2.2: Faults, Failures, and Fault-Tolerant Design

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 system, 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.

---

**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 \(\langle 1 - p \rangle^N\). 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
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.