CS 347:
Parallel and Distributed
Data Management
Notes06: Reliable Distributed Database Management

Hector Garcia-Molina

Reliable distributed database management

- Reliability
- Failure models
- Scenarios

Reliability

- Correctness
  - Serializability
  - Atomicity
  - Persistence
- Availability

Types of failures

- Processor failures
  - Halt, delay, restart, bezerk, ...
- Storage failures
  - Volatile, non-volatile, atomic write, transient errors, spontaneous failures
- Network failures
  - Lost message, out-of-order messages, partitions, bounded delay

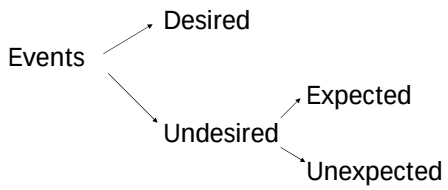
More Types of failures

- Malevolent failures
- Multiple failures
- Detectable failures

Failure models

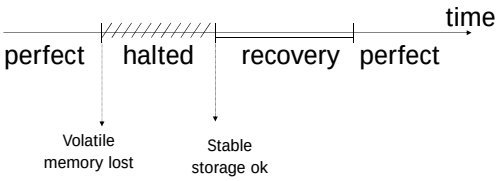
- Cannot protect against everything
- Unlikely failures (e.g., flooding in the Sahara)
- Expensive to protect failures
  (e.g., earthquake)
- Failures we know how to protect against
  (e.g., message sequence numbers; stable storage)

Failure model:



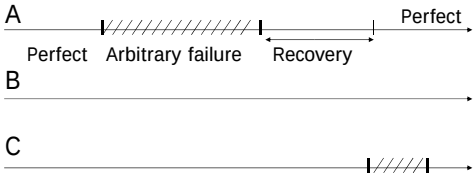
Node models

(1) Fail-stop nodes



Node models

(2) Byzantine nodes



At any given time, at most some fraction f of nodes failed (typically f < 1/2 or f < 1/3)

Network models

(1) Reliable network
- in order messages
- no spontaneous messages
- timeout T_D

If no ack in T_D sec. → Destination down (not paused)

I.e., no lost messages, except for node failures

Variation of reliable net:

- Persistent messages
  - If destination down, net will eventually deliver message
  - Simplifies node recovery, but leads to inefficiencies (hides too much)
  - Not considered here

Network models

(2) Partitionable network
- In order messages
- No spontaneous messages

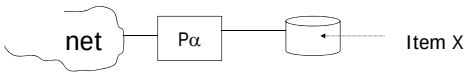
- no timeout; nodes can have different view of failures

Scenarios

- Reliable network
  - Fail-stop nodes
    - No data replication (1)
    - Data replication (2)
- Partitionable network
  - Fail-stop nodes (3)

No Data Replication

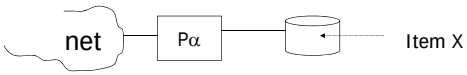
- Reliable network, fail-stop nodes



- Basic idea: node Pα controls X

No Data Replication

- Reliable network, fail-stop nodes

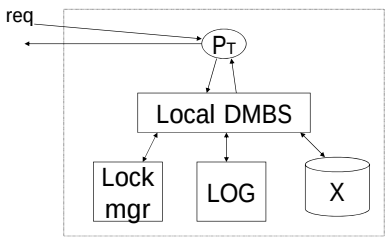


- Basic idea: node Pα controls X

- Single control point simplifies concurrency control, recovery
- Not an availability hit: if Pα down, X unavailable too!

"Pα controls X" means
- Pα does concurrency control for X
- Pα does recovery for X

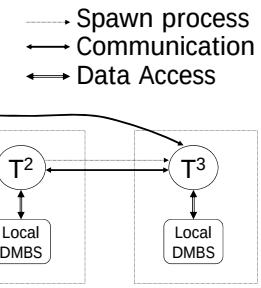
Say transaction T wants to access X:



P_T is process that represents T at this node

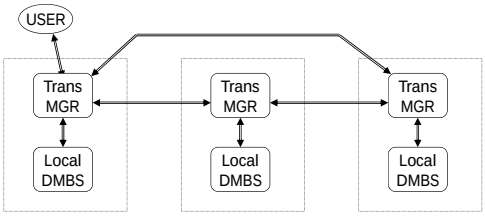
Process models

(A) Cohorts



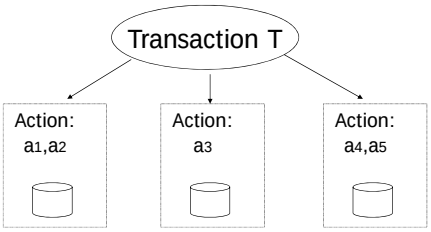
Process models

(B) Transaction servers (manager)

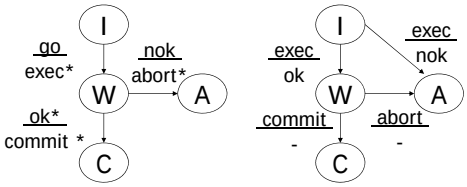


- Cohorts: application code responsible for remote access
- Transaction manager: "system" handles distribution, remote access

Distributed commit problem



Centralized two-phase commit



- Notation: Incoming message
  Outgoing message
  (* = everyone)
- When participant enters "W" state:
  - it must have acquired all resources
  - it can only abort or commit if so instructed by a coordinator
- Coordinator only enters "C" state if all participants are in "W", i.e., it is certain that all will eventually commit

Handling node failures

- Coordinator and participant logs are used to reconstruct state before failure

-> Example: after participant fails:

Log:

	T_1 X undo/redo info	…	T_1 Y info	…	T_1 "W" state	

☛ At recovery:

- T_1 is in "W" state
- Obtain X,Y write locks (no read locks!)
- Wait for message from coordinator

  (or ask coordinator for outcome)

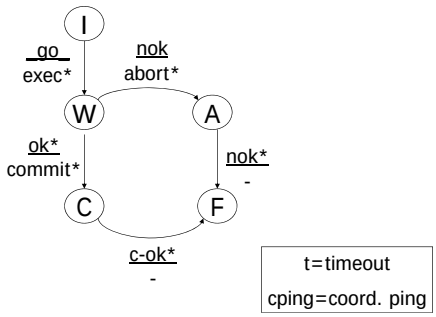
☛ Other examples:

- No "W" record on log ⇒ abort T_1
- See "C" record on log ⇒ finish T_1

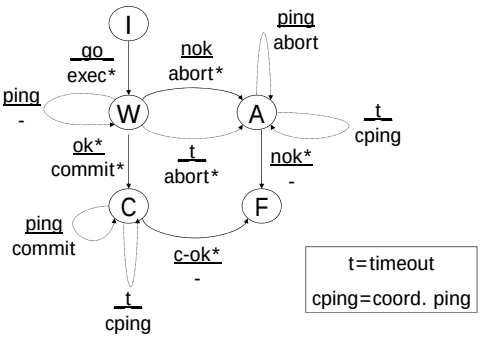
Next

- Add timeouts to cope with messages lost during crashes
- Add finish ("F") state for coordinator – all done, can forget outcome

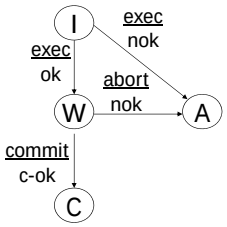
Coordinator



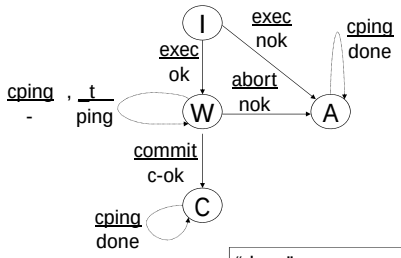
Coordinator



Participant

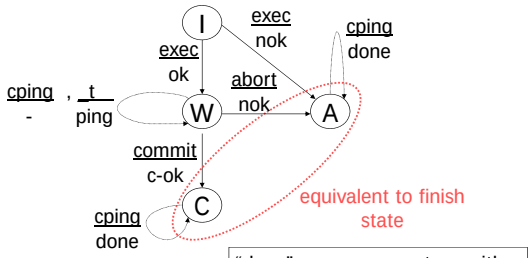


Participant



"done" message counts as either c-ok or n-ok for coordinator

Participant

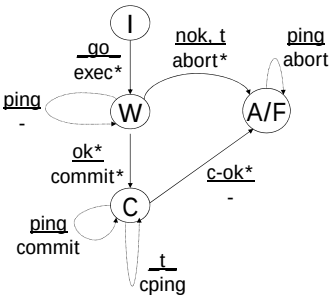


"done" message counts as either c-ok or n-ok for coordinator

Presumed abort protocol

- "F" and "A" states combined in coordinator
- Saves persistent space (forget quicker)
- Presumed commit is analogous

Presumed abort-coordinator (participant unchanged)



Remember:

all state transitions must be logged

Example: tracking who has sent "OK" msgs

Log at coord:

…	T_1 start part={*a*,*b*}	…	T_1 OK from *a* RCV	…

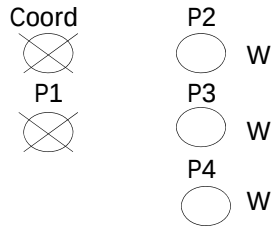
- After failure, we know still waiting for OK from node b
- Alternative: do not log receipts of "OK"s
  abort T_1

Example: logging receipt of C-OK messages

- If logged, can recover state
- If not logged:
  - resend commit *
  - participants reply "done" if duplicate

2PC is blocking

Sample scenario:



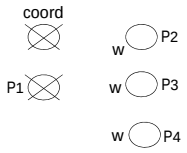
Case I:

$P_1 \rightarrow$ "W"; coordinator sent commits

$P_1 \rightarrow$ "C"

Case II:

$P_1 \rightarrow$ NOK; $P_1 \rightarrow$ A

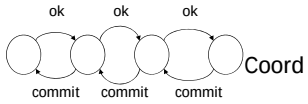


$\Rightarrow$ P_2, P_3, P_4 (surviving participants)

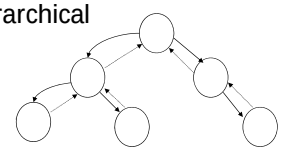
cannot safely abort or commit transaction

Variants of 2PC

- Linear

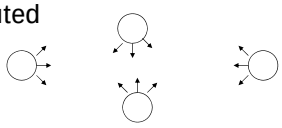


- Hierarchical



Variants of 2PC

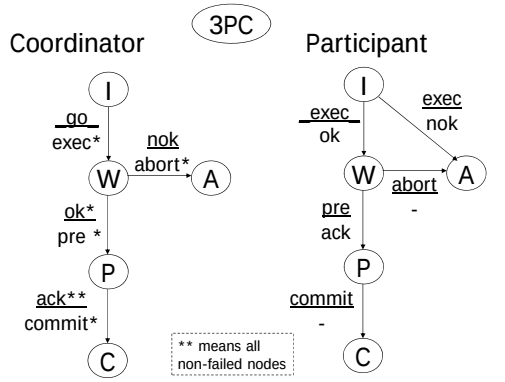
- Distributed



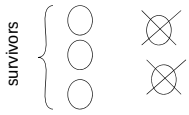
  - Nodes broadcast all messages
  - Every node knows when to commit

3PC = non-blocking commit

- Assume: failed node is down forever
- Key idea: before committing, coordinator tells participants everyone is ok

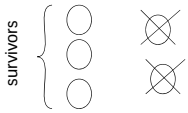


3PC recovery rules: termination protocol

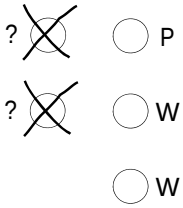


- Survivors try to complete transaction, based on their current states
- Goal:
  - If dead nodes committed or aborted, then survivors should not contradict!
  - Else, survivors can do as they please...

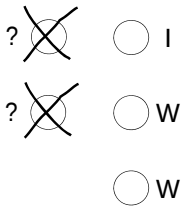
- Let $\{S_1, S_2, \ldots S_n\}$ be survivor sites
- If one or more S_i = COMMIT $\Rightarrow$ COMMIT T
- If one or more S_i = ABORT $\Rightarrow$ ABORT T
- If one or more S_i = PREPARE $\Rightarrow$
  T could not have aborted $\Rightarrow$ COMMIT T
- If no S_i = PREPARE (or COMMIT) $\Rightarrow$
  T could not have committed $\Rightarrow$ ABORT T



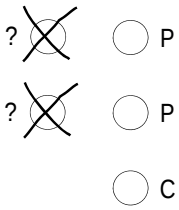
Example:



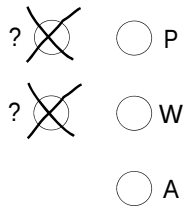
Example:



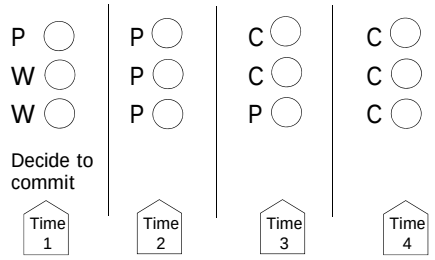
Example:



Example:

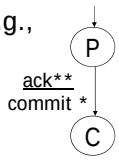


⇔ Once survivors make decision, they must select new coordinator to continue 3PC



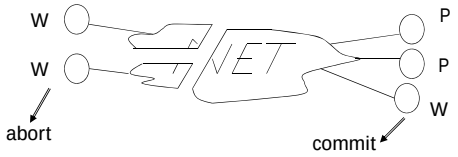
Note: when survivors continue 3PC, failed nodes do not count

E.g.,



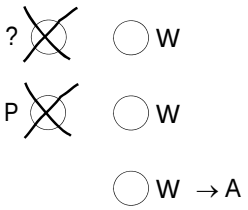
"ack**" ≡ when ack's received from all non-failed nodes

Note: 3PC unsafe with partitions!



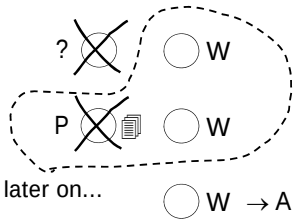
Node recovery:

- After node N recovers from failure:
  - do not participate in termination protocol (why?)



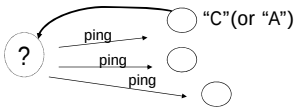
Node recovery:

- After node N recovers from failure:
  - do not participate in termination protocol (why?)



Node recovery:

- After node N recovers from failure:
  - do not participate in termination protocol (why?)
  - wait until it hears commit or abort decision from operational node



- Waiting for commit/abort decision from other node is ok, unless all fail:

Two options for all-failed problem:

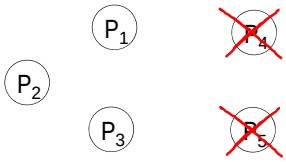
(A) Wait for all to recover
(B) Majority commit

Option A

- Recovering node waits for either:
  (1) commit/abort outcome for T from other node
  (2) all nodes that participated in T are up and recovering:
  $\Rightarrow$ then 3PC can continue
  (no danger that a failed node could have aborted or committed)

Option B

- Want a "gang" of failed but recovered nodes to be able to terminate a transaction even when rest are failed...



Option B

- Nodes are assigned votes, total is V
  Majority is $\frac{V+1}{2}$ e.g., $\begin{cases} V=5 \\ Maj=3 \end{cases}$ $\begin{cases} V=6 \\ Maj=4 \end{cases}$
- To make state transitions, coordinator requires messages from nodes with a majority of votes

Example(1): Coord ⊗ ? $P_2 \rightarrow W$
P_1 ⊗ ? $P_3 \rightarrow W$
? $p_4 \rightarrow W$

- Nodes P_2, P_3, P_4 enter "W" state and fail
- When they recover, coord. and P_1 are down
- Each node has 1 vote, V=5, Maj=3

Example(1): Coord ⊗ ? $P_2 \rightarrow W$
P_1 ⊗ ? $P_3 \rightarrow W$
? $p_4 \rightarrow W$

- Nodes P_2, P_3, P_4 enter "W" state and fail
- When they recover, coord. and P_1 are down
- Each node has 1 vote, V=5, Maj=3
- Since P_2, P_3, P_4 have majority, they know coord. could not have gone to "P" without at least one of their votes
- Therefore, T can be aborted!

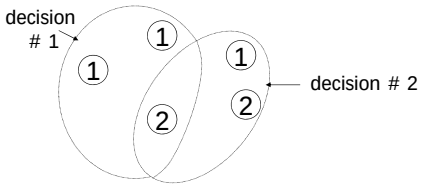
Example(2): Coord ⊗ ? $P_3 \rightarrow$ "P"
P_1 ⊗ ? $P_4 \rightarrow$ "W"
P_2 ⊗

- Each node has 1 vote; V=5, Maj=3
- Nodes fail after entering states shown; P_3, P_4 recover

Example(2): Coord ⊗ ? $P_3 \rightarrow$ "P"
P_1 ⊗ ? $P_4 \rightarrow$ "W"
P_2 ⊗

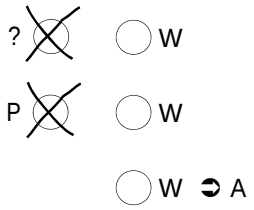
- Each node has 1 vote; V=5, Maj=3
- Nodes fail after entering states shown; P_3, P_4 recover
- Termination rule says we can try to commit, but P_3, P_4 do not have enough votes, so they do nothing!
- P_3, P_4 doing nothing is good because later on, coord. P_1, P_2 could abort T

Summary:
Majority rule ensures that any decision (e.g., Preparing, committing) will be known to any future group making a decision



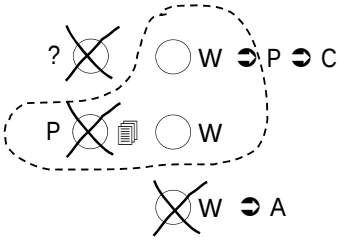
Important Detail for Majority 3PC

- Example:

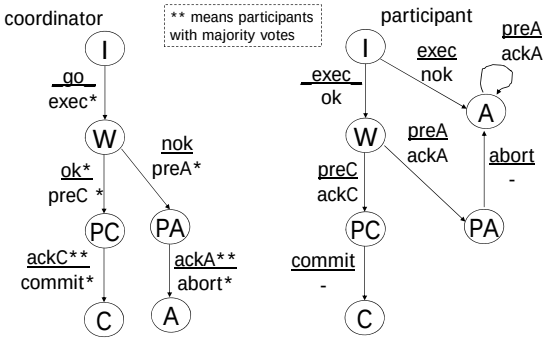


Important Detail for Majority 3PC

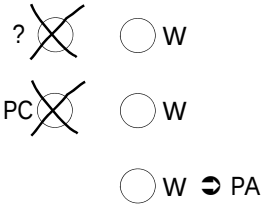
- Example:



Need "Prepare To Abort" State

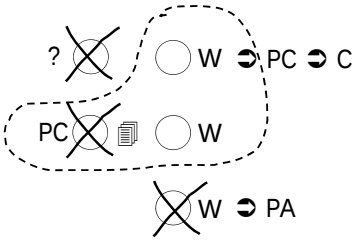


Example Revisited

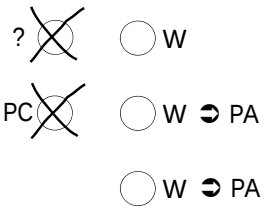


Example Revisited

OK to commit since transaction could not have aborted

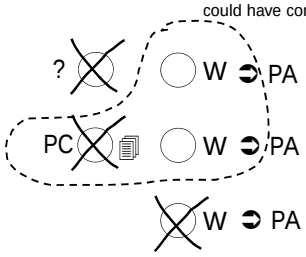


Example Revisited -II



Example Revisited -II

No decision: Transaction could have aborted or could have committed... Block!



3PC with Majority Voting

- If survivors have majority and all states W ⇒ try to abort
- If survivors have majority and states in {W, PC, C} ⇒ try to commit
- If survivors have majority and states in {W, PA, A} ⇒ try to abort
- Otherwise block

Comparison

Option A: only nodes that have not failed participate in 3PC

- Any size group can terminate (even one node)
- If all nodes fail, must wait for all to recover

Comparison

Option B: Majority voting

- A group of failed+recovering nodes can terminate transaction (with majority of votes)
- Need majority for every commit
  - ☒ blocking protocol!

Reminder

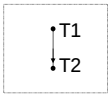
- When node recovers, it uses its log in a normal fashion to determine status of transactions:
  - if commit found in log ⇒ redo if necessary
  - if abort found (or no "W" record) ⇒ rollback if necessary

Reminder - Continued

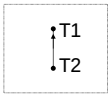
- if in "W" state (or "P" state):
  - reclaim locks held by T before crash
  - try to terminate T (with other nodes)
- after locks claimed for "in doubt" transactions, start normal processing

Final note

- If nodes use 2P locking, global deadlocks possible

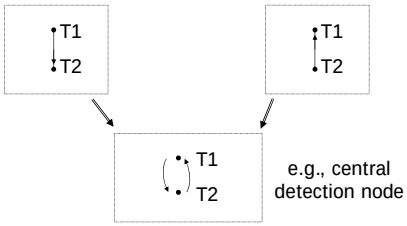


Local WFG: no cycles

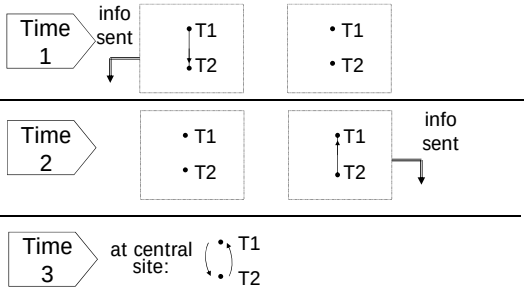


Local WFG: no cycles!

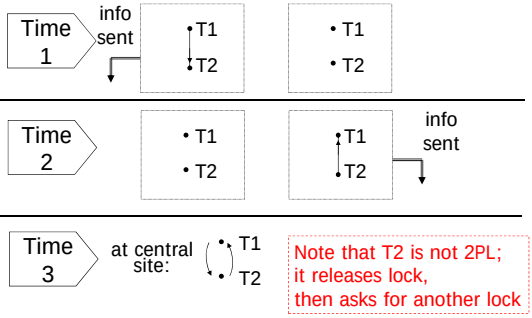
- Need to "combine" WFGs to discover global deadlock



Problem: False deadlocks



Problem: False deadlocks



Exercise

- Assume all waits are due to transaction lock requests
- Assume transactions well formed and 2PL; scheduler legal
- Show that false deadlocks are not possible

- Many deadlock solutions
  - Distributed vs. centralized
  - Detection vs. prevention
    - timeouts
    - wait-die
    - wound-wait
- Covered in CS245