2

Introduction to Fault Tolerance

Foundations and Paradigms

Fault tolerance foundations

The failure of computers

- *Why do computers fail and what can we do about it?*

  [J. Gray]
- Because:
  - All that works, fails
  - We tend to overestimate our HW e SW--- that's called faith☺
- So, we had better prevent (failures) than remedy
  - Must do it in a predictable and repeatable way
- Short of faith, we need:
  - a scientific way to quantify, predict, prevent, tolerate, the effect of disturbances that affect the operation of the system

---

The failure of computers

- *Why do computers fail and what can we do about it?*

  [J. Gray]
- Because:
  - All that works, fails
  - We tend to overestimate our HW e SW--- that's called faith☺
- So:
  - We had better prevent (failures) than remedy
- Dependability is ...
  - that property of a computer system such that reliance can justifiably be placed on the service it delivers
- Why?
  - Because (faith notwithstanding) it is the scientific way to quantify, predict, prevent, tolerate, the effect of disturbances that affect the operation of the system

Does not get better with distribution

- *A distributed system is the one that prevents you from working because of the failure of a machine that you had never heard of.*
  [L. Lamport]
- Since:
  - Machines fail independently, for a start
  - But they may influence each other,
  - They communicate through unreliable networks, with unpredictable delays
- ...gathering machines renders the situation worse:
  - The reliability (<1) of a system is the product of the individual component reliabilities, for independent component failures
  - R(10 @ 0.99)= 0.9910= 0.90; R(10 @ 0.90)= 0.9010= 0.35

Can get much worse with malicious failures

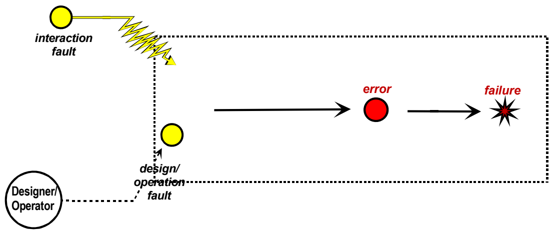
- Failures are no longer independent
- Failures become more severe
- Fault models become less representative

... Hackers don't like stochastics ...

Faults, Errors and Failures

- A system failure occurs when the delivered service deviates from fulfilling the system function
- An error is that part of the system state which is liable to lead to subsequent failure
- The adjudged cause of an error is a fault
- EXAMPLES:
  - Fault --- stuck-at '0' RAM memory register
  - Error --- what happens when the register is read after '1' is written
  - Failure --- the wrong reading ('0') is returned to the user buffer
- SOLUTIONS?
  - Remove the faulty memory chip
  - Detect the problem, e.g. using parity bits
  - Recover from the problem, e.g. using error correcting codes (ECC)
  - Mask the problem, replicating the memory and voting on the readings

sequence fault→ error→ failure

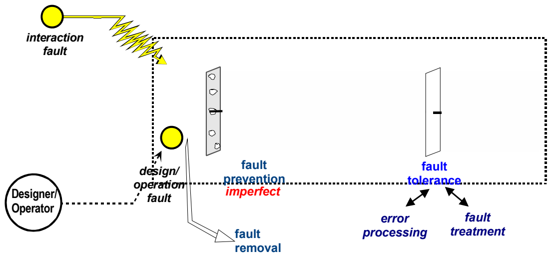


Achieving dependability

- Fault prevention
  - how to prevent the occurrence or introduction of faults
- Fault tolerance
  - how to ensure continued correct service provision despite faults
- Fault removal
  - how to reduce the presence (number, severity) of faults
- Fault forecasting
  - how to estimate the presence, creation and consequences of faults

---

Dependability measures



Types of Faults

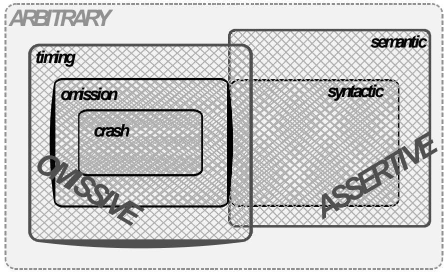
- Physical
- Design
- Interaction (*)
- Accidental vs. Intentional vs. Malicious (*)
- Internal vs. External
- Permanent vs. Temporary
- Transient vs. Intermittent

(*) Especially important in distributed systems and security

Interaction Fault classification

(specially important in distributed systems)

- Omissive
  - Crash
    - host that goes down
  - Omission
    - message that gets lost
  - Timing
    - computation gets delayed
- Assertive
  - Syntactic
    - sensor says air temperature is 100°
  - Semantic
    - sensor says air temperature is 26° when it is 30°



Dependability properties

- Reliability
  - the measure of the continuous delivery of correct service (ex. MTTF)
- Maintainability
  - the measure of the time to restoration of correct service (ex. MTTR)
- Availability
  - measure of delivery of correct service with respect to alternation between correct and incorrect service (ex. MTBF/(MTBF+MTTR))
- Safety
  - the degree to which a system, upon failing, does so in a non-catastrophic manner

---

Forms of redundancy

- Space redundancy
- Time redundancy
- Value redundancy

Error processing techniques

- error detection
  - detecting the error after it occurs aims at: confining it to avoid propagation; triggering error recovery mechanisms; triggering fault treatment mechanisms
- error recovery
  - recovering from the error aims at: providing correct service despite the error

  backward recovery:

  the system goes back to a previous state known as correct and resumes

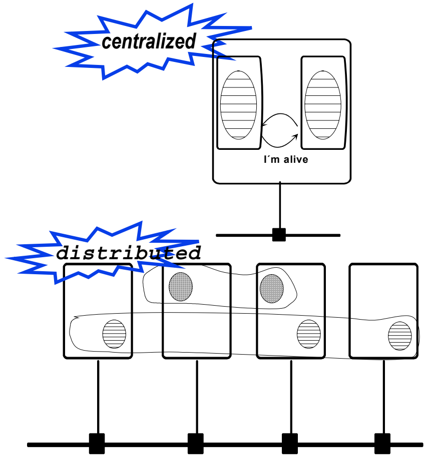
  forward recovery:

  the system proceeds forward to a state where correct provision of service can still be ensured
- error masking
  - the system state has enough redundancy that the correct service can be provided without any noticeable glitch

---

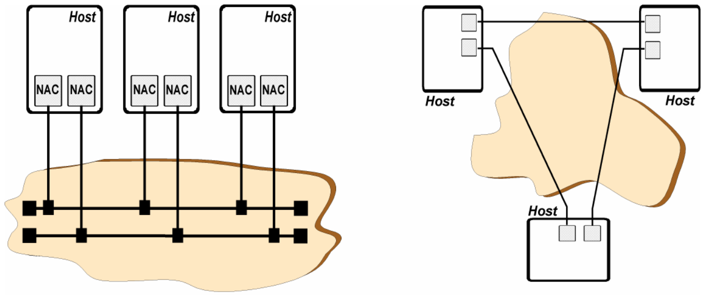
Foundations of modular and distributed fault tolerance

- Topological separation
  - failure independence
  - graceful degradation
- Replication
  - software vs. hardware
  - fine granularity
  - Resource optimization
- incremental T/F by:
  - class (omissive, semantic)
  - number of faults
  - number of replicas
    - pairs, triples, etc.
  - Type of replica control
    - active, passive
    - round robin, voting



Example Fault Tolerant Networks and Architectures

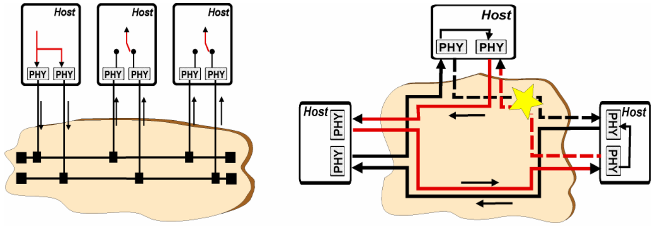
Redundant Networks



(a)

(b)

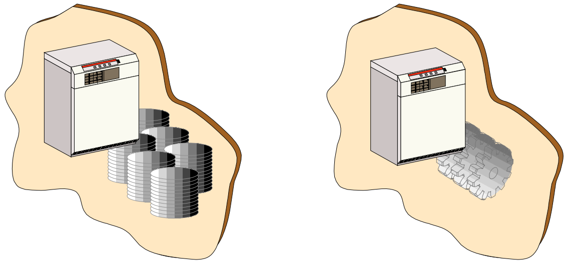
Redundant Media Networks



(a)

(b)

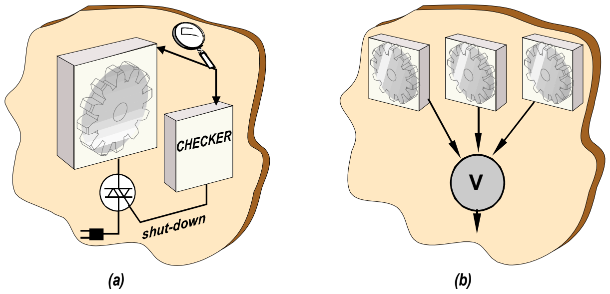
Redundant Storage and Processing



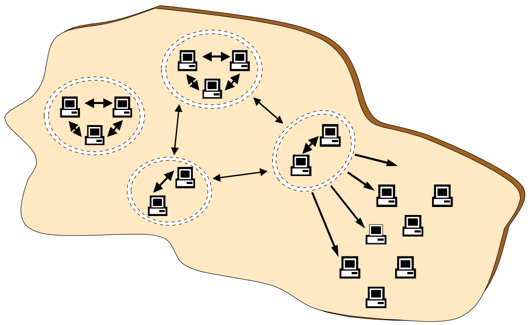
(a)

(b)

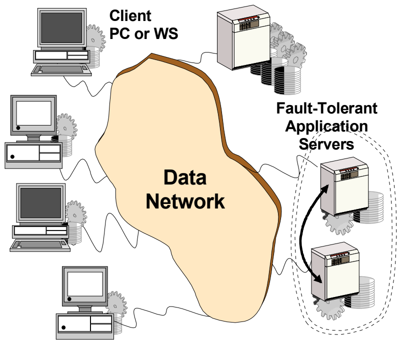
Error Detection and Masking



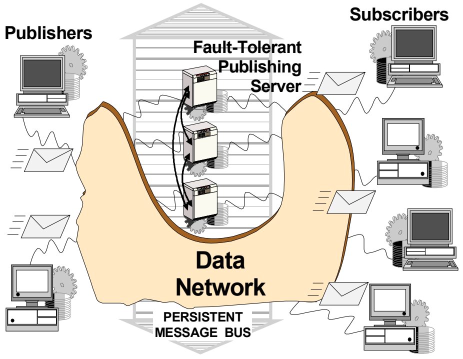
Modular Distributed FT with Replica Sets



Client-Server with FT Servers



FT Publisher-Subscriber



Distributed fault tolerance paradigms

Failure detection

- Crash failure detection:
  - How to detect a node stopping?
- Mechanisms:
  - Heartbeats or Probes
- Heartbeats
  - observed component periodically sends messages
- Probes
  - observed component waits for a probe message and replies
- Decentralisation
  - ideally any process plays the role of an observer (to monitor the activity of other processes) and of a target (i.e., it is monitored by all the other processes)

Failure detection

properties and problems

- consistency of distributed failure detection is a must:
  - when a process goes down all the other processes know about it and can coordinate their actions to implement corrective measures.
- Strong Accuracy
  - a safety requirement, specifying that no correct process is ever considered failed
- Strong Completeness
  - a liveness requirement, specifying that a failure must be eventually detected by every correct process
- If perfect channels are available, heartbeat exchanges meet strong accuracy and strong completeness
- Such a detector is called perfect failure detector :
  - If a node crashes all correct nodes will note the absence of the heartbeat at the same time and will detect the failure.

Failure detection

properties and problems

- channel imperfection impossible to overcome:
  - the lack of bounds for the timely behavior of system components (processes or links) – called *asynchrony*
- “funny” consequence:
  - no way to distinguish a missing from "extremely slow" heartbeat
  - happens if a link can delay a message arbitrarily, or if a process can take an arbitrary amount of time to make a processing step
  - perfect failure detection cannot be implemented in asynchronous systems!!!
  - problem is that for practical purposes, Internet “is” asynchronous

Failure detection
properties and problems

- something in between
- Weak Accuracy
  - at least one correct process is never considered failed by all correct processes
- Weak Completeness
  - a failure must be eventually detected by at least one correct process
- even the above is impossible in asynch systems:
- Eventual Weak Accuracy
  - there is a time after which some correct process is never considered failed by any correct processes

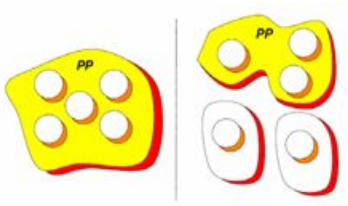
---

Problem of Partitioning

- partitioning is caused by the crash of one or more links that split the network in disjoint subsets or partitions
  - processes within the same partition are able to communicate among themselves but unable to communicate with processes in other partitions
  - serious problem because it prevents processes in different partitions from coordinating their activities
- remedies:
  - allow progress in all partitions, which causes state divergence, which is reconciled after healing
  - allow progress only in one partition
    - + : prevents state divergence
    - - : blocks all processes in the other partitions

Primary Partition

- primary partition
  - only partition that makes process
- how best done:
  - the one with the majority of elements
- caveat:
  - network can be partitioned in such a way that no primary partition can be identified
  - the system blocks completely until the partitions merge



Consensus properties (recap)

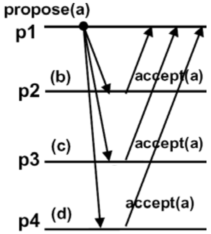
- Validity
  - if a process decides v, then v was proposed by some process
  - no process decides more than once
- Agreement
  - No two correct processes decide differently
- Termination
  - Every correct process eventually decides

- Consensus is equivalent to atomic broadcast
  - That is, one can implement one with the other
  - Does not mean that all such implementations are efficient!

Fault tolerant consensus

(intuition)

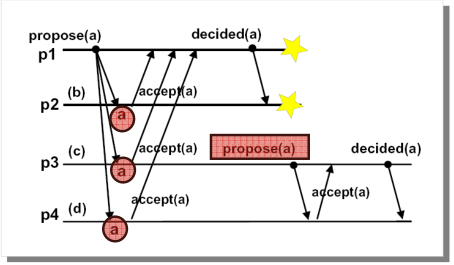
- easy solution:
  - coordinator (p1) sends decision (a), followers accept
- failure of coordinator:
  - pick next (p2), who sends its initial value (b)
- serious problem:
  - if coord crashed during dissemination, some may have (a) and others (b)
  - violates consensus properties
- solution:
  - only decide when sure only one value pending
- how to *lock* such a decision?



Fault tolerant consensus

(intuition)

- when a process receives the initial value from the coordinator, it changes its initial value to that of the coordinator
- any sequence of recovery coordinators will use same value, if it had been proposed



- protocol:
  - coordinator sends its value to every other process
  - processes update their initial value and send an ack back to coordinator
  - when coordinator receives ack from every process, it knows the value is locked
  - even if it crashes, the new coordinator will also propose that same value
  - coordinator has to disseminate a *decided* message to inform the remaining processes of that fact
  - a process that receives *decide* can safely decide on that value

---

Fault tolerant consensus

(intuition)

- previous solution only works with a reliable failure detector
- which cannot be implemented in asynchronous systems
- consensus (and thus atomic broadcast) is not solvable in asynchronous systems
- consensus only solvable in systems with at least an eventually weak failure detector, as long as a majority of processes do not crash.

Fault tolerant consensus

(with an eventually weak failure detector)

- protocol:
  - similar to the protocol described above
  - coordinator simply waits for a majority of acknowledgments to lock a value, instead of all acks
  - allows the system to make progress as long as a majority of processes can communicate
  - no matter whether remaining processes are crashed or slow
- coordinator failure
  - another process pj becomes the coordinator
  - old coordinator pi may have locked a value without pj knowing
  - but majority of processes will know that such a previously locked value exists
  - new coordinator has to contact a majority of processes in order to "check" that, else he can propose his own

Membership

(recap)

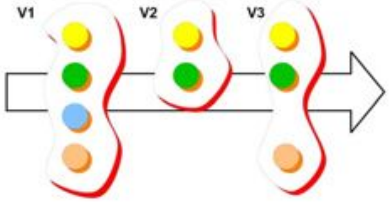
- group membership
  - set of processes belonging to the group at a given point in time
- membership service:
  - keeps track of membership and provides info to group members
- group view:
  - subset of members mutually reachable at a given point
- group membership is often dynamic:
  - in response to user demand or changes in the runtime environment (load, failures, etc)
  - it may grow, by letting new processes join the group
  - it may shrink, by letting members leave the group
  - view changes when processes fail or when they recover

Membership under faults

- group membership is a form of distributed agreement and as such is a hard problem in the presence of faults
- consistent membership view:
  - if membership of the group unchanged and there are no link failures, all members should obtain the same group view
- even this simple predicate can be hard to enforce
  - membership heavily relies on failure detection
  - inaccurate or unreliable failure detection may cause membership to have erratic behavior
- another important predicate is the order in which view changes should be seen by all

---

Linear membership

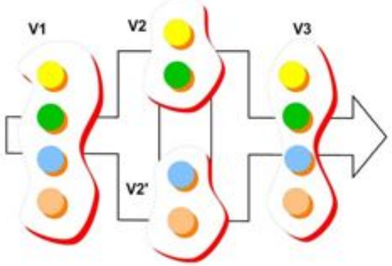


- Linear Membership
  - the history of views delivered to any correct process is a prefix of the history of views delivered to all correct processes
  - characterized by enforcing a total order on all views
  - all correct processes receive exactly the same sequence of views
- easy to enforce on synchronous systems

Linear membership issues

- linear membership not easy or desirable in partitionable systems:
  - should keep delivering views in both partitions
  - group view splits and merges in response to changes in the network connectivity
  - views become partially ordered, and sometimes overlap in different partitions
- the idea is to find a useful partial view ordering paradigm

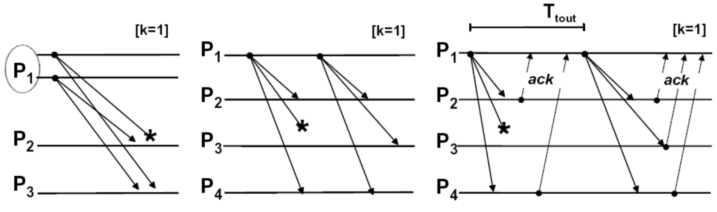
Strong partial membership



- Strong Partial Membership
  - concurrent views never intersect, e.g., V2 and V'2 correspond to two completely disjoint partitions, which later merge into V3
  - Strong partial membership supports virtual synchrony

F/T Communication

(Communication Error Processing techniques)

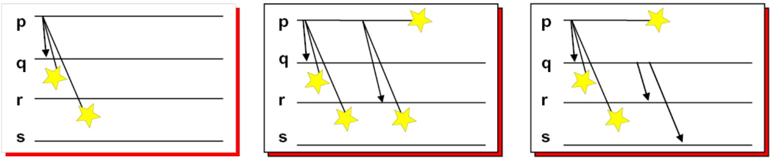


(a) *(b)* *(c)*

- Communication Error Processing techniques:
- (a) Masking (Spatial);
- (b) Masking (Temporal);
- (c) Detection/Recovery

F/T Communication

(Resilience to Link and Sender Failure)



(a) *(b)* *(c)*

- (a) Unreliable multicast
  - no effort is made to overcome link failures, it is as reliable as the link and the sender are
- (b) Best-effort multicast
  - sender takes some steps to ensure the delivery of the message, like retrying or repeating, but not if sender fails
- (c) Reliable multicast
  - participants coordinate to ensure that the message is delivered to all correct recipients, including when sender fails

---

Reliable multicast

- A reliable multicast protocol is defined formally in terms of the following properties:
- **Validity:**
  - If a correct process multicasts (sends) a message M then some correct process in *group(M)* eventually delivers M.
- **Agreement:**
  - If a correct process delivers a message M then all correct processes in *group(M)* eventually deliver M.
- **Integrity:**
  - For any message M, every correct process *p* delivers M at most once and only if *p* is in *group(M)*
  - If process *p* delivers M and *sender(M)* is correct, then M was previously multicast by *sender(M)*.

Byzantine agreement (BA)

(tolerating semantic faults)

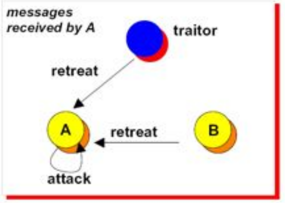
- A fundamental problem illustrating how hard it is ...
  - a number of generals, in face of an enemy army, must decide whether to attack or to retreat, but they cannot meet, they can only do so *by sending messages to each other*
  - most of these generals are loyal to each other (correct) but some are traitors (faulty).
  - in the presence of favorable conditions, the combined force of the loyal armies can defeat the enemy.
  - however, unless all loyal generals attack together, their troops will be defeated.
  - traitors will, maliciously, try to prevent agreement from being reached.
  - traitors may invent messages, omit some or all messages, send conflicting information, etc.

Byzantine agreement

(tolerating semantic faults)

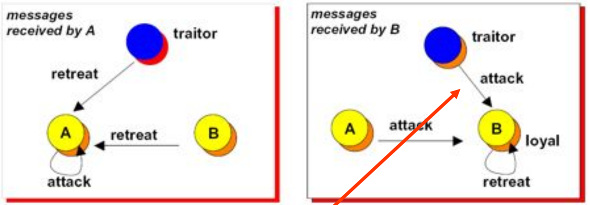
- assume that the BA protocol operates in rounds.
- in each round, generals send messages to each other
- loyal generals *must agree on a single binary value* (attack/retreat) despite the action of traitors
- loyal generals have pre-agreed that they should follow the *majority* and, in case of ties, retreat
- the initial value proposed by each loyal general consists of his own assessment of the correct decision: to *attack* or *retreat*
- *How many traitors are sufficient to prevent agreement?*

Byzantine agreement



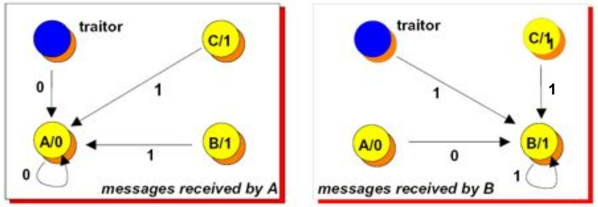
- Scenario:
  - three generals, two of them loyal
  - loyal generals: A, wishes to attack; B, believes they must retreat.
  - A receives two retreat messages, one from B and one from traitor
- intuitively, with majority vote, a correct decision should be possible, right?
  - if we discarded traitor's opinion, decision would be retreat (tie)
  - Unfortunately, *A cannot safely decide*

Byzantine agreement



- Why can't A safely decide?
  - traitor can send a conflicting message to B supporting A's proposal to attack
- A simple majority would force *A to retreat and B to attack*!
  - For A: < attack, retreat, retreat > => retreat
  - For B: < attack, retreat, attack > => attack
- i.e., *2f+1 are not enough, for f traitors*

Byzantine agreement



- Let us add an additional loyal general to the system
  - 'retreat' is 0 and 'attack' is 1 ; four generals, three of them loyal
- Maj of loyals, so it must work, but ... not in one round
  - by sending an attack vote to B and a retreat vote to A, the traitor can force A and B to disagree
- but we need an additional round of messages
  - For each sender p, the other three remaining processes exchange the values they have received from p to agree on the value sent by p
- *BA is possible with n=3f+1, for f faults, in worst-case f+1 rounds*

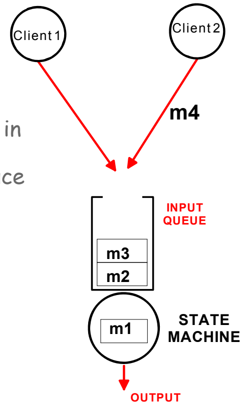
Replicated computations

- decentralized fault-tolerant applications may run replicated pieces of code which should behave in the same way
- replica determinism:
  - two replicas, departing from the same initial state and subject to a same sequence of inputs reach the same final state and produce the same sequence of outputs
- atomic broadcast:
  - guarantees "same sequence of inputs" objective
  - the rest lies with the replica itself
- issues:
  - deterministic coding
  - state divergence with partitioning
  - replica failure and recovery

Replication management

(partition-free systems)

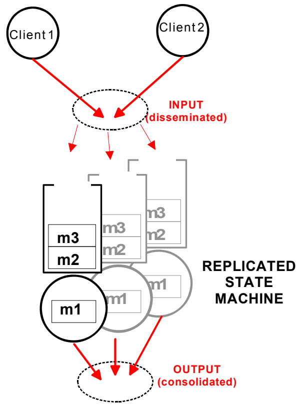
State Machine programming



- Characteristics
  - confinement - atomic commands
  - fault tolerance - easy replication
- Execution model
  - servers start in same state
  - execute same sequence of input commands, in same order
  - commands modify state variables and produce outputs (I/O or return results)
  - THEN: all follow same sequence of state/outputs
- Programming
  - message-based, diffusion (multicast)
  - requires deterministic execution
  - open-loop
  - reduces concurrency if cmds are long

Replicated State Machine
(active replication)

- replicated state machine:
  - all replicas execute at same time
  - achieves error masking
  - determinism mandatory
- replica quorums:
  - benign communication
  - omissive process failures - f+1 replicas
  - affirmative process failures – 2f+1 replicas
- message ordering:
  - total order of commands to replicas
  - same commands in same order => same results



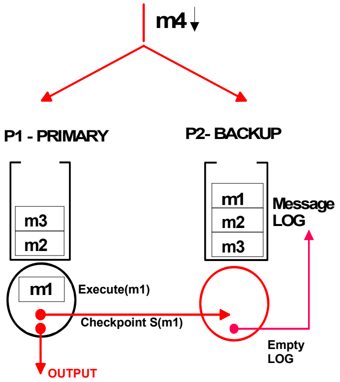
Replicated computations
(issues)

- non-deterministic component
  - state and behavior depend not only on the sequence of commands it executes but also on local parameters that cannot be controlled.
- many mechanisms can cause a non-deterministic behavior:
  - non-deterministic constructs in programming languages such as the Ada select statement;
  - scheduling decisions; resource sharing with other processes;
  - readings from clocks or random number generators; etc.
- state of two non-deterministic replicas may diverge even when they execute same sequence of inputs

---

Replicated State Machine
(passive replication)

- passive replication
  - only Primary executes
  - in the order it decides
  - supports preemption and non-determinism (active rep. doesn't)
- state transferred to Backup(s)
  - inter-replica deferred state-level synchronization (checkpoints)
  - Backup(s) log commands until checkpoint received
  - Primary fails: Backup assumes
  - potentially long *takeover-glitch*
- message ordering:
  - non-ordered message diffusion



Replication management
(partitionable systems)

Rationale

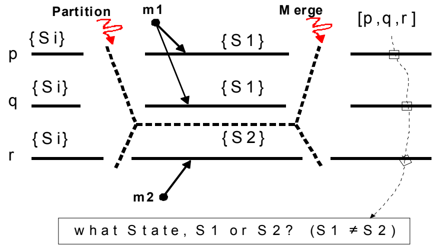
- one should ensure consistent service even when some replicas become mutually unreachable
  - for networks where partitions can occur

Replicated computations

(issues)

- state divergence with partitioning
  - with partitioning, cliques of replicas may mutually think they are dead, and continue computations independently
- one way to solve is to ensure computations only proceed in a primary partition
  - most consistent but also less available way
- another common way is to gather votes or quorums of a minimum number of replicas that guarantee progress with some tradeoff with consistency
  - nothing to do with value masking but with progress

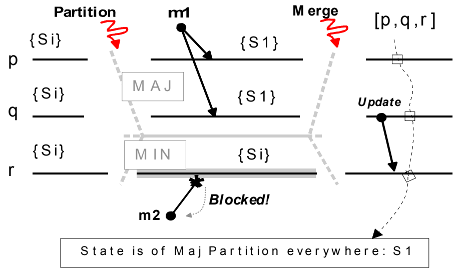
State Divergence with Partitioning



- partitioning occurs, p,q execute cmd m1 and assume state S1
- r executes cmd m2 and assumes state S2
- What is system state after merger?
- e.g., if m1 and m2 produced conflicting results, it is impossible to find a coherent common state without manual *reconciliation* (application dependent)

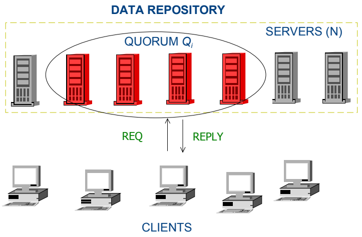
Avoiding State Divergence

primary partition



- a before, but now only *primary partition* continues executing
- PP has majority of replicas, i.e. <p,q>
- <r> stays blocked in state Si
- <p,q> continues, processing m1, and goes to state S1
- after merger, <r> requests state update to set <p,q>
- since Si (of <r>) is a prefix of S1, there is no divergence

Static voting and quorums



- given operation should only be allowed to proceed if a minimum quorum of replicas can perform it
- quorum formation rules:
  - two conflicting operations must always intersect in at least one replica
  - this common replica ensures outcome of first operation is available to all replicas executing the next operation
  - most current state identified by version numbers incremented by each replica upon un update

---

Quorum algorithms

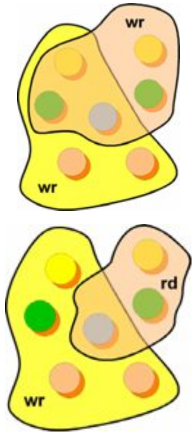
(Weighted voting)

- each copy is assigned a number of votes
- quorums are defined based on the number of votes instead of the number of replicas
- overlapping guarantee rule: 2w > n and w + r > n
- why?
  - n the total number of votes
  - sum of quorums for conflicting operations on an item should exceed the total number of votes for that item (to yield a common replica)

Quorum algorithms

(Weighted voting intuition)

n=7, w=5, r=3



- 2w > n
  - suppose n = 7, w = 5 and r = 3
  - write to partition containing replicas summing at least 5 votes
  - 2 votes left, not enough to write divergently in other partitions
- w + r > n
  - reads and writes to same item, different partitions, are serialized
  - e.g., write occurs first (5 votes), so read must wait (2 votes left, read needs 3 votes)
  - so, read is sure to include at least one of the replicas that have seen previous write
  - this replica can update the others, ensuring sequential consistency of the history of operations

---

Replication management

(recoverable systems)

Rationale

- one should ensure consistent service even when some replicas become mutually unreachable
  - for systems where replicas can crash and later recover

Replicated computations

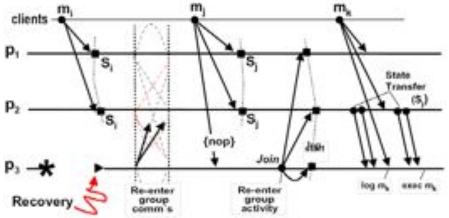
(issues)

- replica failure and recovery
  - when a failed replica recovers, it has an old state
  - how to synchronize with live replicas?

Recovery

- recovery from crashes requires that the recovering replica recovers current state of the other replicas
- without stable storage:
  - cooperative recovery (state transfer) from other replica(s) without stable storage
  - complete state transfer can make recovery be very long
- with stable storage:
  - recover some past state Sx before failure
  - cooperative recovery (command log transfer from Sx) from other replica(s)
  - state recovery is much shorter
  - execute from log of commands until current state

Replica failure and recovery



- recovering replica (p3) starts by resuming communication with the replica set, e.g. if the set was using some form of group comm's
- it starts receiving all messages, but still discards them
- next, sends a request to *join* the replica group activity, delivered in *total order* to all replicas, including the joining replica, marking a cut Sj in the global system state request is which triggers a state-transfer operation
- p2 checkpoints its state at this point (Sj), and sends it to p3
- p3 starts logging any messages that arrive after the cut Sj
- New requests (mk) can continue to be processed by all replicas except p3

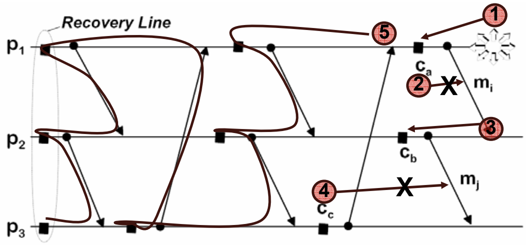
Checkpointing
(checkpoint-based rollback-recovery)

- checkpoint:
  - during normal execution, state is saved at times to prevent log from growing too much or when important actions are done
- rollback:
  - upon recovery after having crashed, the component reads the last checkpoint and resumes operation from there
- consistency
  - checkpoints in a distributed system must lead back to a past consistent global state, called recovery line
- inconsistent global state
  - C1 at p1 and C2 at p2 are mutually inconsistent if C1 contains message m sent by p1 to p2 but C2 has no record of sending m

Checkpointing
(issues)

- uncoordinated checkpoints
  - rollback, if uncoordinated, may bring the computation way back, called domino effect
  - recovery gets immensely slow
- domino effect
  - rollback of one process meets an IGS, forces rollback of another process, which in turn forces first process to rollback again, and so on
- coordinated checkpoints
  - so most schemes are coordinated, having the processes coordinate to meet a CGS before taking the checkpoint
  - thus, after crashes, only need to rollback to last checkpoint

Checkpointing
(domino effect)



1. p1 fails, and then recovers, rolling back to checkpoint Ca
2. evidence of sending message mi no longer exists
3. so, p2 is forced to rollback to checkpoint Cb
4. however, this “unsends" message mj and p3 is forced back to Cc
5. rollback propagation will bring system back to initial state

Resilience

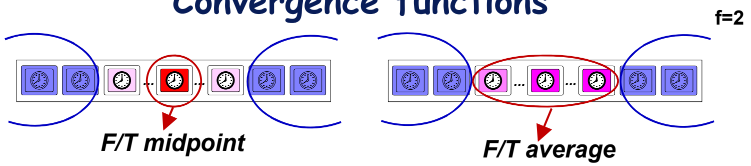
- qualitative aspect:
  - the kind of faults to be tolerated, for example whether or not the system can partition; or whether time- or value-domain faults are assumed.
- quantitative aspect:
  - concerning the number of faults to be tolerated (f)

Detecting/Masking value errors

- tolerating value faults:
  - different sources of the same "logical" value must be available in the system, so that their values can be compared or voted
- voting is simple when values can be compared bitwise
  - vector of values is applied a deterministic function
- exact (bitwise) agreement not always possible
  - when two correct replicas produce different values
  - e.g., vector of values is the result of analog sensor readings
- Inexact Agreement
  - convergence function must be performed on whole vector
  - each run computes a right value, maybe neither of the initial values, maybe different from replica to replica
  - e.g., clock synchronization is another example of inexact agreement

---

Convergence functions

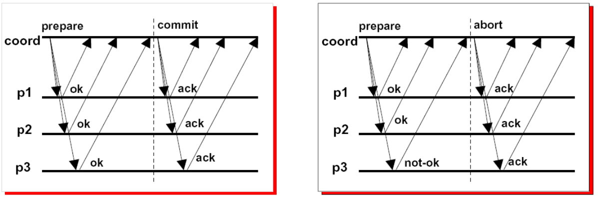


- tolerate up to f faulty values in a vector of n entries
- Fault-tolerant Midpoint
  - selects the midpoint of the values collected after discarding the f highest and f lowest values.
  - Requires at least 2f + 1 values, 3f + 1 with non-masked Byzantine faults
- Fault-tolerant Average
  - selects the average of the values collected after discarding the f highest and f lowest values
  - Requires at least 2f + 1 values, 3f + 1 with non-masked Byzantine faults

---

Atomic Commitment

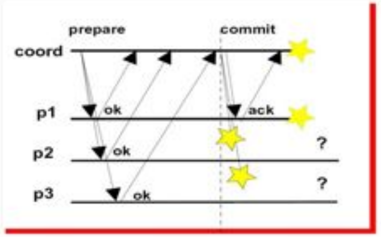
(re-cap)



- Two-phase Atomic Commitment Protocol:
  - (a) commit
  - (b) abort

Atomic Commitment

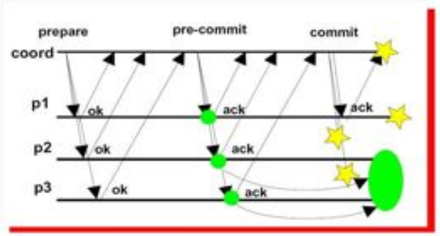
(2-phase commit window of vulnerability)



- Two-phase Commit Protocol blocking scenario:
  - coordinator fails in middle of commit: says commit just to some (p1)
  - p1 committed and then also failed
  - remaining participants will be blocked waiting for the decision
  - they cannot abort: coordinator might have said commit to some (as it did)
  - only when the coordinator recovers can a safe decision be taken
  - these failure scenarios may also take place if the system partitions.

Atomic Commitment

(non-blocking 3-phase commit)



- Three-phase Commit Protocol (non-blocking):
  - idea is to delay the final decision until enough processes “know" which decision is about to be taken
  - coordinator sends a pre-commit message to all processes and waits for an additional round of acknowledgements, only then the commit is sent
  - if coordinator fails before issuing commit, remaining processes may resume the operation since they have received the pre-commit message.
  - 3-phase commit is much more resilient than 2-phase, at the cost of performance