Building Fault Tolerant Systems

One particularly important problem in Robotics is that of the design and implementation of systems that have to perform an important, dangerous, time-sensitive, or otherwise critical task. Such systems can not be allowed to ***fail*** or ***become unable to carry out their task*** because of common issues such as component failures or software bugs.

The design and implementation of systems that can continue to perform their task in the presence of ***faults*** is all the more relevant in the face of increasing reliance in automation, and A.I. powered systems in all areas of daily life. Therefore we should become familiar with the main principles involved so we can apply them consistently and wherever appropriate to any robotic or automated systems we will be working with.

Common applications of fault-tolerant systems

These are only a few of the most common areas of application for fault-tolerant systems. You can imagine that for each of these, the failure of a system to perform their function correctly would cause significant harm to humans.

- ***Transportation***: From airplane design and construction, to the software that powers cars with varying degrees of automation, to delivery drones.

- ***Medical applications:*** Medical scanning technology, robot-assisted or remote surgery, systems that monitor the young or the elderly.

- ***Energy production and distribution:*** Consider the systems that monitor and control critical facilities such as nuclear power plants or electric grid load balancing systems.

- ***Telecommunications and on-line infrastructure:*** Nowadays, a large portion of our technology relies on systems that are hosted on-line, or that, in order to function, require a remote connection with servers that support their function. Failures in cloud infrastructure can have an immediate and extensive effect in our daily lives.

- ***Banking and financial systems:*** A significant portion of our economy is based on electronic transactions, similar to failures with telecommunications or online platforms, failures of these systems can easily have a significant economic impact and cause a major disruption to customers.

The above are only some of the major areas in which we find need for systems that will perform correctly and continuously. Let’s think now about how these systems are built.

Fault tolerance is usually specified in terms of one, or a combination of, common measures of how resilient and robust a system is. Common measures used in the study and design of fault tolerant systems include:

Reliability: This is a measure of the ***probability that the system will perform its function correctly over a specified interval of time***. In particular, the reliability ***R(t)*** for a particular system is the ***conditional probability*** that the system will perform its task correctly in the interval $[t_0, t]$, where t_0 is the initial time and ***we assume the system was performing correctly*** at this time.

This measure is often used whenever we have a system for which even short periods of incorrect behaviour are unacceptable (think, for example, of an airplane's flight control system), or for which repairs are impossible (e.g. planetary exploration robots). Such systems have to function with very high probability throughout the entire duration of their ***mission***. For an airplane's flight control computers, for example, this could be the duration of a flight, so for instance, a design requirement for a commercial airliner's flight computers system may be that it have a reliability ***R(t)=.999999*** over an interval of 6 hours.

Availability: The availability of a system ***A(t)*** is defined as the probability that a system is performing correctly and is available to carry out its function at time ***t***. It is not the same as ***reliability*** because it involves only the specific time instant. This is a measure commonly used for on-line platforms – which we expect to be ***available*** pretty much continuously.

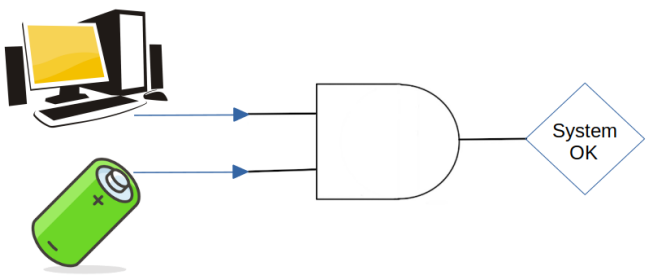
Notice that ***availability*** does not necessarily provide much information regarding ***how often*** the system is ***unavailable*** or ***for how long***. A system may have frequent but short periods of unavailability and yet have an overall high availability value. Similarly, a system might suffer from a longer continuous period of unavailability, and if this happens rarely, its availability figure can still be high.

Other measures exist and can be used depending on the application, see [1] for details on some of these.

Measuring reliability

How are we to estimate the reliability of a system? In practice this is a fairly complicated problem, most of the systems we may have to work with consist of a multiplicity of components, both hardware and software, interact with external entities, receive information that is often noisy and contradictory, and are subject to all kinds of disturbances.

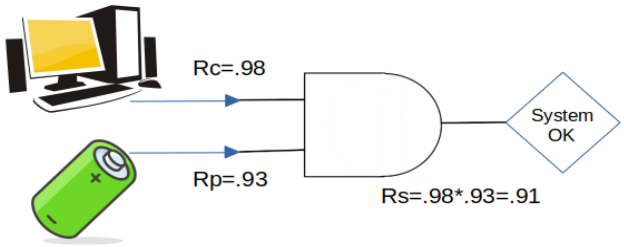
However, we can often look at sets of components, and consider what configurations of these components would result in the most reliable system. This is done through what is called ***success trees.*** As an example, consider the simple system shown below:



The very simple system above contains 2 components, a computer module and a power source. For the system to operate, ***both of these components*** have to be working at any given time. This is represented by the ***AND*** gate joining them to the status of 'System OK'. We can use this very simple diagram to model the reliability of the system above if we have information about the reliabilities of each of the components.

For instance, if ***Rc=.98*** is the reliability of the ***computer module*** (and let's assume here the figure is over a period of one year, since reliability is measured in terms of specific time intervals), and if ***Rp=.93*** is the reliability of the ***power source***.

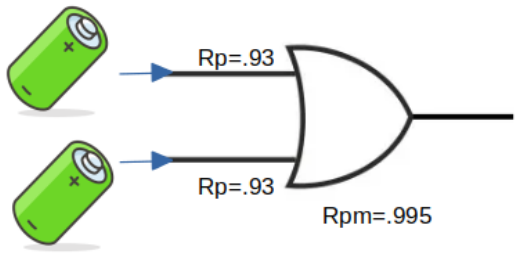
Then we have:



Notice that the reliability of the system is actually lower than that of the individual components. This is what happens when we have multiple parts all of which have to work properly for the system to function. Any fault in the parts will bring the complete system down. The more parts that are required for the system to operate, the higher the likelihood any of of them will fail within some arbitrary time, and render the system inoperable.

A collection of parts is thus less reliable than the individual parts alone, and high reliability requires individual parts with very high reliability themselves.

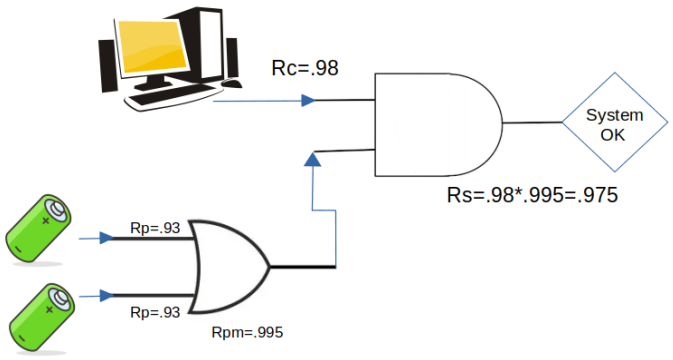
Now consider the following setup for a ***highly reliable power source:***



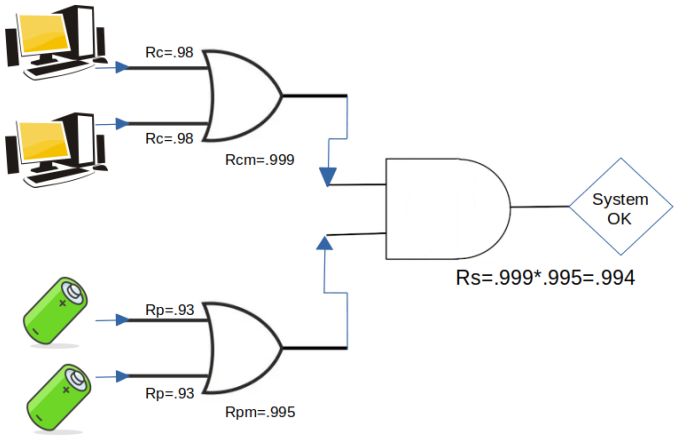
This highly reliable power source uses ***redundancy*** to achieve increased reliability. Two identical batteries are combined into a single power source, ***as long as either of them is operational***, the power source will be operational – this is represented by the ***OR*** gate. To compute the reliability of the power source we note that in order for it to fail, ***both batteries must fail at the same time***. The probability that one of the batteries fail is (1-Rp) = (1-.93). The probability that ***both*** fail is $(1-.93)^2$, and therefore the

probability that the power source remains operational is $1-(1-.93)^2=.995$. Notice that this is significantly higher than the reliability of either battery alone.

With a ***redundant power source***, the reliability of our original system becomes



The use of a redundant power source greatly increases the reliability of the entire system. We could similarly implement a redundant computing module and see how that changes the reliability of the system:



The system above has a fairly high reliability – definitely much higher than our original system with only two components. The cost of course is that now we have to include two computers and two batteries (plus some hardware connecting everything together). Redundancy is limited often by cost, and sometimes also by physical constraints (for instance, the extra weight or space taken by the additional hardware).

The representation shown above is an example of a ***success tree,*** which allows us to inspect and analyze the reliability of components, modules, and entire systems or sub-systems.

We can use ***success trees*** to ask and answer questions regarding which configuration is optimal when choosing among

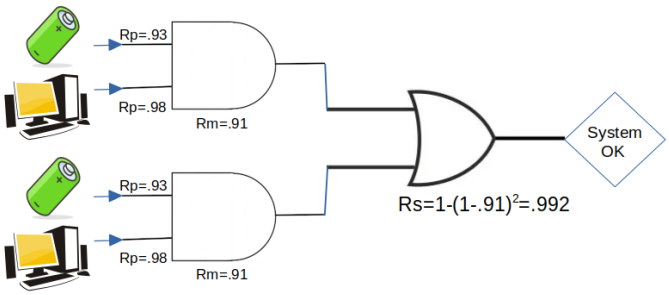
a) Different configurations of the same base components or modules

or

b) Configurations that use different components constrained y a fixed budget

For instance, assume we are given 2 computers and 2 batteries with the reliability values shown above – what is the optimal configuration for them? Is it a redundant battery module combined with a redundant computing module (as shown above), or is it better to have a pair of computer+battery modules?

Let's study the alternate configuration and see what its realibility would be:



which is not as good as having a redundant computing module and a redundant battery module, so let's assume we decide to implement that particular system – now we can consider the choice of specific components to use in building the system. Consider the redundant power module, and suppose that given our budget, we can afford to use ***two expensive batteries with Rp=.93*** as shown in the diagrams above, ***or***, we could instead use ***three less expensive batteries with Rp=.89***. Which configuration would yield the higher reliability for the power module?

We can quickly see that for the same budget, having a larger number of cheaper, less reliable (individually) components provides a power module with higher overall realiability.

This analysis becomes fairly involved for systems consisting of many different components interacting in multiple and complicated ways, but there are software tools to help model and analyze reliability.

A similar analysis can be made for ***faults –*** that is, conditions that may occur that will cause the system to fail. For each of the modules and/or components of a system, we can model the different possible causes of failure using ***AND*** and ***OR*** gates, joining modules and components together as we

did for the reliability modeling above, and eventually come up with a ***failure tree*** for the system, indicating for each module/component, and for the overall system, the probability of failure over some specified interval of time.

As an example, consider the fault tree (shown below, in a separate page) for ***Pressure Tank Rupture*** (courtesy of NASA, see [2] for details). As you can see, the tree can become quite complex and this tree models only one type of possible failure. Similar trees would be used to characterize other failure modes for the system we are studying, and together with the failure probabilities, it can be used to model, estimate, and even simulate the probability and occurrence of different types of fault modes.

Achieving Fault Tolerance Through Redundancy

Fault tolerance is not the same as reliability. Reliability is just an estimate of how likely it is that a system will perform its work correctly over an interval of time, ***fault tolerance*** is a property of a system that allows it to handle ***faults –*** a fault is an imperfection, a deviation from what is expected, a problem or flaw, an unexpected condition in the operation of a system. It can be caused by hardware (e.g. a component failure, incorrect or unexpected behaviour, etc.), or by software (typically a bug, or the result of erroneous information being processed).

Fault tolerance is the property of a system that allows it to handle a certain number of flaws while still fulfilling its correct function. Fault tolerance and reliability are complementary to each other – if we could build a system whose every component is 100% reliable, then we would not need fault-tolerance. This is impossible, so instead we put a lot of effort into ensuring a critical system won't be disabled by possible faults, and in doing so, will increase the overall reliability of the system beyond what can be expected just from its components.

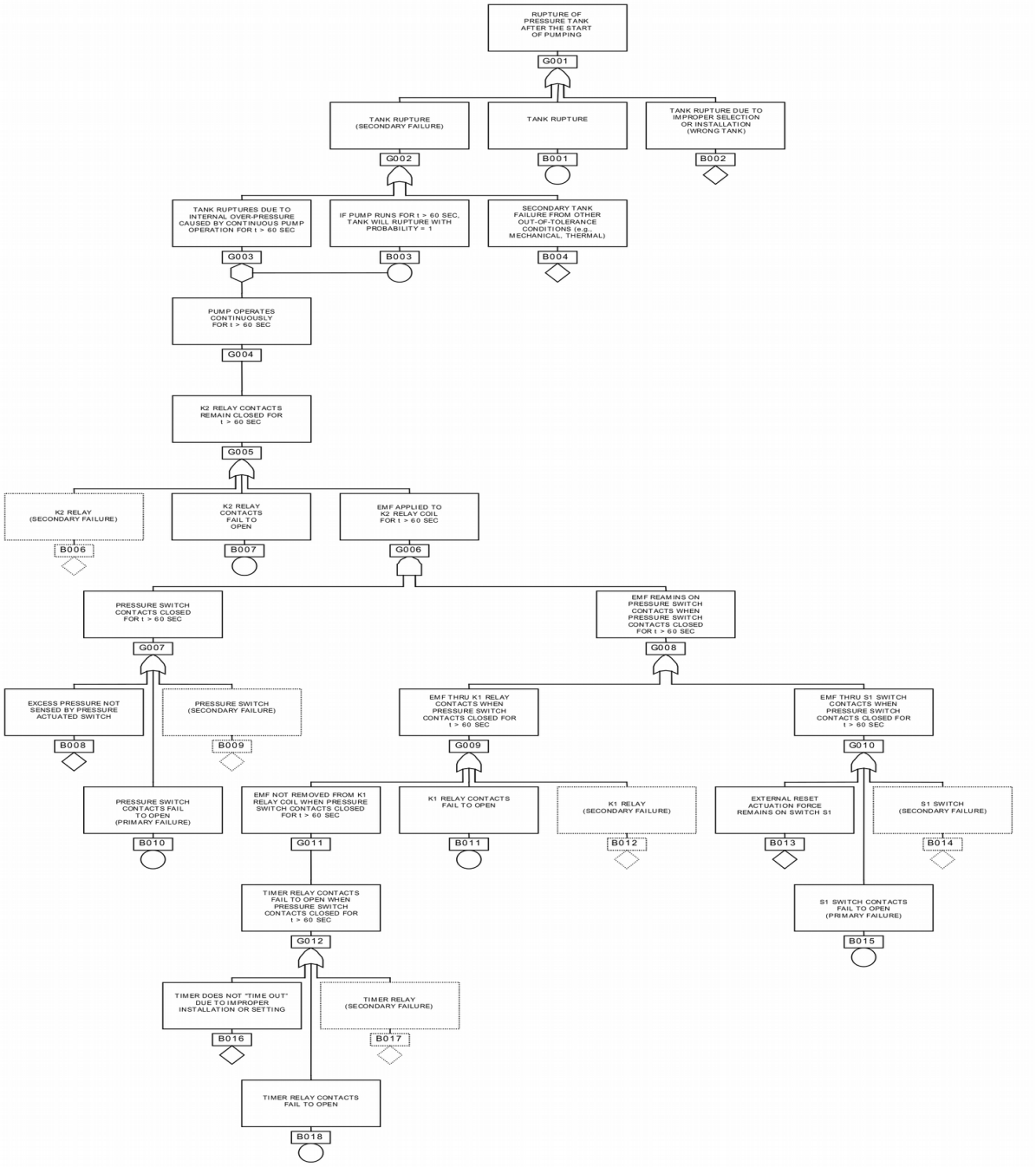
Fault tolerance needs to consider: ***the hardware*** the system consists of, ***the software*** that we are using to operate and run the system, and ***any information or data*** that the system uses and/or produces during its operation.

Hardware Fault Tolerance

Hardware fault tolerance often is equivalent to ***redundancy***. As we saw above, building modules that contain ***replicated components*** each of which can ***independently provide*** some required functionality can greatly increase reliability.

One common approach, which is often taken as the *gold standard* for building fault-tolerant modules is called ***triple modular redundancy***. This approach is a ***passive*** approach to redundancy, and relies on modules that contain ***three independent components*** each of which is separately able to provide the desired function (for instance, the redundant power module we saw above, with 3 identical batteries).

The ***TMR*** approach includes an additional step of ***voting***. To understand what this means, consider a common situation in which triple modular redundancy is used to provide reliability and fault-tolerance: Designing and implementing reliable sensor modules for devices such as airplanes or cars.

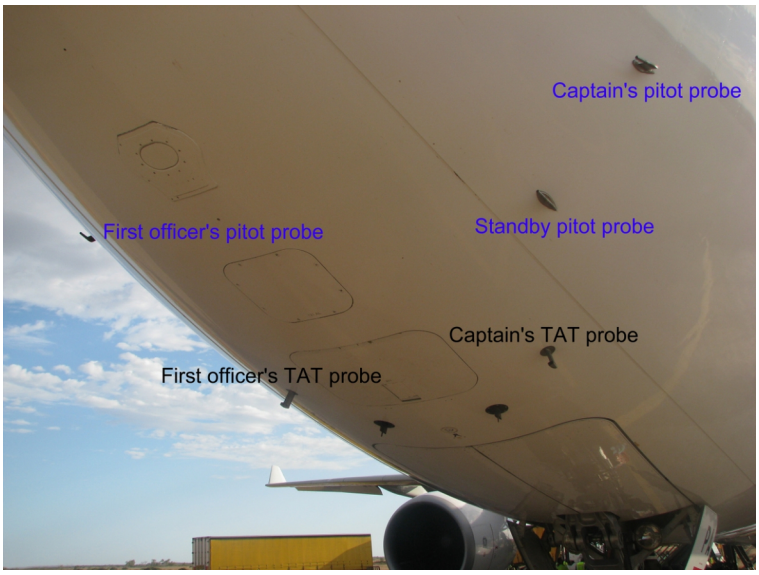


Sensors are an essential component of transportation systems. Airplanes in particular require a wide variety of measurements to determine whether everything is working as it should, what the parameters of flight are, and to maintain the aircraft's integrity and safety despite the constantly changing external forces as well as pilot inputs.

But we know that sensors are noisy, and we know that sensors fail and can give widely erroneous readings either spuriously or continuously. It should not be difficult to see that having a single sensor in

charge of providing some critical piece of information required for the plane to operate safely would be a bad design choice and would provide a ***single point of failure*** that could endanger the plane should the sensor fail.

For this reason, critical sensors in airplanes are typically tripled:



(Image shows the triplicate pitot tube sensor arrangement in an A380 – pitot tubes provide airspeed measurements, critical for safe flight. Image courtesy of the Australian Transportation Safety Board ATSB)

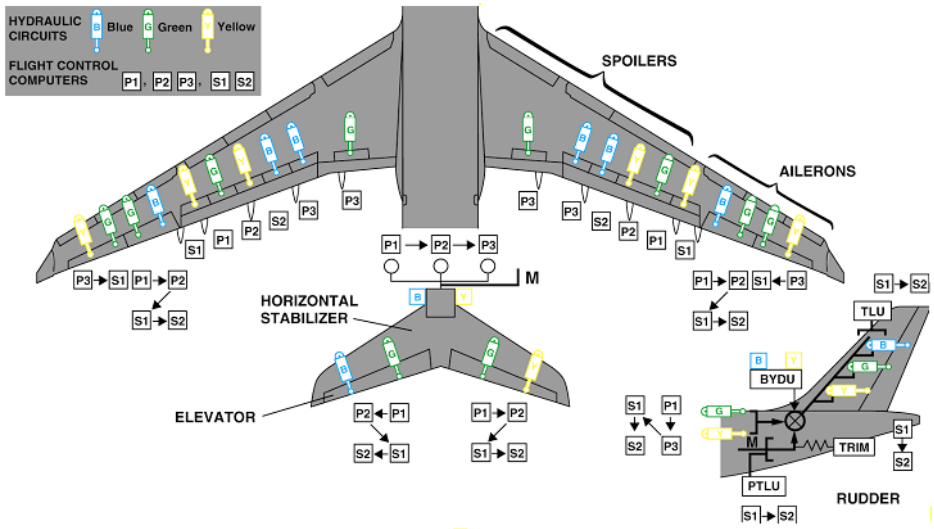
Getting back to the TMR design, the ***voting*** component has the function of determining ***which measurement to use*** among the three readings taken by the sensors. This is not a completely straightforward process because:

- Sensor readings will be different sue to ***noise***
- Sensor readings will be different because the sensors are ***physically separated*** so they are reading information from different locations
- Sensors can not be read ***at exactly the same time***, so the readouts will also differ because of the time-varying characteristics of whatever is being measured

Overall, the ***voting*** module has the task of taking the separate readings from the three sensors, and outputting a ***consensus*** value – this can be a ***majority vote*** (if the sensor is measuring something that can be quantized into a small number of discrete values), ir can be a ***median value*** (which is common in the airline industry, just take the middle value amongst the three being provided), or it can be a more complicated function of sets of rules defined for how the data should be used.

Regardless of how it is used, the TMR module allows for the ***detection of faulty sensors*** – it is reasonably straightforward to determine if one sensor is registering readings that are significantly different from the other two. It can also determine whether the whole suite of sensors is faulty, which happens when all three sensor measurements are significantly different.

TMR can be applied to many kinds of hardware components, and can also be applied to complete sub-systems, not just components. For instance, the flight control systems for all current airliners involve triple redundancy in the the components that power, actuate, and effect changes in the attitude of the plane (i.e. the systems that actually control how the plane is flying):



s(Airbus redundant flight control schematic. Notice three separate (color coded) control paths with their own hydraulic and electronic components, as well as multiple flight computers. Image reproduced from [3])

As with all redundancy-based systems, the factors that place limits on what we can replicate, and to what degree we can replicate components and sub-systems are ***cost***, and ***physical limitations such as weight or space availability.*** These have to be balanced against the safety implications of ***not replicating*** a component.

Case study: fault-tolerance done wrong

The by now well known case of the Boeing 737-MAX illustrates how failure to build proper fault-tolerance into a critical system can have disastrous consequences. The airplane's design includes ***two Angle of Attack (AOA) sensors***. This in itself is common in the airline industry, and not a design issue though it should be clear that if either sensor fails, ***we can detect the failure, but can't tell which of them is faulty***.

The difference between the MAX and other airliners with two AOA sensors is that the MAX was provided with a software system programmed to use AOA sensor information to push the nose of the plane down if the sensor indicated the plane was in danger of stalling (to be fair, this was just one of many things the software system was designed for). Crucially – ***this system relied on input from a single sensor***, ***did not check whether the alternate sensor disagreed (thereby indicating a potential sensor failure), and did not check other flight parameters, sensors, or sources of information*** before deciding to take action. To cap this off, pilots were not properly informed regarding the presence or capabilities of the system, and were not provided clear and effective training on how to use and if necessary disable this system to regain control of their plain in case of system failure.

The result was a ***single point of failure*** should the AOA sensor give the wrong readings. Which it did with disastrous consequences. You can find documentaries and reports describing in detail how this came to be. What's important for us here is to realize that, from the very simple description above, it should be clear that this system could not be fault-tolerant, and therefore should never have been allowed onto a production plane. Any decision taken thereafter, by anyone who was involved with any aspect putting this system into a production plane was the wrong decision.

Software Fault Tolerance

Suppose we have designed and built a fault-tolerant computing system consisting of triple-modular-redundant computing resources, and therefore can be reasonably certain the hardware is solid. However the hardware is not particularly useful unless the software running the system is also reliable and fault-tolerant.

What particular considerations must we keep in mind when designing fault-tolerant software?

- Fault-tolerant software ***must be able to detect and compensate for faults***. Both in hardware components as well as within the software itself
- Fault-tolerant software ***must be as free of bugs as is humanly possible to achieve***
- Fault-tolerant software must be able to handle ***erroneous input*** in a way that does not cause an unsafe situation to arise

We can not achieve software fault tolerance simply by replicating the same software to each computer within a TMR computing module. Any bug present in the software will affect all redundant copies and result in incorrect behaviour that can not be compensated for.

The design and implementation of fault-tolerant software includes

1) A thorough, careful, and detailed process for acquiring, analyzing, and detailing ***specifications*** for what the system must be able to do. This process will also list and categorize possible failure modes, their likely causes, and their severity – using, for example, fault trees as discussed earlier.

2) Implementation of the system using ***N-version programming***. This means having N separate teams implement the system software ***independently*** based on the completed specification. Much like having redundant hardware and a voting mechanism, we can then run these N versions of the software on reliable computing platforms and ***vote*** on the results to determine what the system must do next.

Each of the separate versions must undergo its very own ***rigurous testing process*** to ensure compliance with the specification, and to detect and eliminate as many software problems and bugs as is possible before any of the software goes into operation.

The purpose of having independently designed versions of the software is to greatly reduce the chance that any specific bug will affect all our copies of the software at any given time. We should note

that ***it is still possible for the N versions to contain the same issue, if the issue arises from an incomplete or incorrect specification***.

N-version programming is costly – we are basically replicating the time, effort, and cost required to develop a piece of complex software N times.

However, the end result, combined is a software system that is incredibly reliable.

3) ***Consistency checking***: This means implementing sof tware logic that actively and continuously checks input data, sensor readings, intermediate results, and outputs for the system for consistency – i.e. the system must produce reasonable results for every particular situation. This may involve using a model for the system to ***predict*** what the state of the system will be and to ***check*** that the system's results agree to a reasonable degree with these predictions.

Sources of information for consistency checking:

- A model for the system, which is a common building block in dynamical systems. The model can be used to make predictions about the immediate future state of the system given the current state and any inputs being provided to the system. This can be used to check any sensor measurements, or to predict the effect actions that the automated systems are considering

- Alternate sensors: Losing a sensor for a specific parameter does not necessarily mean we have zero information regarding the parameter with the failed sensor. Often, the system will have additional sensors whose measurements are *related* to the value we are missing in a way that allows us to estimate the missing value to a reasonably good level of accuracy. The simplest example is the set of sensors that measure acceleration and velocity in a moving vehicle. A missing velocity value can be computed from the acceleration measurements, and vice-versa. *Estimating missing parameters does not allow a system to operate safely for a long interval* – the estimated values drift away from the correct ones, and will sooner or later become too inaccurate to use. However, we can gain enough information in the interval immediately following a sensor failure so as to enable the system to *safely stop.*

- *Human in the loop:* This seems to have been overlooked as of late, with emphasis being placed on automation. But many safety critical systems include a human operator as a central part of the operation. A qualified human, presented with sufficient information regarding an existing problem, can quickly and accurately determine if a measurement, or an action being considered makes sense or not.

Any or all of the above should be used to the greatest possible extent to ensure that no decisions or actions are being taken with possibly inconsistent information, and without checking that the resulting actions are consistent with the safe operation of the system.

In the case of the 737-MAX, the software ***failed to check for consistency*** between the sensor reading it was using and the alternate sensor, or between this reading and and the previous values of flight parameters. It failed to account for possible ways in which the AOA reading may change over

time (it did not attempt to check whether the observed change in AOA was physically possible using a *model* of how the plane behaves), did not check for consistency between the AOA readings and all other on-board sensors (including inertial navigation units which provide acceleration and rotation measurements), and did not check that its output – namely commanding the plane's nose down) resulted in a safe configuration for the flight (it should never have produced a configuration in which the altitude of the plane is quickly decreasing) – it also, ***importantly*** disregarded continued and consistent inputs by the pilots trying to push the nose of the plane back up (a clear and unequivocal signal that what the automation was doing was wrong!).

While it is clear that the MCAS system in the 737-MAX did not go through a proper process of specification, design, testing, and validation, the absence of any form of consistency checking in this particular function is a glaring failure to implement basic, well known principles of fault-tolerant design for critical systems.

Consistency checking is difficult and requires considerable thought, adds complexity to the software that runs the system, and therefore adds a significant cost in terms of development, testing, and validation. But done correctly will make a difference between *dumb* decisions being taken where sufficient information was available to avoid them, and *smart* decisions being taken by software that is able to determine that what it is doing makes sense and that it will result in the safe and correct operation of the system.

4) Recovery blocks: Sometimes, instead of a voting process to select outputs based on what is being produced by the N versions of the system's software, we instead implement what is called a ***recovery block***. In a recovery block, there is an ***acceptance test*** block that checks the output of the software. The system uses the 1st version of the software as long as it passes the acceptance test. If the acceptance test fails, then the system moves on to using the 2nd version of the software – it will keep using that version as long as it passes the acceptance test. If the 2nd version fails the test, the system moves on to version 3, and so on.

Voting-out Reconfiguration

For systems that rely on TMR hardware redundancy as well as muti-version software redundancy, a commonly used technique to ***detect and remove from usage faulty modules*** is what is called ***voting-out reconfiguration***.

The principle is fairly simple: A set of redundant hardware/software modules are continuously run and a voting module continuously compares their outputs and determines which of these is used to run the system.

In addition to this, whenever one of the redundant modules is found to produce output that is significantly different from the rest, it gets ***voted out*** and is taken offline. This is slightly different from keeping the faulty module on-line because as long as it remains on-line, it will contribute its own vote and could lead (in the presence of additional faults in other modules) to the wrong output being picked.

As a simple example, given 5 redundant modules and without voting-out reconfiguration, 3 modules failing in sequence over time may cause the wrong output to be selected. With voting-out reconfiguration, we need to have 4 failed modules to get to the point where we don't know what to do (but we still know there is a fault and we don't have information regarding which output to choose among the last remaining modules).

Information Fault Tolerance

Even if we have carefully designed, built, and programmed a fault-tolerant computing system, it is still possible for the entire system to fail if the ***data it is working with is corrupted or erroneous***. This includes not only the actual information being processed, but also the software itself – which is subject to corruption just the same as any other piece of digital information.

It is therefore important that we consider the problem of ***information redundancy*** or, more aptly, ***how to make the information our system works with fault tolerant***.

Data corruption can occur because of multiple factors:

- Data transmission errors, including bit flips, dropouts, or malicious interference
- Corruption because of noise and electromagnetic interference in circuits
- Corruption due to aging or failure of storage media (e.g. degradation of 'permanent' storage)

This topic goes well beyond fault tolerant systems, and is a big area of research and application in distributed systems, cloud computing, archival technologies, and even your own personal computing devices. The issue is not so much that the probability of an error occurring is high – current memory technology is pretty reliable, but given the massive amounts of information stored the occurrence of errors is a certainty.

Here, we will focus on two problems that need to be carefully considered when building information fault tolerance into a system:

1) Error detection – which allows us to determine a ***fault*** has occurred and we need to do something about it, and is analogous to software fault detection, and hardware fault detection.

2) Data replication/redundancy – which allows the system to continue operating until it is safe to stop, even if there are faults in the data storage/retrieval components we are using.

Error detection involves encoding information in such a way that we can determine an error has occurred. There are several techniques for doing this, from the simplest such as ***parity checking*** and ***checksums*** to more advanced (and costly in terms of space and computation) ***error correcting codes***. The choice of which of these to implement depends on the type of system being used and the severity of potential errors for the operation of the system. But in general, a fault tolerant system needs to be protected against data corruption and errors from any of the sources described above.

Data corruption was the root cause of an incident involving a Qantas flight (Flight 72) from Singapore to Perth. The plane suddenly dived for no apparent reason during the cruise phase of the flight causing injuries to passengers and crew. The cause of the incident was erroneous AOA data being provided to the flight control computers (a similar but much less catastrophic problem to what was seen in the 737 MAX above). The erroneous data was the result of data corruption in the CPU of the units in charge of estimating flight parameters.

The plane had built-in software safeties to handle unexpected variations in flight parameters (so it did not simply act on the erroneous readings as they arrived, unlike the MAX), but the nature of the failure interacted with these safeties in such a way that eventually the flight control computers determined the data was real and proceeded to act on it.

One should again question the degree to which ***consistency checking*** should have been used in this case, but the root cause – data corruption, motivates the need for the use of ***error detection*** and/or ***error correction*** codes in all the data used by a safety-critical system.

Data duplication – in terms of data storage, the need for reliable backup systems should be evident if we wish to achieve fault tolerance. All storage media are subject to aging and data degradation, mechanical failure, or damage because of environmental factors. Failure of data storage can result in catastrophic failure.

Data duplication is a commonly used technique, and many different schemes exist that can be easily built into a system. As usual the main limitations in this regard are the additional cost, and the physical space requirements of the data duplication system.

Data replication introduces issues similar to those encountered with redundant sensor modules or with redundant computing systems: Given a set of copies of a particular piece of information, we can check for errors by comparing the value of the data across the multiple copies, and some mechanism must be in place to determine which copy to use in the case of discrepancies. Voting, recovery blocks, and/or voting-out reconfiguration may be used depending on the needs of the system.

One more technique that is often used in achieving information fault tolerance is ***checkpointing***. This technique involves the creation of periodic ***checkpoints*** with the property that the information stored at these checkpoints corresponds to a point in time at which data, outputs, and the state of the system were known to be ***correct***. If at some point a ***fault*** is detected, and we estimate there is a probability that erroneous results or information have been produced, we can ***roll-back*** to the nearest ***checkpoint*** and then ***reconstruct*** the correct sequence of processing having accounted for the detected fault.

Checkpoints are commonly used in operating systems to protect against unforeseen problems with software and system updates – if something goes wrong, the system can simply go back to its previous state hopefully without any loss of important data.

References:

[1] Johnson, Barry, “An Introduction to the Design and Analysis of Fault-Tolerant Systems” - available online from ResearchGate.

[2] Fault Tree Handbook with Aerospace Applications, NASA Office of Safety and Mission Assurance, 2002

[3] Sghairi, et.al., “Architecture Optimization Based on Incremental Approach for Airplane Digital Distributed Flight Control System”, Advances in Electrical and Electronic Engineering.