Technical University of Cluj-Napoca
Computer Science Department

Distributed Systems

Reliable and Fault Tolerant Distributed Systems

Fall 2017

Contents

Part 1 – FT Basics

Introduction and Context

- DS are not fail free
- Fault Tolerance – ability of a system to continue performing its intended functions in presence of faults
- Objective: Techniques developed to add reliability and high availability to DS

Fault

- Physical defect, imperfection, or flaw that occurs in hardware or software units
- A fault is a source that has the potential of generating errors
- Examples: software bug, short circuit, broken pin
- Fault types
  - Component faults
  - Execution faults
  - Communication faults

Fault

- Note. The same fault can occur because of different causes, and hence classified differently
- Example
  - Consider
    - » entities **x** and **y** and a message **m** sent from x to y (according to the protocol)
    - » **m** never arrives
  - Fault possible causes:
    - » execution error: **x** fails to execute *send* operation in the protocol ();
    - » transmission error: loss of message by the transmission subsystem
    - » component failure: link (x, y) goes down

Fault
Fault types

- Based on fault duration
  - Transient faults (also called soft-error)
    - » Dominant type of faults in todays's ICs
    - » Occurs / disappears of its own - usually within short periods
    - » Happens once in a while - it may or may not re-occur
    - » *Intermittent transient faults* - continues to re-occur, not necessarily at regular intervals (difficult to diagnose)
  - Permanent faults
    - » Continue to exist until the fault is repaired
    - » Examples: software bugs, disk head crashes, etc.

Fault
Fault types

- Based on their geographical spread
  - Localized
    - » occur always in the same region of the system
  - Ubiquitous
    - » occur anywhere in the system
- **Note**s
  - Transient failures are usually ubiquitous
  - Intermittent and permanent failures tend to be localized

Fault
Fault types

- Danger of a fault
  - Not all faults are equally dangerous
  - Danger of a fault
    - » Not always related to the severity of the fault itself
    - » But rather in its consequences on the correct functioning of the system
- Danger of a fault (for the system) is intrinsically related to its **detect-ability**
  - If a fault is **easily detected**
    => A remedial action can be taken to limit the damage
  - If a fault is **hard or impossible to detect**
    => The effects of the initial fault may spread throughout the network creating possibly irreversible damage

Fault
Fault types

- Examples
  1. The **permanent fault** of **a link going down** forever is **more severe** than if that link failure is just transient
  2. In contrast, **the permanent failure** of the link might be **more easily detectable**, and thus can be taken care of, than the occasional malfunctioning (**transient fault**) of the link

Faults
Origins of faults

- Incorrect specification (specification faults)
- Incorrect implementation (design faults)
  - Ariane 5 rocket accident, June 4, 1996 exploded after 37 sec due to a software fault generated as a result of converting a 64-bit floating point to a 16-bit integer
- Hardware components defects
- External factors during system operation (sensors, temperature, radiation, etc. that affects the system inputs) as well as hackers' acts

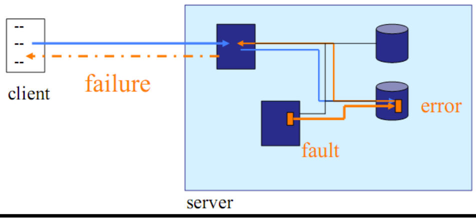
Faults
Origins of faults

- Software Faults
  - The main source of software-related failures is design faults
  - New faults may be introduced in software due to
    - » **Increasing reliability** through redundancy and replication **or**
    - » **Feature upgrades** during its life cycle

  => good intentions can have bad consequences

Error, Failure

- **Error**
  - An error is the manifestation of a fault
  - May happen in a program, a component or a system
  - Errors may occur at distance from fault location
- **System Failures**
  - Failures are caused by errors
  - When system failure occurs?
    - » A system failure occurs when the error due to a fault is causing the system to perform incorrectly one of its functions
  - A system may fail either because it does not act in accordance with the specification, or because the specification did not adequately describe its function



Error, Failure

- Faults are reasons for errors and errors are reasons for failures
- Example (a physical system)

  Steam–based power generation
  - Turbine sensor reporting turbine rotation speed breaks (**fault**) => system send more steam into turbine than is required (**error**) => over speeding the turbine => mechanical safety system stops the turbine to prevent damage => no more power generation (**system failure** of fail-safe type)
- In software systems
  - Fault - bug in a program
  - Error – a possible incorrect value assigned to a variable due to this bug
  - Failure – Possible crash of the application due to error

Error, Failure

- Failure effects
  - The actual or foreseen consequences of a failure
  - Need to be analyzed to identify corrective actions to be taken to mitigate the effect of the failure in the system (this is essential in safety-critical applications and should be considered in the system design stage)
- Failure Model
  - The way in which a system may fail
  - Different systems may fail in different ways (a lock may fail to be opened or to be closed)
  - Failure models are classified based on their domain, the perception of the failure by system users, the effects of the failure, etc.

Resiliency

- Assuming
  - The properties of the system and
  - The types of faults assumed to occur
- Resiliency – the maximum number of faults that can be tolerated

Fact
No DS computation (protocol) can be resilient to an arbitrary number of faults

=> The goal is to design DS computation (protocols) that are able to tolerate up to a certain amount of faults of a given type

Fault/Failure tolerance

- Objective
  - Design systems that tolerate faults **or**
  - Design systems that will behave correctly in spite of failures
- Main facilitator: **redundancy**
  - Redundancy - the addition of information, resource, or time beyond what is necessary for normal system operations

Fault/Failure tolerance

- Problem complexity
  - The unpredictability of fault occurrence
  - The nature of a fault
  - The possibility of multiple faults
- When talking about fault-tolerant protocols and fault-resilient computations
  - Always qualify the statements and clearly specify the **type** and **number of faults** that can be tolerated

Fault/Failure tolerance

- Fault detection
  - The process of determining that a fault has occurred within a system
  - Fault latency - the delay from the manifestation of the fault to the detection of the fault
  - Diagnosis - Obtaining information about a failure’s location and/or properties

Fault/Failure tolerance

- Fault detection techniques
  - Acceptance Tests (ATs) - can be used in systems without replicated components– failing the acceptance test indicates fault
  - Comparison detecting faults in systems with duplicated components (the output of two components is compared – a no match indicates fault)
  - Evaluating invariants
  - Self-checking systems
    - » Check own behavior against some pre-specified specifications to ensure doing the right things
    - » Parity, Checksums, or Cyclic Codes (Cyclic Redundancy Check, CRC)
  - Consistency checking
    - » Achieved by verifying that intermediate or final results are within a reasonable range
  - Capability checking
    - » Guarantees that access to objects is limited to users with proper authorization
    - » Usually part of the operating system
    - » Can also be implemented in hardware

Fault/Failure tolerance

- Fault masking
  - The process of insuring that only correct values get passed to the system output in spite of the presence of a fault
- Techniques
  - Correct the error (e.g. error correcting codes)
  - Compensate the error

Fault/Failure tolerance

- Fault containment
  - The process of isolating a fault and preventing the propagation of its effect throughout the system
- Techniques
  - Frequent fault detection,
  - Multiple request/confirmation protocols
  - Performing consistency checks between modules.

Fault/Failure tolerance

- Fault recovery and reconfiguration
  - Isolate the faulty component
  - Replace faulty component by redundant component
  - Switch off the faulty component and continue operation with a degraded capability (*graceful degradation*)
  - After the elimination of errors, the system operation is normally backed up (rolled back) to some point in its processing that proceeded the fault detection, usually using backup files, checkpoints, etc.
- Restart
  - When recovery is impossible or if the system is not designed for recovery
  - *Hot* restart - resumption of all operations from the point of fault detection
  - *Warm* restart - only some of the processes can be resumed without loss
  - *Cold* restart: complete reload of the system, with no processes surviving

Fault/Failure tolerance

- Fault prevention
  - Set of techniques attempting to prevent the introduction or occurrence of faults in the system
- Techniques for achieving fault prevention
  - Quality control techniques during the specification, implementation, and fabrication stages of the design process
  - For hardware: design reviews, component screening, and testing
  - For software: structural programming, modularization, and formal verification techniques

Dependability and FT

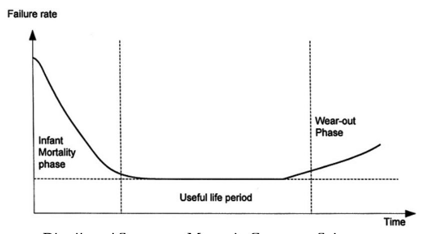
- Dependability
  - The ability of a system to deliver its intended level of service to its users
  - Measures the quality of the system delivered services
- Major goal of FT: development of dependable systems
- Fundamental properties of dependability:
  - (1) reliability;
  - (2) availability;
  - (3) safety;
  - (4) confidentiality;
  - (5) integrity and
  - (6) maintainability
- Note. Security is not included in the list (but covered by 2, 4 and 5)

Dependability and FT

- Informally, it is expected that a dependable system:
  - will be operational when needed (availability),
  - will keep operating correctly while being used (reliability),
  - there will be no unauthorized disclosure (confidentiality) or modification (integrity) of information the system is using,
  - operation of the system will not be dangerous (safety)

Dependability and FT
Metrics – Failure rate

- Failure Rate (λ)
  - The expected number of failures (of a device or system) per a given time period
  - The failure rate λ is measured in units of failures per hour (or per million hours)
  - Example: memory module failure rate: 7 failures per 1 million hours
- Failure rate as function of time
  - **Bath-tube curve** – failure rate function of time for hardware components (see Figure)



Dependability and FT
Metrics – Failure rate

- During the useful life phase of the system, the failure rate is assumed to have a constant value λ.
- Then, the reliability of the system decreases exponentially with time (exponential failure law)

$$R(t) = e^{-\lambda t}$$

- If failure rates of system components are available, raw estimate of the failure rate of a system without redundancy during its useful life phase can be obtained adding up the failure rates of the components:

$$\lambda = \sum_{i=1}^{n} \lambda_i$$

Dependability and FT
Metrics – Failure rate

Example

- A logic circuit with no redundancy consists of 16 two-input NAND gates and three flip-flops
- NAND gates and flip-flops have constant failure rates of 0.311 × 10^(-7) and 0.412 × 10^(-7) per hour, respectively.
- Assuming that component failures are independent events
- The failure rate of the logic circuit is
- *λ*circuit = 16*λ(*AND) + 3*λ(*FF) = 0.6212 × 10^(-6) per hour
- The reliability of the logic circuit for a 1-year mission
  R(t) = *e^(-λt)* = *e^((*-0.6212)×10-6×8760) = 0.9946.
  => The reliability of the circuit for 1 year mission is 00.46% (1 year = 8760 hours)

Dependability and FT
Metrics – Failure rate

Case of software systems

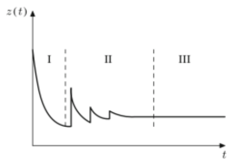
- In software systems, failure rate usually decreases as a function of time (as opposite to hardware)
- As a generalization, time-varying failure rate functions can be modeled using *Weibull distribution*:
  z(t) = *αλ(λt)^(α*-1)

where *α* and *λ* are constants determining the behavior of *z(t)* over time

1. If *α* = 1, then *z(t)* = *λ*.
2. If *α* > 1, then *z(t)* increases with time.
3. If *α* < 1, then *z(t)* decreases with time

Note. For software systems *α* < 1

- A typical evolution of failure rate in a software system is shown in figure. The three phases of the evolution are: (I) test/debug, (II) useful life (II), and (III) obsolescence.



Dependability and FT
Metrics – Failure rate

Hardware vs software failure rates
Major differences

- (1) In the useful-life phase, software usually experiences periodic increases in failure rate due to feature upgrades.
  - Since a feature upgrade enhances the functionality of software, it also increases its complexity, which makes the probability of faults higher.
  - After a feature upgrade, the failure rate decreases gradually, due to the bugs found and fixed.
- (2) The second difference is that, in the last phase, the software failure rate does not increase.
  - In this phase, the software is approaching obsolescence. Therefore, there is no motivation for more upgrades or changes.

Dependability and FT
Metrics - Reliability

- Reliability
  - The conditional probability that a system survives for the time interval [0, t], given that it was operational at time t=0
    R(t) = P { 0 failures in [0, t] | no failure at t = 0}
- Consider:
  - $N_0(t)$ be the number of components that are operating correctly at time t
  - $N_f(t)$ be the number of components that have failed at time t
  - N, the the number of components that are in operation at time t
- Reliability R as a function of t

$$R(t) = \frac{N_o(t)}{N} = \frac{N_o(t)}{N_o(t) + N_f(t)}$$

- Unreliability Q

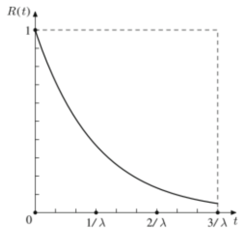
$$Q(t) = \frac{N_f(t)}{N} = \frac{N_f(t)}{N_o(t) + N_f(t)}$$

- Invariant: R(t) = 1 – Q(t), for any t

Dependability and FT
Metrics - Reliability

- Reliability as exponential failure law
- Assuming
  - The system is in the useful-life stage
  - The failure rate function has a constant value of λ (which is usual in useful-life stage)
- Exponential failure law - The reliability function is an exponential function of parameter λ

$$R(t) = e^{-\lambda t}$$



Dependability and FT
Metrics – Availability

- Availability
  - Percentage calculated as the operational time per total time considered
- Availability is the intuitive sense of reliability - a system is reliable if it is able to perform its intended function at the moment the function is required
- Metrics related to availability
  - Mean Time to Failure (MTTF),
  - Mean Time to Repair (MTTR),
  - Mean Time Between Failures (MTBF)
- See more about availability related topics in the course "*Features of Distributed Algorithms*"

Dependability and FT
Metrics – Safety

- Safety as an extension of reliability
  - Reliability in respect to failures that may generate safety hazards
- From reliability point of view, all failures are equal
- For safety considerations failures are of two categories
  - Fail-safe (ex. An alarm system may give a false alarm when no danger is present)
  - Fail-unsafe (ex. An alarm system may fail to operate correctly even if a danger exists)
- Formal definition
  - *Safety S(t)* of a system at time *t* is the probability that the system either performs its function correctly or discontinues its operation in a fail-safe manner in the interval [0, *t*], given that the system was operating correctly at time 0.
- Safety is required in *safety-critical applications* where a failure may result in human injury, loss of life, or environmental disaster

Dependability and FT
Metrics – Fault Coverage

- Fault coverage
  - A measure of a system's ability to perform fault tolerance
- Types of fault coverage – measuring system ability to detect, locate, contain and recover from faults:
  - fault detection,
  - fault location,
  - fault containment (limits faults to certain boundaries)
  - **fault recovery coverage** (the most commonly considered).

Dependability and FT
Metrics – Fault Coverage

- Fault detection coverage
  - The conditional probability that, given the existence of a fault, the system detects it

  C = P(faultDetection | faultExistence)
- Fault location coverage
  - The conditional probability that, given the existence of a fault, the system locates it

  C = P(faultLocation | faultExistence)
- Fault containment coverage
  - The conditional probability that, given the existence of a fault, the system contains it

  C = P(faultContainment | faultExistence)
- **Fault recovery coverage** (C) - The conditional probability that, given the existence of a fault, the system recovers :

  C = P(faultRecovery | faultExistence)

Part 2 – Modeling DS Faults

3. Modeling DS Faults

- DS failure models differ by the factor responsible for failure
  - **Component failures** models
  - **Communication failure** models

3. Modeling Faults
Component Failures Models

- Model assumptions
  - Only components can fail
  - If something goes wrong, it is due to an involved component that is faulty

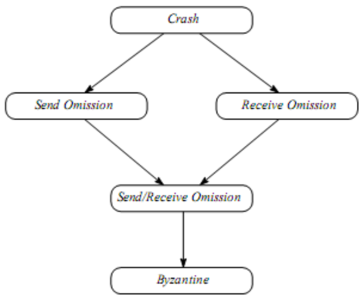
- Types of component failure models (depending on which components are blamed)
  - Entity (or Node) Failure (EF) models,
  - Link Failure (LF) models and
  - Hybrid Failure (HF) models

3. Modeling Faults
Component Failures Models
EF Model

- In the **Entity Failure (EF)** model **only system nodes** can fail
- Examples
  - If a node crashes (for whatever reason) => that node will be declared faulty
  - The node is also declared faulty for related incidents
    - » A link going down
      - ◆ Modeled by declaring one of the two incident nodes to be faulty
      - ◆ That node will lose all the message to and from its neighbors
    - » The corruption of a message during transmission
      - ◆ One of the two incident nodes will be declared to be faulty

3. Modeling Faults
Component Failures Models
EF Model

- In this model only entities (nodes) can fail
- Hierarchy of EF failures (figure, source [7])



Hierarchy of EF failures

3. Modeling Faults
Component Failures Models
EF Model (cont.)

- Crash Faults
  - A faulty entity works correctly according to the protocol, then suddenly just stops any activity (processing, sending, and receiving messages)
  - These are also called **fail-stop faults**
    - » The most benign from the overall system point of view

3. Modeling Faults
Component Failures Models
EF Model (cont.)

- Send/receive omission faults
  - A faulty entity occasionally
    - » Loses some received messages or
    - » Fails to send some of the prepared messages
  - May be caused by buffer overflows
  - More difficult to detect such faults than crash faults
- **Note**. Crash faults are particular cases of this type of failure
  - A crash is a send/receive omission in which all messages sent to and from that entity are lost

3. Modeling Faults
Component Failures Models
EF Model (cont.)

- Byzantine faults
  - A faulty entity is not bound by the protocol and can perform any action such as:
    - » Omit to send or receive any message,
    - » Send incorrect information to its neighbors, behave maliciously so as to make the protocol fail
  - Undetected software bugs often exhibit Byzantine faults
  - Dealing with Byzantine faults is much more difficult than dealing with the previous ones

3. Modeling Faults
Component Failures Models
LF Model

- In the **Link Failure (LF)** model, **only links** can fail
- Examples
  - If a message over a link is lost
    => that link being declared faulty
  - In this model, the crash of a node is modeled by the crash of all its incident links
  - If an entity computed some incorrect information (because of a execution error) and sent it to a neighbor => the link connecting the entity to the neighbor will be blamed
    - » in particular, the link will be declared to be responsible for corrupting the content of the message

3. Modeling Faults
Component Failures Models
HF Model

- In the **Hybrid Failure (HF)** model, both links and nodes can be faulty
- Although more realistic, this model is little known and seldom used

- **NOTE**
  - In all three component failure models, the status faulty is permanent and is not changed
    - » In other words, once a component is marked with being faulty, that mark is never removed
  - For example, in the link failure model, if message is lost on a link, that link will be considered faulty forever, even if no other message will ever be lost there

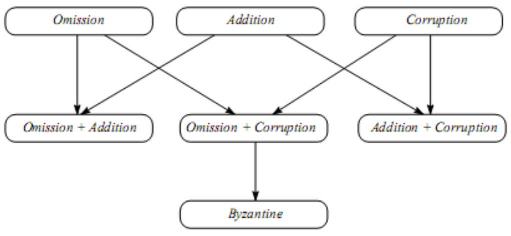
3. Modeling Faults
Communication Failures Models

- Communication Failure Model is also known as **Dynamic Fault (DF)** model
- In this model, only the communication system can be faulty
  - The communication system can lose, corrupt, and deliver to an incorrect neighbor
  - A component fault (such as the crash failure of a node) is modeled by the communication system losing all the messages sent to and from that node

3. Modeling Faults
Communication Failures Models

- In the communication failure model, the communication subsystem can cause three types of faults:
  - Omission
    - » A message sent by an entity is never delivered
  - Addition
    - » A message is delivered to an entity, although none was sent (noise, unauthorized message, spam, etc.)
  - Corruption
    - » A message is sent but one with different content is received

3. Modeling Faults
Communication failures models



Hierarchy of combination of faults types in the DF model (source [7])

3. Modeling Faults

- **Note**
- No protocol can tolerate any number of faults of any type
  - if the entire system collapses, no computation is possible)

  => when discussing about
  - fault-tolerant protocols and
  - fault-resilient computations,

  clearly specify
  - the type and
  - number of faults that can be tolerated

Part 3 – FT Main Facilitators and Techniques

-Stable Storage

-Fail-Stop Processors

-Atomic Actions

-Redundancy

Stable Storage

- Stable storage
  - logical abstraction for a special storage that can survive system failure
  - contents of stable storage are not destroyed or corrupted by a failure
- In an ideal stable storage
  - *read* always returns good data and
  - *write* always succeeds
- Key issue in implementing a stable storage
  - correctly execute the basic operations of *read* and *write* through which a process can interact with storage media such as disks
- Example
  - **Disk shadowing** uses two disks which can also handle errors like disk failure
  - Use of redundant arrays of inexpensive disks (**RAID**)
    - » Data are spread over multiple disks using bit-interleaving to provide high I/O performance
    - » RAID - significant advantages over conventional disks and can tolerate multiple failures

Fail-stop processors

- A **fail-stop** processor
  - Automatically halts in response to any internal failure and
  - Does so before the effects of that failure become visible
  - When a processor fails it simply ceases to function and most likely it does not perform any incorrect action
- The effects of a fail-stop processor:

  (1) The processor stops execution

  (2) The volatile storage is lost but the stable storage is unaffected

  (3) Any processor can detect the failure of a fail-stop processor

- Methodologies for the design of FT DS based on the fail-stop processors are developed in the literature [see Schlichting (Univ Arizona) & Schnider (Cornell Univ)]

Atomic actions

- Atomic action
  - A set of operations which are executed indivisibly
    - » Either operations are completed successfully or
    - » The state of the system remains unchanged (operations are not executed at all)
- All-or-nothing property of atomic actions
  - The process executing the action is not aware of the existence of any outside activities and cannot observe any outside state change
  - Similarly, any outside processes cannot view any internal state change of an atomic action
  -

Redundancy

- **Redundancy** can be of:
  - (1) space redundancy or time redundancy
  - (2) static or dynamic redundancy
- **Space redundancy** - Provides additional components, functions, modules, data items that are not necessary for fault-free operation
  - Space redundancy can be of **hardware, software or information types**
  - **Hardware redundancy** (for example, extra PEs, I/Os)
  - **Software redundancy** (for example, extra versions of software modules)
  - **Information redundancy** (extra bits used by error detecting codes)
- **Time redundancy** – computation or data transmission is repeated and the result is compared to a stored copy o the previous result

Redundancy

- **Static redundancy**
  - Uses redundant components to mask the effects of hardware or software failures within a given module
  - The output of the module remains unaffected, i.e., error free, as long as the protection due to redundancy is effective
- **Dynamic redundancy**
  - Allows errors to appear at the output of modules
  - When a fault is detected, a recovery action eliminates or corrects the resulting error
  - Dynamic redundancy implies
    - » fault detection,
    - » fault location and
    - » recovery processes

Hardware Redundancy

- Providing two or more physical copies of a hardware component
  - redundant processors,
  - memories,
  - buses,
  - communication channels,
  - power supplies, etc.
- Penalties due to HW redundancy:
  - Increase in weight, size, power consumption, cost, time to design, fabricate, and test
  - Increase in complexity
- Types of hardware redundancy:
  - passive,
  - active
  - hybrid.

Hardware Redundancy

Passive redundancy

- Achieves fault tolerance by masking the faults that occur without requiring any action from the system or an operator

Active redundancy

- Requires a fault to be detected before it is tolerated.
- After fault detection, the actions of location, containment and recovery are performed to remove the faulty component from the system.
- Active techniques require that a system is stopped and reconfigured to tolerate faults.

Hybrid redundancy

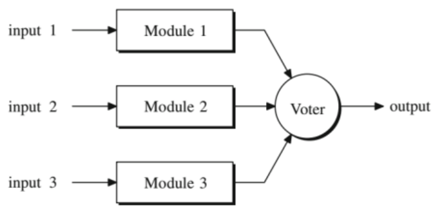
- Combines passive and active approaches.
- Fault masking is used to prevent generation of erroneous results.
- Fault detection, location, and recovery are used to replace the faulty component with a spare.
- Allows reconfiguration with no system downtime

Hardware Redundancy
Passive Redundancy

- Achieves fault tolerance by masking the faults that occur without requiring any action from the system or an operator
- Masking insures that only correct values are passed to the system output in spite of the presence of a fault.
- Passive redundancy techniques are usually used in high-reliability applications in which even short interruptions of system operation are unacceptable, or in which it is not possible to repair the system.
- Examples of such applications: aircraft flight control systems, embedded medical devices such as heart pacemakers, and deep-space electronics
- TMR (Triple Modular Redundancy) – the most used technique

Hardware Redundancy
Passive Redundancy
TMR

- The basic configuration is shown in Figure.
- The components are triplicated to perform the same computation in parallel
- Majority voting is used to determine the correct result
- If one of the modules fails, the majority voter masks the fault by recognizing as correct the result of the remaining two fault-free modules

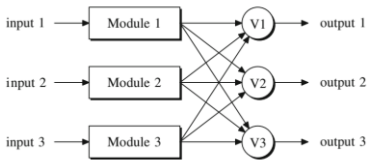


Hardware Redundancy
Passive Redundancy
TMR

- Depending on the application, the triplicated modules can be processors, memories, disk drives, buses, network connections, power supplies, and so on
- A TMR system can mask only one module fault
- A failure in either of the remaining modules would cause the voter to produce an erroneous result
- The dependability of a TMR system can be improved by removing failed modules from the system
- TMR is usually used in applications where a substantial increase in reliability is required for a short period
- Examples: (1) The logic section of the launch vehicle digital computer of the NASA Saturn V rocket (1967-73) (2) enhance device tolerance to transient faults caused by radiation in FPGAs)

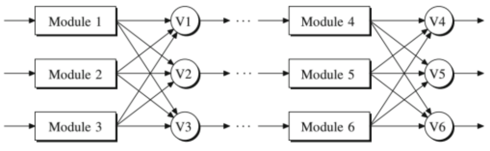
Hardware Redundancy
Passive Redundancy
TMR

- TMR main drawback - single point of failure
- TMR improvements – schemes with redundant voters
- See figure with 3voters
  - Requires consensus algorithms (exchange rounds of messages to establish consensus)



Hardware Redundancy
Passive Redundancy
TMR

- Multiple stages of TMR with triplicated voters (used in Boeing 777 aircraft to protect all essential hardware resources, including computing systems, electrical power, hydraulic power, and communication paths)
- Other techniques:
  - Replace a failed voter with a standby voter (additional fault-detection unit is required to detect primary voter failure)



Hardware Redundancy
Passive Redundancy
N Modular Redundancy (NMR)

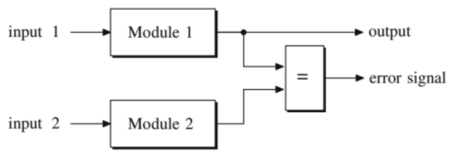
- N-modular redundancy (NMR) approach is based on the same principle as TMR, but it uses n modules instead of three

Hardware Redundancy
Active Redundancy

- Achieves fault tolerance by first detecting the faults which occur, and then performing the actions needed to return the system back to an operational state.
- Active redundancy techniques are typically used in applications requiring high availability, such as time-shared computing systems or transaction processing systems, where temporary erroneous results are preferable to the high degree of redundancy required for fault masking.
- Infrequent, occasional errors are allowed, as long as the system returns back to normal operation within a specified period of time.
- Techniques:
  - Duplication with comparison,
  - Standby
  - Pair-and-a-spare.

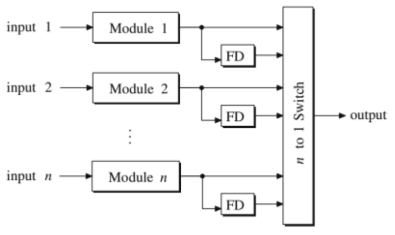
Hardware Redundancy
Active Redundancy
Duplication with Comparison

- Two identical modules operate in parallel.
- Their results are compared using a comparator.
- If the results disagree, an error signal is generated.
- Can detect only one module fault.
  - After the fault is detected, no actions are taken by the system to return back to the operational state.
- Depending on the application, the duplicated modules can be processors, memories, busses, etc.



Hardware Redundancy
Active Redundancy
Standby

- The basic configuration is shown in Figure.
- One of the *n* modules is active. The remaining *n*-1 modules serve as spares (or back-ups).
- A switch monitors the active module and switches operation to a spare if an error is reported by a Fault-Detection (FD) unit
- Types of standby techniques:
  - hot standby and
  - cold standby



Hardware Redundancy
Active Redundancy
Standby

Hot Standby
- Both operational and spare modules are powered up.
- Thus, spares can be switched into use immediately after the active module has failed.
- This minimizes the downtime of the system due to reconfiguration.

Cold standby
- The spares are powered down until they are needed to replace a faulty module.
- Increases reconfiguration time by the amount of time require to power and initialize a spare.
- Since spares do not consume power while in the standby mode, such a trade-off is preferable for applications where power consumption is critical, e.g., satellite systems

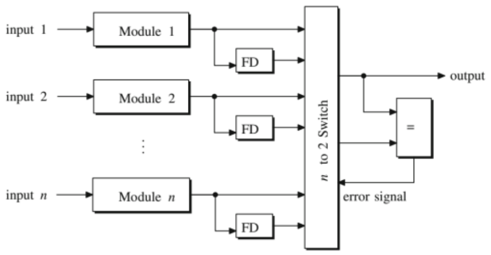
Hardware Redundancy
Active Redundancy
Standby

- A standby redundancy with *n* modules can tolerate *n* - 1 module faults.
- Tolerate = (system detects and locates a fault) + (recovers from fault) + (continues normal operation)
- The *n*th fault is detected but not tolerated – no more spares

Usage examples
- NonStop Himalaya server (from Tandem Computers)
  - Cluster of processors working in parallel
  - Each processor has its own memory and copy of the OS
  - A primary process and a spare process are run on separate processors.
  - Spare process mirrors all information from primary and is able to take over in the case of primary failure
- Apollo spacecraft telescope mount pointing computer used in Skylab American space station
  - Two identical computers, an active and a spare, are connected to a switch that monitors the active computer and shifts operation to the spare in the case of a malfunction.

Hardware Redundancy
Active Redundancy
Pair-and-a-Spare

- Combines the standby redundancy and the duplication and comparison approaches.
- Similar to standby redundancy except that two active modules instead of one are operated in parallel.
- As in the duplication with comparison case, the results are compared to detect disagreement.
- If an error signal is received from the comparator, the switch analyzes the report from the FD blocks and decides which of the two modules is faulty
- The faulty module is removed from operation and replaced with a spare module



Hardware Redundancy
Active Redundancy
Pair-and-a-Spare

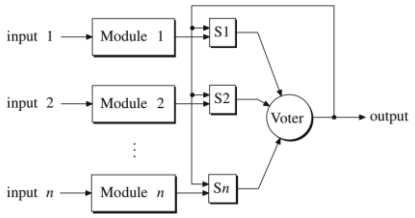
- A pair-and-a-spare system with *n* modules can tolerate *n*-2 module faults.
- When the *n* - 1st fault occurs, it will be detected and located by the switch and the correct result will be passed to the system's output.
- However, since there will be no more spares available, the switch will not be able to replace the faulty module with a spare module. Therefore, the system will not be able to continue its normal operation
- It is possible to design an extended version of the pair-and-a-spare system in which the system is reconfigured to operate with one active module after n − 1st fault.
- Upon receiving a signal that there are no spares left, the comparator can be disconnected and the switch can be modified to an n-to-1 switch which receives an error signal from the fault detection unit, as in the standby redundancy case.
- Such an extended pair-and-a-spare system would tolerate n − 1 module faults and detect n module faults

Hardware Redundancy
Hybrid Redundancy

- Hybrid redundancy combines advantages of passive and active approaches.
- Fault masking is used to prevent the system from producing momentary erroneous results.
- FD, location, and recovery are used to reconfigure a system when a fault occurs.
- Hybrid redundancy techniques are usually used in safety-critical applications such as control systems for chemical processes, nuclear power plants, weapons, medical equipment, aircrafts, trains, automobiles, etc.
- Techniques
  - Self-purging redundancy
  - N-modular redundancy with spares

Hardware Redundancy
Hybrid Redundancy
Self-purging Redundancy

- Self-purging redundancy consists of *n* identical modules which perform the same computation in parallel and actively participate in voting
- The output of the voter is compared to the outputs of each individual module to detect disagreement.
- If a disagreement occurs, the switch opens and removes, or *purges*, the faulty module from a system
- The voter is designed as a *threshold gate*, capable of adapting to the decreasing number of inputs



Hardware Redundancy
Hybrid Redundancy
Self-purging Redundancy

Threshold gate operation

- Each input $x1$, $x2$, . . . , xn has a weight $w1$, $w2$, . . . , wn.
- The output f is determined by comparing the sum of weighted input values to some threshold value, T (addition and multiplication are regular arithmetic operations):

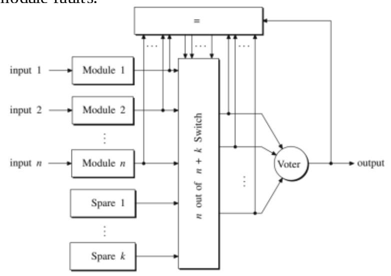
f = 1, if sum(wi * xi) >= T

f = 0, otherwise

- The threshold voter removes a faulty module from voting by forcing its weight to zero. In this way, faulty modules do not contribute to the weighted sum
- Such a system with n modules can mask n - 2 module faults.
- When n - 2 modules are purged and only two are left, the system will be able to detect the next, n - 1th fault, but, as in the duplication with comparison case, the voter will not be able to distinguish which of the two results is correct.

Hardware Redundancy
Hybrid Redundancy
N-Modular Redundancy with Spares

- In an N-modular redundancy with k spares, n modules operate in parallel to provide input to a majority voter
- Other k modules serve as spares.
- If one of the primary modules becomes faulty, the switch replaces the faulty module with a spare one.
- Various techniques are used to identify faulty modules.
- One approach is to compare the output of the voter with the individual outputs of the modules, as shown in Figure. A module which disagrees with the majority is declared faulty. After the pool of spares is exhausted, the disagreement detector is switched off
- and the system continues working as a passive NMR system. Thus, N-modular redundancy with k spares can mask |_n/2_| + k module faults.



Software Redundancy

- Software FT techniques can be of two types:
  - Single version
  - Multi version
- Single-version techniques target the improving FT of a software component by adding to it mechanisms for fault detection, containment, and recovery.
- Multi-version techniques use redundant software components which are developed following design diversity rules
  - Redundancy apply to procedure, class, process, whole system
  - Components with low reliability are made redundant
  - Redundancy generates complexity that has to be managed

Software Redundancy
Single version techniques

- The functional capabilities added to a software component by the single-version techniques are not necessary in a fault-free environment.
- The software structure and its actions are modified to allow for
  - Fault detection
  - Fault location of a fault,
  - Preventing fault propagation through the system
- This description contains techniques for
  - Fault detection,
  - Fault containment
  - Fault recovery in software systems.

Software Redundancy
Single version techniques
Fault detection techniques

- Objective - determine if a fault has occurred within a system
- Usually uses different *acceptance test*s to detect faults
  - If the program passes the test, it continues execution.
  - A failed test indicates a fault
- *Timing checks* are applicable to programs whose specification includes timing constraints.
  - Based on these constraints, checks can be developed to indicate a deviation from the required behavior.
  - A *watchdog timer* is an example of a timing check.
  - Watchdog timers can be used to monitor the performance of a program and detect lost or locked out modules
- *Reasonableness checks* use semantic properties of data to detect faults.
  - Example. A range of data can be examined for overflow or underflow to indicate
    a deviation from the system's requirements
- *Structural checks and Invariants* are based on known properties of data structures.
  - For example, a number of elements in a list can be counted, and also count the links (they should match). The counter could be a redundant data to the data structure

Software Redundancy
Single version techniques
Fault containment techniques

- It can be achieved by modifying the structure of the system and by imposing a set of restrictions defining which actions are permissible within the system.
- Common techniques:
  - modularization,
  - partitioning,
  - system closure, and
  - atomic actions

Software Redundancy
Single version techniques
Fault containment techniques

Modularization

- Prevent the propagation of faults by
  - decomposing a system into modules,
  - eliminating shared resources, and
  - limiting the amount of communication between modules to carefully monitored messages
- Before an evaluation needs to be performed to determine which module possesses the highest potential to cause the system failure

System closure

- The technique is based on the principle that no action is permissible unless explicitly authorized

Software Redundancy
Single version techniques
Fault containment techniques

Atomic actions

- Atomic action among a group of components - is an activity in which the components interact exclusively with each other
  - There is no interaction with the rest of the system for the duration of the activity
  - The participating components neither import nor export any type of information from nonparticipating components of the system.
- An atomic action can have two possible outcomes: either it terminates normally, or it is aborted upon a fault detection
  - If an atomic action terminates normally, its results are correct
  - If a fault is detected, then this fault affects only the participating components

Software Redundancy
Single version techniques
Fault recovery techniques

- Once a fault is detected and contained, a system attempts to **recover** from the faulty state and **regain operational status**
- If fault detection and containment mechanisms are implemented properly, the effects of the faults are contained within a particular set of modules at the moment of fault detection

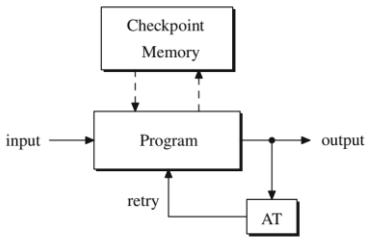
Main techniques

- Checkpoint and restart
  - Static checkpoints
  - Dynamic checkpoints
    - » Backward recovery (Rollback in DBs) – see details in the next Lecture
- Roll-forward recovery (look-ahead execution with rollback)
- Process pairs
- Exception handling

Software Redundancy
Single version techniques
Fault recovery techniques

Checkpoint and restart

- Most of the software faults are design faults, activated by some unexpected input sequence
- See figure
  - AT (Acceptance Test) – checks the acceptance of the results
  - If a fault is detected, a "retry" signal is sent to the module to re-initialize its state to the checkpoint state stored in the checkpoint memory



Software Redundancy
Single version techniques
Fault recovery techniques

Checkpoint and restart (cont.)
- Two types of checkpoints: static and dynamic

- **Static Checkpoints**
- Takes a single snapshot of the system state at the beginning of the program execution and stores it in the memory
- Fault-detection checks are usually placed at the output of the module.
- If a fault is detected, the system returns to the stored state and starts the execution from the beginning

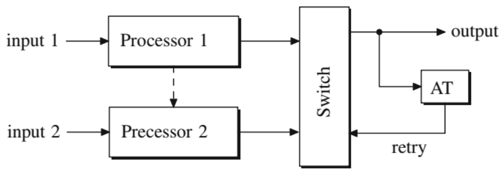
Software Redundancy
Single version techniques
Fault recovery techniques

Checkpoint and restart (cont.)
- **Dynamic checkpoints**
- Checkpoints are created dynamically at various points during the execution.
- If a fault is detected, the system returns to the last checkpoint and continues the execution.
- Fault-detection checks need to be embedded in the code and executed before the checkpoints are created
- When to take dynamic checkpoints
  - *Equidistant* - at deterministic fixed time intervals. The time between checkpoints is chosen depending on the expected fault rate.
  - *Modular* - at the end of the submodules in a module, after the fault-detection checks for the submodule are completed. The execution time depends on the distribution of the submodules and expected fault rate.
  - *Random*.

Software Redundancy
Single version techniques
Fault recovery techniques

Process Pairs

- Runs two identical versions of the software on separate processors (primary and secondary)
- Main advantage – provides service operation after the occurrence of the fault
- See figure
  - Primary Processor 1 is active. Executes the program and sends the checkpoint data to the secondary processor Processor 2
  - If fault is detected by AT module, Primary Processor is switched off
  - Secondary processor loads the last checkpoint data and continues execution
  - Meanwhile Processor 1 may execute diagnostic tests and recover. After recovery it may become the secondary

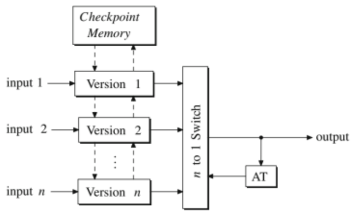


Software Redundancy
Multi-version techniques

- Uses multiple versions of the same software component, which are developed following design diversity rules
  - Examples: different teams, different coding languages, or different algorithms can be used to maximize the probability that different versions do not have common faults.
- Main multi-version techniques
  - Recovery blocks,
  - *N*-version programming,
  - *N* self-checking programming

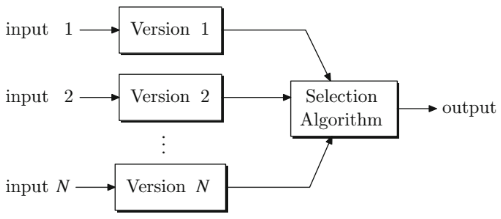
Software Redundancy
Multi-version techniques
Recovery Blocks

- Applies checkpoint and restart to multiple versions of a software module
- Versions 1 .. *n* are functionally equivalent versions of the same module
- At start, Version 1 generates system's output.
- If error detected by AT
  => Retry signal is sent to the switch.
  => The system is rolled back to the last state stored in the checkpoint memory
  => the execution is switched to the next version.
- Checkpoints - created before a new version executes.
- If all *n* versions are tried and none of the results are accepted => the system fails



Software Redundancy
Multi-version techniques
N-Version Programming

- Concurrent execution of n different software implementations (language, tool sets, environments (if possible) of the same module
- All versions are functionally equivalent.
- The selection algorithm decides which of the answers is correct and returns this answer as a result of the program execution.
- The selection algorithm - implemented as a generic voter.
  - Advantage over the recovery blocks fault-detection mechanism, which requires an application-dependent acceptance test
  - Different voting types can be considered: majority, generalized median, weighted averaging, etc.



Software Redundancy

Multi-version techniques
N Self-checking Programming

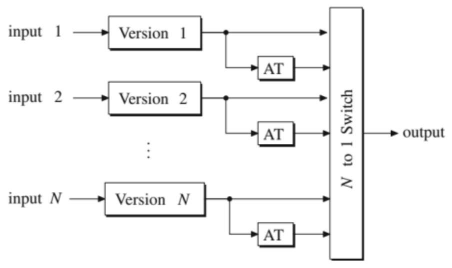
- Combines recovery blocks and *N* version programming
- The checking is performed either
  - By using acceptance tests (fig 1), or
  - By comparing pairs of modules (Fig 2)
- Examples of applications:
  - The 5ESS Switching System (which services approximately half of all US telephone exchanges) and
  - The Airbus A-340 flight control computer
-

Software Redundancy

Multi-version techniques
N Self-checking Programming

Checking by ATs

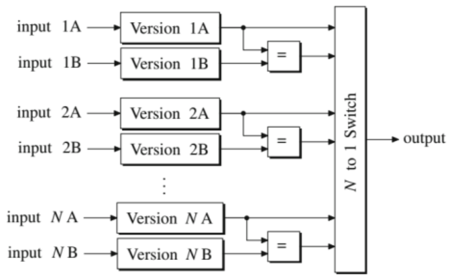
- Different versions of a software module and the ATs are developed independently from a common specification.
- The individual checks for each version are either embedded in the code, or placed at the output of the modules.
- The use of separate acceptance tests for each version is the main difference from the recovery block approach.
- The execution of each version can be done serially, or concurrently. In both cases, the output is taken from the highest-ranking version which passes its acceptance test.



Software Redundancy
Multi-version techniques
N Self-checking Programming

Checking by comparison

- Advantage over the *N* self-checking programming using ATs is that an application-independent decision algorithm is used for fault detection.



Software Redundancy
Multi-version techniques
Design diversity considerations

- Design diversity to assure independence between the different versions of a software module – critical issue in multi-version techniques
- Design diversity aims to protect multiple versions of a module from common design faults
- Software systems are vulnerable to common design faults if they implement the same algorithm, use the same program language, or if they are developed by the same design team, tested using the same technique, etc.
- Design decisions to be made when developing a multi-version software system include:
  - which modules to be made redundant;
  - the level of redundancy (procedure, process, class, and whole system);
  - the required number of redundant versions;
  - the required diversity (diverse specification, algorithm, code, programming language, team, testing technique, etc.)
  - rules of isolation between the development teams, to prevent the flow of information that could result in common design error
  - evaluate the cost (which is usually high)