Dependable Systems

Summary

Dr. Peter Tröger
Lena Herscheid
Dependability

- **Umbrella term** for operational requirements on a system

  - IFIP WG 10.4: "[..] the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers [..]"

  - IEC IEV: "dependability (is) the collective term used to describe the availability performance and its influencing factors: reliability performance, maintainability performance and maintenance support performance"

  - Laprie: "Trustworthiness of a computer system such that reliance can be placed on the service it delivers to the user"

- Adds a third dimension to system quality

- General question: How to deal with unexpected events?

- In German: 'Verlässlichkeit' vs. 'Zuverlässigkeit'
Course Topics

• Definitions and metrics
  • Fault, error, failure, fault models, fault tolerance, fault forecasting, ...
  • Reliability, availability, maintainability, error mitigation, damage confinement, ...
  • Reliability engineering, reliability testing, MTTF, MTBF, failure rate, ...

• Fault tolerance patterns
  • N-out-of-M, voting, heartbeat, ...

• Analytical evaluation
  • Reliability block diagrams, fault tree analysis, ...
  • Petri nets, Markov chains
  • Root cause analysis, risk analysis and assessment
Course Topics

• Hardware-Implemented Dependability
  • Failure rates, testing, hardware redundancy approaches

• Software-Implemented Dependability
  • Fault-tolerant programming: simplex, recovery blocks, checkpointing, roll-back, ...
  • Testing, software metrics
  • Fault-tolerant distributed systems: Transactions, consensus, clocks, ...

• Advanced approaches
  • Autonomic computing, rejuvenation, online failure prediction, performability

• Case studies from practice
Dependability Tree (Laprie)
Dependability Stakeholders

• **System** - Entity with function, behavior, and structure
  • A number of components or subsystems, which interact under the control of a design [Robinson]

• **Service** - System behavior abstraction, as perceived by the user

• **User** - Human or physical system that interacts with the systems service

• **Specification** - Definition of expected service and delivery conditions
  • On different levels, can lead to specification fault

• Reliance demands assessment of **non-functional dependability attributes**

• Provide ability for trustworthy service delivery by **dependability means**

• Undesired (maybe expected) circumstances form **dependability threats**
2 - Dependability Threats

• System **failure** - *Ausfall*
  • Event that occurs when the service no longer complies with the specification / deviates from the correct service.

• System **error** - *Fehler(zustand)*)
  • Part of system state that can lead to subsequent failure
  • Some sources define errors as active faults - not in this course ...

• System **fault** - *Fehler(ursache)*)
  • Adjudged or hypothesized cause of an error
  • Failure occurs when error state alters the provided service
  • Systems are build from connected components, which are again systems
  • Fault is the consequence of a failure of some other system to deliver its service
Chain of Dependability Threats (Avizienis)
Faults

• High diversity in possible sources and types
  
  • Fault nature
    
    • Accidental faults (‘Zufallsfehler’) vs. intentional faults (‘Absichtsfehler’)
    
    • Intentional faults are created deliberately, presumably malevolently
  
  • Fault origin viewpoints (not exclusive)
    
    • Phenomenological causes: Physical / natural faults vs. human-made faults
    
    • System boundaries: Internal faults (part of system state that produces an error) vs. external faults (interference with the environment)
    
    • Phase of creation: Design faults vs. operational faults
  
  • Temporal persistence
    
    • Permanent faults vs. temporary faults
System-Level Fault Model

- Fault model idea originates from hardware
  - How many faults of different classes can occur? What do I tolerate?
  - Timing of faults: Fault delay, repeat time, recovery time, ...
- Also mappable to software or even complete systems
  - Activities as black box, only look on input and output messages
- Link faults are mapped to the participating components
- Every participating component would need a fault model - pick the most urgent ones
Error Propagation

(C) Avizienis
Error Message Occurrence (Hansen & Siewiorek)

- Same fault can lead to different (detected or undetected) errors
- Errors become detected by error detection mechanism
  - Some undetected errors are detected by several detectors
  - Some detectors report several undetected errors as one
  - Some undetected errors are never uncovered
- Detected errors might not be logged, if the system stops too fast
Failures

• Non-compliance with the specification - arbitrary failure (‘willkürlicher Ausfall’)

• System failures can be further categorized in failure modes
  • Fail-silent / crash failure mode - incorrect results are not delivered
  • Fail-stop mode - constant value is delivered

• Failure mode domain - what is influenced
  • Service result - value failures
  • Service timeliness - timing failures
  • Service availability - stopping failures

• User perception in the mode - consistent / inconsistent for all users

• Failure mode consequences for ranking the identified issues
Failure Severity (‘Schweregrad des Ausfalls‘)

• Denotes consequences of failure

• **Benign failures** (‘unkritische Ausfälle‘)
  • Failure costs and operational benefits are similar
  • Sometimes also umbrella term for failures only detected by inspection
  • A system with only such failures is **fail-safe**

• **Catastrophic failures** (‘kritische Ausfälle‘)
  • Costs of failure consequences are much larger than service benefit

• **Significant / serious failures** - Intermediate steps expressing reduced service

• Grading of failure consequences on overall system depends on application
  • Flying airplane - Catastrophic stopping failure, Train - Benign stopping failure

• **Criticality** - Highest severity of possible failure modes in the system
Example: DO-178B Standard
3 - Means for Dependability

- **Fault prevention** - Prevent fault occurrence or introduction
- **Fault tolerance** - Provide service matching the specification under faults
- **Fault removal** - How to reduce the presence of faults
- **Fault forecasting** - Estimate the present number, future incidence, and the consequences of faults
- Combined utilization
Fault Tolerance

• Fault tolerance is the ability of a system to operate correctly in presence of faults.

or

• A system S is called **k-fault-tolerant** with respect to a set of algorithms \( \{A_1, A_2, \ldots, A_p\} \) and a set of faults \( \{F_1, F_2, \ldots, F_p\} \) if for every k-fault F in S, \( A_i \) is executable by a subsystem of system S with k faults. (Hayes, 9/76)

or

• Fault tolerance is the use of **redundancy** (time or space) to achieve the desired level of system dependability - costs!

• Accepts that an implemented system will not be fault-free

• Implements automatic recovery from errors

• Is a recursive concept (voter replication, self-checking checkers, stable memory)
Phases of Fault Tolerance (Hanmer)

Latent Fault → Error → Error Mitigation → Error Recovery → Normal Operation

Error Detection → Error Processing

Fault Activation
Fault Tolerance - Error Processing Through Recovery

- **Forward error recovery**
  - Error is masked to reach again a consistent state (*fault compensation*)
  - Corrective actions need detailed knowledge (*damage assessment*)
  - New state is typically computed in another way
    - Examples with compensation: Error correcting codes, non-journaling file system check, advanced exception handlers, voters

- **Backward error recovery**
  - Roll back to previous consistent state (*recovery point / checkpoint*)
  - Very suitable for transient faults
  - Computation can be re-done with same components (*retry*), with alternate components (*reconfigure*), or can be ignored (*skip frame*)
„Fail-Fast“

- A common concept from system engineering, company management, ...

- „Report failure and stop immediately without further action“
  - Discussed by Jim Gray in 1985 as part of his famous article „Why do computers stop and what can be done about it ?“

- Useful when benefit from recovery is not good enough for its costs, or if error propagation is highly probable
  - Single units of a redundant set
  - Deeply interwired IT system components
  - Components under heavy request load
  - And ... crappy start-up companies
4 - Fault Tolerance Patterns

• **Architectural patterns**
  • Considerations that cut across all parts of the system
  • Need to be applied in early design phase

• **Detection patterns**
  • Detect the presence of root faults, error states, and failures
  • Errors vs. failures -> a-priori knowledge vs. comparison of redundant elements

• **Error Recovery Patterns**
  • Methods to continue execution in a new error-free state
  • Undoing the error effects + creating the new state
4 - Fault Tolerance Patterns

• **Error Mitigation Patterns**
  
  • Do not change application or system state, but mask the error and compensate for the effects
  
  • Typical strategies for timing or performance faults

• **Fault Treatment Patterns**
  
  • Prevent the error from reoccurring by repairing the fault
  
  • System verification
  
  • Diagnosis of fault location and nature
  
  • Correction of the system and / or the procedures
5 - Attributes of Dependability

• Non-functional attributes such as reliability and maintainability

• Complementary nature of viewpoints in the area of dependability

• In comparison to functional properties
  • ... hard to define
  • ... hard to abstract
  • ... 'Divide and conquer' does not work as good
  • ... difficult interrelationships
  • ... often probabilistic dependencies
In Detail

• **Reliability** - Function $R(t)$
  
  • Probability that a system is functioning properly and constantly over time period $t$
    
    • Assumes that system was fully operational at $t=0$
    
    • Denotes failure-free interval of operation

• **Availability** - Statement if a system is operational at a point in time / fraction of time
  
  • Describe system behavior in presence of error treatment mechanisms

• **Instantaneous availability (at t)** - Probability that a system is performing correctly at time $t$; equal to reliability for non-repairable systems: $A_i(t) = R(t)$

• **Steady-state availability** - Probability that a system will be operational at any random point of time, expressed as the fraction of time a system is operational during its expected lifetime: $A_s = Uptime \ / \ Lifetime$
Why Exponential?

- Distribution function that models the **memoryless property** of the Poisson process
  
  - \( P(T > t + s | T > t) = P(T > s) \), e.g. \( P_{\text{Failure}}(5 \text{ years} | T > 2 \text{ years}) = P_{\text{Failure}}(3 \text{ years}) \)

- Failure is not the result of wear-out

- Models 'intrinsic failure' behavior, assumed for the majority of hardware life time

- Weibull distribution as alternative, can also model **tear-in** and **wear-out**

- Some natural phenomena have constant failure rate (e.g. cosmic ray particles)
Variable Failure Rate in Real World

- Failure rate is treated as constant parameter of the exponential distribution
- (maybe invalid) simplification, mostly combined solution in practice:
  - Exponential distribution when failure rate is constant
  - Weibull distribution when failure rate is monotonic decreasing / increasing
Steady-State Availability

- **Mean time to failure (MTTF)** -
  Average time it takes to fail
  \(\rightarrow\) average uptime

- **Mean time to recover / repair (MTTR)** -
  Average time it takes to recover

- **Mean time between failures (MTBF)** -
  Average time between two successive failures

- **Availability** = Uptime / Lifetime
  \(=\) MTTF / MTBF

\[
\text{MTBF} \quad \text{MTTF}
\]

\[
\text{up} \quad \text{down} \quad \text{up} \quad \text{down} \quad \text{up}
\]
Steady-State Availability

\[ A = \frac{\text{Uptime}}{\text{Uptime} + \text{Downtime}} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \]

<table>
<thead>
<tr>
<th>Availability</th>
<th>Downtime per year</th>
<th>Downtime per week</th>
</tr>
</thead>
<tbody>
<tr>
<td>90.0 % (1 nine)</td>
<td>36.5 days</td>
<td>16.8 hours</td>
</tr>
<tr>
<td>99.0 % (2 nines)</td>
<td>3.65 days</td>
<td>1.68 hours</td>
</tr>
<tr>
<td>99.9 % (3 nines)</td>
<td>8.76 hours</td>
<td>10.1 min</td>
</tr>
<tr>
<td>99.99 % (4 nines)</td>
<td>52.6 min</td>
<td>1.01 min</td>
</tr>
<tr>
<td>99.999 % (5 nines)</td>
<td>5.26 min</td>
<td>6.05 s</td>
</tr>
<tr>
<td>99.9999 % (6 nines)</td>
<td>31.5 s</td>
<td>0.605 s</td>
</tr>
<tr>
<td>99.99999 % (7 nines)</td>
<td>0.3 s</td>
<td>6 ms</td>
</tr>
</tbody>
</table>
MTTR << MTTF [Fox]

• Armando Fox on ′Recovery-Oriented Computing′
  
  • A = MTTF / (MTTF + MTTR)
    
    • 10x decrease of MTTR as good as 10x increase of MTTF ?
    
    • MTTF′s are not claimable, but MTTR claims are verifiable
    
    • Proving MTTF numbers demands system-years of observation and experience
    
    • Lowering MTTR directly improves user experience of one specific outage, since MTTF is typically longer than one user session
    
  • HCI factor of failed system
    
    • Miller, 1968: >1sec “sluggish”, >10sec “distracted” (user moves away)
    
    • 2001 Web user study: ~5sec „acceptable“, ~10sec „excessively slow“
6 - Dependability Modeling

- Default approach: Utilize a formalism to model system dependability
  - Quantify the availability of components, calculate system availability based on this data and a set of assumptions (the availability model)
    - Most models expose the same expressiveness
    - Each formalism allows to focus on certain aspects
    - Structure-based models: Reliability block diagram, fault tree
    - State-based models: Markov chain, petri net
- System understanding evolved from hardware to software to IT infrastructures
  - Example: Organization management influence on business service reliability
    - Information Technology Infrastructure Library (ITIL)
    - CoBiT(Control Objectives for Information and related Technology)
Dependability Modeling

- The *Failure Space-Success Space* concept
  - Often easier to agree on what constitutes a system failure
  - Success tends to be associated with system efficiency, which makes it harder to formulate events in the model („The car drives fast.“, „The car stops driving.“)
  - In practice, there are more ways to success than to failure
Dependability Modeling

• System analysis approaches

  • **Inductive methods** - Reasoning from specific cases to a general conclusion
    • Postulate a particular fault or initiating event, find out system effect
    • Determine what system (failure) states are possible
    • Trivial approach: „parts count“ method
    • Examples: Failure Mode and Effect Analysis (FMEA), Preliminary Hazards Analysis (PHA), Event Tree Analysis, Reliability Block Diagrams (RBD), ...

  • **Deductive methods** - Postulate a system failure, find out what system modes or component behaviors contribute to this failure
    • Determine how a particular system state can occur
    • Examples: Fault Tree Analysis (FTA)
Examples

\[ \phi_S = c_{LB} \land (c_{WS1} \lor c_{WS2}) \land (c_{DB1} \lor c_{DB2}) \]

\[ A_{site} = a_{LB} \times A_{WS\text{set}} \times A_{DB\text{set}} \]

\[ = a_{LB} \times [1 - (1 - a_{WS})^{n_{WS}}] \times [1 - (1 - a_{DB})^{n_{DB}}] \]
Reliability Block Diagrams (RBD)

• Model logical interaction for success-oriented analysis of system reliability

• Building blocks: **series structure, parallel structure, k-out-of-n structure**

• System is available only if there is a path between s and t

• Granularity based on data and **lowest actionable item** concept

• Structure formula can be obtained from RBD by identifying the **subset of nodes that disconnects s from t** if removed

---

![Diagram](image-url)
Deductive Analysis - Fault Trees

• Structure analysis effort grows exponentially with the number of components

• Fault Trees
  • Invented 1961 by H. Watson (Bell Telephone Laboratories)
    • Facilitate analysis of the launch control system of the intercontinental Minuteman missile
  • Used by Boeing since 1966, meanwhile adopted by different industries
  • Root cause analysis, risk assessment, safety assessment

• Basic idea
  • Technique for describing the possible ways in which an undesired system state can occur
  • Complex system failures are broken down into basic events
7 - State-Based Modeling

- Structural vs. state-based modeling
- State Transition Diagrams
- Markov Chains
- Petri Nets

The complexity of the petri net does not depend on the number of components!

Availability = P[#up >= 2]
Real World Systems → Model → Solution Technique → Evaluations

Input Parameters

Modeling Errors
- Structural Errors: Initial state, missing or extra states / transitions
- Error Propagation Model
- Parametric Models: Failure and repair rates, coverage parameters
- Errors due to non-independence

Solution Errors
- Approximation Errors: System partition, state aggregation
- Numerical Errors: Truncation, Round-off
- Programming Errors
- Estimation Errors: Stochastic estimators, not enough data

Parametric Errors
- Different component parameter sources
- Projected stress factors assume unrealistic operational conditions

Errors due to non-independence
8 - Qualitative Dependability Investigation

• Different approaches that focus on structural (qualitative) system evaluation
  • Root cause analysis
    • Broad research / industrial topic targeting error diagnosis
    • Specialized topic in quality methodologies
  • Development process investigation
    • Procedures for ensuring industry quality in production
    • Software development process
  • Organizational investigation
    • Non-technical influence factors on system reliability
FMEA

• **Failure Mode and Effects Analysis**

• Engineering quality method for early concept phase - identify and tackle weak points

• Introduced in the late 1940s for military usage (MIL-P-1629)
  
  • Later also used for aerospace program, automotive industry, semiconductor processing, software development, healthcare, ...

• Main goal is to identify and prevent critical failures

• Performed by cross-functional team of **subject matter experts**

• Most important task in many reliability programs
  
  • Six Sigma certification, reliability-centered maintenance (RCM) approach
  
  • Automotive industry (ISO16949, SAE J1739)

• Medical devices
Hazard & Operability Studies (HAZOPS)

• Process for identification of potential hazard & operability problems, caused by deviations from the design intend
  • Difference between deviation (failure) and its cause (fault)
  • Conduct intended functionality in the safest and most effective manner
  • Initially developed to investigate chemical production processes, meanwhile for petroleum, food, and water industries
  • Extended for complex (software) systems

• Qualitative technique
  • Take full description of process and systematically question every part of it
  • Assess possible deviations and their consequences
  • Based on guide-words and multi-disciplinary meetings
Root Cause Analysis

• What caused the fault? - Starting point of dependability chain
  • Peeling back the layers
  • Must be performed systematically as an investigation
  • Establish sequence of events / timeline
Reliability Models for IT Infrastructures

- System reliability in a commercial environment is determined by many factors:
  - Software and hardware reliability
  - Training of maintenance personnel
  - „Business processes“ how maintenance is handled
  - The way the IT department is organized
  - ...
- Impact of management organization on reliability is an emerging research field
- Standards for IT organization, based on best practices
  - Describe which processes have to be established in an IT department
  - Provide reference models for organization of IT department
CMMI Maturity Levels

Characteristics of the Maturity levels

- **Level 1: Initial**
  - Processes unpredictable, poorly controlled and reactive

- **Level 2: Managed**
  - Processes characterized for projects and is often reactive.

- **Level 3: Defined**
  - Processes characterized for the organization and is proactive. (Projects tailor their processes from organization's standards)

- **Level 4: Quantitatively Managed**
  - Processes measured and controlled

- **Level 5: Optimizing**
  - Focus on process improvement

(C) Wikipedia
ITIL

- Information Technology Infrastructure Library (ITIL), latest version v3
  - Started as set of recommendations by the UK Government
  - Concepts and guides for IT service management
  - Supports to deliver business-oriented quality IT services
  - Core publications: Service Strategy, Service Design, Service Transition, Service Operation, Continual Service Improvement
- Broad tool support from vendors
- High costs for certification and training
- Methodology sometimes over-respected at the expense of pragmatism
9 - Predicting System Reliability

• Feasibility evaluation for given model, identification of potential problem sources

• Comparison of competing designs

• Identification of potential reliability problems - low quality, over-stressed parts

• Input to other reliability-related tasks
  • Maintainability analysis, testability evaluation
  • Failure modes and effects analysis (FMEA)

• Ultimate goal is the prediction of system failures by a reliability methodology

• Approach depends on nature of component (electrical, electronic, mechanical)
Reliability Data

- **Field (operational) failure data**
  - Meaningful source of information, experience from real world operation
  - Operational and environmental conditions may be not fully known

- **Service life data with / without failure**
  - Helpful in assessing time characteristics of reliability issues

- **Data from engineering tests**
  - Example: Accelerated life tests
  - Results from controlled environment
  - Trustworthy for analysis purposes

- Lack of failure information is the ‘greatest deficiency‘ in reliability research [Misra]
Reliability Prediction Models

• Economic requirements affect field-data availability
  • Customers do not need generic field data, so collection is extra effort
• Numerical reliability prediction models available for specific areas
  • **Hardware-oriented reliability modeling**
    • MIL-HDBK-217, Bellcore Reliability Prediction Program, ...
  • **Software-oriented reliability modeling**
    • Jelinski-Moranda model, Basic Execution model, software metrics, ...
• Prediction models look for corrective factors, so they are always focused
• Demands confidence in understanding of environmental conditions
• Reconciliation is often needed between the different values of failure data from various sources
Software Reliability Assessment

- Software bugs are permanent faults, but behave as transient faults
  - Activation depends on software usage pattern and input values
  - Timing effects with parallel or distributed execution
  - Software aging effects
- Fraction of system failures reasoned by software increased in the last decades
- Possibilities for software reliability assessment
  - **Black box models** - No details known, observations from testing or operation
    - Reliability growth models
  - **Software metric models** - Derive knowledge from static code properties
Halstead Metric

- Statistical approach - Complexity is related to number of operators and operands
- Only defined on method level, OO code analysis must consider additional structure
  - Program length: $N = \text{Number of operators } N_{OP} + \text{total number of operands } N_{OD}$
  - Vocabulary size: $n = \text{Number of unique operators } n_{OP} + \text{number of unique operands } n_{OD}$
  - Program volume: $V = N \times \log_2(n)$
    - Describes size of an implementation by vocabulary and (mainly) by program length
    - In-place changes are ok ...
  - Difficulty level: $D = (n_{OP} / 2) \times (N_{OD} / n_{OD})$, program level $L = 1 / D$
    - Error proneness is proportional to number of unique operators
      -> since most languages only offer a few, this should be small
    - Also proportional to the level of operand reuse through the code

```
x = x + 1;
```

$N_{OP}=n_{OP}=3$

$N_{OD}=3$

$n_{OD}=2$
10 - Distributed Systems Theory

• Fallacies of Distributed Computing
  • The network is reliable, Latency is zero, Bandwidth is infinite, The network is secure, Topology doesn’t change, There is one administrator, Transport cost is zero, The network is homogeneous

• Timing Models
  • Logical time, Lamport clocks

• Fault Models

• Consensus Problems
  • 2PC
  • Paxos
11 - Fault-Tolerant Distributed Systems

• Consistency Models
  • Strict consistency, sequential consistency, eventual consistency

• Trade-Offs
  • Brewer, CAP Theorem

• Replication
  • State Machine Replication, …
12 - Dependable Distributed Applications

- Frameworks - Erlang / FT-CORBA
- Coordination Services
  - Chubby, Zookeeper
- Distributed Storage
  - Cassandra, RIAK, HDFS …

![Diagram of Distributed Systems](image-url)
13 - Hardware Diagnosis

• Contains of fault detection (what ?) and fault location (where ?)

• Fault detection
  • Replication checks - Test operation / execution against alternate implementation
  • Timing checks - Test operation / execution against timing constraints
  • Reversal checks - Make use of operation reversibility
  • Coding checks - Utilize redundant (but different) representation of data
  • Reasonableness checks - Check against known system / data properties
  • Structural checks - Ensure consistent structure of data, diagnostic tests, ...
  • Diagnostic checks - Use set of inputs for which the outputs are known
  • Algorithmic checks - Check invariants of an algorithm
14 - Hardware Redundancy

• Redundancy for **error detection** and **forward error recovery**

• Redundancy types: **spatial**, **temporal**, **informational** (presentation, version)
  - Redundant not mean identical functionality, just perform the same work

• **Static redundancy** implements error mitigation
  - Fault does not show up, since it is transparently removed
  - Examples: Voting, error-correcting codes, N-modular redundancy

• **Dynamic redundancy** implements error processing
  - After fault detection, the system is reconfigured to avoid a failure
  - Examples: Back-up sparing, duplex and share, pair and spare

• **Hybrid approaches**
Sphere of Replication

- Components outside the sphere must be protected by other means
- Level of output comparison decides upon fault coverage
- Larger sphere tends to decrease the required bandwidth on input and output
  - More state changing happens just inside the sphere
- Vendor might be restricted on choice of sphere size
Masking / Static Redundancy: Voting

- **Exact voting:** Only one correct result possible
  - **Majority vote** for uneven module numbers
  - **Generalized median voting** - Select result that is the median, by iteratively removing extremes
  - **Formalized plurality voting** - Divide results in partitions, choose random member from the largest partition

- **Inexact voting:** Comparison at high level might lead to multiple correct results
  - **Non-adaptive voting** - Use allowable result discrepancy, put boundary on discrepancy minimum or maximum (e.g. 1,4 = 1,3)
  - **Adaptive voting** - Rank results based on past experience with module results
    - Compute the correct value based on „trust“ in modules from experience
    - Example: Weighted sum \( R = W_1 \cdot R_1 + W_2 \cdot R_2 + W_3 \cdot R_3 \) with \( W_1 + W_2 + W_3 = 1 \)
N-Modular Redundancy (with perfect voter)

\[ R_{NMR} = \sum_{i=0}^{m} \binom{N}{i} (1 - R)^i R^{N-i} \]

\[ \binom{n}{k} = \frac{n!}{k!(n-k)!} \]

\[ R_{2-of-3} = \binom{3}{0}(1 - R)^0 R^3 + \binom{3}{1}(1 - R)R^2 \]

\[ R_{2-of-3} = R^3 + 3(1 - R)R^2 \]

\[ R_{3-of-5} = \ldots \]
Pair and Spare

- Special cases for combination of duplex (with comparator) and sparing (with switch)

- **Pair and spare** - Multiple duplex pairs, connected as standby sparing setup
  - Two replicated modules operate as duplex pair (lockstep execution), connected by comparator as voting circuit
  - Same setting again as spare unit, spare units connected by switch
  - On module output mismatch, comparators signal switch to perform failover
  - Commercially used, e.g. Stratus XA/R Series 300
Hybrid Approaches

• **N-modular redundancy with spares**
  
  • Also called *hybrid redundancy*
  
  • System has basic NMR configuration
  
  • Disagreement detector replaces modules with spares if their output is not matching the voting result
  
  • Reliability as long as the spare pool is not exhausted
  
  • Improves fault masking capability of NMR

  • Can **tolerate two faults with one spare**, while classic NMR would need 5 modules with majority voting to tolerate two faults
Coding Checks in Memory Hardware

- **Primary memory**
  - Parity code
  - $m$-out-of-$n$ resp. $m$-of-$n$ resp. $m/n$ code
  - Checksumming
  - Berger Code
  - Hamming code

- **Secondary storage**
  - RAID codes
  - Reed-Solomon code
Memory Redundancy

- Standard technology in DRAMs
  - Bit-per-byte **parity**, check on read access
  - Implemented by additional parity memory chip
  - **ECC** with Hamming codes - 7 check bits for 32 bit data words, 8 bit for 64 bit
    - Leads to 72 bit data bus between DIMM and chipset
    - Computed by memory controller on write, checked on read
    - Study by IBM: ECC memory achieves R=0.91 over three years
    - Can correct single bit errors and detect double bit errors
  - Hewlett Packard **Advanced ECC** (1996)
    - Can detect and correct single bit and double bit errors
RAID

- **Redundant Array of Independent Disks (RAID)** [Patterson et al. 1988]
  - Improve I/O performance and/or reliability by building *raid groups*
  - Replication for information reconstruction on disk failure (*degrading*)
  - Requires computational effort (dedicated controller vs. software)
  - Assumes failure independence
Parity With XOR

- Self-inverse operation
  - 101 XOR 011 = 110, 110 XOR 011 = 101

<table>
<thead>
<tr>
<th>Disk</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 0 0 1 0 0 1 1</td>
</tr>
<tr>
<td>2</td>
<td>0 1 1 0 1 1 1 0 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 1 0 0 1 1 1</td>
</tr>
<tr>
<td>4</td>
<td>1 1 1 0 1 0 1 1 1</td>
</tr>
<tr>
<td>Parity</td>
<td>0 1 0 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disk</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 0 0 1 0 0 1 1</td>
</tr>
<tr>
<td>Parity</td>
<td>0 1 0 1 1 1 1 1 1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 1 0 0 1 1 1</td>
</tr>
<tr>
<td>4</td>
<td>1 1 1 0 1 0 1 1 1</td>
</tr>
<tr>
<td>Hot Spare</td>
<td>0 1 1 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>
15 - Software Dependability

• Fault elimination
  • Reduce number of dormant faults at development time

• Fault-tolerant software
  • Techniques to achieve fault tolerance for software faults
  • Application of redundancy idea to software modules

• Software fault tolerance
  • Techniques to achieve fault tolerance by software mechanisms
  • Typically for hardware failures on lower levels in the system stack
  • Redundancy managed by operating system, cluster framework, application code
Fault Elimination through Software Testing

- **Component Testing**
  - Object Class Testing
  - Interface Testing
- **System Testing**
  - Integration Testing
  - Release Testing
    - Performance Testing
    - Stress Testing

Responsibility of the component developer, based on experience.

Responsibility of an independent testing team, based on specification.

Access to source code, tested while components are integrated.

Steadily increasing load.

Excess maximum design load.

(C) Ian Sommerville
Testing Through Software Fault Injection

• Intentionally trigger erroneous behavior of the execution environment
  • Typical approach for communicating software entities
  • Orientation towards implementation details - program state, functional behavior
  • Several non-intrusive implementation techniques, such as AOP
  • Injection can be done as part of execution under real conditions
  • Huge variety of fault classes, including SWIFI fault classes
  • New approaches support remote fault injection (e.g. fuzzing)

• **Compile-time injection:** Leads to erroneous image being executed

• **Run-time injection:** Demands some altering of application state during runtime

• Typical triggers: Time-out, exception, debugging trap, code insertion
Software Fault Model [Goloubeva]

- Example: Control loop writing computation result to a variable
  - **Temporary fault**: Write NULL to result variable after end of usage
  - **Permanent fault**: Compute and store incorrect output from input data

- Single vs. multiple faults depends on granularity level of investigation

- **On source code level**
  - Program: Structured collection of features with syntax and semantic
  - Syntax allows to describe fault model
  - Negation of semantic properties defines a fault model
  - Examples: Calling non-existent functions, Functions not returning a value, values out of their type range, missing input parameters

- **On executable level**, e.g. stack overflow due to recursion
Fault-Tolerant Software - Another categorization [Lyu 95]

- **Single version techniques**
  - Add mechanisms for detection, containment, and handling of errors to the software component itself
  - Examples: Software structure and actions approaches, error detection, exception handling, checkpointing and restart, process pairs, data diversity

- **Multi version techniques**
  - Rely on structured utilization of variants of the same software
  - Examples: Recovery blocks, N-version programming
  - Principles can be applied to any software layer
    - Identify source of most design faults
    - Typically no problem with parallel application, beside cost factor
Single-Version Approaches - Wrapper

• Piece of software that encapsulates a given program when it is being executed

• Typical approach for operating systems and middleware stacks

• Structure: **Wrapper software** and **wrapped entity**

• Inputs and outputs are checked by the wrapper

• Examples:
  
  • Dealing with buffer overflow, checking scheduler correctness (e.g., EDF), bypassing known bugs, checking output correctness

  • When pre- or postconditions are violated, usually an exception is being raised

  • Wrapper forms an **acceptance test** in the **dependability rings**

• Good approach for fixing issues in the operational phase of software
Single Version Approaches - Checkpointing

- Save application state data at recovery points
  - Can be reloaded on crash or any other kind of data loss
  - Possible on different levels: local per process, partial, complete, distributed
- Optimum checkpointing interval
  - Checkpointing too frequent: Majority of time spent for data saving
  - Checkpointing too rare: May take long time to recover
- Several specialized solutions for C / C++ language, easier with reflection support
- Popular approach in clusters / high-performance computing
  - Latest trend: In-memory checkpointing

taken from Software Fault Tolerance: A Tutorial
Single Version Approaches - Data Diversity [Ammann 88]
Single Version Approaches -
High-Level Instruction Duplication [Goloubeva]

- Introduce data and code redundancy through high-level transformation
  - Duplicate every variable
  - Perform every write operation on both copies of the variable
  - After each read operation, the copies must be checked for consistency
    - Should be close to read operation, in order to avoid error propagation
    - Includes also expression evaluation
  - Procedure parameters treated as variables
- Independent from underlying hardware, targets cache / main memory faults

```
a=b;
.. becomes ...
a0=b0;
a1=b1;
if (b0 != b1)
  error();
...
```

```
a=b+c;
... becomes ...
a0=b0+c0;
a1=b1+c1;
if ((b0!=b1) || (c0!=c1))
  error();
```
Control Flow Error

- Control flow error (CFE) leads to unexpected instruction execution

- Terminology:
  - Basic block (BB) - branch-free serial code fragment
    - Branch / jump instruction is modeled as last instruction of a basic block
  - Control flow graph (CFG) - one node per basic block
  - Illegal branch - Node transition is not part of the CFG
  - Wrong branch - Node transition is already part of the CFG
  - Inter-block error - Erroneous branch to different block
  - Intra-block errors - Erroneous branch inside a block

```c
i=0;
while (i<n) {
    if (a[i] < b[i])
        x[i] = a[i];
    else
        x[i] = b[i];
    i++;
}
```
Multi-Version Approaches
Recovery Blocks

- Redundant system implementations are typically used simultaneously, best answer is picked i.e. by voting

- Alternative way: Sequential execution of recovery blocks
  - Introduced in 1974 by Horning et. al.
  - Dynamic fault tolerance approach, related to stand-by sparing in hardware

```python
establish Checkpoint Primary Module
Acceptance Test, else load Checkpoint
Alternative Module 1
Acceptance Test, else load Checkpoint
Alternative Module 2
... else Failure Exception
```
Multi-Version Approaches - N-Version Programming

- Common mode errors are only catchable by **design diversity**
- Design diversity is a complex issue
  - Design philosophies, software tools, programming languages, test philosophies
- Typical approaches try to utilize randomness - separate teams on different locations

**N-Version Programming**

- Suggested by Elmendorf in 1972, developed by Avizienis & Chen in 1977
- Static approach, combination of decision mechanism and forward recovery
- At least two independently designed and functionally equivalent variants
- Variants are executed in parallel, decision mechanism selects the „best“ result
- Can support reliability, but also system security against **malicious logic**
Simplex

- Idea: Using simplicity to control complexity
  - Forward recovery approach, based on feedback loop
  - **HAC subsystem** - Simple construction, formal methods, reliable hardware
  - **HPC subsystem** - Complex technology, advanced features

- HPC can use HAC output, but not vice versa
- Decision logic based on control loop output
- Typically performance degradation with HAC
- Example: Boeing 777 primary and secondary flight controller