## Failure Detection (7.1)

- **Failure detection key to DFT**
  - Often have to detect to mask/recover/etc
  - Even if can mask, may want to replace for redundancy level
  - Performance: don’t bother trying to communicate with dead
- **Components involved:** target, detector, channel between
  - Adding two other components to the system so can detect!
  - These can fail, too!
- **Failure detectors are imperfect**
  - Try to make them “better” than the targets monitored
  - Still can declare a correct target as failed
    - Q: why?
    - How can work around?
- **Q:** why “failure detection”, not “fault detection” or “error detection”?

## Local Failure Detection

- **Local:** detector and target “close” enough so a “perfect” observing channel exists
- **Examples**
  - **Self-checking** components: SW (sanity check) or HW (parity etc)
    - Example: SR language stack “magic number” set/checked by RTS
  - **Guardian** component: check validity of outputs/actions
    - Memory ranges
    - Kinds/patterns of system calls (Krings et al), ...
  - **Watchdog** components: verify a computation progresses
    - HW: countdown clock, process must reset
    - SW: memory location set, process must reset and watchdog checks
- **Q:** is local detection perfect?

## System Diagnosis

- **Previous model:** target and detectors
- **Generalization:** all components equal
  - Dual role: normal functionality plus checking others
- **System diagnosis:** identifying which system components have failed based on results of component cross-checks
- **Starting assumptions**
  - Correct components report failures correctly
  - Failed components may get it wrong
    - Correct component reported as failed
    - Incorrect component reported as OK
**System Diagnosis (cont.)**

### Representation:
- Directed graph of components
- Arrow $A \rightarrow B$ means $A$ reports $B$ as failed
- Blue/dark node means actually failed

### Figure 7.1(a): Symmetric detection with one failure
- Both components have declared the other failed
- Cannot tell which is right!
- Fundamental bounds: need $\geq 2f+1$ to correctly detect $f$ components

---

**Distributed Failure Detection**

- Harder than local failure detection (why?)

- Assumptions for now
  - Detecting failure of a process
  - Crash failure model
  - Synchronous system (delays bounded)

- Assumed correct if provides evidence of activity
  - Why? Examples?

- Further simplifying assumption: full network connectivity (any-any direct)
  - Abstracts out network connections issues
  - Any process can be both observer and target

- Goal: “consistent” failure detection
  - Possible definitions?

---

**System Diagnosis (cont.)**

- Another organization: **Diagnosis ring**
- 3 nodes can identify a bad failure detector: will have
  - Edge into it from detector marked ‘Failed’
  - Detector node reporting it marked ‘Correct’
  - E.g., ‘B’ above

- Need a centralized component to collect and analyze reports

---

**Distributed Failure Detection (cont.)**

- Properties of failure detectors (Chandra and Toueg)
  - **Strong accuracy**: no correct process ever reported failed
  - **Strong completeness**: a failure is eventually detected by every correct process

- Q: are these safety or liveness conditions?

- A **perfect failure detector** provides strong accuracy and strong completeness
  - With perfect channels, heartbeat message protocol suffices (why?)

- Failure detection with imperfect channels: 2 cases
  1. Imperfection fixable by simple protocols
  2. Imperfect channel not so fixable

- Case 1: transform into a perfect channel
  - E.g., if omissive failures w/degree k, retransmit k+1 times
Imperfect Failure Detection

- Perfect failure detection (FD) convenient, but not always possible
- Problem 1: no bounds on # and type of failures of comm. channel
  - Don’t know if heartbeats lost, and cannot work around
  - Subcase: partitions (failure detect. \(\Rightarrow\) reachability detect.)
- Problem 2: no bounds for timely behavior of system components (processes or channels)
  - Can’t distinguish between missing vs. “slow” heartbeat
  - Ergo, if asynchronous system perfect detectors impossible
- Fallback goal: something between perfect and no FD

Imperfect Failure Detection (cont.)

- Fallback definitions from Chandra and Toueg
  - Weak Accuracy: at least one correct process is never reported failed by all correct processes
  - Weak Completeness: a failure must eventually be detected by at least one correct process
  - Different algorithms provide different combinations of \{weak, strong\} \{accuracy, completeness\}
  - “Weak” properties are not quite useful enough…

Asynchronous Failure Detection

- Asynchronous systems w/failures: impossibility results
  - Perfect failure detectors
  - Coordination: consensus, atomic broadcast, atomic commit
  - FLP results: consensus not possible even with 1 failure
- Issue: what are min. synchrony requirements to solve consensus?

Asynchronous Failure Detection (cont.)

- Chandra and Toueg: consensus can be solved in asynch. system augmented with FDs with
  - Eventual weak accuracy: there is a time after which some correct process is never suspected by any correct process
  - Weak completeness: a failure must be eventually detected by at least one correct process
  - These called “eventually weak” FDs
- Note: weak accuracy only required to be satisfied at some time
  - Idea: period of stability allows process to coordinate
  - One non-suspected process can be the coordinator
- Applicability of eventually weak FDs?
  - Oops, can’t be implemented in a pure asynch. system
  - But many algorithms using them assume very little so safety not violated (but progress not made)
Partitioning

- Failures can split network into disjoint partitions
  - Huge problem: partitions cannot coordinate
  - Two approaches for solutions...
- Soln1: Allow uncoordinated progress in different partitions
  - Have to reconcile state of partitions when healed
  - Cannot do automatically
- Soln2: Allow progress in one partition exclusively
  - “Primary partition approach” from Chapter 2
  - Not always possible to have a primary

Q: which approach is “best” for which kinds of apps, and why?

Fault-Tolerant Consensus (7.2)

- Recall consensus:
  - Each process proposes an initial value
  - All correct processes must agree on same value from proposed
- Consensus is a key (conceptual) building block:
  - Membership (agree on set or processes in system)
  - Ordering messages (agree on sequence # for a message)
  - Atomic commitment (agree “yes” or “no” to commit trans.)
  - System diagnosis (agree which processes are faulty)
- Fault-free solution
  - Process with lowest ID is coordinator
  - Others send values to coordinator, who chooses one
- Extremely difficult to extend this to handle failures

Fault-Tolerant Consensus (cont.)

- Extending the simple coordinator model
  - What happens if coordinator fails?
  - Possible fix: when coordinator failure detected, next-lowest ID process takes over
  - Works???

Fault-Tolerant Consensus (cont.)

- Oops: first coordinator crashed after some but not all got the value it decided on
  - Some processes could use the first’s value, even though second could decide differently
- Problem: have to solve consensus so a process does not decide on a value until its guaranteed to be the only decision
  - Value is locked then, even if not every process has decided
- Locking a value
  - When a process receives initial value from coord, changes its initial val to coord’s
  - If later becomes coord, it proposes this value
Fault-Tolerant Consensus (cont.)

- Improved protocol
  - Coord. sends value to every other process
  - Processes do not immediately decide; update their initial value
  - When coord. gets ACK from every process, it knows value locked
    - Even if it crashes, new coordinator will propose that value
  - Coord sends DECIDED(val) message to every processes

Q: is this a “good” or “practical” algorithm?

Uniformity (7.3)

- FT Consensus problem has to categories
  - **Uniform Consensus**: if two processes decide, they decide on the same value
  - **Non-uniform consensus**: if two correct processes decide, they decide on the same value
    - Allows a process to decide on a different value from one that crashed

Non-uniformity Example

- Scenario
  - p sends decision m to q, but r & s do not get
  - p and q crash or are partitioned from r and s
  - r and s decide on k
  - (if partitioned) p and q heal
- Why not always assume stronger (uniform) consensus?
Non-uniformity (cont.)

- Weaker assumptions can be more efficient!
- Example protocol (relies on perfect FDs)
  - Assume total order on process IDs
  - Coord. sends val. to each process
  - Upon receipt each process decides immediately
- Note: If coord. doesn’t fail all processes eventually decide same val.
- Coord fails: next process becomes coord.
  - New coord. asks other correct processes if have decided
  - If any have decided, new coord. forward that val on
  - If none have decided, coord. decides and disseminates its own initial value
- Comparison
  - Non-uniform (above): process decides once coord. tells it
  - Uniform: proposal has to be ACK’d by at least a majority to decide
  - Ergo, if crashed processes cannot harm a system, non-uniform better

Membership (7.4)

- Process group: a set of cooperating processes
- Membership (of a group): set of processes belonging to the group at a given point in time
- Membership service: keeps track of the group members, provides info via a group view, the subset of members mutually reachable
  - Very dynamic: processes fail, restart, join, leave, ...

Group Membership

- Group membership is a form of dist. agreement
  - Not as simple as it may seem at first!
  - Agree on set of processes in the system
- Hard problem, multiple flavors of definition even
  - Informally: consistent membership: if
    - group remains unchanged
    - no link failures
    then all members receive the same group view
- Major requirement: must remove from group view processes that have failed
  - Has to be accurate, not just consistent
  - Oops, accurate FD very hard.....
  - Q: what should a membership service do?? Alternatives??
- Scenario: member P is suspected of having failed
- Choice 1: leave P in group
  - Oops, application trusts P to work properly
  - Might send subtask to do, etc.
- Choice 2: remove P from group
  - Oops, what if it really is still functioning?
  - Can’t do useful work until re-synch with group (state transfer or update)
- Order of group view delivery is important
  - Often needs to be consistently delivered w.r.t. app msgs
  - E.g., multicast of parallel search/reduction on part of data
  - Each process does 1/Nth if N processes in view
**Linear Membership**

- **Linear membership service**: enforces total order on all views
  - I.e., all correct processes receive exact same sequence of views

- **Implementability**
  - Synchronous system w/o partitions: easy agreement
  - Partitions or Asynch.: much harder (minority partitions must not deliver views)

**Partial Membership**

- Non-primary partitions must block or crash procs
- Alternative: **partial membership service**
  - No longer totally ordered across all live members
  - Many flavors (few practical)
  - **Strong partial membership**: concurrent views never overlap
    - Supports virtual synchrony paradigm

**Fault-Tolerant Communication (7.5)**

- FT-Comm delivers messages despite
  - Failure(s) of comm link(s)
  - Failure of some participating processes

- **Main kinds of failures to tolerate**
  - Timing (link and process)
  - Omission (link and process)
  - Value (usually only link)

- **If bad guys are an issue, also**
  - Value failures (process)
  - Spurious message generation (not sent by a process)

**Reliable Delivery**

- Start easy: omission failure tolerance (degree k).
- Design choices:
  a) Error masking (spatial): several (>k) links
  b) Error masking (temporal): repeat K+1 times
  c) Error recovery: detect error and recover
**Reliable Delivery (cont.)**

- Error detection and recovery: ACKS and timeouts
- Positive ACK: sent when a message is received
  - Timeout on sender without ACK: sender retransmits
- Negative ACK: sent when a message loss detected
  - Needs sequence #s or time-based reception semantics
- Tradeoffs
  - Positive ACKs faster failure detection usually
  - NACKs require above, not always great
- Q: what kind of situations are good for
  - Spatial error masking
  - Temporal error masking
  - Error detection and recovery with positive ACKs
  - Error detection and recovery with NACKs

**Resilience to Sender Failure**

- Multicast FT-Comm harder than point-to-point
  - P2P problem boils down to failure detection
  - Subsets of receivers may receive msg, then sender fails
- Solutions depend on flavor of multicast reliability
  a) Unreliable: no effort to overcome link failures
  b) Best-effort: some steps taken to overcome link failures
  c) Reliable: participants coordinate to ensure that all or none of correct recipients get it (notice sender failed in b)

**Achieving Reliable Multicast**

- Mainly via error masking or error recovery
- Error masking approach
  - All recipients retransmit message to all others when received
- Error recovery approach
  - Recipients buffer a copy of message for a while (till safe to discard)
- Analysis
  - Error masking more suited when accurate failure detection impossible (async. system)
    - But have to retransmit infinite # of times, in theory (not in practice)
  - Error recovery assumes that failures can be detected
- Which is “better” for what kinds of apps/systems/situations?

**Tolerating Value Faults**

- Link value failures: garbled in transit
  - Normally checksums used
  - Garbled received: discard (translate value fault into omission fault, handle as before)
- Value faults by faulty sender
  - Checksum can not help!
  - Can only be tolerated by spatial redundancy
- Technique: obtain different sources (replicas) of the same logical value, compare values
- Multiple receivers? Consistency requires comparison same
  - i.e., all correct recipients must agree on outcome of comparison
  - Requirement: use same deterministic selection algorithm
  - Often need same order of messages (not always), at least with failures
Tolerating Arbitrary Faults

- Very hard to tolerate
  - Faulty sender may exhibit two-faced behavior
  - Link (or faulty sender) may generate spontaneous message that is syntactically correct (impersonating a legitimate sender)

- Byzantine agreement: name for the problem of reaching agreement in the presence of arbitrary faults
  - Number of encamped generals must decide: attack/retreat
  - Most loyal but some faulty
  - All loyal generals must decide to attack or defeat possible

- Hard problem
  - Loyal generals must agree on a binary value despite traitors that will try to thwart correct agreement

Tolerating Arbitrary Faults (cont.)

- Assume for now: synchronous system, agreement protocol operates in rounds
  - Each round generals send to other generals
  - Traitors may omit or send conflicting messages

- Simple majority vote sufficient?

Tolerating Value Faults (cont.)

- Partial view of Byzantine agreement (correct sender)
  a) First round
  b) Second round

Note: recursive solution works: one less degree of freedom (doubt over one general’s value) each round

(Leaving Section 7.55 and beyond for next slide set)
### Byzantine Agreement Redux (TvS 7.2.3)

- Complicated stuff, explanation series #2:
- Recall if
  - Independent failures (no collusion/conspiracies)
  - No two-faced behavior
- Then can
  - Vote on $2k+1$ replies/values
  - Tolerate $k$ bad values
- Harder if either condition not true
- Generic goal:
  - Have all non-faulty processes reach consensus
  - Do it in a finite number of steps

### Baseline Problem

- Easier base case: “two-army problem”
  - Red army in valley
  - 2 Blue armies camped on hillsides (Blue1 and Blue2)
  - Blue1 and Blue2 must agree on one bit (attack/retreat)
  - Complication: unreliable message delivery: courier can be captured
    - I.e., omission failure of link
- Bottom line
  - After any step, Blue1 and Blue2 cannot be sure the other got the last message
  - Ergo, if that step was needed in the protocol, one cannot finalize the agreement
  - Intuitive inductive argument shows agreement can never be reached.

### Harder Problem

- Assumptions
  - Communication is perfect
  - Processes (generals) are not perfect
  - “Byzantine Generals problem”
- Definition
  - Red army is still encamped in valley
  - \{Blue(1), ..., Blue(N)\} armies camped in hills
  - Communication is point-point and perfect
    - Maybe telephone, or radio with no jamming
  - M of N generals are traitors (faulty) and try to prevent the (N-M) loyal (correct) generals from correct agreement
  - Problem: algorithm where correct generals reach agreement

### Harder Problem (cont)

- Generalize the problem
  - Each general knows his/her troop strength
- Goal: exchange troop strengths so when done
  - Each general has vector of N troop strengths
  - General(i) loyal ➔ all loyal generals have correct #Blue(i)
  - General(i) traitor: undefined for loyal and traitor generals
- Recursive algorithm from Lamport et al 1982
Harder Problem (cont.)

Example where $N=4$ and $M=1$, takes 4 steps:
1. Every general sends a (reliable) message to every other general with its troop strength (a)
2. Each general sends the others vector received in #1 (b)
3. Each general sends others vectors received in #2 (c)
4. Generals vote; correct ones decide (1,2,UNKNOWN,4)

Steps again
3. Each general sends others vectors received in #2 (c)
4. Generals vote; correct ones decide (1,2,UNKNOWN,4)

Algorithm generalizes to more than $N=4$, $M=1$ recursively
- Will cover when we go through the paper(s), Lamport et al 1982…

Harder Problem (cont.)

Consider case $N=3$, $M=1$ & same algorithm:
• Intuition: need to achieve a majority vote among loyal generals
• Need to ensure that
  – Vote with $M$ traitors,
  – And any loyalists misled (temporarily confused) by traitors
  – Still adds up to the majority vote of the loyalists
• Can only ensure this when >2/3 of votes same
• I.e., if >2/3 of generals agree on same decision, must be the same majority vote by the loyal generals
• $M=3$, $N=(3M+1)=10$ …. 
  – 3 traitors, detected as such
  – 3 loyal but misled generals
  – Leaves 4 to outvote the 3 misled loyal generals
FIFO Order

- **FIFO Order**: any two messages sent by the same participant, and delivered to any participant, are delivered in the same order
  - Note this ordering is per-receiver
- **Implementation**
  - Sender timestamps messages with a local sequence num.
  - Receiver delivers in order of sequence num., not arrival
  - Q: what does receiver have to do to deliver this way?
- **FIFO Example**
  - Paul is at site r working hard...
  - Solves first phase of a problem by executing modules m1, m2, m3 in sequence
  - Sends intermediate results to Mary at site s and John at site q

Causal Order

- Problem in Fig 2.14b: FIFO cannot be used if competing senders to a site also exchange messages among themselves! Why?
- **Causal ordering** uses potential causality across sites
  - Ensures messages obey causal delivery or causal order
- **Causal delivery**:
  - for any two messages $m_1$ sent by $p$ and $m_2$ sent by $q$ to the same destination $r$,
  - if $sendp(m_1) \rightarrow sendq(m_2)$
  - then $deliverr(m_1) \rightarrow deliverr(m_2)$
  - i.e., $m_1$ is delivered to $r$ before $m_2$
- **Implementation**: track precedences (happened-before) in message headers, in various ways
  - works only if message exchange is only communication
Causal Order (cont.)

- Uses of Causal Order
  - Most apps don’t have peers interacting; FIFO works
  - Causal for peer interactions: teleconf, interactive multimedia

- Causal order not always sufficient
  - Say Paul accumulates result in variable W
  - All Paul’s helpers work in parallel and also accumulate W
  - Compare W to each updated result message to update W (max+3)

Total Order

- Notice causal order lets concurrent events happen without ordering them
  - Usually nice: allows parallel computations to proceed without unnecessary constraints
  - Sometimes need to totally order events

- Total Order: any two messages delivered to any set of participants are delivered in the same order to both participants
  - Note: does not have to be causal (why); almost always is

- Uses for total ordering
  - Achieving determinism of replicated process executions
    - “State machine approach” most common example
  - Ensuring different participants get same perception of system evolution and its state
    - “Common knowledge”

Total Order Example

- Solution to last problem (causal limitation)

Temporal Order

- Logical order (total, causal, ...) assumes participants only communicate via the ordering protocol
  - Could interact with another protocol
  - Could interact via the outside world

- Problem: hidden channels or covert channels allow other ways for participants to create causality chains
  - Ordering protocol has no way of knowing about these!

- Examples
  - Using another protocol
  - Hidden channels through feedback in environment (process control)
Causal Ordering Algorithms

- One possibility: make message carry its causal history, past(msg)
  - Message sends past history in message header
  - After sending \( m \), sender adds \( m \) to its past list
  - When receiving \( m \), checks its past(\( m \)), and delivers all messages in past(\( m \)) first, then delivers \( m \)

- Comments
  - It works!
  - Advantages?
  - Disadvantages?

Causal Ordering Algorithms (cont.)

- Previous (naïve) algorithm overkill, messages rarely dropped
  - Sending entire message is overkill

- Refinement: include only message IDs in past(\( m \))
  - (Note: for now assuming some 3rd party handles retransmitting a lost message.)
  - Sender: add IDs of causal history to past(\( m \))
  - Sender: add \( m \) to past list
  - Receiver: check past(\( m \)), if contains messages not yet received, queue up \( m \) for later delivery
  - Receiver: after all messages in past(\( m \)) have been delivered, then deliver \( m \), and add \( m \) to past list

Causal Ordering Algorithms (cont.)

- Discussion of refinement
  - Have to wait until all “past” messages arrive
  - Helps reduce control field size (how much?)
  - But how to remove obsolete information from the past?

- Bigger issues
  - Even with all obsolete info removed, what is worst case header info (control data structure) needed without forcing additional synchronization on the system?
  - In general case, with \( N \) processes, you need \( N^2 \) size
    - i.e., at least one message ID kept for each pair of communicating processes (members if group)

Causal Ordering Algorithms (cont.)

- Example coding and storing causal info
  - system with \( N \) processes \( p_1 \) to \( p_N \)
  - Each msg identified by sender ID and local sequence #
    - A kind of local clock, counting send events
  - Each process keeps the seqn of last msg sent to each proc
    - E.g., SENT\(_i\)=\([0,2,4,2]\), \( p_1 \) last sent to \( p_3 \): msg ID \((p_1,4)\)

- We need more!
  - Causal past of process consists of its messages sent, plus causal past of messages delivered
  - Ergo, process has to log its knowledge of messages other processes have sent
    - i.e., send
      - Own SENT array
      - Approximation of other process’s SENT arrays
    - Called a matrix clock often
      - MATRIX\(_i\)[i,j] keeps seqn of last msg \( p_i \) sent to \( p_j \) known by \( p_k \)
Causal Ordering Algorithms (cont.)

- Still $n^2$ is expensive!
  - Optimization: If all msgs sent are multicasts, all elements of row are same so matrix becomes a vector, a vector clock

Logical Clocks

- Can have smaller header size if don’t need exact causality
- Logical Clock, a monotonically increasing counter.
- Let
  - Each process $p$ keeps its own logical clock, $C_p$, which it uses to timestamp events
  - $C_p(a)$ is the logical time at process $p$ at which event $a$ occurred
  - $C(a)$ is the logical time at which event $a$ occurred at the process it occurred at
- Processes keep their own logical clocks, initialized to 0.
  Updated by rules:
  - LC1: Before each event occurs, increment $C_p$
  - LC2: When a process $p$ sends a message $m$, it piggybacks on $m$ value $t = C_p$
    When process $q$ receives $<m,t>$, $q$ computes $C_q = \max(C_q, t) + 1$ then timestamps $m$

Logical Clock Example

- Note if $a \rightarrow b$ then $LC(a) < LC(b)$
- However, $LC(a) < LC(b)$ does not imply $a \rightarrow b$
  - Above, $C(e) < C(b)$ yet $b \parallel e$
  - Also note that concurrency is not transitive: $a \parallel e$ and $e \parallel b$ yet $a \rightarrow b$

Total Order Algorithms

- Requirement: deliver all msgs to all recipients in same order
- One way: extend logical clock algorithm
  - Still deliver in order of timestamps
  - But only deliver when all lower timestamps delivered
  - Break ties (same value for logical clock) by host ID
  - Works!
- A kind of symmetric algorithm: all processes execute the same steps
  - Simple to build, only need one mech. (lclock) for both causal and total
  - Fairly low overhead
  - Downsides?
- If have synchronized clocks (GPS), can use instead of lclock:
  - only have to wait delta (max latency), not wait on slowest sender
**Total Order Algorithms (cont.)**

- Another approach: one special proc to give msg Ids
  - Process is **sequencer**, AKA **token site**
  - All senders send to sequencer
  - Sequencer assigns unique ID (local counter) to msg
  - Sequencer sends msg to group
  - Total order is the order msgs received by sequencer
  - Two implementation versions shown below

**Pros and cons of sequencer approach(es)?**

---

**Discussion of Sequencer**

- Sequencer works best when token site sends a lot
- Improvement: rotate token site among senders, ones sending most/lots
- Time warp, now back to Chapter 7…

---

**Implementing Causal Order (7.5.5)**

- Failures can cause problems
  - Previous algorithms assumed a 3rd party to store/retransmit, has to be done in real life in protocol
- Another problem: causal holes and contamination
  - Example below: m2 depends on m1 (pt2pt p-p), but m1 lost
  - Can r deliver m2? By local criteria only, yes
  - But now r is **contaminated**: its state depends on a message that cannot be delivered
  - Creates a **causal hole** in its history
  - E.g., m3 precedes m1, but q will never get m1 sent to it

**Avoiding contamination:** before delivering msg assure that
- all preceeding msgs already delivered, or
- enough copies exist of preceeding msgs

---

**Implementing Total Order**

- Need some kind of agreement between processes
- Has been shown that uniform consensus and uniform atomic broadcast (total+atomic) equivalent
  - I.e., can build one from the other
- Failures complicate centralized sequencer protocol
  - Electing a new leader is easy
  - Ordering defined by order sequencer decided on
  - Issue: what did it decide before it crashed, and who knows?
  - Has to find out what what was partially broadcast…
Implementing Total Order (cont.)

- Failure example
  - m1 and m4 delivered to q and r, but not to p
  - New sequencer has to retransmit m1 and m4 to p, but in right order
  - Q: how can the new sequencer do this?
  - Q: can it achieve uniform total order?

Fault Tolerance of previous algorithm (Chap 2.7.6)…
- Token-site total order survives F failures by rotating token and copying key ordering info F+1 times before message considered stable

Note: protocols providing nonuniform total order can cause contamination of system (ouch!)
- Sequencer receives m1 and m2 and multicasts in this order
- Sequencer later delivers m3, whose contents depend on m1 and m2
- Sequencer crashes before sending ordering info (in separate message) to group members
- New sequencer can decide to deliver m2 < m1 < m3
- Uniform protocol does not allow, but at greater cost

Partition-Free Replication Management (7.6)

- Goal: provide availability via spatial redundancy in form of replication (assume for now no partitions)
- State machine: one way, using deterministic components
- Output (and internal state)
  - Are completely determined by initial state and sequence of requests
  - Do not depend on time or other activity in the system

Note: often useful to distinguish between
- Write commands: modify state
- Read commands: do not modify state

State Machines (cont.)

- Other possibility: state and behavior (output) of non-deterministic component depend also on local parameters that cannot be controlled
- Lots of mechanisms can cause this
  - Some nondeterministic language constructs
  - Scheduling decisions
  - Local clocks or sensors or random number generators
  - States of two non-deterministic replicas will likely diverge even if given the same input sequence!

- Intermediate model: piecewise deterministic components
  - Execute deterministically for a while, then non-deterministically does some things
**Active Replication**
- Active replication can be applied to state machine components
  - Have several replicas of a state machine
  - Use atomic+total multicast to disseminate commands
  - All replicas thus produce same output (if not failed)
  - Output is consolidated into one (choices?)

**Note:** consolidation like Voting VM

**Pros and cons of Active?**

**How can read-only commands be ordered?**

**Semi-Active Replication (Leader-Follower)**
- Aiming for realtime support
- Can be used with piecewise-deterministic components
- Technique
  - All replicas execute the commands (but in pieces)
  - Leader decides non-deterministic decisions in code (tells followers)
  - Leader also decides message ordering (tells followers)
  - Leader also tells followers when to preemption points (so catch up)

**Discussion**
- Can always guarantee consistency?
  - (BBN Planet replication+MS anecdote)
  - Tradeoffs between checkpoint size, log size, checkpoint frequency, recovery glitch?

**Passive Replication (primary/backup)**
- Only primary executes commands
- Backups are idle (but log commands)
- Periodically primary replica updates state to backups, they trim logs
- Primary fails: backup takes over
  - Starts from last checkpoint (state update)
  - Replays logged messages since then

**Replication Management with Partitions**
- **Voting:** pessimistic concurrency control mechanism that ensures conflicting operations cannot occur
  - Technique: operation proceeds only if a minimum quorum of replicas can perform it (they vote “yes” to do it)
  - Define quorums so conflicting operations intersect in at least one replica
  - This common replica can make output of previous operation available to new operation
  - Keep version number of state to help track this

**Notes**
- Other use of “voting”: output consolidation (ala VVM)
- I tend to call this “synchronization”
- First part of voting is “static voting”: quorums do not change
- Later “dynamic voting” where they do
Static voting

- Simple scheme (read-one write-all)
  - Read operations can be done on any replica (quorum=1)
  - Write operations must be done on all replicas (quorum=N)
  - Read operations: high availability and low cost
  - Write operations: high cost and block if one failure
  - Q: preserves causality for a given application?

Weighted Voting

- Extend above scheme:
  - Each replica gets a number of votes
  - Quorums based on # votes not # replicas
  - Sum of quorums of conflicting ops must exceed total votes

  Quorum definitions
  - N votes in the system
  - R votes is the read quorum
  - W votes is the write quorum

  Quorum requirements
  - 2W > N
  - W + R > N
  - Why?
  - Examples?

Weighted Voting (cont)

- Example
  - N=7, R=4, W=4
  - Partition happens: now minority partition cannot write
  - Note: reads and writes can never overlap (why)

- Keeping # replicas and # votes distinct key
  - Can assign votes different ways for different tradeoffs
  - What ones?

Coteries

- Alternate to describing quorums: use explicit set of processes, quorum groups
  - Collection of quorum groups for an op is quorum set
  - To ensure overlap, each group in a quorum set must overlap with every other group in that set
  - Such a quorum set also called a coterie
  - Can be used to achieve mutual exclusion
  - Some quorum sets exist that cannot be defined by voting algorithms

- Problem: defining quorum sets with specific properties/tradeoffs is hard!

- Workaround: find some (or impose artificial) structure in the processes
Structural Representations of Quorum Sets

- Typical way: use well-understood structures as a paradigm
- **Tree quorum** algorithms
  - Main idea: select path from root to any leaf
- Grid algorithm
  - Read quorum must contain a node from each column
  - Write quorum must contain a node from each row
  - Have to overlap

Byzantine Quorum Systems

- Previous quorum systems assume benign faults
  - Ergo, intersection of quorums of conflicting ops $\geq 1$
- Can tolerate more types of faults by having larger intersections
  - E.g., ensure quorums contain a majority of correct processes (Malkhi and Reiter, 1998)
  - E.g. read operation can accept value returned by $F+1$ servers and ignore other values from replicas

Dynamic Voting

- Previous algorithms choose quorum sets (or quorums) at startup time
- Dynamic voting: quorum groups can be changed at runtime
  - Use info about system configuration etc. to choose best tradeoffs
  - Examples of tradeoffs and conditions to base them on?

Review: Replication Strategies & Features/Costs

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Error Processing Overhead (Failover time)</th>
<th>Replica Non-Determinism</th>
<th>Component Model</th>
<th>Byzantine Behavior</th>
<th>Faults Tolerated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>Lowest</td>
<td>Forbidden</td>
<td>Deterministic</td>
<td>Tolerated</td>
<td>Crash, Omission, Value</td>
</tr>
<tr>
<td>Passive</td>
<td>Highest</td>
<td>Allowed</td>
<td>Non-deterministic</td>
<td>Forbidden</td>
<td>Crash, Omission</td>
</tr>
<tr>
<td>Semi-Active</td>
<td>Low</td>
<td>Resolved</td>
<td>Piecewise Deterministic</td>
<td>Forbidden</td>
<td>Crash, Omission</td>
</tr>
</tbody>
</table>
**AQuA Handlers**

- **AQuA project at BBN**
  - BBN (Bakken, Schantz), UIUC (Sanders, Cukier), Cornell (Birman)
  - Quality Objects (QuO) used
  - AQuA provided fault tolerance
  - Active replication, tolerate non-malicious value faults
  - Clients can be replicated
  - (Some references on my web, others [www.crhc.uiuc.edu/PERFORM](http://www.crhc.uiuc.edu/PERFORM) (Bill Sanders site))
- **Lesson #1:** very wide range of impl. choices
- **Lesson #2:** bitwise voting does not work
- **Note:** *most details here not directly testable*, but can provide very good insights into practical issues
- Caveat: some slides densely packed…

**AQuA Handlers: Design Space has Many Variables!**

- **Client group has leader or has no leader**
  - how much do you trust client group?
- **Server group has leader or has no leader**
- **Multicast strengths (total, causal, FIFO, …) used in connection group**
- **Which members of client and server groups are in connection group**
- **Location and algorithm for voting**
- **How many rounds of multicasts (e.g., for byzantine)**
- **Location of buffering of requests/replies**
  - Caveat: not shown in following diagrams
- **Also:** interaction with handler “upstream” or “downstream” in a nested call
  - A → B → C: handlers A → B and B → C need to be managed together, for reasons of performance and possibly correctness

**AQuA Scheme1 Request Steps**

1. **(Leader)**
   - C-Rep1
   - C-Rep2
   - C-RepN

2. GW
   - GW
   - GW
   - GW

3. GW
   - GW
   - GW
   - GW

4. GW
   - GW
   - GW
   - GW

5. GW
   - GW
   - GW
   - GW

6. GW
   - GW
   - GW
   - GW

7. GW
   - GW
   - GW
   - GW

8. GW
   - GW
   - GW
   - GW

9. GW
   - GW
   - GW
   - GW

10. GW
    - GW
    - GW
    - GW

11. GW
    - GW
    - GW
    - GW

12. GW
    - GW
    - GW
    - GW

**GWs in Client Group**

**GWs in Connection Group**

**GWs in Server Group**

**AQuA Scheme1 Reply Steps**

1. **(Leader)**
   - C-Rep1
   - C-Rep2
   - C-RepN

2. GW
   - GW
   - GW
   - GW

3. GW
   - GW
   - GW
   - GW

4. GW
   - GW
   - GW
   - GW

5. GW
   - GW
   - GW
   - GW

6. GW
   - GW
   - GW
   - GW

7. GW
   - GW
   - GW
   - GW

8. GW
   - GW
   - GW
   - GW

9. GW
   - GW
   - GW
   - GW

10. GW
    - GW
    - GW
    - GW

11. GW
    - GW
    - GW
    - GW

12. GW
    - GW
    - GW
    - GW

**GWs in Client Group**

**GWs in Connection Group**

**GWs in Server Group**
### Scheme 1

**Arch. (a tad obsolete)**

**Scheme 1 Steps (a tad obsolete)**

1. **Sender (“client”)** Side
   - GW_Scheme1_Handler::SendRequest() does
     - S-IIOPGW enqueues msg
     - GW_ReqIDSet stores ReqID

2. **Receiver (“Server”)** Side
   - GW_Sequencer::PrepareReply():
     - GW_Sequencer stores ReqID with this msg seq#

Sender (“client”) Side

- **GW_Sequencer**::PrepareReply():
  - Stores ReqID with this msg seq#

Receiver (“Server”) Side

- **GW_Sequencer**::PrepareReply():
  - Stores ReqID with this msg seq#

- **GW_WrapperSet**::PrepareReply():
  - Stores ReqID with this msg seq#

- **GW_Wrapper**::PrepareReply():
  - Stores ReqID with this msg seq#

**When to Compare Results**

- Tolerating value faults requires spatial redundancy
  - Different sources of same “logical” value
  - Only handled by active replication
    - Occasionally “reasonableness tests” on server’s output can work
  - Need to compare or “vote” on the different values
    - Warning: “vote” used in two different ways
      - “Synchronization voting”: a pessimistic concurrency control technique
      - “Collation voting”: combining multiple logical values to obtain 1 value
    - Collation voting very simple when values comparable bitwise
      - No heterogeneity or floating point (oops, quite common…)

**Resilience (7.8)**

- Resilience degree provided depends on at least:
  - Qualitative Parameters
    - Kind of faults to tolerate
    - Assuming partitions
  - Quantitative parameters
    - Number of faults to tolerate

- Questions to answer
  - When (and why) to compare results?
  - Agreement: is exact possible or necessary?
  - How many replicas or spares to maintain?
**Exact and Inexact Agreement**

- When bitwise comparison works, one can have **exact agreement**:
  - Exact same value sent by replicated servers
  - Exact same value chosen by replicated clients
    - Very hard to do with two-faced behavior!
- **Cannot have exact agreement in some cases**
  - Heterogeneity and floating point (c.f. Voting VM paper)
  - Apps where two correct replicas can return different values
    - Floating point and heterogeneity
    - Replicas not deterministic
    - Replicas not precisely identical: pair of “close” analog sensors
- **If cannot have same exact replies**
  - Take the vector of replies
  - Apply a **convergence function** to choose or create one
  - Called **inexact agreement**

**Convergence Functions**

- Example convergence functions
  - Tolerating up to F faulty values
    - **Fault-Tolerant Midpoint**
      - Discard highest F and lowest F values
      - Take the midpoint of values remaining (arithmetic mean of \{lo,hi\})
      - Optimization I suggested: discard the F values furthest from the median value (VVM has *furthest* “exclusion” primitive)
    - **Fault-Tolerant Average**
      - Discard highest F and lowest F values
      - Take the arithmetic mean of values remaining
    - Both take 2F+1 values, and 3F+1 values for Byzantine

**Convergence Functions and App-level QoS**

- Convergence functions (and voting algorithms in general) provide a tradeoff of
  - precision
  - fault tolerance
  - performance
for a given operational point in runtime conditions
  - network latency and bandwidth
  - failure rates
  - available CPU cycles
  - etc
- E.g., DSN-2001 Fast Abstract by Rupa, Doug Blough, Bakken (separate slides)

**Presenting Research Papers**

- **Presentation Notes**
  - 2-3 minutes/slide
  - Less for simple context bullets, more for detailed drawings
- **Typical Outline of 25 minute talk**
  - Introduction: context of work (2-3 slides)
  - Summary of contributions (1 slide), much like conclusions
  - Main body: 10-15 slides, maybe in 2-3 topics/sections
  - Related work: 1-2 slides
  - Conclusions
- **Goals of a conference talk:**
  - Motivate them to read your conference paper
- **Non-goal of a conference talk:** help them to understand all the details
- **(Now we do the Voting VM DSN-2001 talk here)**
Recovery (7.9)

- Techniques so far: mainly ensuring availability of a correct result
  - Can be quite costly!
  - Sometimes it suffices to just restart in a consistent state:
    recovery
  - Need to do this anyway when ensuring availability!

- Key building block: stable storage
  - Persistent: survives the failure of the entity that create/initialized/used it
  - Reliable: very low probability of losing or corrupting info

- Implementation overview
  - Typically non-volatile media (disks)
  - Can sometimes use replicated volatile memories
  - Make sure one replica at least always survives!

Stable Storage

- Building from volatile memory
  - Simple
  - Good latency compared to disk
  - Not generally useful: worse coverage of
    - Loss of power
    - Other common mode failures (correlated)

- Building a stable store
  - Single disk?
  - Value redundancy: checksums
  - Spatial redundancy: multiple disks
  - Redundant Array of Inexpensive Disks (RAID)

Checkpointing

- Main use of stable store: save application and system state (a checkpoint)
  - After failure, recovery moves system back to last checkpoint; a rollback
  - Technique called checkpoint-based rollback-recovery

- Limitation: components do not operate in isolation!
  - Q: how could this affect things?

Checkpointing (cont.)

- Rollback problem: all messages sent since last checkpoint are lost
- Solution: travel back in time!
  - Make all components periodically checkpoint state
  - Roll back to a consistent global state including the recovering process
  - recovery line: earliest such consistent global set of checkpoints

- Huge amount of literature on this field
  - Good topic area for 1-2 literature surveys!
**Approaches for Checkpointing**

- **Coordinated checkpointing**: have processes coordinate before taking a checkpoint
  - Pro: always taking globally consistent checkpoints
  - Con: introduces delays
- **Uncoordinated checkpointing**: processes take checkpoints independently
  - Pro: no delays
  - Con: no guarantees of a consistent set
    - Can roll system back to initial state! *Domino effect* (Fig. 7.20…)
- **Communication-induced checkpointing**: checkpoint when receiving and prior to processing messages that may introduce conflicts
- Note: if replication to stable store, do not have to wait for failed component to recover (ala passive replication)

**Domino Affect Example**

- P₁ fails, recovers, rolls back to Cₐ
- P₂ finds it received message (mᵢ) never sent, rollback to Cₐ
- P₃ finds it received message (mⱼ) never sent, roll back to Cₐ
- …

**Logging**

- Many apps can live with periodic checkpoints so rollback delay not too large…. Limitations?
  - Problems
    - Computation not fully deterministic
    - Some actions since last checkpoint may have left a trace outside the system: *real actions* that cannot be undone
    - Checkpointing takes time!
  - Opportunity for piecewise-deterministic systems
    - Logging non-deterministic events between consec. chkpts
    - Can minimize number of required checkpoints
    - Recovery: reconstruct state of failred component from most recent checkpoint and log
    - May allow recovery without forcing other rollbacks

**Logging Approaches**

- **Pessimistic logging**: ensure info about non-det. event logged before it affects computation
  - Pro: guaranteed to execute same events
  - Con: lots of log operations
- **Optimistic logging**: log these events asynch.
  - Computation proceeds, overlapping with logging
  - Faults infrequent so most logging operations succeed
  - Occasionally a failure can cause inconsistency
- **Causal logging**: track causal relations between events
  - keeps most benefits of optimistic logging without its (over-) optimistic assumptions
Atomic Commitment and Window of Vulnerability

- So far, recovery of actions that can be individually rolled back,…
- Better idea:
  - Encapsulate actions in sequences that cannot be undone individually
  - Have the system guarantee this
  - **Atomic transactions** provide this
- Properties: ACID
  - Atomicity: transaction is an indivisible unit of work
  - Consistency: transaction leaves system in correct state or aborts
  - Isolation: transactions’ behavior not affected by other concurrent transactions
  - Durability: transaction’s effects are permanent after it commits
  - (some would add **Serializable**)

Atomic Commit (cont.)

- To impl. transactions, processes must coordinate!
  - API for bundling related events
  - Coordination between processes
- One protocol: two-phase commit

atomic commit in context

- Problem: coordinator failure after PREPARE & before COMMIT blocks participants waiting for decision (a)
- Three-phase commit overcomes this (b)
  - Idea: delay final decision until enough processes “know” which decision will be taken

Q: can this block somehow?

Two-phase commit (cont.)

- Atomic commit is related to agreement
  - Atomic commit is agreement
  - Participants agree on the outcome of the transaction
  - Restriction: outcome can commit only if all participants ready
  - Since failed processes must not disagree with active processes, it is a variant of uniform agreement
- Describing atomic commit using consensus
  - One participant has to send PREPARE
  - All participants multicast OK/not-OK to all participants
  - Participants use consensus to decide
    - Restriction: if any not-OK or suspected failures then must abort
    - Consensus guarantees all participants decide on same outcome
**State Transfer**

- Reintegrating a failed component requires state transfer!
  - If checkpoint/log to stable storage, recovering replica can do incremental transfer
    - Recover first from last checkpoint
    - Get further logs from active replicas
  - Goal: minimal interference with remaining replicas
  - Problem: state is being updated!
    - Might result in incorrect state transfer (have to coordinate with ongoing messages)
    - Might change such that the new replica can never catch up!
  - Solution: give higher priority to state-transfer messages
    - Lots of variations...

**State Transfer (cont.)**

- One solution for the never-catch-up problem
  - Recovering replica ($p_3$) initiates recovery; gets msgs but drops them
  - $p_3$ sends JOIN msg (total ordered) to initiate state transfer (marks cut)
  - $p_3$ starts logging messages (stops dropping what it gets)
  - $p_3$ gets incremental updates from one of the replicas ($p_2$)
  - $p_3$ executes logged messages after state caught up

**Last Process to Fail**

- If all replicas fail, last to fail has most current state
- How can a recovering process know it failed last?
  - Maintain a **version number** for state
  - All recovering replicas check version numbers
  - Problem: all replicas must recover to know that the latest version is present!
- Better approach: resume as soon as the last replica to fail recovers
  - Technique: use failure detectors
  - When $p_i$ detects $p_j$ has failed, it adds this to its **local obituary log** saved in stable storage
  - The last process to fail is the one that recorded every other processes' failure