Software Based Fault Tolerance

István Majzik
Technical University of Budapest

August 29, 1997

TEMPUS S_JEP-8333-94

Overview

- Introduction
- System structure and dependability
- Fault tolerance: how and where
- Error detection
- Damage confinement and assessment
- Error recovery
- Fault treatment and continued service
- Software fault tolerance
- Error recovery in concurrent systems
- Error recovery based on message logging
- Consistency in distributed systems
- Managing replicated data

Introduction

The concept of dependability

- **Dependable system**: *reliance can justifiably be placed the service it delivers*
  - Attributes of dependability
    * Reliability: continuity of service
    * Availability: readiness of usage
    * Safety: non-occurrence of catastrophic consequences
    * Security: prevention of unauthorized access
  - Impairments of dependability
    * Failure: service does not comply with specification
    * Error: part of the system state which is liable for failure
    * Fault: adjudged or hypothesized cause of an error
  - Means of dependability
    * Fault prevention: how to prevent fault occurrence
    * Fault tolerance: how to ensure a service in presence of faults
    * Fault removal: how to reduce the presence of faults
    * Fault forecasting: how to estimate the number/consequences of faults

- On impairments of dependability
  - Fault: active/dormant
    - Nature: accidental/intentional
    - Origin: physical or human-made; internal or external; design or operational
    - Persistence: permanent or temporary
  - Error: latent or detected
  - Failure: value or timing; benign or catastrophic; consistent or Byzantine
  - Manifestation: ... fault $\mapsto$ error $\mapsto$ failure ...
- On attributes of dependable systems
  - Integrity: avoidance of improper alterations
  - Confidentiality: non-occurrence of unauthorized disclosure of information
  - Maintainability: ability to undergo repairs and evolution
  - Performability: performance related measure
  - Testability: ability to test for certain attributes
- On means for dependability in system design phases
  - Design and implementation: Fault prevention
  - Testing and debugging: Fault removal
  - In service: Fault tolerance

Software systems

- Fault prevention by formal methods
  - Design methodology
  - Consistency checks (syntax + type checking)
  - Testing (executable specification)
  - Formal verification (model-, equivalence- and preorder checking)
- Requirements for dependability
  - Component reliability
  - Managing system complexity
    (e.g. structured, modular, object oriented programming)
- Fault types from the point of view of fault tolerance
  - Anticipated faults: Taken into account during the design phase
    Consequences: Fault tolerance measure
  - Unanticipated faults: Impossible to enumerate the consequences
    E.g.: design faults

System structure and dependability

Basics

- System: Behavior + interface to its environment
- Interface: Place of interaction between system and its environment
- Environment: Another system
  provides input (request) and receives output (service)
- Internal structure: Set of components (subsystems) interacting under the control of the design
- Software and hardware interaction:
  Program: design of an abstract system HW components: objects manipulated by the program

Interpreter model of systems

- Interpreter: hardware + related system software (OS)
- Interpreted: Application program
- Interpretive interface: Primitives given by the interpreted part

Measures and mechanisms of fault tolerance

- Mechanism: Construction within an interpreter
  implements a facility available at the interpretive interface
  - general purpose
  - lower level
  - simple and reliable
- Measure: Instructions in the interpreted system
  intended to perform some specific task
  - application dependent
  - higher level
  - specialized

Component model of systems

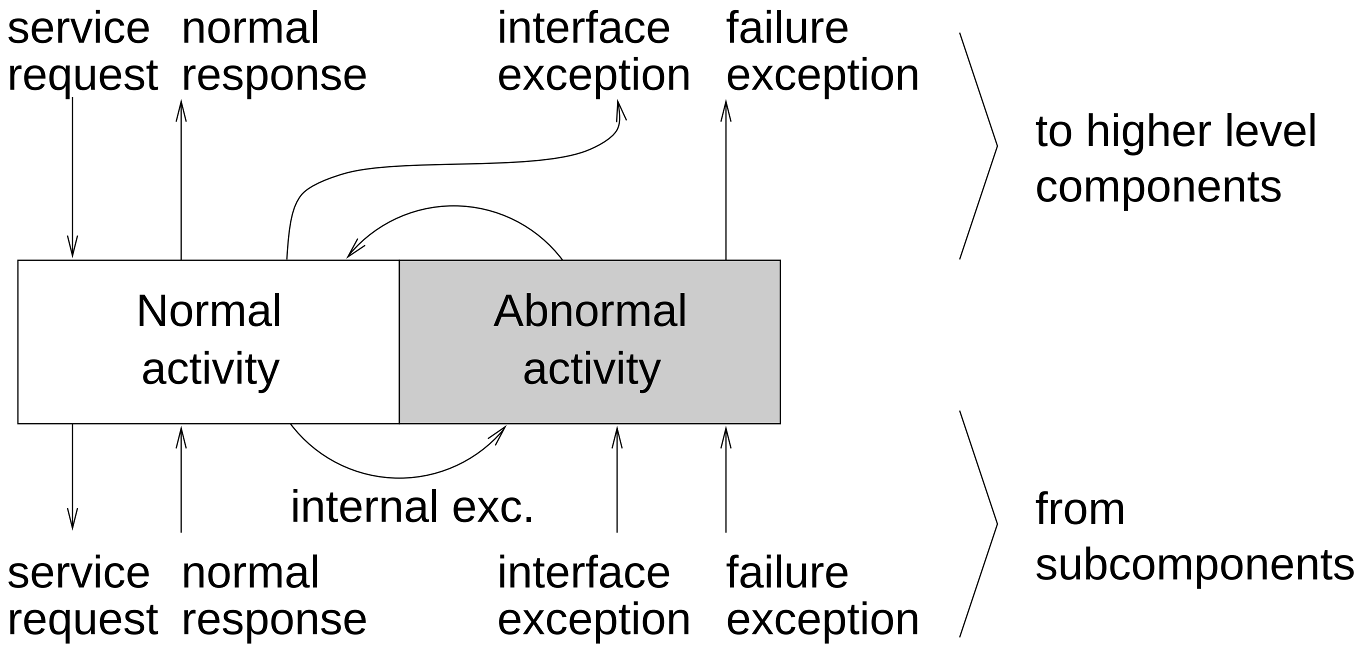
- Containing system: requests to and responses from components
- Component: requests to and responses from subcomponents
- Subcomponents

Detailed implementation framework (component model)

- Fault tolerant component: Abnormal activities separated and handled
- Classification of exceptions
  - Interface exception (IE): error detected in the request
    e.g. illegal instruction, protection violation, under- overflow
  - Internal exception: error detected in the component
    e.g. parity check during data manipulation
  - Failure exception (FE): error that has to be propagated
    error of the component that can not be tolerated

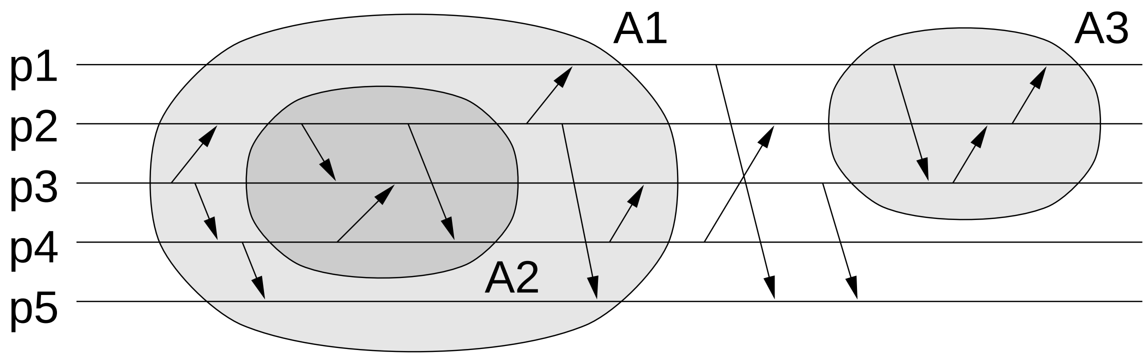
- Idealized FT component
  - All exceptions (IE and FE) from subcomponents are handled
    return to normal operation if possible
  - Interface exception: part of the normal operation
  - Internal exception: trying to handle and return to normal operation
  - Consequences of an exception:
    * resumption mechanism: call handler then resume operation
      dangerous, the component left in inconsistent state
    * termination mechanism: exit activity, recover consistent state
      better mechanism
- Practical issues in exception handling
  - Exceptions with name and parameters
  - A handler for each possible exception
  - Exceptions from subcomponents: propagate in proper form to higher levels
    handling (at least renaming) is needed
  - Propagation:
    * calls hierarchy: to current requesters only
    * uses hierarchy: to all possible requester (forewarn)

Idealized FT component



Structuring information flow: Atomic actions

- Dynamic system structure: interactions
- Atomic action: activity of a group of components
  no interaction between the group and the rest of the system
- Useful in: transactions, recovery



Fault tolerance: how and where

How: Preventing errors from leading to a failure

- Error detection (fault cannot be detected by the system)
- Damage confinement and assessment
  - Assess the extent to which the state was damaged
  - Limit the propagation of errors
  - Problems: delay between fault manifestation and error detection
- Error recovery
  - transform the erroneous state into a well-defined error-free one
- Fault treatment and continued service
  - identify the fault
  - permanent fault: repair or reconfigure
  - transient fault: continue system service

Assumptions in software

- design faults: cannot be predicted, permanent
- correlated failures in similar components (instances)

Redundancy: utilization of extra components in the system

- Hardware redundancy: static (masking) or dynamic (error correction)
- Software redundancy
- Temporal redundancy

Fault tolerance: Where?

- Aspects: reliability requirements, cost, power, space, time, ...
- Alternatives
  - Low level redundancy: effective against particular faults
  - High level redundancy: more general, less effective
- Quantitative evaluation: reliability models
  - reliability is proportional with remaining number of faults
  - relationship between exposed and undiscovered faults
  - estimate the remaining number of faults

Error detection

Error detection measures
(can be incorporated into any system)

- Ideal checks
  - based solely on specification (complete spec. needed)
  - based on alternative implementation (high costs)
  - independent checker (access to information still required)
- Acceptability checks
  - assumptions about possible faults
  - no guarantee of the absence of errors
- Last moment checks: before the output leaves the component
  Earliest possible checks: minimize wasted activity and damage spread

- Types of checks

replication: alternative implementation or repeated component
(repeated use: only for transient faults)
+ powerful, general
– costly (in mission critical systems)

timing checks: check timing constraints
+ watchdog timer
– reveal the presence (not the absence) of faults

reversal checks: re-calculating and comparing the input
– if one-to-one mapping exists between input and output
+ simple if inverse exists

coding checks: redundancy in information representation
fixed relationship between original and redundant data
error: corruption in the relationship
+ error detection and correction codes
+ signature-based checks
+ efficient implementation
– only acceptability checks

reasonableness checks: based on intended usage or purpose

+ range check (predefined limits)
+ type checking
+ can be specialized
– time overhead

structural checks: based on complex/redundant data structures

+ number of elements
+ pointers (double linking)
+ status info
+ N-1 changes are detected if new valid structure requires N changes
– intuitive

diagnostic checks: checking the components (not the system)

+ test vectors, examining output
+ running periodically or in spare time
– expensive in time and resources

Error detection mechanisms

- Interface checks: available services (objects or operations)
  Results:
  - illegal instruction
  - protection violation
  - over/underflow
- Mechanisms:
  - tagged storage (objects with type, rights)
  - descriptors (bounds, limits)

Structuring error detection

- Ideal case:
  - Interface exception: design fault
  - Failure exception: component fault, handled
- Required:
  - Self-checking components
  - Full diagnostic checks in component
  - Minimized latency
- Worst case: Component faults propagated to higher levels
  Diagnostic checks are needed

Damage confinement and assessment

Basics

- Damage: unwanted consequences of a fault
- Identification: not guaranteed by error detection (delay before detection)
- Damage depends on: system activity + flow of information
- Damage confinement: limit the propagation of errors
- Damage assessment: establish the extent to which the state was damaged

Damage confinement measures: Enforce constraints

- physical separation (e.g. TMR)
- logical separation (e.g. protection)

Damage confinement mechanisms: Structuring information flow

- Atomic actions (AA): planned or enforced
  - message passing: only between processes in the AA
    retaining (buffering) external messages
  - shared objects: constraining the access sequence (interface)
- Constraining concurrency: inhibit interrupt or process switching
- Periods processing: Isolate the objects used in AA from others
  (e.g. security levels, specific periods to processing objects)
- Protection mechanisms:
  - List-based schemes: list of users for each sharable object
    efficient administration but high run-time overhead
  - Ticket based schemes: processes have rights to objects
    low run-time overhead but complex administration
    shared objects: protection domains (rights) overlap
    prevent overlapping: AA is needed

- Capability based mechanisms:
  capability=protected ticket to name an object and list access controls
  shared object: more processes possess capability
- Locking systems
  a lock is set for every shared object; it can be accessed by a key
  Atomic action:
  1. lock the objects to be accessed
  2. give key for the concerned processes

  Implementation:
  * Operating systems: semaphore, monitor (based on test-and-set primitive)
  * Database systems: transaction

- Problems:
  - restricted parallelization (low performance)
  - administration overhead
  - possibility of deadlock (circular dependency)

Damage assessment measures

- Initial estimate: provided by error detection
- Static technique: specified at design time
  - all objects in the initial estimate are damaged
  - a priori damage assessment for HW (limited interconnection)
- Dynamic technique: refinement of the initial estimate (heuristics)
  - exploring the system state + data structures
  - diagnostics checks performed
  - dynamically identified atomic actions

Damage assessment mechanisms: Recording information flow

- Sequential programs: recording the update of variables
  (recovery cache, audit trail)
- Distributed systems: OS-level dynamical monitoring
  Problem: interdependencies (A to B and B to C then A to C?)
  Information collection has to be limited (recovery points)

Error recovery

Basics

- Error detection + damage assessment: passive phases
- Error recovery + fault treatment: active phases (change the system state)
- Erroneous system state: could lead to a failure!
- Error recovery: Eliminate the errors (after the damage assessment)

Types of damage

- Anticipated damage: the designer can predict it (anticipated fault)
- Unanticipated damage: accurate prediction/assessment cannot be made

General techniques

- Forward recovery:
  Making selective corrections (limited changes) to system state
  Limited to identified areas of damage: anticipated damage
  Implemented as a measure
- Backward recovery:
  Restoring a prior state of the system (the entire state)
  Full system state replaced: unanticipated damage is handled
  Implemented as a mechanism (unanticipated fault could corrupt the base of the measure!)

Simple state restoration techniques

- cold start (fixed reset): restoring the initial (given) state
- warm start: reseting the data structures only
- frame boundary: periodic initialization (in RT systems)

Forward error recovery

- Dependent on damage assessment and fault prediction (anticipated)
- Inappropriate for unanticipated faults
- System (and application) specific
- Measure (not a mechanism), e.g.
  - error correcting codes
  - structural redundancy
  - fixed error-free state
- Cost-effective

Backward error recovery

- Complete state restoration
  - independent on damage assessment
  - recovery from unanticipated faults
  - general concept
  - easily provided as a mechanism
  - may be expensive and slow
- Provisions:
  - prior state must be successfully restored
    (all components have to be recoverable)
  - restored state is free from errors
    (preceded the manifestation of the fault)
- Facilities:
  - Recovery point: Point in time, for which the state restored
  - Recovery data: Preserved state information
  - Operations: Establish, discard, restore recovery data/point
  - Active recovery point: From establishment to discard

  - Recovery region: active recovery point exists
    (multiple recovery regions can overlap)
- Mechanisms:
  - State based backward recovery: complete system state is preserved
    Recovery point is called checkpoint
    * restoration: replace the system state
    * no reliance is placed on the erroneous state
    * overhead in recovery data
  - Operation based backward recovery: activities of the system are logged
    Recovery data is a historical record called audit trail
    * restoration: undo the effects
    * placing reliance on the erroneous state
    * overhead in data and processing time
  - Practical compromise:
    * infrequent checkpoints
    * audit trail between checkpoints

- Recovery in database systems
  - incremental dumping: copies written after transactions
  - careful replacement: copies preserved before transactions
  - differential files: originals + changes are recorded
- Unrecoverable components: what to do?
  - Recovery avoidance: delaying operations until no recovery is needed (buffering data, printer spool)
  - Compensation: compensate the unrecoverable operation (invalidation, sending an undo message)

Idealized FT component

- Backward error recovery
- Support requests for establish, discard, restore a recovery point

Fault treatment and continued service

Basics

- Previous 3 steps: errors were removed from system state
  Now: Faults have to be treated
- Fault treatment = fault location + system repair
  (similar: damage assessment + error recovery)
- Measure in hw (in sw: on-line difficult!)
- Minimal approach: Ignoring the fault
  - fault is transient (not for sw and design faults!)
  - recovery is powerful to cope with recurring faults
  - future operation avoids the manifestation of the fault

Fault location

- Exploratory approach: diagnostic checking
  (signature of faults + pattern matching)

System repair

- Reconfiguration:
  - manual
  - dynamic (in response to instructions)
  - spontaneous (by the system itself)
- Replace: stand-by spare (independent design: against design faults)
- Graceful degradation: disabled components but continued service

Idealized FT component

- Exception handling:
  - resumption model (resume the operation)
  - termination model (terminate operation): simple and safe
- Internal transient faults: retry the operation

Software fault tolerance

Basics

- Fundamental assumption: SW contains residual faults (despite of testing)
- SW FT: tolerate the faults in the software
- Basic techniques:
  - Recovery block scheme
  - N-version programming scheme
  - (N self-checking programming)

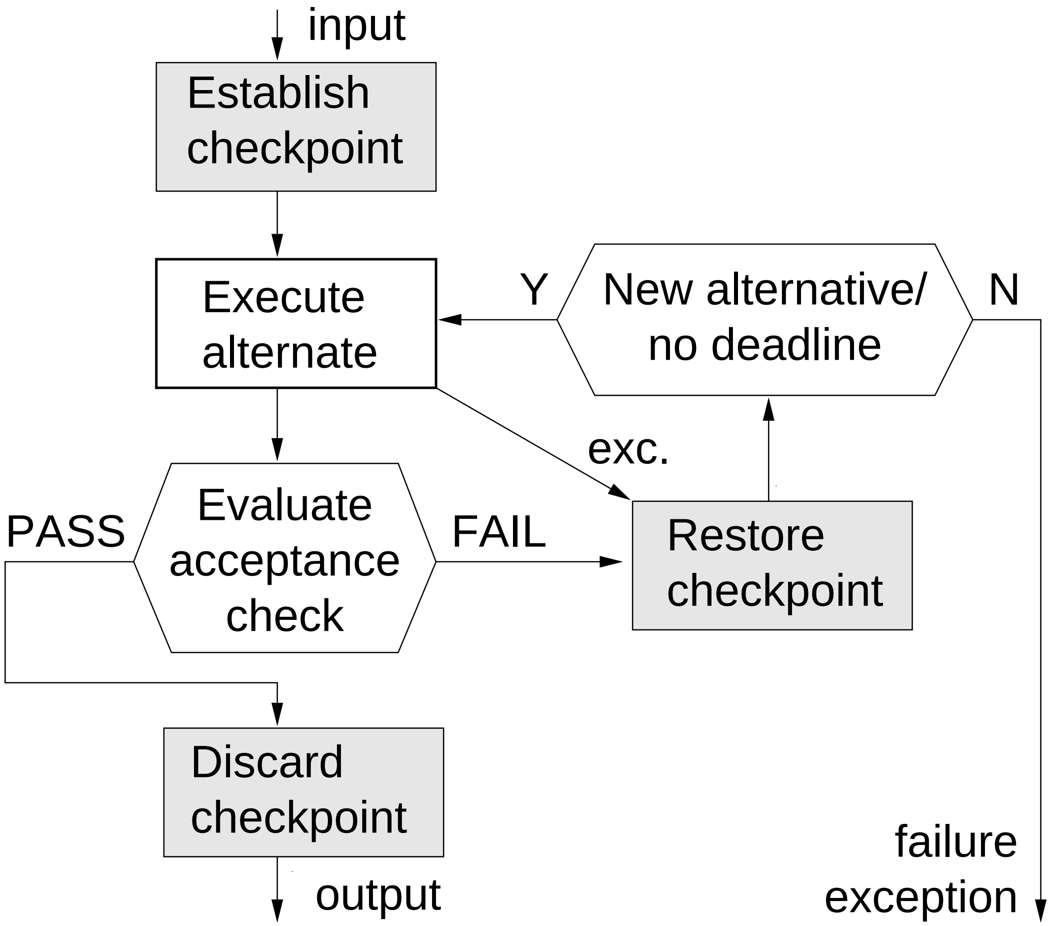
Recovery block scheme

- Aim: Masking module faults by passive redundancy
  Similar to stand-by spare in hardware, but
  - different design of modules
  - not a permanent replacement

- Phases of execution:
  1. Establish recovery point
  2. Execute primary module
  3. Acceptance test (error detection);
     if results accepted then exit; otherwise
  4. Restore initial state
  5. Execute alternate module (fault treatment)
  6. Continue in step 3 until alternate modules exit
  7. Error report
- General syntax:

```
ensure {acceptance_test}
by {primary_module}
else by {alternate_module_1}
else by {alternate_module_2}
...
else {error}
```

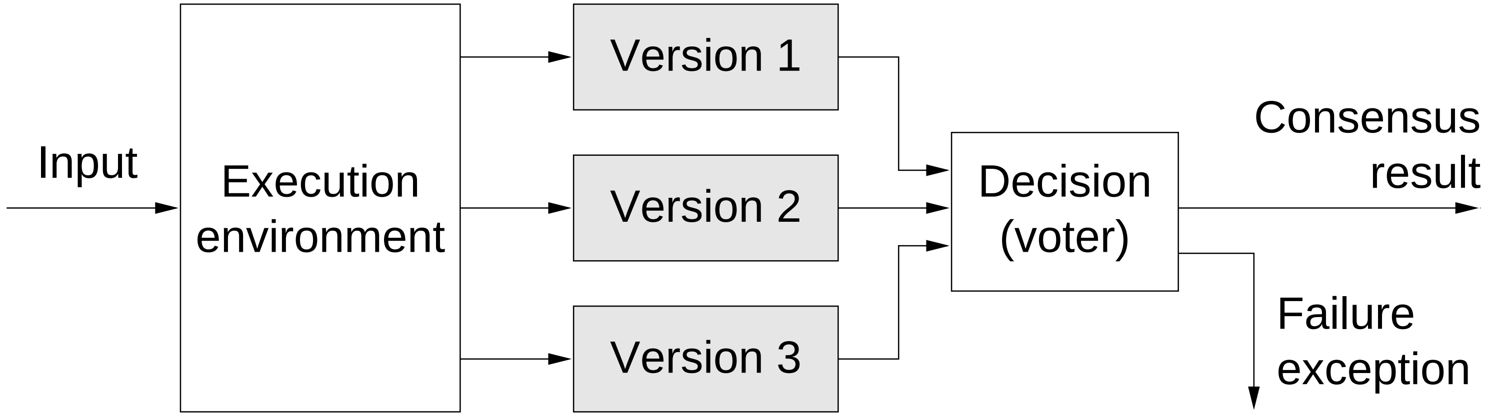
- Block diagram



- Acceptance test is required
  - comprehensive, but fast and simple
  - without design faults
  - check for *acceptability* (not correctness)
  - programmer required to deal with test (advantage!)
  - construction: based on (formal) specification
  - often cannot be performed (results are not known in advance)
- Application of recovery blocks
  - Interpreter support: special flow of control
  - More sw modules, starting in the same state
  - Alternate modules are less effective, more robust
  - Avoiding common design/specification faults by design diversity
  - Overhead: extra storage + execution time (checks+recovery always)

N-version programming

- Aim: Masking module faults by N-modular (static) redundancy
  - N variants are designed to satisfy the same specification
  - results are compared by replication checks (error detection)
  - error handling by (majority) voting
- System elements
  - N variants
  - decision algorithm (replication check)
  - execution environment



- Requirements for variants
  - common specification (functionality + input/output format)
  - diverse implementation
    * programmer (training, experience)
    * application algorithms and data structures
    * software development methods
    * programming language, environment and tools
    * testing methods and tools
- Decision algorithm
  - slightly different results of variants
  - allowable ranges, acceptable differences
- Execution environment
  - invoking the variants with consistent input
  - waiting and synchronizing results
  - start decision algorithm
- Problems
  - communication of (large-scale) inputs and results
  - exact voting often cannot be performed

Comparison of the two schemes

	Recovery block scheme	**N-version programming**
Checking of results	Acceptance check Absolute (specification) Results cannot be predicted	Voting check Relative (variants) Multiple solutions (ranges)
Execution	Sequential	Parallel (efficient) sequential (slow)
Redundancy	In case of an error	Always
Execution time	Summary of modules	Time of longest module
Error handling	Backward recovery (slow)	Error is masked (fast)
Minimal overhead	Acceptance test	Voting
Problems	Recovery	Distributing, collecting data
Fault tolerance	f faults by f+1 modules	f faults by f+2 modules
Versions/modules	Multiple versions are needed (design diversity)	

N self-checking programming

- Aim: Masking module faults by active dynamic redundancy
  - N self-checking software components (error detection)
  - parallel execution of components
  - one component is active at a time (delivers output)
  - error detection and switching of results
- Phases of execution
  1. Normal operation of the active component
  2. Failure of the active component: switch to a spare component
  3. Normal operation of the spare component
  4. Failure of the spare component: switch back or to other spare

Error recovery in concurrent systems

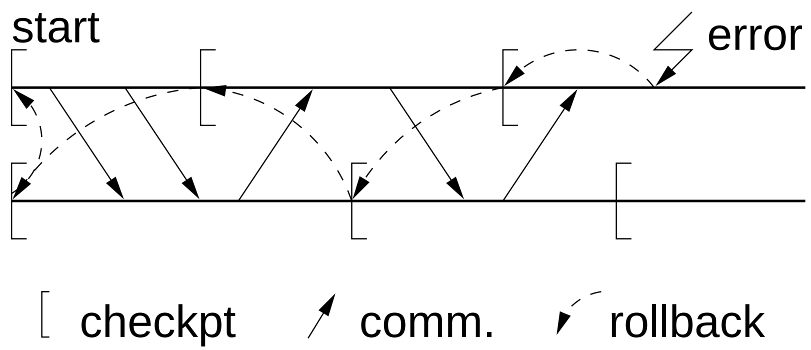
Basics

- Independent processes: disjoint sets of object used
  Recovery: individually in single processes
- Competing processes: Shared resources, but no information flow
  Recovery: individually
- Cooperating processes: Information flow (communication) between them
  Recovery: Consistent recovery is needed
  Individual recovery: leads to domino effect!

Domino effect

- Uncontrolled roll-back (propagation of recovery) due to communication
  - P1 recovered, but communication with P2
  - P2 recovered, but communication with P1, ..., and so on
- Inconsistent sets of checkpoints: roll-back to initial state

Domino effect



Consistency

- Goal: If an error is reported, find a consistent set of recovery points
- Definition: P1,...,Pn processes establish recovery points at t1,...,tn.
  The set of recovery points is consistent at t if
  - in period [ti,tj], Pi and Pj do not communicate
  - in period [ti, t], Pi does not communicate with external processes not in the subset
- Recovery line: A consistent set of active recovery points
- Backward recovery: to the closest (recent) recovery line

Recovery lines

- Planned: Constraints on communication to establish them
  Coordination (synchronization) of processes
- Associated with atomic actions: establish at the entry point
  Restorable atomic action:
  - at the entry, all processes establish recovery point
  - in the action, recovery points are not discarded
  - at the end, processes leave and discard recovery point

  Mechanism: Enforce atomic actions (structuring information flow)
- Unplanned recovery lines: Search for a recovery line
  not guaranteed to find one (domino effect)
- Compromise: Planned system-wide recovery lines
  Search for consistent set in the mean time

Comparison of techniques

	Advantages	Disadvantages
Planned recovery lines	guaranteed existence, simple mechanism, bounded overhead	constraints on communication
Unplanned recovery lines	no restriction on comm.	not guaranteed, complex searching (monitor), risk of domino effect

Distributed systems

- No centralized control for establish, discard, restore operations
- Checkpoint is a distributed database: consistency required
  - distributed commitment protocols (two-phase, three-phase protocols)
  - database management

Error recovery based on message logging

Classification of recovery techniques

- Semi-automatic techniques
  - Recovery blocks, conversations, atomic transactions
  - Global checkpointing
- User-transparent techniques
  - **Message logging**
  - Coordinated checkpointing
  - Hybrid techniques

Basics of message logging

- Automatic (user-transparent) technique
  - Processes save their state independently
  - Messages are stored in a message log

- Recovery:
  - The erroneous process restores its recently saved state
  - Lost messages are read from the log
  - Duplicated messages are skipped (on the basis of the log)
- Requirement: Deterministic execution of the process
- Advantage: Recovery of a *single* process
- Disadvantage: Large-scale message log is needed
- Basic techniques:
  - Pessimistic: Process is suspended until the message is logged
  - Optimistic: Asynchronous logging with bookkeeping
    - which messages were logged
    - which process depends on unlogged messages
    - rollback of processes whose messages were not logged.

Possible problems

- Domino effect
- Inconsistent messages (p sends m to q):
  - Lost message:
    - p restored to a state where m was sent
    - q restored to a state where m was not received
  - Duplicate message:
    - p restored to a state where m was not sent, and sends it again
    - q restored to a state where m was received
  - Orphan message:
    - p restored to a state where m was not sent, but does not send it again
    - q restored to a state where m was received

  Avoiding inconsistent messages: version numbers, bookkeeping
- Livelock
  - error in p after sending m1 and before receiving n1
  - p rolls back, sends m2 and receives n1
  - q is notified to roll back (undo the effects of m1)

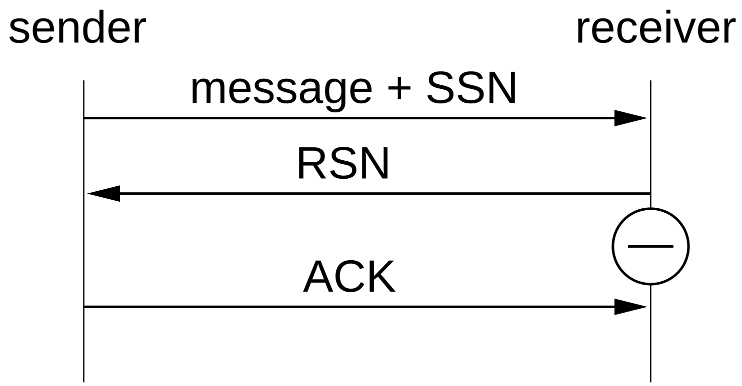
- q rolls back, sends n2 and receives m2
- p has to roll back to undo receiving of n1; sends m3
- q has to roll back to undo receiving of m2; sends n2 ...

A pessimistic approach: Publishing

- A recorder node (Ethernet) stores all messages reliably
  Synchronous saving: consistent with sender/receiver
- Processes save checkpoint independently
- Error recovery:
  - process restores its state
  - messages are replayed from the log
  - duplicated messages are absorbed (by kernel)
- Discarding a checkpoint: previous messages are deleted
- Disadvantages: Overhead; recorder is a bottleneck

An optimistic approach: Sender-based message logging

- Messages are stored in the sender
- Communication protocol:
  - Sender: sends a message with a SSN (sender sequence number) and stores it in the log
  - Receiver: receives a message and returns a RSN (receiver seq. nr.) it stores the highest SSN from this sender then suspends sending new messages
  - Sender: stores the RSN in the log and sends an ACK (acknowledgment)
  - Receiver: receives ACK and can continue sending new messages



- Recovery of the erroneous process p:
  - restoring the state of p
  - request for messages
  - logged messages are resent by senders in the order or RSN
  - partially logged messages are resent by senders in any order
  - p resends its messages with the same SSN as before
  - receivers ignore the (duplicated) messages
    with SSN not greater than the stored highest SSN from p
  - receivers send the same RSN for the duplicated messages as before
    (unless a checkpoint was taken)
- Establishing a checkpoint in a receiver:
  the senders are notified that the stored messages can be deleted
- Role of sequence numbers
  - SSN: recovery of a sender
    duplicated messages are discarded in receivers based on the table of highest SSN
  - RSN: recovery of a receiver
    resent messages from senders are received in the order of the RSN

Consistency in distributed systems

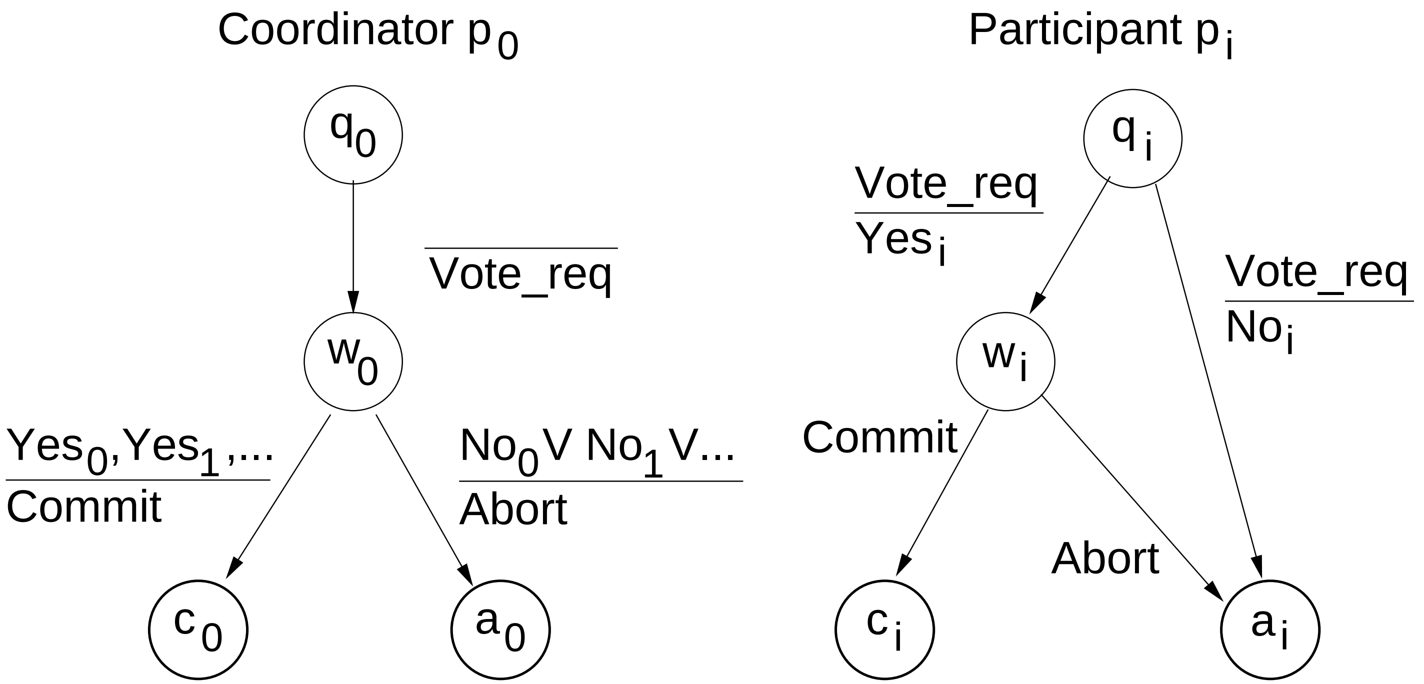
Basics

- Goal: Reach a consistent state after a distributed operation
- Agreement about the results of a distributed operation
  - distributed checkpoint establishment
  - recovery of cooperating processes
  - transactions in a distributed database

  The operation is successful if all participants executed it successfully, otherwise the previous (original) state is restored.
- Solution:
  - The decision is initiated by a coordinator
  - The participants *vote* (by Yes/No messages)
  - Agreement (Commit or Abort) by a commitment protocol

- Atomic commitment protocol
  - all participants get the same decision
  - participants cannot change their vote posteriorly
  - the decision is Commit if all participants vote Yes
  - if all participants vote Yes and there is no failure then the decision is Commit
  - after a repair of the failure, each participants gets the same decision
- Possible failures: site, communication (time-out)
- Uncertain state of a participant:
  it voted Yes but did not get the decision yet
- Blocking: The decision cannot be get until the erroneous participant(s) are repaired.
  - resources are reserved
  - the operation is delayed (which will delay further ones)

Two-phase commitment protocol (2PC)



Time-out in 2PC

- Participant waits for Vote_req: Decides to Abort
- Coordinator waits for Yes/No: Decides to Abort
- Participant p waits for Commit/Abort: Uncertain state!
  Ask the other participants: *Termination protocol*
  - Participants know each other (list with Vote_req)
  - p sends Decision_req for participant q
  - q got the decision: sends it to p
  - q voted No: sends Abort to p
  - q is uncertain: no answer

Blocking in 2PC: All living participants are in uncertain state

- Coordinator is dead
- Participants which might get the decision are dead
- Living participants cannot decide to Abort,
  because some dead participants might get Commit from the Coordinator

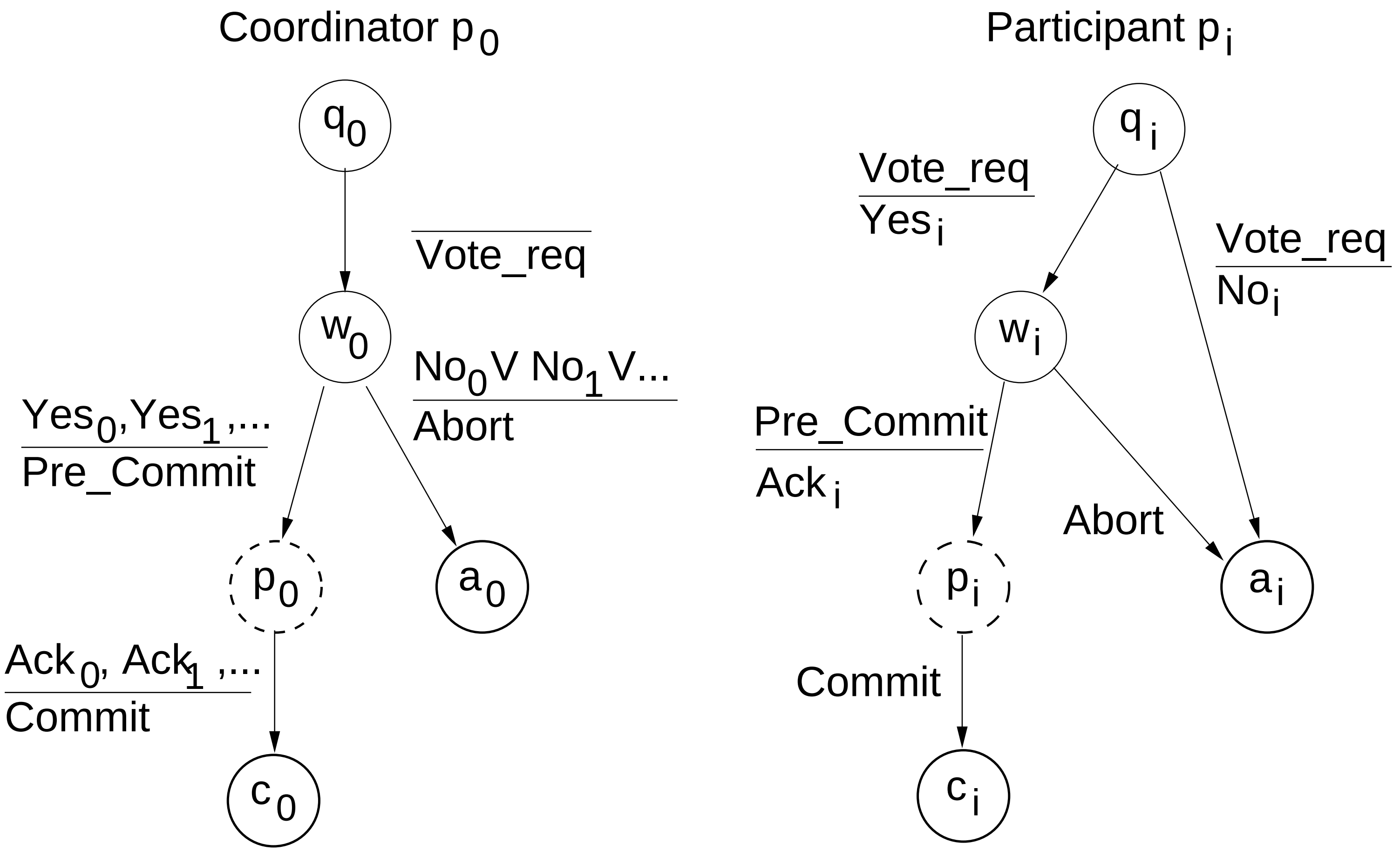
Recovery in 2PC

- State of the participant is saved in log (Start_2PC, Commit, Abort, Yes)
- Coordinator: Start_2PC in log
  - Commit/Abort in log: Decision
  - No Commit/Abort in log: Decide to Abort
- Participant: No Start_2PC in log
  - Commit/Abort in log: Decision
  - Yes in log: Uncertain state; termination protocol needed
  - No vote in log: Decide to Abort

Evaluation of 2PC

- Resilience: site + communication failures (time-out)
- Blocking: Exists (all living participants are uncertain)
- Time complexity: 3 (+ 2 for termination)
  (decentralized: 2, linear: $2n$)
- Message complexity: $3n$ (n participants)
  (decentralized: $n^2 + n$, linear: $2n$)

Three-phase commitment protocol (3PC)



New elements in 3PC: To avoid blocking

- In 2PC: Blocking if all participants are uncertain
  They can not decide to Abort: One might got Commit
  Problem: At the same time uncertain participants + ones that got decision
- In 3PC: New messages and additional states
- New message: Pre_Commit
  Pre_Commit means: All voted Yes; the decision will be Commit if nobody will fail
- New message: Ack
  Ack means: The participant finished the uncertain state
- If a participant is uncertain, there are no participants which got the decision
- Decision is sent only if there are no uncertain participants

Time-out in 3PC

- Participant waits for Vote_req: Decides to Abort
- Coordinator waits for Yes/No: Decides to Abort
- Coordinator waits for Ack: Decides to Commit
  - All participants voted Yes
  - Some participants failed
  - After recovery, they will get the decision
- Participant waits for Pre_Commit/Abort: Termination protocol started
  Coordinator is dead; participants have to decide
- Participant waits for Commit: Termination protocol started
  Why not decide Commit?
  - Participant got Pre_Commit
  - Other participants may be in uncertain state:
    They might not get Pre_Commit before the failure of the coordinator

Termination protocol in 3PC

1. Election of a new coordinator (election protocol)
   Initiated by the participant which detected the time-out
2. New coordinator asks the state of participants: State_Req
   - Aborted: got Abort, voted No or not voted
   - Uncertain: voted Yes, did not get Pre_Commit or Abort
   - Committable: got Pre_Commit, did not get Commit
   - Committed: got Commit
3. Decision based on the state information:
   - exists Aborted: Abort
   - exists Committed: Commit
   - all Uncertain: Abort (in 3PC it is possible!)
   - exists Committable (others Uncertain): Pre_Commit for the uncertains, waiting for Ack; sending then Commit
4. Time-out in the termination protocol:
   - In election protocol: Dead participants are ignored
   - failure of the new coordinator: new election

Coordinator election in 3PC termination protocol

- Participants have priorities; the one of highest priority is the candidate
- List of living participants is maintained
- Time-out detected: You_Are_Elected message for the candidate
  (Previous coordinator is removed from the list)

Recovery in 3PC

- State of the participant is saved in log (Yes, Commit, Abort)
  - No vote in log: Decide to Abort
  - Commit/Abort in log: Decision got
  - Yes in log, no Commit/Abort: Ask the other participants
    (Termination protocol is not started!)
  - Pre_Commit in the log: Ask the other participants
    If coordinator fails, the others might decide Abort (termination protocol)
    Pre_Commit is not stored in the log

Blocking in 3PC: Only if a total site failure occurs

- If all living participants are uncertain, they can decide to Abort (there is no participant which might got the decision)
- Total site failure: Recovered participants have to wait for
  - a participant recovers which got the decision
  - a participant recovers which may decide (voted No)

Evaluation of 3PC

- Resilience: Site failures only
- Blocking: Only if a total site failure occurs
- Time complexity: 5 (+ 6 for termination)
- Message complexity: $5n$

Communication failures and 3PC

- Communication failures: Inconsistent decision in termination protocol
- Example: Two unconnected partitions
  1: All participants Uncertain, decide to Abort
  2: All participants Committable, decide to Commit
- Enhanced termination protocol: A new coordinator can decide only if the majority of participants can get the decision
  - Asks the state of participants as before
  - Sends Pre_Commit or Pre_Abort
  - Waits for Ack
  - Decides if the majority of participants sent Ack
  - Sends Commit or Abort
- Enhanced coordinator election protocol:
  Candidate checks whether there is a participant of higher priority

Managing replicated data

Basics: Consistency and recoverability in distributed databases

- Basic operations in transactions: Read, Write, Abort, Commit
- Abort operation:
  - Undo the effects of Write operations
  - Abort the transactions which read the results of Write: Cascade Abort
  - *Commit is delayed until all transactions that wrote the data read by this transaction has committed.*
- Avoiding cascade abort (not necessary always)
  - *Read is delayed until all transactions that wrote the data read by this transaction has committed.*
- Restoring the original value: before image is result of other transactions!
  - *Write is delayed until all transactions that wrote the same data has committed.*
- User I/O:
  - Output is delayed until the transaction is committed
  - After an abort, a response to the output of the transaction must not be used again

Managing concurrent transactions

- Interleaving may lead to inconsistency

Lost update		Inconsistent retrieval	
Tr1	Tr2	Tr1	Tr2
read(x)			write(x)
increase(x)		read(x)	
	read(x)	read(y)	
write(x)	decrease(x)		write(y)
	write(x)	print(x+y)	

- Serial execution: Inefficient
- Serializability: *The result of a serializable interleaved execution is the same as a serial execution of the transactions.*
- If the execution order is important, the user can decide

Replicated data

- Goals of replication:
  - Fault tolerance (site or communication failure)
  - Increased performance (one of the replicas is accessible fast)
- Condition for correctness:
  - The user manages a one-copy database
  - One-copy serializability: *The interleaved execution of transactions on a replicated database is the same as a serial execution of the same transactions on a one-copy database*
- System model:
  - data manager: starting transactions
  - local scheduler: managing serializability conflicts on local data
  - transaction scheduler: atomic commitment
- Possible failures
  - Site failure: No response for a request (time-out)
  - Communication failure: No response from a given direction
    Special problem: partitions

- Writing replicated data
  - immediately with a Write operation
  - delayed writing: before commit

    Advantages:

    - buffering writes to a same site (reducing the number of messages)

    - simple abort (skip the write buffer)

    Disadvantages:

    - commit delayed

    - detection of conflicts delayed (solution: execution on one copy)

Overview

- Basic techniques:
  - Write all, Write all available
- In case of site faults:
  - Available copies with validation
- In case of communication faults:
  - Site quorum
  - Quorum consensus
  - Virtual partitions

Write all approach

- Implementation:
  - Read the closest replica: Read(x)=Read(x_i)
  - Write all replicas (commit): Write(x)= { Write(x_1), Write(x_2), . . . ,Write(x_n) }
- Ideal case (basic approach)
- Failures: Write can not be performed

Write all available approach

- Implementation:
  - Read the closest replica:
    Read(x)=Read(x_i)
  - Write all available replicas:
    Write(x)={ Write(x_1), Write(x_2), . . . ,Write(x_k)}, $k \geq n$
- Database consistency is not assured
  - Recovered replicas has invalid data (update needed)
  - One-copy serializability is not valid:
    Local scheduler of failed replicas does not detect conflict
    Result is different from any serial execution

Tr1: Read(x), Write(y)	Tr2: Read(y), Write(x)
Read(x_a)	Read(y_a)
y_a replica fails	x_a fails
Write(y_a) is not possible	Write(x_a) is not possible
Write(y_b)	Write(x_b)
Commit	Commit

Available copies with validation

- Implementation:
  - Read the closest replica: Read(x)=Read(x_i)
  - Write all available replicas: Write(x)={ Write(x_1), . . . ,Write(x_k)}, $k \geq n$
- Managing recovered replicas: initialization is needed (cannot be read until it is written)
- Validation protocol before commit:
  - Missing writes validation: The failed replicas are still not available
    - Unavailable(x_k) message for each failed replica
    - if it returns OK (it was initialized), then abort the transaction
  - Access validation: The written replicas are still available
    - Available(x_k) message for each replica
    - commit only if all replicas return OK
- **Validation protocol:**
  - Avoids inconsistency of Available copies approach
  - Missing writes validation is necessary only if failed replica exists
  - Access validation can be performed together with a commit protocol

Communication failures

- The database is split into partitions
  Independent transactions in different partitions (Available copies approach):
  - P1: x_a, x_b, y_a;
    T1: Read(y_a), Write(x_a), Write(x_b), Commit
  - P2: x_c, y_b, y_c, y_d;
    T2: Read(x_c), Write(y_b), Write(y_c), Write(y_d), Commit
- Goal: **A given data can be accessed in one partition only**
- Active partitions:
  - the majority of sites
    P2 is active
  - the majority of replicas
    P1: x can be accessed
    P2: y can be accessed

Site quorum

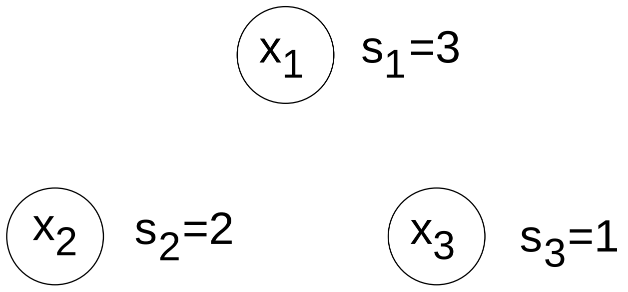
- Implementation
  - Weights are assigned to sites
  - Quorum: the majority of weights
  - Active partition: the quorum is in the partition
  - In the active partition: Available copies (with validation) approach
- Problems:
  - Too much partitions: there is no quorum
  - Inconsistent view of partitions if there are common sites
    P1: x_a, x_b; T1: Read(x_a), Write(x_a), Write(x_b), Commit
    P2: x_a, x_c; T2: Read(x_c), Write(x_a), Write(x_c), Commit
    (x_a in both partitions; different replicas in a partition)
- Solution: Enhanced algorithms

Quorum consensus

- Definition of quorum
  - Weights are assigned to replicas: $s_1, s_2, \ldots, s_n$; $S = \sum_{i=1}^{n} s_i$
  - Thresholds:
    - write threshold WT with $2WT > S$
    - read threshold RT with $WT + RT > S$
  - Quorum:
    - write quorum: set of replicas where $\sum s_j \geq WT$
    - read quorum: set of replicas where $\sum s_l \geq RT$
  - Consequences: Each write quorum has a common replica
    - with any other write quorum
    - with any other read quorum
  - Version numbers:
    - new data is written using an increasing version number

- Implementation in the active partition
  - Write(x) = { Write(x_1), Write(x_2), . . . ,Write(x_k)},
    where the replicas form a write quorum
  - Read(x) = { Read(x_a), Read(x_b), . . . ,Read(x_m)},
    where the replicas form a read quorum
- Managing version numbers
  - Read: Get the data of highest version number from the read quorum
  - Write: Update the version number in write quorum
    - read the version number of each replica in write quorum
    - increase the highest version number
    - write back the new number in each replica of the write quorum
- Advantages
  - Recovery is automatical
- Disadvantages
  - Read(x) accesses multiple replicas
  - Write(x) maintains version numbers
  - Fault tolerance is inefficient (like a majority vote)
  - New replica needs the re-definition of WT and RT

Quorum consensus: An example



$S = s_1 + s_2 + s_3 = 6$

$WT = 4 \quad (2WT > S)$

$RT = 3 \quad (RT + WT > S)$

Write quorums: (x_1, x_2, x_3), (x_1, x_2), (x_1, x_3)

Read quorums: (x_1), (x_1, x_2), (x_1, x_3), (x_2, x_3)

Virtual partitions

- Definition of quorum: like in Quorum consensus
- Definition of a view:
  - view of a site $v(A)$: set of sites reachable from A
  - view of a transaction $v(T)$: view of the site initiating T
- Implementation: **Write all approach in $v(T)$**
  - Read(x) can be executed if there is a read quorum in $v(T)$
  - Write(x) can be executed if there is a write quorum in $v(T)$
  - version numbers are maintained by the Write operation
- Managing $v(t)$: **View formation protocol**
  - Started if:
    - a site in the view fails (no answer in transactions, results in Abort)
    - a site not in the view will be available (request)
  - Basic tasks:
    - update the view in each site of the view
    - update data in each replica in the view

- Identification:
  - VID is assigned to each view
- Steps of the protocol:
  1. Site A detecting the change of view generates a $newVID > VID$
  2. Join_View($newVID$,A) is sent to each available site
  3. Site B accepts and acknowledges the message if $VID_B < newVID$
     otherwise rejects it
  4. Site A got rejection: restarts the protocol with greater $newVID$
  5. No rejection: Forms the new view (the answering sites)
     sends the new view to each answering site (View_Formed message)
  6. Each site sets the new view and $newVID$
  7. Each read quorum of the new view is read;
     replicas are updated using the data of highest version number

- Properties
  - Read(x) accesses a single copy
  - Write(x) maintains version numbers
  - Replicas are updated only when a new view is formed (maintenance)