Principles of Distributed Database Systems

TS. Phan Thị Hà

Outline

- Introduction
- Distributed and parallel database design
- Distributed data control
- Distributed Transaction Processing
- Data Replication
- Database Integration – Multidatabase Systems
- Parallel Database Systems
- Peer-to-Peer Data Management
- Big Data Processing
- NoSQL, NewSQL and Polystores
- Web Data Management

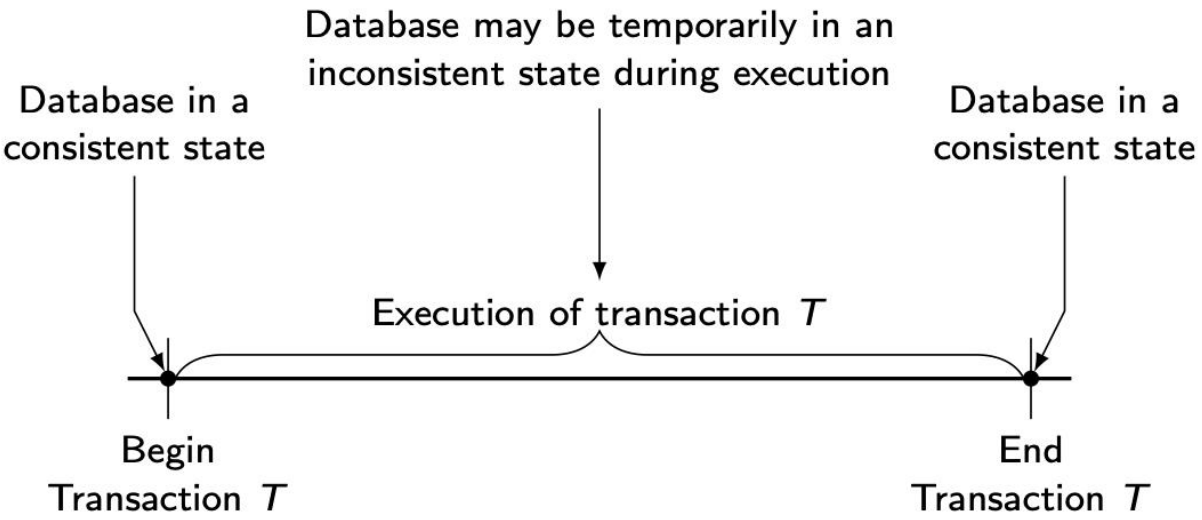
Outline

- Distributed Transaction Processing
  - Distributed Concurrency Control
  - Distributed Reliability

Transaction

A transaction is a collection of actions that make consistent transformations of system states while preserving system consistency.

- concurrency transparency
- failure transparency



Transaction Characterization

Begin_transaction
…
Read
Read
…
Write
Read
…
Commit

- Read set (RS)
  - The set of data items that are read by a transaction
- Write set (WS)
  - The set of data items whose values are changed by this transaction
- Base set (BS)
  - RS ∗ WS

Principles of Transactions

ATOMICITY

- all or nothing

CONSISTENCY

- no violation of integrity constraints

ISOLATION

- concurrent changes invisible $\Rightarrow$ serializable

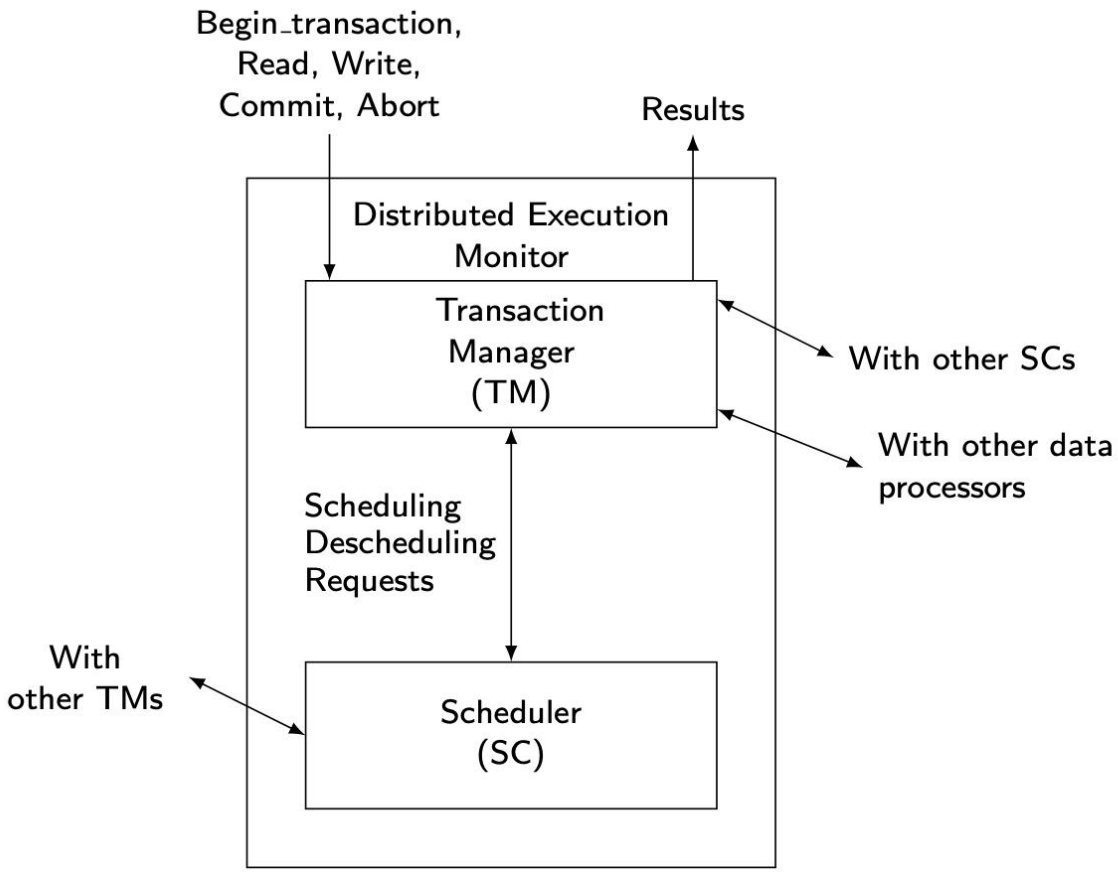
DURABILITY

- committed updates persist

Transactions Provide…

- *Atomic* and *reliable* execution in the presence of failures
- *Correct* execution in the presence of multiple user accesses
- Correct management of *replicas* (if they support it)

Distributed TM Architecture



Outline

- Distributed Transaction Processing
  - Distributed Concurrency Control
  - Distributed Reliability

Concurrency Control

- The problem of synchronizing concurrent transactions such that the consistency of the database is maintained while, at the same time, maximum degree of concurrency is achieved.
- Enforce isolation property
- Anomalies:
  - Lost updates
    - The effects of some transactions are not reflected on the database.
  - Inconsistent retrievals
    - A transaction, if it reads the same data item more than once, should always read the same value.

Serializability in Distributed DBMS

- Two histories have to be considered:
  - local histories
  - global history
- For global transactions (i.e., global history) to be serializable, two conditions are necessary:
  - Each local history should be serializable → local serializability
  - Two conflicting operations should be in the same relative order in all of the local histories where they appear together → global serializability

Global Non-serializability

T_1: Read(x)
x ←x-100
Write(x)
Read(y)
y ←y+100
Write(y)
Commit

T_2: Read(x)
Read(y)
Commit

- *x* stored at Site 1, *y* stored at Site 2
- LH_1, LH_2 are individually serializable (in fact serial), but the two transactions are not globally serializable.

$$LH_1=\{R_1(x),W_1(x), R_2(x)\}$$

$$LH_2=\{R_2(y), R_1(y),W_1(y)\}$$

Concurrency Control Algorithms

- Pessimistic
  - Two-Phase Locking-based (2PL)
    - Centralized (primary site) 2PL
    - Primary copy 2PL
    - Distributed 2PL
  - Timestamp Ordering (TO)
    - Basic TO
    - Multiversion TO
    - Conservative TO
- Optimistic
  - Locking-based
  - Timestamp ordering-based

Locking-Based Algorithms

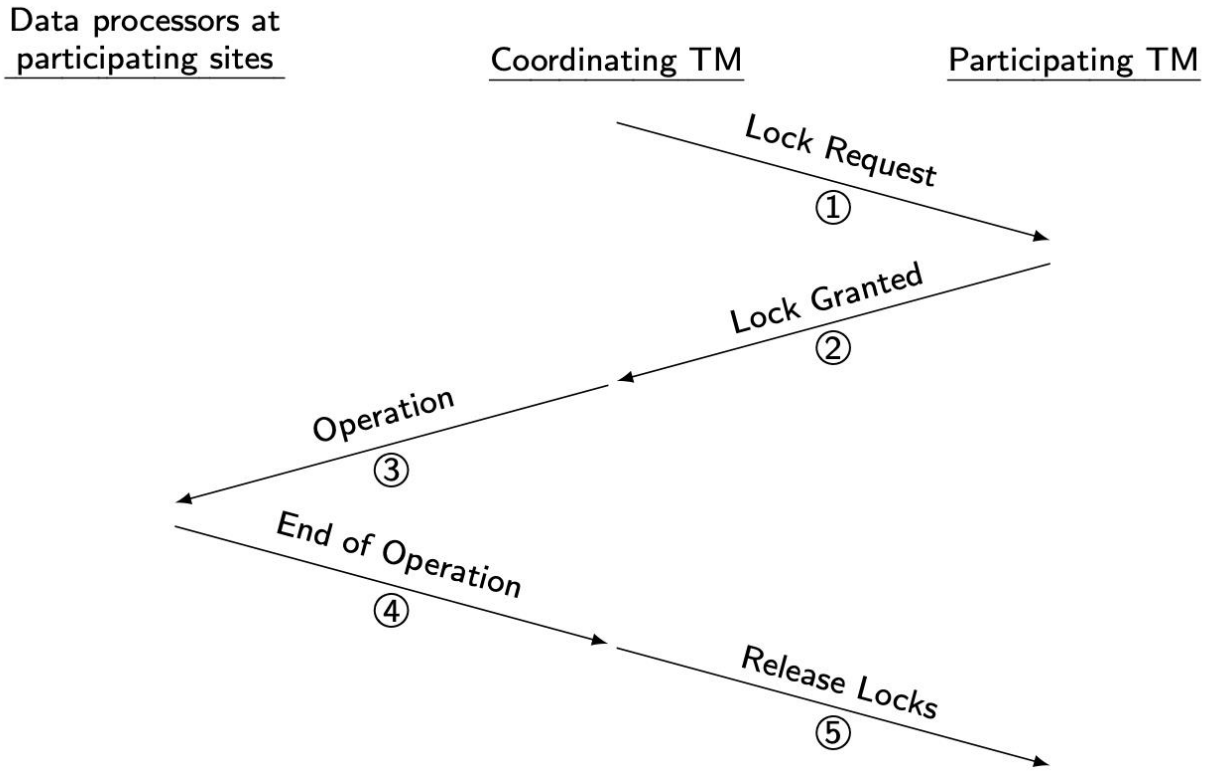
- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager).
- Locks are either read lock (*rl*) [also called shared lock] or write lock (*wl*) [also called exclusive lock]
- Read locks and write locks conflict (because Read and Write operations are incompatible

	rl	*wl*
rl	yes	no
wl	no	no

- Locking works nicely to allow concurrent processing of transactions.

Centralized 2PL

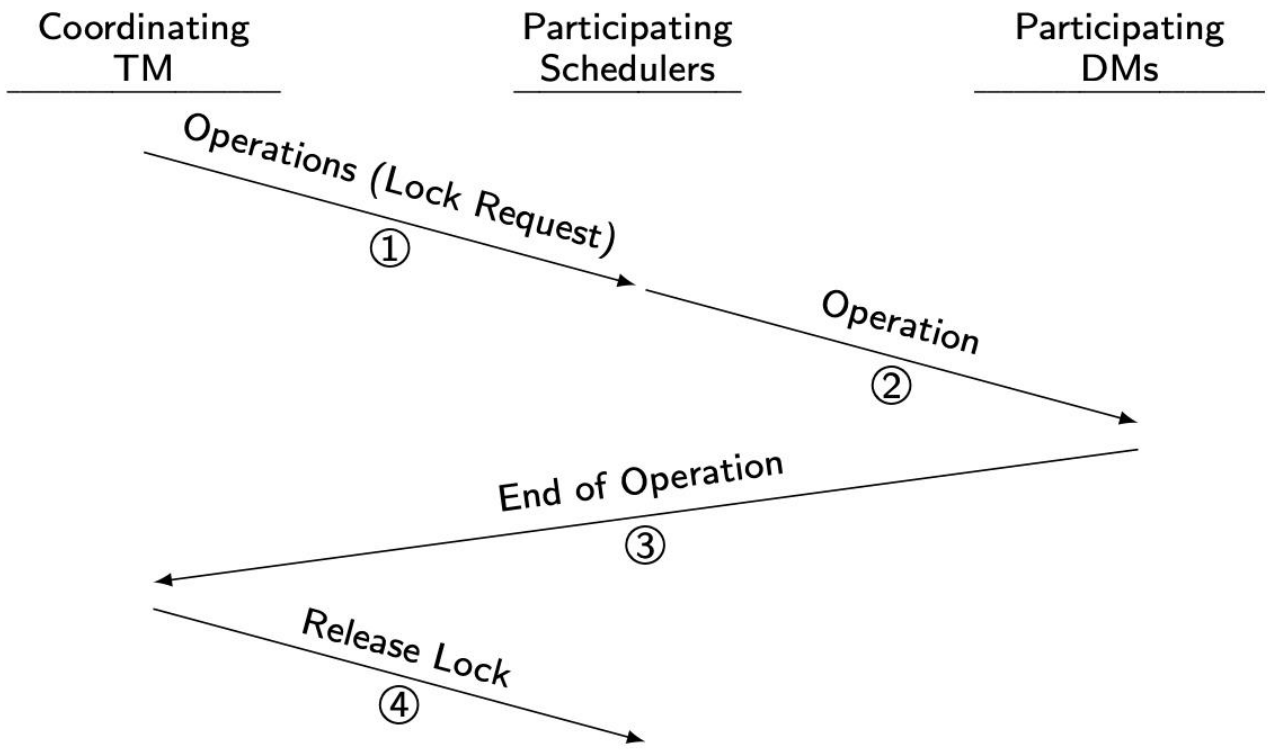
- There is only one 2PL scheduler in the distributed system.
- Lock requests are issued to the central scheduler.



Distributed 2PL

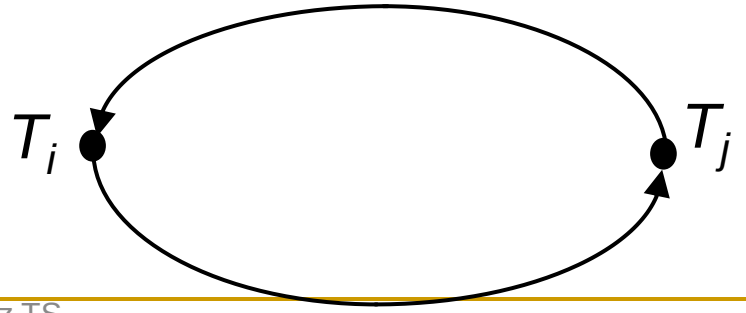
- 2PL schedulers are placed at each site. Each scheduler handles lock requests for data at that site.
- A transaction may read any of the replicated copies of item *x*, by obtaining a read lock on one of the copies of *x*. Writing into *x* requires obtaining write locks for all copies of *x*.

Distributed 2PL Execution



Deadlock

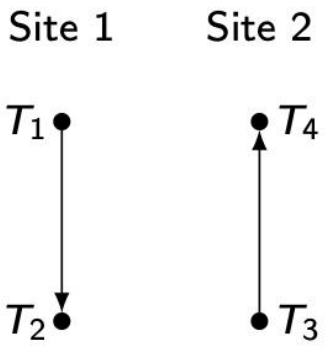
- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- TO-based algorithms that involve waiting may cause deadlocks.
- Wait-for graph
  - If transaction T_i waits for another transaction T_j to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG.



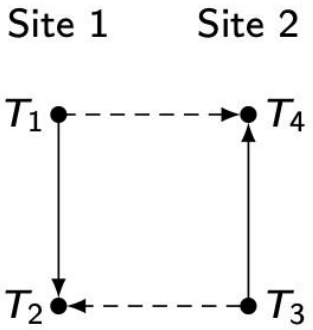
Local versus Global WFG

- T_1 and T_2 run at site 1, T_3 and T_4 run at site 2.
- T_3 waits for a lock held by T_4 which waits for a lock held by T_1 which waits for a lock held by T_2 which, in turn, waits for a lock held by T_3.

Local WFG



Global WFG



Deadlock Detection

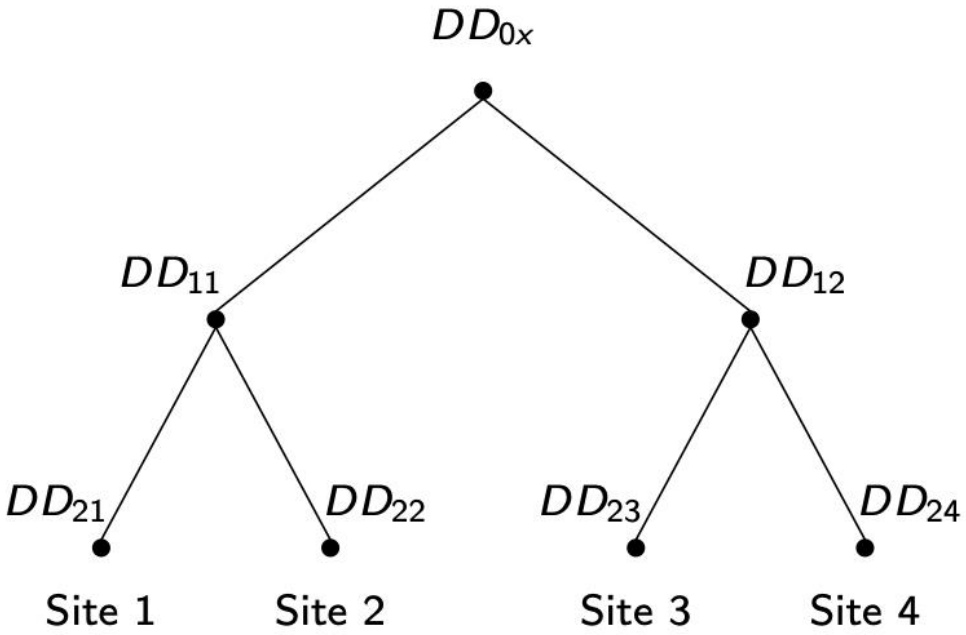
- Transactions are allowed to wait freely.
- Wait-for graphs and cycles.
- Topologies for deadlock detection algorithms
  - Centralized
  - Distributed
  - Hierarchical

Centralized Deadlock Detection

- One site is designated as the deadlock detector for the system. Each scheduler periodically sends its local WFG to the central site which merges them to a global WFG to determine cycles.
- How often to transmit?
  - Too often ⇒ higher communication cost but lower delays due to undetected deadlocks
  - Too late ⇒ higher delays due to deadlocks, but lower communication cost
- Would be a reasonable choice if the concurrency control algorithm is also centralized.
- Proposed for Distributed INGRES

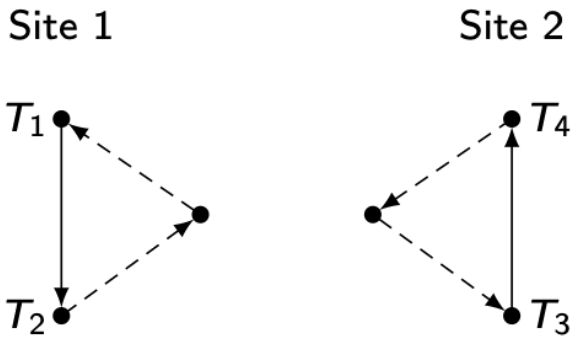
Hierarchical Deadlock Detection

Build a hierarchy of detectors



Distributed Deadlock Detection

- Sites cooperate in detection of deadlocks.
- One example:
  - Form local WFGs at each modified as follows:
    1) Potential deadlock cycles from other sites are added as edges
    2) Join these with regular edges
    3) Pass these local WFGs to other sites



  - Each local deadlock detector:
    - looks for a cycle that does not involve the external edge. If it exists, there is a local deadlock which can be handled locally.
    - looks for a cycle involving the external edge. If it exists, it indicates a potential global deadlock. Pass on the information to the next site.

Timestamp Ordering

1. Transaction (T_i) is assigned a globally unique timestamp $ts(T_i)$.
2. Transaction manager attaches the timestamp to all operations issued by the transaction.
3. Each data item is assigned a write timestamp (*wts*) and a read timestamp (*rts*):
   - $rts(x)$ = largest timestamp of any read on x
   - $wts(x)$ = largest timestamp of any read on x
4. Conflicting operations are resolved by timestamp order.

Basic T/O:

for $R_i(x)$

if $ts(T_i) < wts(x)$
then reject $R_i(x)$
else accept $R_i(x)$
$rts(x) \leftarrow ts(T_i)$

for $W_i(x)$

if $ts(T_i) < rts(x)$ **and** $ts(T_i) < wts(x)$
then reject $W_i(x)$
else accept $W_i(x)$
$wts(x) \leftarrow ts(T_i)$

Basic Timestamp Ordering

Two conflicting operations O_{ij} of T_i and O_{kl} of $T_k \rightarrow O_{ij}$ executed before O_{kl} iff $ts(T_i) < ts(T_k)$.

- T_i is called older transaction
- T_k is called younger transaction

for $R_i(x)$

if $ts(T_i) < wts(x)$
  then reject $R_i(x)$
  else accept $R_i(x)$
$rts(x) \leftarrow ts(T_i)$

for $W_i(x)$

if $ts(T_i) < rts(x)$ **and** $ts(T_i) < wts(x)$
  then reject $W_i(x)$
  else accept $W_i(x)$
$wts(x) \leftarrow ts(T_i)$

Conservative Timestamp Ordering

- Basic timestamp ordering tries to execute an operation as soon as it receives it
  - progressive
  - too many restarts since there is no delaying
- Conservative timestamping delays each operation until there is an assurance that it will not be restarted
- Assurance?
  - No other operation with a smaller timestamp can arrive at the scheduler
  - Note that the delay may result in the formation of deadlocks

Multiversion Concurrency Control (MVCC)

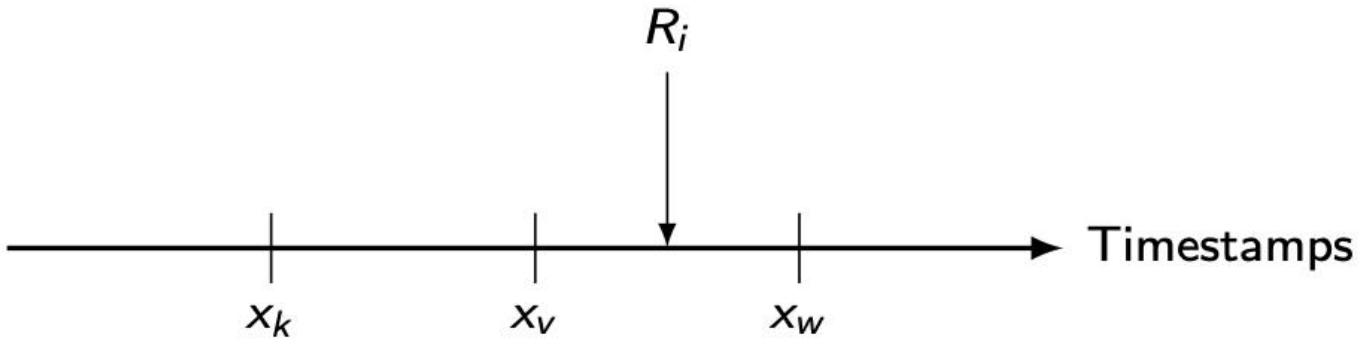
- Do not modify the values in the database, create new values.
- Typically timestamp-based implementation

$$ts(T_i) < ts(x_r) < ts(T_j)$$

- Implemented in a number of systems: IBM DB2, Oracle, SQL Server, SAP HANA, BerkeleyDB, PostgreSQL

MVCC Reads

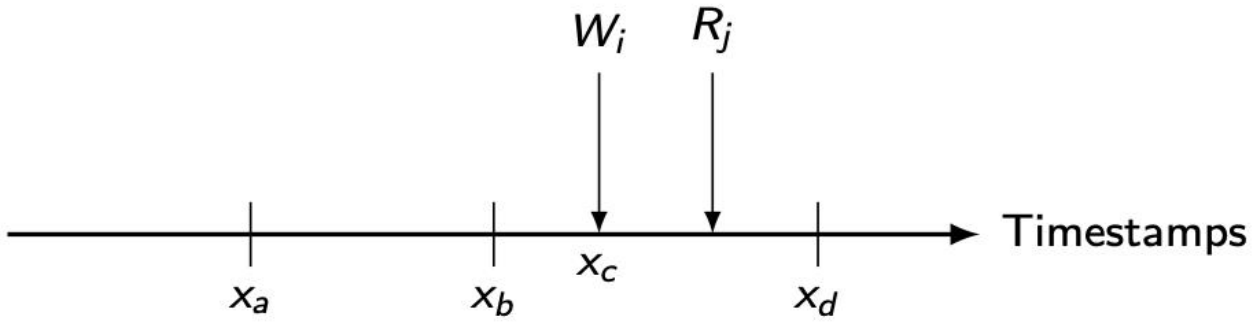
- A $R_i(x)$ is translated into a read on one version of x.
  - Find a version of x (say x_v) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$.



MVCC Writes

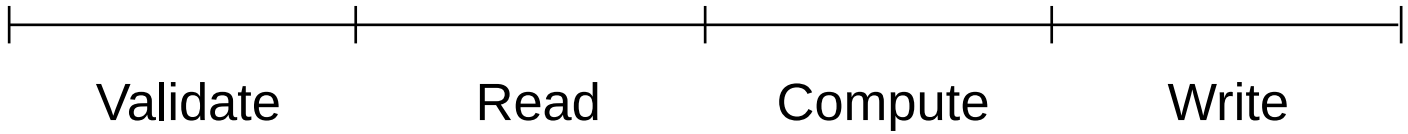
- A $W_i(x)$ is translated into $W_i(x_w)$ and accepted if the scheduler has not yet processed any $R_j(x_r)$ such that

$$ts(T_i) < ts(x_r) < ts(T_j)$$

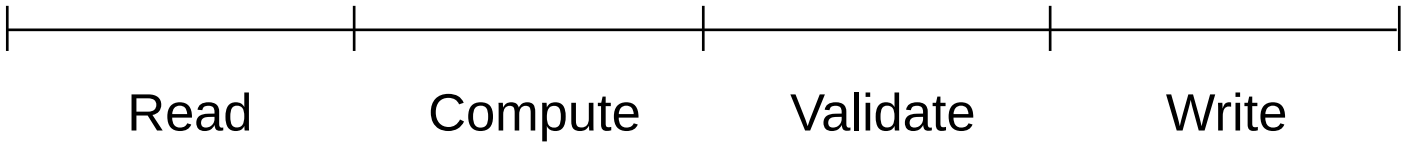


Optimistic Concurrency Control Algorithms

Pessimistic execution



Optimistic execution

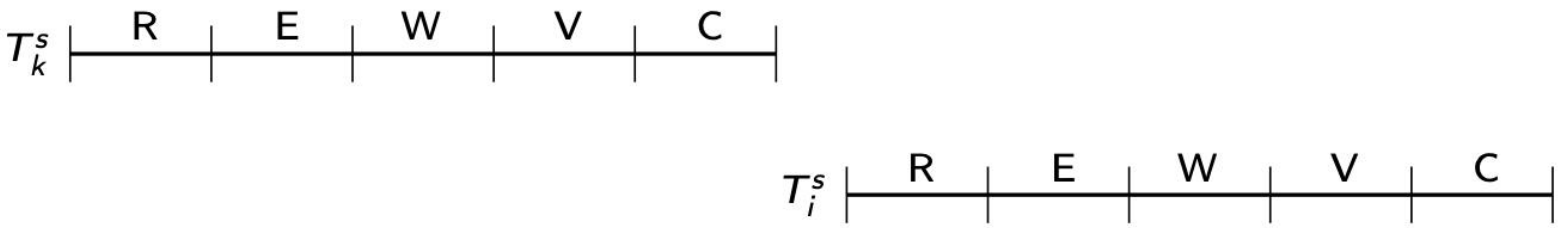


Optimistic Concurrency Control Algorithms

- Transaction execution model: divide into subtransactions each of which execute at a site
  - T_{ij}: transaction T_i that executes at site j
- Transactions run independently at each site until they reach the end of their read phases
- All subtransactions are assigned a timestamp at the end of their read phase
- Validation test performed during validation phase. If one fails, all rejected.

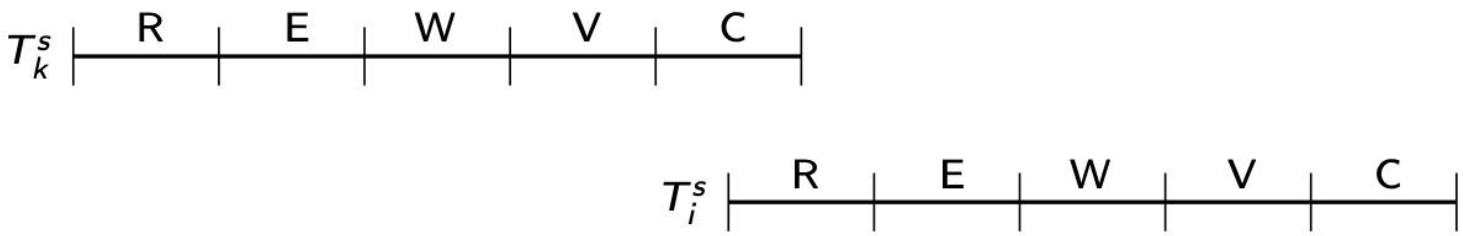
Optimistic CC Validation Test

1. If all transactions T_k where $ts(T_k) < ts(T_{ij})$ have completed their write phase before T_{ij} has started its read phase, then validation succeeds
   - Transaction executions in serial order



Optimistic CC Validation Test

- If there is any transaction T_k such that $ts(T_k)<ts(T_{ij})$ and which completes its write phase while T_{ij} is in its read phase, then validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \emptyset$
  - Read and write phases overlap, but T_{ij} does not read data items written by T_k

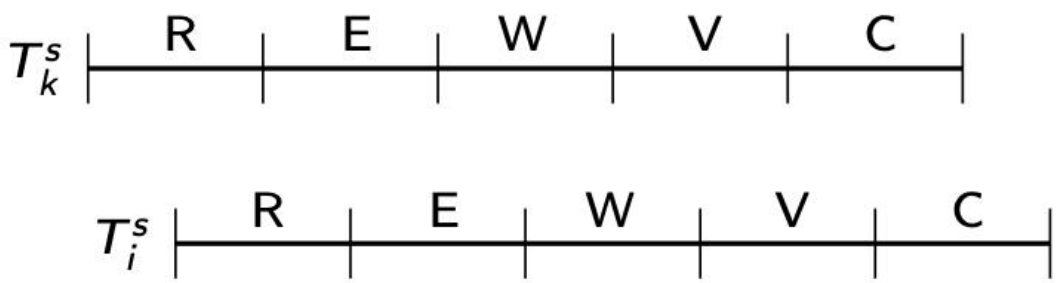


Optimistic CC Validation Test

❸ If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$ and which completes its read phase before T_{ij} completes its read phase, then validation succeeds if

$$WS(T_k) \cap RS(T_{ij}) = \emptyset \text{ and } WS(T_k) \cap WS(T_{ij}) = \emptyset$$

- They overlap, but don't access any common data items.



Snapshot Isolation (SI)

- Each transaction “sees” a consistent snapshot of the database when it starts and R/W this snapshot
- Repeatable reads, but not serializable isolation
- Read-only transactions proceed without significant synchronization overhead
- Centralized SI-based CC
  1) T_i starts, obtains a begin timestamp $ts_b(T_i)$
  2) T_i ready to commit, obtains a commit timestamp $ts_c(T_i)$ that is greater than any of the existing ts_b or ts_c
  3) T_i commits if no other T_j such that tsc(T_i) [$ts_b(T_i)$, $ts_c(T_i)$]; otherwise aborted (first committer wins)
  4) When T_i commits, changes visible to all T_k where $ts_b(T_k) > ts_c(T_i)$

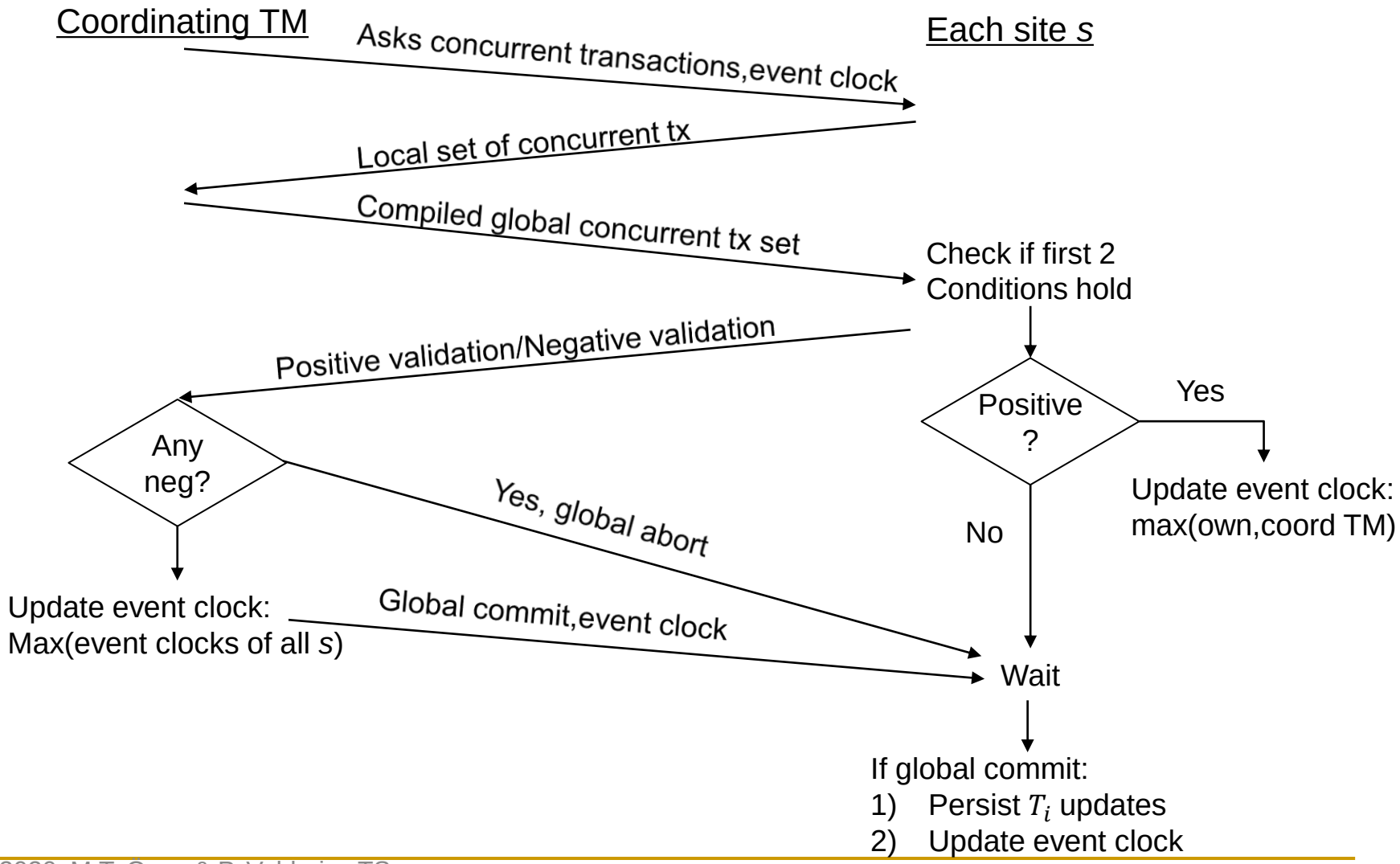
Distributed CC with SI

- Computing a consistent distributed snapshot is hard
- Similar rules to serializability
  - Each local history should be SI
  - Global history is SI → commitment orders at each site are the same
- Dependence relationship: T_i at site s (T_i^s) is dependent on T_j^s (*dependent*(T_i^s, T_j^s)) iff

$$(RS(T_i^s) \cap WS(T_j^s) \neq \emptyset) \vee (WS(T_i^s) \cap RS(T_j^s) \neq \emptyset) \vee (WS(T_i^s) \cap WS(T_j^s) \neq \emptyset)$$

- Conditions
  1) $dependent(T_i, T_j) \wedge ts_b(T_i^s) < ts_c(T_j^s) \Longrightarrow ts_b(T_i^t) < ts_c(T_j^t)$ at every site t where T_i and T_j execute together
  2) $dependent(T_i, T_j) \wedge ts_c(T_i^s) < ts_c(T_j^s) \Longrightarrow ts_c(T_i^t) < ts_b(T_j^t)$ at every site t where T_i and T_j execute together
  3) $ts_c(T_i^s) < ts_c(T_j^s) \Longrightarrow ts_c(T_i^t) < ts_b(T_j^t)$ at every site t where T_i and T_j execute together

Distributed CC with SI – Executing T_i



Outline

- Distributed Transaction Processing
  - Distributed Concurrency Control
  - Distributed Reliability

Reliability

Problem:

How to maintain

atomicity

durability

properties of transactions

Types of Failures

- Transaction failures
  - Transaction aborts (unilaterally or due to deadlock)
- System (site) failures
  - Failure of processor, main memory, power supply, …
  - Main memory contents are lost, but secondary storage contents are safe
  - Partial vs. total failure
- Media failures
  - Failure of secondary storage devices → stored data is lost
  - Head crash/controller failure
- Communication failures
  - Lost/undeliverable messages
  - Network partitioning

Distributed Reliability Protocols

- Commit protocols
  - How to execute commit command for distributed transactions.
  - Issue: how to ensure atomicity and durability?
- Termination protocols
  - If a failure occurs, how can the remaining operational sites deal with it.
  - Non-blocking: the occurrence of failures should not force the sites to wait until the failure is repaired to terminate the transaction.
- Recovery protocols
  - When a failure occurs, how do the sites where the failure occurred deal with it.
  - Independent: a failed site can determine the outcome of a transaction without having to obtain remote information.
- Independent recovery $\Rightarrow$ non-blocking termination

Two-Phase Commit (2PC)

Phase 1 : The coordinator gets the participants ready to write the results into the database

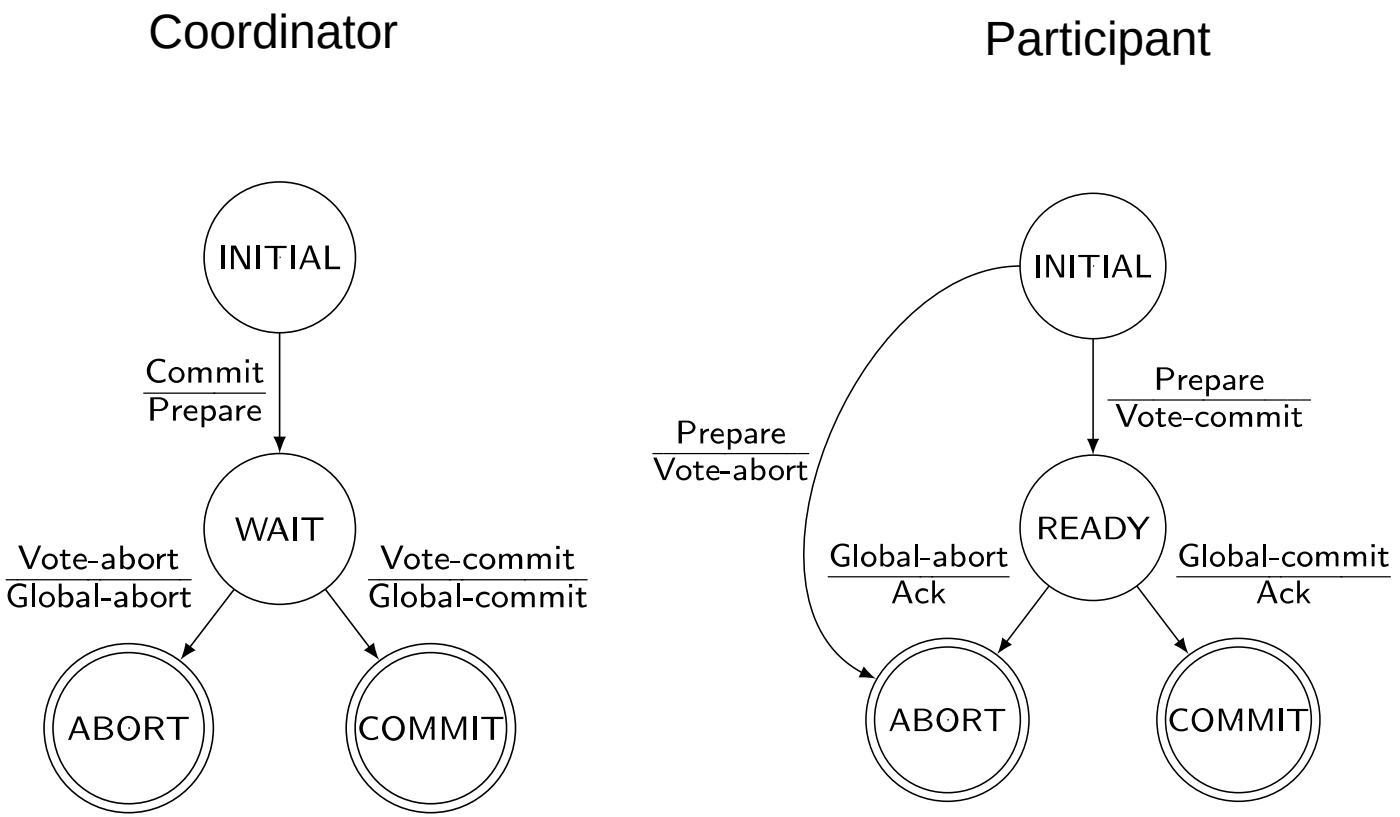
Phase 2 : Everybody writes the results into the database

- **Coordinator** :The process at the site where the transaction originates and which controls the execution
- **Participant** :The process at the other sites that participate in executing the transaction

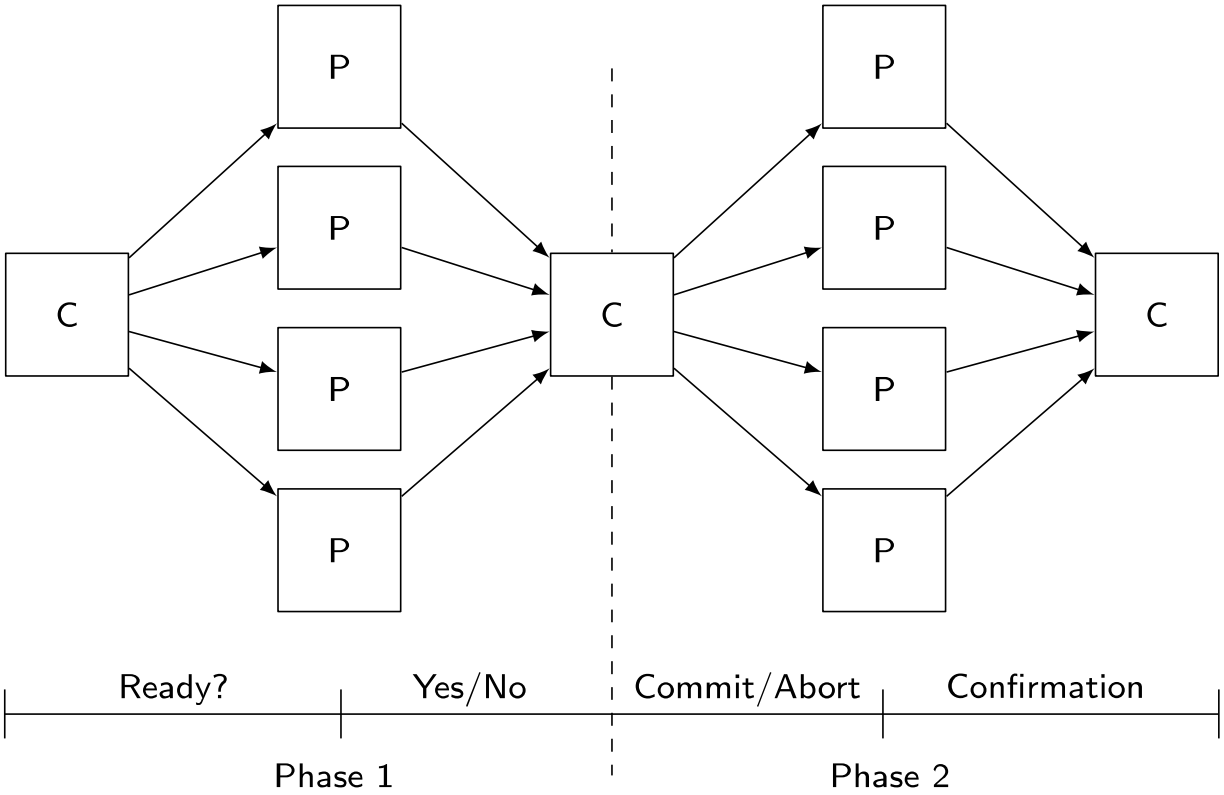
Global Commit Rule:

1. The coordinator aborts a transaction if and only if at least one participant votes to abort it.
2. The coordinator commits a transaction if and only if all of the participants vote to commit it.

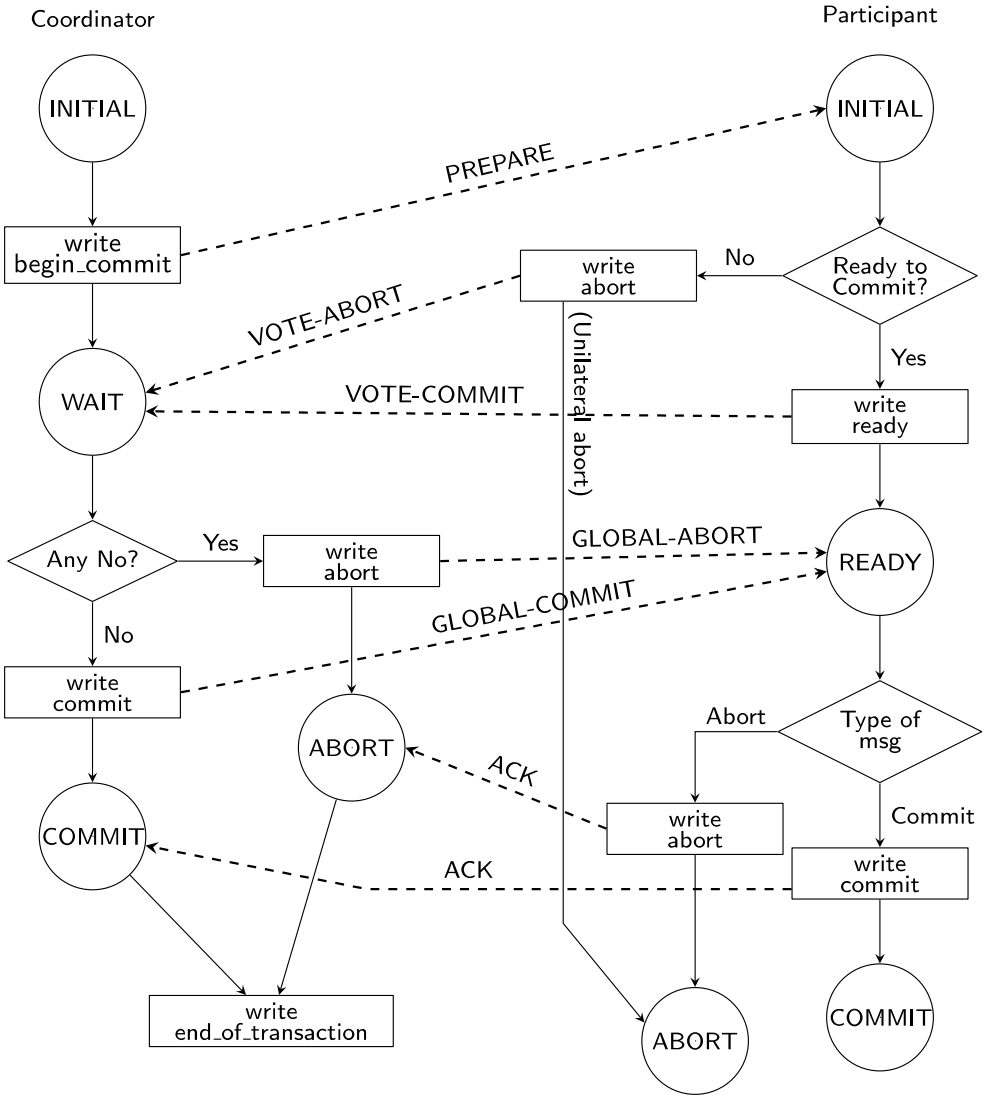
State Transitions in 2PC



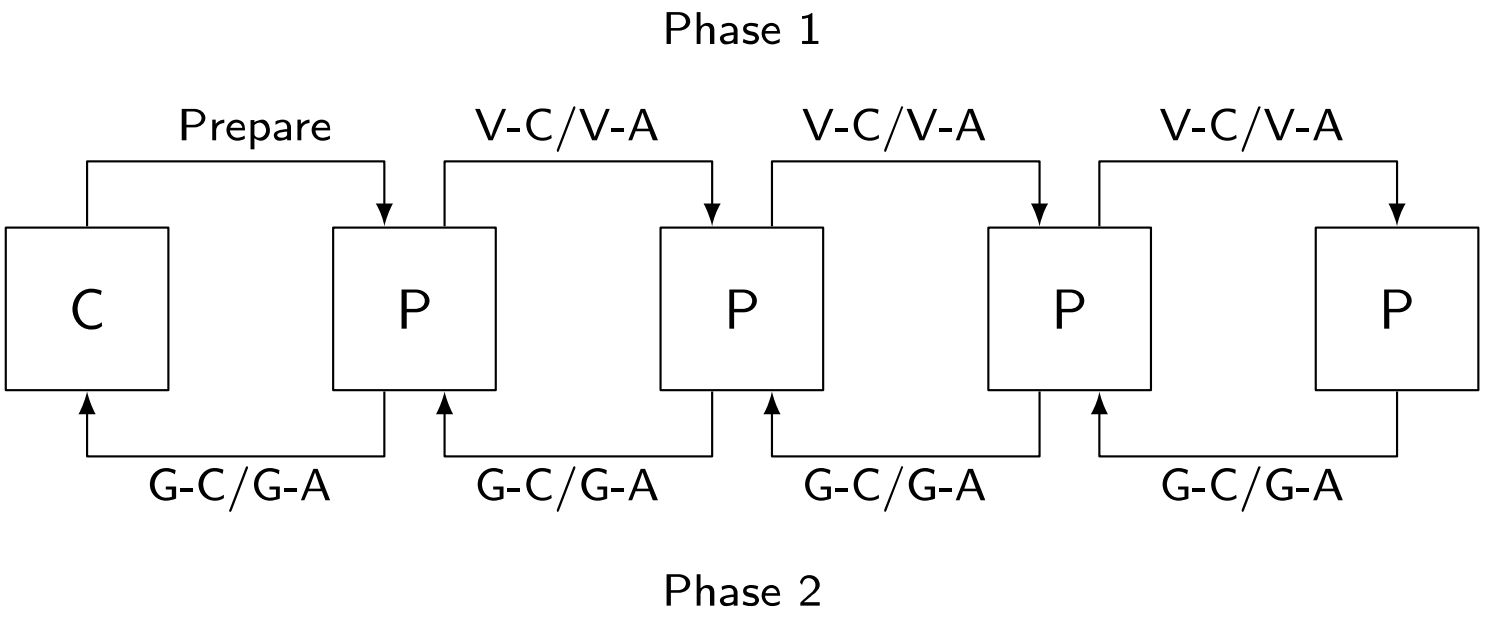
Centralized 2PC



2PC Protocol Actions

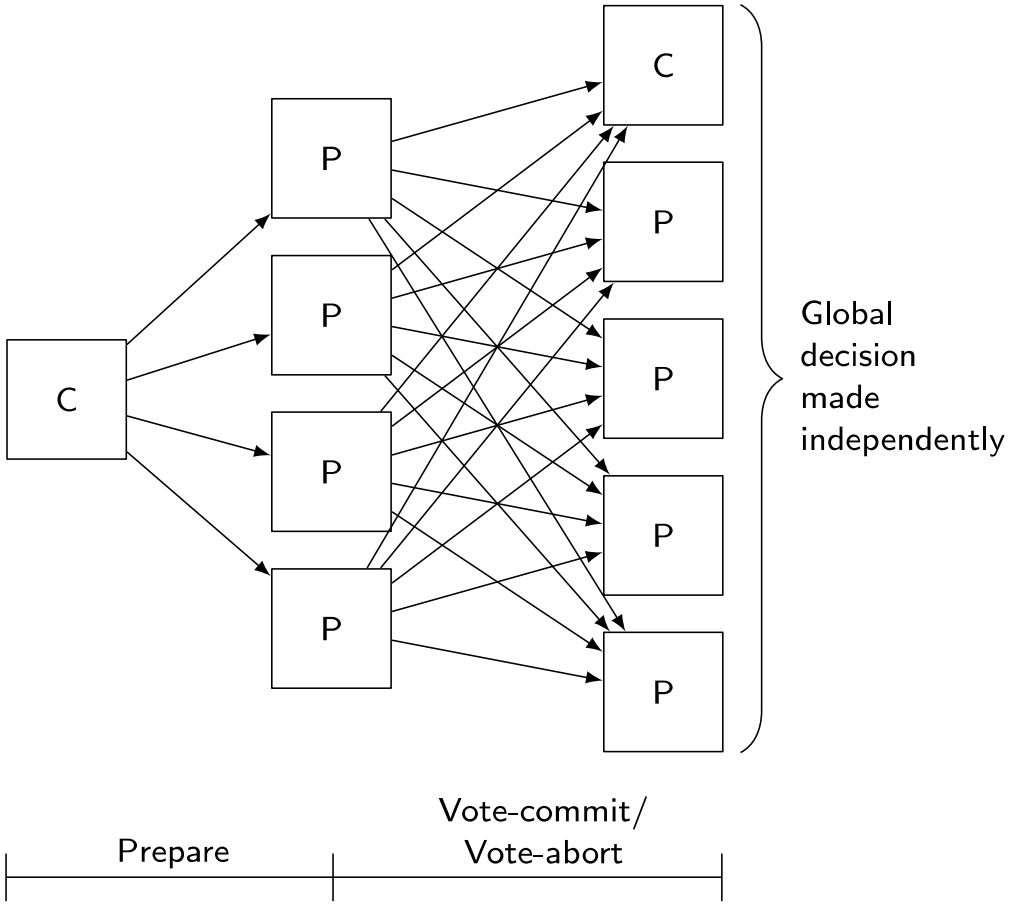


Linear 2PC



V-C: Vote-Commit, V-A: Vote-Abort, G-C: Global-commit, G-A: Global-abort

Distributed 2PC



Variations of 2PC

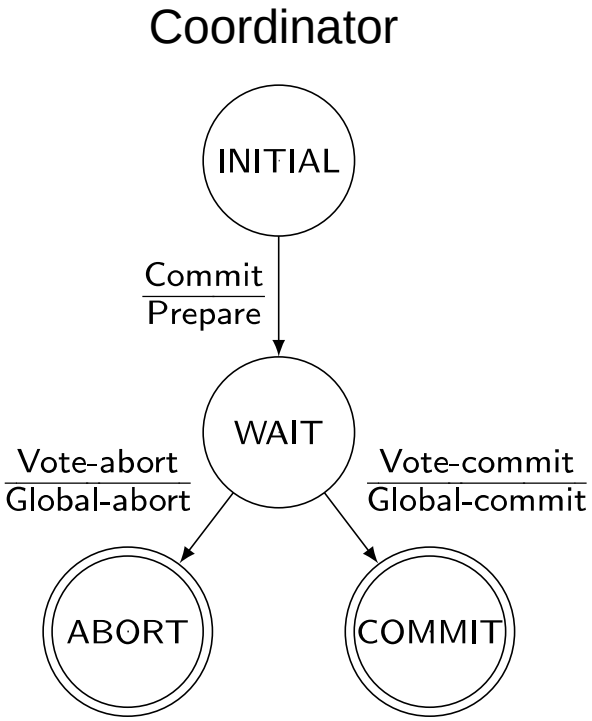
To improve performance by

1) Reduce the number of messages between coordinator & participants
2) Reduce the number of time logs are written

- Presumed Abort 2PC
  - Participant polls coordinator about transaction's outcome
  - No information → abort the transaction
- Presumed Commit 2PC
  - Participant polls coordinator about transaction's outcome
  - No information → assume transaction is committed
  - Not an exact dual of presumed abort 2PC

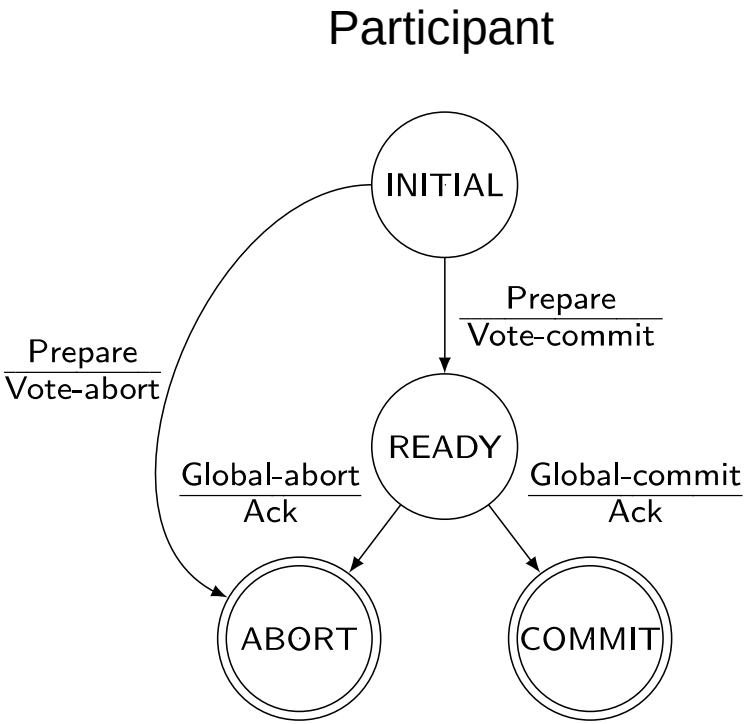
Site Failures - 2PC Termination

- Timeout in INITIAL
  - Who cares
- Timeout in WAIT
  - Cannot unilaterally commit
  - Can unilaterally abort
- Timeout in ABORT or COMMIT
  - Stay blocked and wait for the acks



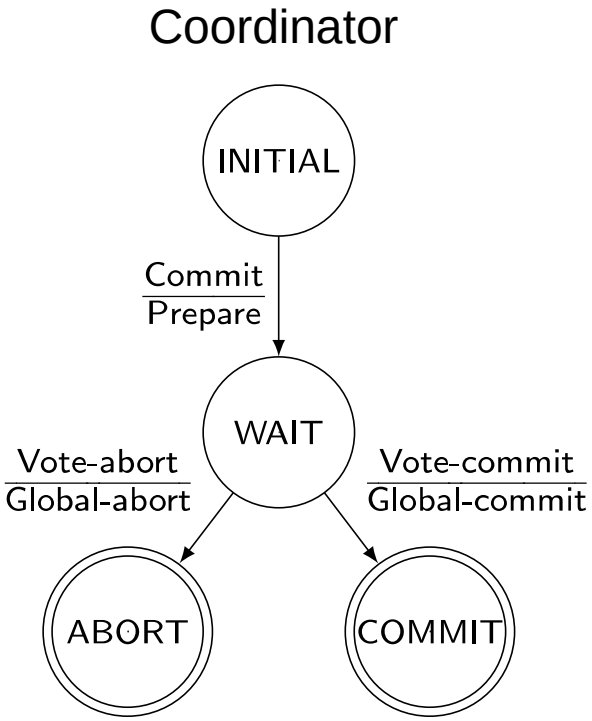
Site Failures - 2PC Termination

- Timeout in INITIAL
  - Coordinator must have failed in INITIAL state
  - Unilaterally abort
- Timeout in READY
  - Stay blocked



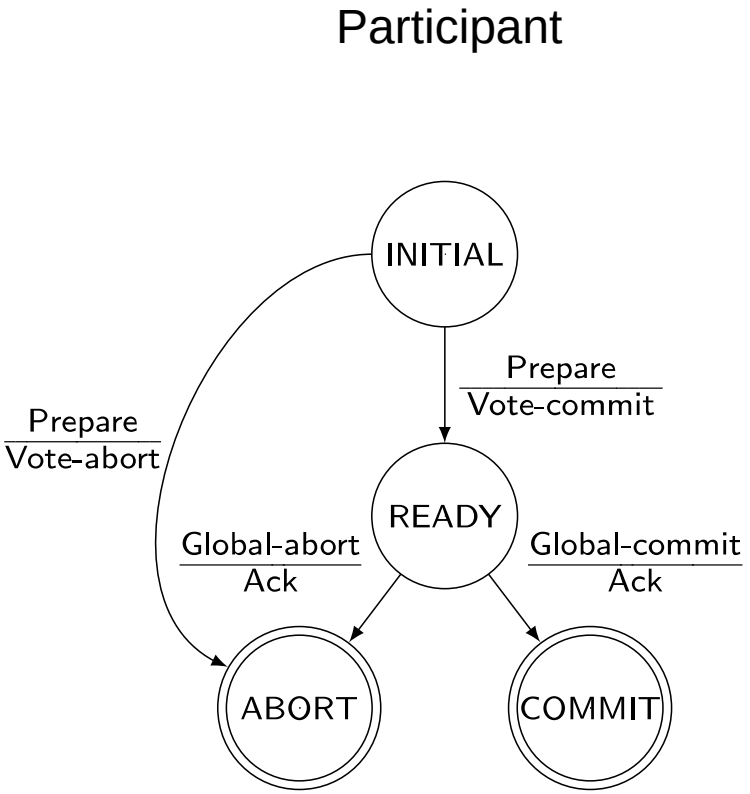
Site Failures - 2PC Recovery

- Failure in INITIAL
  - Start the commit process upon recovery
- Failure in WAIT
  - Restart the commit process upon recovery
- Failure in ABORT or COMMIT
  - Nothing special if all the acks have been received
  - Otherwise the termination protocol is involved



Site Failures - 2PC Recovery

- Failure in INITIAL
  - Unilaterally abort upon recovery
- Failure in READY
  - The coordinator has been informed about the local decision
  - Treat as timeout in READY state and invoke the termination protocol
- Failure in ABORT or COMMIT
  - Nothing special needs to be done



2PC Recovery Protocols – Additional Cases

Arise due to non-atomicity of log and message send actions

- Coordinator site fails after writing “begin_commit” log and before sending “prepare” command
  - treat it as a failure in WAIT state; send “prepare” command
- Participant site fails after writing “ready” record in log but before “vote-commit” is sent
  - treat it as failure in READY state
  - alternatively, can send “vote-commit” upon recovery
- Participant site fails after writing “abort” record in log but before “vote-abort” is sent
  - no need to do anything upon recovery

2PC Recovery Protocols – Additional Case

- Coordinator site fails after logging its final decision record but before sending its decision to the participants
  - coordinator treats it as a failure in COMMIT or ABORT state
  - participants treat it as timeout in the READY state
- Participant site fails after writing “abort” or “commit” record in log but before acknowledgement is sent
  - participant treats it as failure in COMMIT or ABORT state
  - coordinator will handle it by timeout in COMMIT or ABORT state

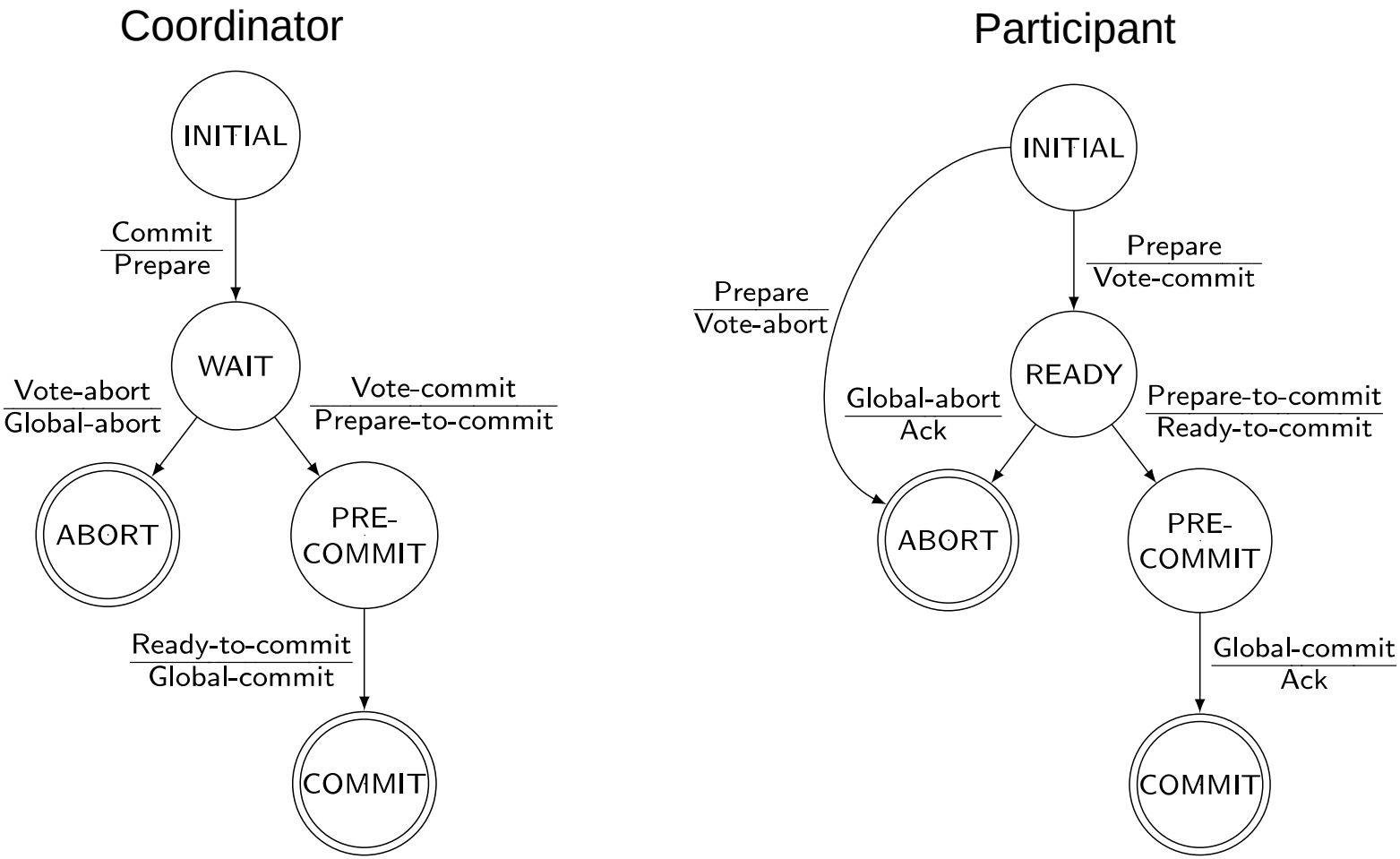
Problem With 2PC

- Blocking
  - Ready implies that the participant waits for the coordinator
  - If coordinator fails, site is blocked until recovery
  - Blocking reduces availability
- Independent recovery is not possible
- However, it is known that:
  - Independent recovery protocols exist only for single site failures; no independent recovery protocol exists which is resilient to multiple-site failures.
- So we search for these protocols – 3PC

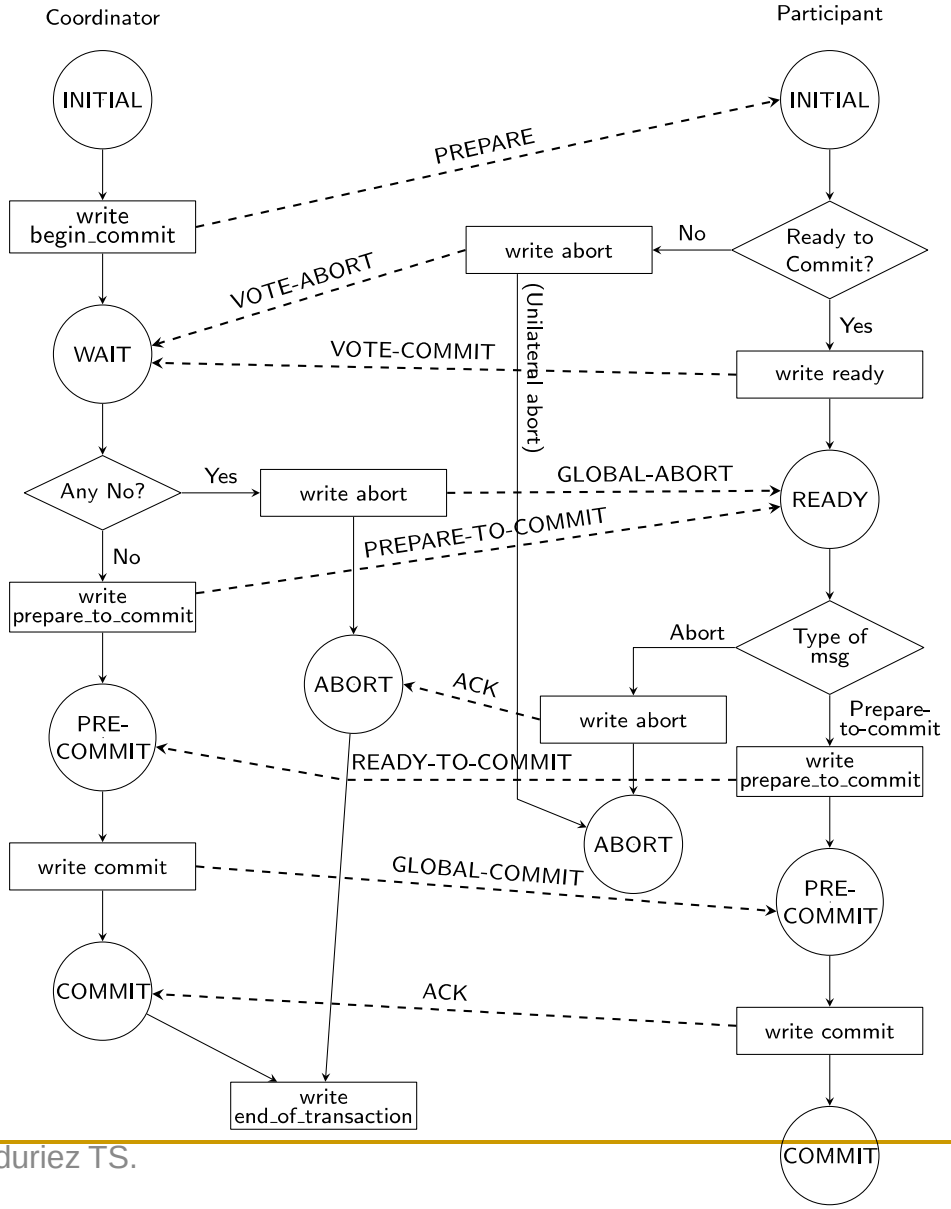
Three-Phase Commit

- 3PC is non-blocking.
- A commit protocols is non-blocking iff
  - it is synchronous within one state transition, and
  - its state transition diagram contains
    - no state which is “adjacent” to both a commit and an abort state, and
    - no non-committable state which is “adjacent” to a commit state
- Adjacent: possible to go from one stat to another with a single state transition
- Committable: all sites have voted to commit a transaction
  - e.g.: COMMIT state

State Transitions in 3PC



3PC Protocol Actions



Network Partitioning

- Simple partitioning
  - Only two partitions
- Multiple partitioning
  - More than two partitions
- Formal bounds:
  - There exists no non-blocking protocol that is resilient to a network partition if messages are lost when partition occurs.
  - There exist non-blocking protocols which are resilient to a single network partition if all undeliverable messages are returned to sender.
  - There exists no non-blocking protocol which is resilient to a multiple partition.

Independent Recovery Protocols for Network Partitioning

- No general solution possible
  - allow one group to terminate while the other is blocked
  - improve availability
- How to determine which group to proceed?
  - The group with a majority
- How does a group know if it has majority?
  - Centralized
    - Whichever partitions contains the central site should terminate the transaction
  - Voting-based (quorum)

Quorum Protocols

- The network partitioning problem is handled by the commit protocol.
- Every site is assigned a vote V_i.
- Total number of votes in the system V
- Abort quorum V_a, commit quorum V_c
  - $V_a + V_c > V$ where $0 \leq V_a , V_c \leq V$
  - Before a transaction commits, it must obtain a commit quorum V_c
  - Before a transaction aborts, it must obtain an abort quorum V_a

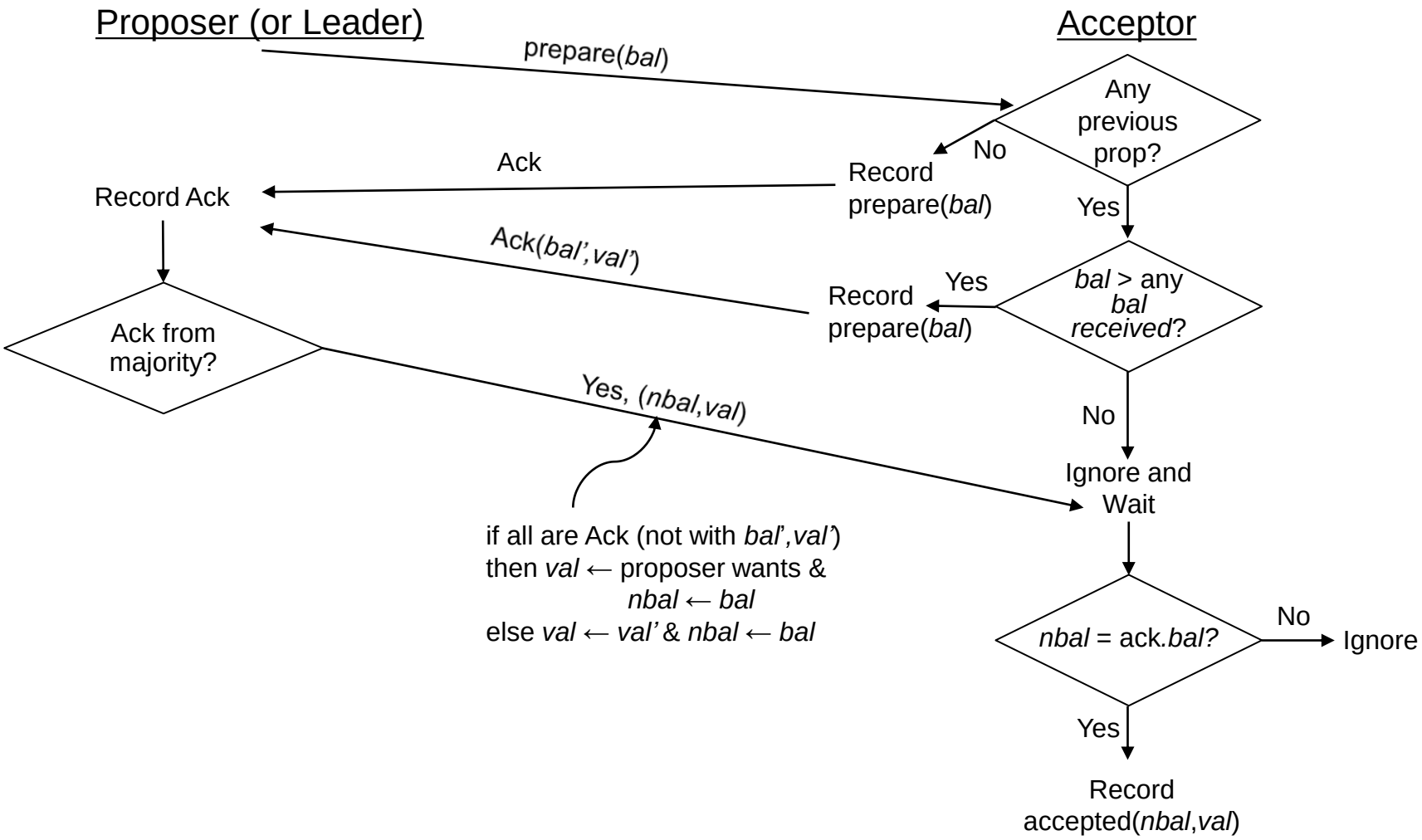
Paxos Consensus Protocol

- General problem: how to reach an agreement (consensus) among TMs about the fate of a transaction
  - 2PC and 3PC are special cases
- General idea: If a majority reaches a decision, the global decision is reached (like voting)
- Roles:
  - Proposer: recommends a decision
  - Acceptor: decides whether to accept the proposed decision
  - Learner: discovers the agreed-upon decision by asking or it is pushed

Paxos & Complications

- Naïve Paxos: one proposer
  - Operates like a 2PC
- Complications
  - Multiple proposers can exist at the same time; acceptor has to choose
    - Attach a ballot number
  - Multiple proposals may result in split votes with no majority
    - Run multiple consensus rounds → performance implication
    - Choose a leader
  - Some accepts fail after they accept a decision; the remaining acceptors may not constitute majority
    - Use ballot numbers

Basic Paxos – No Failures



Basic Paxos with Failures

- Some acceptors fail but there is quorum
  - Not a problem
- Enough acceptors fail to eliminate quorum
  - Run a new ballot
- Proposer/leader fails
  - Choose a new leader and start a new ballot