Topics in Reliable Distributed Systems

049017

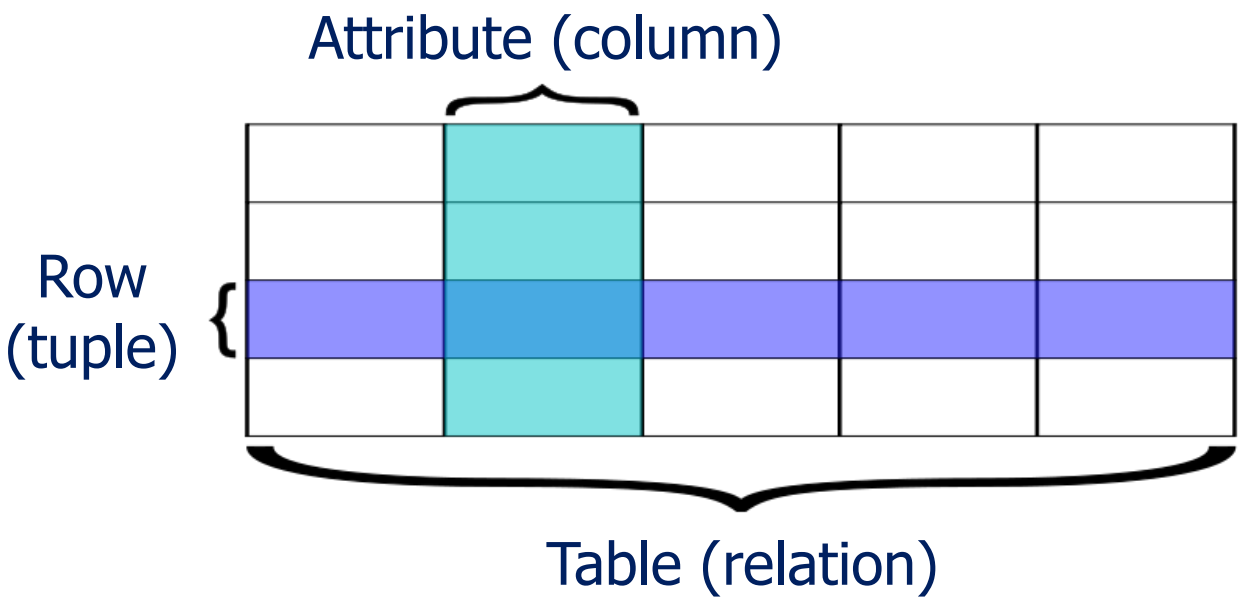
TRANSACTION SYSTEMS

What is A Database?

- Organized collection of data
  - typically persistent
  - organization models: relational, object-based, hierarchal
- Retrieval using query languages
- Concurrently accessed by many users
  - supports transactions
- Managed by DBMS (DataBase Management System) software

Relational Database

- A collection of tables (relations)
- A table is defined with a fixed set of attributes (columns)
  - e.g., person table attributes: id, name, age, address
- Data is stored in rows (tuples)



Querying Relational Databases

- High-level query languages based on set-theory, relations
  - SQL most popular
- Select data from tables
  - conditionally select based on attributes
    - e.g., where name = Smith and age > 16
  - project – keep only some columns
    - e.g., only name and address
  - join – combine info from multiple tables
- Use set theory operations on the results
  - union, difference, Cartesian product

Join Example

Course Registration Table

Student ID	Course
123456789	046209
…	…

Exam Dates Table

Course	Exam Date
046209	25/6
…	…

- We can retrieve student IDs of all students who have exams on 25/6

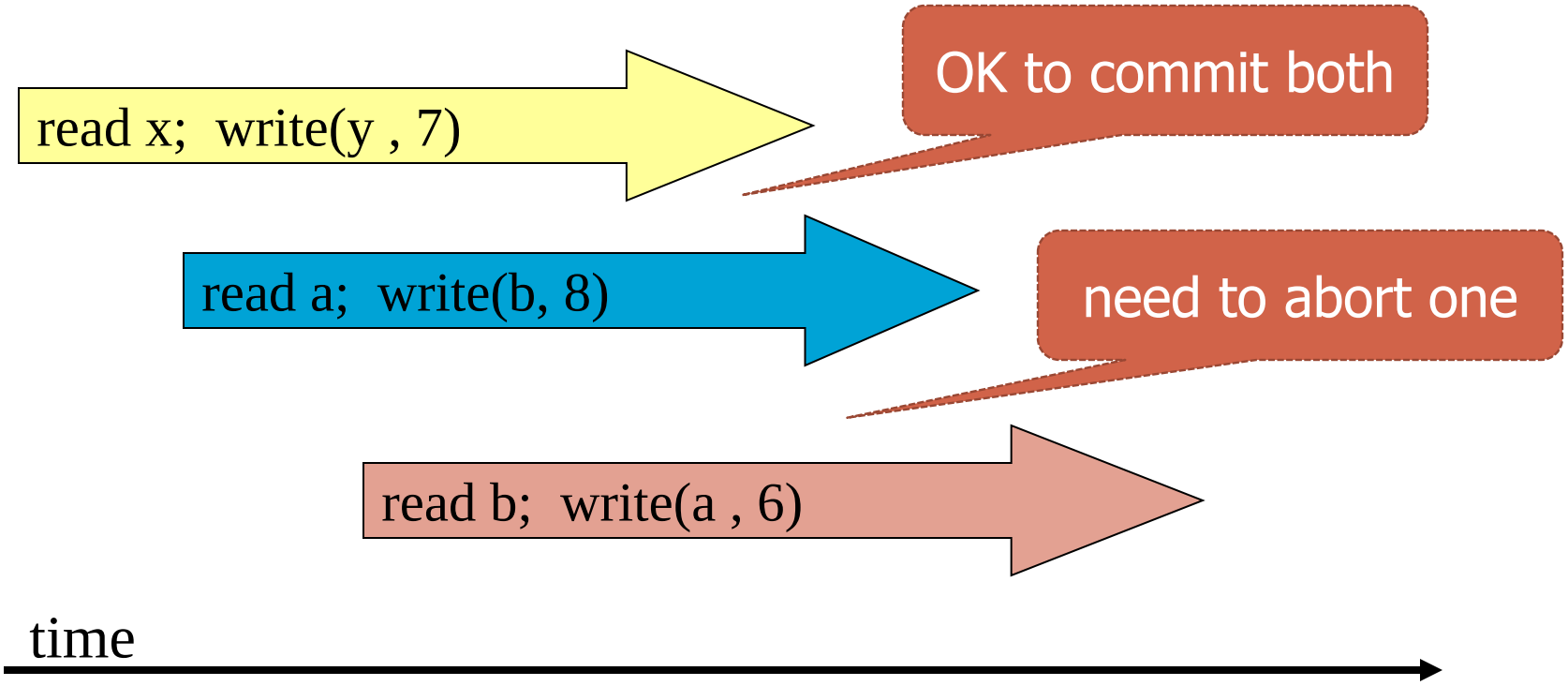
Transactions [Gray 78]

- A collection of query and update operations executed atomically
  - may succeed and commit
  - may fail and abort
    - by the programmer or by the DBMS
- So-called ACID properties:
  - Atomicity: all-or-nothing
  - Consistency: correct execution
  - Isolation: no partial results observed
  - Durability: persistence

Serializability

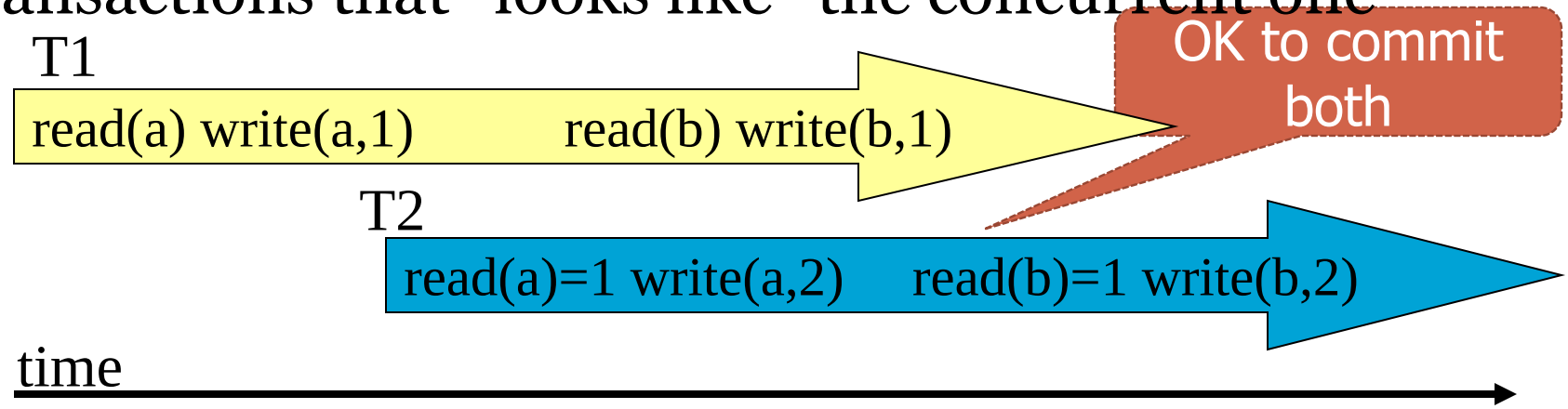
Transactions and Concurrency

- Concurrent transactions are allowed, unless they conflict

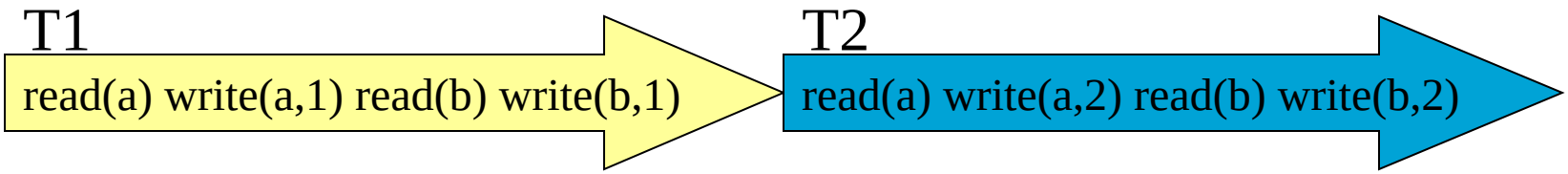


Formal Criterion: Serializability

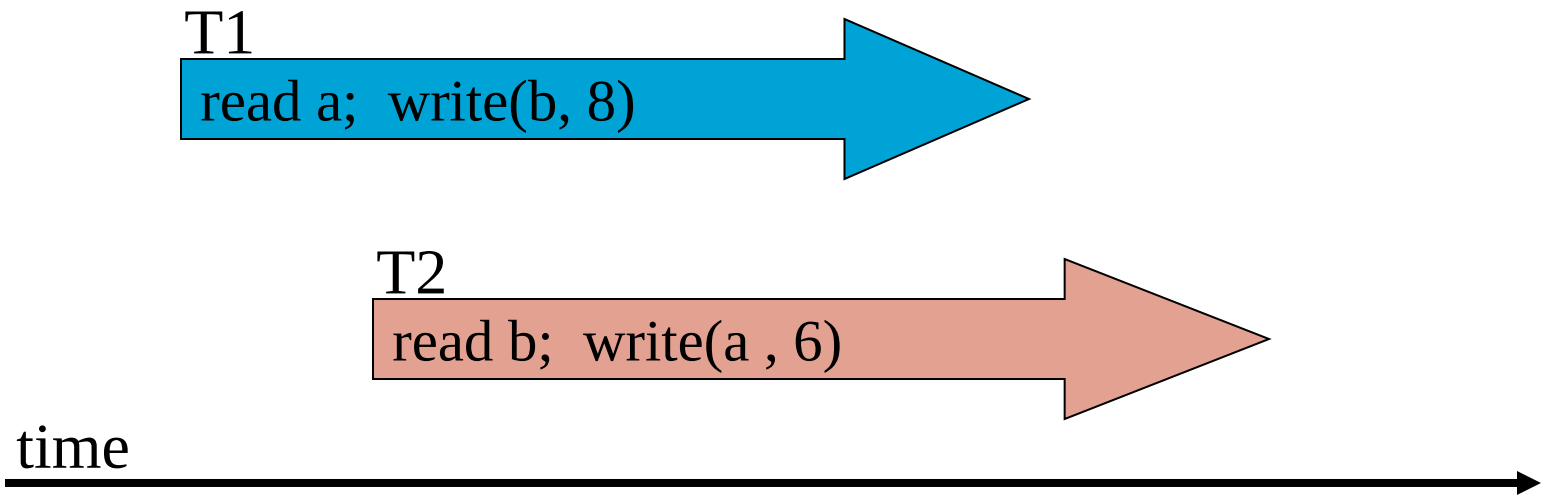
- There is some sequential execution of the same transactions that "looks like" the concurrent one



Looks like:



When Serializability is Violated



- Can't order T1 before T2 – why?
  - reads old value of a
- And vice-versa
- What do we do?

Ensuring ACID

- DBMSs use two complementary mechanisms to achieve the ACID properties

1. Concurrency control
   - controls concurrent executions
2. Recovery
   - kicks in on recovery from crash

Concurrency Control Approaches

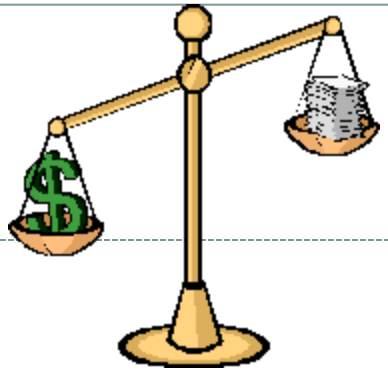
Pessimistic approach

- a transaction locks all items before accessing them
- releases locks at the end (2-phase locking)
- downsides?

Optimistic approach

- ongoing transactions write to private copy
- recall the old value of every read object
- at the end – check if old values changed
- what do we do if they changed?

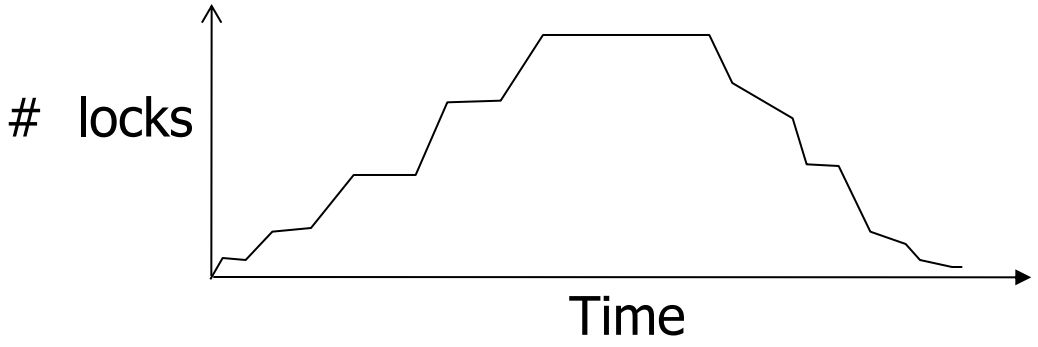
Tradeoffs



- Pessimistic may deadlock
- Optimistic may lead to livelock
- Optimistic may allow more parallelism
  - commit in more cases
- Optimistic may waste more work
  - because of being “too optimistic”

2-Phase Locking (2PL)

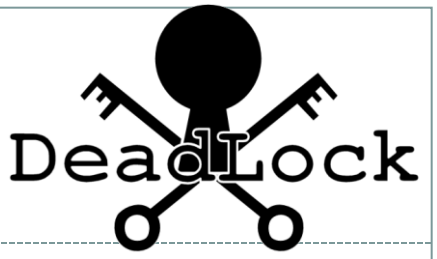
- Transaction T must obtain a lock on every item before accessing it
- Two types of locks (readers-writers problem)
  - exclusive – for writes
  - shared – for reads
  - conflicts: read-write or write-write
- If T releases some lock, it cannot acquire any locks afterwards
- Ensures serializability



When To Release?

- To release, we need to know T won't need any more locks
  - typically not known before T tries to commit
- Moreover, if we release exclusive locks before commit, other transactions may read "dirty" values
  - this may lead to cascading aborts
- Solution: Strict 2PL
  - release locks after commit/abort

Deadlock Happens

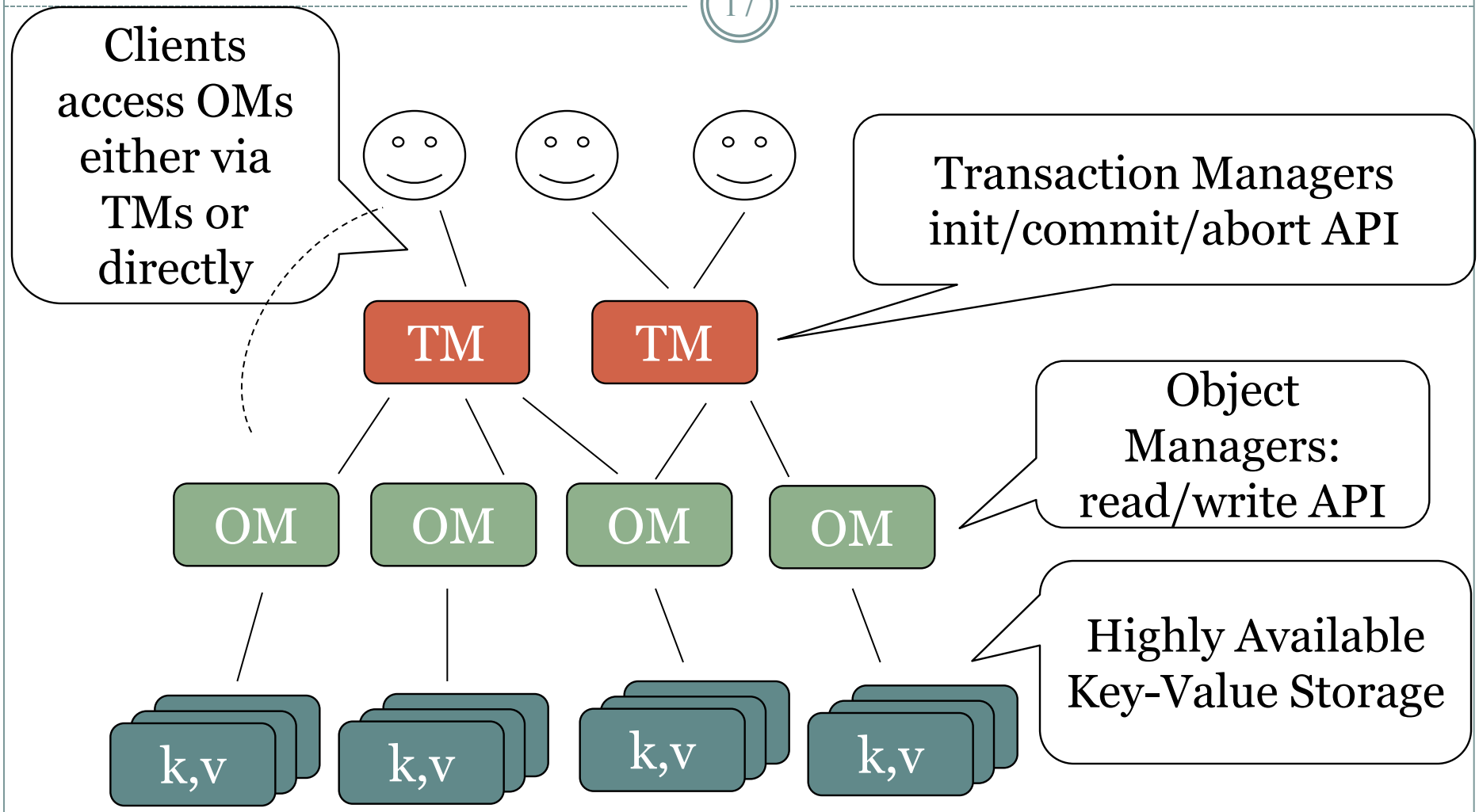


- Construct a deadlock scenario with 2 concurrent transactions
- Possible solutions:
  - periodic cycle detection
  - timeout-based: if a transaction waits for a lock for “too long” it aborts

Ensuring Atomicity + Persistence

- Based on logs (journals)
  - WAL – Write Ahead Log: write to the log before writing to the DB
- Log records for
  - writes – includes T's id, old and new values
  - T end – commit or abort
- Used to
  - undo (rollback) operations of aborted transactions
  - redo updates lost due to crash (on recovery)
- Two approaches:
  - write-in-place – need to undo database updates of aborted transactions
  - delayed-write – write in log first, lazily write in place committed transactions only

Multi-Tier Distributed Architecture



Distributed Transactions

- Databases nowadays are often partitioned/sharded across multiple machines
- Data items are also replicated, but usually at a lower level of abstraction
  - Highly Available (HA) Key-Value Store or Object Store
- Transactions may span multiple sites
- Need to ensure atomic commit: either a transaction commits at all sites, or aborts at all of them
- If any site needs to abort the transaction, it aborts

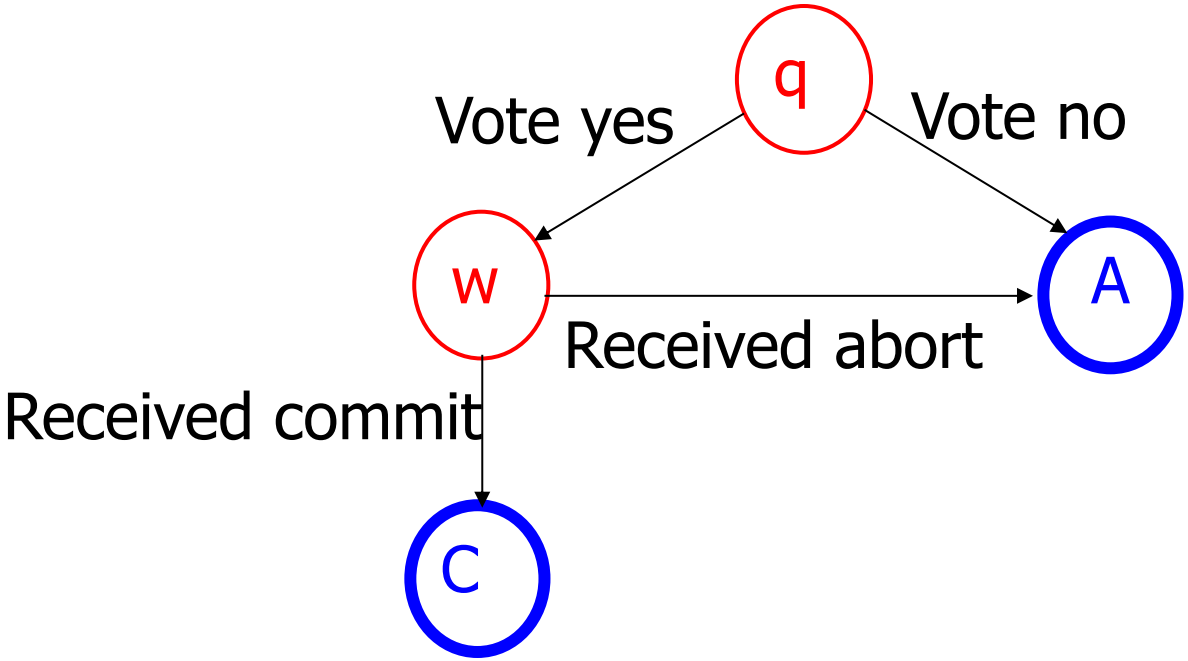
Non-Blocking Atomic Commit

- Uniform agreement: all processes that decide, decide on the same value
  - decisions are not reversible
- Validity: commit can only be reached if all processes vote 'yes'
- Non triviality: if all vote 'yes' and there are no (suspicions of) failures, then the decision is commit
- Termination: all live processes eventually decide

2 Phase Commit (2PC)

- One node is the coordinator C
  - C sends the transaction to everyone
- Every other node sends its vote to C
- C waits until either
  - it receives replies from all,
  - it receives one ‘no’ vote,
  - or some node fails (suspected to have failed)
- If all vote ‘yes’, then send commit to all
- Otherwise, send abort to all
- Other nodes do what C says

2PC Flow (Non-Coordinator)



2PC Properties

- Simple, fast ⇒ most commonly used
- Problem?
- Blocking: if C fails, everyone is stuck
  - e.g., if everyone voted yes but did not receive an answer, it is unknown whether C committed or aborted before failing
  - Skeen & Stonebraker proved that if the network can partition, blocking is unavoidable

3PC Protocol [Skeen '82]

- Avoids blocking when the coordinator crashes
- Assumes:
  1. failures are accurately detected, aka fail-stop model or perfect failure detector
  2. at most one process crashes during a transaction's execution
- Adds a prepare phase before commit
  - the coordinator announces what it plans to decide before it decides

3PC Normal Operation

Coordinator

Send Transaction

If all replied yes, send PreCommit

Else send Abort

If got ACKs from all, send Commit

Else send Abort

Participant

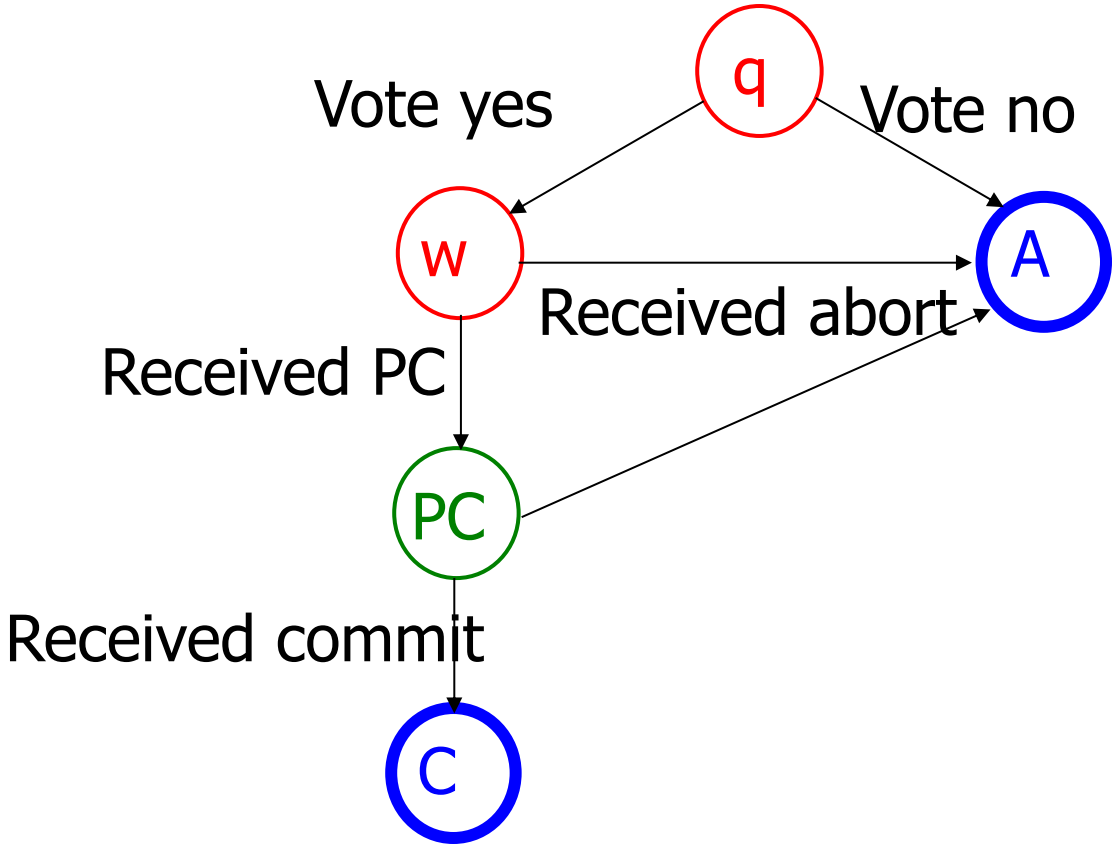
Send vote (Yes/No)

If got PreCommit, send ACK

If got Commit, commit

If got Abort, abort

3PC Flow (Non-Coordinator)



Rationale

- Normal mode is correct: all do what C says, and C doesn't change its mind
  - this was true for 2PC too ☺
- Recovery from one failure now possible
  - if C committed, then every other process is either in the C state or in the PC state
  - if all remaining nodes are in the PC state (recall, only C failed), no node aborted
  - no ambiguity on whether C committed or aborted before failing

3PC Recovery

- On detecting C's failure (recall, this is accurate!) run recovery procedure:

1. Elect new coordinator C'
   - e.g., live node with smallest id
2. C' collects the states of all live processes
3. C' chooses its state as follows:
   - if one process said Abort— A
   - if one process said Commit — C
   - if one process said PC — PC
   - else Abort (why is this valid?)
4. Continue as in the normal mode

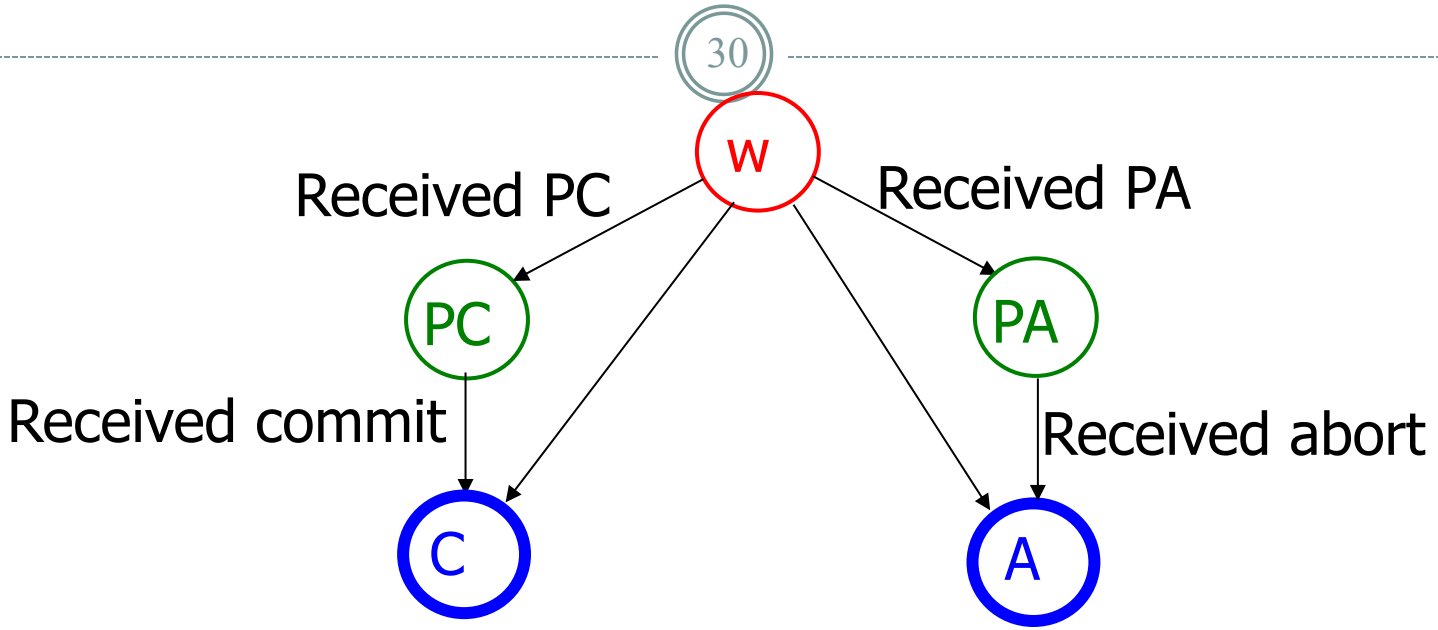
Notes on 3PC

- 3PC is incorrect when there are false suspicions
  - i.e., when nodes suspect a non-faulty coordinator
  - doesn't work if the network can partition!
- 3PC is incorrect when C and one more node fail during the same transaction!
  - C can send abort from PC state due to another node's failure
  - other nodes cannot tell the difference between this and C sending commit
- Homework: construct two indistinguishable executions s.t. the recovery phase must commit in one and abort in the other

A Correct Solution

- Quorum-based recovery
  - coordinator needs majority to support any decision (commit or abort)
  - add pre-abort state
- Option 1: Direct Quorum-based recovery, E3PC [Keidar, Dolev '98]
  - Ensures safety with imperfect failure detection, network partitions, unbounded number of failures
  - Requires live connected majority in order to make progress
  - blocks otherwise (this is inherent)
- Option 2: Use 2PC, but replace coordinator with replicated state machine, e.g., using Paxos

Quorum-Based Recovery



- If some node committed, then a majority is in the C or PC states
- If some node aborted, then a majority is in the A state or PA states
- If we talk to a majority – no ambiguity on whether the previous C committed or aborted before failing

Distributed Commit Summary

- Distributed transactions are tricky
  - especially if we want to tolerate partitions
- Impossible to overcome unbounded message loss
  - cf. 2-Generals Problem
- Impossible to ensure progress without a live, connected majority
- In real-life, 2PC is “good enough”
  - may sometimes block
  - HA solutions replicate the coordinator