DEPENDABLE SYSTEMS AND CRITICAL INFRASTRUCTURES DESIGN

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΜΣ Κυβερνοασφάλεια και Επιστημη Δεδομενων
MSc Cybersecurity and Data Science
DEPT OF INFORMATICS
UNIVERSITY OF PIRAEUS

RELIABILITY ENGINEERING AND HARDWARE FAULT-TOLERANCE

DIMITRIS AGIAKATSIKAS, PANAGIOTIS KOTZANIKOLAOU, MIHALIS PSARAKIS

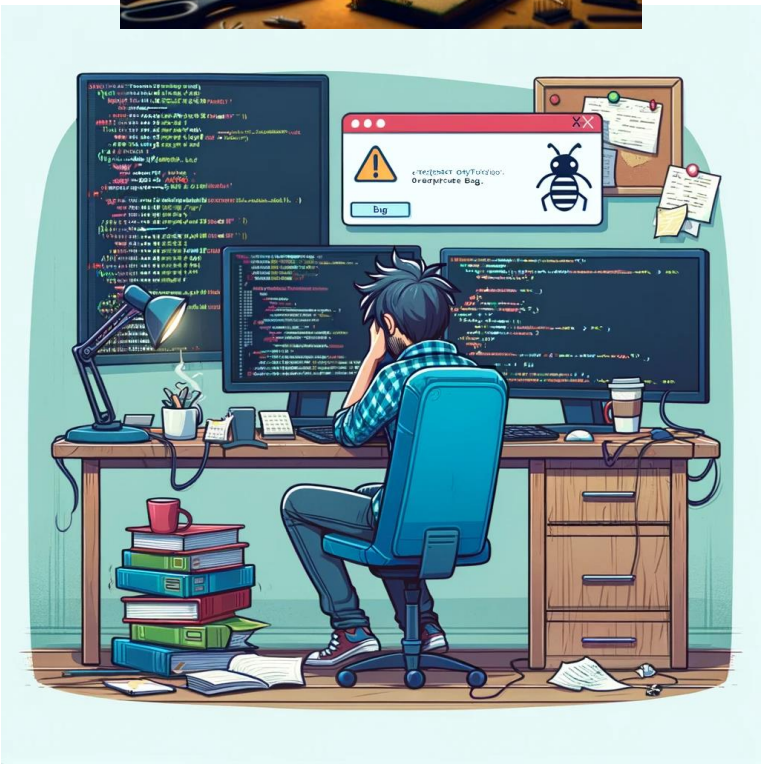
WHAT IS FAULT TOLERANCE (AKA DEPENDABLE COMPUTING)

It is fair to say

- There is no **hardware** with zero probability to fail
- There is no complex **software** that is free of bugs

So how should we cope with faults, especially **safe critical** computing systems ????

- We need to acknowledge the existence of faults and incorporate techniques to **tolerate** these faults while delivering an acceptable level of service



BRIEF HISTORY OF DEPENDABLE COMPUTING

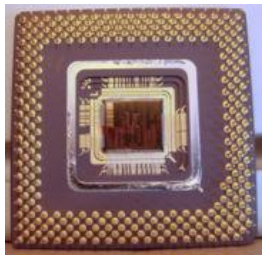
1940s: ENIAC, with 17.5K vacuum tubes and 1000s of other electrical elements, failed once every 2 days (avg. down time = minutes)

1950s: Early ideas by von Neumann (multichannel, with voting) and Moore-Shannon ("crummy" relays)

1960s: NASA and military agencies supported research for long-life space missions and battlefield computing

1970s: The field developed quickly (international conference, many research projects and groups, experimental systems)

1980s: The field matured (textbooks, theoretical developments, use of ECCs in solid-state memories, RAID concept), but also suffered some loss of focus and interest because of the extreme reliability of integrated circuits

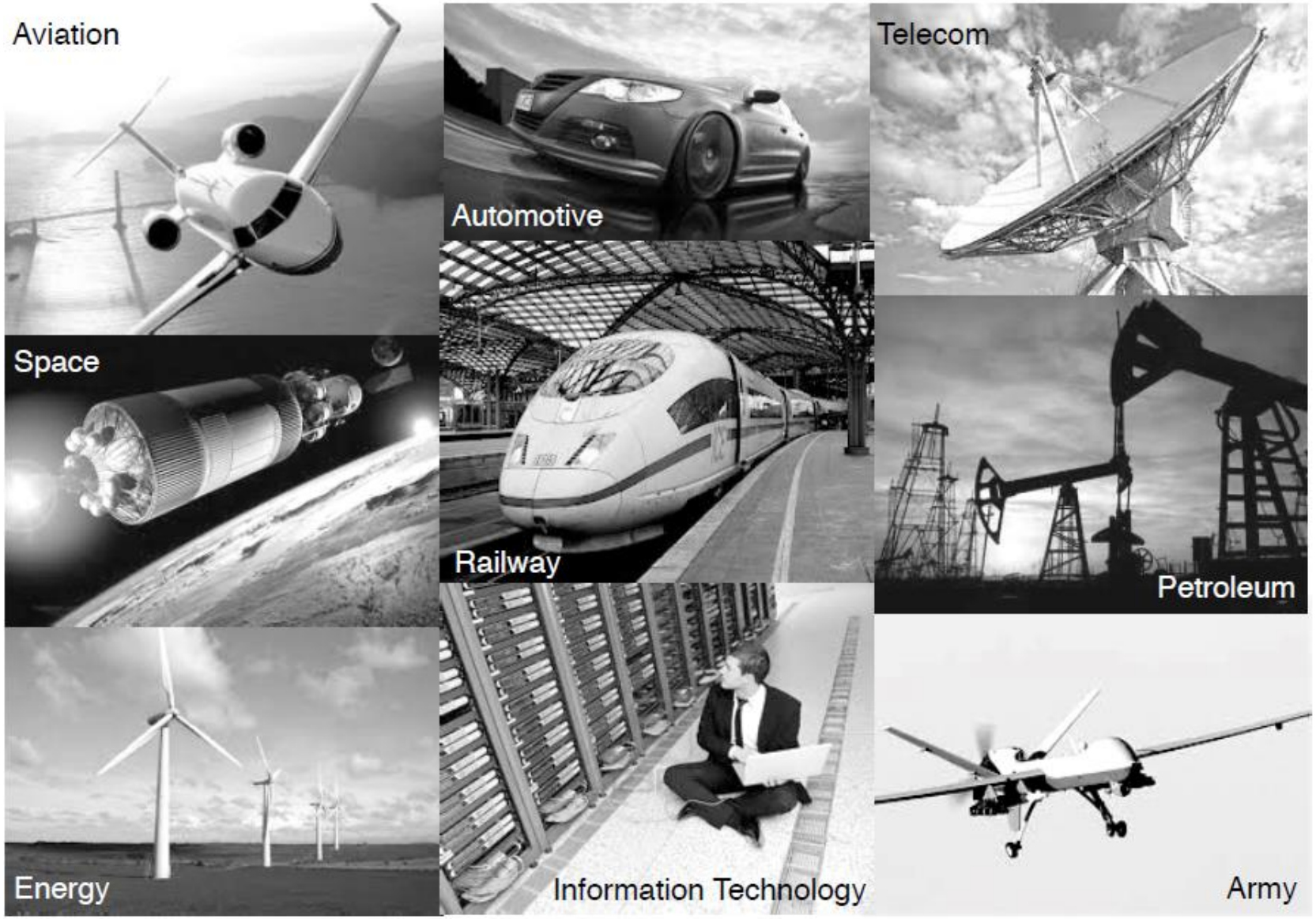


1990s: Increased complexity at chip and system levels made verification, testing, and testability prime study topics

2000s: Resurgence of interest owing to less reliable fabrication at ultrahigh densities and "crummy" nanoelectronic components

2010s: Integration of reliability, safety, privacy, and security concerns, particularly in the cloud, artificial intelligence systems, and IoT

FAULT TOLERANT APPLICATIONS

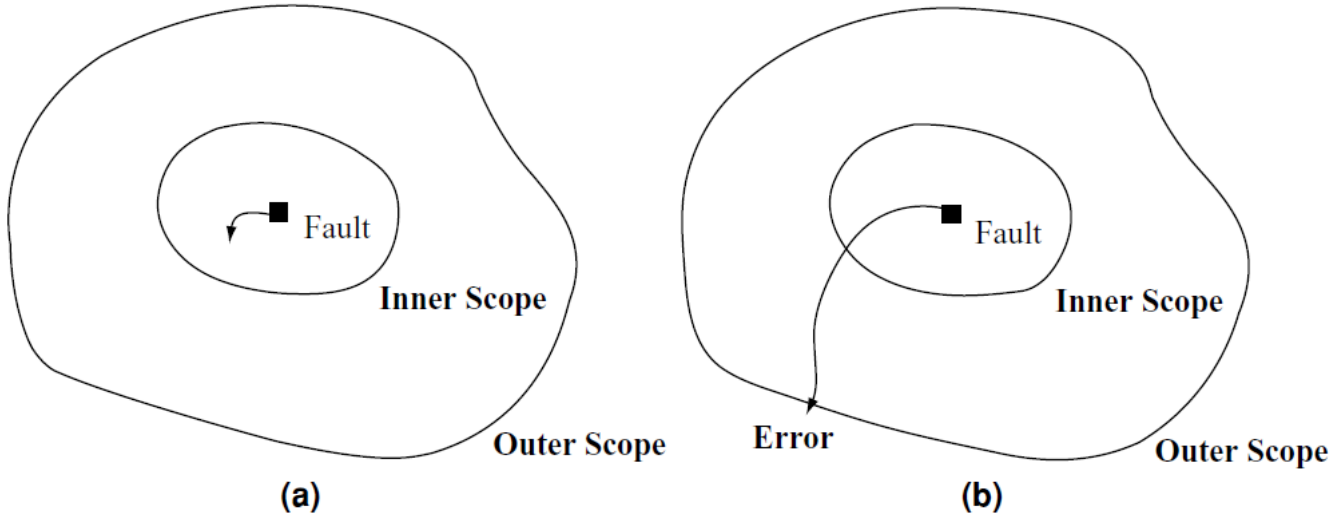


WHAT IS A FAULT? WHAT IS AN ERROR? WHAT IS A FAILURE

Deviation of the system's behavior out of specs

User Applications
Operating System
Firmware
Architecture
Circuits
Process Technology

Abstraction Layers of a computing system



- (a) Fault within the inner scope masked and not visible outside the inner scope.
- (b) Fault propagated outside the outer scope and visible as an error.
- **The manifestation of a fault will produce error in the system's state, which could lead to a system failure**

EXAMPLES AND ANALOGIES → FUNCTIONAL FAILURES

Example:	**An application ranks images into three categories, A, B and C.**
Golden rank	A=90%, B=4%, C=6% → This is the correct rank result
Fault	A overflow occurred in one variable of the application
Error	A=89%, B=6%, C=5% → We have a deviation from the golden rank
Failure	We do not have any failure since the system's behavior did not deviate from the system's specifications, i.e., the image ranking is still correct

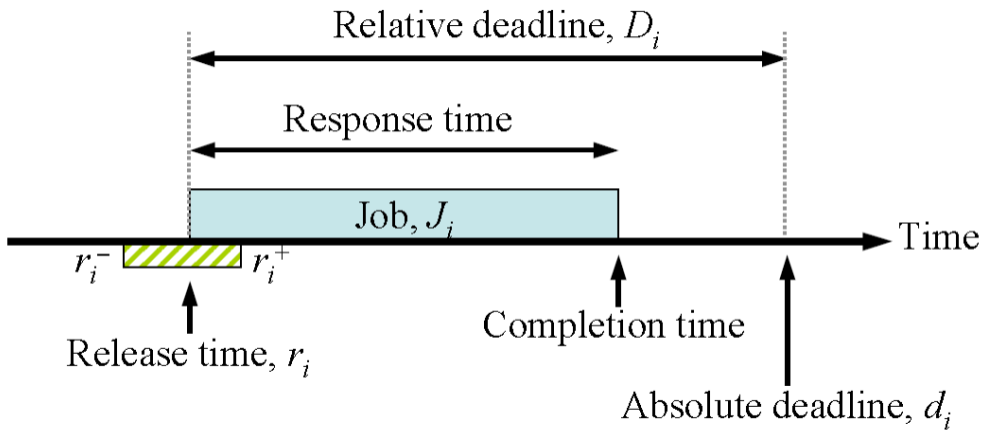
Therefore, not every defect, fault, error, malfunction, or degradation leads to failure

EXAMPLES AND ANALOGIES → TIMING FAILURES

Example: A real time system.

Error free completion time	Tc = 10ms
Absolute Deadline	Tc = 15ms
Fault	CPU register corrupted due to a soft error
Error	Tc-erroneous = 14.99ms

Failure Do we have a failure??



MORE EXAMPLES AND ANALOGIES

Example:	**Automobile brake system**
Defect	Brake fluid piping has a weak spot or joint
Fault	Brake fluid starts to leak out
Error	Brake fluid pressure drops too low
Malfunction	Braking force is below expectation
Degradation	Braking requires higher force or takes longer
Failure	Vehicle does not slow down or stop in time

FAULT CLASSIFICATION

Duration

- Permanent
- Transient
- Intermittent

When they were introduced

- Design phase
- Implementation phase
- System operation

Intent

- Unintentional
- Intentional
  - Non malicious
  - Malicious
    - Intentional
    - Non-intentional, i.e., a faulty sensor (Byzantine faults)

FAILURES CRITICALITY

- The Windows OS on your laprop crashes (blue screen) → Noncritical
- A bank transaction is lost → Critical
- Two trains collide
- The automatic landing system of an airplane fails three second before touch down

→ Safety critical

DEPENDABLE COMPUTER SYSTEMS

Long-life systems: Fail-slow, Rugged, High-reliability
Spacecraft with multiyear missions, systems in inaccessible locations
Methods: Replication (spares), error coding, monitoring, shielding

Safety-critical systems: Fail-safe, Sound, High-integrity
Flight control computers, nuclear-plant shutdown, medical monitoring
Methods: Replication with voting, time redundancy, design diversity

Non-stop systems: Fail-soft, Robust, High-availability
Telephone switching centers, transaction processing, e-commerce
Methods: HW/info redundancy, backup schemes, hot-swap, recovery

Just as performance enhancement techniques gradually migrate from supercomputers to desktops, so too dependability enhancement methods find their way from exotic systems into personal computers

ASPECTS OF DEPENDABILITY

DEPENDABILITY-RELATED TERMS WITH THEIR MOST COMMON QUALITATIVE USAGES AND QUANTIFICATIONS (IF ANY).

Term	Qualitative Usage(s)	Quantitative Measure(s)
Availability	Highly available, High-availability, Continuously available	Steady availability, availability, MTBF, MTTR
Integrity	High-integrity, Tamper-proof, Tamper-proof	
Maintainability	Easily maintainable, Maintenance-free, Self-repairing	
Performability		MEBF, MWBF, MCBF
Reliability	Reliable, Highly reliable, Highly reliable, Ultrareliable	Reliability, MTTF or MTFF
Resilience	Resilient	
Robustness	Robust	Impairment tolerance count
Safety	High-safety, Fail-safe	Risk
Security	Highly secure, High-security, Fail-secure	
Serviceability	Easily serviceable	
Testability	Easily testable, Self-testing, Self-checking	Controllability, Observability

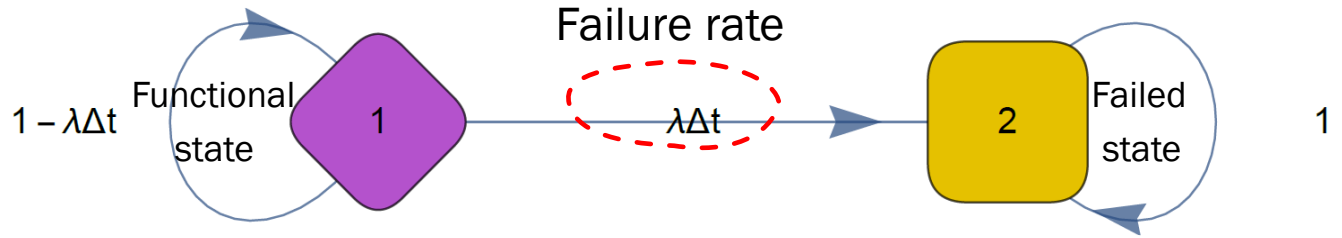
RELIABILITY - DEFINITION

The ability of a system or component to perform its required functions under stated conditions for a specified period of time

[IEEE610]: IEEE Standard Glossary of Software Engineering Terminology,

IEEE Std 610.12-1990 (R2002).

RELIABILITY

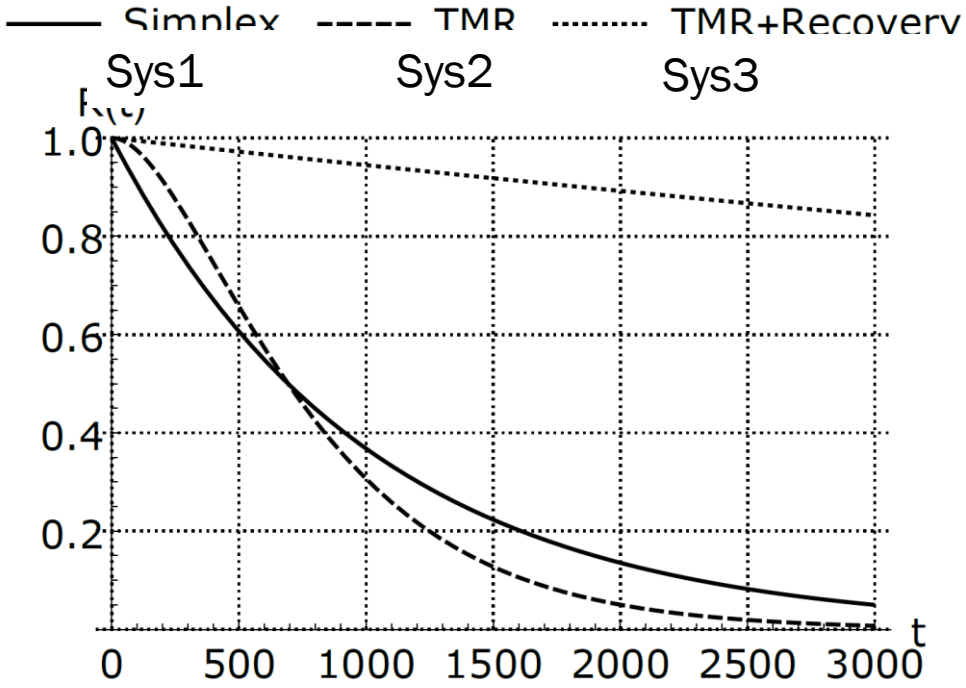


- **R(t): probability that the system remains in the "Functional" state through the interval [0, t]**

  R(t) = P(not failed during [0, t])

  assuming it was functional at time t = 0

- **t = duration of the mission. If a system needs to work for slots of ten hours at a time, then (t = 10hrs)**
- **R(t) for system with exponential distributed failure rate λ is**
  - **R(t) =** $e^{-\lambda t}$



R(t) is a non-increasing function varying from 1 to 0 over [0,+∞)

EXAMPLE

Context

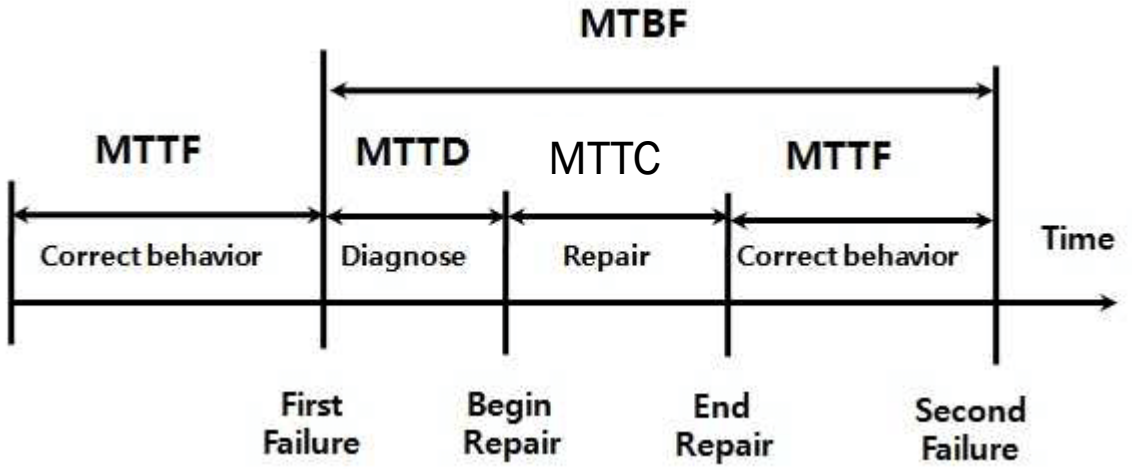
- We are developing a computing system for a Cubsat to fly in Low Earth Orbit (LEO)
- The mission will last 1 month
- We need to choose RAM1 or RAM2 for the system such that R(t=1 month) of the RAM is not lower than 0.83.
- λ of RAM1 is 1 failure per year, while for RAM2 it is twice the failure rate of RAM1.

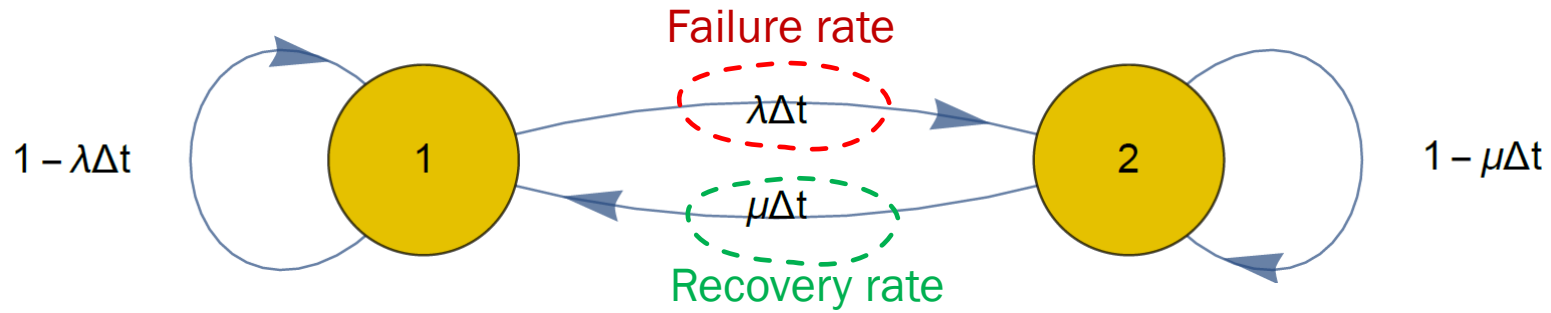
Question?

Which RAM should I choose???

MEAN TIME TO FAILURE (NON REPAIRABLE SYSTEMS)
MEAN TIME BETWEEN FAILURE (REPAIRABLE SYSTEMS)

- MTTF = $\int_0^\infty R(t) = \int_0^\infty e^{-\lambda t} = 1/\lambda$
- MTTD → Average time to detect
- MTTC → Average time to correct
- MTTR = MTTD + MTTC
- MTBF = MTTF / MTTR





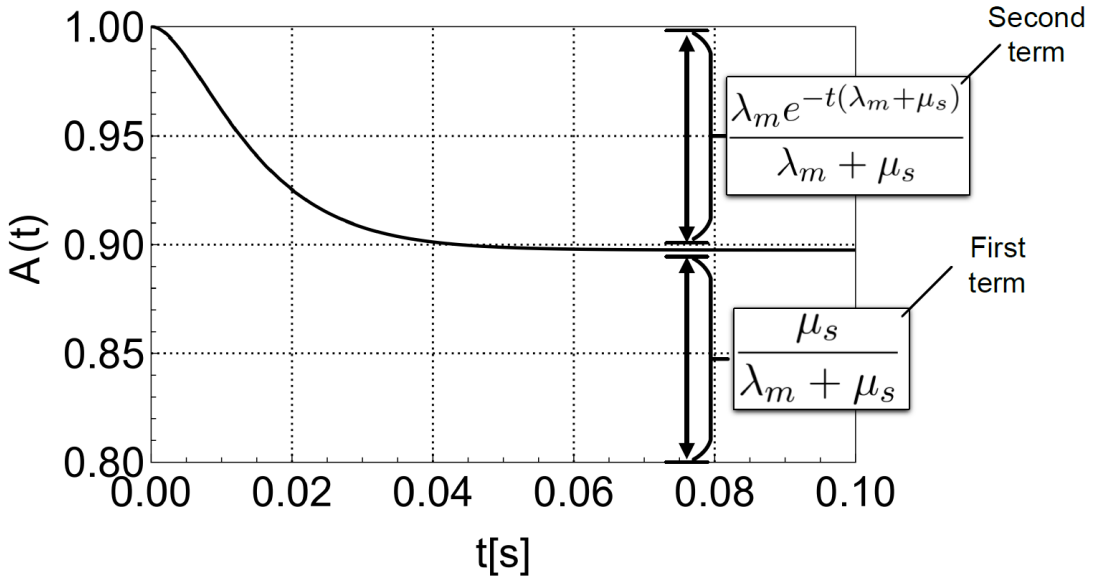
AVAILABILITY VS STEADY AVAILABILITY

- Let's examine the Availability *A(t)* function

  $A(t) = \frac{\mu}{\lambda + \mu} + \frac{\lambda e^{-t(\lambda + \mu)}}{\lambda + \mu}$ Eq. (14)

- The system starts at state 1 (working) and at time t=0, the availability A(0) = 1
- As the time passes, the system reaches the steady state availability
- In Eq. (14), the first term captures the steady state expression, and the second term captures the transitory behaviour as the system approaches steady state.
- Therefore, the steady state *A* of the system is

$$A = \lim_{t \to \infty} A(t) = \frac{\mu}{\lambda + \mu} + \frac{\lambda e^{-t(\lambda + \mu)}}{\lambda + \mu} = \frac{\mu}{\lambda + \mu}$$



SYSTEM IN SERIES

- Assume a system with n components, e.g. CPU, memory, disk, etc.
- All components must be UP for the system to be operational
- The reliability of the system is $R_{series}(t) = \prod_{i=1}^{n} R_i(t)$

  where Ri is the reliability of component i
- Assuming constant failure rate λ:

$$R_{series}(t) = \prod_{i=1}^{n} \exp(-\lambda_i t) = \exp(-\sum_{i=1}^{n} \lambda_i t) = \exp(-\lambda_{system} t)$$

$$R_{sys} = R_1 \bullet R_2 \bullet R_3 \bullet \ldots \bullet R_n$$

SYSTEM IN SERIES

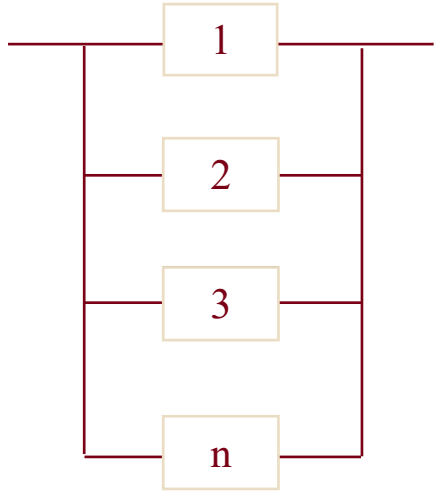
- System failure rate:

$$\lambda_{system} = \sum_{i=1}^{n} \lambda_i$$

- MTTF = $1/n\lambda$
- Example:
  - If each component has reliability 0.99 for one year, then a system with 10 components has reliability 0.99^{10} ή 0.9

SYSTEM IN PARALLEL

- Assume a system with n spare units (redundancy)
- Only one unit must be UP for the system to be operational
  - Reliability of the unit i (probability to be up): Ri
  - Unreliability of the unit i (probability to be down): 1-Ri
- Probability all the units to be down
  - $(1-R_1)(1-R_2)\ldots(1-Rn)$



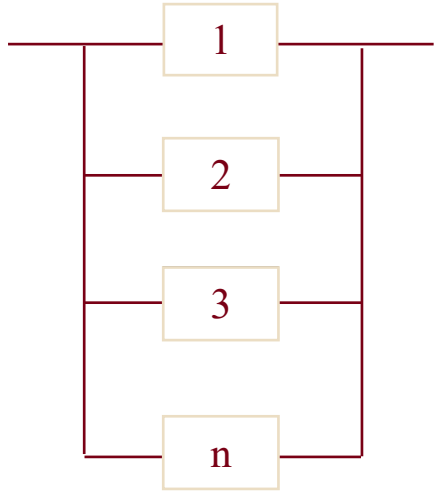
- System reliability:

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

SYSTEM IN PARALLEL

- Example:
  - A system consists of 10 spare units, and
  - The reliability of each unit is 0.75
  - The system reliability is:

    $1-(1-0.75)^{10} \approx 0.9999$

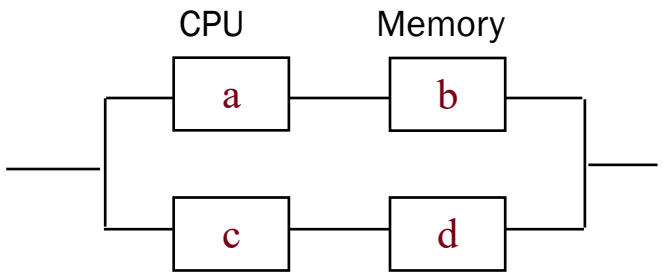


$$R_{parallel}(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t))$$

SERIAL-PARALLEL SYSTEMS

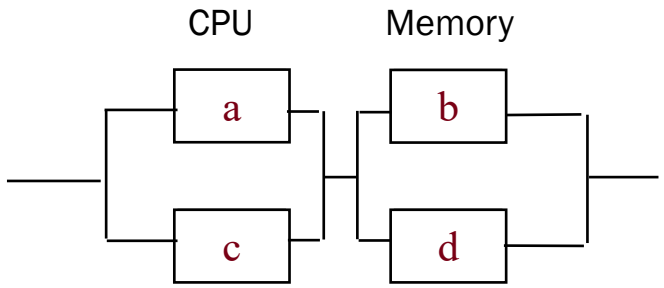
- Example: Two CPUs (a, c) are connected with two memories (b, d) in different ways

$$R_{SP} = 1 - (1 - Ra\ Rb)\ (1 - Rc\ Rd)$$



Serial-to-parallel

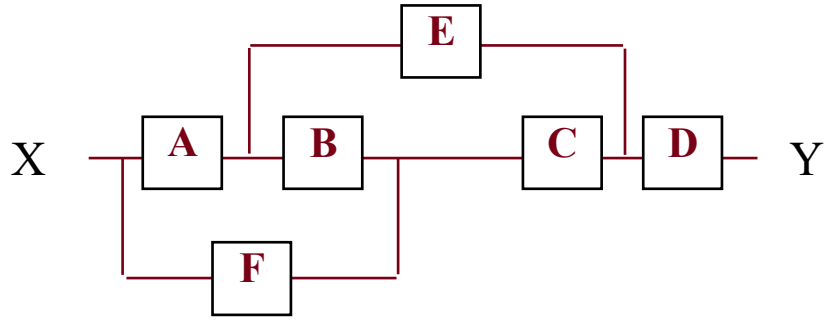
$$R_{PS} = (1 - (1 - Ra)(1 - Rc))\ (1 - (1 - Rb)(1 - Rd))$$



Parallel-to-serial

NON SERIAL-PARALLEL SYSTEMS

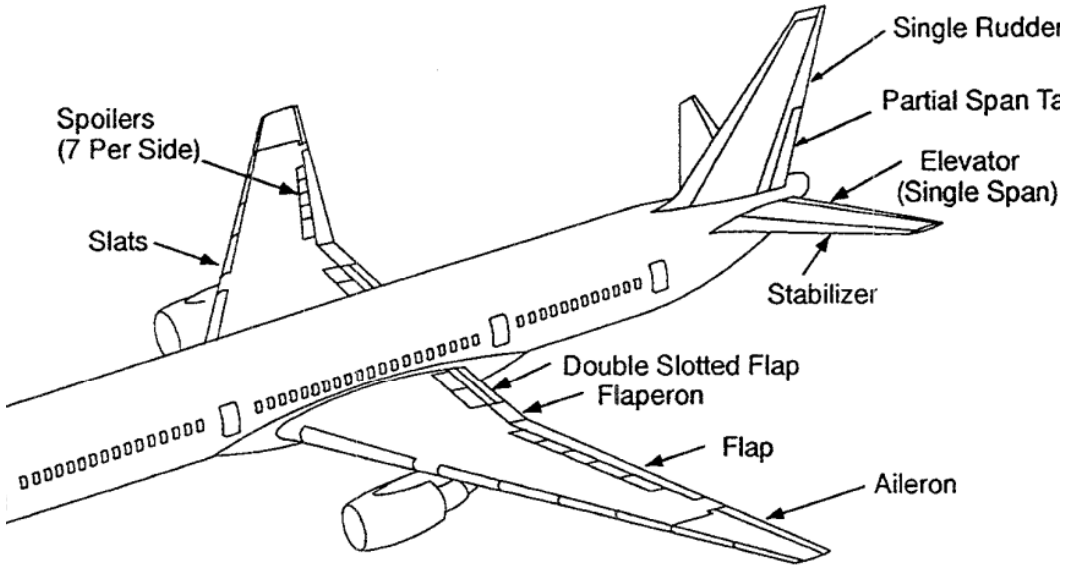
- Every path from X to Y is a system configuration that allows the system to be operational
- The system reliability can be measured by examining the operation of the system for each unit m



Rsys = $Rm \times P(\text{system works} | m \text{ works}) + (1\text{-}Rm) \times P(\text{system works} | m \text{ fails})$

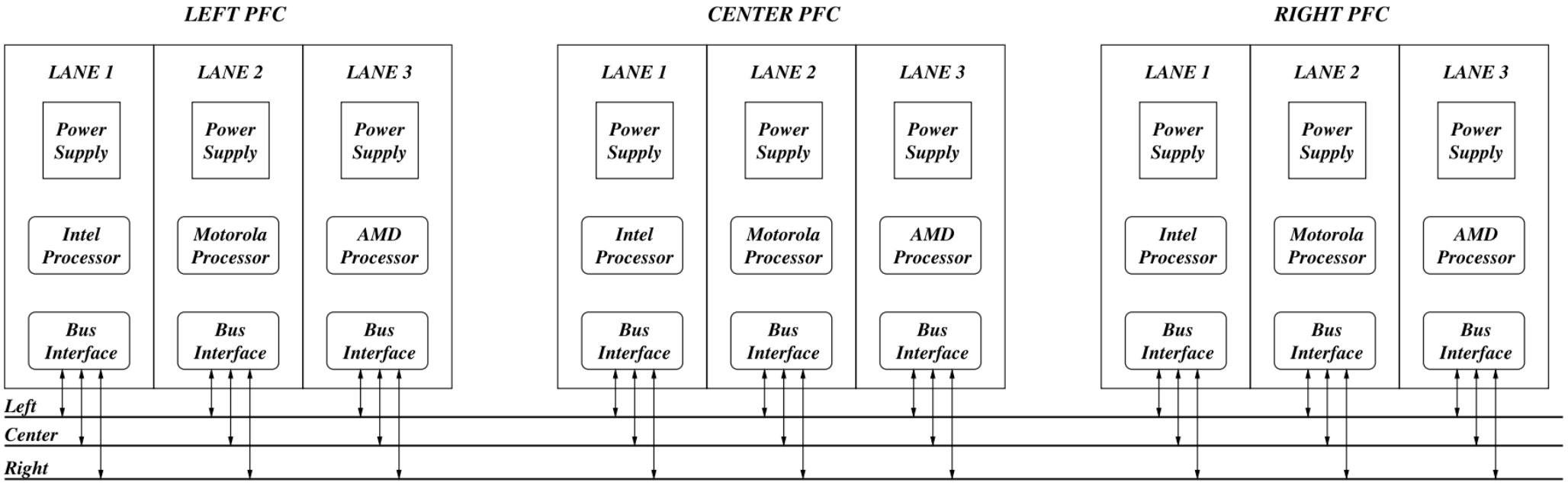
FLIGHT CONTROL SYSTEM OF BOEING 777

- Boeing's first commercial jet with FBW was the 777 (1995)
- In planes with no FBW, the actuators moving the surfaces of the plane are controller with mechanical interfaces
- Planes with FBW use a flight computer that reads the input from pilots and sensors (e.g., speed, angle of attack etc.) and controller the actuators in a closed feedback loop.
- In simple words, the pilots gives simple instructions to the computer and the computer in turn controls the plane. Needless to say, this computer should be very reliable!
- The A(T=1h) of 777's computer is 0.999999999, i.e., the probability of a fault impacting the integrity and availability of the computer should be less than 10^{-10} / 1h



BLOCK DIAGRAM OF THE PRIMARY FLIGHT COMPUTER (PFC) SYSTEM

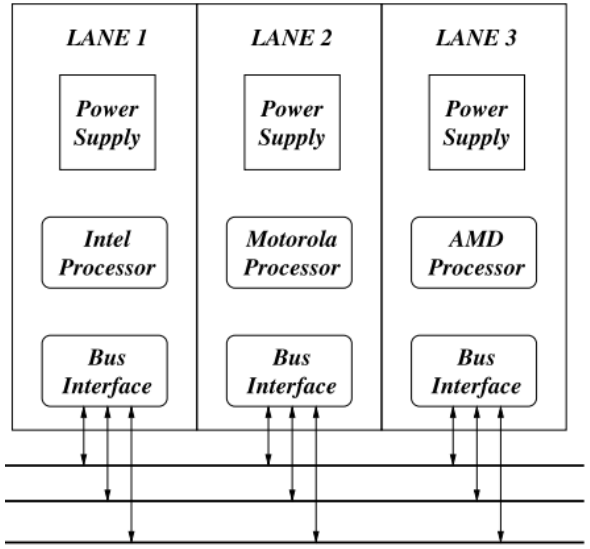
- The PFC is a TMR system consisting of three computing channels (left, center, right).
- Each computing channel consists of three computing lanes.
- Each lane has its own power supply.
- Each channel is physically and electrically isolated via ARINC 629 data bus



digital autonomous terminal access communication (DATAC), also called ARINC 629 data bus

BLOCK DIAGRAM OF THE PRIMARY FLIGHT COMPUTER (PFC) SYSTEM

1 of the 3
computing channels



- All communications over the ARINC 629 data bus are CRC checked
- Each computing lane uses a different processor (Intel, AMD and Motorola)
- Each processor runs control software that is compiled by a different compiler
- Each computing lane does not operate in TMR. Instead one of the three lanes serves as the command processor and the other two monitor the outputs generated by the designated command processor.
- Only the command processor is communicating through the data buses with the remaining two channels; it transmits its proposed flight surface command to the other two channels
- Each command lane receives three values of the proposed commands, and performs a median value select to determine what is called the “selected” surface command.

Initially, Boeing management decided to use also 3-version programming for developing the control software.

But each team of programmers asked to many questions to clarify software requirements and the management cancelled the 3-version programming approach