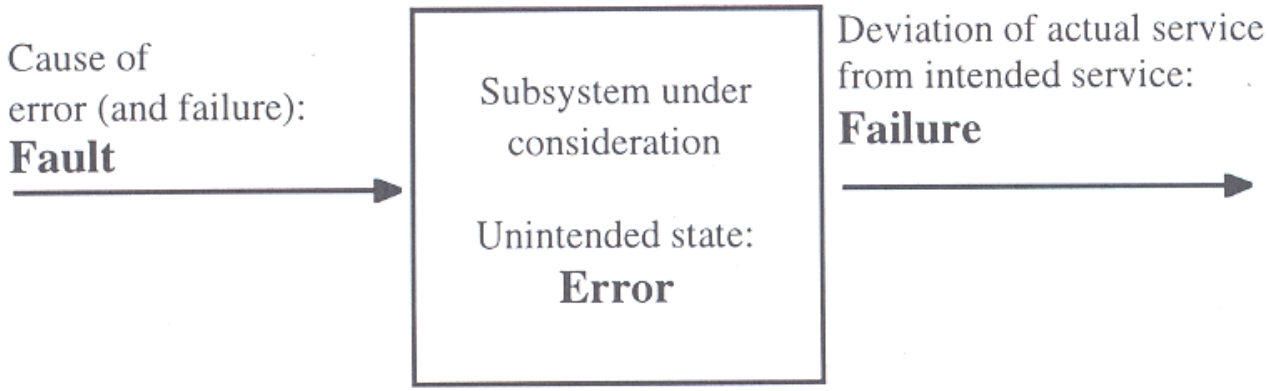
Chapter 6
Fault Tolerance

Overview

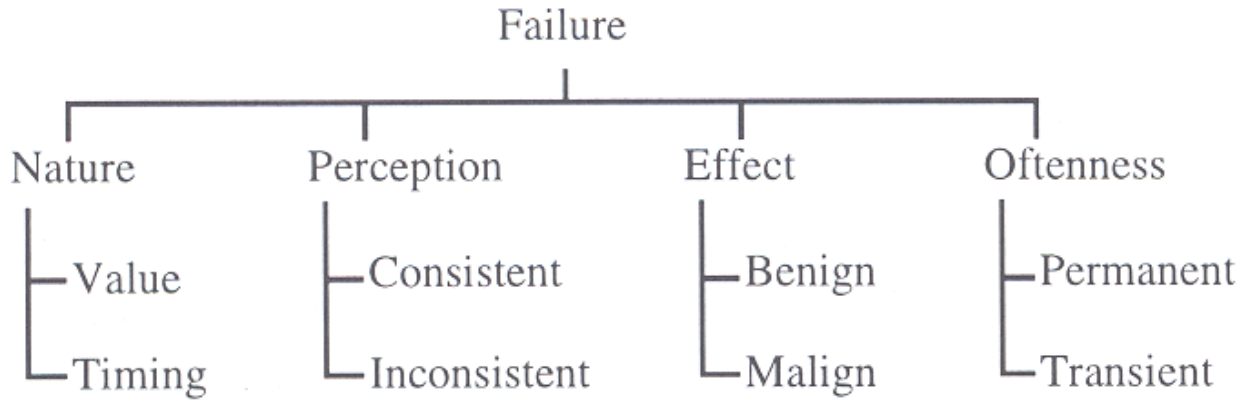
- Fault tolerance in safety-critical real-time systems
- Failures, errors, and faults
- Error detection from known correct behavior or comparison of two redundant channels
- Internal node failures and mapping into simple external failure mode
- Set of replica-determinate nodes to form a fault-tolerant unit
- Reintegration of nodes and selection of proper reintegration points
- h-state minimization to obtain valid reintegration points
- Design diversity for implementation of safety-critical systems

6.1 Failures, Errors, and Faults



Failures

A failure is an event that denotes a deviation between the actual service and the intended or specified service.



6.1 Failures, Errors, and Faults

Failure nature:

We distinguish between

- value failure (incorrect value is presented at the user interface)
- timing failure (value is presented outside the specified interval of real-time)

Failure perception:

In a system with more than one user, we can distinguish between

- consistent failure (all users see the same wrong result)
- inconsistent failure, a.k.a. two-faced failures, malicious failures, Byzantine failures (users see different false results)

Consistent failures are called fail-silent failures if they produce no result at all instead of delivering the correct service.

Consistent failures are called crash failures if the system stops operating after the first fail-silent failure.

A crash failure made known to the rest of the system is a fail-stop failure.

To tolerate k failures of a specific type, we need

- $k+1$ components if the failures are fail-silent
- $2k+1$ components if the failures are fail-consistent
- $3k+1$ components if the failures are malicious

Goal is to provide fail-silent behavior at the system level.

6.1 Failures, Errors, and Faults

Failure effect:

We distinguish between

- benign failures (failure cost is same order of magnitude as loss of normal utility of system, e.g. airplane is grounded)
- malign failures (failure cost is orders of magnitude higher than normal utility of the system, e.g. airplane crashes)

We call applications where malign failures can occur safety-critical applications.

Failure oftenness:

We distinguish between

- single failures (failure occurs only once; if system ceases operation thereafter until it is repaired this is called a permanent failure).
- transient failures (system continues to operate after failure)

If a transient failure occurs again and again, we call it an intermittent failure.

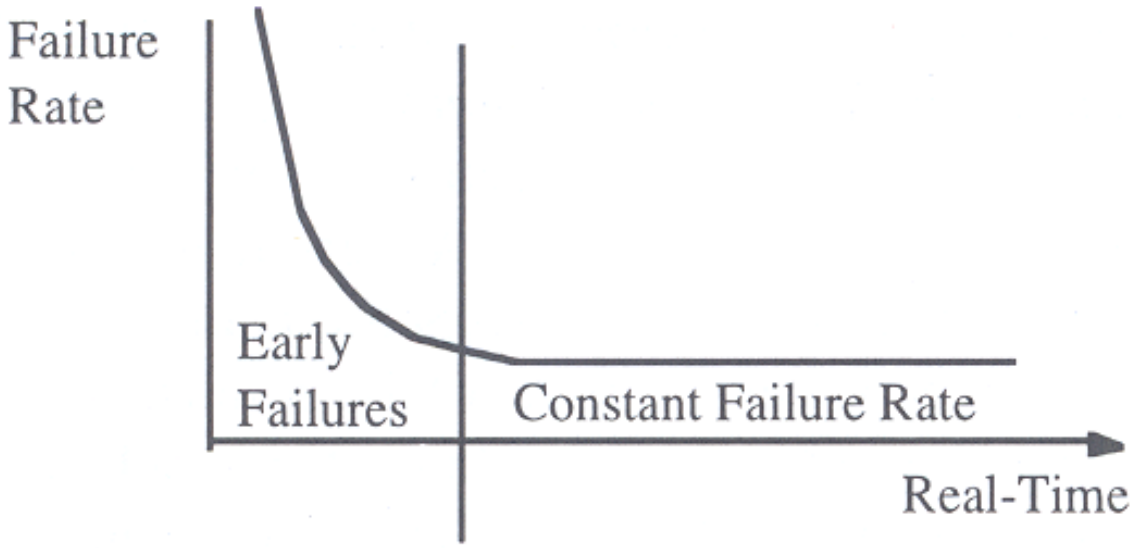
Permanent failures:

The failure rate of a typical VLSI device (permanent failures) changes over time as shown below. It stabilizes after some time between 10 and 100 FITS.

1 FIT means 1 failure per 10^9 hours, i.e. an MTTF of 115000 years.

Failure rate depends mostly on number of pins and packaging, and environmental conditions, and not so much on number of transistors.

6.1 Failures, Errors, and Faults

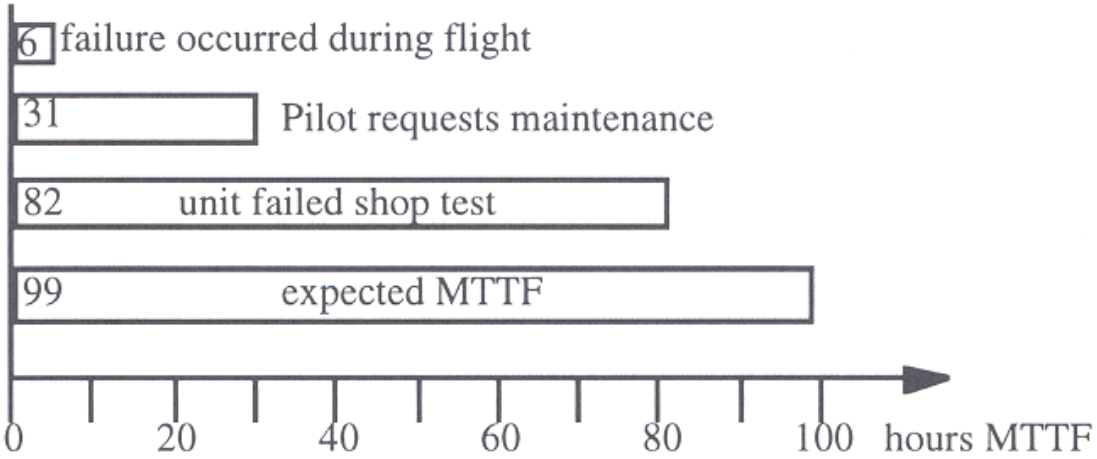


Cabling constitutes a significant source of failure in distributed systems. Even a high-quality connection is expected to fail with a failure rate of 0.1 – 1 FIT/wiring.

Transient failures:

At the chip level, the failure rate of transient failures is 10 to 100000 times higher than the permanent chip failure rate. Most common cause is EMI, disturbances in power supply, and high-energy particles.

6.1 Failures, Errors, and Faults



Example investigation on transient failures in the F16 fire-control radar, 150 aircrafts were observed over a period of 6 months. Less than 10% of the transient failures observed during operation (every 6 hour pilots noticed a failure) could be reproduced in the controlled environment of the repair shop.
This failure pattern is typical for modern distributed real-time systems.

Errors

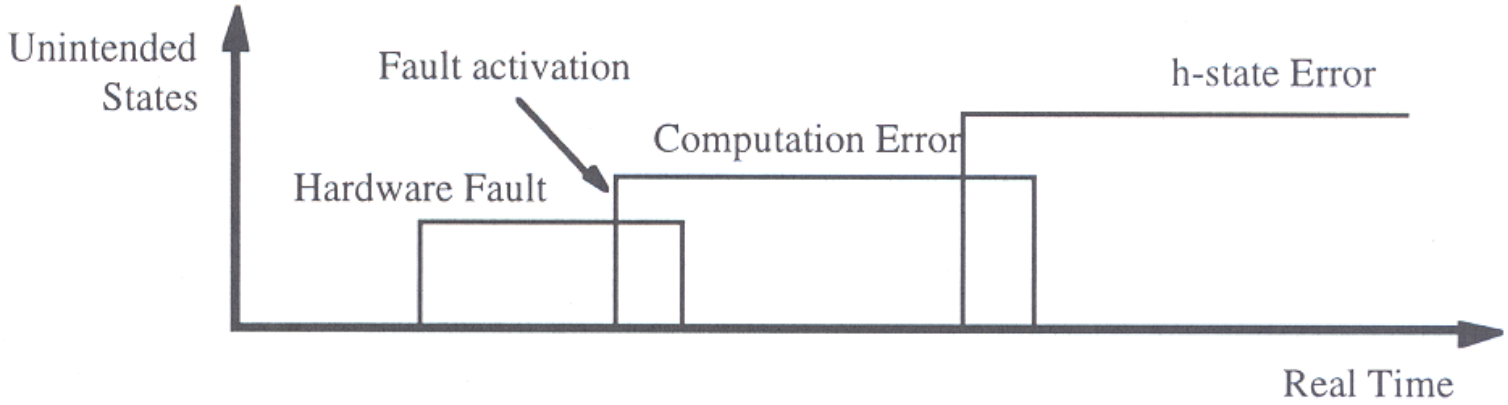
An error is an unintended state, e.g. a wrong data element in the memory. In a fault tolerant architecture, every error must be confined to a particular error containment region to avoid error propagation throughout the system. The boundaries of the error containment regions must be protected by error detection interfaces.

Transient errors:

If an error exists only for a short time interval, and disappears without explicit repair action, it is called a transient error. In many computer systems, transient errors form the predominant class of errors. Systems that periodically reach an empty h-state, i.e. where the internal data structures are periodically initialized, often exhibit this kind of behavior. Example: control loops.

Permanent errors:

If an error remains in the system until an explicit repair action is invoked to repair the state, we call it a permanent error. Permanent errors appear for example in systems where h-state data is stored in a database. If a database transaction is disturbed by a transient fault, a wrong value will be written to the database and remains there as a permanent error.

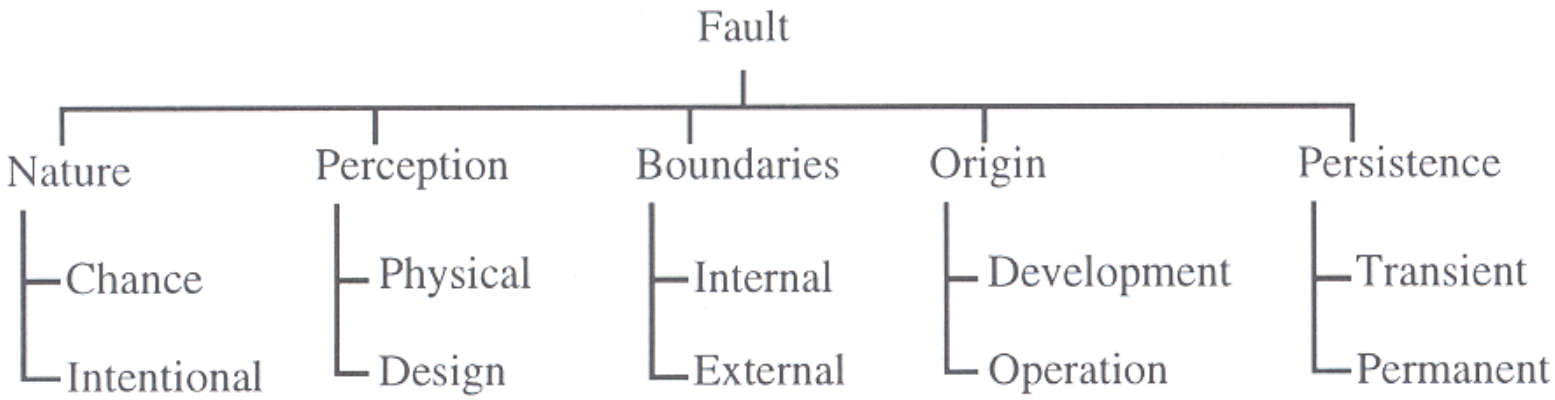


If database elements are used as inputs to future database transactions, database erosion can result, i.e. the number of errors in the database increases steadily.

Faults

Faults are the cause of an error.

6.1 Failures, Errors, and Faults



Fault nature:

A chance fault has its origin in a chance event, like the random break of a wire.
An intentional fault is the result of an intentional action, like the introduction of a computer virus into a system.

Fault perception:

Faults can be caused by physical phenomena, or by errors in the design. Faults caused by physical phenomena can be handled by specific techniques, such as active hardware redundancy. Faults caused by design errors are almost impossible to detect by testing, they can only be avoided by following stringent design methodologies, such as partitioning a complex system into autonomous subsystems interconnected by small, stable, and testable interfaces.

Fault boundaries:

Fault can be caused by a deficiency within the system, like a hardware error, or by some external disturbance, like a lightning stroke or spikes on the power line.

Fault origin:

The origin of a fault can lie in the development phase of a system, or it can be related to system operation, e.g. wrong input from an operator.

Fault persistence:

We distinguish between faults that occur only once and disappear by themselves, like a lightning stroke, and faults that remain in the system until they are removed by an explicit repair action.
In systems with a h-state, transient faults can cause permanent errors.

Systematic versus Application-Specific Fault Tolerance

The designer of a safety-critical system has two options to implement fault-tolerance:

- At the architecture level (systematic fault tolerance). The architecture must provide replica determinism and special or temporal replication of computations.
- At the application level (application-specific fault tolerance). Within the application code normal processing functions are intertwined with error-detection and fault-tolerance functions.

6.2 Error Detection

An error is a discrepancy between the intended correct state and the current state of a system. The knowledge about the intended correct state can arise from two sources:

- a priori knowledge about intended properties of states and behaviours of computation
- comparison of results from two redundant computation channels

In either case, error detection is based on redundancy.

Error Detection Based on A priori Knowledge

This technique requires restrictions on expected behaviour in either the temporal domain or the value domain, or both.

Syntactic knowledge about the code space: symbols are encoded with more bits than required for the regular code space, e.g. 128 symbols are encoded with 8 bits. This permits to a priori define what are valid code words, and a set of invalid ones.
One plus the maximum number of bit errors that can be detected in a codeword is called the Hamming distance of the code. Examples for error-detecting codes are parity bits and error-detecting codes in memory, CRC polynomials in data transmission, and check digits at the MMI.

Assertions and acceptance tests: application-specific knowledge about restricted ranges and temporal behaviour of the values of RT entities can be used to detect errors. For example, the crankshaft turn rate can only change with certain speed due to its mechanical inertia. Plausibility checks on the crankshaft turn rate can ensure that the values obtained are

plausible. Plausibility checks can be expressed in form of assertions within a program, or can be used to check for the plausibility of a result at the end of a program by applying an acceptance test. Assertions and acceptance tests are effective to detect errors in the value domain.

Activation patterns of computations: knowledge about the regularity in the activation pattern of a computation can be used to detect errors in the temporal domain (e.g. a computation is to be activated every (full) second). Jitter increases error detection latency.

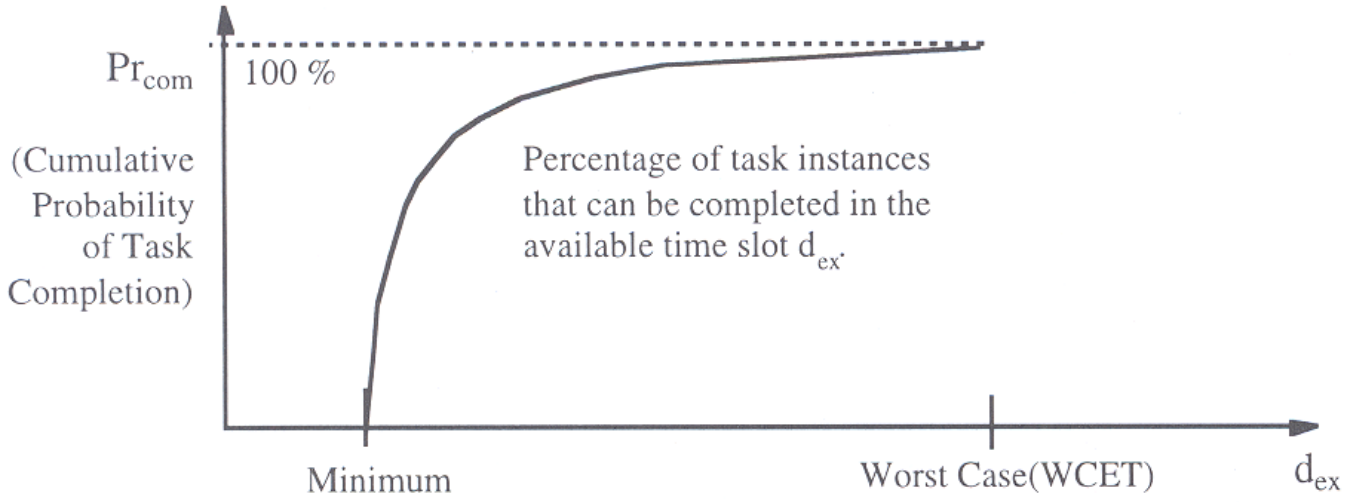
Worst-case execution time of tasks: in a real-time system, the worst-case execution time (WCET) of the hard real-time tasks must be known a priori for the calculation of the schedules. This knowledge can be used at run-time to detect errors in the temporal domain.

Error Detection Based on Redundant Computations

Type of redundancy	Implementation	Type of detected Errors
Time redundancy	Same software is executed on same hardware during two different time intervals	Errors caused by transient physical hardware faults with duration of less than one execution time slot
Hardware redundancy	Same software executes on two independent hardware channels	Errors caused by transient and permanent physical hardware faults
Diverse software on the same hardware	Different software versions are executed on same hardware during two different time intervals	Errors caused by independent software faults and by transient physical hardware faults with duration of less than one execution time slot
Diverse software on diverse hardware	Two different versions of software are executed on two independent hardware channels	Errors caused by independent software faults and transient and permanent physical hardware faults

Duplicate Execution of Tasks

Duplicate execution of application tasks at different times is an effective technique for the detection of transient hardware errors.



If execution time slot with length between WCET and 2xWCET, then the probability that a transient error is detected by double execution is given by

$$E_{double} \cdot \Pr_{double}(d_{ex})$$

where E_{double} is the error-detection coverage of duplicate executions for the detection of transient hardware faults.

6.3 A Node as a Unit of Failure

A node is considered a suitable unit of failure in a distributed real-time system. A node is a self-contained unit that provides a function across a small well-defined interface.

At architectural level, a node should display simple failure modes (e.g. fail-silent), node is operational or not. In this case fault-tolerance mechanism at architecture level must perform two major tasks:

1. Membership service: detect a node failure, and report this failure consistently to all operating nodes of the cluster within a short latency.
2. Redundancy management: mask the node failure by active redundancy, and to reintegrate repaired nodes into the cluster as soon as they become available again.

Minimum Service Level of a Node

Large systems have more than the two operational states fully operational and non-operational. Example: MMI of an industrial plant control with two displays. With one display working, the system delivers the minimal level of service. Every service level below this is considered a failure.

As long as the node is delivering the minimal level of service, it is considered operational. The membership service classifies nodes according to this binary scheme: operational or non-operational.

6.3 A Node as a Unit of Failure

Error Detection within a Node

A node must detect all internal failures (value and timing) within a short latency, and must map these failures to a single external failure mode, a fail-silent node failure.

A particular severe failure is when a node sends messages at the wrong moment or even monopolizes the communication channel (babbling idiot timing failure).
Error detection in the temporal domain at the external node interface (below the CNI) can only be performed if a priori knowledge is available about when a node is allowed to send a message.

In a TT system, this information is static and stored in the communication system. In an ET architecture, messages are send dynamically, and it is thus much more difficult to advise an error detection scheme for the detection of timing failures.

Execption Handling

After an exception has been raised, control is transferred to an exception handler. After the exception handler has been executed, operation resumes from the point of exception, or the task is terminated.

In real-time systems, exception handling can be problematic. The WCET of a task is extended by the WCET of all exception handlers that can possibly be activated during the execution of that task.

If the exception handler inside a node repairs the damage within the given time constraints, the fault is corrected; otherwise the node fails as a unit.

6.4 Fault Tolerant Units

A fault-tolerant unit (FTU) is supposed to mask the failures of a node.

If a node implements the fail-silent abstraction, then duplication of nodes is sufficient to tolerate a single node failure.

If a node does not implement the fail-silent abstraction, but can exhibit value errors at the CNI, triple modular redundancy (TMR) must be implemented.

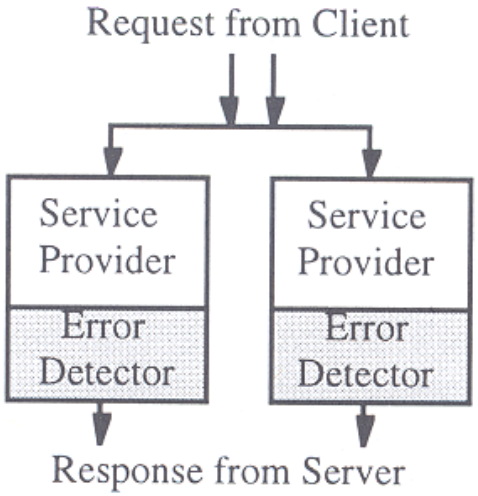
If no assumptions can be made about the failure behaviour of a node, i.e., a node can exhibit Byzantine failures, four nodes are required to form a fault-tolerant unit.

Fail-Silent Nodes

A fail-silent node either produces correct results or none at all. In a TT-architecture, an FTU that consists of two fail-silent nodes produces either zero, one, or two correct result messages.

If it produces no message, the FTU has failed. If it produces one or two messages, the system is operational. The receiver must discard redundant result messages (easy if the messages are idempotent).

6.4 Fault Tolerant Units



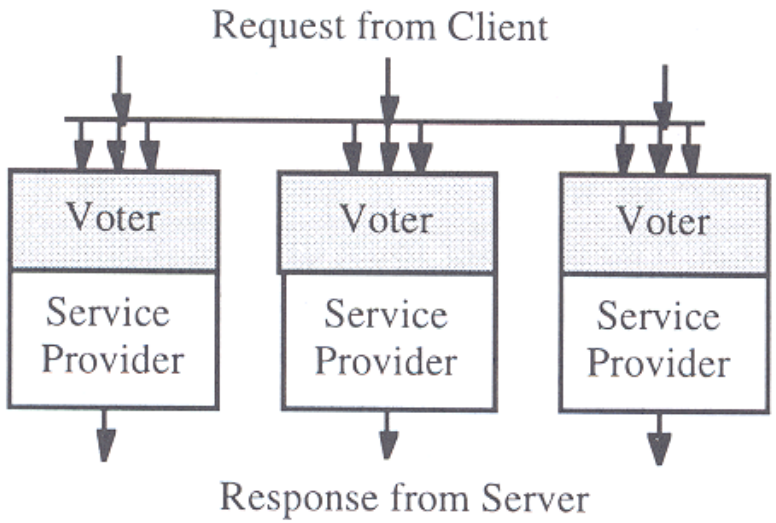
In a bus based system, an FTU can comprise a shadow node in addition to the two active nodes. A shadow node acts as a warm standby: it reads all messages from the bus, and is fully synchronized with the active nodes, but does not produce any output. This setup has the following advantages:

1. Whenever an active node fails, the redundancy within the FTU is re-established within a short time interval.
2. During normal operation the shadow node does not consume any bandwidth of the communication system
3. During repair of the failed node, the redundancy within the FTU is maintained.

6.4 Fault Tolerant Units

Triple Modular Redundancy

If a node can exhibit value failures at the CNI, a fault-tolerant unit must consist of three nodes and a voter. The voter detects and masks errors in one step by comparing the three independently computed results, and then selecting the result that has been computed by the majority.



There are two different kind of voting strategies:

- Exact voting: bit-by-bit comparison of the data fields for the result messages. If two out of three result messages have exact same bit pattern, one of these is selected as output.
- Inexact voting: two messages are assumed to contain same result if the results are within some application-specific interval. Must be used if replica determinism cannot be guaranteed. In practice this approach is difficult to implement.

6.4 Fault Tolerant Units

Byzantine Resilient Fault-Tolerant Unit

Four nodes are required to form a fault-tolerant unit if no assumptions can be made about the failure mode of a node.
These four nodes must execute a Byzantine-resilient agreement protocol to agree on a malicious failure of a node. The following requirements pertain to this protocol to tolerate the Byzantine failures of k nodes:

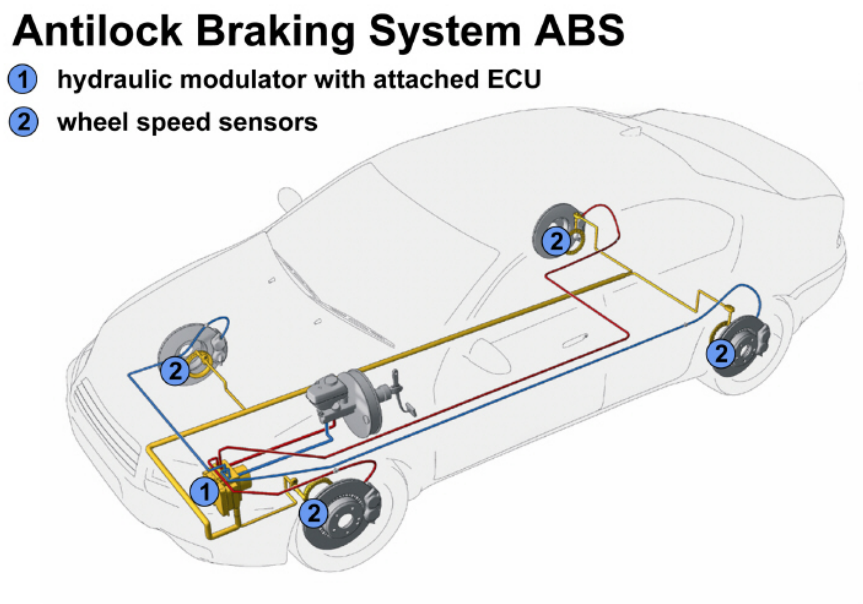
1. An FTU must consist of at least 3k+1 nodes.
2. Each node must be connected to all other nodes of the FTU by k+1 disjoint communication paths.
3. To detect the malicious nodes, k+1 rounds of communication must be executed among the nodes. A round of communication requires every node to send a message to all other nodes.
4. The nodes must be synchronized to within a known precision.

The Membership Service

The failure of an FTU must be reported in a consistent manner to all operating FTUs with a low latency by the membership service.
A point in real-time when the membership of a node can be established, is called a membership point of the node.
A small temporal delay between the membership point of a node and the instant when all other nodes are informed about the membership is critical for the correct operation of many safety-relevant applications.

Example: intelligent antiblock system (ABS), with a node placed at each wheel.

6.4 Fault Tolerant Units



In case a wheel computer fails, the hydraulic brake-force actuator goes into a safe state, e.g. the wheel is free running. If the other nodes learn about the failure within a short period of time, e.g. a single control loop cycle of 5 ms, they can increase their brake force to compensate for the lost brake.
If the loss of a node is not recognized within a short interval, the total break force suddenly changes and the car can get out of control.

6.4 Fault Tolerant Units

ET architecture: In an ET architecture, messages are only sent when a significant event happens at a node. Silence of a node thus either means no significant event has happened, or a fail-silent failure has occurred. An additional time-triggered service, e.g., a periodic watchdog service, must be implemented in an ET architecture to solve the membership problem.

TT architecture: In a TT architecture the periodic message send times are the membership points of the sender. From the arrival of an expected message it is possible to conclude that the sender node is alive. Since message send times are known a priori, it possible to derive an a priori bound for the temporal accuracy of the membership service.

6.5 Reintegration of a Repaired Node

After the occurrence of a node failure, a self test is performed. If self-test is successful, a transient fault can be assumed and the reintegration of the node can start immediately.

If the self-test shows a permanent error, or if a transient failure occurs repeatedly within a specified time interval, a permanent hardware fault must be assumed. In this case, the node needs to be replaced.

In a fault-tolerant system it must be possible to replace a node while the system is under power.

Finding a Reintegration Point

Key issue is to find a future point in time when h-state of node is synchronous with the node environment. This may not always be possible.

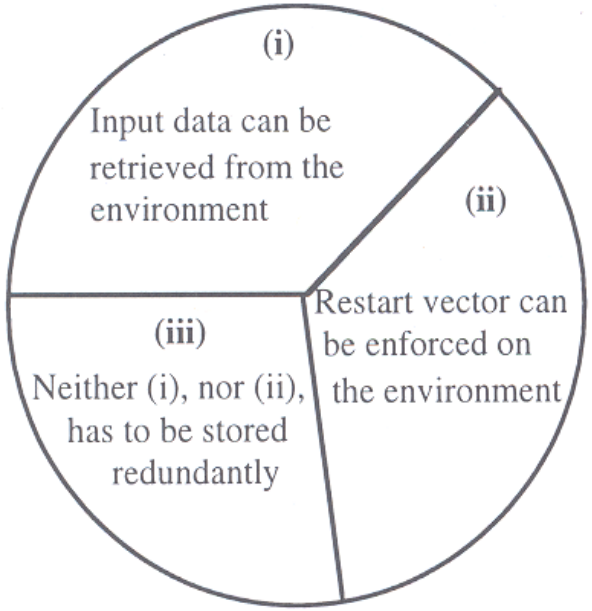
An ideal reintegration point is when a component is in ground state, since in this state the h-state is minimized (ground state: no task is active, all communication channels are flushed).

If in the ground state the h-state is empty, reintegration of a repaired node is trivial.

Minimizing the H-State

The points in time when a repaired node can be reintegrated need to be planned during system design.
Investigation of all system data structures is required. Good programming practice: output h-state of a task in a special output message, and re-read the h-state of the task when the task is re-activated.

6.5 Reintegration of a Repaired Node



Suggestion: divide h-state into three parts:

1. input data from the environment. Example: scan all sensors to synchronize the node with the external world
2. output data that are in control of the computer, and can be enforced on the environment. Example traffic control system: set all lights to yellow, then to red.
3. Any other h-state data. This may have to be retrieved from an operator, or from a node that has stored this information redundantly.

In a system with replicated nodes in an FTU, the h-state must be communicated from one node of the FTU to the other nodes of the FTU, it cannot be retrieved from the environment.

Node Restart

6.5 Reintegration of a Repaired Node

Possible procedure

1. Power up
2. self test
3. verify correctness of i-state by checking signatures
4. If i-state is erroneous, reload from stable storage
5. Scan all instruments
6. Wait for a cluster cycle to acquire all available current information about environment
7. Analyze information, decide on mode of controlled object
8. Retrieve class (iii)

6.6 Design Diversity

Observations indicate that many computer system failures result from design errors in the software, and not physical faults of the hardware. Using FTU does not help, since they will probably exhibit common-mode failures.

Main cause for software design errors is unmanaged complexity of a design.

There are three major strategies to attack the problem of unreliable software:

Use clean concep-tional structure	Use formal methods for rigorous form specifications	Employ design di-versity

These approaches complement each other and should be employed in ultra-high dependable safety-critical real-time systems.

Diverse Software Versions

6.6 Design Diversity

Design diversity is based on the assumption that different programmers using different programming languages and different development tools, don’t make the same programming errors.

However, what about specification flaws? Thus, the different software would have to be based on different specifications. Example: VOTRICS train signalling system by Alcatel.

- An industrial example of applying design diversity in a safety-critical RT environment
- Objective of train signalling system:
  - Collect data about the state of the tracks in a train station—current position and movements of trains, position of points
  - Set signals and shift points such that trains can move safely through the station according to a given time table

VOTRICS is partitioned into two independent subsystems

- First system:
  - Accepts commands from operators
  - Collects data from tracks
  - Calculates intended positions of signals and points
  - Uses a standard programming paradigm
  - Uses a Triple-Modular Redundancy (TMR) architecture to tolerate single HW fault

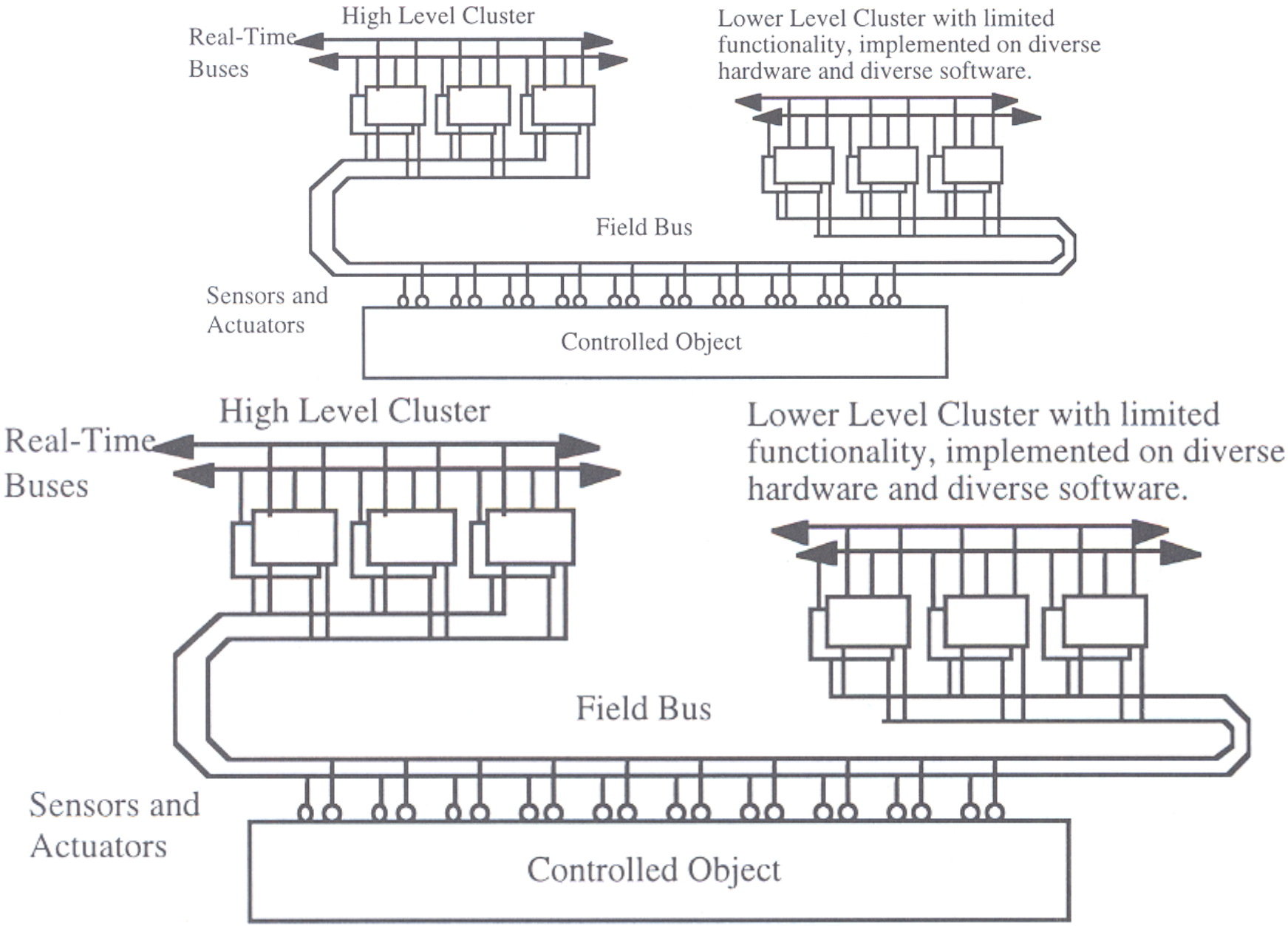
- The second system, the “safety bag”:

- Monitors safety of the state of the station
- Has access to RT data base and intended outputs of 1st system
- Dynamically evaluates safety predicates derived from the “rule book” of the railway authority
- Based on expert-system technology
- Also implemented on TMR HW architecture

Multilevel System

Higher level system provides full functionality, with high-error detection coverage. In case of failure of the high-level computer system, the lower-level system takes over, with reduced functionality. The lower-level system must still guarantee safety.

6.6 Design Diversity



Exercise

Tolerating Arbitrary Faults

A number of generals, facing an enemy army, must decide whether to attack or retreat. Their armies can defeat the enemy if they all attack. If at least one army does not attack, they will be defeated.

Most of the generals are loyal to each other, but some are traitors. The problem to solve is to reach a single binary agreement (attack/retreat), despite the effort of the traitors which try to prevent such an agreement.

Communication takes place in synchronous rounds: each general sends a message to all other generals. The traitors may omit some or all messages and may send conflicting messages to different generals. Each general decides based on simple majority.

- Scenario 1: Two loyal generals, one traitor
- Scenario 2: Three loyal generals, one traitor