2: FAULT TOLERANCE - RELIABLE SYSTEMS FROM UNRELIABLE COMPONENTS

Jerome H. Saltzer & M. Frans Kaashoek
Massachusetts Institute of Technology

2.1: 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, some of these ideas have 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 5, 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.

2.2: Faults, Failures, and Fault-Tolerant Design

2.2.1: 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.

2.2.2: 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 2.4:
   - Identify every potential fault.
   - Estimate the risk of each fault, as described in Section 2.3.
   - 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 2.5:
   - 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 the *safety margin principle*. 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 mindset 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 5 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.

2.3: Measures of Reliability and Failure Tolerance

2.3.1: 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:

$$Availability = \frac{\text{time system was running}}{\text{time system should have been running}} = \frac{\sum_{i=1}^{N} TTF_i}{\sum_{i=1}^{N} (TTF_i + TTR_i)} \tag{2.3.1}$$

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):

$$MTTF = \frac{1}{N}\sum_{i=1}^{N} TTF_i \qquad MTTR = \frac{1}{N}\sum_{i=1}^{N} TTR_i \tag{2.3.2}$$

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

$$Availability = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR} = \frac{MTBF - MTTR}{MTBF} \tag{2.3.3}$$

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

$$\text{Down time} = (1 - Availability) = \frac{MTTR}{MTBF} \tag{2.3.4}$$

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 2.3.2. 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 2.3.2, 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 2.3.9 in Section 2.3.2, 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 2.3.1 and defined more carefully in Section 2.3.2, 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.

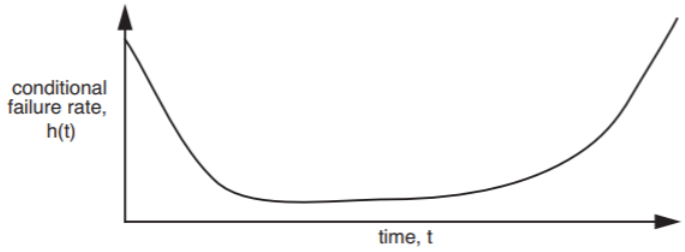

Figure 2.3.1: A bathtub curve, showing how the conditional failure rate of a component changes with time.

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.

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, can then 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 only provides information about availability, not 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 2.4.

2.3.2: 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

$$R(t) = Pr(\text{the module has not yet failed at time } t, \text{ given that the module was operating at time } 0) \qquad (2.3.5)$$

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

$$f(t) = Pr(\text{module fails between } t \text{ and } t+dt) \qquad (2.3.6)$$

(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 2.3.1) Once $f(t)$ is at hand, one can directly calculate the MTTF:

$$MTTF = \int_0^\infty t \cdot f(t)\, dt \qquad (2.3.7)$$

One must keep in mind that this MTTF is predicted from the failure rate function $f(t)$, in contrast to the MTTF of Equation 2.3.2, 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.

Sidebar 2.3.1

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 2.3.6, 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 2.3.1reports 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

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

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:

$$R(t) = e^{-\left(\frac{t}{MTTF}\right)} \tag{2.3.8}$$

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 conditional failure rate does change with time. This misappropriation starts with a fallacy: an assumption that the MTTF, as defined in Equation 2.3.7, can be calculated by inverting the measured failure rate. The fallacy arises because in general,

$$E(1/t) \neq 1/E(t) \tag{2.3.9}$$

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 Equation 2.3.8. 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 2.3.2. Suppose, for example, that a production 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.

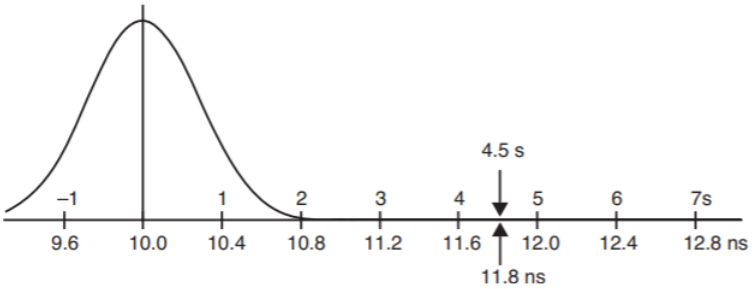

Figure 2.3.2: The normal probability density function applied to production of gates that are specified to have mean propagation time of 10 nanoseconds and maximum propagation time of 11.8 nanoseconds. The upper numbers on the horizontal axis measure the distance from the mean in units of the standard 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.

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 σ 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 2.3.2. 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 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.

2.3.3: 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 2.5 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.

2.4: Tolerating Active Faults

2.4.1: 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 5 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 its 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 2.4.1.

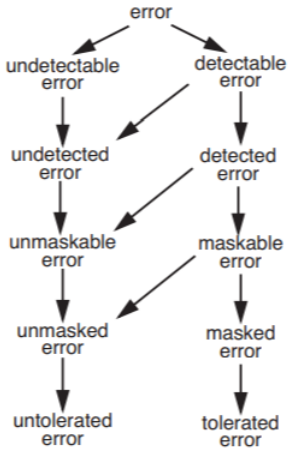

Figure 2.4.1: Classification of errors. Arrows lead from a category to mutually exclusive subcategories. For example, unmasked errors include both unmaskable errors and maskable errors that the designer decides not to mask.

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.

2.4.2: 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 2.2.2, 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 discussed in Section 1.10.1 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 2.8. This fault tolerance model employs error detection and masking techniques that are described first in Section 2.5, so the reader may prefer to delay detailed study of that section until completing Section 2.5.

2.5: 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.

2.5.1: 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. However, it is not possible to figure out what the original data pattern was. 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 1 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 23 = 8 possible patterns for this 3-bit quantity, only 4 of which represent legitimate data. As illustrated in Figure 2.5.1, 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.

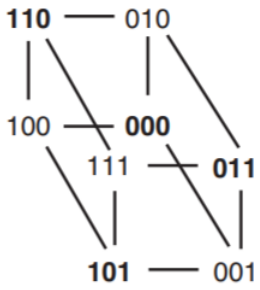

Figure 2.5.1: Patterns for a simple parity-check code. Each line connects patterns that differ in only one bit; **bold-face** patterns are the legitimate ones.

The second example, in Figure 2.5.2, 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 P1, P2, and P4 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 $P_4P_2P_1$ in order, representing the troubled bits as ONE 's and untroubled bits as ZERO 's, 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.)

	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 2.5.2: 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.

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 2.5.1, 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 2.5.2. 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 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 2.5.2. (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 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 2.10.9). 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 2.5.3. 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.

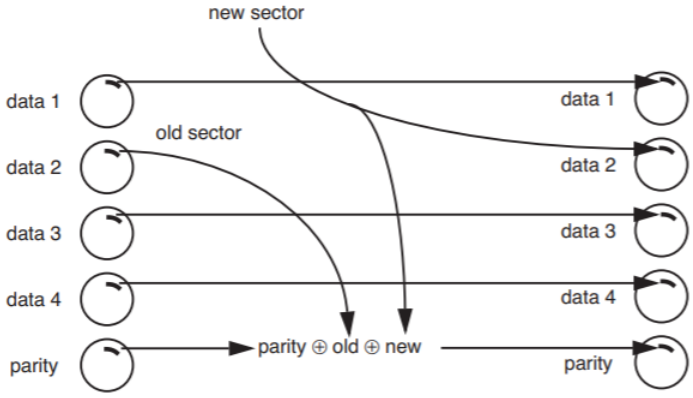

Figure 2.5.3: Update of a sector on disk 2 of a five-disk RAID 4 system. The old parity sector contains parity $\leftarrow$ data 1 $\oplus$ data 2 $\oplus$ data 3 $\oplus$ data 4 . To construct a new parity sector that includes the new data 2, one could read the corresponding sectors of data 1, data 3, and data 4 and perform three more XOR 's. 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 $\leftarrow$ old parity $\oplus$ old data $\oplus$ new data 2 .

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 2.9 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 schoolchildren 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.

2.5.2: 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.

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 2.5.4, 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.

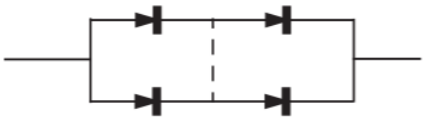

Figure 2.5.4: A quad-component superdiode. The dotted line represents an optional bridging connection, which allows the superdiode to tolerate a different set of failures, as described in the text.

2.5.3: 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 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 2.3.2, 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

$$R_{supermodule} = R^3 + 3R^2(1-R) = 3R^2 - 2R^3 \tag{2.5.1}$$

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

$$1 - R_{supermodule} = (1 - 3R^2 + 2R^3) \tag{2.5.2}$$

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 2.5.5. 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**.

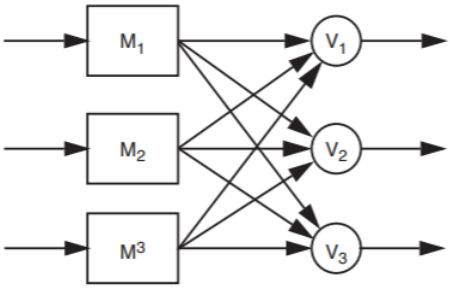

Figure 2.5.5: Triple-modular redundant supermodule, with three inputs, three voters, and three outputs.

The numerical effect of fail-vote TMR is impressive. If the reliability of a single module at time T is 0.999, Equation 2.5.1 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 2.5.6 illustrates the reliability of our hypothetical 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 2.5.7 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 2.5.1 elaborates.)

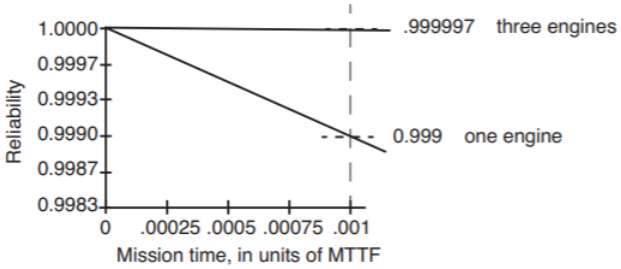

Figure 2.5.6: Reliability with triple modular redundancy, for mission times much less than the MTTF of 6,000 hours. The vertical dotted line represents a six-hour flight.

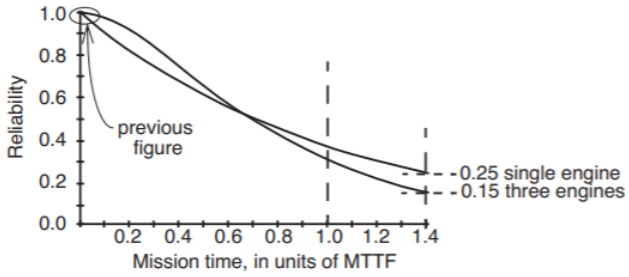

Figure 2.5.7: Reliability with triple modular redundancy, for mission times comparable to the MTTF of 6,000 hours. The two vertical dotted lines represent mission times of 6,000 hours (left) and 8,400 hours (right).

Sidebar 2.5.1

Risks of manipulating MTTFs

The apparently casual manipulation of MTTFs in Sections 2.5.3 and 2.5.4 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 $\rightarrow$ 1.5 weeks between failures

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 (2.5 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.

MTTFs 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 failfast replicas and a cleverer voter, as described in Section 2.5.4 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,

$$MTTF_{system} = 1 + 1/2 + 1/3 + \ldots (1/N) \tag{2.5.3}$$

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**.

2.5.4: 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 one of the two 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

$$Pr(\text{supermodule fails while waiting for repair}) = \frac{2 \times MTTR}{MTTF} \tag{2.5.4}$$

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):

$$MTTF_{\text{supermodule}} = \frac{MTTF_{replica}}{3} \times \frac{MTTF_{replica}}{2 \times MTTR_{replica}} = \frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} \tag{2.5.5}$$

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 Equation 2.5.5, the MTTF of the data is

$$\frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} = \frac{(5 \text{ years})^2}{6 \cdot (10 \text{ hours})/(8760 \text{ hours/year})} = 3650 \text{ years} \tag{2.5.6}$$

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 4 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 2.5.5 we have $MTTR_{replica} = 1$ (in this application, Equation 2.5.5 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 2.5.8 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.

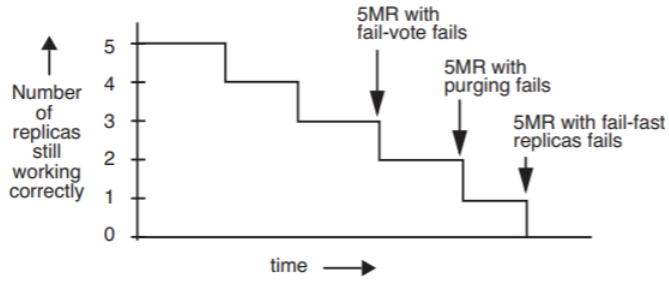

Figure 2.5.8: Failure points of three different 5MR supermodule designs, if repair does not happen in time.

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 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 $\mathbf{N+1}$ **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 2.5.1, or some more general replica management system such as those discussed in Section 4.4.7.

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.
