Dependable Systems

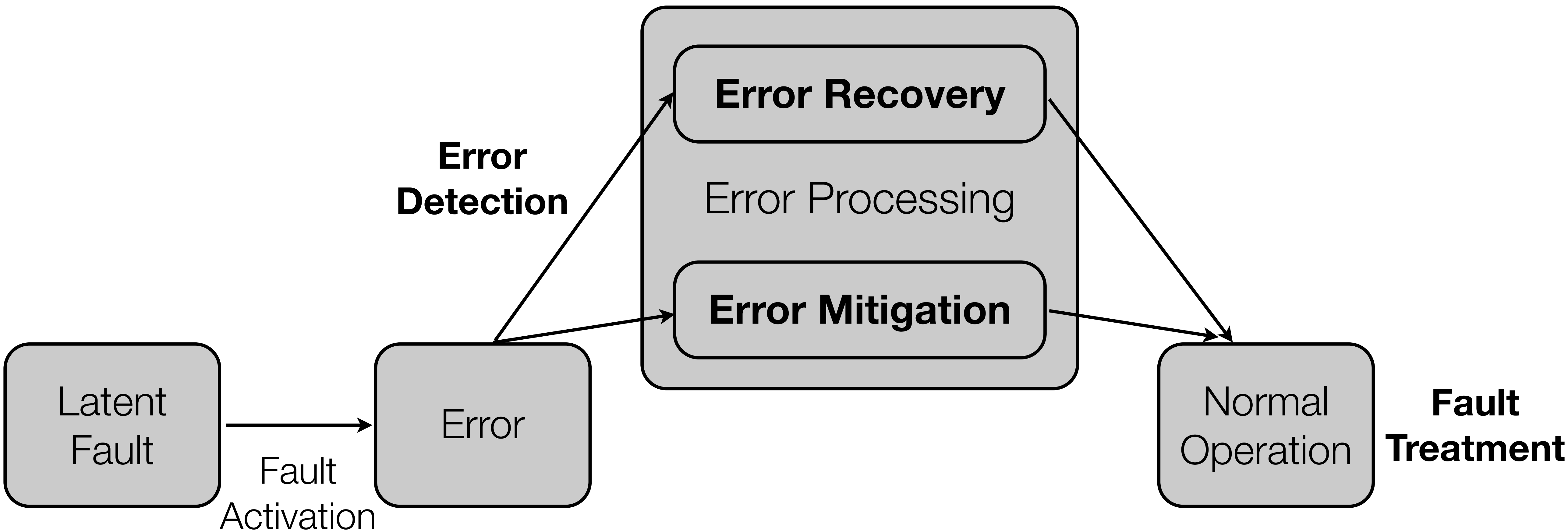
Fault Tolerance Patterns (I)

Dr. Peter Tröger

Source:

Hanmer, Robert S.: Patterns for Fault Tolerant Software. Wiley, 2007.

Phases of Fault Tolerance (Hanmer)



Design Pattern

- 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
- Shared context for fault tolerance patterns
  - Patterns might be suited for stateless / stateful / both kinds of system
  - Fault tolerant systems have observers and monitors (humans / computers)
  - On-top-of application functionality, orthogonal to primary function

Fault Tolerance Patterns

- Architectural patterns
  - Considerations that cut across all parts of the system
  - Need to be applied already in early design
- 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 reoccuring by repairing the fault
  - System verification, diagnosis of fault location and nature, and correction of the system and / or the procedures

Architectural Patterns

Units of Mitigation

- Only parts of the system should potentially get into error state
- Design *units of mitigation* that contain errors and error recovery
    - Component size vs. bookkeeping overhead vs. fault tolerance options
    - Should contain independent atomic actions without communication focus
    - Architectural style (e.g. n-tier architecture), sizes, function and memory / processor boundaries can provide hints
    - Entities of a group of similar functionalities are good candidates (thread pool)
    - Unit design depends on 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
- Example: n-tier architecture

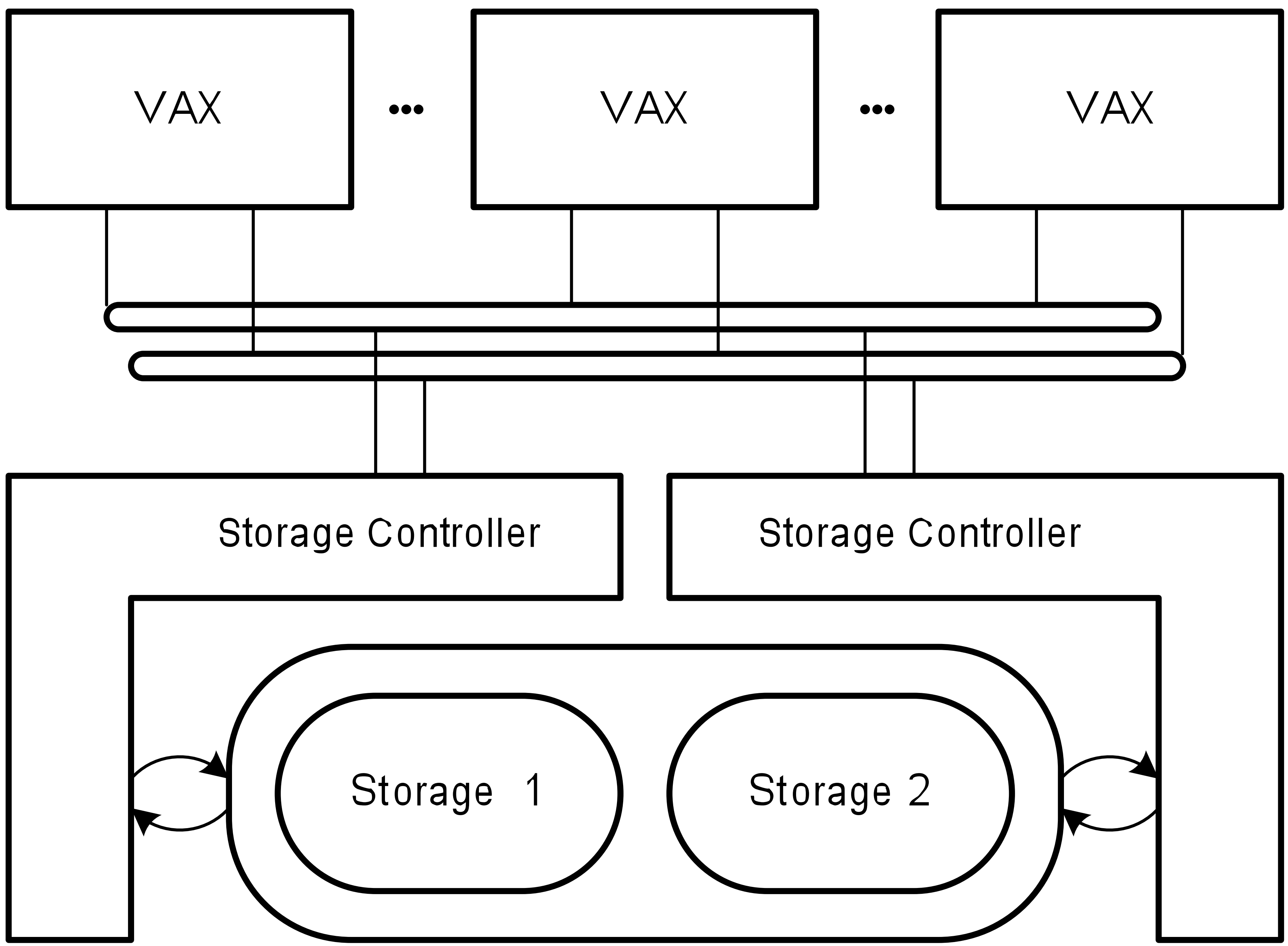
Correcting Audits

- Data element corruption can occur - low level hardware, random and transient physical faults, and software (data type, currencies, pointers, ...)
- Checking resp. auditing data for errors 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 check related item consistency
- Actions: Correction, logging, resume execution
- Errors from faulty data easily propagate, common audit infrastructure helps

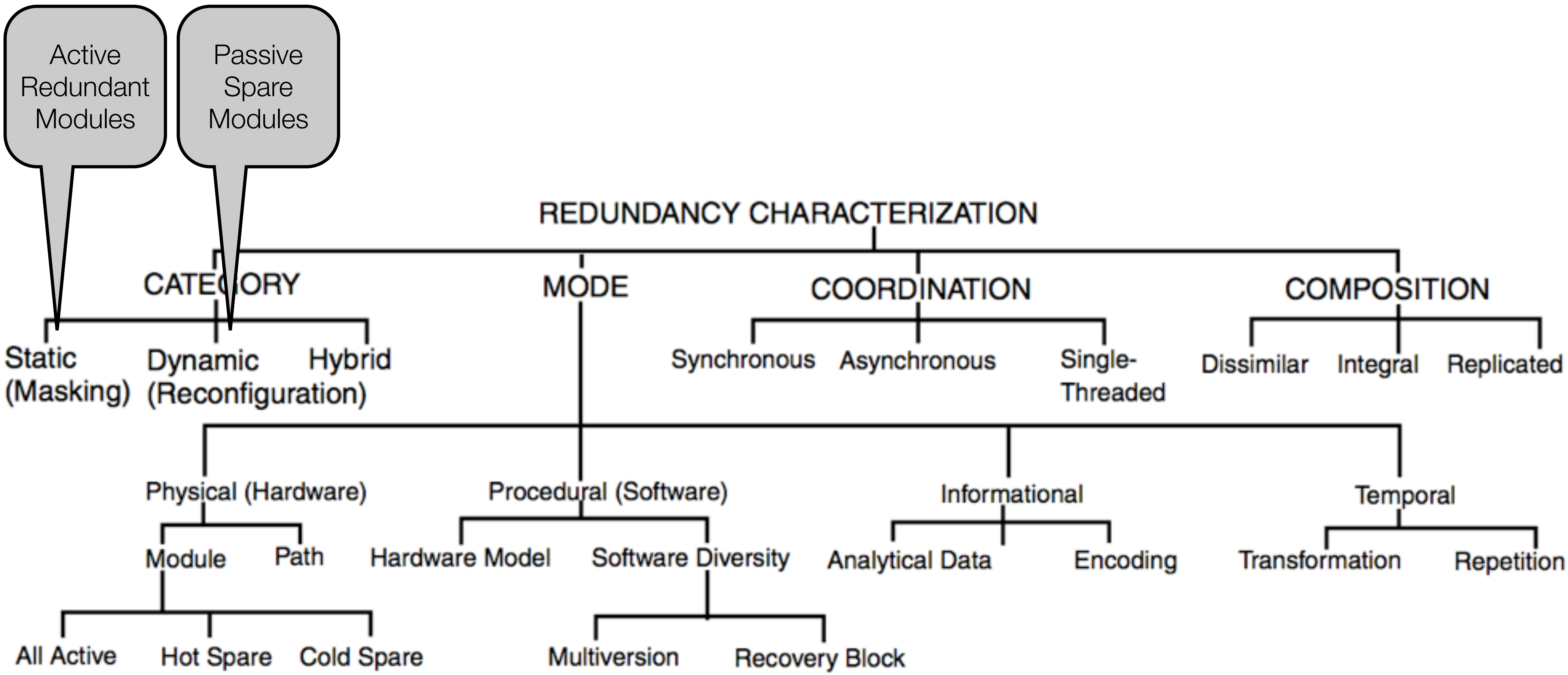
Redundancy

- Wish for minimal MTTR, all software and hardware components are important
- Error recovery makes the effect of the error undone - phase must be minimized
  - Idea: Resume execution before bad effects are undone, by using identical copy
  - Another way to accomplish the same work on different hardware / software
  - Quick activation of redundant feature needed
- Redundancy types: spatial, temporal, informational (presentation, version)
  - Does not mean identical functionality, just perform the same work
  - Danger with deterministic software execution in case of identical copies
- Redundancy for performance improvement, availability then by excess capacity

Example: VAX Spatial Hardware Redundancy



Redundancy Classification (Hitt / Mulcare)



Example: Data Management Tradeoffs

Disk Replication	Disk Access Switchover
Easier to add to a given system	Demands altering of given system
Independent configuration	Synchronized configurations
Nodes can be apart	Co-location
Fault-intolerant storage ok	Demands fault-tolerant storage
Demands active copying	Single data storage
CPU & I/O overhead	No overhead
Tight vs. loose synchronization	No synchronization needed
Failback brings re-synchronization issues	Switch back is painless

(adopted from Pfister)

Example: PostgreSQL 9 Redundancy Options

- **Shared-Disk**
  - **Failover** - Avoids synchronization overhead, demands network storage resp. file system, mutual access exclusion from active / passive node must be ensured
- **Shared-Nothing**
  - **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

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

Recovery Blocks

- *Redundant* system implementations are typically used simultaneously, best answer is picked i.e. by voting
- Alternative way: Sequential execution of *recovery blocks*
    - Limited overhead (execution only in error case), redundancy in time
    - Diversity of *redundancy* implementation is relevant
    - Acceptance tests per block, might lead to final *error handler*
    - Checkpoint before first block needed, to ensure same preconditions
    - Make successive block more simple, maybe loose parts of the result
- Example: Monitor software update process for correct finalization
- Problems: Shared global data, lack of alternative algorithms, added complexity

Humans

- *Minimize Human Interaction*
  - Errors in HA system: Hardware, Software, Procedural / Operational
  - Humans are bad in: Long series of steps, routine tasks, operation, response time
  - Reduce failure risk due to procedural errors - process errors automatically
    - Operational staff should be able to monitor, but not be required for the solution
    - Use patterns for effective communication with people
- *Maximize Human Participation*
  - System should support design / operational / external experts in contributing to an error solution - Humans can draw meaning from sequence of unrelated events
  - Examples: Reporting prioritization, context information (timestamp etc.)
  - Safe mode: Wait for human participation

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 hidden trap door, well well-protected dedicated path into the system
- Prevent application workload from using it
- Also useful for alike functions, such as log information fetching

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
- Component must monitor progress and might initiate alternative actions
- Dual masters problem (also with *voting*) - unique index approaches (e.g. time)

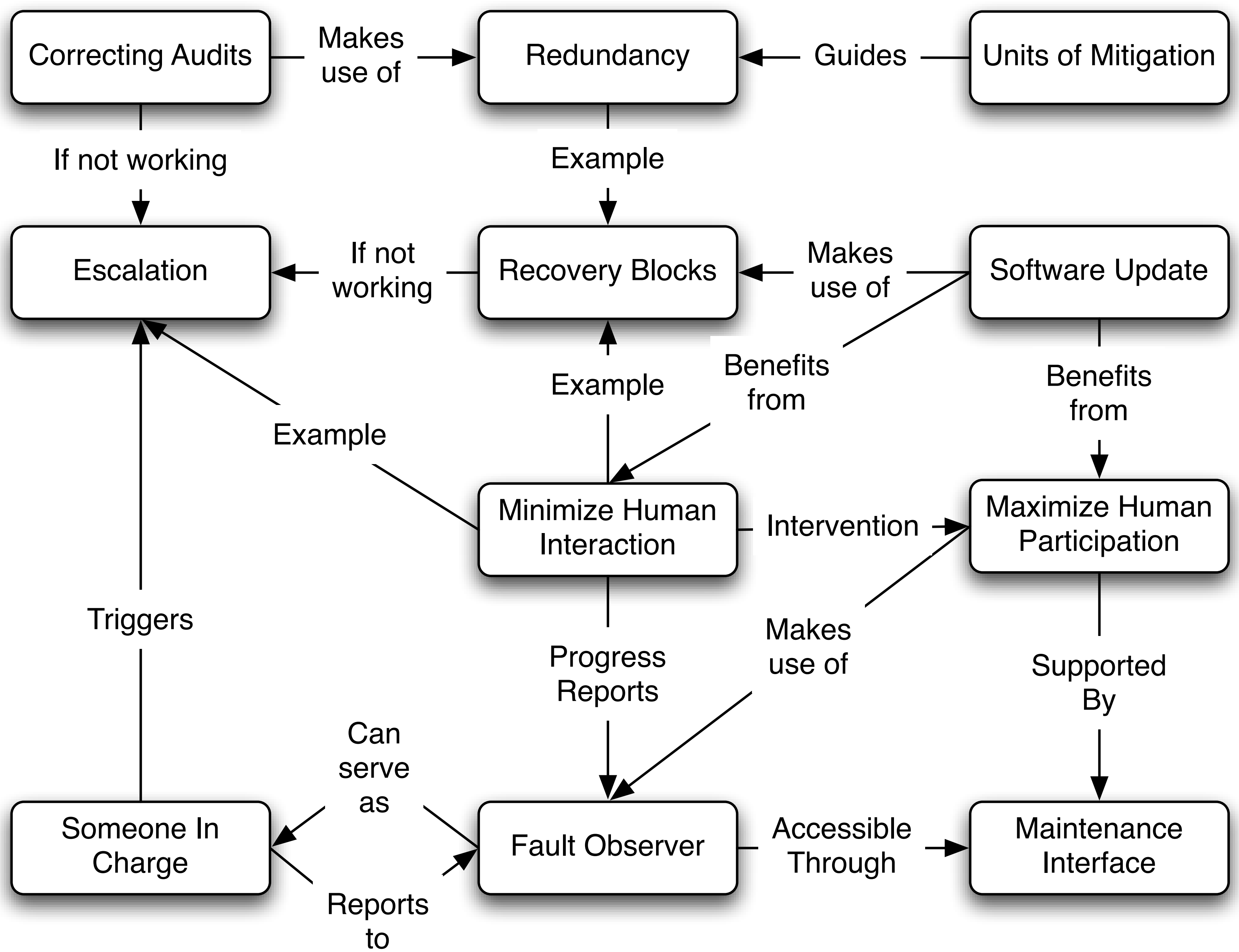
Escalation

- Error processing is stalled - *correcting audits* unsuccessful, *rollback / roll-forward* failed, goal to *minimize human intervention*, some components are *in charge*
- Endless recovery attempts might be valid (transient errors)
- Step by step, make the error processing less local and more drastic
  - Identifying the escalation steps is true fault tolerant system design, demands understanding of faults and failure modes
  - Options: Resume partial operation, perform partial service degradation

Fault Observer

- Faults and errors are detected and processed - tell all interested parties
  - Notify from processing component (error), not detection component (fault)
  - Observer can publish to personal over *maintenance interface*
- Can be performed by an external entity
- Good application of publish / subscribe pattern
- *Someone in charge* needs the information to steer the recovery process
- Report reception usually leads to logging
  - Make data storage again fault tolerant

Relations (Example)



Detection Patterns

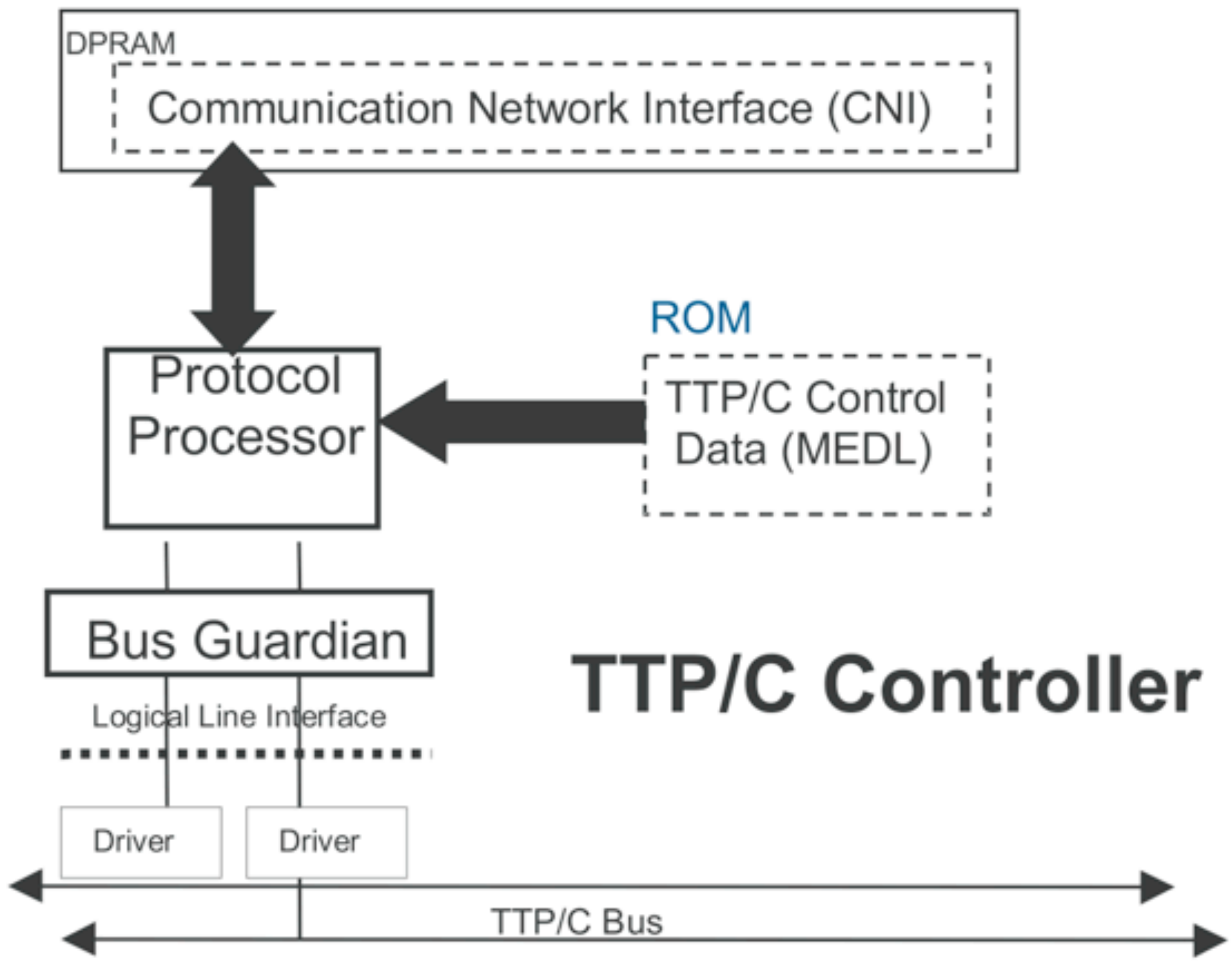
Fault Correlation

- Fault removal during design and test uncovered common error types
- Look at unique signature of error to identify error type and according fault category
  - Enables the application of a known proper error processing
  - Examples: Many off-by-one errors found in testing, prepare system for this class
  - Data errors - correlation also demands identification of related data to be checked
- Multiple errors can happen close in time - use to triangulate the fault location
- *fault - error - error* chain
  - In best case, process the initial fault that triggered the chain

Error Containment Barrier

- Errors spread through several mechanisms - messages, memory, follow-up actions
- Error mitigation or ignorance does not always work
- Software: Barrier for errors is the *unit of mitigation* boundary
  - Barrier must resist error state itself, and 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 - *bus guardian* as barrier implementation
  - Idea - suspicious nodes should never be in control of the communication bus

Guardian Example: Temporal Firewall in the Time-Triggered Architecture (TTA)



(C) Kopetz et al., TU Wien

Complete Parameter Checking

- Minimize time from fault activation to error detection
- Perform frequent checks on data and operations to detect errors quickly
- Strongest realization with *lock step* approach in *active / active redundancy*
  - Relaxation by checking only computational end results
  - Value ranges for function / method arguments, less costly approximations
  - *Design by contract* approach
- Frequency of checks and resulting error detection time vs.
  system performance / maintenance effort / development time
- Variation: Mask detected error into an acceptable result

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

Acknowledgment / Watchdog

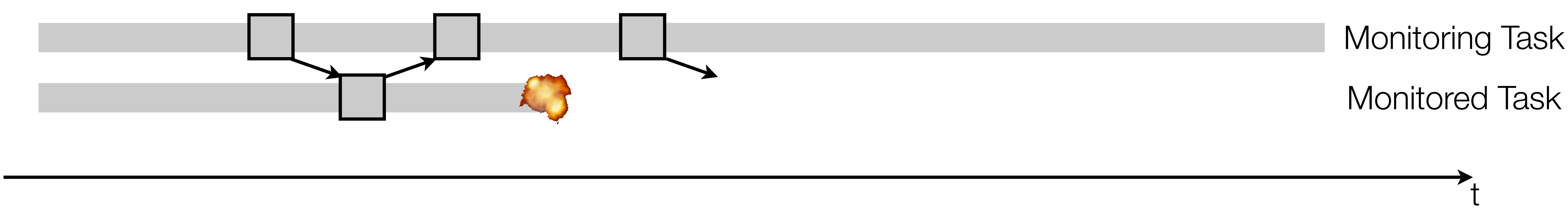
- **Acknowledgment**
  - Alternative for *heartbeat*, does not demand additional messaging (as error source)
  - **Piggybacking** - Add acknowledgment information to data frame
    - Prominent approach in bidirectional networking protocols
- **Watchdog**
  - Ensure that a task is alive, without messaging / processing overhead
  - Watch visible effects of the monitored task, without adding complexity to it
  - Strategies: Timers, peepholes, hardware test points

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 + restart 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

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 = 1000ms: All true failures reported, but long reporting delay



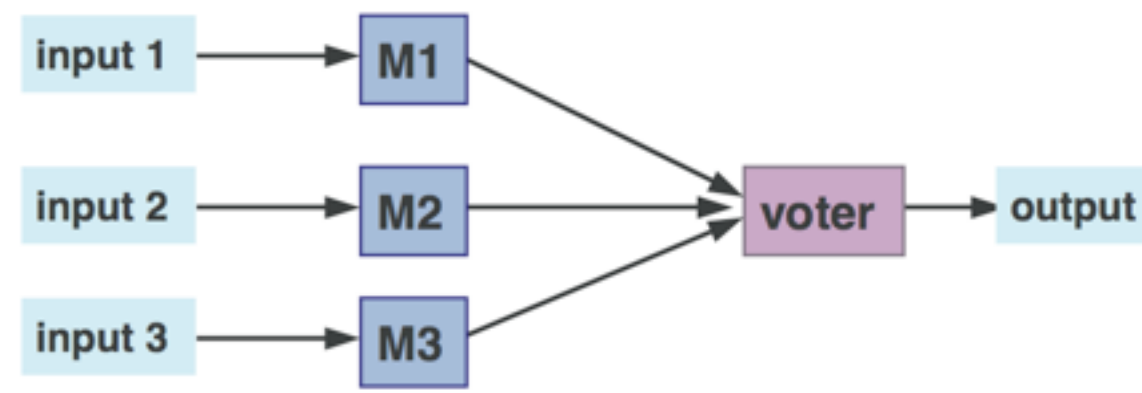
Voting

- Redundancy in space provides multiple answers - devise a voting strategy
  - Comparison at high level might lead to multiple correct results - **inexact voting**
    - **Adaptive voting** - Rank results based on past experience
      - Predict what the correct value should be and take the closest result

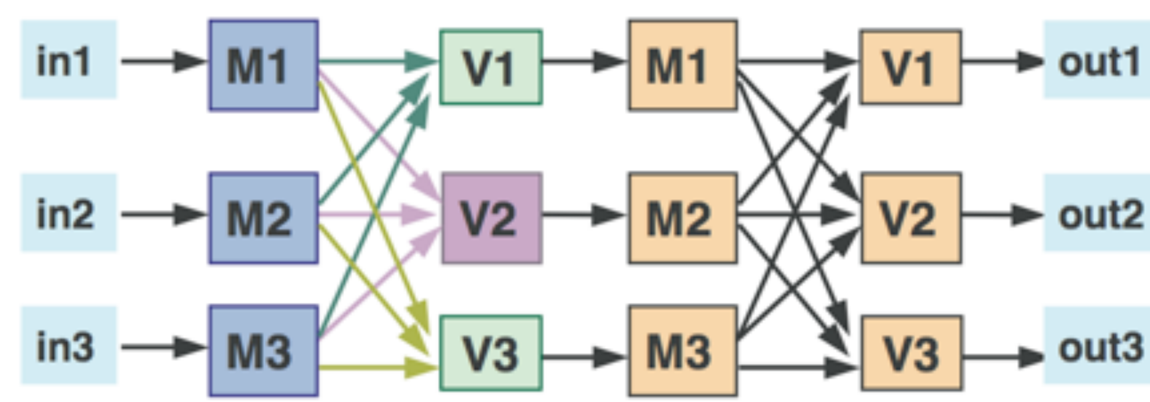
        Example: Weighted sum of the different results
        $R=W_1*R_1 + W_2*R_2 + W_3*R_3$ with $W_1+W_2+W_3=1$
    - **Non-adaptive voting** - Use allowable result discrepancy, put boundary on discrepancy minimum or maximum
  - With **exact voting**, decision leads to correct result or uncertainty state notification
- With large answers, only checksums could be compared
- Communication latency shall not influence voter operation

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



Triple Modular Redundancy (TMR)



N-Modular Redundancy (NMR)

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

Routine Audits / Checksums

- **Routine Audits**
  - 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*

Riding Over Transients / Leaky Bucket Counter

- **Riding Over Transients**
  - Avoid wasting resources on processing of transient error states
    - Examples: Retry on parity error with optical disk read, ignore error code return values from operating system API functions
    - With easy error propagation, number of tolerated transient faults should be low
    - Correlate faults and tolerate only well-known transient fault, keep statistics
- **Leaky Bucket Counter**
  - Distinguish between transient and intermittent repeating faults
    - Assign a *leaky bucket counter* (== error counter) to each *unit of mitigation*
    - Decrement the counter periodically, but never below initial value
    - Exceeding the predefined upper limit identifies a permanent fault