

Why redundant systems aren't always redundant

Let's not confuse effort with outcomes



Workbook

Presented by

Dr Chris Jackson

Reliability engineer



Copyright © 2022 Christopher Jackson

All rights reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means, including photocopying or other electronic or mechanical methods without the prior written permission of the author (Christopher Jackson), except as permitted under Section 107 or 108 of the 1976 United States Copyright Act and certain other non-commercial uses permitted by copyright law.

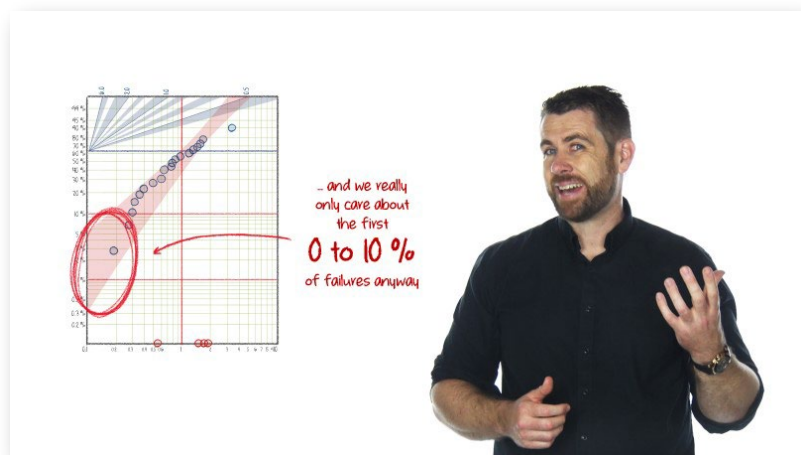
For permission requests, write to Christopher Jackson using the details on the following pages.

Limit of Liability/Disclaimer of Warranty: While the author has used his best efforts in preparing this document, he makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. The author shall not be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Who is your teacher? ... Dr Chris Jackson

Dr Jackson holds a Ph.D. and MSc in Reliability Engineering from the University of Maryland's Center of Risk and Reliability. He also holds a BE in Mechanical Engineering, a Masters of Military Science from the Australian National University (ANU) and has substantial experience in the field of leadership, management, risk, reliability and maintainability. He is a Certified Reliability Engineer (CRE) through the American Society of Quality (ASQ) and a Chartered Professional Engineer (CPEng) through Engineers Australia.

Dr Jackson is the cofounder of is4.com online learning, was the inaugural director of the University of California, Los Angeles (UCLA's) Center of Reliability and Resilience Engineering and the founder of its Center for the Safety and Reliability of Autonomous Systems (SARAS). He has had an extensive career as a Senior Reliability Engineer in the Australian Defence Force, and is the Director of Acuitas Reliability Pty Ltd.



He has supported many complex and material systems develop their reliability performance and assisted in providing reliability management frameworks that support business outcomes (that is, make more money). He has been a reliability management consultant to many companies across industries ranging from medical devices to small satellites. He has also led initiatives in complementary fields such as systems engineering and performance-based management frameworks (PBMFs). He is the author of two reliability engineering books, co-author of another, and has authored several journal articles and conference papers.

Table of Contents

1. Variability control	5
2. Fault Containment	5
3. Fault Isolation	6
4. Proactive Maintenance	6
5. Reversion.....	6
6. Redundancy	7
PARALLEL SYSTEMs	7
reliability when FAILURE IS NOT INDEPENDENT	10
Redundancy – in software	14
Human error	15

Once we have exhausted all practical and feasible **smart design activities** AND we still need to improve **reliability**, we might need to start **tolerating faults**. This could be entirely acceptable if you have really thought about **reliability** critically from the start. But like most **reliability activities**, it needs to be done right!

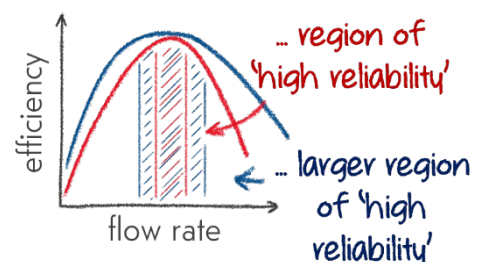
There are many things we need to do before we start thinking about **tolerating faults**. Starting with a (properly done) **FMEA** is always a good thing. You might use your **FMEA** to help **allocate reliability goals** to your subsystems and components. And when this is done well, you have well thought out and flexible goals that help keep your team on track, on time, and on budget.

Individual design teams can then use a host of **Design for Reliability (DfR)** activities at their disposal. Perhaps they will conduct **Highly Accelerated Life Testing (HALT)** on initial prototypes. Perhaps they will derate. Once they have identified the **VITAL FEW failure mechanisms**, they can be targeted for design improvement.

But there comes a point where we cannot eliminate any more faults from our design – at least not efficiently or cheaply. This means that we now need to **tolerate faults**.

1. Variability control

Variability control involves using a **component that can tolerate larger variation in inputs caused by failures or faults in other components**. This is sometimes called **robustness**. It essentially means that the component keeps working well when confronted with 'out of spec' conditions.



2. Fault Containment

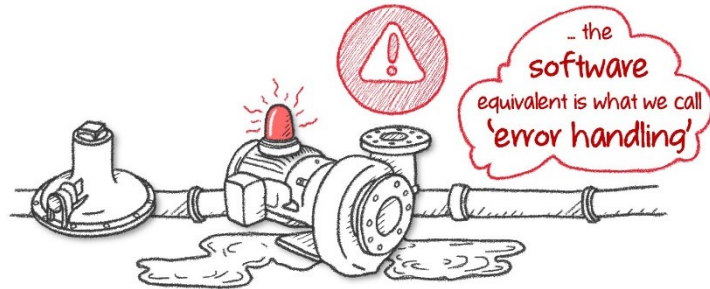
Fault containment is the **prevention of the propagation of a failure mechanism across a system due to a fault in a single component**. For our pump, this could involve an upstream regulator.



There are lots of examples of **fault containment** that range from voltage regulators through to filters. Each component has a different range of options based on the dominant **failure mechanisms (VITAL FEW)**.

3. Fault Isolation

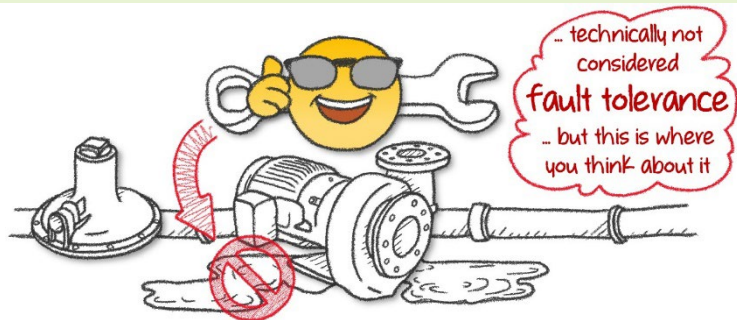
Fault isolation involves **diagnosing a fault when it occurs**. This allows immediate action to address the **fault** to either prevent it from progressing to a **failure**, or to minimise the time spent in a 'failed' state.



There are things like **Built-in Testing (BIT)**, **Built-in Self Testing (BIST)** and **Built-in Test Equipment (BITE)** which can help automate **fault isolation**. These systems might investigate the entire system during start up to make sure that performance characteristics are 'in specification,' the output voltage is within limits et cetera. In software, this process is largely analogous to the **error handling** process where software can detect when an **error** has occurred, what piece of code was responsible for the **error**, and otherwise advise the user on what he or she might be able to do.

4. Proactive Maintenance

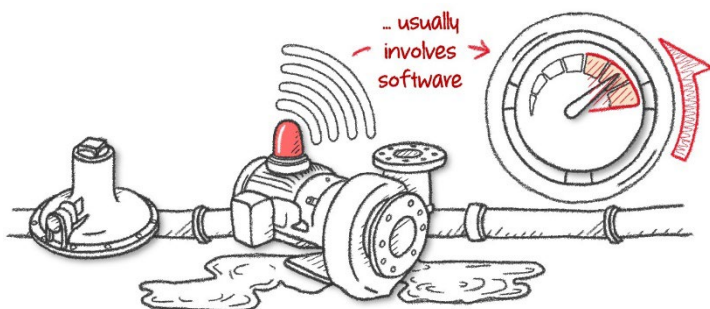
Proactive maintenance or **Condition Based Maintenance (CBM)** is all about **addressing faults before they become failures**. So it has a natural 'relationship' with **fault isolation**.



Many reliability engineers don't consider **proactive maintenance** or **CBM** a true '**fault tolerance**' activity. But in practice ... you need to think about **proactive maintenance** and **CBM** at the point of design where you start thinking about **tolerating faults**. Instead of **BIT**, **BIST** or **BITE**, systems might have sensors that detect the accumulation of damage (lubricant contaminants, fatigue crack length, temperature et cetera) to allow maintenance to occur before **failure**.

5. Reversion

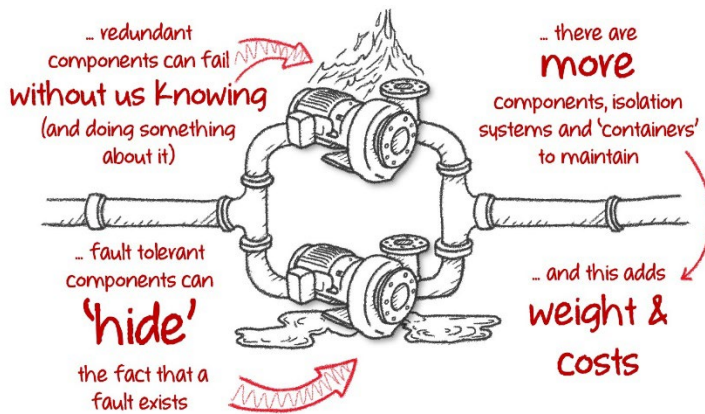
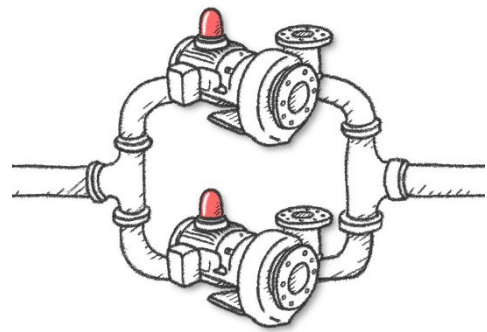
Reversion (which is sometimes referred to as **derating**) is the **transition to a sub-optimal operation state that prevents imminent failure caused by an isolated fault**. This too is related to **fault isolation**.



An example of reversion is an engine RPM limiter that enacts when a leak in the coolant system is identified. This will reduce the top speed of a vehicle but will allow the vehicle to continue operating. While **reversion** can be manual (as in, an operator manually detects a fault and adapts the use conditions accordingly), it is typically automated using sensors and software.

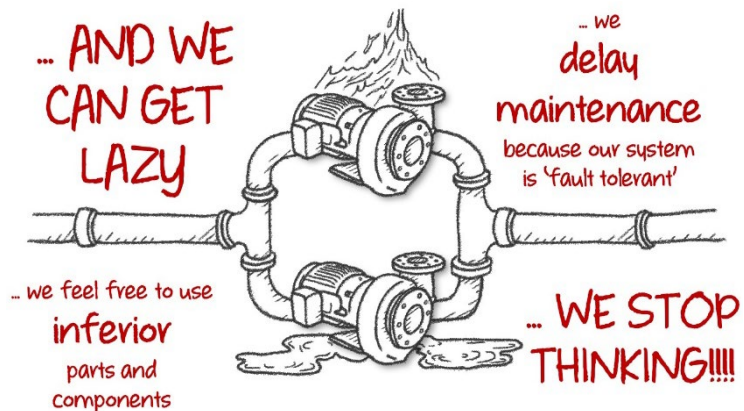
6. Redundancy

Redundancy is the **duplication of components or functions**. For a pump, this could involve the installation of a second **active redundant, standby redundant or load sharing pump**. The idea is that if one **fails**, the other will allow the system to continue operating.



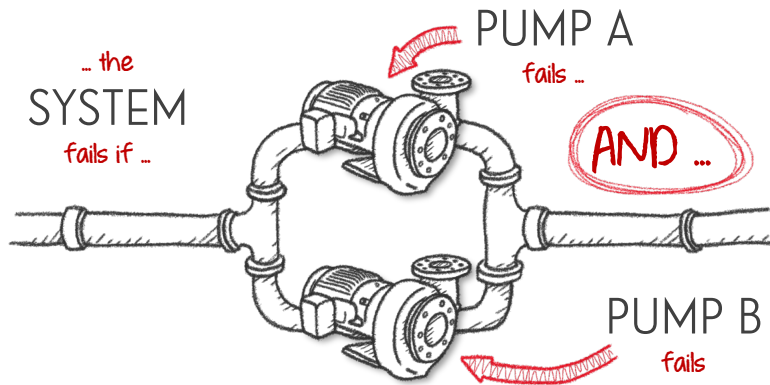
Redundancy comes at a cost – so we need to think about it. There is a hidden issue with redundancy – **Common Cause Failure (CCF)**. Single **faults** often cause failure in **primary** and **standby** components, meaning we don't get the **reliability** improvements we hope for.

But perhaps one of the more egregious issues with **redundancy** can be the effect it can have on **culture**. **Redundancy** (and **fault tolerance**) often introduces a false sense of security. So we stop thinking and start assuming that everything is going to be OK. Our systems degrade to a point where we have no **fault tolerance**.

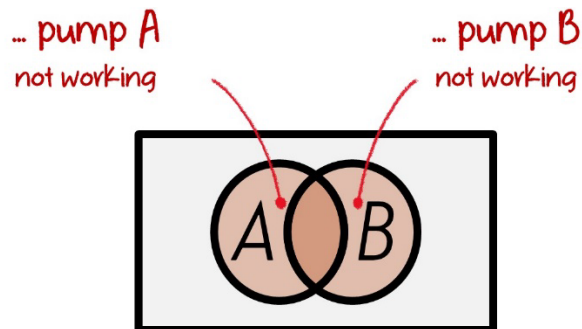


PARALLEL SYSTEMS

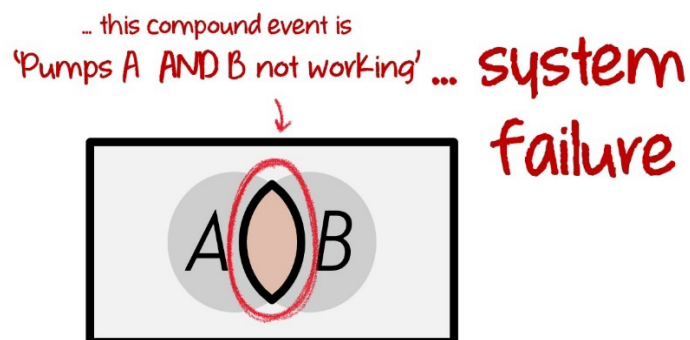
Let's now look at our two-pump **parallel system** where the system fails if *pump A* fails **AND** *pump B* fails.



Going back to the **Venn diagram** we looked at in our [Basic probability and stuff](#) lesson.

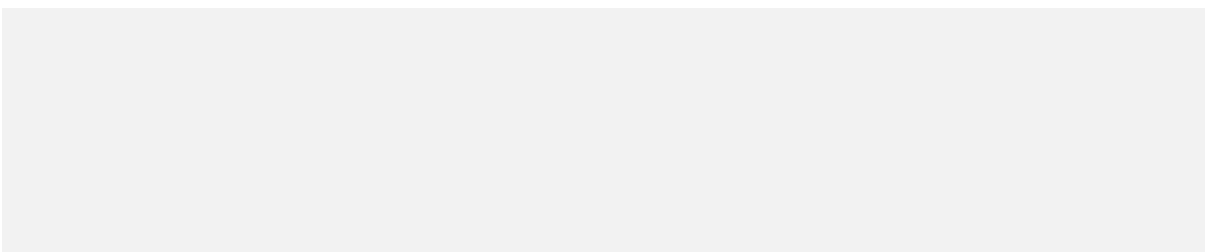


So because **failure** occurs when *pump A* AND *pump B* fails ...

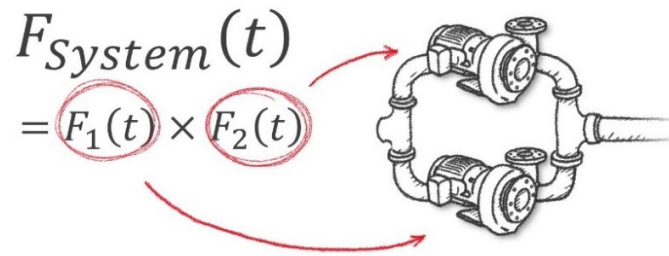


this is an **INTERSECTION** event

You can see how much 'smaller' the system **failure event** is when compared to the **series system**.



Remember that **series system reliability** is the product of **component reliability** ... **parallel system failure probability** is the product of **component failure probabilities**.

$$F_{System}(t) = F_1(t) \times F_2(t)$$


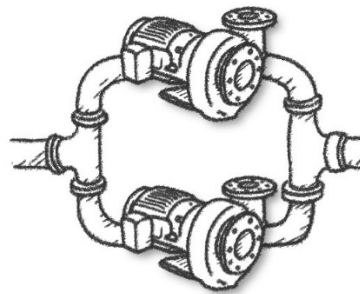
But of course, **parallel systems** can have lots of components. If we have a **parallel system** with n components (and **failures** are **statistically independent**), **system failure probability** is ...

$$F_{System}(t) = F_1(t) \times F_2(t) \times \dots \times F_n(t) = \prod_{i=1}^n F_i(t)$$

$$R_{System}(t) = 1 - \prod_{i=1}^n 1 - R_i(t)$$

EXERCISE

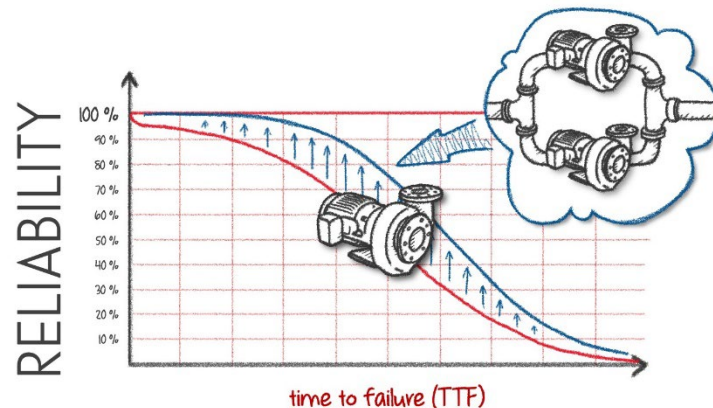
Consider the two-pump parallel system below.



What is the reliability of the system at 2 years if the reliability of each pump at 2 years is 70%?

$$\begin{aligned}
 R_{System}(t) &= 1 - \prod_{i=1}^n 1 - R_i(t) \\
 &= 1 - \prod_{i=1}^2 1 - R_i(t) = 1 - \prod_{i=1}^2 0.3 \\
 &= 1 - 0.3 \times 0.3 = 1 - 0.09 \\
 &= 0.91
 \end{aligned}$$

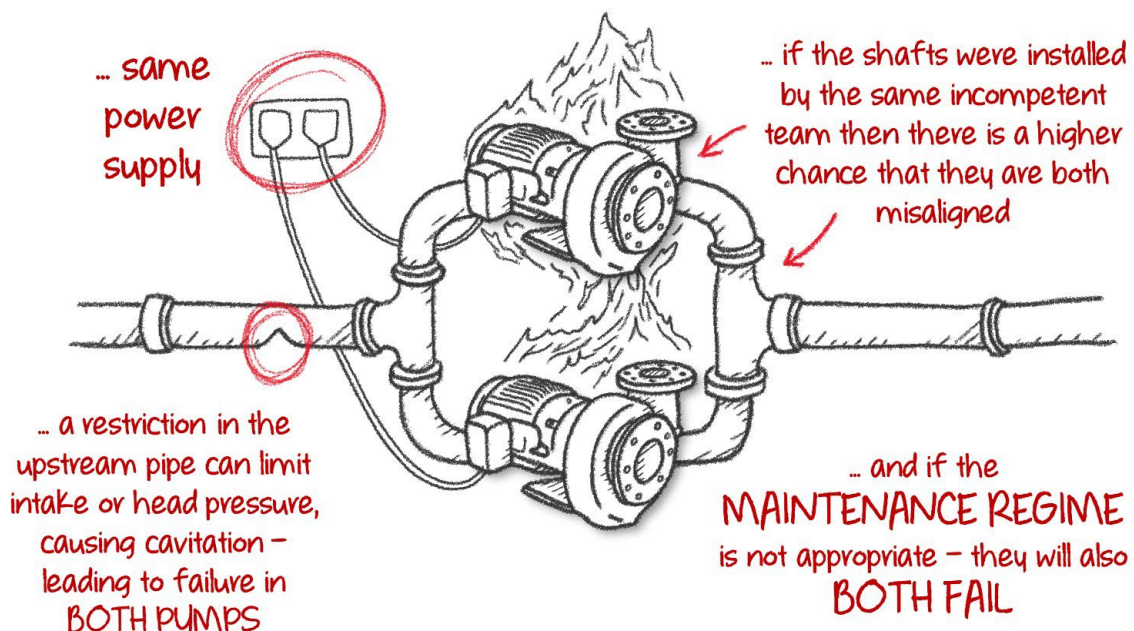
Using the techniques we have just learned, we can calculate system **reliability** at any time or usage for **parallel systems**. This is how we get the **reliability curves** illustrated below.



reliability when FAILURE IS NOT INDEPENDENT

Failure is rarely **statistically independent** – and this affects our system **reliability** estimates when there is **redundancy** involved (**parallel** or ' k out of n ' systems).

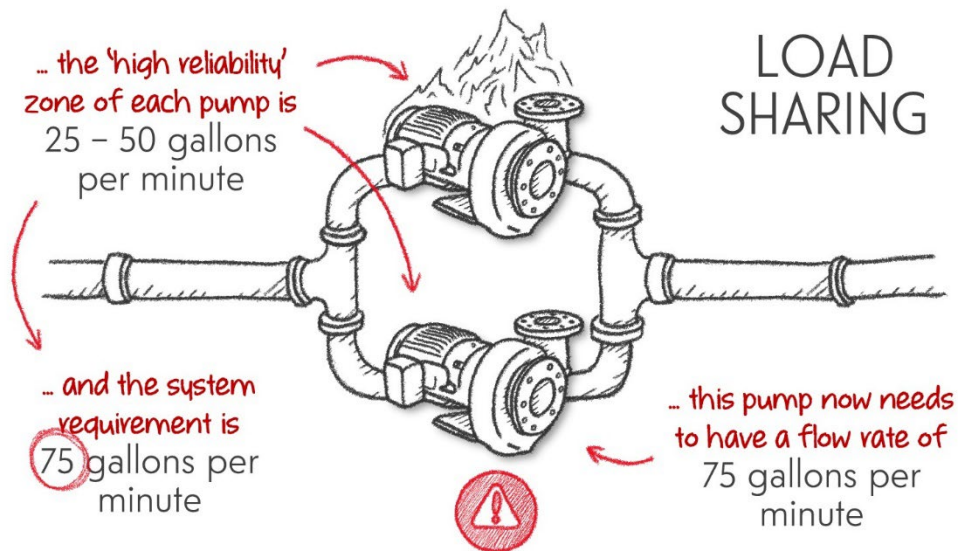
We know many **failures** have **common causes**. Some example **common cause failures** for our **two-pump parallel system** are illustrated below.



Common cause failure is most relevant for **parallel systems**. As **parallel systems** involve **redundancy**, this also (unfortunately) means that the **reliability** improvements we hope see by using **parallel systems** are eroded by any failures that have a **common cause**.

This is not a significant issue for **series systems**, where if any one component fails the entire system fails. **Common cause failures** mean it is more likely for other components to fail after the first component fails. Given the **series systems** fails when the first component fails, this does not change any **system reliability**.

Another form of **dependent failure** occurs if the system is configured in 'load sharing' configuration. Consider the following, seemingly **parallel two-pump system**.

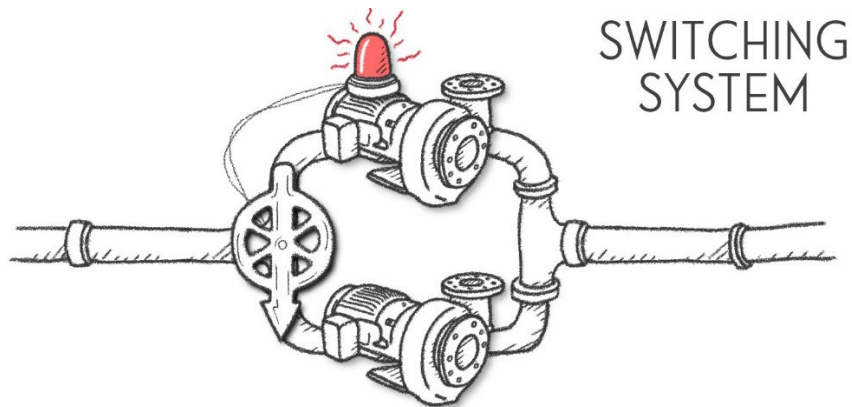


The system is required to pump 75 gallons per minute. Both pumps can have their service life extended by making sure that they pump somewhere between 25 and 50 gallons per minute. When both pumps are working, they need to pump at a rate of 37.5 gallons per minute for the. This means they both remain in a 'high-reliability' zone.

However, when one of them **fails**, the remaining functional pump needs to pump at 75 gallons per minute. It can still do this, but it increases the likelihood of failure dramatically.

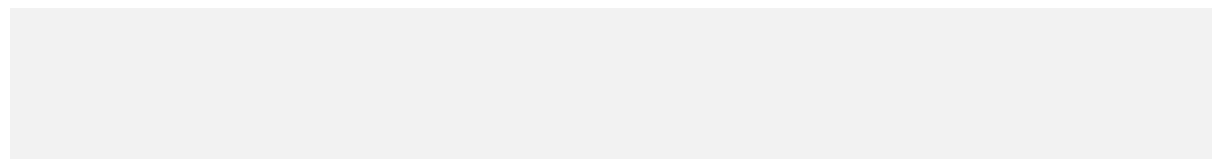
So even though one pump **failing** won't cause the other pump to **fail** immediately, it will certainly result in an increased **failure probability** of that remaining pump. The **equation** or **function** that provides system reliability for a **load sharing** configuration requires complex modelling outside the scope of this course.

A **switching system** is a **redundant system** where additional components kept in a **'standby state'** until the **primary component** fails.

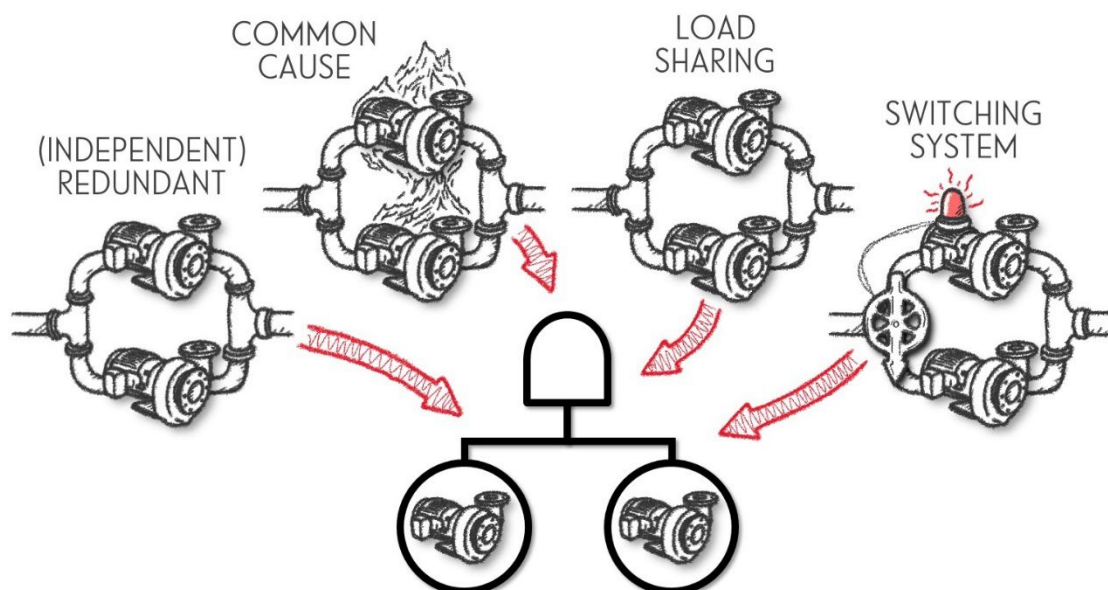


A **switching system** needs a switch mechanism that can recognize when the first pump **fails**, and then successfully **'switch'** the **standby pump** to an **'active'** state. This introduces additional ways the system can **fail** but can substantially increase system **reliability**.

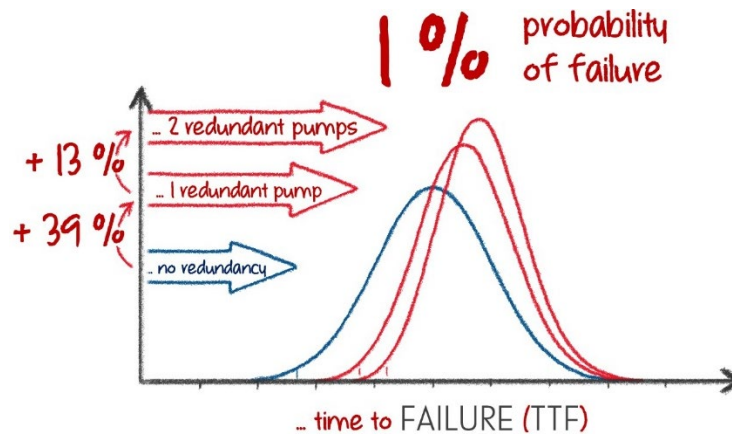
The system **reliability** equation that takes into primary pump reliability, pump reliability in a **standby state** and switch mechanism **reliability** is complex and outside the scope of this course.



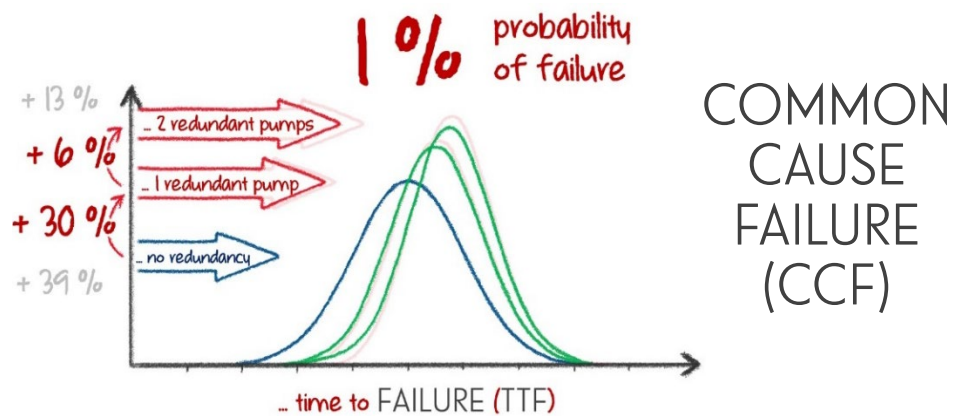
There are many ways of configuring **redundant systems**. However, **RBDs** and **fault trees** are not always good at representing them in a way that allows us to identify which **redundancy configuration** is being represented. Further, **common cause failure** exists in virtually every real-life scenario. While we can incorporate **common cause failure** into an **RBD** or **fault tree** most **RBDs** and **fault trees** don't inherently do this.



Going back to **common cause failure** which we first started talking about in our [Fault tolerance](#) lesson, we looked at the effect on how long it takes before we expect 1 % of our systems to fail when we add one and two redundant components.



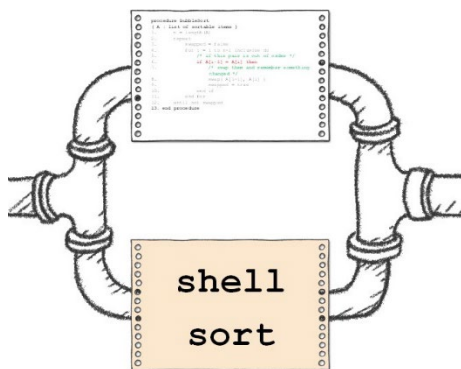
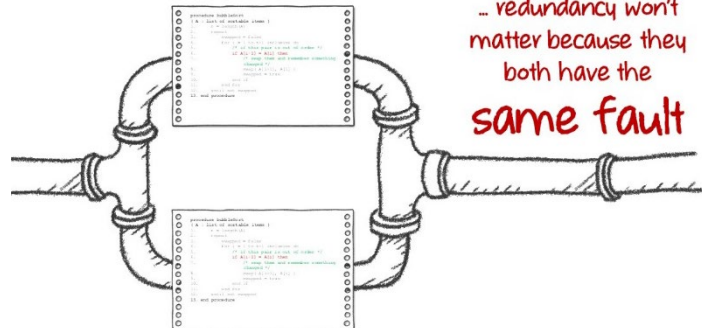
But if just one in ten failures have a **common cause**, then **we don't have redundancy one in ten times.** So instead of getting an increase of 39 % on our **warranty period** (for a 99 % warranty reliability), **common cause failure** reduces this increase to 30 %.



Redundancy – in software

Software failure is random – but in a different way to **hardware failure**. **Software failures** are caused by **coding errors** – which are obviously somewhat random. But once code is complete, software will behave in repeatable, non-random ways. So having defective code in '**redundancy**' will not work.

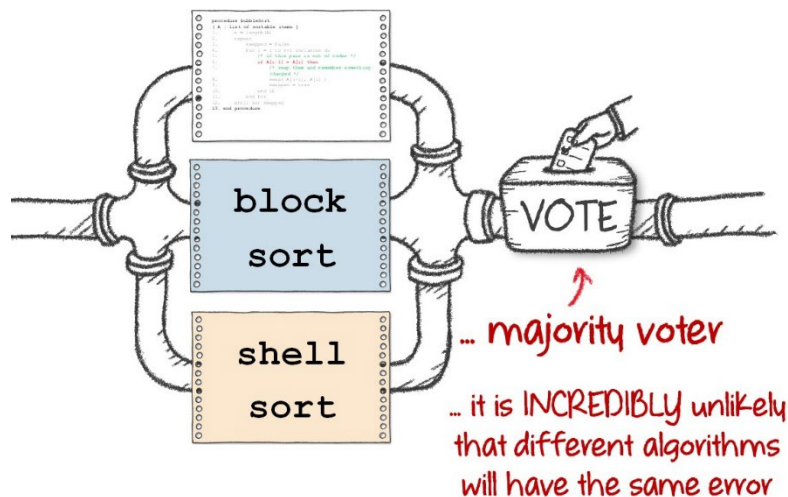
... the ultimate
common cause failure



One approach is to have different code in a **redundant configuration**. For example, let's say we are coding software to simply sort numbers. A well-known algorithm is the '**bubble sort algorithm**.' We might get a team of software engineers to code a program based on the '**bubble sort algorithm**.' We then might ask another team to code another program based on the '**shell sort algorithm**.' Both algorithms achieve the same thing, but used different methods.

So we can put the **shell sort algorithm** in redundancy to support the **bubble sort algorithm**. Any **coding errors** in one program are unlikely to manifest themselves in exactly the same way in the other. But there is still a problem. If one program has an **error**, then this means the two programs will have different outputs. How do we know which one is correct?

The solution involves having three different programs, coded by different teams, all acting in '**redundancy**.' We then have a '**voter program**' which can identify when at least two outputs are identical. Because it is highly unlikely for two identical outputs to result from two different errors, we likely have a 'correct' output.



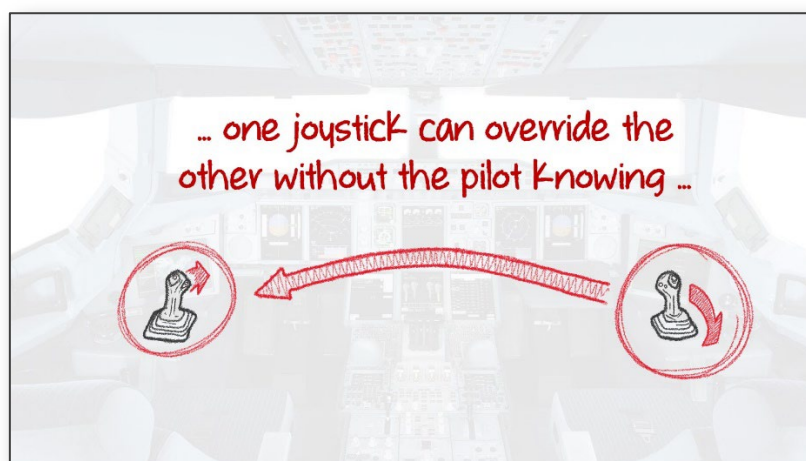
This is called **N-version programming** – where '**N**' refers to the number of 'redundant' programs.

There are lots of different approaches to software that make it **fault tolerant**. **N-version programming** is only one approach. Remember that **software error handling** is a form of **fault isolation**. This comes down to the coding approach and strategy, where 'little' things like coding in a coherent, commented manner that allows other software engineers to modify your code in the future is very valuable.

Human error

Human reliability and **human factors** are separate areas of study. Nuclear power plants are examples of complex systems where humans are just as important as the valves and pumps of the emergency cooling system. Any scenario where humans need to be trained and skilled to operate something automatically make them an integral part of a system.

Airplanes are also examples of systems where human (pilots) are an integral part of the system – and by extension reliability. Air France 447 was a flight where an Airbus A330 plunged into the Atlantic Ocean, killing all 228 occupants. One of the issues identified as a contributing factor was the sidestick or joystick controls in the cockpit (as illustrated below).



These controls are fundamentally different to the highly visible and connected control yokes of other airplanes.



The Air France 447 copilot for whatever reason pulled 'back' on the side stick, pushing the nose of the aircraft up. This placed it in a stall condition and it started plummeting toward the ocean. The other pilot was trying to push the nose down with his side stick – and he couldn't understand why the plane wasn't responding. By the time the crew realized what was happening – it was too late.

There is no point designing something that is 'very reliable' if it is difficult to operate in a reliable way.

