EDA122/DIT061 Fault-Tolerant Computer Systems
DAT270 Dependable Computer Systems

Welcome to Lecture 14

Byzantine failures
Layered fault tolerance
Error detection

Outline

- Characterization of Failure Modes
- Byzantine failures
- Layered fault tolerance
- Error detection techniques
- Self-checking nodes

Failure modes
(from lecture 3)

- A failure mode describes the nature of a failure, i.e., the way in which a ***service provider*** (a system, subsystem, or module) can fail
- A ***service provider*** can have many failure modes
- Examples of failure modes:
  - ***Value failure*** – a service provider delivers an erroneous result
  - ***Timing failure*** – a service provider delivers a result too late, or too early
  - ***Silent failure*** – a service provider delivers no result
  - ***Signaled failure*** – a service provider sends a failure signal
- A service provider must have internal mechanisms for error detection to enforce silent or signaled failures

Note: A v*alue failure* is the same as a *content failure*. Both terms are used in the literature.

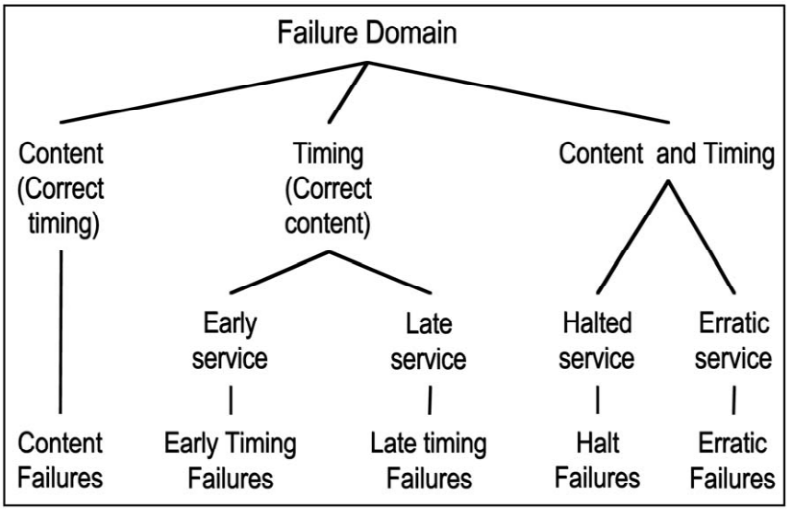
Characterization of Failure Modes

- Failure domain
  - Content (Value)
  - Timing
- Detectability of failures
- Consistency of failures
- Consequences of failures on the environment

See “Basic Concepts and Taxonomy of Dependable and Secure Computing”, pp. 18-19.

Failure domain

- **Content failures.** The content of the information delivered at the service interface deviates from implementing the system function.
- **Timing failures.** The time of arrival or the duration of the information delivered at the service interface (i.e., the timing of service delivery) deviates from implementing the system function.
- **Failure for which both content and timing are incorrect**
  - **Halt failure,** or simply **halt**, when the service is halted; a special case of halt is silent failure, or simply **silence**, when no service at all is delivered at the service interface.
  - **Erratic failure**, when a service is delivered (not halted), but is *erratic* (e.g., babbling)



Failure Detectability

- **Signaled failures –** the system signals service failures to the user(s).
- **Unsignaled failures –** no warning from the system is given
- Signaling at the service interface originates from detection mechanisms checking the correctness of the service.
- The detection mechanisms have themselves two failure modes:
  1. Signaling a failure when no failure has actually occurred; a **false alarm**
  2. Not signaling a service failure; an **unsignaled failure**

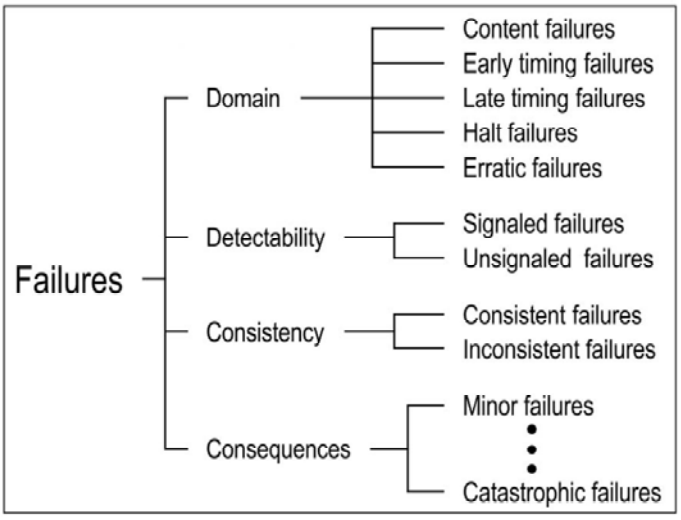
Failure Consistency

- **Consistent failures.** The incorrect service is perceived identically by all system users.
- **Inconsistent failures (or Byzantine failures).** Some or all system users perceive the incorrect service differently. Some users may perceive a correct service.

Consequence of Failures on the Environment

- Grading the *consequences of the failures* upon the system environment involves a definition of *failure severities*
- The definition of the severity levels depends on the application
- Examples of criteria for determining the classes of failure severities include:
  1. the outage duration (for availability)
  2. the possibility of human lives being endangered (for safety)
  3. the type of information that may be unduly disclosed (for confidentiality)
  4. the extent of the corruption of data and the ability to recover from these corruptions (for integrity)

Summary of taxonomy of failure modes



Reading Instructions:
"Basic Concepts and Taxonomy of Dependable and Secure Computing"
(Careful reading)

The following sections shall be read carefully:

2. The Basic Concepts
3.2.1 The Taxonomy of Faults
3.3.1 Service Failures
3.4 Errors
3.5 The Pathology of Failure: Relationship between Faults, Errors, and Failures
4.1 The Definitions of Dependability and Security
5.1 Fault Prevention
5.2 Fault Tolerance
5.3 Fault Removal
5.4 Fault Forecasting

Check out the reading instructions about parts designated as "Normal reading".

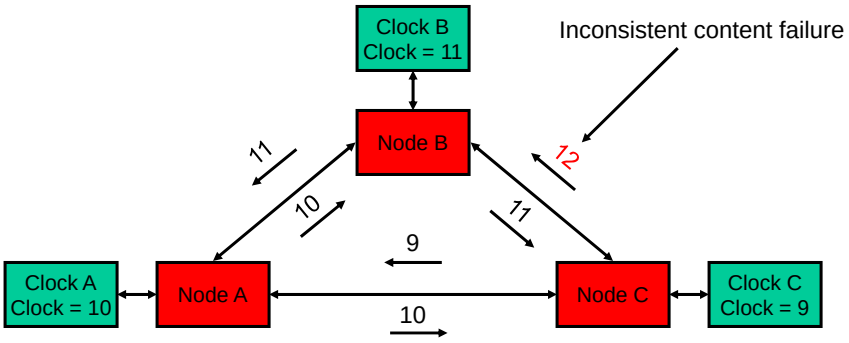
Outline

- Characterization of Failure Modes
- Byzantine failures
- Layered fault tolerance
- Error detection techniques
- Self-checking nodes

Agreement in presence of Byzantine failures

- A Byzantine failure is an inconsistent, non-detectable, content failure.
- The Byzantine generals problem:
  - How do replicated units reach agreement on a non-replicated value in the presence of inconsistent content failures?
- This problem is also known as the interactive consistency problem

Impossibility of Reaching Agreement on Clock Value



Voting algorithm: Use median value to resynchronize local clock
Node C sends *Clock C = 9* to Node A and *Clock C =12* to Node B
Node A uses *median*{9,10,11} = 10, Node B uses *median*{10,11,12} = 11

Node C is said to be maliciously faulty, since it fools the non-faulty nodes to set their clocks to different values.

Byzantine Generals Problem
Scenario

Several armies are camped outside a city which they plan to attack.

Each army is led by a commander. One of the commanders is a general that will issue the order *attack or retreat.*

One or several commanders, including the general, may be traitors.

To win the battle all loyal commanders must attack at the same time.

The traitors will attempt to fool the loyal commanders so that not all of them attack at the same time.

To stop the traitors' malicious plan, all commanders exchange messages directly with each other.

Algorithm ICA(m)

- Tolerates *m* byzantine failures using oral messages (oral message = unprotected message)
- Objectives of the algorithm
  - O1: All fault-free commanders (subordinates) agree on the same command.
  - O2: If the general (the sender) is fault-free, then all the fault-free commanders agree on the command issued by the general.

Algorithm ICA(m)

- Requirements
  - R1: At least *3m+1* units must participate
  - R2: At least *m+1* rounds of communication must take place
  - R3: All units must be synchronized within a known skew of each other
- Assumptions about the message passing system
  - A1: Every message that is sent by a node is delivered correctly by the message passing system to the receiver.
  - A2: The receiver of a message knows which node has sent the message
  - A3: The absence of a message can be detected.

Algorithm ICA(m) (cont'd)

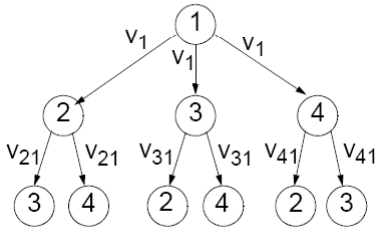
Algorithm ICA(0)

1. The value from the general is sent to every subordinate
2. Each subordinate uses the value received from the general or uses the value RETREAT.

Algorithm ICA(t>0)

1. The value from the general is sent to every subordinate.
2. For each *i,* let v_i be the value that subordinate *i* received from the general, or else be RETREAT if none received. Subordinate *i* acts as the general in Algorithm ICA(*t-1)* to send the value v_i to each of the other (*n-2)* subordinates.
3. For each *i and each j≠i,* let v_j be the value subordinate *i* received from subordinate *j* in step 2 (using Algorithm ICA(t-1)), or else RETREAT if none is received. Subordinate *i* uses the value majority(v_1,v_2, ..., v_{n-1})

Tolerating one Byzantine failure

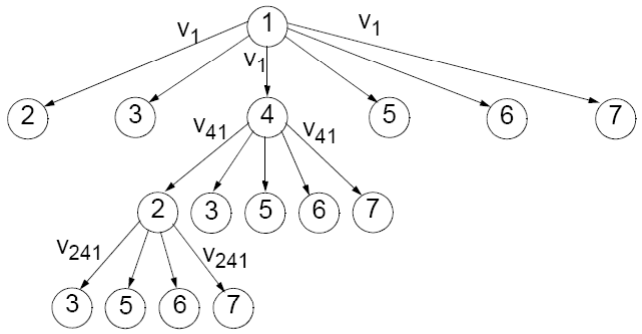


Example: Unit 2 uses *v'1 = majority{v1, v31, v41}*

Number of messages for agreement on one value is: 3 + 2·3 = 9 messages

A majority decision involving 4 units (e.g. 4 sensor values) requires: 9·4 = 36 messages

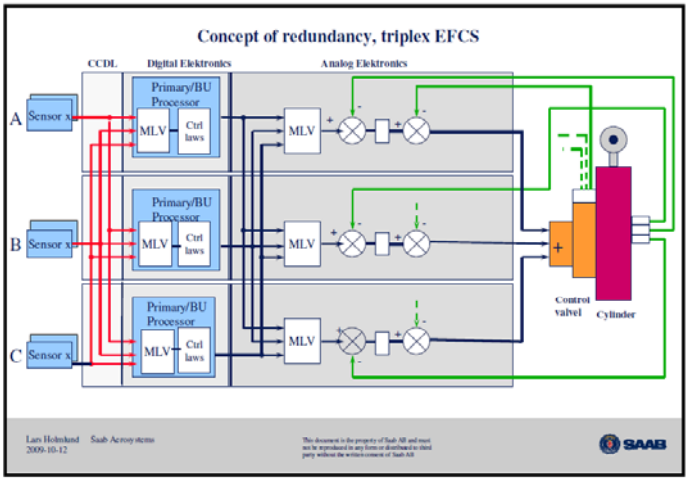
Tolerating two Byzantine failures



Example: Unit 2 uses *v'1 = majority{v1, v'31, v'41, v'51, v'61,v'71}*
v'41 = majority{v41, v341,v541,v641, v741}

Number of messages for agreement on one value is 6 + 6(5 +5·4) = 156 messages

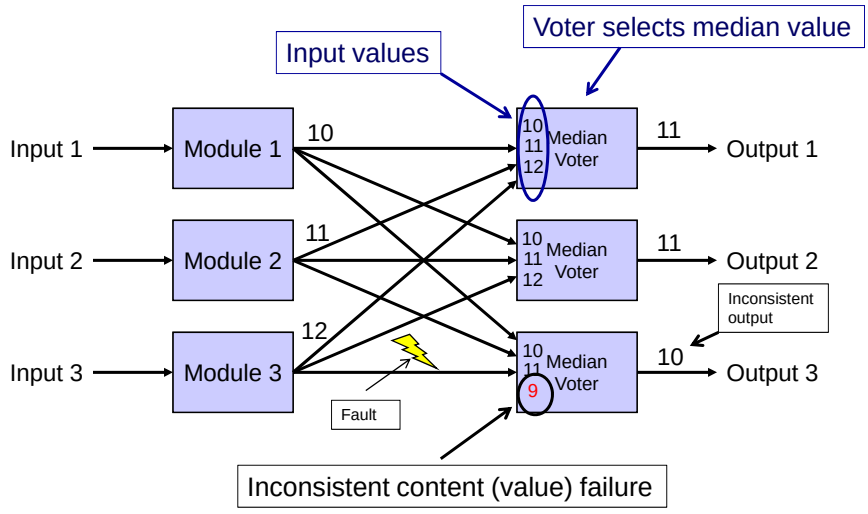
Mid-level voters in the JAS Gripen Flight Control System



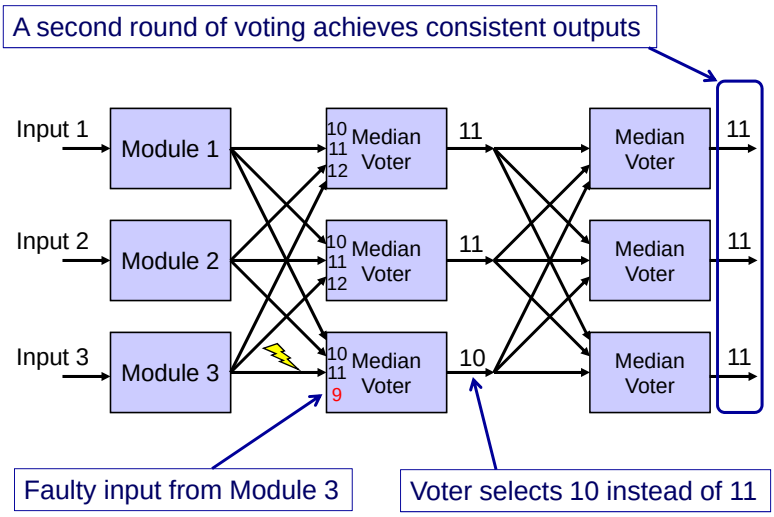
Tolerating Byzantine Failures Using Multiple Voting Planes

- Lars Holmlund mentioned in his guest lecture that the JAS Gripen flight control system can tolerate Byzantine failures
- How is this possible for a three channel system?
- We will look at Byzantine failures in a multi-stage TMR system

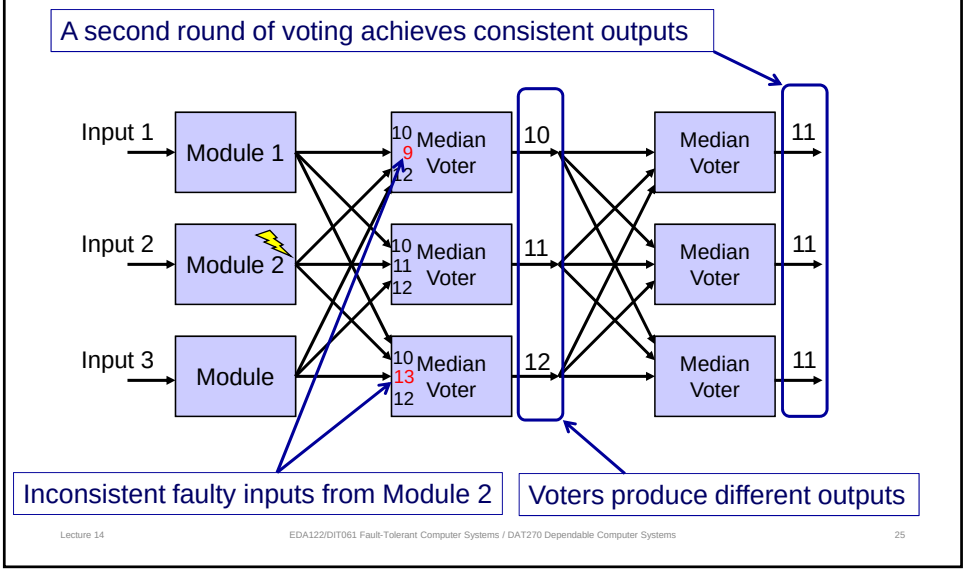
Inconsistent Content Failure in a TMR system



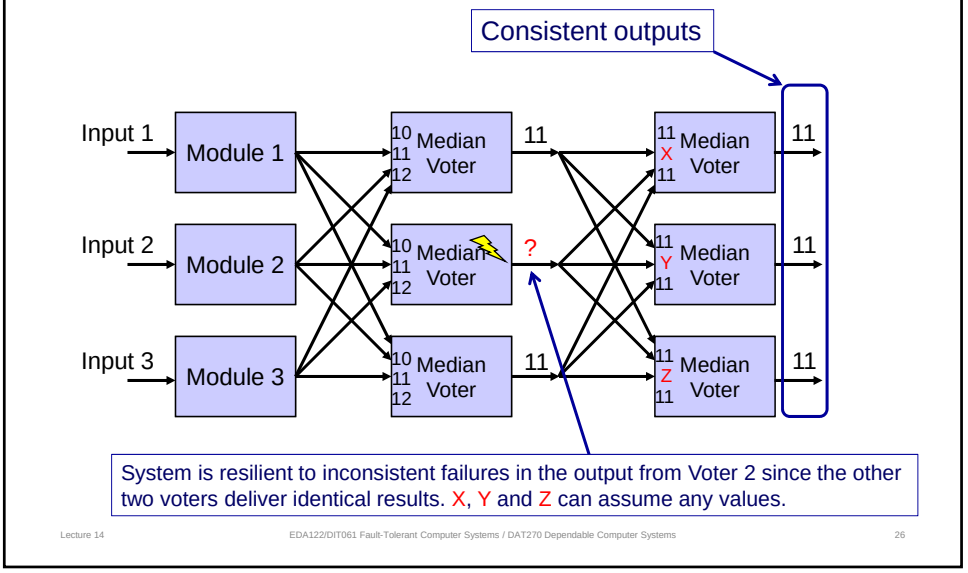
Masking Inconsistent Content Failures with Two Voting Planes



Masking Inconsistent Content Failures with Two Voting Planes



Masking Inconsistent Content Failures with Two Voting Planes



The US space shuttle's computer system

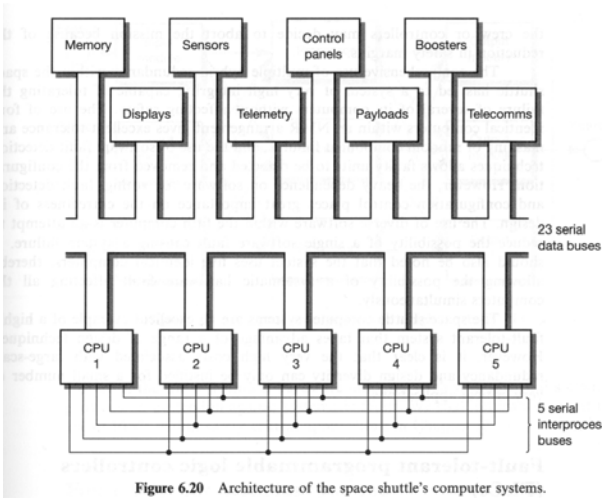


Figure 6.20 Architecture of the space shuttle's computer systems.

Main features of the Space Shuttle Computer

- Consist of five "off-the-shelf" computers with identical hardware
- Four computer executes critical functions in 4MR*
- The computers exchange results via special inter-processor buses and vote on the results in software
- One computer is a backup equipped with an independently developed**, stripped-down version of the critical flight control software
- The fifth computer performs non-critical functions under fault-free circumstances
- The computers send independent commands to actuators, which use hardware voting the mask errors in the commands

* 4MR = NMR, with N=4.
** Design diversity

Cracked diode in US Space Shuttle

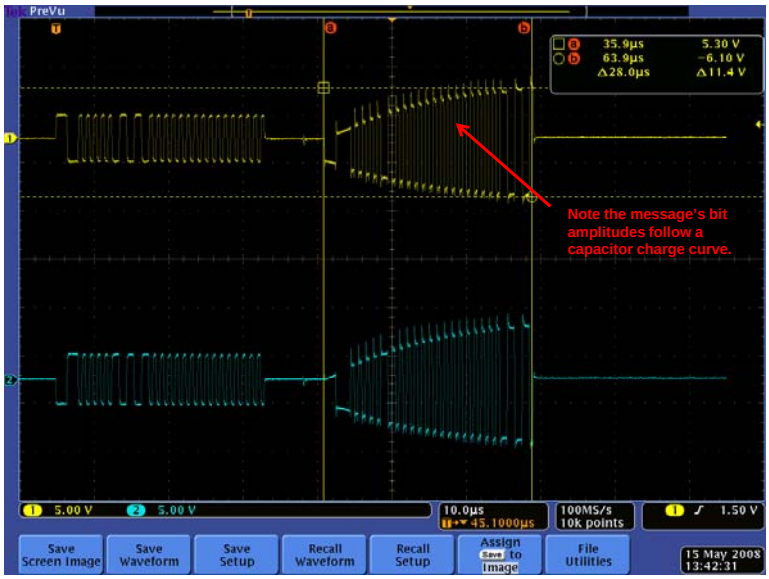
Figure 1. Two views (90 degrees apart) of a fissure that appears to go through the silicon dice - Red arrows.

A crack (fissure) through a diode appeared in a Space Shuttle while on the launch pad preparing for mission STS-124. At 12 minutes and 38 seconds past noon May 13, 2008 this caused a 3-1 split of the four computers that control the Shuttle. Three seconds later, the split became 2-1-1. But, none of the processors or their intercommunications were faulty. The fault was in a box that sends messages to the computers.

Source and text: Kevin R. Driscoll, Honeywell Aerospace, USA

Normal messages

Faulty Message on the Right



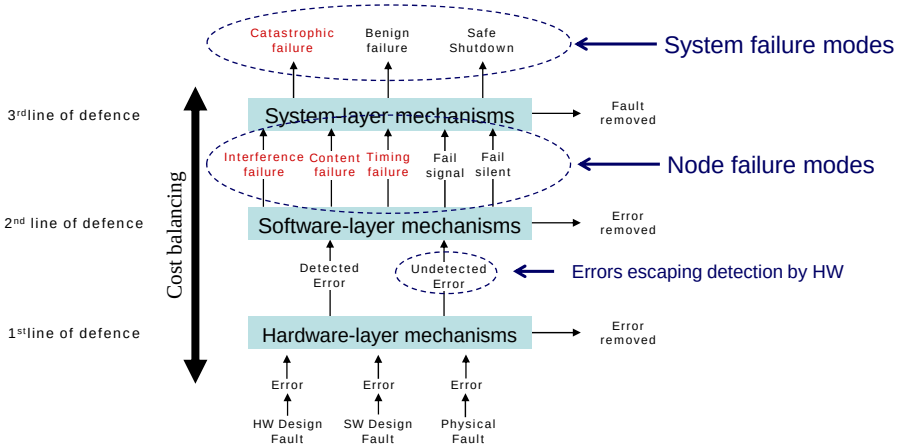
Outline

- Characterization of Failure Modes
- Byzantine failures
- Layered fault tolerance
- Error detection techniques
- Self-checking nodes

Layered fault tolerance

- Fault-tolerance is typical achieved by combining several mechanisms for error detection, error masking and system recovery located at different abstraction layers in a computer system
- The division of a system into layers can be done in different ways. We will use a simple model with three layers:
  - Hardware layer – mechanisms implemented in hardware either within an integrated circuit or by replication of integrated circuits within a node.
  - Software layer – mechanisms implemented in software dealing with errors occurring within a node
  - System layer – mechanisms implemented in software dealing with errors occurring in other nodes in the system, or in the system's communication network

Layered fault tolerance in distributed systems



Undesirable and severe failure modes are marked in red!

Purpose of different layers

- **Hardware layer** – serves as a first line of defense that should
  - Correct as many errors as is economically feasible
  - Detect most other errors
- **Software layer** – serves as a second line of defense that should
  - Correct most errors detected, but not corrected by the hardware layer.
  - Detect most errors that are not detected or corrected by the hardware layer
  - Ensure appropriate *failure semantics* for the node for any error that cannot be corrected. (Failure semantics is the same as failure mode assumptions.)
- **System layer** – serves as a third line of defense that should
  - Detect and correct any errors that are not detected or corrected by the software and hardware layers

Outline

- Characterization of Failure Modes
- Byzantine failures
- Layered fault tolerance
- Error detection techniques
- Self-checking nodes

Fault Detection and Error Detection

- Terminology
  - Both the terms *fault detection* and *error detection* are used in the literature, see discussion in the beginning of Chapter 6.4 in the course book
- We distinguish between
  - Concurrent (on-line) error detection
    - Detection of errors during operation
    - Purpose is to mask or minimize adverse effects of errors
  - Non-concurrent (off-line) fault detection
    - Testing to find physical hardware faults while the system is off-line
    - Purpose is to identify faulty hardware units
- We will focus on techniques for *concurrent error detection*

Off-line fault detection techniques

- Functionality checking
  - Examples:
    - Test of random access memory (RAM) by writing and reading back test patterns to all memory words
    - Test of CPU by running special test programs
- Loop back testing
  - Example: “echo” testing of communication paths

On-line error detection techniques mentioned in the course book (1)

- Duplication and comparison
  - Comparison of redundant signals
  - Self-checking pair
- Consistency checking
  - Uses a priori knowledge about information.
  - Examples:
    - Hardware exceptions in CPUs, e.g., division by zero, memory access to odd addresses.
    - Range checking in software of constrained program variables.
- Information redundancy
  - Use of error detecting and error correcting codes

On-line error detection techniques mentioned in the course book (2)

- Watchdog timer
- Bus monitoring
  - Checking the range of addresses generated by a CPU
  - Examples
    - Checking that CPU use an even address when reading a 32 or 64-bit word.
    - Checking CPU memory access using a memory management unit.
- Power supply monitoring

Watchdog timer

- Principle:
  - When a program starts to execute it initiates a hardware timer which is periodically reset by special instructions inserted into the program
  - If the program fails to reset the timer within a prescribed deadline, the timer generates an interrupt to the CPU to signal an error
- Detects slow programs and programs that hang in infinite loops.
- Very common in embedded real-time systems

Watchdog Timer as a Node Restart Mechanism

- Watchdog timer is often used to restart a node that has failed
- This simplifies error handling: whenever an error is detected by some mechanisms, the node will simply stop executing until it is restarted by the interrupt signal from the watchdog timer
- Restarting a node involves an elaborate set of actions, including recovering the node's view of the system state and reintegrating the node into the set of operational nodes through a node membership service. These actions are usually handle by system-layer mechanisms

End-to-end Checksums

- An end-to-end checksum protects the result of a computation from the producer to the consumers
- It is appended to the result by its producer and is checked by any consumer of that result
- It protects a result while it is being transferred from the producer to the consumers
- The producers and consumers are often application programs

Examples of error detection mechanisms at different layers

Hardware layer

- **CPU hardware exceptions:** *Bus error, Address error, Illegal opcode, Privilege violation, Division by zero, Spurious interrupt, etc.*
- **Error detecting and correcting codes in main memory, caches, internal buffers**
- **Special hardware circuits (often connected to the non-maskable interrupt signal of a CPU):** *Power supply monitor, Network (bus) guardian.*
- **Watchdog timer (sometimes implemented as combination of HW and SW)**

Software layer

- **Compiler:** *Type checking of constrained variables, Value range overflow, Loop iteration bound overflow*
- **OS:** *Processing time overflow, consistency checks on OS data,*
- **Application:** *time-redundant execution of tasks, application specific consistency checks*

System layer

- **End-to-end checksums**
- **Comparison of results produced by two nodes**
- **Voting on results produced by three or more nodes**

Exceptions in CPU:s

- Modern central processing units (CPUs) are equipped with hardware implemented error detection mechanisms called *hardware exceptions*
- The number and type of hardware exceptions varies depending on the CPU design
- When a hardware exception is raised, the CPU stops the program execution and jumps to an exception routine
- The handling of exceptions is very similar to how a CPU responds to interrupt signals
- Some examples of common hardware exceptions is given in the next two slides

Examples of CPU exceptions (1)

Bus error: detects errors during read and write accesses to the main memory. This exception is raised (triggered) when the CPU attempts to access an address to which no memory or any I/O device is connected.

Address error: detects when the CPU makes an attempt to access memory using an odd numbered address; only even numbered addresses are allowed in many CPUs.

Illegal opcode: detects if the CPU during an instruction fetch reads a value from memory (or the instruction cache) that doesn't correspond to a valid instruction. This error can occur if the program counter is erroneously loaded with an address pointing to a data area rather than a program code area.

Examples of CPU exceptions (2)

Privilege violation: detects if a user program attempts to execute an instruction which is allowed only for programs that execute in the superuser mode (privileged mode), such as the operating system or device drivers. User programs normally executes in user mode (normal mode).

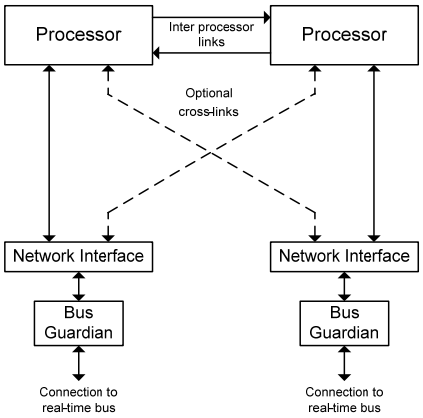
Division by zero: detects if a program tries to divide a number with zero.

Spurious interrupt: detects if an interrupt is signalled but no interrupt vector is provided by the interrupting device. (The interrupt vector tells the CPU which device it was that raised the interrupt signal and thereby indicates which interrupt service routine that the CPU shall execute.)

Outline

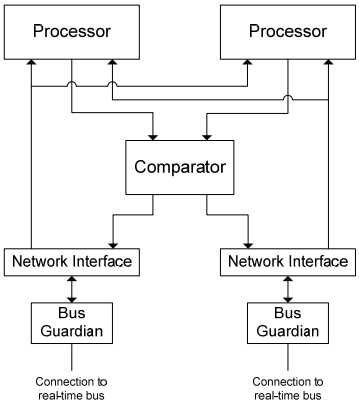
- Characterization of Failure Modes
- Byzantine failures
- Layered fault tolerance
- Error detection techniques
- Self-checking nodes

Self-checking node supporting software implemented message comparison



- The processors executes the same programs and exchange copies of outgoing messages via the inter processor links
- They compare the message copies and stops execution if the copies do not match.
- An error counter stores the number of mismatches that has occurred.
- The node is restarted after a mismatch only if the value of the error counter is below a predefined threshold
- The bus guardian protects the bus from erratic behavior (e.g., babbling idiot) of the network interfaces

Self-checking node with message comparison in hardware



- Processor failures are detected by duplication and comparison
- The processors produce replicated messages that are compared by the comparator.
- The network interfaces receive messages from the comparator and send them to other nodes via two redundant real-time busses.
- The payload in the messages are protected by end-to-end checksums added by the processors.
- End-to-end checksums can be used to ensure that faults in the comparator and network interfaces are detectable by the service users (other nodes).

Overview of Lecture 15

- Time-Triggered Systems
- Preparations:
  - Lecture slides
  - The Time-Triggered Architecture (see reading instructions)