Dependable Systems

Fault Tolerance Patterns

Dr. Peter Tröger

Source:

Phases of Fault Tolerance (Hanmer)

- Latent Fault
- Error Detection
- Error
- Error Mitigation
- Error Recovery
- Error Processing
- Normal Operation
- Fault Treatment
Design Pattern

• Definition from software engineering: „A general reusable solution to a commonly occurring problem“
  • No finished / directly applicable solution, but a template
  • On the level of components and interactions
• Popular approach in computer science (Gang of Four, Portland Pattern Repository)
• Fault tolerance patterns
  • ... might be suited for stateless / stateful / both kinds of system
  • ... are based on observers and monitors (humans / computers)
  • ... work orthogonal to primary function
• Note: ‘Hamner’-Book is about software fault tolerance, but the patterns are generic (enough)
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
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
Architectural Patterns
Architectural Patterns -
Units of Mitigation

• Only parts of the system should potentially get into error state

• Design *units of mitigation* that contain errors and their error recovery mechanism

• Tradeoff:
  Component size vs. bookkeeping overhead vs. fault tolerance options

• Should contain independent atomic actions without communication focus

• Hints for granularity
  • Architectural style (n-tier)
  • Functional and resource (memory, CPU) boundaries
  • Choice of recovery action (e.g. restart)

• Should perform **self checks and fail silently**, act as barrier to an error state

• Units without any recovery / mitigation possibility are too small
Architectural Patterns - Error Containment Barrier

- Errors spread through several mechanisms - messages, memory, follow-up actions
- *Unit of mitigation* boundary implemented by *error containment barrier*
  - Treated as *separate system component*
  - Barrier must encapsulate error state, should trigger recovery / mitigation
  - In best case, perform detection close to the fault (structural proximity / time)
- Hardware: Isolate faulty components by state bit
- *Babbling idiot* problem
  - Suspicious nodes should never be in control of the communication bus
- *Bus guardian* as barrier implementation
Guardian Example: Temporal Firewall in the Time-Triggered Architecture (TTA)
Architectural Patterns - Correcting Audits

- Data element corruption can occur on hardware level (external physical faults) and software level (data types, currencies, pointers, ...)

- Auditing data demands correctness criteria
  - Structural properties of the data structure (linked lists, pointer boundaries, ...)
  - Known correlations (multiple locations, known conversion factors, cross linkage)
  - Sanity checks (value boundaries, checksums)
  - Direct comparison (duplication, mostly of static data)

- Automatic correction is usually easy, but must consider item consistency

- Actions: Correction, logging, resume execution

- Errors from faulty data easily propagate, common audit infrastructure helps
Architectural Patterns - Redundancy

• Improving availability by reducing MTTR is the easiest way

• Error recovery phase makes the effect undone, but must be short

• Idea: Resume execution before bad effects are undone, by using identical copy
  • Accomplish the same work on different hardware / software
  • Does not mean identical functionality, just perform the same work
  • Quick activation of redundant feature needed

• Redundancy types: spatial, temporal, informational (presentation, version)
  • Special issues with software redundancy regarding deterministic behavior

• Redundancy for performance improvement, availability then by excess capacity

• Example: Checkpointing vs. fail-over
Spatial Redundancy through Replication

• **Replication**: Process of ensuring consistency between redundant resources
  • Mostly applied for data replication
  • *Active (synchronous) replication* performs the same activity on every replica
    • First introduced by Leslie Lamport as *state machine replication*
    • Demands a deterministic processing of activities
  • *Passive replication* performs activity on one replica, and transmits the delta
    • *Primary server vs. backup servers*
    • Delayed response in failover case
    • Works also for non-deterministic processes
  • Example: Master-Slave vs. Master-Master replication setup
Example: VAX Spatial Hardware Redundancy
Example: Persistency in Redundant Systems

**Shared Disk**

CPU
Memory
SAN

**Shared Nothing**

Network

CPU
Memory

Network
## Example: Persistency in Redundant Systems

<table>
<thead>
<tr>
<th></th>
<th>Shared Disk / Multi-Homing</th>
<th>Shared Nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
<td>• Good availability</td>
<td>• Very good availability</td>
</tr>
<tr>
<td></td>
<td>• Good load-balancing</td>
<td>• Unlimited scalability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Low cost due to standard components</td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
<td>• Limited scalability</td>
<td>• Difficult for load balancing</td>
</tr>
<tr>
<td></td>
<td>• Synchronization for concurrent update</td>
<td>• Difficult for performance optimization</td>
</tr>
</tbody>
</table>
Example: PostgreSQL 9 Redundancy Options

- **Shared-Disk setup**
  - Avoids synchronization overhead, but demands network storage resp. file system
  - Mutual access exclusion from active / passive node must be ensured

- **Shared-Nothing setup**
  - **Block-device replication** - Operating system can mirror file system modifications (e.g. GFS, DRBD)
  - **Point-In-Time Recovery (PITR)** - Passive nodes receive stream of write-ahead log (WAL) records, after each transaction commit
  - **Master-Slave / Multimaster Replication** - Batch updates on table granularity
  - **Statement-Based Replication Middleware** - SQL is sent to all nodes
**Example: PostgreSQL 9 Redundancy Options**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Shared Disk Failover</th>
<th>File System Replication</th>
<th>Hot/Warm Standby Using PITR</th>
<th>Trigger-Based Master-Slave Replication</th>
<th>Statement-Based Replication Middleware</th>
<th>Asynchronous Multimaster Replication</th>
<th>Synchronous Multimaster Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most Common Implementation</td>
<td>NAS</td>
<td>DRBD</td>
<td>PITR</td>
<td>Slony</td>
<td>pgpool-II</td>
<td>Bucardo</td>
<td></td>
</tr>
<tr>
<td>Communication Method</td>
<td>shared disk</td>
<td>disk blocks</td>
<td>WAL</td>
<td>table rows</td>
<td>SQL</td>
<td>table rows</td>
<td>table rows and row locks</td>
</tr>
<tr>
<td>No special hardware required</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allows multiple master servers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No master server overhead</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No waiting for multiple servers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Master failure will never lose data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slaves accept read-only queries</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Per-table granularity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No conflict resolution necessary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Architectural Patterns - Humans

- **Minimize Human Interaction**
  - Error state root causes: Hardware, Software, Procedural / Operational
  - **Humans are bad** in: Long series of steps, routine tasks, operation, response time
  - Reduce failure risk due to mistakes in error treatment
    -> process errors automatically
  - Operational staff should be able to monitor, but not be required for the solution

- **Maximize Human Participation**
  - System should support experts in contributing to an error solution
    - **Humans are good** in drawing meaning from sequence of unrelated events
  - Examples: Reporting prioritization, context information (timestamp etc.)
  - Safe mode: Wait for human participation
Architectural Patterns - Maintenance Interface

- **Making maintenance task visible** to the outside world - additional form of input
- Separated interfaces and handling needed
  - Shed load approach or any other overload defense will affect operator
  - Intermixed interfaces might bring security problems
- Not a hidden trap door, but well well-protected dedicated path into the system
- Prevent application workload from using it
- Also useful for alike functions, such as log information fetching
Architectural Patterns - Someone in Charge

- Anything can go wrong, even during error processing
- If something does not work, some entity must be able to restart processing action
- For any fault tolerance activity, there must be a clearly identifiable responsible
  - Example: Active / Passive standby
- Single component in charge means single failure point, also increases complexity
  - Examples: Initialization module, cluster management node
  - Multiple fault tolerance activities may be needed at the same time
- Component must monitor progress and might initiate alternative actions
- Dual masters problem (also with voting)
Architectural Patterns - Escalation

- Endless recovery attempts might be valid in some cases (transient faults)

- But error processing becomes stalled when:
  - Correcting audits remain unsuccessful
  - Rollback / roll-forward remain unsuccessful
  - Still human intervention should be minimized

- Escalation of the processing makes the error less local and more drastic
  - Demands understanding of faults and failure modes
  - Some options: Resume partial operation, perform partial service degradation
Architectural Patterns - Fault Observer

- Faults and errors are detected and processed - tell all the interested parties
  - Observer can publish to humans through the maintenance interface
- Can be performed by an external entity
- Good application of publish / subscribe design pattern
- Someone in charge needs the information to steer the recovery process
- Report reception usually leads to logging
  - Make data storage again fault tolerant
- Typically part of some IT management software (‘cockpit’)
Architectural Patterns -
Examples for Pattern Relation

Correcting Audits
  Makes use of
  Redundancy
    Guides
      Units of Mitigation

  If not working
    Escalation
      If not working
        Recovery Blocks
          Example
            Makes use of
              Software Update
                Benefits from
                  Maximize Human Participation
                    Supported By
                      Maintenance Interface

            Example
              Minimize Human Interaction
                Intervention
                  Accessible Through
                    Fault Observer
                      Accessible Through
                        Progress Reports
                          Example
                            Example
                              Make use of
                                Progress Reports
                                  If not working
                                    Triggers
                                      Someone In Charge
                                        Can serve as
                                          Reports to
                                            Maintenance Interface
                                              Accessible Through
                                                Maximize Human Participation
Detection Patterns
Detection Patterns - Fault Correlation

• Prerequisite: Early fault removal uncovered common error types

• Look at unique signature of an error to identify an according fault category
  • Enables the activation of a well-known matching error processing
  • Examples:
    • Many off-by-one errors found in testing, prepare system for this
    • On data errors, related data to be checked should be known beforehand

• Multiple errors can happen close in time - useful to triangulate the fault location

• fault - error - error chain
  • In best case, take care of the initial fault that started the error chain
Detection Patterns -
System Monitor / Heartbeat

• **System Monitor**
  
  • How can one part keep track that another part is functioning?
    
    • Monitor for system (or system parts) behavior
    
    • Might be part of *fault observer* or *someone in charge*, or separate element
    
    • Location of the monitor is highly application-dependent

• **Heartbeat**
  
  • How does *system monitor* knows that a task is still working?
    
    • Send health reports at regular intervals (cost / benefit tradeoff)
    
    • Ping-alike messages, heartbeat function, push / pull approach
Detection Patterns - Acknowledgment / Watchdog

- **Acknowledgment**
  - Typical part of protocol definitions
  - Alternative for *heartbeat*, does not demand additional messaging
- **Piggybacking** - Add acknowledgment information to response data frame
  - Prominent approach in bidirectional networking protocols
- **Watchdog**
  - Watch visible effects of the monitored task, without adding complexity to it
  - Ensure that a task is alive, without messaging / processing overhead
  - Strategies: Timers, peepholes, hardware test points
Detection Patterns - Realistic Threshold

- How much time should elapse before the system monitor takes action?
  - **Message latency** (e.g. heartbeat interval) vs. **detection latency** (e.g. number of missed heartbeat messages)

- Balance between short intervals (hypersensitive monitoring) and long intervals (possibility for silent failures)
  - Influenced by communication round trip time and severity of undetected errors

- Message latency is typically worst case communication time + processing time

- Maximum unavailability > message latency + detection latency + repair time

- System can automatically adjust thresholds based on experience

- Example: Voyager spacecraft sends one heartbeat to command computer every 2s, failure when one is skipped
  - Overload condition detected during tests with 1s heartbeat
Detection Patterns -
Realistic Threshold - Example

• Message roundtrip time: 50ms - 100ms

• Heartbeat message: Preparation on monitor task - 20ms, Processing and reply on monitored task - 15ms, processing of reply - 15ms

• Detection latency: One message

• Scenarios
  • Messaging latency = 50ms : All true failures reported, but many false errors
  • Messaging latency = 100ms: All true failures reported, but long reporting delay
Detection Patterns - Voting

• Redundancy in space provides multiple answers - devise a voting strategy
  
  • **Exact voting**: Decision leads to correct result or uncertainty state notification
  
  • **Inexact voting**: Comparison might lead to multiple correct results
    
    • **Non-adaptive voting**: Use allowable result discrepancy, put boundary on discrepancy minimum or maximum
    
    • **Adaptive voting**: Rank results based on past experience
      
      • Predict what the correct value should be and take the closest result
      
      • Example: Weighted sum of the different results
        \[ R = W_1 R_1 + W_2 R_2 + W_3 R_3 \text{ with } W_1 + W_2 + W_3 = 1 \]
    
  • Different optimizations for large answers (e.g. compare only checksum)
  
  • Communication latency shall not influence voter operation
Detection Patterns - Voting

- Selection in case of multiple events:
  - **Majority vote** (uneven node number)
  - **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
  - **Weighted average** technique

- Components that disagree (to some extend) with the vote are marked as erroneous
Detection Patterns -
Maintenance and Exercises

- **Routine Maintenance**
  - Through operator on the *maintenance interface*, or built in
  - Typical strategy in operating systems for idle processors
  - Relies on concept of checkable resources - connections, memory allocations, ...

- **Routine Exercises**
  - Make sure that *redundant* spare components truly work in the *failover* case
  - Identify latent faults by checks during light workload - typical in hardware
  - Reproducible error is still better than the failure case on high workload
Detection Patterns - Routine Audits / Checksums

• **Routine Audits (‘scrubbing’)**
  - Find data errors in a controlled way, usually by low priority maintenance task
  - Logging is important for causal analysis - high possibility of related data errors
  - Identifies latent faults

• **Checksums**
  - Detect incorrect data by storing aggregate information along with the value
  - Example: Space shuttle counts number of integers in a data structure
  - Many options - parity bits, hashing
  - Checksums are only for detection, recovery through *error correcting codes*
Detection Patterns - Leaky Bucket Counter

- Distinguish between transient and intermittent repeating faults

- Assign a *leaky bucket counter* to each unit of mitigation
  - Increment for each event / fault
  - Decrement periodically until initial value -> fault events are periodically *leaked*
  - Exceeding the pre-defined upper limit of the bucket identifies a permanent fault

- Examples
  - Faulty messages filling a buffer
  - Correctable memory errors
Error Recovery Patterns
Error Recovery Patterns -
Quarantine / Concentrated Recovery

• **Quarantine**
  
  • Activate the prevention of error spreading and work contribution
    
    • Relies on *units of mitigation* in the architecture
    
    • Activate barrier around the component
    
    • Example: State indicator from voting unit

• **Concentrated Recovery**
  
  • Minimize unavailability by focusing all resources on recovery activity
    
    • Inform *fault observer* about recovery activity, stay inside *unit of mitigation*
    
    • Establish *quarantine* around recovery activity
    
    • Well established in systems with high survivability demands (e.g. telco industry)
Error Recovery Patterns -
Rollback / Roll-Forward

• How to resume processing after error recovery / error handler execution

• Rollback
  • Timing of the checkpoint / last requests decides about the rollback point
  • Consider side effects of repeated work
  • Errors might re-occur, so limit retries

• Roll-Forward
  • Resynchronization of systems tasks might be faster
  • Especially useful for event-driven stateless services
  • Demands proper damage mitigation and containment
Error Recovery Patterns - Restart / Limit Retries

- **Restart**
  
  - Way to resume execution when *recovery / escalation* is not possible
  
  - *cold / warm restart* - skip some of the initial checks, hardware vs. software restart
  
  - Supported by *checkpoints*

- **Limit Retries**
  
  - Scenario: Faults are deterministic (latent fault -> same stimuli -> activation)
    
    - *Rollback* might not solve the problem when the error activation reason remains
      
      - Example: 'Killer messages' marked as being unprocessed, faulty checkpoints
    
    - Problem: Propagation of error within itself, must be stopped by limiting retries
  
  - Solution: *Safeguarding* and *roll-forward*
Error Recovery Patterns - Failover

- Restoring of error-free operation in active element did not succeed
- Switch to redundant resource, based on replication
- Important factors are failover time and common data access
- Establish *someone in charge* for steering
- Needs proper *quarantine* for the faulty system part
Redundancy Configurations for Failover

- *N-to-1* and *N+1* are special cases of *Active / Passive* with multiple services

- *Active / Active* has no downtime, but leads to degraded system performance in failover case and might demand specialized data redundancy

- *N-to-1* demands a fail-back step, which is not needed with *N+1*

- *Hot standby*: No ramp-up needed on failover, no service failure for the user
  - Natural property of *Active / Active* setups
  - Possible even with *Active / Passive* setting through continuous replication, stateless services or static data

- *Warm standby / log shipping*: Synchronize data block-wise on spare

- *Failover* is typically used as synonym for *Active / Passive*

- Orthogonal: *Shared Nothing* vs. *Shared Disk* data management
Failover - Dual Master Problem

- Current active element might not relinquish control - **dual master** problem
- Typical problem in high-availability clusters
  - **Split brain** - Cluster interconnect is broken, several sub-cluster partitions start up
  - Establish **resource fencing** to let only one sub-group of the cluster work
  - **Amnesia** - Cluster restart with outdated configuration information
- **Quorum** - "The number (as a majority) of officers or members of a body that when duly assembled is legally competent to transact business" [Merriam-Webster]
  - 'Transact business' in the sense of 'provide service' - only one side should operate
  - Quorum allows fencing the other sub-cluster without communication
  - Loss of quorum should lead to node suicide, if possible
Failover - Quorum Approaches

- **Central arbitration** - Manual quorum, centralized server / admin sets master
- **Simple majority** - More than the half of the nodes must form a group
- **Weighted majority** - Votes for each node, group with higher vote count wins
  - Group decision is based on static data (nr. of votes, majority needed)
- **Tie-breaker** - Lightweight resolving strategy before decision inside the sub-group
  - Example: Ping response from common upstream router
- Whenever node connectivity changes, quorum decision should happen again
- Split brain has different faces
- Example in DRBD file system: Multiple replication masters by human error or temporary connectivity lost lead to difficult data merging demand
Failover - Weighted Majority with Quorum Device

• With even node number, provide additional external vote through **quorum device**
  • Number of votes by the quorum devices should be less than node votes
    • Allows cluster to operate with failed quorum device
  • Connection scheme of the quorum device decides upon valid cases of partitioning
  • Quorum device is typically a shared disk
• Only used when communication with other nodes fails
  • Implemented by SCSI RESERVE, Fibre Channel, or iSCSI
Example - Windows File Server

- Quorum case: Heartbeat line broken, Node 1 itself still alive
  - Demands utilization of cluster storage as quorum device
SCSI Quorum Device

• Only one SCSI device can use the bus at a time - *arbitration process*
  • LUN acts as priority, so host bust adapters typically have the highest one

• SCSI commands RESERVE and RELEASE allow to lock one SCSI device for exclusive usage by another device
  • Automated release on device / bus reset
  • Periodical renewing of reservation by driver, or persistent reservation feature

• Example MS Windows Cluster Server
  • Master node acts as *defender*, renews reservation every 3 seconds
  • One node communication loss, *challenger* nodes resets the bus, waits for 7 seconds, and tries to get the reservation again
Example - Quorum in Clusters

In this configuration, each pair must be available for either pair to survive.

In this configuration, the combination of any one or more nodes and the quorum device can form a cluster.
Example - Windows Server 2008 Failover Cluster

- Voting elements: Nodes, disk witness, file share witness (= tie-breaker)

- Quorum modes
  - *Node majority*, works with odd node number
  - *Node and disk majority*, for even node number with shared storage
  - *Node and file share majority*, for even node number in multi-site cluster
  - *No majority: disk only*, disk-based quorum as in Windows Server 2003

- File share / disk contains information about most recent cluster configuration (amnesia prevention)

- Disk mode: Hardware must offer persistent arbitration (e.g. SCSI reserve and release)

- File share mode: Active node keeps open file lock on the share (SMB feature)
Example - Windows Server 2008 Failover Cluster

• Permanent point-to-point heartbeat surveillance on each node

• Process of achieving quorum
  • As the node comes up, determine if other cluster members can be contacted
  • Members compare their membership view on the cluster and agree on one (group communication)

• Member collection determines if it has quorum
  • Without enough votes, it is waited for more members to appear
  • With quorum attended, resources and applications are brought into service
Example - Windows Multi-Site Clustered File Server

- **Normal Conditions**
  - Main site: nodes at this site usually own the clustered file, print, or application server.
  - Secondary site: nodes at this site rarely own the clustered file, print, or application server.

- **Main Site Gone**
  - Cluster storage for main site: read-write.
  - Secondary site: read-only.
  - Main site is down.

- **Communication Lost**
  - Replication of data to secondary site.

- **Split Brain**
  - Secondary site, with loss of connectivity between sites, has only two votes (out of five) and therefore stops running as a cluster.

(C) Microsoft
Example - Exchange 2007 Clustering

- Hub Transport Server allows messaging about the witness file share
- Witness share is checked ...
  - ... when a cluster node comes up and only one cluster node is available
  - ... when a previously reachable node is gone
  - ... when a node leaves the cluster (release lock)
  - ... periodically for validation purposes
Example - VMWare HA Split Brain Situation

- Even number of hosts run virtual machine images, stored on iSCSI / NFS
- Virtual machine image is protected by file lock with timeout
- Single host running a VM looses overall network connectivity
  - Other hosts restart the VM (due to lost external reachability)
  - Prevent the case that the VM on primary host will continue to run in this case
- Primary host gets connectivity back
  - Takes back the virtual machine image file since it has the according processes
Error Recovery Patterns - Checkpoint

- Avoid loss of results during recovery by saving global state information
  - Focus on long duration data that is hard to achieve
  - Checkpoint data consistency and checkpointing interval are relevant
- The "snapshot" problem - how to achieve global (distributed) consistency?
  - Global state == local states + messages
  - Snapshot algorithms: Determine past, consistent, global state
  - Chandy & Lamport (1985) landmark paper
    - Relies on flushing principle of FIFO communication channels
    - Control messages 'push out' pending messages
Error Recovery Patterns - Remote Storage

- Storage location for checkpoints is relevant in failover / rollback case
  - Should not be the single point of failure
  - Pattern is good decision point for level of redundancy needed
- Real-world application: iSCSI Multi-Homing
Error Recovery Patterns - Individuals Decide Timing / Data Reset

• **Individuals Decide Timing**
  
  • **Independent checkpoints**: Opposite approach to global checkpoints
    
    • Each process takes a **dynamic local snapshot** when it needs to
    
    • Consistency establishment overhead at recovery time vs. global checkpoint overhead during operation
  
• **Data Reset**
  
  • Recover from an uncorrectable data error by taking / computing initial values and approximate value

  • Relationship to *return to reference point* pattern - data reset is often a correlated activity
Error Mitigation (= mostly Overload Handling) Patterns
Error Mitigation Patterns - Marked Data

• Data error detected, but no recovery option available, error mitigation is acceptable

• Data should be *quarantined* - do not use it, do not derive actions from it

• Example: IEEE 'Not a Number' (NaN)
  • Result of division by zero, square root of -1, ...

• IEEE 754-1985: Standard representation for binary floating point numbers

• Rules for computation when operand is NaN - typically result is again NaN
  • Options: Assume default value, skip operation, mark result as erroneous
Error Mitigation Patterns - Overload Toolboxes

• Handle overload situation with too many requests for the system

• Each resource class needs dedicated overload treatment
  
  • Memory: Exhaustion hinders new request from entering the system
  
  • CPU: Overload slows overall processing down
    
    • Patterns: *Fresh work before stale, share the load, shed load*
  
  • Tangible resources: Processing demands exclusive system resources
    
    • Network ports, shared storage, devices, ...
      
      • Patterns: *Queue for resources, equitable resource allocation*

• Consider user demands
  
  • Patterns: *Fresh work before stale, finish work in progress*
Error Mitigation Patterns - Shed Load

• Throw away a minority of requests to serve the majority
  • As early as possible, to minimize resource consumption
  • Rejection method to be considered, e.g. do not send acknowledgements

• Example: ICMP
  • Type 3: Destination Unreachable - not time-out on client host
  • Type 4: Source Quench - typically only between routers, also used by mail servers
  • Type 11: Time Exceeded - through congestion (or circular packets)

• Example: HTTP 5XX error codes
  • 503: Service Unavailable

• Specialized case: Shed work at periphery
Error Mitigation Patterns - Finish Work in Progress / Fresh Work Before Stale

• **Finish Work in Progress**
  
  • What to process, what to reject?
    
    • Best case is labeling of requests: „new“ vs. „continuation“
    
    • Distinguish „continuation“ processes on their resource usage
    
    • Try aggressively to get rid of resource hogs
  
  • Can lead to oscillation when system is „starving“ for new requests after cleanup
    
    • Solution: Let small portion of new requests through

• **Fresh Work Before Stale**
  
  • If requester gives up, his retry eats up even more resources
    
    • Perform LIFO queue handling or non-queueing for premium requesters
Error Mitigation Patterns - Slow It Down

- Handle overload cases and avoid saturation by multi-step *escalation*
  - Restrict request processing with increasing severity per level
  - Goal: Slow things down until the system can catch up with the load
  - Feedback system, demands dedicated resources for the controller part
- Add **hysteresis effect** to prevent oscillation for level changes
  - Different trigger values to enter / leave an escalation level

![Graph showing Overload State and Thresholds with hysteresis effect](image-url)
Error Mitigation Patterns -
Deferrable Work / Equitable Resource Allocation

• **Deferrable Work**
  
  • High load: Shed incoming work vs. shed routine maintenance workload
  
  • Make routine work (only relevant in error case) deferrable

• **Equitable Resource Allocation**
  
  • Scenario: Handling of many requests for a set of resources, some of them are rare
  
  • Request-level handling would render some resources unnecessarily idle
  
  • Solution: Pool similar requests, allocate resources to pools
  
  • Additional bookkeeping needed for managing the requests and their related resource demands
  
  • Might lead to priority-inversion scenario
Error Mitigation Patterns - Expansive / Protective Automatic Controls

• **Expansive Automatic Controls**
  - Design some system parts for only being used in case of overload
    - Example: No 100% CPU utilization in normal operation of HA clusters
    - Example: Dynamic Offloaded Work - Cloud Computing
  - Increases request processing overhead, so take only as temporary solution

• **Protective Automatic Controls**
  - Overload options: Shed internal work, shed incoming load, do nothing
  - Put restrictions on how much work the system accepts while still functioning
  - System throughput can drop due to contention, but should not drop to zero
Fault Treatment Patterns
Fault Treatment Patterns - Let Sleeping Dogs Lie / Reintegration

• **Let Sleeping Dogs Lie**
  
  • Treating faults by system change can introduce new faults
    
    • Known latent fault: Risk of reoccurrence, damage assessment possible
    
    • Potential new fault: Additional risk of miss-applied correction, no damage assessment possible for accidentally added faults

• **Reintegration**
  
  • Different steps needed to reintegrate repaired component
    
    • Take off *riding over transient* and *isolation* lists
    
    • Watch new component for a while: *hardening / soaking / trailing*
    
    • Follow deterministic procedure, use as *standby* if possible
Fault Treatment Patterns - Reproducible Error / Small Patches / Revise Procedure

• **Reproducible Error**
  - Apply stimuli again under *quarantine* in order to prove fix
    - Can be automated (regression test)
    - Compare system output with *golden unit* output

• **Small Patches**
  - Design system update as small as possible

• **Revise Procedure**
  - When predetermined procedures contributed to failure duration, fix them