

FIT3142 Distributed Computing

Topic 10: Reliability of Distributed Applications

Dr Carlo Kopp

Based on “*System Reliability and Metrics of Reliability*”

© 1996 PHA Pty Ltd, by Carlo Kopp, used with permission.

Faculty of Information Technology

Monash University

© 2009-2016 Monash University

Why Study Grid, Web Services and Cluster Reliability?

- All distributed applications, whether running as part of a cluster, grid or other web services framework, are highly sensitive to the reliability of the underlying network and host hardware, host operating systems, and libraries and toolsets used to implement the distributed application.
- Because tens, hundreds or thousands of hosts and networking devices might be involved, reliability becomes a “mission critical” consideration in design of the application.
- Poor reliability in a shared service or critical hardware component can have a serious impact.



What is Reliability?

- **Probability of System ‘Survival’ $P[S](t)$ over time T.**
- $P[S](t) = R(t) = 1 - Q(t)$
- **A measure of the likelihood of no fault occurring.**
- **Related to system function and architecture.**
- **‘All systems will fail, the only issue is when, and how frequently.’**

System Reliability

- **Hardware Reliability.**
- **Software Reliability.**
- **Reliability of interaction between hardware and software.**
- **Reliability of interaction between the system and the operator.**

Hardware Reliability

- Electronic component, printed circuit board, interconnection reliability, and related failure modes.
- Hard, transient & intermittent failures.
- Random failures - exponentially distributed.

$$R(t) = \exp(-\lambda t)$$

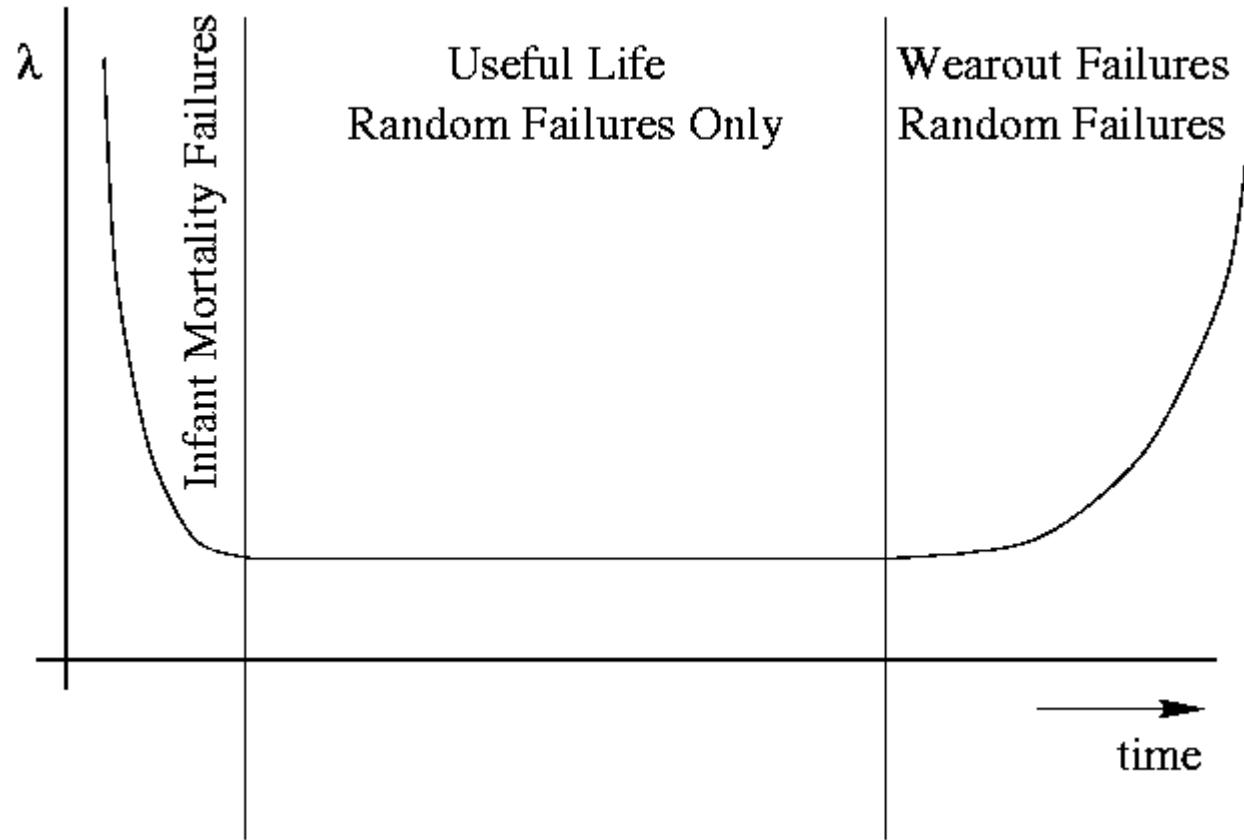
- Wearout failures - normally distributed

$$R_{wearout}(t) = \frac{1}{\sigma\sqrt{2\pi}} \int \exp\left(-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2\right) dt$$

- Infant Mortality



Hardware “Lifecycle” - Bathtub Curve Diagram



Measures of Hardware Reliability

- **MTBF = Mean Time Between Failures – easy to measure across large ensemble of components.**

$$MTBF = \frac{1}{\lambda} \quad \lambda = \frac{1}{MTBF}$$

- **MTTR = Mean Time To Repair – easy to measure across large ensemble of components.**
- **Temperature dependency of lambda - failure rates always increase at high operating temperatures.**
- **Voltage dependency of lambda - failure rates always increase at higher electrical stress levels.**
- **High stress - high lambda !**



Lusser's Product Law

- Discovered during A4/V2 missile testing in WW2
- Superseded dysfunctional ‘weak link’ model
- Describes behaviour of complex series systems.
- Theoretical basis of US Standards/methods defined in Mil-Hdbk-217 and Mil-Std-756

$$R_s = \prod_{i=1}^N R_i$$

- The survival of the series system depends on the survival of every element in the system.

Lusser's Product Law - History



The German A4 was one of the most complex and high stressed items of technology in existence during 1940s period. It forced the development of modern mathematical reliability theory.



Serial Systems Reliability

- Failure of single element takes out system.
- Use Lusser to quantify total λ and $P[S]$ for some given time interval T .

$$R_s = \prod_{i=1}^N R_i = \exp\left(-\sum_{i=1}^N \lambda_i t\right)$$

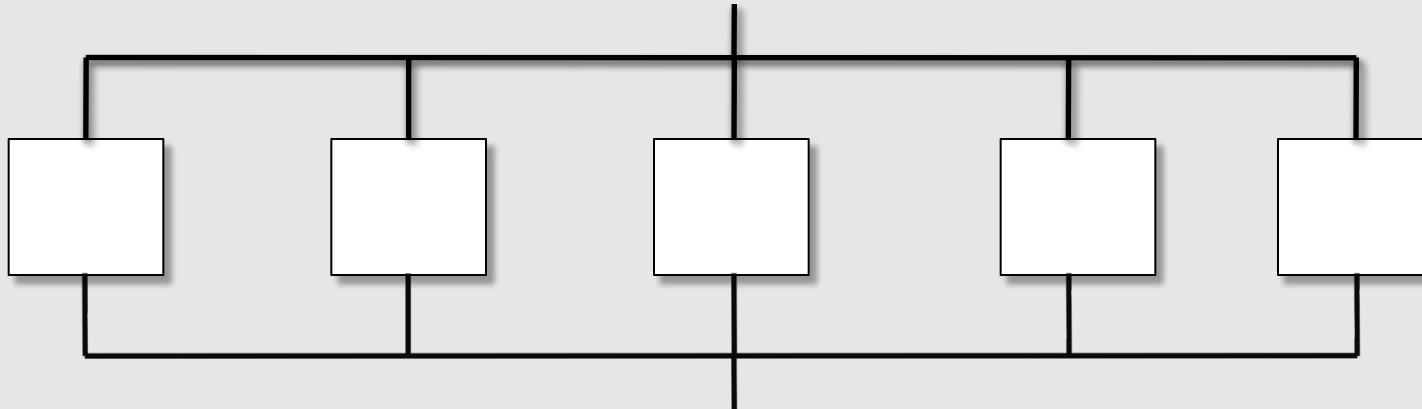


Parallel Systems Reliability

- Failure of single element is survivable, but $P[S]$ reduced.

$$R_p = 1 - Q^N$$

- Used in aircraft flight control systems, Space Shuttle and critical control applications.



Complex Systems

- **Combine parallel and serial models.**
- **Required detailed analysis to determine $R(t)$**
- **Must analyse for dependencies.**
- **Must avoid Single Point of Failure (SPoF) items.**
- **The higher the complexity of the system, the higher the component reliability needed to achieve any given MTBF.**



Example RAID Array (1999)

- **N x 1 array with single fan and PSU**
- **Drive redundancy is OK, PSU or fan failures are SPoF.**
- **Problem fixed with redundant fans and PSU.**
- **No SPoF items - significantly improved reliability.**

Example P-38 Twin Engine Fighter (1944)

- Electrical propeller pitch control, radiator and intercooler doors, dive flap actuators, turbocharger controls.
- Twin engine aircraft, only one generator on one of the engines.
- Loss of generator equipped engine - feather propeller, fail over to battery.
- Once battery flat, prop unfeathers, windmills, turbo runaway -> aircraft crashes.
- Problem fixed with dual generators, one per engine.
- Significant loss of pilot lives until problem solved.



79th F S/20th F G Arthur Heiden May 1944



Software vs Hardware Reliability

- **Hardware failures can induce software failures.**
- **Software failures can induce hardware failures.**
- **Often difficult to separate H/W and S/W failures.**
- **Cannot apply physical models to software failures.**
- **Result is system failure.**



Cascade Failures

- Cascade failures have become an area of research, following a number of such events arising in electricity grids;
- A cascade failure can occur in any system, in which there are a large number of interdependent nodes or subsystems;
- In a cascade failure event, loss of a node causes workload to be migrated to other nodes, if this workload is sufficient to saturate other nodes, these may also fail, resulting in a “cascade failure” as loss of nodes causes other nodes to fail;
- The well documented “congestion collapse” failures seen during early use of the Internet qualify as cascade failures;
- Any distributed application with automatic failover mechanisms is a candidate system for a cascade failure;
- *Therefore, care must be taken with design to ensure this cannot happen – automation can produce unwanted effects!*



Modes of Software Failure

- **Transient Failure** - incorrect result, program continues to run.
- **Hard Failure** - program crashes (stack overrun, heap overrun, broken thread).
- **Cascaded Failure** - program crash takes down other programs.
- **Catastrophic Failure** - program crash takes down OS or system -> total failure.

Types of Software Failure

- Numerical Failure - bad result calculated.
- Propagated Numerical Failure - bad result used in other calculations.
- Control Flow Failure - control flow of thread is diverted.
- Propagated Control Flow Failure - bad control flow propagates through code.
- Addressing Failure - bad pointer or array index.
- Synchronisation Failure - two pieces of code misunderstand each other's state.



Runtime Detection of Software Failures

- **Consistency checks on values.**
- **Watchdog timers.**
- **Bounds checking.**



Consistency Checking

- Can identify a bad computational result.
- Exploit characteristics of data to identify problems.
- Protect data structures with checksums.
- Parallel dissimilar computations for result comparison.
- Recovery strategy required.



Watchdog Timers

- **Require hardware support to interrupt tasks or processes.**
- **Watchdog timer periodically causes status check routine to be called.**
- **Status check routine verifies that code is doing what it should.**
- **Can protect against runaway control flow.**
- **Recovery strategy required.**



Bounds Checking

- Compare results of computation with known bounds to identify bad results.
- Requires *apriori* knowledge of bounds upon results.
- Cannot protect against bad results which have ‘reasonable’ values.
- Recovery strategy required.



Recovery Strategies

- **Redundant data structures - overwrite bad data with clean data.**
- **Signal operator or log problem cause and then die.**
- **Hot Start - restart from known position, do not reinitialise data structures.**
- **Cold Start - reinitialise data structures and restart, or reboot.**
- **Failover to Standby System in redundant scheme (eg flight controls).**

Case Studies

- **Why Case Studies - explore how and why failures arise.**
- **Define the nature of the failure.**
- **Describe the result of the failure.**
- **Look at possible consequences of the failure.**
- **Try not to repeat other peoples' blunders.**



Prototype Fighter Testing #1

- **Test pilot selects wheels up while on the ground.**
- **Aircraft retracts undercarriage and falls to the ground.**
- **Immediate cause: software failed to scan the ‘undercarriage squat switch’.**
- **Reason: *programmer did not know what a squat switch was for.***
- **Possible consequences: *destroyed jet, dead pilot.***



Prototype Fighter Testing #2

- Radar altimeter and barometric altimeter fail.
- Pilot notices altitude reading at 10,000 ft, yet when he looks outside, the aircraft is flying at very different altitude.
- Immediate cause: software default action on altimeter fail is ‘set constant altitude value’.
- Reason: *programmer did not understand how aircraft fly.*
- Possible consequences: *destroyed jet, dead pilot.*



Prototype Fighter Testing #3

- Aircraft crossed equator on autopilot.
- Aircraft attempts to roll itself inverted (upside down).
- Immediate cause: navigation software failed to interpret sign change.
- Reason: *unknown, programmer may have assumed operation only North of equator.*
- Possible consequences: *midair collision, destroyed jets, dead pilots.*

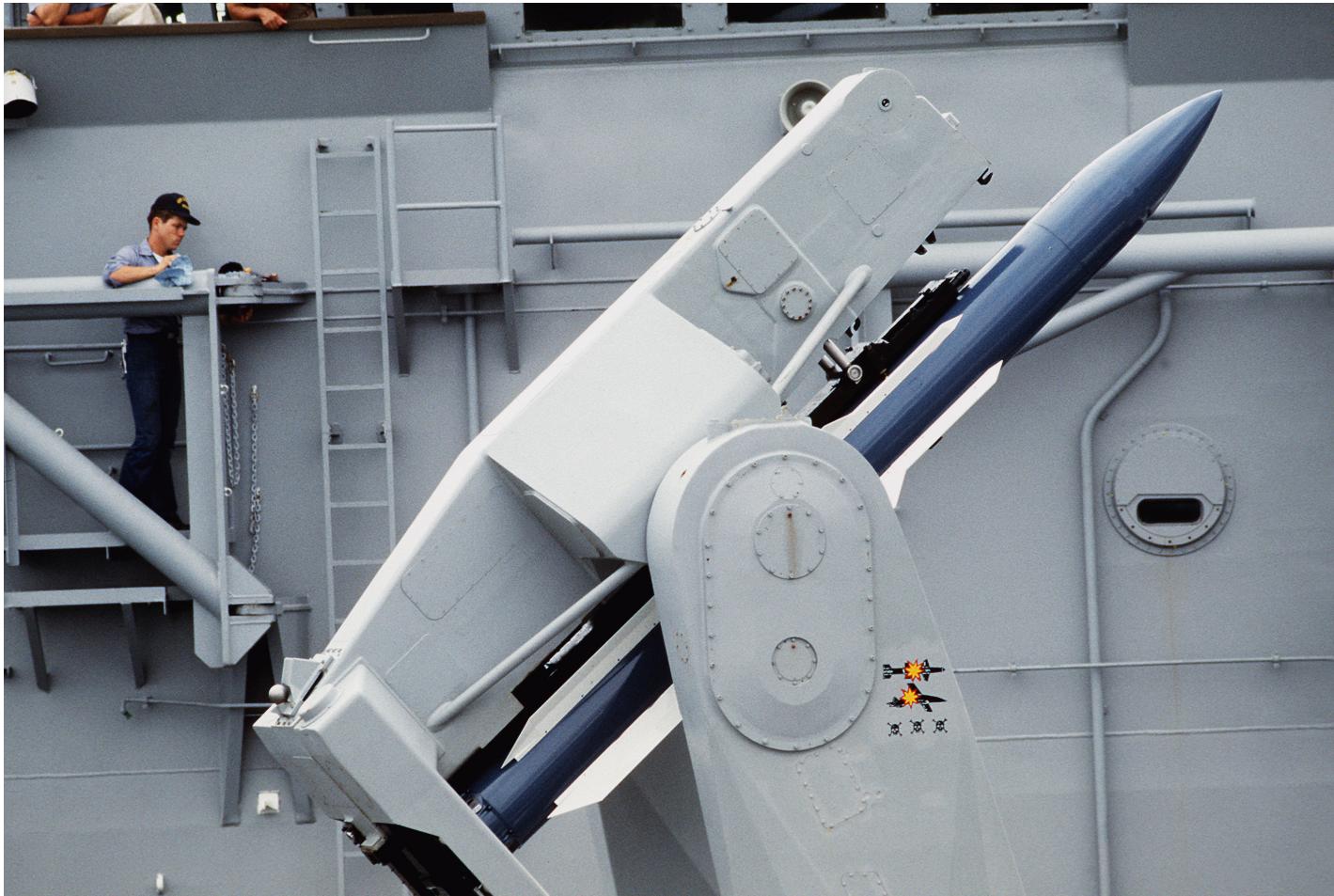


Naval Warship Fire Control Software

- Late 1980s Persian Gulf – United States and Iranian forces in conflict.
- Forward missile launcher engaged to fire RIM-66 surface to air missile.
- Missile ejected off launcher.
- Missile engine does not ignite.
- Missile worth US\$250,000 falls into ocean and sinks.
- Warship was fortunate that opposing warships were not able to attack it.



RIM-66 Launcher



Naval Cruiser Fire Control

- Cause of fault initially unclear .
- Hardware is 100% fault free.
- Software operating normally with no fault status.
- Possible consequences serious since cruiser defends a carrier battle group from missile attacks.



Naval Cruiser Fire Control

- Repeated simulated and real launches on test ranges fault free.
- Fault eventually replicated when total CPU load extremely high.
- Conditions for fault extremely infrequent and difficult to replicate.
- Fault found to be relatively easy to fix once known.

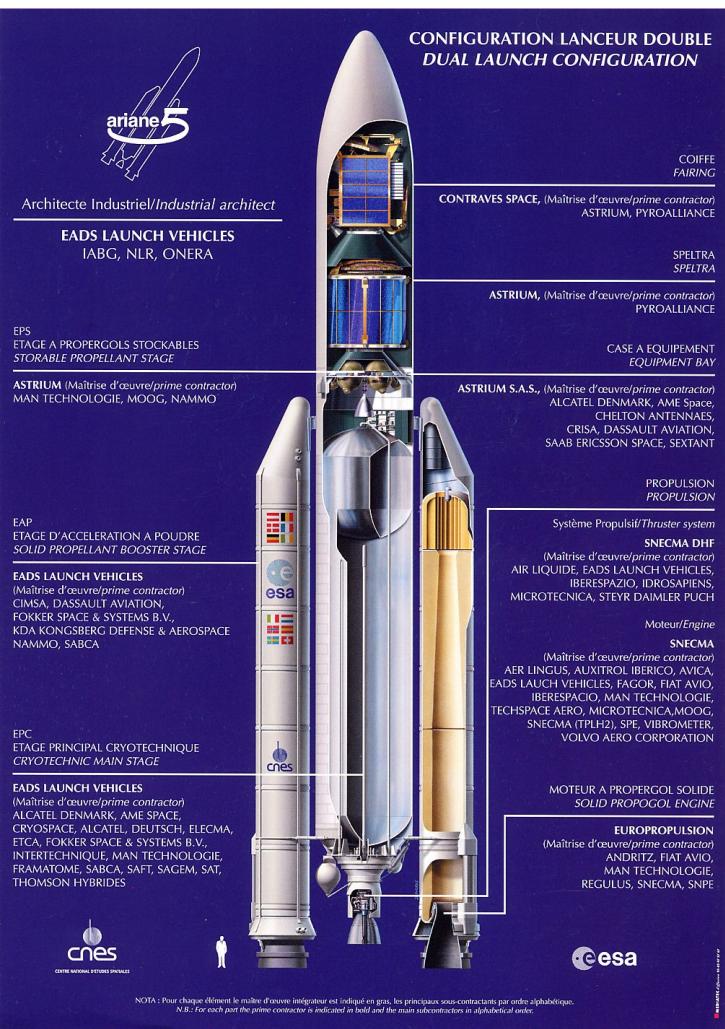


Naval Cruiser Fire Control

- Cause of fault is use of switch state polling, rather than interrupts.
- Launcher rail uses position switch to sense when the missile is about to leave the rail.
- Once missile about to leave rail, ignition signal sent to ignite engine.
- Under heavy CPU load the frequency of switch state polling too low.
- Missile left rail before switch state sampled
- Software ‘thought’ the missile was still on the launch rail.



Ariane 501 Booster Prototype Loss



https://www.youtube.com/watch?v=gp_D8r-2hwk

Ariane 501 Booster Prototype Loss

- New Ariane '5' booster launched with payload of several satellites.
- Ariane 5 uses digital redundant multiple CPU flight control system.
- Soon after launch, travelling at about Mach 1, booster attempts 90 degree turn.
- Acceleration so large that booster breaks up and fuel explodes.
- Hundreds of millions of dollars worth of hardware lost.
- Major environmental hazard due to unburned toxic propellant spill.

Ariane 501 Booster Prototype Loss

- Flight control hardware recovered and found to be fault free.
- Flight control software cause of disaster.
- Code for new booster developed by reusing code from Ariane 4 design.
- Different system design caused code to produce exception and crash.
- Rocket engine nozzles lock into full deflection position as a result.
- Aerodynamic forces cause vehicle breakup.



Mariner Venus Probe Loss



- Flight control software failure.
- Expensive satellite and booster lost.
- Fault traced to broken Fortran DO loop.
- Typographical error in source code.
- <https://www.youtube.com/watch?v=VjJgiDuHIRw>



Mars Climate Orbiter Probe Loss

- Trajectory error as space probe approached Mars.
- Attempt at orbit insertion fails and probe burns up in the atmosphere.
- Arthur Stephenson, chairman of the *Mars Climate Orbiter Mission Failure Investigation Board*: “The 'root cause' of the loss of the spacecraft was the failed translation of English units into metric units in a segment of ground-based, navigation-related mission software, as NASA has previously announced.”



Typical Causes of Software Failures

- Programmer did not understand the system design very well.
- Programmer made unrealistic assumptions about operating conditions.
- Programmer made coding error.
- Programmers and hardware engineers did not talk to each other.
- Inadequate or inappropriate testing of code.



Dormant Fault Problem

- Statistical models used for hardware are irrelevant.
- Code may be operational for years with a fatal bug hidden somewhere.
- A set of conditions may one day arise which trigger the fault.
- If major disaster arises it may be impossible to recreate same conditions.



Complex System Problem

- **Extremely complex system will be extremely difficult to simulate or test.**
- **Complexity may result in infeasible regression testing time.**
- **Components of system may interact in ‘unpredictable’ ways .**
- **Synchronisation failures may arise.**
- **Fault may be hidden and symptoms not easily detectable due complexity.**



Coding for Reliability

- **Problem must be well understood, especially conditions which may arise.**
- **Hardware can NEVER be trusted!**
- **Operating Systems can NEVER be trusted!**
- **Libraries can NEVER be trusted!**
- **Documentation can NEVER be trusted!**
- **Compilers can NEVER be trusted!**



Coding for Reliability

- Design objectives must be understood.
- Each module should check the bounds on arguments.
- Each module should sanity check its results.
- Datastructures should be redundant or checksummed.
- Consistency checking should be used generously.
- Each module should be tested thoroughly before use.
- Recycled code should be tested thoroughly before use and well understood.



System Design for Reliability

- **Design objectives must be understood.**
- **Redundancy should be used as appropriate.**
- **Failure modes and consequences should be understood.**
- **Each module should be tested thoroughly before use.**
- **Recycled modules should be tested thoroughly before use and understood.**



Conclusions

- Deterministic proof of code reliability difficult or impossible.
- Regression testing may miss dormant faults or complex system faults.
- Human error by programmers and operators should be assumed.
- Hardware, operating systems, networks, libraries, documentation and compilers may have hidden or unknown problems.
- Complexity introduces unexpected interactions.



Axioms to Memorise

- Murphy's Law applies 99% of the time (Vonada's Law)
- Simpler solutions are usually easier to prove correct (Occam's Razor)
- Paranoia Pays Off (Kopp's Axiom)

Reading/References

- <http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html>
- <http://www.around.com/ariane.html>
- ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf
- ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_charts.pdf