**ECE 350: Real-Time Operating Systems** Spring 2023

# Lecture 25 — Reliability: Fail-Soft Operation

*Jeff Zarnett* *2023-09-09*

## Task Failed Successfully

Except in the recent discussion of how hard drives may die and in a previous course's discussion of the Byzantine Generals Problem, we usually go through life assuming that computer systems will work as they are intended. That does not rule out the possibility of a design problem or implementation bug, of course. But in many situations, we actually have surprisingly low expectations of computer systems. Things crash, we reboot them. Something going wrong? Retry it.

Downtime is sometimes okay to address a problem, whether it's by fixing the system or replacing it. If my laptop should go for an unplanned swim in a lake, it's out of action, probably permanently. I can just buy a new laptop. I'm not happy, to be sure, but my laptop makes no promises of uptime. In a less dramatic scenario, I might install an OS update and it will be down for some period of time. The OS gives me an estimate of how long it's going to be, but that's a guess and not a promise. But maybe I need a system where that doesn't happen, even if some hardware or software update is required.

Reliability is an important element for real-time systems. Downtime may be intolerable in the case of life- or safety-critical systems, or just very expensive in the sense of needing to meet a service level agreement and having to pay contract penalties to customers if the agreement is not maintained.

If I am travelling to work via a car and get a flat tire, the car is out of action while the tire is changed. After that, I can go on: that's a repair with downtime. But some cars have the concept of run-flat tires, which allow you to travel for a limited distance at reduced speed if a tire is flat. This is to say, the system remains functional at reduced capacity until the problem can be addressed.

What we'd like to have for a real-time system is generally two distinct things: resiliency and fail-soft operation. A system that is more resilient is capable of carrying on in the event of a failure, even if not at full capacity; and fail-soft operation says that if things do indeed fail then the system will preserve as much capability as it can or terminate gracefully if it must [Sta18].

In the previous topic, we discussed what we can do to mitigate the impact of hard drive failure. That established the fact that it's not realistic to try to get ironclad devices that will last a hundred years, but instead through redundancy. If you want the system to be capable of surviving the death of a CPU, it needs to have more than one CPU in the system. Right?

### Resiliency

In the words of Rocky Balboa, "It's not about how hard you hit. It's about how hard you can get hit and keep moving forward. How much you can take and keep moving forward.". When designing a system for resiliency, the question to consider is how much resiliency do you really need?

If you are designing something to not have downtime because the company loses some money when the system is down, your considerations are different to a situation where you need to avoid downtime because lives are on the line. Remember also that there are some scenarios that you cannot and should not handle because they're either too outlandish or just outright apocalyptic. If Russia decides it wishes to launch its nuclear missiles[1] and your data

[1] August 29, 1997 2:14 AM Eastern Time... `https://www.youtube.com/watch?v=4DQsG3TKQ0I`

centre is in one of the target zones, does it matter if your app is going to have some downtime right now? I think there are bigger problems to worry about.

Nevertheless, this is an engineering design tradeoff between cost and the amount of resiliency you want to have. Up to a certain point, more money will get more redundancy and more resiliency. Remember the earlier example about how a lack of spare crew capacity can result in delaying of your flight from Calgary to Vancouver? If a certain airline were more open to increasing the cost, they could have more capacity. That would make the system more resilient: a snowstorm in Nova Scotia wouldn't ruin the travel plans of unrelated people several thousand kilometres away. On the other hand, at some point it becomes wasteful: the airline should not be paying excessive amounts of people to hang around doing nothing just in case.

Let's imagine that something has actually gone wrong, for whatever reason. There are a few different ways to respond or handle this situation. For the sake of the example, we'll consider primarily hardware failure, but software failures matter too.

**Can We Fix It? Yes We Can!** The best option for resiliency is if the system can correct the problem and carry on! That may not be sensible if the scenario is an unrecoverable hardware failure: if the magic smoke has come out of the hard drive, no amount of rebooting it will make it go back in again. However, if the failure would be solved by re-initializing the device, why not?

Fixing some things may be easier if it's a software problem. Remember that we talked in a previous course about the ideas around deadlock recovery. Deadlock is a failure, and perhaps the system can recover from failure by pursuing one of those approaches, such as killing processes or rebooting the system.

During the process of recovery, though, the system is probably running at a lower capacity. So let's consider that...

**Stiff Upper Lip.** If the system cannot be restored to full capacity automatically, then the next best thing is if the system can continue as best it can, most likely at a reduced capacity.

Suppose the system has 4 CPU cores and one of them dies but the CPU was never used at more than 50% capacity even when all CPU cores were healthy. The system is worse off, yes, and its maximum capacity has been reduced, but in practice the system is still running at its original capacity. That's one reason for redundancy in the system, to be sure. That's the ideal case, but did the system really get designed with that much extra capacity?

The next possibility is that the system is running at reduced capacity until external forces come to repair the system. Reduced capacity in this sense may be that it takes longer to get answers, or less work can be done in the same amount of time.

In a real-time operating system, this loss of capacity might mean it's no longer possible to meet all deadlines. The system is considered *stable* if it will always meet the deadlines of its most critical tasks even if lower-priority tasks do not get completed [Sta18]. In this sense, it means that in the event that painful choices must be made, the most important things are prioritized. We'll consider some other scenarios for handling a problem, but then we'll return to focus on how to carry on at reduced capacity.

**Shut It Down!** Perhaps instead of throwing the emergency stop when things go wrong, it's possible to do an orderly shutdown. The traditional UNIX system, if it detects kernel data corruption, will write the contents of memory to disk for analysis and stop execution [Sta18]. If the problem encountered is something we cannot live with but also cannot fix, then this is a valid solution to making sure no additional damage has been done.

**Stop. Hammertime.** One option for failing is to just cease all execution or operation immediately. If the industrial machinery controller is in a bad state, maybe the best thing to do is actually to stop everything so the equipment stops moving. That may reduce the potential for damage to the materials being manufactured or risk to the people working in the facility. Downtime for the system is bad, but not as bad as the system killing someone.

## Faults and Fault Tolerance

In the previous section, we just said vague things about something going wrong like hardware dying or something like that. But let's take a look at what is a fault and what does it mean to be fault tolerant.

A failure is the term we use when the response (outcome) deviates from the specification as a result of an error; and an error is a manifestation of a fault [HZMG20]. But what's a fault? The IEEE Standards Dictionary and some textbooks define a fault as an erroneous hardware or software state of some variety [Sta18]:

- **Permanent**: A fault that is always present after it occurs and can only be fixed by actually replacing or repairing the faulty component. If a GPU ceases to work due to overheating causing a failure of its internal components, that is now permanently broken and can maybe be repaired but probably has to be replaced, so it's a permanent fault. Software bugs are also permanent faults: the bug was always in the code, even if it wasn't triggered for some reason (e.g., the bug occurs only on the 29th of February) and accordingly must be fixed by a code change.
- **Intermittent**: A fault that recurs at random, unpredictable times. A faulty chip may produce the right answer most of the time but the wrong answer rarely and unpredictably. That's super frustrating because it's hard to track down and identify what's wrong.
- **Transient**: A fault that occurs once but does not recur. The standard example of this is cosmic radiation flipping a random bit from 0 to 1, but I don't think this is actually that great of an example because things like satellites (especially the ones outside the earth's magnetic field) have to contend with this as just a fact of life, making it more intermittent than transient. For terrestrial systems, maybe this is an appropriate example.

**Fault Prevention.** Although it's tempting to jump straight to the idea of fault tolerance, which is about how to carry on in the face of the problem, it's even better if we can prevent the problem from occurring in the first place. That leads us to fault avoidance, which is some combination of some of the following steps [HZMG20]:

1. Make sure you understand the physical environment and protect the system from physical damage through appropriate shielding, waterproofing, etc.;
2. Use hardware with the intended reliability and lifetime (don't cheap out on the components!);
3. Within software development...
   (a) Have a sufficiently detailed and rigorous specification;
   (b) Use an appropriate design and test methodology;
   (c) Use the right programming language (Rust?);
   (d) Use analysis tools.
4. Verify that components and tools work as they advertise!

Following appropriate design, implementation, and testing processes will only make faults less likely. It is also important to resist the pressure to drop or degrade these processes when user time pressure. Perfection is generally unachievable, so it's more likely we'll have to think about how to deal with faults.

**Fault Tolerance.** Some of the things we covered much earlier in the course, or in an earlier course, play a role in supporting the resilience of the system in the face of software faults [Sta18]:

- **Process Isolation**: The most obvious example of how this is relevant is that the OS prevents one process from accessing the memory of another or the memory of the operating system itself. This limits the potential damage if something goes wrong in one process (e.g., dereference a bogus pointer), it does not affect other processes or the operation of the OS.

- **Dual-Mode Operation**: The OS monopolizing access to hardware devices and other common resources will, again, reduce the potential damage of a thread, maliciously or otherwise, interfering with the execution of others by cancelling other tasks.
- **Preemptive, Priority-Based Scheduling**: As with the previous example, preemptive and priority-based scheduling mean that misbehaving processes or threads have limited impact on other threads that want to run. Without this, a rogue thread could monopolize CPU time and negatively impact others.
- **Checkpoints, Transactions, Rollback**: These were covered in the concurrency course and can be used to recover from failures, transient or otherwise.

As for hardware failure, we already learned some of the important elements about fault tolerance through discussing the various RAID levels. In that topic we covered the idea of having backups or mirrored/striped volumes which gave is information redundancy: there's more than one copy of the data so the loss of one copy does not result in the permanent loss of data.

Most likely you covered some ideas of information redundancy in other courses around communication. Checksums, parity bits, error correcting codes, etc. all allow for detecting and possibly correcting errors through the use of data redundancy.

There is also the idea of physical redundancy, which is what happens if we have, say, two CPU chips in the machine such that if one of them dies then the system can carry on with reduced capacity. But it also describes the idea of having more than one physical machine or instance of your application, so if one instance or machine goes down (even for maintenance) the other can continue on. Ideally this sort of physical redundancy extends to having the systems in more than one physical location so that your application can still be available even if the Amazon US East 1 data centre is currently offline.

There's a third kind of redundancy called temporal redundancy: repeating an operation when an error is detected [Sta18]. This is what happens in, for example, TCP network communication, where a receiver who notices a packet is missing or damaged can request that this packet be re-sent to make sure the data received is correct. You may also consider the kind of redundancy where you have more than one execution of the task to verify that the results agree.

The space shuttle had a combination of physical and temporal redundancy for important calculations. It had five computers, but normally only four of them were used in a majority vote system. If one of the systems fails, it is outvoted by the others; if two systems fail in different ways then they get different answers and the two correct systems win the vote; if two systems fail in the same way and there's a tie, the fifth computer is activated and it will compute the problem and then vote to break the tie [HZMG20].

The important thing is that the system should have no single point of failure. A system that has such a single point of failure is vulnerable to total failure if that one component is out of action. And while it's sometimes possible to fix this in the system design phase, the deployment matters too: if you have two systems with multiple CPUs and redundant disks and extra RAM, but they share a network connection, they're not really independent because of that shared component.

## Now I Have Two Problems

If we have two distinct systems that need to work together, we've just unlocked a whole new tier of problems called distributed systems problems. We actually covered one of the possible issues in distributed systems in the past when we talked about the Byzantine Generals Problem. That was framed around finding traitors when trying to complete the mission of the Emperor, but there's lots of other things that we will have to consider. We should examine one in detail – clock synchronization.

**What Time Is It?** The idea of temporal redundancy introduces the problem of time and that it is not trivial to get two independent systems to agree on what time it is. Inevitably, all clocks except the universal reference clock

are off by some amount; it's just a question of how much! This is not really an indication of anything wrong with the clock synchronization, but just a question of the laws of physics as we'll see.

It's easy to imagine scenarios where independent systems who don't agree on what time it is will misbehave. Suppose the primary system is supposed to take some action at exactly 08:00 and this action takes 0.5 seconds to complete and it should signal completion. If the action is not completed within 1 second, the backup system will assume that the primary has failed and it will perform the action. If their clocks disagree by 1 second, then the backup system will do the action again unnecessarily (or perhaps cause an error by doing so). Is that example a little bit silly? Yes, but if you've ever been in a video call with delay and it's really frustrating because you each keep talking over one another ("you go ahead", "no, after you..."), you have experienced the pain of what happens when we don't agree on what time it is.

Time zones also matter, but I'm going to recommend using UTC because it doesn't have weird things like Daylight Savings Time where you can get a jump in the time forward or back. And back is worse because it means a certain minute appears to happen more than once (defeating our normal expectation of linear time). That can really be a problem if an action that is supposed to be taken once per day is taken twice or skipped. UTC also avoids issues where the California system thinks it's Monday and the Singapore system thinks it's Tuesday. UTC might be weird, though, if your application is running on a Mars Rover; at that point you might as well use Stardates (Captain's Log, Stardate 46119.1...).

Clocks frequently use quartz for synchronization, but there is always drift and measurement error; a quartz clock will typically vary by about half a second per day, so the idea of systems being off by a full second is quite reasonable if System A is fast by 0.5s and System B is slow by 0.5s [HZMG20].

In a non-real time operating system, it's often okay to just change the clock to the correct time and just jump there. But for a real-time system, breaking the expectation of linear time can cause events to run again, so typically we do not wish to do that. The solutions are effectively a graduated slowdown or speedup (i.e., counting a second as a little longer or shorter until things are back in sync). But that presupposes that we know what time we are trying to adjust to, and how do we know that?

That's a more complicated question than we want to discuss here, but a possible solution is something like the Network Time Protocol – NTP – which allows querying a reference source. The protocol itself is designed to compensate for some forms of network delay, although not all. Effectively, it's difficult or impossible to get more than one system to agree on what time it is.

Ultimately, this means the design of communication and synchronization between the systems must account for the communication time but also that the clock times will not be identical, to the point where messages could appear to arrive before they're sent.

**Distributed Systems Course.** All of this is just a very simple overview of some of the issues that might arise when we have multiple systems for redundancy. This is a complicated subject and is a whole 4th year ECE technical elective (ECE 454) that you could take if this is of interest to you. Did I just plug a 4th year TE that isn't ECE 459? Oh my!

## References


[HZMG20] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2020. Online; version 0.2020.07.19.

[Sta18] William Stallings. *Operating Systems Internals and Design Principles (9th Edition)*. Pearson, 2018.