IV.2 Fault Tolerance

- To ensure safety, the system's design must "deal" with all anticipated faults
- One strategy to do this are ***execution-time techniques*** that cope with the effects of faults and reduce its effects to an acceptable level!

Fault tolerance (FT): Providing a service that is consistent with its specification in spite of faults.

Limitations of Fault Tolerance

Can work only for anticipated faults:

- You can't tolerate what you don't expect
- But if we expected it, we would avoid or eliminate the fault!

⇨ employ only for faults you cannot avoid/eliminate

In general:

- We can itemize the classes of faults that can occur
- If the fault occurs (the error is detected) we can react on this

Four Phases of Fault Tolerance

(1) Error detection:
- You must know there is a problem in order to deal with it

(2) Damage assessment:
- You must know or at least estimate the damage so as to know how bad the situation is

(3) State restoration:
- A consistent state is needed to continue

(4) Continued service:
- Do something useful with what is left

(1) Error Detection Techniques

- Functionality checking
  Only hardware: e.g., memory checks via checksums
- Consistency Checking
  e.g., range checks
- Signal Comparison
  - Checking pairs
- Information redundancy
  - Parity checking, checksums, …
- Instruction monitoring
  - CPU reavtion when an invalid instruction code is detected
- Loopback testing
  Feedback output to compare it with source
- Watchdog timers
  reset CPU when a timer is not incremented
- Bus monitoring
  check address ranges on the bus
- Power supply monitoring
  - Power supply monitor initiates emergency action before voltage reaches dangerous level
  - Uninterruptible power source when no disruption can be permitted

(2) Damage Assessment

- How much damage to the system occurs when a component fails? It might be a lot.....

Example: an Ada exception (Ariane 5 accident)

- **Failure semantics** = defines which divergent behaviour is possible if faults are present
- Components that are expected to fail must have **predefined** failure semantics
- **Hazard analysis** reveals which hazardous behaviour of the system might result
- If critical hazards can result, the design has to take care to **exclude** the involved chain of events

(3) State Restoration / Error Recovery

- There are two possibilities to transform a currently erroneous system state into an error-free system state:

Backward recovery:
- system state is reset to a previously store error-free system state
- Re-execution of failed processing sequence

e.g., **database systems** (predict valid system states is not possible)

Forward recovery:
- system state is set to a new error-free system state

typical for **real-time systems** with periodic processing patterns
(it *is* possible to predict valid system states)

(4) Continued Service

- Some kind of **redundancy** is required to tolerate faults, because whether or not an error actually leads to a failure depends on the following facts:
  - the system **composition** and the existence of **redundancy** (intentional or unintentional redundancy)
  - the system **activity** after the introduction of an error (the error may get overwritten)
  - the definition of the correct operation (which implicitly defines what is a failure or not)

IV.2.1 Fault-Tolerance and Redundancy

Redundancy can occur in 3 different domains.

(1) Domain of information:

redundant information e.g. error correcting codes, robust data structures

(2) Domain of space:

replication of components, e.g. 2 CPU's, UPS (uninterruptible power supply)

(3) Domain of time:

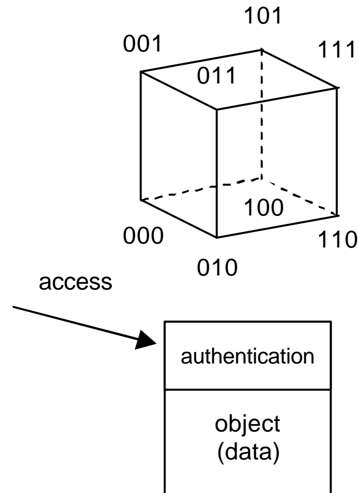
replication of computations, e.g. calculate results by same (or different) algorithm a second time, sending messages more than once

FT in the Domain of Information

error correcting codes:
- for all error correcting codes (ECC)
- $(2t + p + 1) = d$
- d .. Hamming distance of code
- t .. number of single bit errors to be tolerated
- p .. number of additional detected errors

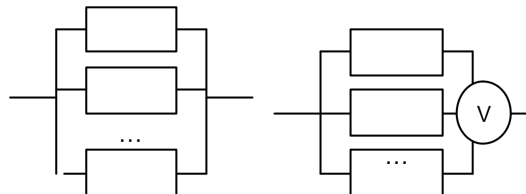
robust data structures:
- store the number of elements
- redundant pointers
  (e.g. double linked chains with status)
- status or type information
  (e.g. authenticated objects)
- checksum or CRC

application specific knowledge

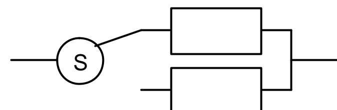


FT in the Domain of Space

Active redundancy
- parallel fail-silent components
- voting, triple modular redundancy (TMR)



passive or standby redundancy
- **hot standby:** standby component is operating in background
- **cold standby:** standby components starts only when required



FT in the Domain of Time

allows tolerance of temporary faults

multiple calculation:

- a function is calculated *n* times with the same inputs
- the result is checked by an acceptance test
- or the multiple results are voted

sending messages multiple times:

- message transmission is repeated *n* times
- retransmission only in case of failures (positive acknowledge retransmit PAR)
- retransmission always *n* times (reduces temporal uncertainty for real-time systems)

Redundancy and Diversity

- Redundancy with identical components protects against random hardware component failures, but not systematic ones (common mode failures)

⇨ **diversity** is also required

- Hardware diversity: micro-controller and hard wired or programmable logic controller (PLC)
- Software diversity: Common mode failures can always result form the specification

IV.2.2 Techniques for FT

There are two fundamental approaches to fault-tolerance:

- **Systematic fault-tolerance**
  - replication of components
  - divergence of components is used for fault-detection
  - redundant components are used for continued service
- **Application-specific fault-tolerance**
  - reasonableness checks for fault detection (based on model of real world)
  - state estimations for continued service

Application-specific Fault-Tolerance (1/2)

- the computer system interacts with some physical process, the behaviour of the process is constrained by the law of physics
- these laws are implemented by the computer system to check its state for reasonableness
- for example:
  - the acceleration/deceleration rate of an engine is constrained by the mass and the momentum that affects the axle
  - signal range checks for analogue input signals
- reasonableness checks are based on application knowledge
- fail-stop behaviour can be implemented based on reasonableness checks

Application-specific Fault-Tolerance (2/2)

- the laws of physics constraining the process can be used to perform state estimations in case some component has failed
- for example:
  - if the engine temperature sensor fails a simple state estimation could assume a default value
  - a better state estimation can be based on the ambient temperature of the engine, engine load and thermostatical behavior of the engine
  - the speed of a vehicle can be estimated if the engine speed and the transmission ratio is known
- state estimations are based on application knowledge
- fail-operational behaviour can be implemented based on reasonableness checks and state estimations

Systematic Fault-Tolerance (1/2)

- does not use application knowledge, makes no assumptions on the physical process or controlled object
- uses **replicated components** instead
- if among a set of replicated components, some — but not all — fail then there will be divergence among replicas
- information on divergence is used for fault detection
- replicas are therefore required to deliver corresponding results in the absence of faults
- The problem of **replica determinism**:

  due to the limited accuracy of any sensor that maps continuous quantities onto computer representable discrete numbers it is impossible to avoid non-deterministic behaviour

Systematic Fault-Tolerance (2/2)

- systematic fault-tolerance requires agreement protocols due to replica non-determinism
- the agreement protocol has to guarantee that correct replicas return corresponding results (the problem of replica determinism is discussed later)
- fail-stop behaviour can be implemented by using the information of divergent results
- fail-operational behaviour can be implemented by using redundant components

Comparison of Techniques (1/3)

Systematic fault-tolerance	**Application-specific fault-tolerance**
replication of components	no replication necessary
divergence among replicas in case of Faults	—
no reasonableness checks necessary	reasonableness checks for fault detection
requires replica determinism	—
no application knowledge necessary	depends on application knowledge
exact distinction between correct and faulty behaviour	fault detection is limited by a *grey zone*

Comparison of Techniques (2/3)

Systematic fault-tolerance	Application-specific fault-tolerance
no state estimations necessary	state estimations for continued service
independence of application areas	missing or insufficient reasonableness checks for some application areas
service quality is independent of whether replicated components are faulty or not	quality of state estimations is lower than quality delivered during normal operation
correct system function depends on the number of correct replicas and their failure semantics	correct system function depends on the severity of faults and on the capability of reasonableness checks and state estimations
only backward recovery	forward and backward recovery

Comparison of Techniques (3/3)

Systematic fault-tolerance	Application-specific fault-tolerance
additional costs for replicated components (if no system inherent replication is available)	no additional costs for replicated components
no increase in application complexity	considerable increase in application complexity
considerable increase of system level complexity	no increase of system level complexity
separation of fault-tolerance and application functionality	application and fault-tolerance are closely intertwined
fault-tolerance can be handled transparently to the application	fault-tolerance is not handled transparently to the application

Systematic and Application-specific FT

- under practical conditions there will be a compromise between systematic and application-specific fault-tolerance
- usually cost, safety and reliability are the determining factors to choose a proper compromise
- **software complexity** plays an important role:
  - for complex systems software is almost unmanageable without adding fault-tolerance (fault containment regions and software robustness)
  - therefore systematic fault-tolerance should be applied in favor of application-specific fault-tolerance to reduce the software complexity
  - systematic fault-tolerance allows to test and to validate the mechanisms independently of the application software (divide and conquer)

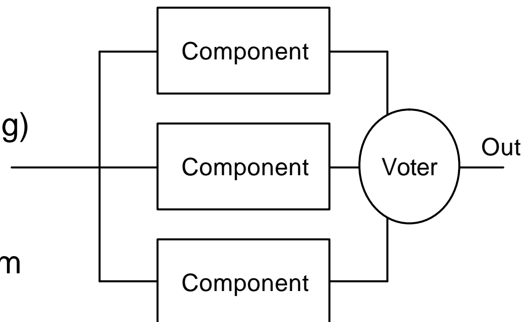
IV.2.3 Hardware Fault Tolerance

- **Static redundancy**
  - Fault masking to prevent error propagation
- **Dynamic redundancy**
  - Detection of faults plus actions to nullify them
- **Hybrid redundancy**
  - Fault masking to prevent error propagation
  - Detection of faults and reconfiguration to remove faulty units from the system

Triple Modular Redundancy (TMR)

Redundancy

- Domain of space
- Static
- Signal comparison (voting)



Advantages:

- Protection against random component failures

Disadvantages:

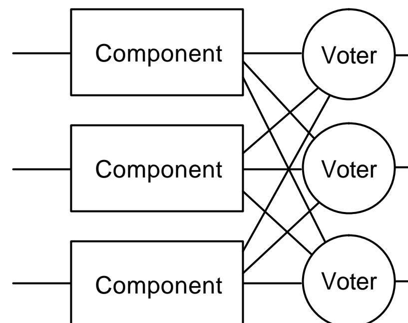
- Voter a single point of failure
- High redundancy costs

University of Paderborn
Software Engineering Group

Triplicated Voting

Redundancy

- Domain of space
- Static
- Signal comparison (voting)



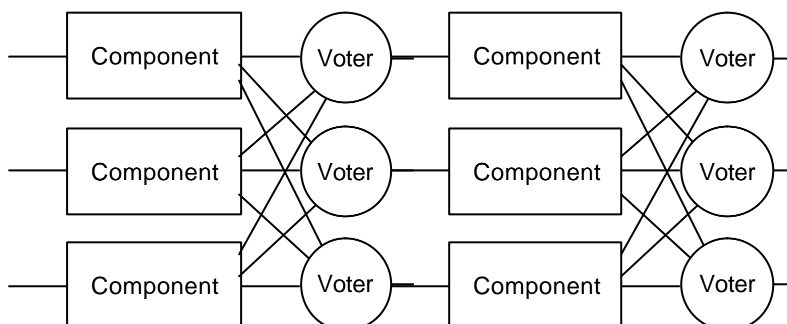
Advantages:

- Protection against random component and voter failures

Disadvantages:

- Even higher redundancy costs

Multistage TMR Arrangement



- Even more expensive, but also do not mask two failures occurring in two components of one stage

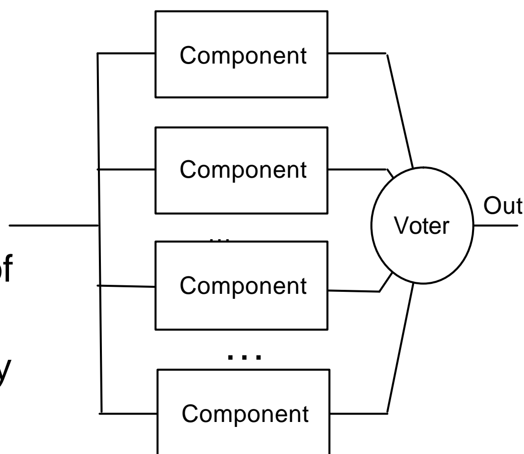
N-Modular Redundancy

Advantages:

- Protection against (N-1)/2 random component failures

Disadvantages:

- Voter a single point of failure
- Very high redundancy costs



Cold Standby Spares

Redundancy

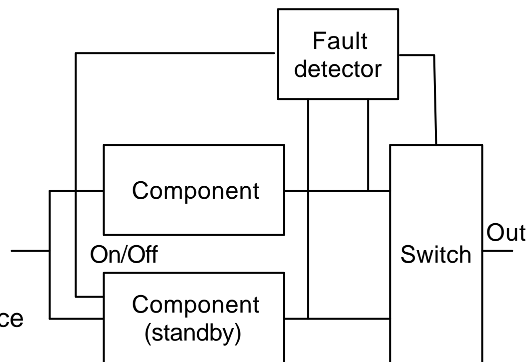
- Passive/standby and dynamic
- cold standby

Advantages:

- Lower costs than redundancy (component twice + fault detector + switch)

Disadvantages:

- No fault masking
- Reconfiguration may cause a momentary disruption of service while standby unit is activated



Hot Standby Spares

Redundancy

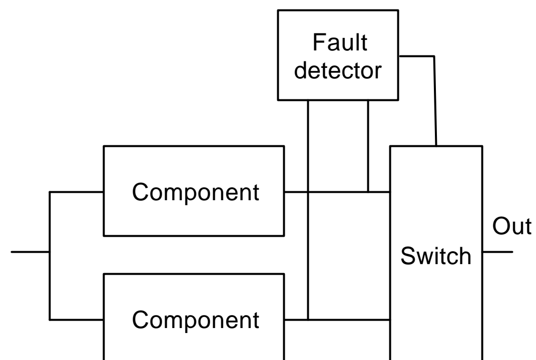
- Passive/standby and dynamic
- hot standby

Advantages:

- Lower costs than redundancy (component twice + fault detector + switch)

Disadvantages:

- No fault masking
- Increases power consumption
- Standby unit has the same stress as the active unit

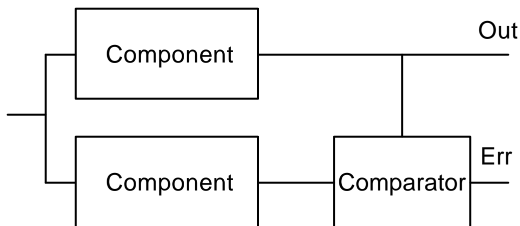


Self-Checking Pair

Redundancy
- Passive/standby and dynamic
- hot standby

Advantages:
- Lower costs than redundancy (component twice + Comparator)

Disadvantages:
- No fault masking
- Increases power consumption
- Standby unit has the same stress as the active unit
- Comparator single point of failure

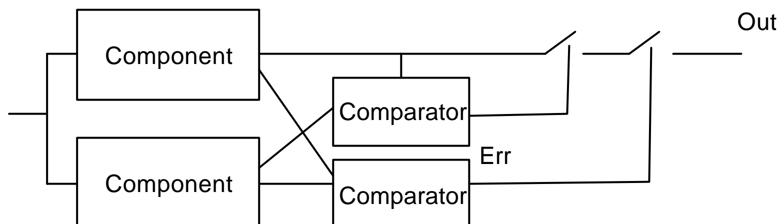


Self-Checking Pair (Fail-Silent)

Redundancy
- Passive/standby and dynamic
- hot standby

Advantages:
- Lower costs than redundancy (component + Comparator twice)

Disadvantages:
- No fault masking
- Increases power consumption
- Standby unit has the same stress as the active unit
- No output must result in safe state



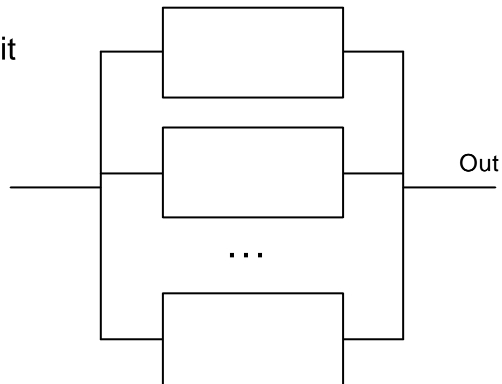
Parallel Fail-Silent Components

Fail-Silent Component

- Internal error detection unit prevents faulty result form occurring on the output

passive or standby redundancy

- **hot standby:** standby component is operating in background
- **cold standby:** standby components starts only when required



N-Modular Redundancy with Spares

Redundancy

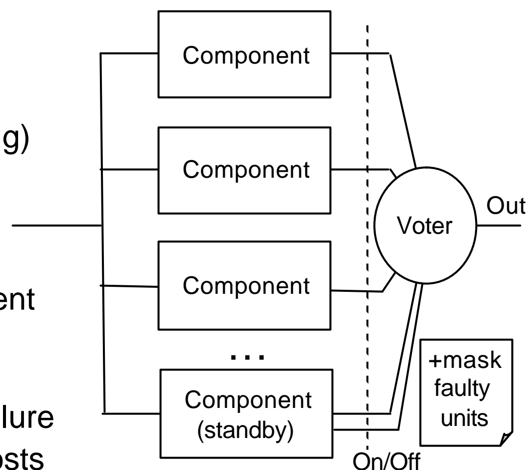
- Domain of space
- Active and hybrid
- Signal comparison (voting)

Advantages:

- Protection against (N-1)/2 random component failures

Disadvantages:

- Voter a single point of failure
- Very high redundancy costs



IV.2.4 Software Fault Tolerance

Two meanings:

- "Tolerance of software faults"
  - Can be addresses by the techniques for hardware fault tolerance using software diversity
- **"Tolerance of faults by the use of software"**
  - Includes first case plus effects of the underlying hardware

Achieve diversity:

- Usually the same requirements (weakness)
- different programmers, contractors?

Hardware vs. Software

- hardware components are more reliable compared to software components
- very mature technology for hardware process validation
- but: "build it in hardware instead" is no solution at all since the problem of design dependability arises because of the system inherent complexity:
  - very complex systems are realized in software because of their complexity
  - software is often used to implement radically new systems
- higher flexibility of software is often exploited by very short modification cycles

- Outages in % by fatal faults for the **Tandem** system illustrates the shift from hardware to software (cf. [Gray1985]):

	1985	1987	1989
Software	34%	39%	62%
Hardware	29%	22%	7%
Maintenance	19%	13%	5%
Operation	9%	12%	15%

Exception Handling (1/3)

- to detect erroneous states of software modules the exception mechanism can be used (software and hardware mechanisms for detection of exceptional states)
- a procedure (method) has to satisfy a pre condition before delivering its intended service which has to satisfy post conditions afterwards
- the state domain for a procedure can be subdivided:
  - Anticipated exceptional domain
  - Unanticipated exceptional domain
  - Standard domain

Exception Handling (2/3)

- an exception mechanism is a set of language constructs which allows to express how the standard continuation of module is replaced when an exception is raised
- exception handlers allow the designer to specify recovery actions (forward or backward recovery)

Use run-time system to handle faults:

- raise an exception when an erroneous state is detected
- pass control to appropriate handler
- could be on another processor
- Propagate to outmost scope then fail

Example:

- Ada...

Exception Handling (3/3)

Advantages:

- no voting required
- fault detection distributed in the code (easier)

Disadvantages:

- fault detection distributed in the code (structure?)
- correct run-time handling of exceptions required

Remark:

- complex control structures (difficult verification)

Recovery Blocks (1/4)

- a method to apply diverse designs to provide design fault-tolerance based on an acceptance test—which detects erroneous states—different modules are tried until an acceptable state is reached
- Examples for Acceptance tests:
  - Checks for run-time errors
  - Checks for reasonability
  - Excessive execution time
  - Mathematical errors
- acceptance tests are application-specific, they have only limited error detection coverage

Recovery Blocks (2/4)

Program scheme:

```
primary module
acceptance test
secondary module
acceptance test
```

Problem:

- execution of a module might corrupt system state

⇨ Recovery Point:

- backward error recovery!
- use entire system state is inefficient

Idea:

- add recovery point before primary module execution

Program scheme:

```
Establish recovery point
primary module
acceptance test
alternative module 1
acceptance test
alternative module 2
acceptance test
...
```

Recovery Blocks (3/4)

Advantages:

- no voting required
- can also handle (transient) hardware faults
- can be used to implement graceful degradation, when different modules provide different levels of service

Disadvantages:

- additional acceptance test required
- delay for backward recovery in real-time systems
- It is difficult to development acceptance tests
- the quality of acceptance test is often questionable

Recovery Blocks (4/4)

Remarks:

- mixture of systematic and application-specific fault-tolerance:
  - systematic method to apply n diverse modules by rollback recovery
  - acceptance test is application specific
- recovery blocks can be nested such that a module itself is a recovery block
- can also be supported with the exception mechanism (e.g. standard exception handler for unidentified exceptions can be used)
- modeling of recovery block with primary and one alternative is equivalent to passive redundant system (acceptance test ? switch)

Distributed Recovery Blocks

- for uniform treatment of software and hardware failures
- the primary module is executed on the primary processor, the alternate is executed on a backup processor
- both processors use duplicated acceptance test
- if the primary module fails, a message is sent to the backup and the backup then forwards its results
- combination of software and hardware diversity

N-Version Programming (1/4)

- n non-identical replicated software modules are applied and instead of an acceptance test a voter takes a m out of n or majority decision
- driver program to invoke different modules (different processes for module execution), wait for results and voting require more resources than recovery blocks but less temporal uncertainty (response time of slowest module)

Redundancy

- Domain of space or time
- Signal comparison (voting)

Example:

- Primary flight control of the Airbus A330/A340

N-Version Programming (2/4)

Advantages:

- For N=2 like self-checking pair (repeating the execution helps for transient faults; diagnosis to determine faults routine)
- Protection against (N-1)/2 faulty program versions

Disadvantages:

- High implementation costs (>N due to the voting)
- performance costs of n executions and voting
- Common mode faults are not excluded

N-Version Programming (3/4)

Problems of replica non-determinism:

- the real-world abstraction limitation is no problem (all modules get exactly the same inputs from driver program)
- consistent comparison problem: diverse implementations, different compilers, differences in floating point arithmetic, multiple correct solutions (n roots of nth order equation), …

- What can be done?
  - there is no systematic solution for the consistent comparison problem
  - either very detailed specification with many agreement points (limits diversity)
  - or approximate voting to consider non-determinism (application-specific)

N-Version Programming (4/4)

- n-version programming is approach to systematic fault-tolerance:
  - there is no application specific acceptance test necessary
  - exact voting on every bit is systematic
- modeling of n-version programming is equivalent to active redundant systems with voting

Remark:

- costs make N>2 very uncommon
- Only highly safety-critical systems

N self-checking programming

- n versions are executed in parallel (similar to N-version programming)
- each module is self-checking, an acceptance test is used (similar to recovery blocks)
- mixture of application specific and systematic fault-tolerance
- requires no backward recovery and no voting

Deadline Mechanism

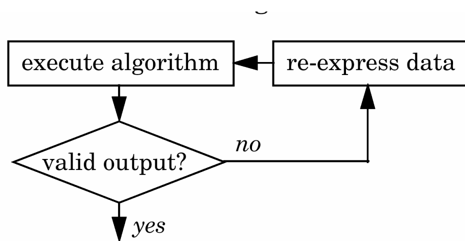
- based on recovery blocks, but deadline instead of acceptance test
- used to avoid timing failures in real-time systems

```
service name
  within response-period
  by primary_module
else
  by alternate_module
```

- it is assumed that an upper execution bound for the alternate is known
- for the primary it is assumed that the execution is timely in most cases
- if the primary does not finish within the slack time (response-period – execution bound for alternate) then the primary is aborted and the alternate is used

Data Diversity

- it is assumed that software fails on some “special” inputs
- if the inputs are changed slightly then the same software may work correctly



- data re-expression is necessary to generate different but logically equivalent data sets (application specific)
  - for real variables the value may be changed slightly
  - coordinate transformation to new origin
- cheaper alternative to diverse software

Independence Assumption

- empirical studies have shown that diverse designed software does not fail independently (co-dependent failures)
  - 27 program versions have been written by two universities
  - failure probabilities for 1 10-6 test cases with 351 2-version systems and 2925 3-version systems were calculated
- for the average 3-version system the failure probability improved by a factor 19, compared to the average single version
- if the **independence assumption** would hold, the failure probability should have decreased by at least three orders of magnitude

Problems Due to Co-Dependence

- software fault-tolerance is based on the **inpependence assumption** that predicts that diverse designed models fail independently
  - different programmer teams
  - different programming language and tools, …

But …

- only modest increase for very high effort
- development costs are main costs for software
- replica non-determinism or application-specific methodology
- Increasing costs and time for handling problems for multiple version systems (project management, configuration control, versioning, modifications and updates)

Forms of Redundancy

Control redundancy includes:

- exception handling
- recovery blocks
- n-version programming
- n self-checking programming
- deadline mechanism
- data diversity

Data redundancy uses extra data

- to check the validity of results
- Error correcting/detecting codes
- Checksum agreements etc.

Summary

- N-version programming similar to N-modular Redundancy
- Recovery blocks similar to dynamic redundancy
- Duplicated identical hardware modules provide fault tolerance for some form of hardware faults whereas duplication of identical software has little benefit (only transient faults)
- Software redundancy required diversity (due to the high costs usually preserved for highly critical applications)

IV.2.5. Replication

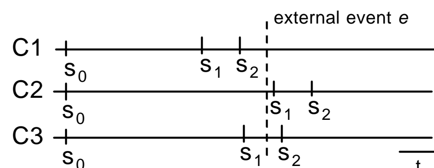
- **Problems**
  - **The Problems of *Replica Determinism***
  - **Non-deterministic behaviour**
  - **Limits of Redundancy**
- **Replica control**
  - **Internal vs. external**
  - **Centralized vs. distributed**
  - **Control strategies**
  - **Failure recovery**
  - **Redundancy preservation**
- **complexity**

The Problem of *Replica Determinism*

- For systematic fault-tolerance it is necessary that replicated components show consistent or deterministic behaviour in the absence of faults
- If for example two active redundant components are working in parallel, both have to deliver corresponding results at corresponding points in time
- This requirement is fundamental to differentiate between correct and faulty behaviour
- At a first glance it seems trivial to fulfil replica determinism since computer systems are assumed to be examples of deterministic behaviour, but
- in the following it is shown that computer systems behave *almost* deterministically

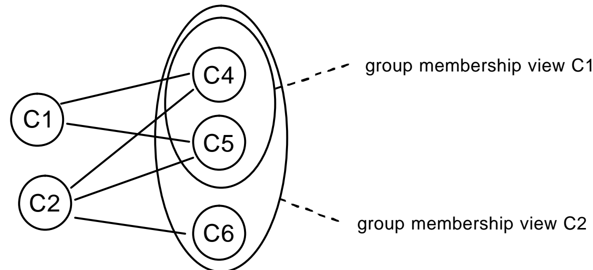
Non-deterministic Behaviour (1/6)

- **Inconsistent inputs:** If inconsistent input values are presented to the replicas then the results may be inconsistent too.
  - a typical example is the reading of replicated analogue sensors
    - *read*(C1) = 99.99 °C, *read*(C2) = 100.00 °C
- **Inconsistent order:** If service requests are presented to replicas in different order then the results will be inconsistent.



Non-deterministic Behaviour (2/6)

- **Inconsistent membership information:** Replicas may fail or leave groups voluntarily or new replicas may join a group. If replicas have inconsistent views about group membership it may happen that the results of individual replicas will differ.



Non-deterministic Behaviour (3/6)

- **Non-deterministic program constructs:** Besides intentional non-determinism, like random number generators, some programming languages have non-deterministic program constructs for communication and synchronization (Ada, OCCAM, …).
- **Ada example:**

```
task server is
 entry service_1();
 ...
 entry service_n();
end server;
```

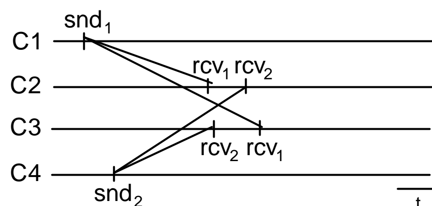
```
task body server is
begin
 select
  accept service_1() do
   action_1();
  end;
  ...
 or
  accept service_n() do
   action_n();
  end;
 end select;
end server;
```

Non-deterministic Behaviour (4/6)

- **Local information:** If decisions with a replica are based on local knowledge (information which is not available to other replicas) then the replicas will return different results.
  - system or CPU load
  - local time
- **Timeouts:** Due to minimal processing speed differences or due to slight clock drifts it may happen that some replicas locally decide to timeout while others do not.
- **Dynamic scheduling decisions:** Dynamic scheduling decides in which order a series of service requests are executed on one or more processors. This may cause inconsistent order due to:
  - non-identical sets of service requests
  - minimal processing speed differences

Non-deterministic Behaviour (5/6)

- **Message transmission delays:** Variability in the message transmission delays can lead to different message arrival orders at different servers (for point-to-point communication topologies or topologies with routing).



Non-deterministic Behaviour (6/6)

- **The consistent comparison problem:**
  - computers can only represent finite sets of numbers
  - it is therefore impossible to represent the real numbers exactly, they are rather approximated by equivalency classes
  - if the results of arithmetic calculations are very close to the border of equivalency classes, different implementations can return diverging results
  - different implementations are caused by: N-version programming, different hardware, different floating point libraries, different compilers
  - for example the calculation of $(a - b)2$ with floating point representation with a mantissa of 4 decimal digits and rounding where a = 100 and b = 0.005 gives different result for mathematical equivalent formulas.

    $(a - b)2 = 1.000\ 104$ $\qquad$ $(a - b)2 = a2 - 2ab + b2 = 9.999\ 103$

Limitations to Replication (1/2)

- **The *real world abstraction limitation*:**
  - dependable computer systems usually interface with continuos real-world quantities:
    - *quantity SI-unit*
    - distance meter [*m*]
    - mass kilogram [*kg*]
    - time second [*s*]
    - electrical current ampere [*A*]
    - thermodynamic temperature degree kelvin [*K*]
    - gramme-molecule mol [*mol*]
    - luminous intensity candela [*cd*]
  - these continuous quantities have to be abstracted (or represented) by finite sets of discrete numbers
  - due to the finite accuracy of any interface device, different discrete representations will be selected by different replicas

Limitations to Replication (2/2)

- **The *impossibility of exact agreement*:**
  - due to the real world abstraction limitation it is impossible to avoid the introduction of replica non-determinism at the interface level
  - but it is also impossible to avoid the once introduced replica non-determinism by agreement protocols completely
  - exact agreement would require ideal simultaneous actions, but in the best case actions can be only simultaneous within a time interval
- **Intention and missing coordination:**
  - replica non-determinism can be introduced intentionally
  - or unintentionally by omitting some necessary coordinating actions

Replica Control

Replica control

- Due to these fundamental limitations to replication it is **necessary** to enforce
- replica determinism which is called *replica control*.

> ***Replica Determinism:*** Correct replicas show *correspondence* of service outputs and/or service states under the assumption that all servers within a group start in the same initial state, executing *corresponding* service requests within a *given time interval*.

Remarks:

- this generic definition covers a broad range of systems
- *correspondence* and within a *given time interval* needs to be defined according to the application semantics

Internal vs. External Replica Control

- **Internal replica control:**
  - avoid non-deterministic program constructs, uncoordinated timeouts, dynamic scheduling decisions, diverse program implementations, local information, and uncoordinated time services
  - can only be enforced partially due to the fundamental limitations to replication
- **External replica control:**
  - control non-determinism of sensor inputs
  - avoid non-determinism introduced by the communication service
  - control non-determinism introduced by the program execution on the replicated processors by exchanging information

Groups and Replication Level

- Replicated entities such as processors are called **groups**.
- The number of replicas in a group is called **replication level**
- A group is said to be ***n*-resilient** if up to *n* processor failures can be tolerated

Group vs. hierarchical failure masking

- **Group failure masking:** The group output is a function of the individual group members output (e.g. a majority vote, a consensus decision). Thus failures of group members are hidden from the service user.
- **Hierarchical failure masking:** The processors within a group come up with diverging results and the faults are resolved by the service user one hierarchical level higher.

Basic Services for Groups

The basic services for replicated fault-tolerant systems

- **Membership:** Every non-faulty processor within a group has timely and consistent information on the set of functioning processors which constitute the group.
- **Agreement:** Every non-faulty processor in a group receives the same service requests within a given time interval.
- **Order:** Explicit service requests as well as implicit service requests, which are introduced by the passage of time, are processed by non-faulty processors of a group in the same order.

Central vs. Distributed Replica Control

- **Strictly central replica control:**
  - there is one distinguished processor within a group called *leader* or *central processor*
  - the leader takes all non-deterministic decisions
  - the remaining processors in the group, called *followers*, take over the leaders decisions
- **Strictly distributed replica control:**
  - there is no leader role, each processor in the group performs exactly the same way
  - to guarantee replica determinism the group members have to carry out a consensus protocol on non-deterministic decisions

Replica Control Strategies (1/4)

Lock-step execution:

- processors are executing in synchronous
- the outputs of processors are compared after each single operation
- typically implemented at the hardware level with identical processors

Advantages:

- arbitrary software can be used without modifications for fault-tolerance (important for commercial systems)

Disadvantages:

- common clock is single point of failure
- transient faults can affect all processors at the same point in the computation
- high clock speed limits number and distance of processors
- restricted failure semantics

Replica Control Strategies (2/4)

Active replication:

- all processors in the group are carrying out the same service requests in parallel
- strictly distributed approach, non-deterministic decisions need to be resolved by means of an agreement protocol
- the communication media is the only shared resource

Advantages:

- unrestricted failure semantics
- no single point of failure

Disadvantages:

- requires the highest degree of replica control
- high communication effort for consensus protocols
- problems with dynamic scheduling decisions and timeouts

Replica Control Strategies (3/4)

Semi-active replication:

- intermediate approach between distributed and centralized
- the leader takes all non-deterministic decisions
- the followers are executing in parallel until a potential non-deterministic decision point is reached

Advantages:

- no need to carry out a consensus protocol
- lower complexity of the communication protocol (compared to active replication)

Disadvantages:

- restricted failure semantics, the leaders decisions are single points of failures
- problems with dynamic scheduling decisions and timeouts

Replica Control Strategies (4/4)

Passive replication:

- only one processor in the group – called *primary* – is active
- the other processors in the group are in *standby*
- checkpointing to store last correct service state and pending service requests

Advantages:

- requires the least processing resources
- standby processors can perform additional tasks
- highest reliability of all strategies (if assumption coverage = 1)

Disadvantages:

- restricted failure semantics (crash or fail-stop)
- long resynchronization delay

Failures and Replication (1/2)

Centralized replication:

- semi-active and passive replication
- the leading processor is required to be fail restrained
- Byzantine or performance failures of the leader cannot be detected by other processors in the group ("heartbeat" or "I am alive" messages)
- to tolerate t failures with crash or omission semantics $t + 1$ processors are necessary
- the result of any processor (e.g. the fastest) can be used
- if no reliable broadcast service is available $2t + 1$ processors are necessary

Failures and Replication (2/2)

Distributed replication:

- active replication
- no restricted failure semantics of processors
- to tolerate t crash or omission failures $t + 1$ processors are necessary
- to tolerate t performance failures $2t + 1$ processors are necessary
- to tolerate t Byzantine failures $3t + 1$ processors are necessary
- for crash or omission failures it is sufficient to take 1 processor result
- for performance or Byzantine failure $t + 1$ identical results are required

Failure Recovery (1/2)

- After occurrence of a failure (that is covered by the fault hypothesis) the group has to perform some recovery actions

Centralized replication:
- failures of followers require no recovery actions
- if a leader fails a new leader needs to be elected
- then the new leader has to take over the service of the failed leader
- typically solved by backward recovery (reexecution from last fault free state)
- recovery time needs to be considered for real-time services
- window of vulnerability where new leader cannot decide whether the last output was made successfully or not
- output devices typically require at least once semantics (state semantics)

Failure Recovery (2/2)

Distributed replication:
- no special recovery actions necessary since all services are executed in parallel
- no election in case of processor failures
- output devices have to consider the results of all group members or each group member has its own output device (idempotence)
- no state semantics for output devices necessary (exactly once semantics possible)

Redundancy Preservation (1/2)

- to guarantee fault-tolerance and to cover the fault hypothesis the replication level has to be kept above a given threshold
- assuming n processors are in a group where f have failed and up to t failures have to be tolerated then one of the following combining conditions needs to be satisfied:
  - $n - f > 2t$ for Byzantine failures
  - $n - f > t$ for performance failures
  - $n - f > 0$ for crash or omission failures
- if this combining condition is violated
  - a new processor needs to be added to the group (redundancy preservation)
  - or the service of the group has to be abandoned

Redundancy Preservation (2/2)

- real-time requirements for redundancy preservation need special consideration
- faults in the reconfiguration service need to be considered
- f is therefore the number of failed processors plus the number of correct processors that are configured by a faulty reconfiguration service
- this requires a membership protocol:
  - detect departures and joins of processors to groups
  - provide consistent and timely group membership information on a system wide basis
- joins of processors are difficult to handle:
  - the new processor needs to be synchronized to the service state of the group
  - but the groups service state is evolving over time
  - after synchronization it has to be guaranteed that all further service requests are delivered to the new group member as well

Failure Coverage vs. Complexity

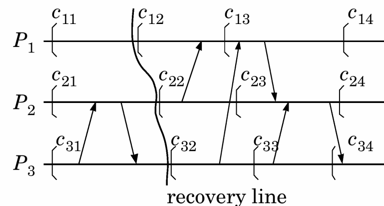
- High assumption coverage implies high complexity
  - for Byzantine faults the assumption coverage is 1
  - Byzantine faults require consensus protocols and very complex fault-tolerance mechanisms
  - high probability of faults in the fault-tolerance mechanisms (35% ESS-1)
  - due to the high complexity the system will have a low dependability
- Low assumption coverage implies low dependability
  - low assumption coverage implies high possibility of assumption violations
  - in case of assumption violations a fault-tolerant system can fail completely
  - the system will therefore have a low dependability
- for optimal dependability a ***compromise*** between the assumption coverage rate and complexity of the fault-tolerance mechanism has to be made

IV.2.6 Recovery

- systematic fault-tolerance is often based on backward recovery to recover a consistent state
- in distributed systems a state is said to be consistent if it could exist in an execution of the system
- Recovery line: A set of recovery points form a consistent state—called recovery line—if they satisfies the following conditions:
  (1) the set contains exactly one recovery point for each process
  (2) No orphan messages: There is no receive event for a message m before process Pi's recovery point which has not been sent before process Pj's recovery point.
  (3) No lost messages: There is no sending event for a message m before process Pi's recovery point which has not been received before process Pj's recovery point.

The Domino Effect

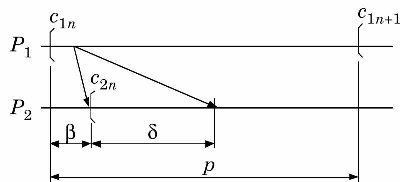
- the consistency requirement for recovery lines can cause a flurry of rollbacks to recovery points in the past



- to avoid the domino effect:
  - coordination among individual processors for checkpoint establishment
  - restricted communication between processors

Synchronous Checkpointing

- based on synchronized clocks check points are established with a fixed period p by all processes, where ß is the clock synchronization precision and d temporal uncertainty of message transmission
- if a message is sent during [T – ß – d, T] it will be received before T + ß + d



- to achieve a consistent state two possibilities exists:
  - prohibit message sending during interval ß after checkpoint establishment
  - establish checkpoint earlier, at kp – ß – d and log messages during the critical instant

Stable Storage (1/2)

- stable storage is an important building block for many operations in fault-tolerant systems (fail-stop systems, dependable transaction processing, …)
- there are two operations which should work correctly despite of faults (as covered by the fault hypothesis):
  - procedure writeStableStorage(address, data)
  - procedure readStableStorage(address) returns (status, data)
- many failures can be handled by coding (CRC's) but other types cannot be handled by this technique:
  - Transient failures: The disk behaves unpredictably for a short period of time.
  - Bad sector: A page becomes corrupted, and the data stored cannot be read.
  - Controller failure: The disk controller fails.
  - Disk failure: The entire disk becomes unreadable.

Stable Storage (2/2)

Disk shadowing

- a set of identical disk images is maintained on separate disks
- in case of two disks this technique is called disk mirroring
- for performance and availability reasons the disks should be "dual-ported" (e.g. Tandem system)

Redundant Array of Inexpensive Disks (RAID)

- data is spread over multiple disks by "bit-interleave" (individual bits of a data word are stored on different disks)
- in the following example single bit failures can be tolerated since a parity bit is stored on a check disk and disks are assumed to detect single bit failures
- RAID's provide high reliability and I/O throughput (parallel read/write)

Example: Fail Stop Processors

- the visible effects of the failure of a fail stop processor are:
  (1) It stops executing
  (2) The internal state of the processor and the volatile storage connected to the processor are lost; the state of the stable storage is unaffected.
  (3) Any processor can detect the failure of a fail stop processor.
- real processors do not have such a simple well defined semantics
- typically fail stop processors are implemented by a group of regular processors
- **k-fail-stop processor:** A processor is said to be k-fail-stop if it can tolerate up to k component (processor) failures while preserving its fail-stop property.

Fail Stop Processors with Stable Storage

Assumptions:

- the stable storage is reliable
- n + 1 normal processors
- communication is reliable
- message origin can be authenticated (point-to-point or cryptographic check)
- synchronous system model (synchronized clocks, bounded communication)

Implementation:

- requests to the stable storage are only granted if k + 1 requests are received within a time interval d:

S-process:

```
R:= bag of received requests with proper timestamp
if (|R| = n+1 ?  all requests are identical ?  ¬failed
    ?  all requests are from different processors)
  then
    if (request is write) then
      writeStableStorage
    elseif (request is read) then
      readStableStorage and send result to all processors
    fi
else // k-fail stop processor has failed
  writeStableStorage failed
fi
```

Fail Stop Processors without Stable Storage

Assumptions:

- the storage processor are not reliable and can fail byzantine
- k + 1 p-processors (program processors)
- 2k + 1 s-processors (storage processors)
- each s-process has a copy of the stable storage
- communication is reliable
- message origin can be authenticated (point-to-point or cryptographic check)
- synchronous system model

Implementation:

- requests to the stable storage subsystem are only granted if k + 1 requests are received within a time interval d
- failures of individual storage processors are masked by Byzantine agreement (under the assumption of authentification detectable failures)

IV.2.7 Summary

- Forms of Redundancy
- Techniques for Fault Tolerance
- Hardware Fault Tolerance Techniques
- Software Fault Tolerance Techniques
- Problems of Replication
- Recovery

Remember…

- Fault tolerance is not a system property.
- It is a technique by which dependability might be achieved.

Dependability/safety are the required system property.