Distributed Databases

CS347
Lecture 15
June 4, 2001

Topics for the day

- Concurrency Control
  - Schedules and Serializability
  - Locking
  - Timestamp control
- Reliability
  - Failure models
  - Two-phase commit protocol

Example

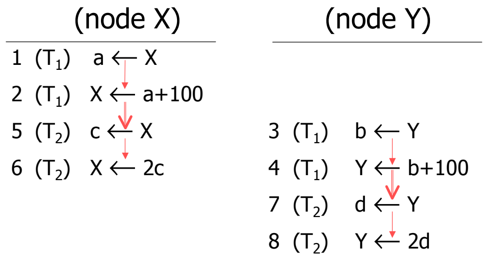
X	Y

Node 1 ↗ ↖ Node 2

constraint: X=Y

	T_1		T_2
1	$a \leftarrow X$	5	$c \leftarrow X$
2	$X \leftarrow a+100$	6	$X \leftarrow 2c$
3	$b \leftarrow Y$	7	$d \leftarrow Y$
4	$Y \leftarrow b+100$	8	$Y \leftarrow 2d$

Possible Schedule



(node X)	(node Y)
1 (T_1) $a \leftarrow X$	
2 (T_1) $X \leftarrow a+100$	
5 (T_2) $c \leftarrow X$	3 (T_1) $b \leftarrow Y$
6 (T_2) $X \leftarrow 2c$	4 (T_1) $Y \leftarrow b+100$
	7 (T_2) $d \leftarrow Y$
	8 (T_2) $Y \leftarrow 2d$

If X=Y=0 initially, X=Y=200 at end

Precedence: intra-transaction ↓
inter-transaction ↓

Definition of a Schedule

Let T= {T_1, T_2,…, T_N} be a set of transactions. A schedule S over T is a partial order with ordering relation $<_S$ where:

1. $S = \cup T_i$
2. $<_S \supseteq \cup <_i$
3. for any two conflicting operations $p,q \in S$, either $p <_S q$ or $q <_S p$

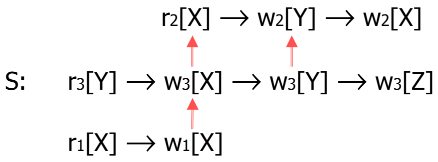
Note: In centralized systems, we assumed S was a total order and so condition (3) was unnecessary.

Example

(T_1) $r_1[X] \rightarrow w_1[X]$

(T_2) $r_2[X] \rightarrow w_2[Y] \rightarrow w_2[X]$

(T_3) $r_3[X] \rightarrow w_3[X] \rightarrow w_3[Y] \rightarrow w_3[Z]$



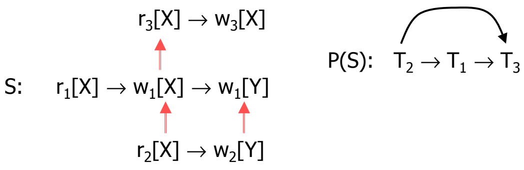
$r_2[X] \rightarrow w_2[Y] \rightarrow w_2[X]$

S: $r_3[Y] \rightarrow w_3[X] \rightarrow w_3[Y] \rightarrow w_3[Z]$

$r_1[X] \rightarrow w_1[X]$

Precedence Graph

- Precedence graph P(S) for schedule S is a directed graph where
  - Nodes = {T_i | T_i occurs in S}
  - Edges = {$T_i \rightarrow T_j$ | $\exists\, p \in T_i, q \in T_j$ such that p, q conflict and $p <_S q$}



$r_3[X] \rightarrow w_3[X]$

S: $r_1[X] \rightarrow w_1[X] \rightarrow w_1[Y]$

$r_2[X] \rightarrow w_2[Y]$

P(S): $T_2 \rightarrow T_1 \rightarrow T_3$

Serializability

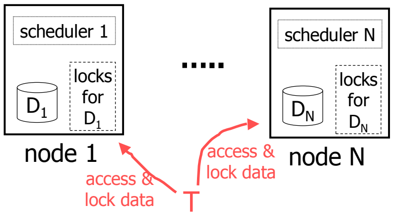
Theorem: A schedule S is serializable iff P(S) is acyclic.

Enforcing Serializability

- Locking
- Timestamp control

Distributed Locking

- Each lock manager maintains locks for local database elements.
- A transaction interacts with multiple lock managers.



Locking Rules

- Well-formed/consistent transactions
  - Each transaction gets and releases locks appropriately
- Legal schedulers
  - Schedulers enforce lock semantics
- Two-phase locking
  - In every transaction, all lock requests precede all unlock requests.

These rules guarantee serializable schedules

Locking replicated elements

- Example:
  - Element X replicated as X_1 and X_2 on sites 1 and 2
  - T obtains read lock on X_1; U obtains write lock on X_2
  - Possible for X_1 and X_2 values to diverge
  - Possible that schedule may be unserializable
- How do we get global lock on logical element X from local locks on one or more copies of X?

Primary-Copy Locking

- For each element X, designate specific copy X_i as primary copy
- Local-lock(X_i) $\Rightarrow$ Global-lock(X)

Synthesizing Global Locks

- Element X with n copies X_1 …. X_n
- Choose "s" and "x" such that
  - $2x > n$
  - $s + x > n$
- Shared-lock(s copies) $\Rightarrow$ Global-shared-lock(X)
- Exclusive-lock(x copies) $\Rightarrow$ Global-exclusive-lock(X)

Special cases

Read-Lock-One; Write-Locks-All ($s = 1, x = n$)

- Global shared locks inexpensive
- Global exclusive locks very expensive
- Useful when most transactions are read-only

Majority Locking ($s = x = \lceil (n+1)/2 \rceil$)

- Many messages for both kinds of locks
- Acceptable for broadcast environments
- Partial operation under disconnected network possible

Timestamp Ordering Schedulers

Basic idea: Assign timestamp ts(T) to transaction T. If $ts(T_1) < ts(T_2) \ldots < ts(T_n)$, then scheduler produces schedule equivalent to serial schedule $T_1 T_2 T_3 \ldots . T_n$.

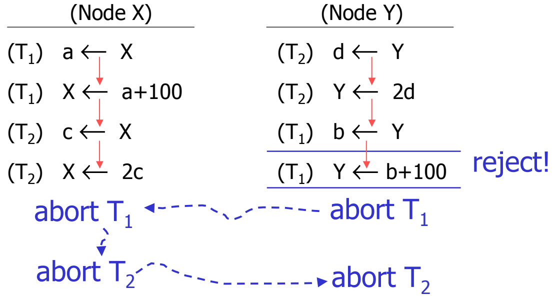
TO Rule: If $p_i[X]$ and $q_j[X]$ are conflicting operations, then $p_i[X] <_S q_j[X]$ iff $ts(T_i) < ts(T_j)$.

Supply proof.

Theorem: If S is a schedule that satisfies TO rule, P(S) is acyclic (hence S is serializable).

Example

$ts(T_1) < ts(T_2)$



(Node X)	(Node Y)
(T_1) $a \leftarrow X$	(T_2) $d \leftarrow Y$
(T_1) $X \leftarrow a+100$	(T_2) $Y \leftarrow 2d$
(T_2) $c \leftarrow X$	(T_1) $b \leftarrow Y$
(T_2) $X \leftarrow 2c$	(T_1) $Y \leftarrow b+100$

reject!

abort T_1 ← abort T_1

abort T_2 → abort T_2

Strict T.O

- Problem: Transaction reads "dirty data". Causes cascading rollbacks.
- Solution: Enforce "strict" schedules in addition to T.O rule

Lock written items until it is certain that the writing transaction has committed.

Use a commit bit C(X) for each element X. C(X) = 1 iff last transaction that last wrote X committed. If C(X) = 0, delay reads of X until C(X) becomes 1.

Revisit example under strict T.O

$ts(T_1) < ts(T_2)$

(Node X)	(Node Y)
(T_1) a ⟵ X	(T_2) d ⟵ Y
(T_1) X ⟵ a+100	(T_2) Y ⟵ 2d
(T_2) c ⟵ X delay	(T_1) ~~b ⟵ Y~~ reject!
abort T_1	abort T_1
(T_2) c ⟵ X	
(T_2) X ⟵ 2c	

Enforcing T.O

For each element X:

MAX_R[X] → maximum timestamp of a transaction that read X

MAX_W[X] → maximum timestamp of a transaction that wrote X

rL[X] → number of transactions currently reading X (0,1,2,…)

wL[X] → number of transactions currently writing X (0 or 1)

queue[X] → queue of transactions waiting on X

T.O. Scheduler

r_i [X] arrives:

- If $(ts(T_i) <$ MAX_W[X]) abort T_i
- If $(ts(T_i) >$ MAX_R[X]) then MAX_R[X] = $ts(T_i)$
- If (queue[X] is empty and wL[X] = 0)
  - rL[X] = rL[X]+1
  - begin r_i[X]
- Else add (r,Ti) to queue[X]

Note: If a transaction is aborted, it must be restarted with a <u>larger</u> timestamp. Starvation is possible.

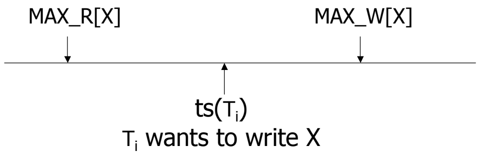
T.O. Scheduler

w_i[X] arrives:

- If $(ts(T_i) <$ MAX_W[X] or $ts(T_i) <$ MAX_R[X]) abort T_i
- MAX_W[X] = $ts(T_i)$
- If (queue[X] is empty and wL[X]=0 AND rL[X]=0)
  - wL[X] = 1
  - begin w_i[X]
  - wait for T_i to complete
- Else add (w, Ti) to queue

Work out the steps to be executed when r_i[X] or w_i[X] completes.

Thomas Write Rule

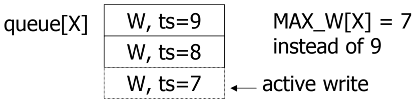


$w_i[X]$ arrives:

- If (ts(T_i) < MAX_R[X]) abort T_i
- If (ts(T_i) < MAX_W[X]) ignore this write.
- Rest as before…..

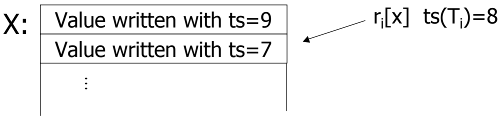
Optimization

- Update MAX_R and MAX_W when operation is executed, not when enqueued. Example:

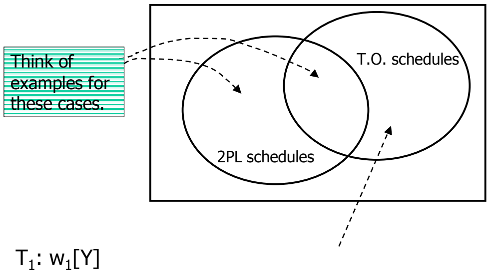


queue[X]		
	W, ts=9	MAX_W[X] = 7 instead of 9
	W, ts=8	
	W, ts=7	← active write

- Multi-version timestamps



2PL ≠ T.O



T_1: $w_1[Y]$

T_2: $r_2[X]$ $r_2[Y]$ $w_2[Z]$ $ts(T_1)<ts(T_2)<ts(T_3)$

T_3: $w_3[X]$

Schedule S: $r_2[X]$ $w_3[X]$ $w_1[Y]$ $r_2[Y]$ $w_2[Z]$

Timestamp management

	MAX_R	MAX_W
X_1		
X_2		
⋮		
	⋮	⋮
X_n		

- Too much space
- Additional IOs

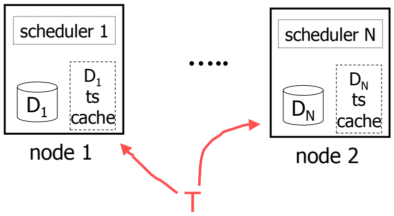
Timestamp Cache

Item	MAX_R	MAX_W
X		
Y		
⋮		
Z		

ts_{MIN}

- If a transaction reads or writes X, make entry in cache for X (add row if required).
- Choose $ts_{MIN} \approx$ current time – d
- Periodically purge all items X with MAX_R[X] < ts_{MIN} & MAX_W[X] < ts_{MIN} and store ts_{MIN}.
- If X has cache entry, use those MAX_R and MAX_W values. Otherwise assume MAX_R[X] = MAX_W[X] = ts_{MIN} .

Distributed T.O Scheduler



- Each scheduler is "independent"
- At end of transaction, signal all schedulers involved, indicating commit/abort of transaction.

Reliability

- Correctness
  - Serializability
  - Atomicity
  - Persistence
- Availability

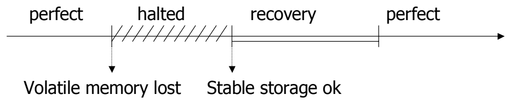
Types of failures

- Processor failures
  - Halt, delay, restart, berserk, ...
- Storage failures
  - Transient errors, spontaneous failures, persistent write errors
- Network failures
  - Lost messages, out-of-order messages, partitions

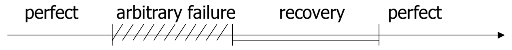
- Other ways of characterizing failures
  - Malevolent/Unintentional failures
  - Single/Multiple failures
  - Detectable/Undetectable failures

Models for Node Failure

(1) Fail-stop nodes



(2) Byzantine nodes



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

Models for Network Failure

(1) Reliable network

- in order messages
- no spontaneous messages
- timeout T_D

If no ack in T_D sec. → Destination down

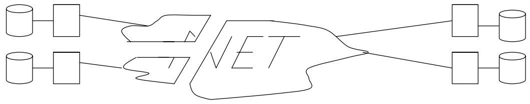
(2) Persistent messages

- if destination is down, network will eventually deliver messages.
- simplifies node recovery but inefficient (hides too much in network layer)

Models for Network Failure

(3) Partitionable network

- in order messages
- no spontaneous messages
- no timeouts

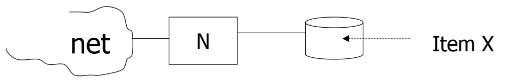


Scenarios

- Reliable network and Fail-stop nodes
  - No data replication (1)
  - Data replication (2)
- Partitionable network and Fail-stop nodes
  - No data replication (3)
  - Data replication (4)

Scenario 1

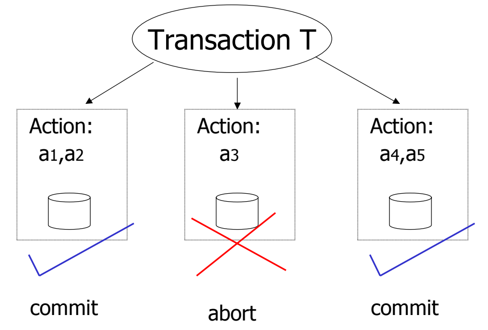
Reliable network, fail-stop nodes, no data replication



Key consequence: node N "controls" X

- N is responsible for concurrency control and recovery of X
- Single control point for each data element
- If N is down, X is unavailable

Distributed commit problem

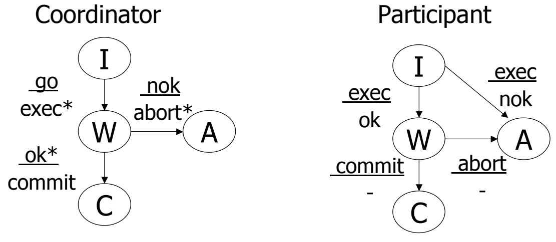


Distributed Commit

- Make global decision on committing or aborting a distributed transaction
- Assume atomicity mechanisms at each site ensure each local component is atomic
  - Each component either commits or has no effect on local database
- Enforce rule that either all components commit or all abort

Centralized two-phase commit

State Transition Diagram



Notation: Incoming Message / Outgoing Message (* = everyone)

Key Points

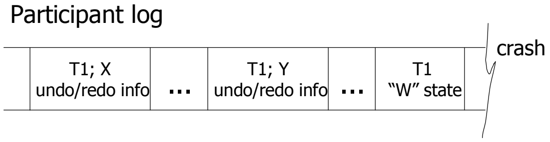
- When participant enters "W" state:
  - It must have acquired all resources (e.g. locks) required for commit
  - But, it can only commit when so instructed by the coordinator
- After sending "nok" participant can unilaterally abort.
- Coordinator enters "C" state only if <u>all</u> participants are in "W", i.e., it is certain that all participants will <u>eventually</u> commit.

Handling node failures

- Coordinator and participant logs used to reconstruct state before failure.
- Important that each message is logged before being sent
- Coordinator failure may require <u>leader election</u>
- Participant failure: recovery procedure depends on last log record for T
  - "C" record: commit T
  - "A" record: abort T
  - "W" record: obtain write locks for T and wait/ask coordinator or other participant
  - No log records for T: abort T

Example

Participant log

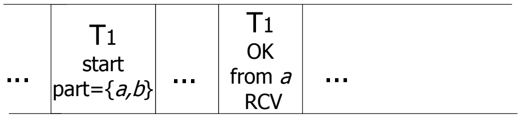

T1; X undo/redo info
…
T1; Y undo/redo info
…
T1 "W" state
crash

- During recovery at participant:
  - Obtain write locks for X and Y (no read locks)
  - Wait for message from coordinator (or ask coordinator)

Logging at the coordinator

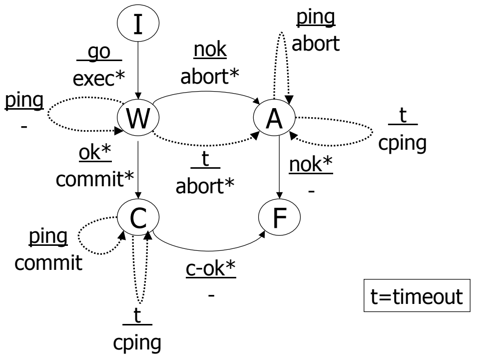
<u>Example</u>: tracking who has sent "OK" msgs

Log at coord:


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

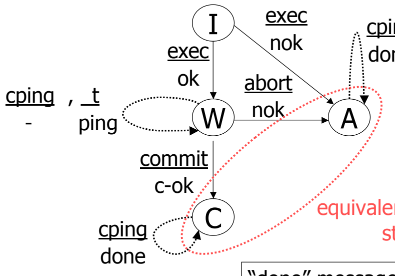
- After failure, we know still waiting for OK from node b
- Alternative: do not log receipts of "OK"s. Simply abort T_1

Coordinator (with timeouts and finish state)



t=timeout

Participant (with timeouts and finish state)



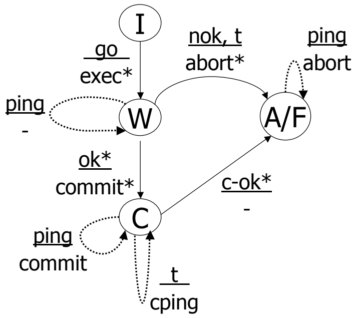
equivalent to finish state

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

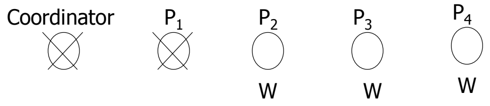
Presumed abort protocol

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

Presumed abort-coordinator (participant unchanged)



2PC is blocking



Case I: P1 → "W"; coordinator sent commits

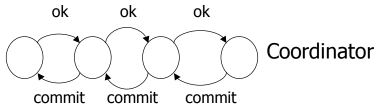
P1 → "C"

Case II: P1 → NOK; P1 → A

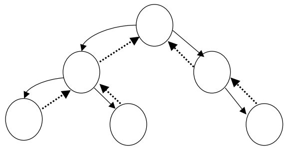
⇒ P2, P3, P4 (surviving participants) cannot safely abort or commit transaction

Variants of 2PC

Linear

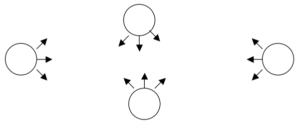


Hierarchical



Variants of 2PC

Distributed



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

Resources

- "Concurrency Control and Recovery" by Bernstein, Hardzilacos, and Goodman
  - Available at http://research.microsoft.com/pubs/ccontrol/
- Timestamp control
  - Chapter 9 of the CS245 Textbook ("Database System Implementation" by Garcia-Molina, Ullman, and Widom)