to mask the undetectable error. Compare with *maskable error* and *tolerated error*. [Ch. 8]

down time—A measure of the time that a system was not usable, as a fraction of the time that it was intended to be usable. Compare with its complement, *availability*. [Ch. 8]

durable storage—Storage with the property that it (ideally) is decay-free, so it never fails to return on a GET the data that was stored by a previously successful PUT. Since that ideal is impossibly strict, in practice, storage is considered durable when the probability of failure is sufficiently low that the application can tolerate it. Durability is thus an application-defined specification of how long the results of an action, once completed, must be preserved. Durable is distinct from *non-volatile*, which describes storage that maintains its memory while the power is off, but may still have an intolerable probability of decay. The term *persistent* is sometimes used as a synonym for durable, as explained in Sidebar 2.7, but to minimize confusion this text avoids that usage. [Ch. 8]

erasure—An error in a string of bits, bytes, or groups of bits in which an identified bit, byte,

or group of bits is missing or has indeterminate value. [Ch. 8]

ergodic—A property of some time-dependent probabilistic processes: that the (usually easier to measure) ensemble average of some parameter measured over a set of elements subject to the process is the same as the time average of that parameter of any single element of the ensemble. [Ch. 8]

error—Informally, a label for an incorrect data value or control signal caused by an active fault. If there is a complete formal specification for the internal design of a module, an error is a violation of some assertion or invariant of the specification. An error in a module is not identical to a failure of that module, but if an error is not masked, it may lead to a failure of the module. [Ch. 8]

error containment—Limiting how far the effects of an error propagate. A module is normally designed to contain errors in such a way that the effects of an error appear in a predictable way at the module's interface. [Ch. 8]

error correction—A scheme to set to the correct value a data value or control signal that is in error. Compare with *error detection*. [Ch. 8]

error detection—A scheme to discover that a data value or control signal is in error. Compare with *error correction*. [Ch. 8]

fail-fast—Describes a system or module design that contains detected errors by reporting at its interface that its output may be incorrect. Compare with *fail-stop*. [Ch. 8]

fail-safe—Describes a system design that detects incorrect data values or control signals and forces them to values that, even if not correct, are known to allow the system to continue operating safely. [Ch. 8]

fail-secure—Describes an application of fail-safe design to information protection: a failure is guaranteed not to allow unauthorized access to protected information. In early work on fault tolerance, this term was also occasionally used as a synonym for *fail-fast*. [Ch. 8]

fail-soft—Describes a design in which the system specification allows errors to be masked by degrading performance or disabling some functions in a predictable manner. [Ch. 8]

fail-stop—Describes a system or module design that contains detected errors by stopping the system or module as soon as possible. Compare with *fail-fast*, which does not require other modules to take additional action, such as setting a timer, to detect the failure. [Ch. 8]

fail-vote—Describes an *N*-modular redundancy system with a majority voter. [Ch. 8]

failure—The outcome when a component or system does not produce the intended result at its interface. Compare with **fault**. [Ch. 8]

failure tolerance—A measure of the ability of a system to mask active faults and continue operating correctly. A typical measure counts the number of contained components that can fail without causing the system to fail. [Ch. 8]

fault—A defect in materials, design, or implementation that may (or may not) cause an error and lead to a failure. (Compare with *failure*.) [Ch. 8]

fault avoidance—A strategy to design and implement a component with a probability of faults that is so low that it can be neglected. When applied to software, fault avoidance is sometimes called *valid construction*. [Ch. 8]

fault tolerance—A set of techniques that involve noticing active faults and lower-level subsystem failures and masking them, rather than allowing the resulting errors to propagate. [Ch. 8]

forward error correction—A technique for controlling errors in which enough redundancy to correct anticipated errors is applied before an error occurs. Forward error correction is particularly applicable when the original source of the data value or control signal will not be available to recalculate or resend it. Compare with *backward error correction*. [Ch. 8]

Hamming distance—in an encoding system, the number of bits in an element of a code that would have to change to transform it into a different element of the code. The Hamming distance of a code is the minimum Hamming distance between any pair of elements of the code. [Ch. 8]

hot swap—To replace modules in a system while the system continues to provide service. [Ch. 8]

intermittent fault—A persistent fault that is active only occasionally. Compare with *transient fault*. [Ch. 8]

latency—As used in reliability, the time between when a fault becomes active and when the module in which the fault occurred either fails or detects the resulting error. [Ch. 8]

latent fault—A fault that is not currently causing an error. Compare with *active fault*. [Ch. 8]

margin—The amount by which a specification is better than necessary for correct operation. The purpose of designing with margins is to mask some errors. [Ch. 8]

maskable error—An error or class of errors that is detectable and for which a systematic recovery strategy can in principle be devised. Compare with *detectable error* and *tolerated error*. [Ch. 8]

masking—As used in reliability, containing an error within a module in such a way that the module meets its specifications as if the error had not occurred. [Ch. 8]

mean time between failures (MTBF)—The sum of MTTF and MTTR for the same component or system. [Ch. 8]

mean time to failure (MTTF)—The expected time that a component or system will operate continuously without failing. "Time" is sometimes measured in cycles of operation. [Ch. 8]

mean time to repair (MTTR)—The expected time to replace or repair a component or system that has failed. The term is sometimes written as "mean time to restore service", but it is still abbreviated MTTR. [Ch. 8]

memoryless—A property of some time-dependent probabilistic processes, that the

probability of what happens next does not depend on what has happened before. [Ch. 8]

mirror—(n.) One of a set of replicas that is created or updated synchronously. Compare with *primary copy* and *backup copy*. Sometimes used as a verb, as in "Let's mirror that data by making three replicas." [Ch. 8]

N + 1 redundancy—When a load can be handled by sharing it among N equivalent modules, the technique of installing $N + 1$ or more of the modules, so that if one fails the remaining modules can continue to handle the full load while the one that failed is being repaired. [Ch. 8]

N-modular redundancy (NMR)—A redundancy technique that involves supplying identical inputs to N equivalent modules and connecting the outputs to one or more voters. [Ch. 8]

N-version programming—The software version of N-modular redundancy. N different teams each independently write a program from its specifications. The programs then run in parallel, and voters compare their outputs. [Ch. 8]

page fault—See *missing-page exception*.

pair-and-compare—A method for constructing fail-fast modules from modules that do not have that property, by connecting the inputs of two replicas of the module together and connecting their outputs to a comparator. When one repairs a failed pair-and-compare module by replacing the entire two-replica module with a spare, rather than identifying and replacing the replica that failed, the method is called **pair-and-spare.** [Ch. 8]

pair-and-spare—See *pair-and-compare*.

partition—To divide a job up and assign it to different physical devices, with the intent that a failure of one device does not prevent the entire job from being done. [Ch. 8]

persistent fault—A fault that cannot be masked by retry. Compare with *transient fault* and *intermittent fault*. [Ch. 8]

prepaging—An optimization for a multilevel memory manager in which the manager predicts which pages might be needed and brings them into the primary memory before the application demands them. Compare with *demand algorithm*.

presented load—See *offered load*.

preventive maintenance—Active intervention intended to increase the mean time to failure of a module or system and thus improve its reliability and availability. [Ch. 8]

purging—A technique used in some N-modular redundancy designs, in which the voter ignores the output of any replica that, at some time in the past, disagreed with several others. [Ch. 8]

redundancy—Extra information added to detect or correct errors in data or control signals. [Ch. 8]

reliability—A statistical measure, the probability that a system is still operating at time t,

given that it was operating at some earlier time t_0. [Ch. 8]

repair—An active intervention to fix or replace a module that has been identified as failing, preferably before the system of which it is a part fails. [Ch. 8]

replica—1. One of several identical modules that, when presented with the same inputs, is expected to produce the same output. 2. One of several identical copies of a set of data. [Ch. 8]

replication—The technique of using multiple replicas to achieve fault tolerance. [Ch. 8]

single-event upset—A synonym for *transient fault*. [Ch. 8]

soft state—State of a running program that the program can easily reconstruct if it becomes necessary to abruptly terminate and restart the program. [Ch. 8]

supermodule—A set of replicated modules interconnected in such a way that it acts like a single module. [Ch. 8]

tolerated error—An error or class of errors that is both detectable and maskable, and for which a systematic recovery procedure has been implemented. Compare with *detectable error*, *maskable error*, and *untolerated error*. [Ch. 8]

transient fault—A fault that is temporary and for which retry of the putatively failed component has a high probability of finding that it is okay. Sometimes called a *single-event upset*. Compare with *persistent fault* and **intermittent fault**. [Ch. 8]

triple-modular redundancy (TMR)—*N*-modular redundancy with $N = 3$. [Ch. 8]

untolerated error—An error or class of errors that is undetectable, unmaskable, or unmasked and therefore can be expected to lead to a failure. Compare with *detectable error*, *maskable error*, and *tolerated error*. [Ch. 8]

valid construction—The term used by software designers for *fault avoidance*. [Ch. 8]

voter—A device used in some NMR designs to compare the output of several nominally identical replicas that all have the same input. [Ch. 8]

Index of Chapter 8

Design principles and hints appear in *underlined italics*. Procedure names appear in SMALL CAPS. Page numbers in **bold face** are in the chapter Glossary.

A

accelerated aging 8–12
active fault 8–4, **8–69**
adopt sweeping simplifications 8–8, 8–37, 8–51
availability 8–9, **8–69**
avoid rarely used components 8–51, 8–60

B

backward error correction 8–22, **8–69**
bathtub curve 8–10
be explicit 8–7
burn in, burn out 8–11
Byzantine fault 8–53, **8–69**

C

careful storage 8–45
close-to-open consistency **8–69**
coding 8–21
conditional failure rate function 8–14
consistency
  close-to-open **8–69**
continuous operation 8–35, **8–69**

D

decay 8–41
  set 8–42, **8–69**
defense in depth 8–3
design for iteration 8–8, 8–11, 8–15, 8–37
design principles
  adopt sweeping simplifications 8–8, 8–37, 8–51
  avoid rarely used components 8–51, 8–60
  be explicit 8–7
  design for iteration 8–8, 8–11, 8–15, 8–37
  end-to-end argument 8–49, 8–52
  keep digging principle 8–8, 8–64
  robustness principle 8–15
  safety margin principle 8–8, 8–16, 8–58
detectable error 8–17, **8–69**
Domain Name System
  fault tolerance of 8–36, 8–39
down time 8–9, **8–69**
durability 8–39
durable storage 8–38, 8–46, **8–69**

E

end-to-end argument 8–49, 8–52
erasure 8–23, **8–69**
ergodic 8–10, **8–70**
error 8–4, **8–70**
  containment 8–2, 8–5, **8–70**
  correction 8–2, 8–57, **8–70**
  detection 8–2, **8–70**

F

fail-
  fast 8–5, 8–17, **8–70**
  safe 8–17, **8–70**
  secure 8–17, **8–70**
  soft 8–17, **8–70**
  stop 8–5, **8–70**
  vote 8–27, **8–70**
failure 8–4, **8–70**
  tolerance 8–16, **8–70**
fault 8–3, **8–70**
  avoidance 8–6, **8–71**
  tolerance 8–5, **8–71**
  tolerance design process 8–6
  tolerance model 8–18
forward

error correction 8–21, **8–71**

H

Hamming distance 8–21, **8–71**
hard error 8–5
hazard function 8–14
heartbeat 8–54
hot swap 8–35, **8–71**

I

incremental
redundancy 8–21
infant mortality 8–11
intermittent fault 8–5, **8–71**

K

keep digging principle 8–8, 8–64

L

latency 8–5, **8–71**
latent fault 8–4, **8–71**

M

margin 8–20, **8–71**
maskable error 8–18, **8–71**
masking 8–2, 8–17, **8–71**
massive redundancy 8–25
mean time
between failures 8–9, **8–71**
to failure 8–9, **8–71**
to repair 8–9, **8–71**
memoryless 8–13, **8–71**
mirror **8–72**
module 8–2
MTBF (see mean time between failures)
MTTF (see mean time to failure)
MTTR (see mean time to repair)

N

N + 1 redundancy 8–35, **8–72**
N-modular redundancy 8–26, **8–72**
N-version programming 8–36, **8–72**
NMR (see N-modular redundancy)

P

pair-and-compare 8–33, **8–72**
pair-and-spare **8–72**
partition 8–34, **8–72**
persistent
fault 8–5, **8–72**
prepaging **8–72**
preservation 8–40
preventive maintenance 8–12, **8–72**
purging 8–33, **8–72**

Q

quad component 8–26

R

RAID
RAID 1 8–47
RAID 4 8–24
RAID 5 8–67
raw storage 8–42
recovery 8–38
recursive
replication 8–27
redundancy 8–2, **8–72**
reliability 8–13, **8–72**
repair 8–31, **8–73**
replica 8–26, **8–73**
replication **8–73**
recursive 8–27
revectoring 8–46
robustness principle 8–15

S

safety margin principle 8–8, 8–16, 8–58
safety-net approach 8–7
single
-event upset 8–5, **8–73**
point of failure 8–63
Six sigma 8–15
soft
error 8–5
state **8–73**
storage
careful 8–45

durable 8–38, 8–46, **8–69**
fail-fast 8–43
raw 8–42
supermodule 8–27, **8–73**

T

Taguchi method 8–16
TMR (see triple-modular redundancy)
tolerated error 8–18, **8–73**
transient fault 8–5, **8–73**
triple-modular redundancy 8–26, **8–73**

U

untolerated error 8–18, **8–73**

V

valid construction 8–37, **8–73**
validation (see valid construction)
voter 8–26, **8–73**

W

watchdog 8–54

Y

yield (in manufacturing) 8–11