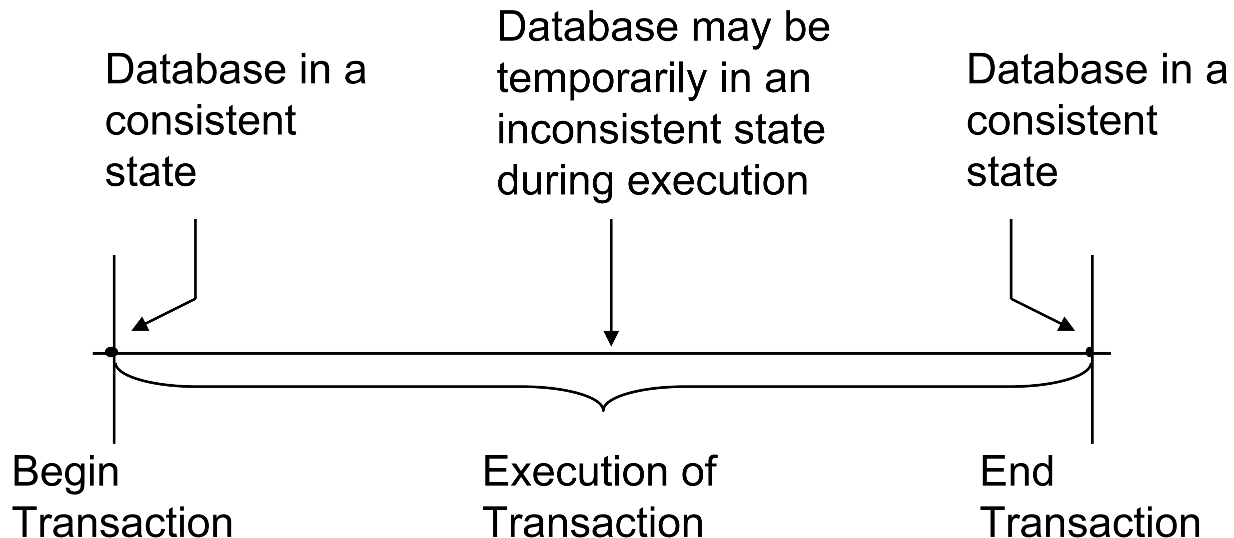
Outline

- Introduction
- Distributed DBMS Architecture
- Distributed Database Design
- Distributed Query Processing
- Distributed Concurrency Control
  - Transaction Concepts & Models
  - Serializability
  - Distributed Concurrency Control Protocols
- Distributed Reliability Protocols

Transaction

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

- concurrency transparency
- failure transparency



Example Database

Consider an airline reservation example with the relations:

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)

CUST(CNAME, ADDR, BAL)

FC(FNO, DATE, CNAME,SPECIAL)

Example Transaction

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);
    EXEC SQL  UPDATE    FLIGHT
              SET       STSOLD = STSOLD + 1
              WHERE     FNO = flight_no AND DATE = date;
    EXEC SQL  INSERT
              INTO      FC(FNO, DATE, CNAME, SPECIAL);
              VALUES    (flight_no, date, customer_name, null);
    output("reservation completed")
end . {Reservation}
```

Termination of Transactions

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);
    EXEC SQL    SELECT      STSOLD,CAP
                INTO        temp1,temp2
                FROM        FLIGHT
                WHERE       FNO = flight_no AND DATE =  date;
    if temp1 = temp2 then
        output("no free seats");
        Abort
    else
        EXEC SQL    UPDATE  FLIGHT
                    SET     STSOLD = STSOLD + 1
                    WHERE   FNO = flight_no AND DATE = date;
        EXEC SQL    INSERT
                    INTO    FC(FNO, DATE, CNAME, SPECIAL);
                    VALUES  (flight_no, date, customer_name, null);
      Commit
      output("reservation completed")
   endif
end . {Reservation}
```

Properties of Transactions

ATOMICITY

- all or nothing

CONSISTENCY

- no violation of integrity constraints

ISOLATION

- concurrent changes invisible È 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)

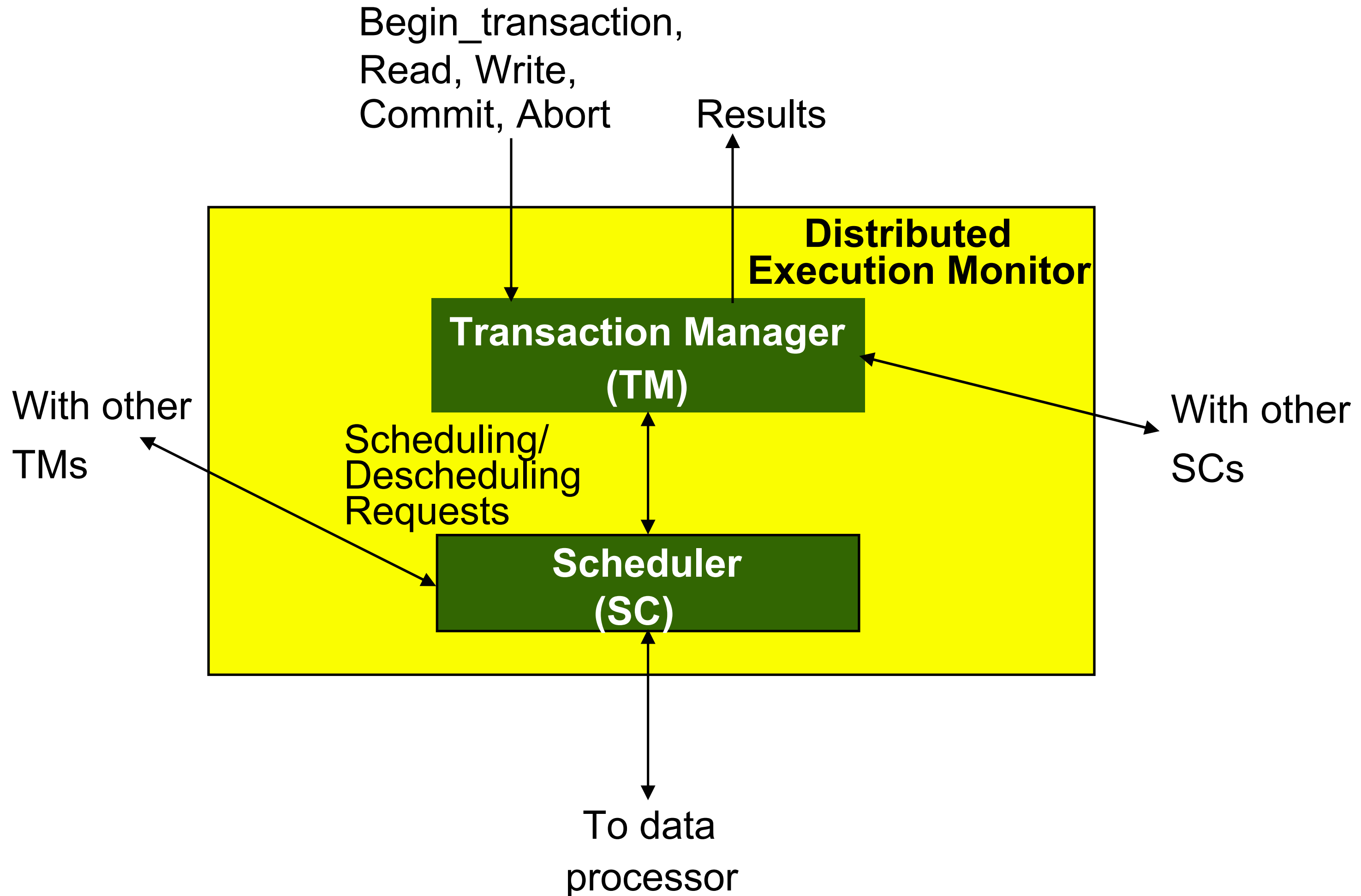
Transaction Processing Issues

- Transaction structure (usually called transaction model)
  - Flat (simple), nested
- Internal database consistency
  - Semantic data control (integrity enforcement) algorithms
- Reliability protocols
  - Atomicity & Durability
  - Local recovery protocols
  - Global commit protocols

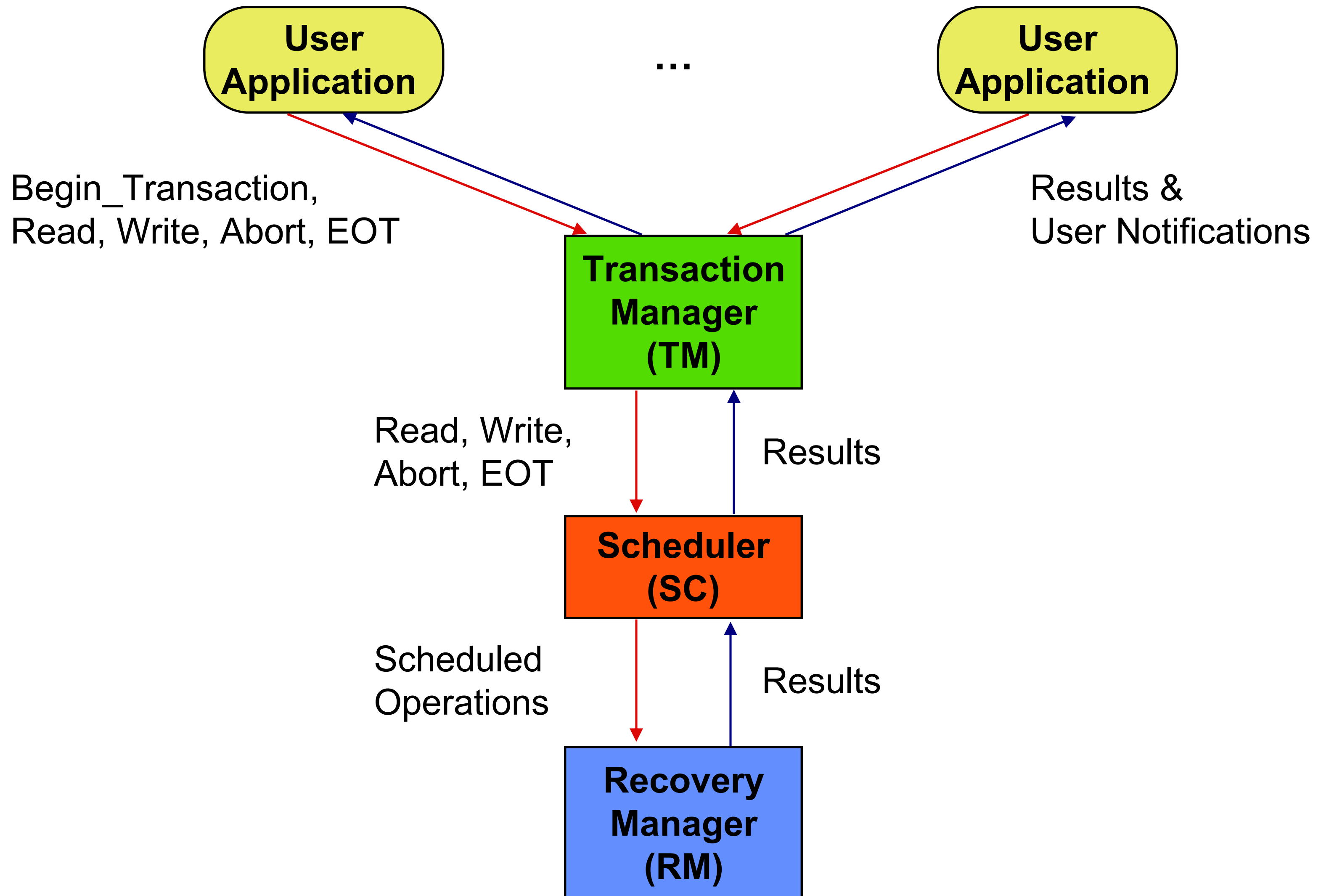
Transaction Processing Issues

- Concurrency control algorithms
  - How to synchronize concurrent transaction executions (correctness criterion)
  - Intra-transaction consistency, Isolation
- Replica control protocols
  - How to control the mutual consistency of replicated data
  - One copy equivalence and ROWA

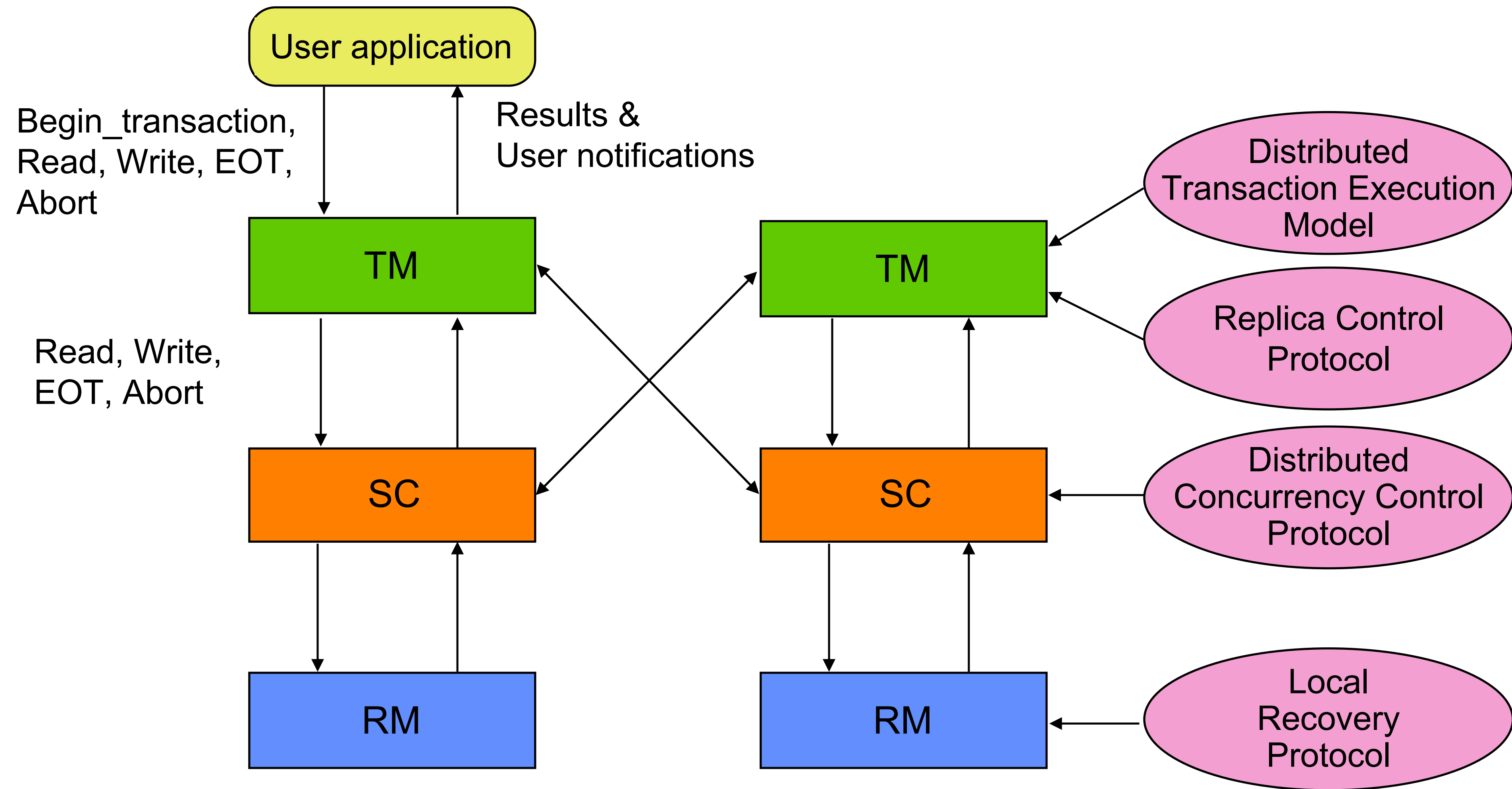
Architecture Revisited



Centralized Transaction Execution



Distributed Transaction Execution



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.
- 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.

Execution Schedule (or History)

- An order in which the operations of a set of transactions are executed.
- A schedule (history) can be defined as a partial order over the operations of a set of transactions.

T_1:	T_2:	T_3:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

$$H_1=\{W_2(x),R_1(x), R_3(x),W_1(x),C_1,W_2(y),R_3(y),R_2(z),C_2,R_3(z),C_3\}$$

Serial History

- All the actions of a transaction occur consecutively.
- No interleaving of transaction operations.
- If each transaction is consistent (obeys integrity rules), then the database is guaranteed to be consistent at the end of executing a serial history.

T_1: Read(x)	T_2: Write(x)	T_3: Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

$$H_s=\{W_2(x),W_2(y),R_2(z),C_2,R_1(x),W_1(x),C_1,R_3(x),R_3(y),R_3(z),C_3\}$$

Serializable History

- Transactions execute concurrently, but the net effect of the resulting history upon the database is *equivalent* to some *serial* history.
- Equivalent with respect to what?
  - ***Conflict equivalence***: the relative order of execution of the conflicting operations belonging to unaborted transactions in two histories are the same.
  - ***Conflicting operations***: two incompatible operations (e.g., Read and Write) conflict if they both access the same data item.
    - Incompatible operations of each transaction is assumed to conflict; do not change their execution orders.
    - If two operations from two different transactions conflict, the corresponding transactions are also said to conflict.

Serializability in Distributed DBMS

- Somewhat more involved. 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.
  - Two conflicting operations should be in the same relative order in all of the local histories where they appear together.

Global Non-serializability

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x+5$		$x \leftarrow x*15$
	Write(x)		Write(x)
	Commit		Commit

The following two local histories are individually serializable (in fact serial), but the two transactions are not globally serializable.

$$LH_1 = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$$

$$LH_2 = \{R_2(x), W_2(x), C_2, R_1(x), W_1(x), C_1\}$$

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
  - Hybrid
- Optimistic
  - Locking-based
  - Timestamp ordering-based

Locking-Based Algorithms

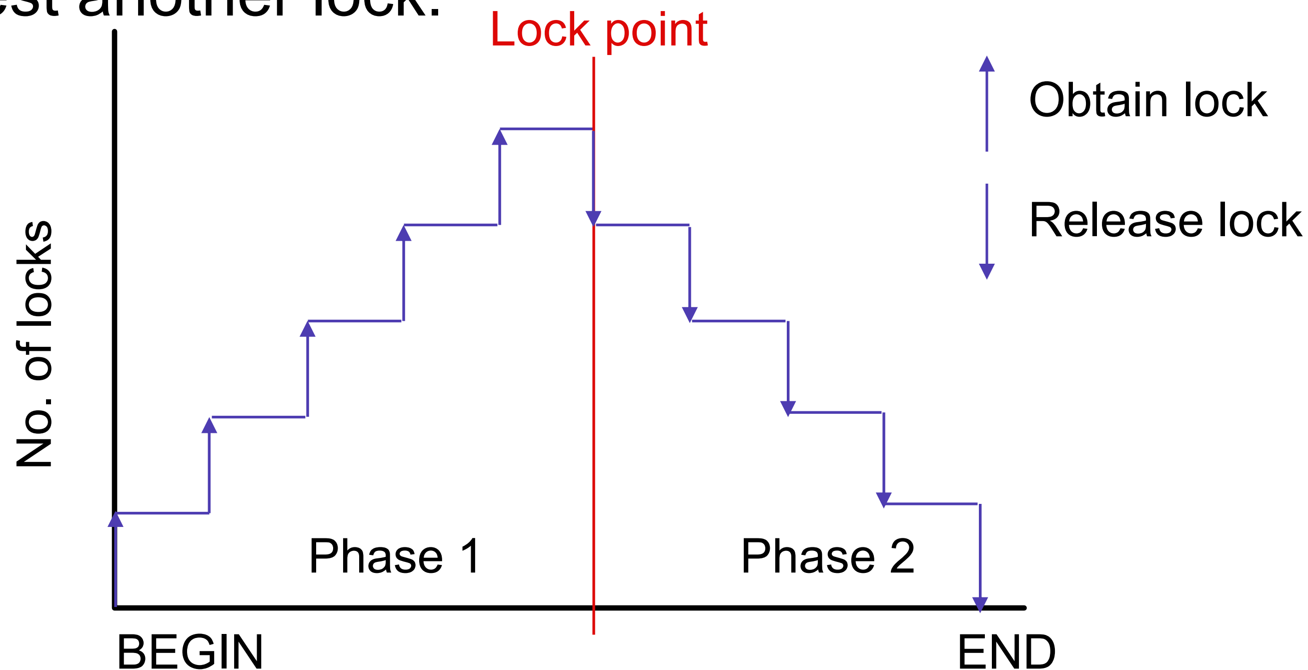
- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager).
- Locks are either read lock (*rl*) [also called shared lock] or write lock (*wl*) [also called exclusive lock]
- Read locks and write locks conflict (because Read and Write operations are incompatible

	rl	*wl*
rl	yes	no
wl	no	no

- Locking works nicely to allow concurrent processing of transactions.

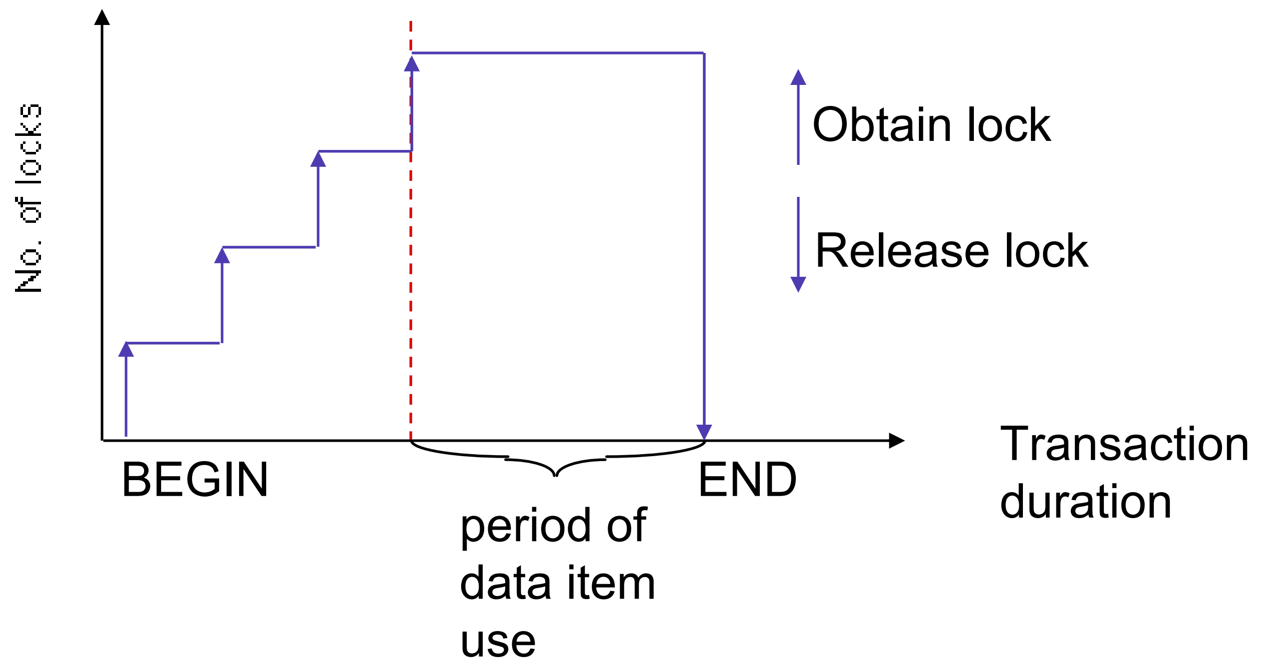
Two-Phase Locking (2PL)

1. A Transaction locks an object before using it.
2. When an object is locked by another transaction, the requesting transaction must wait.
3. When a transaction releases a lock, it may not request another lock.



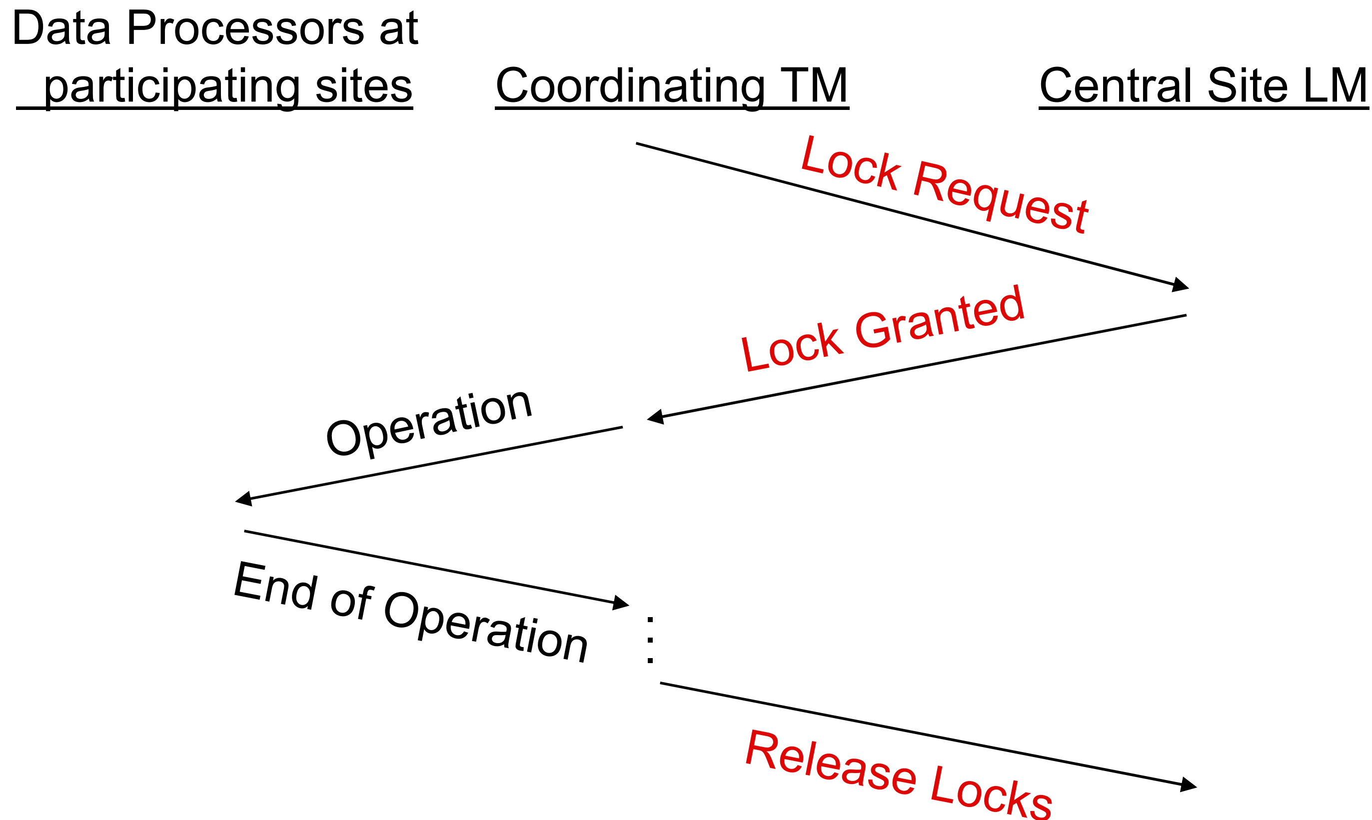
Strict 2PL

Hold locks until the end.



Centralized 2PL

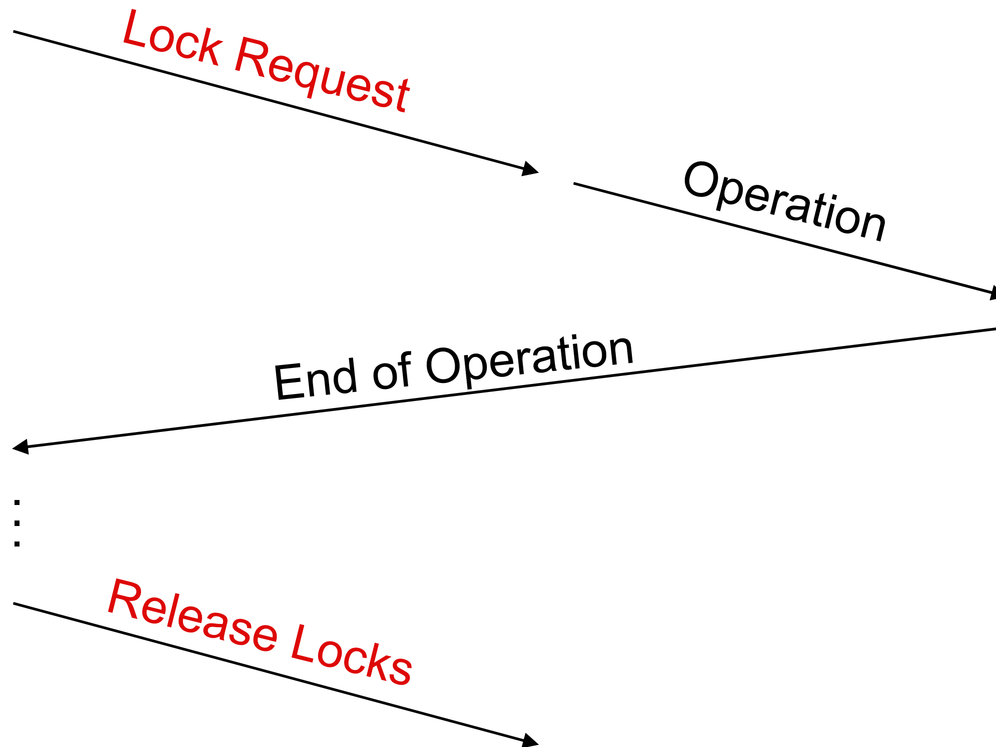
- There is only one 2PL scheduler in the distributed system.
- Lock requests are issued to the central scheduler.



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



Timestamp Ordering

❶ Transaction (T_i) is assigned a globally unique timestamp $ts(T_i)$.

❷ Transaction manager attaches the timestamp to all operations issued by the transaction.

❸ 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

❹ 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 $W_i(x)$
$wts(x) \leftarrow ts(T_i)$

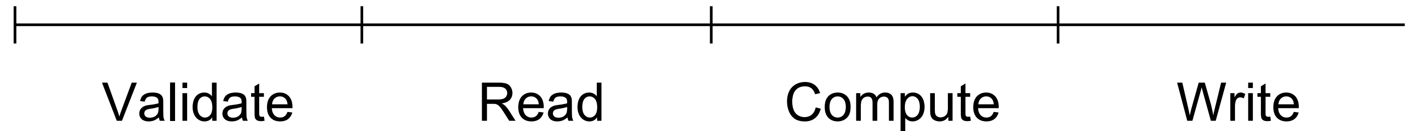
Multiversion Timestamp Ordering

- Do not modify the values in the database, create new values.
- 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 $ts(T_i)$.
- 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

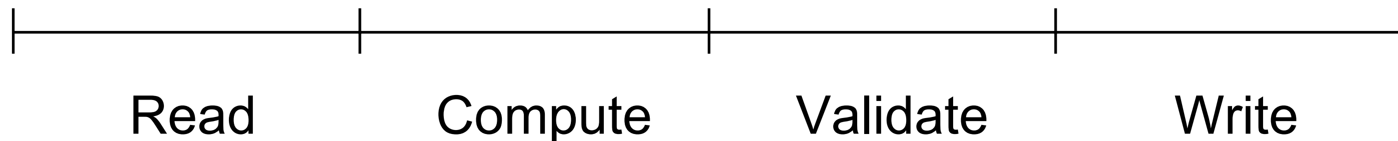
$$ts(T_i) < ts(x_r) < ts(T_j)$$

Optimistic Concurrency Control Algorithms

Pessimistic execution



Optimistic execution



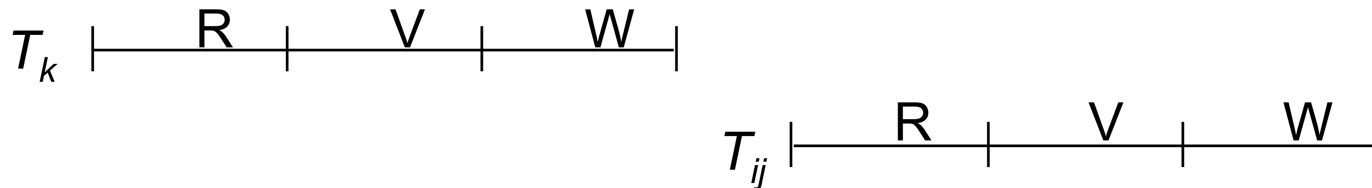
Optimistic Concurrency Control Algorithms

- 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

❶ 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

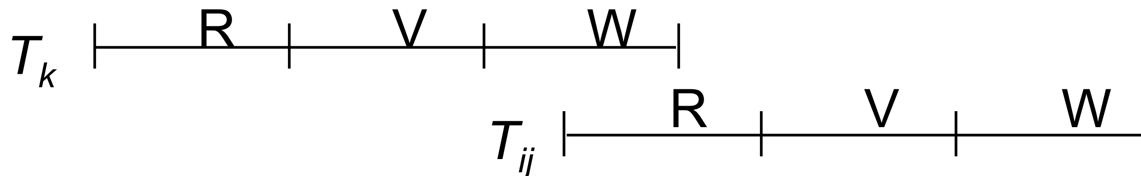
- Transaction executions in serial order



Optimistic CC Validation Test

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

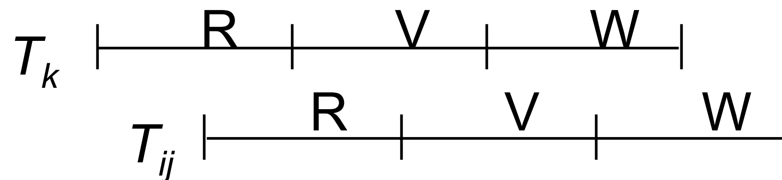
- Read and write phases overlap, but T_{ij} does not read data items written by T_k



Optimistic CC Validation Test

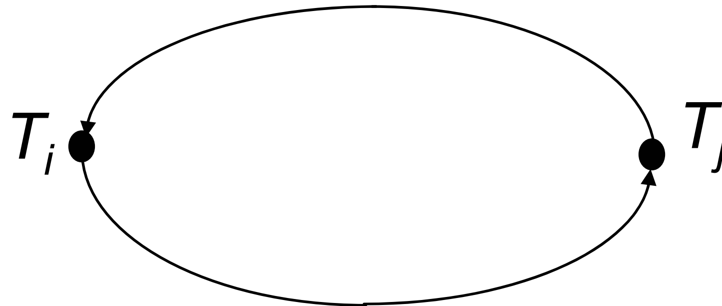
❸ If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$ and which completes its read phase before T_{ij} completes its read phase, then validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \varnothing$ and $WS(T_k) \cap WS(T_{ij}) = \varnothing$

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



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

Assume T_1 and T_2 run at site 1, T_3 and T_4 run at site 2. Also assume 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 Management

- Prevention
  - Guaranteeing that deadlocks can never occur in the first place. Check transaction when it is initiated. Requires no run time support.
- Avoidance
  - Detecting potential deadlocks in advance and taking action to insure that deadlock will not occur. Requires run time support.
- Detection and Recovery
  - Allowing deadlocks to form and then finding and breaking them. As in the avoidance scheme, this requires run time support.

Deadlock Prevention

- All resources which may be needed by a transaction must be predeclared.
  - The system must guarantee that none of the resources will be needed by an ongoing transaction.
  - Resources must only be reserved, but not necessarily allocated a priori
  - Unsuitability of the scheme in database environment
  - Suitable for systems that have no provisions for undoing processes.
- Evaluation:
  - – Reduced concurrency due to preallocation
  - – Evaluating whether an allocation is safe leads to added overhead.
  - – Difficult to determine (partial order)
  - \+ No transaction rollback or restart is involved.

Deadlock Avoidance

- Transactions are not required to request resources a priori.
- Transactions are allowed to proceed unless a requested resource is unavailable.
- In case of conflict, transactions may be allowed to wait for a fixed time interval.
- Order either the data items or the sites and always request locks in that order.
- More attractive than prevention in a database environment.

Deadlock Detection

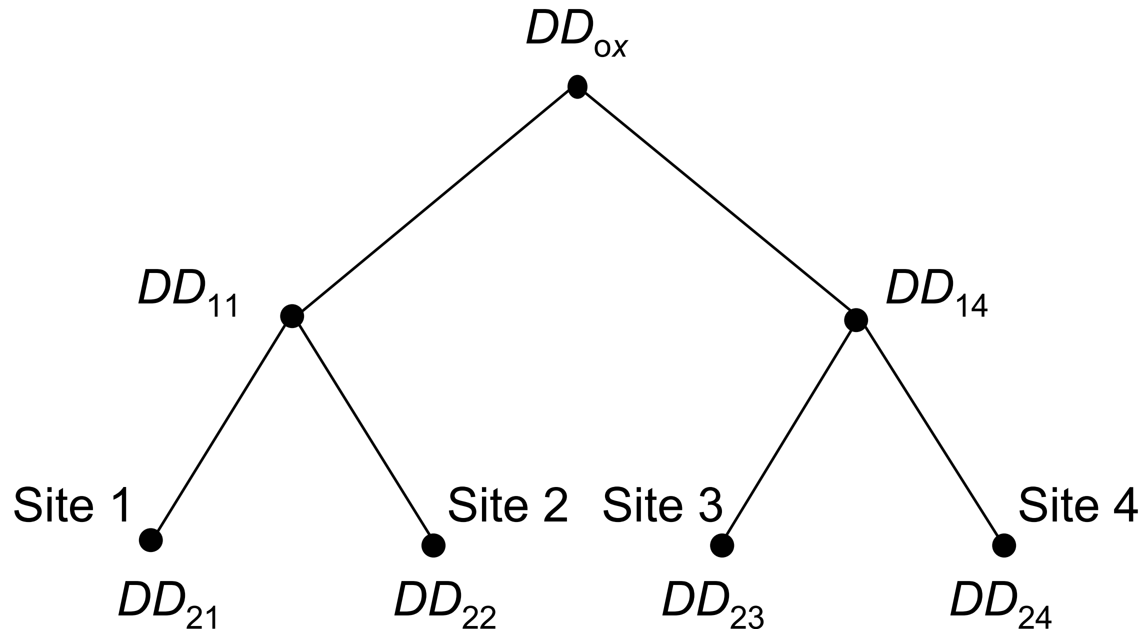
- Transactions are allowed to wait freely.
- Wait-for graphs and cycles.
- Topologies for deadlock detection algorithms
  - Centralized
  - Distributed
  - Hierarchical

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 $\Rightarrow$ higher communication cost but lower delays due to undetected deadlocks
  - Too late $\Rightarrow$ 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:
  - The local WFGs are formed at each site and passed on to other sites. Each local WFG is modified as follows:
    1. Since each site receives the potential deadlock cycles from other sites, these edges are added to the local WFGs
    2. The edges in the local WFG which show that local transactions are waiting for transactions at other sites are joined with edges in the local WFGs which show that remote transactions are waiting for local ones.
  - 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.

Outline

- Introduction
- Distributed DBMS Architecture
- Distributed Database Design
- Distributed Query Processing
- Distributed Concurrency Control
- Distributed Reliability Protocols
  - Distributed Commit Protocols
  - Distributed Recovery Protocols

Reliability

Problem:

How to maintain

atomicity

durability

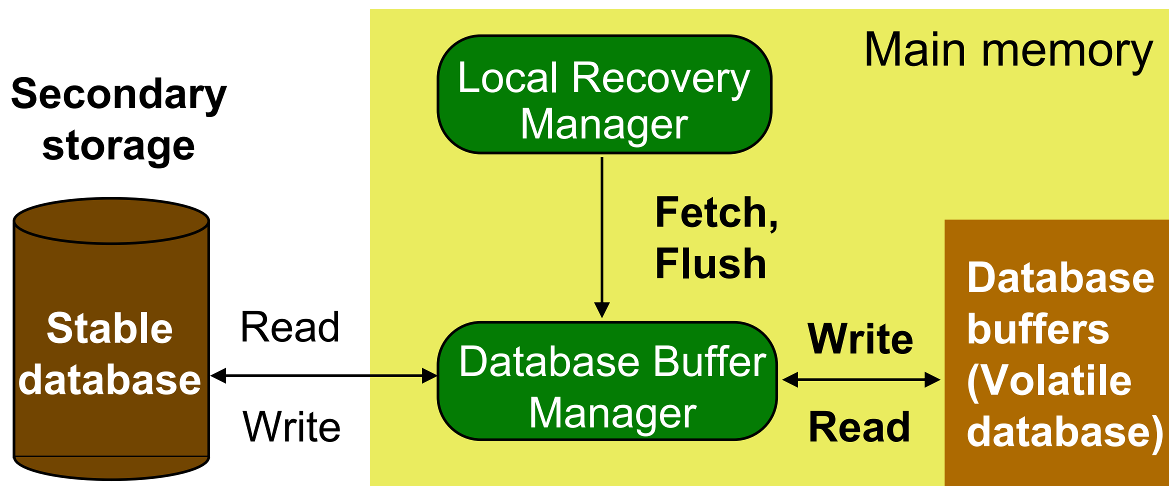
properties of transactions

Types of Failures

- Transaction failures
  - Transaction aborts (unilaterally or due to deadlock)
  - Avg. 3% of transactions abort abnormally
- 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 such that the stored data is lost
  - Head crash/controller failure (?)
- Communication failures
  - Lost/undeliverable messages
  - Network partitioning

Local Recovery Management – Architecture

- Volatile storage
  - Consists of the main memory of the computer system (RAM).
- Stable storage
  - Resilient to failures and loses its contents only in the presence of media failures (e.g., head crashes on disks).
  - Implemented via a combination of hardware (non-volatile storage) and software (stable-write, stable-read, clean-up) components.



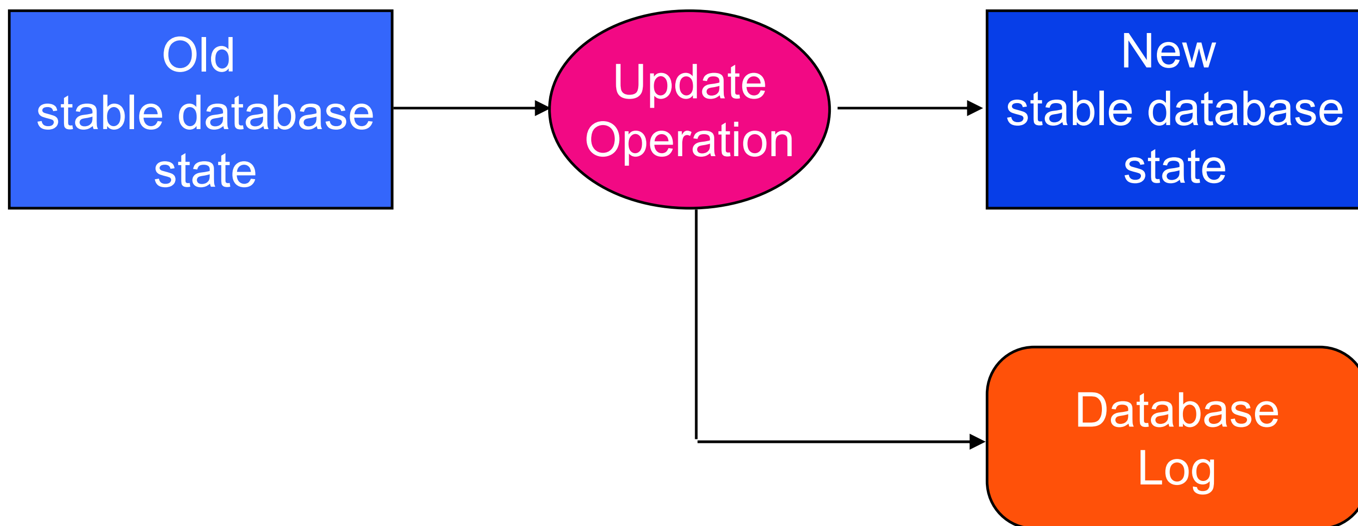
Update Strategies

- In-place update
  - Each update causes a change in one or more data values on pages in the database buffers
- Out-of-place update
  - Each update causes the new value(s) of data item(s) to be stored separate from the old value(s)

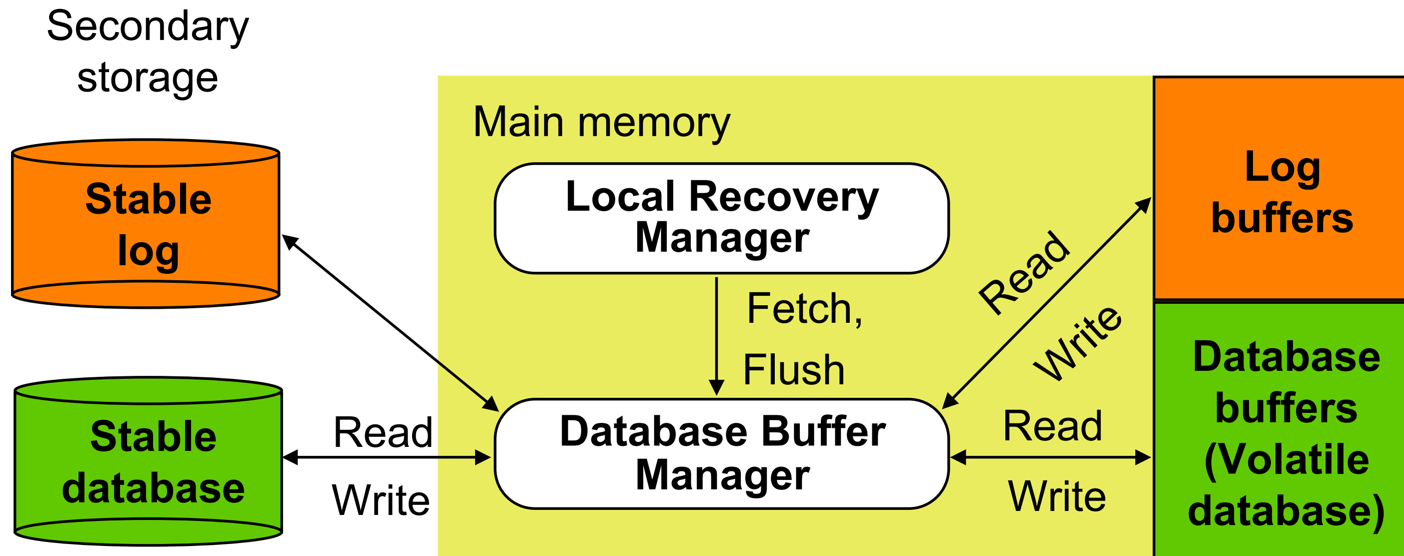
In-Place Update Recovery Information

Database Log

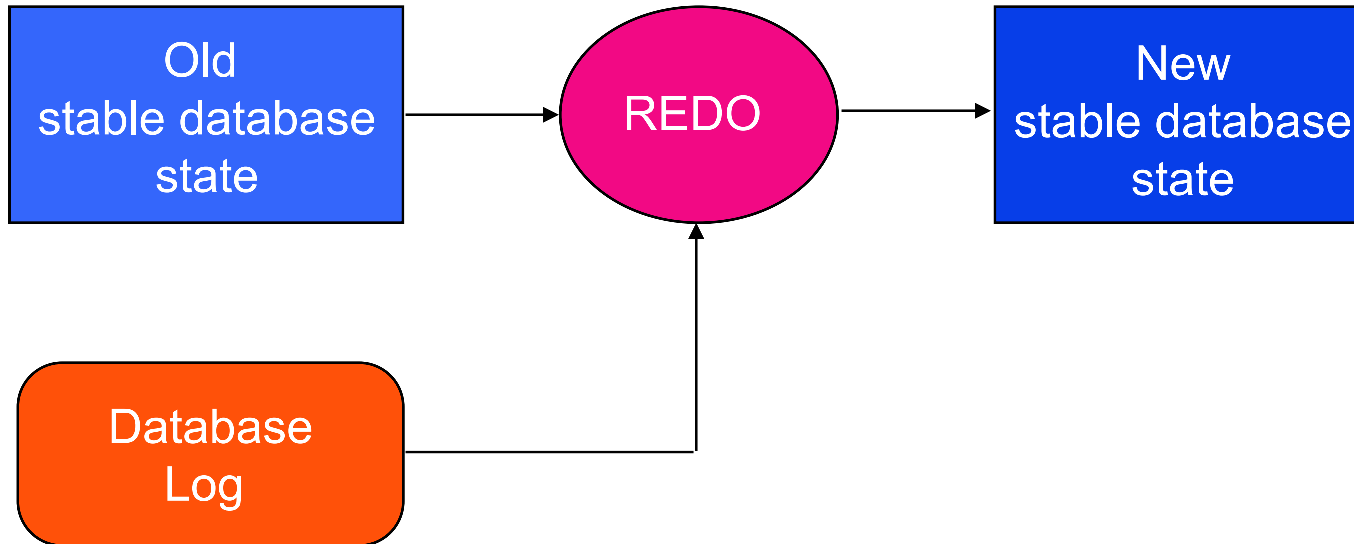
Every action of a transaction must not only perform the action, but must also write a *log* record to an append-only file.



Logging Interface

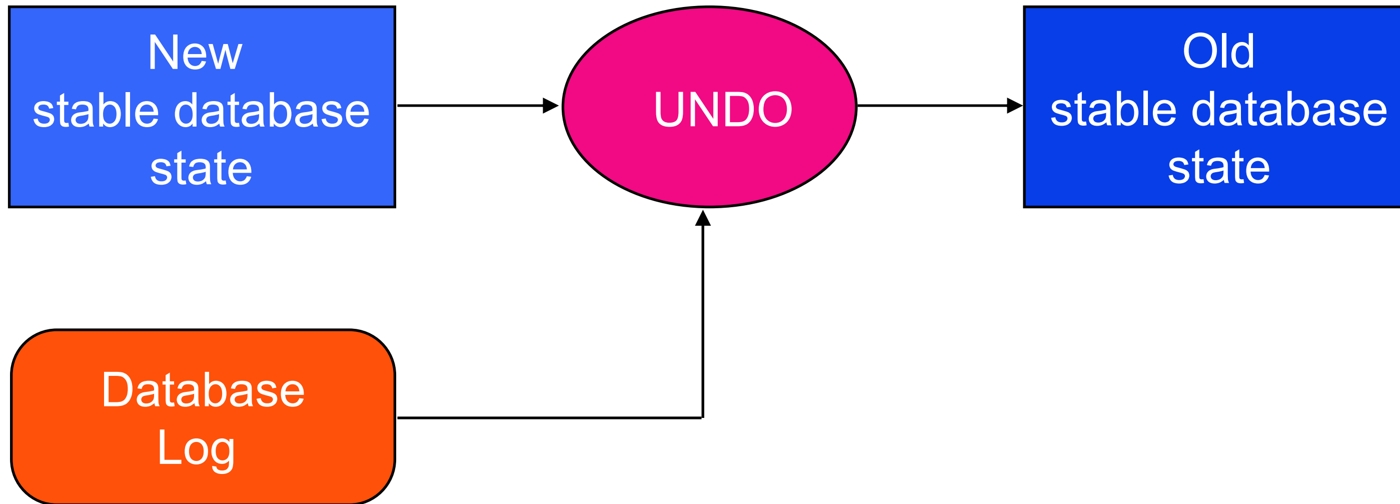


REDO Protocol



- REDO'ing an action means performing it again.
- The REDO operation uses the log information and performs the action that might have been done before, or not done due to failures.
- The REDO operation generates the new image.

UNDO Protocol



- UNDO'ing an action means to restore the object to its before image.
- The UNDO operation uses the log information and restores the old value of the object.

Write–Ahead Log (WAL) Protocol

- Notice:
  - If a system crashes before a transaction is committed, then all the operations must be undone. Only need the before images (undo portion of the log).
  - Once a transaction is committed, some of its actions might have to be redone. Need the after images (redo portion of the log).
- WAL protocol :
  1. Before a stable database is updated, the undo portion of the log should be written to the stable log
  2. When a transaction commits, the redo portion of the log must be written to stable log prior to the updating of the stable database.

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

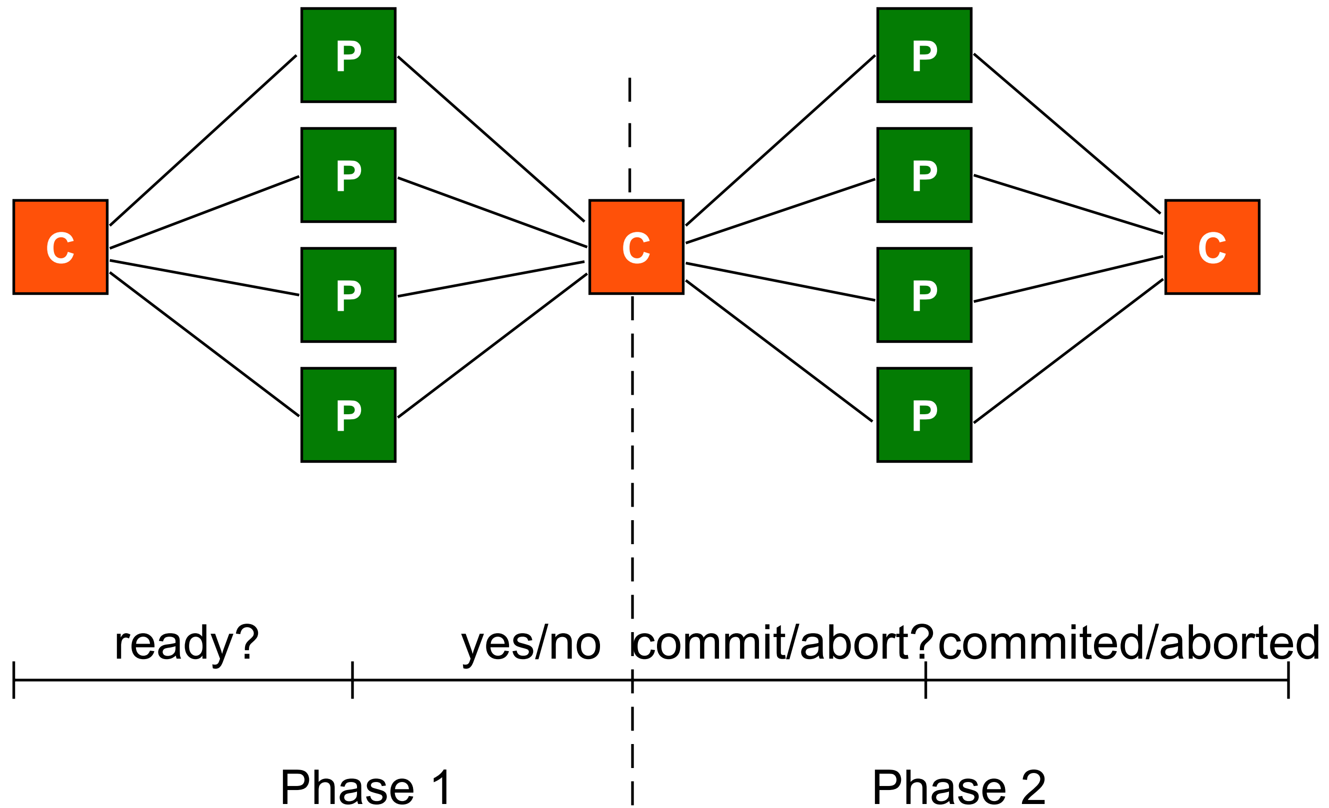
Phase 2 : Everybody writes the results into the database

- **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

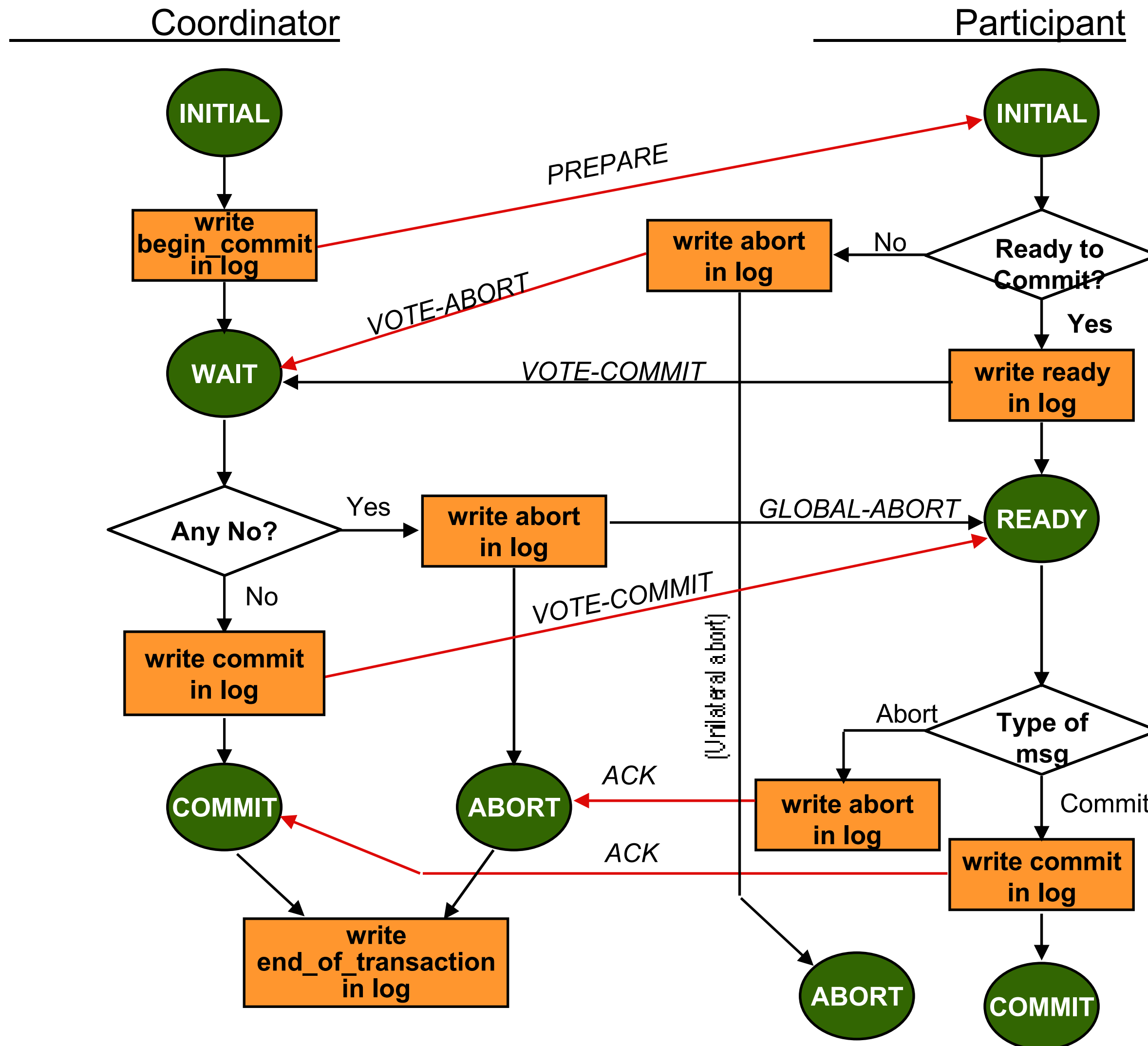
Global Commit Rule:

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.

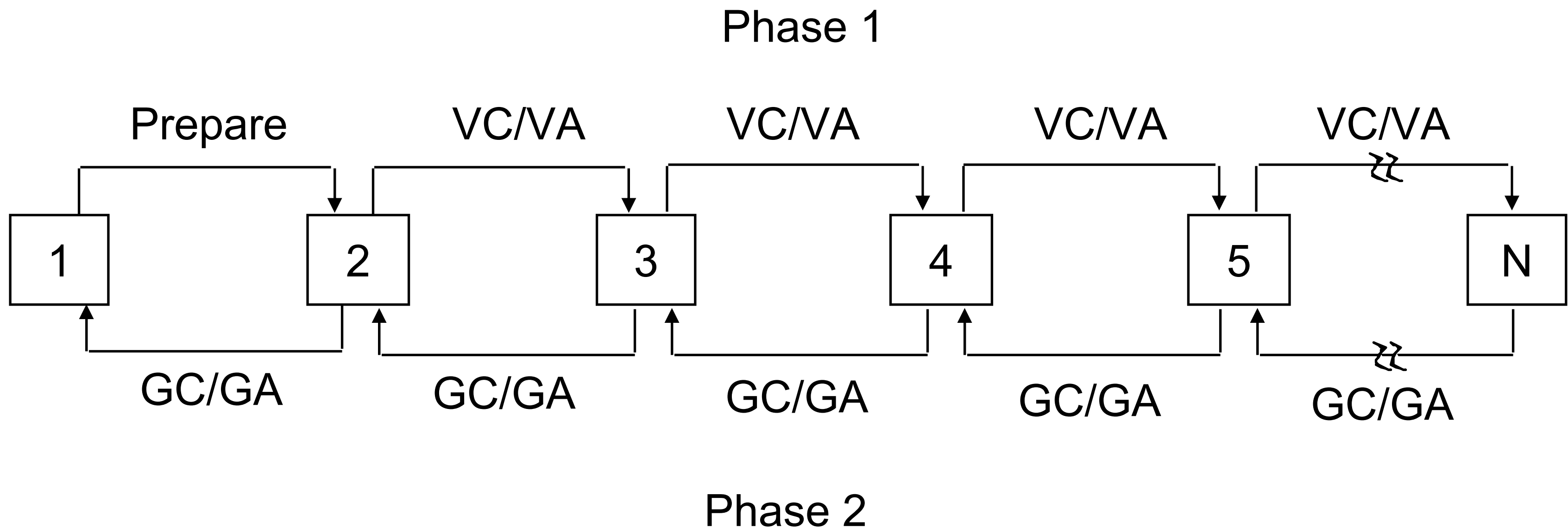
Centralized 2PC



2PC Protocol Actions

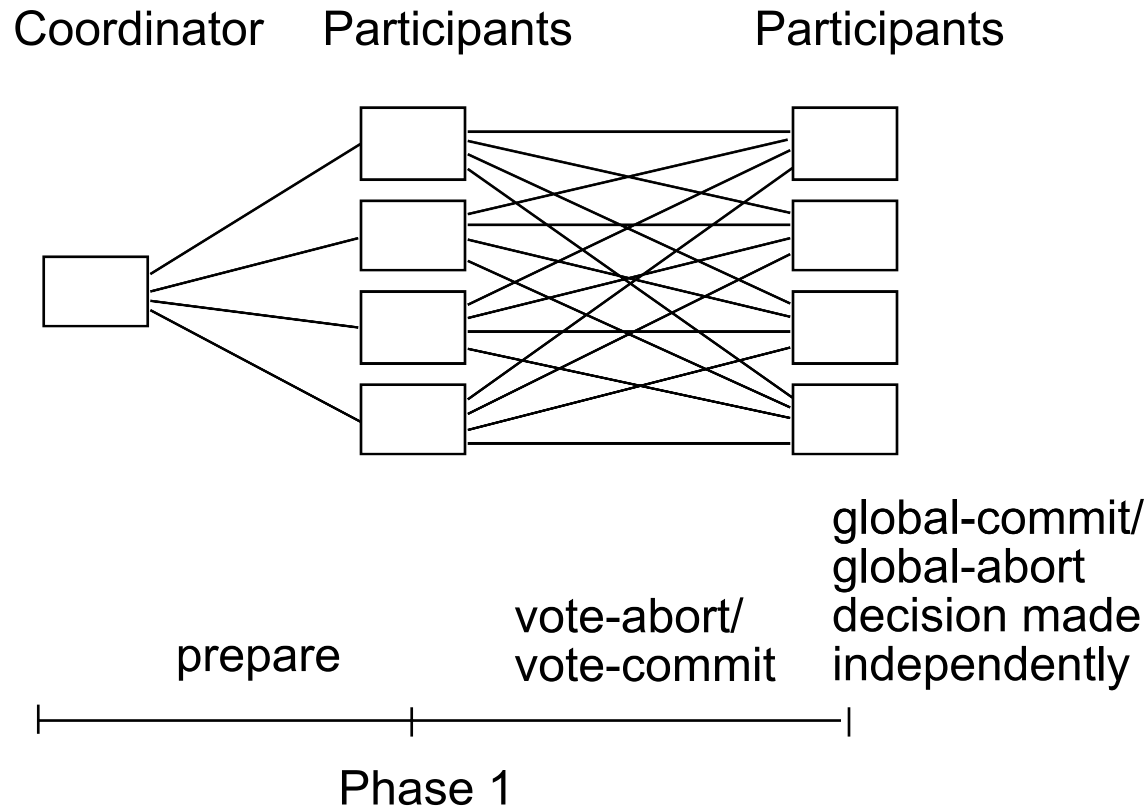


Linear 2PC

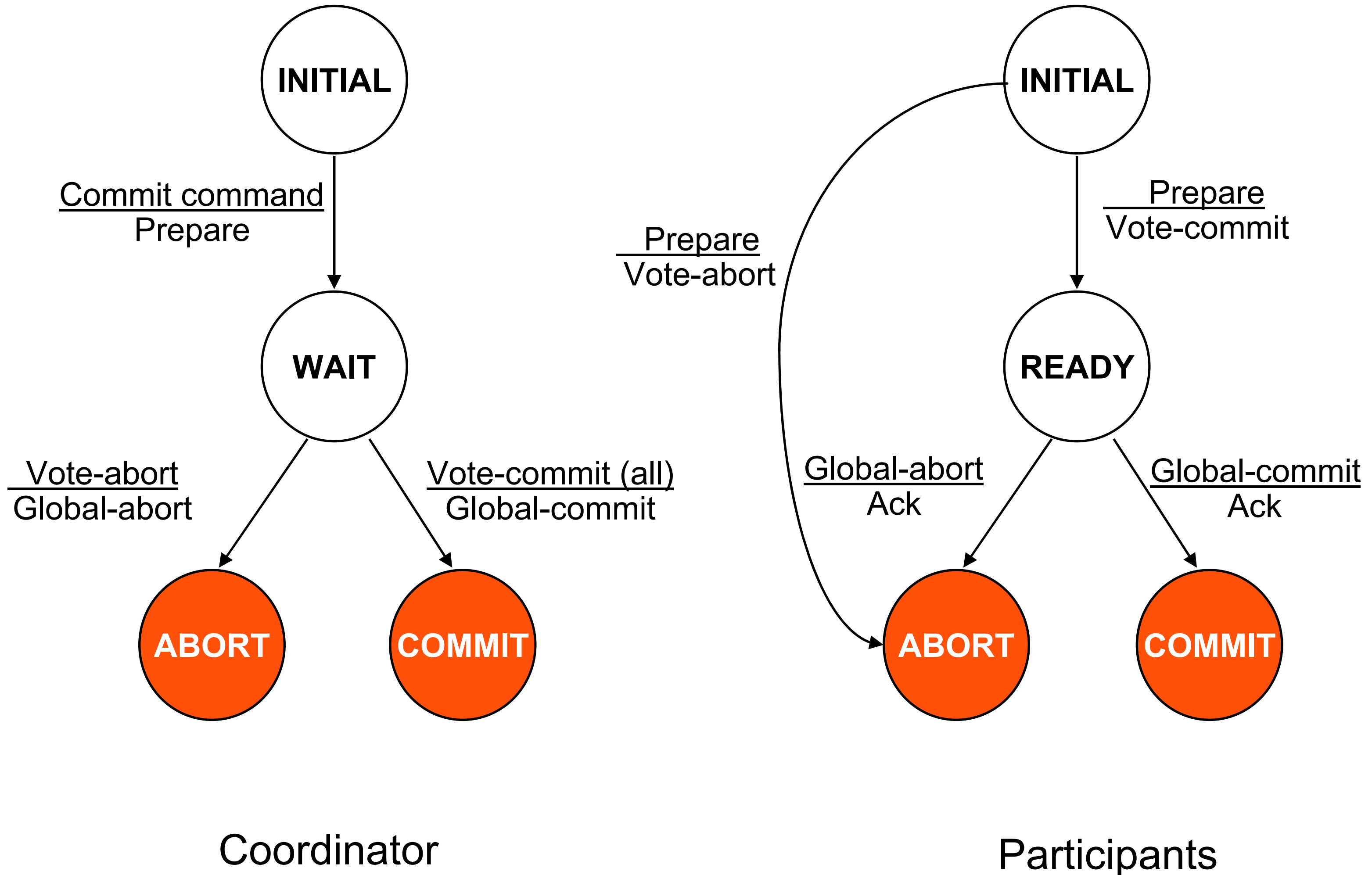


VC: Vote-Commit, VA: Vote-Abort, GC: Global-commit, GA: Global-abort

Distributed 2PC

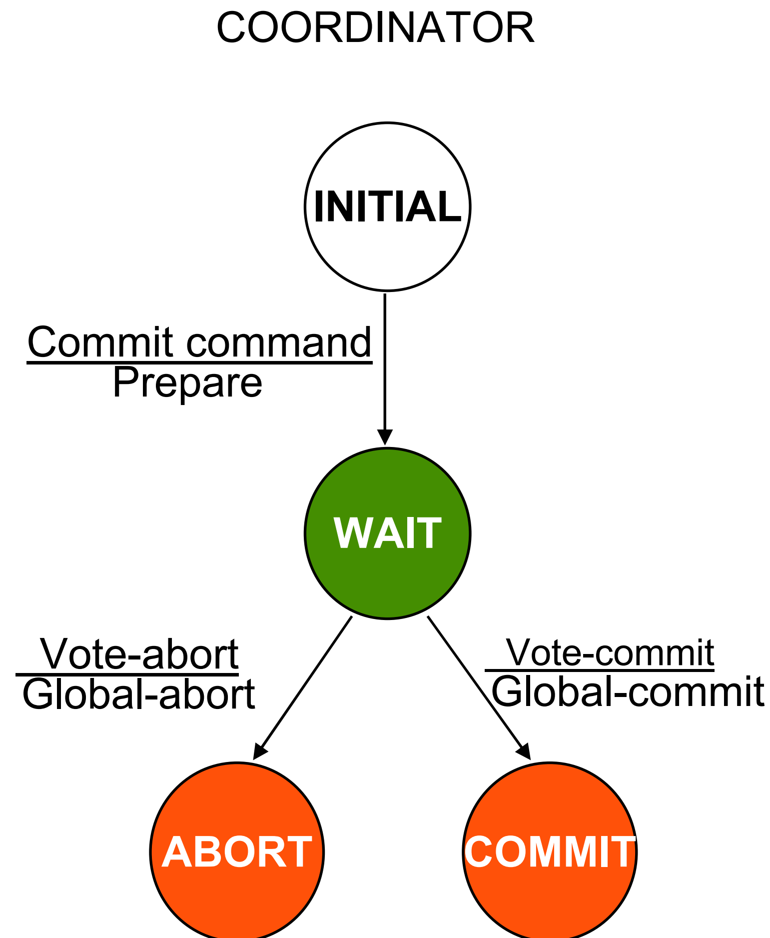


State Transitions in 2PC



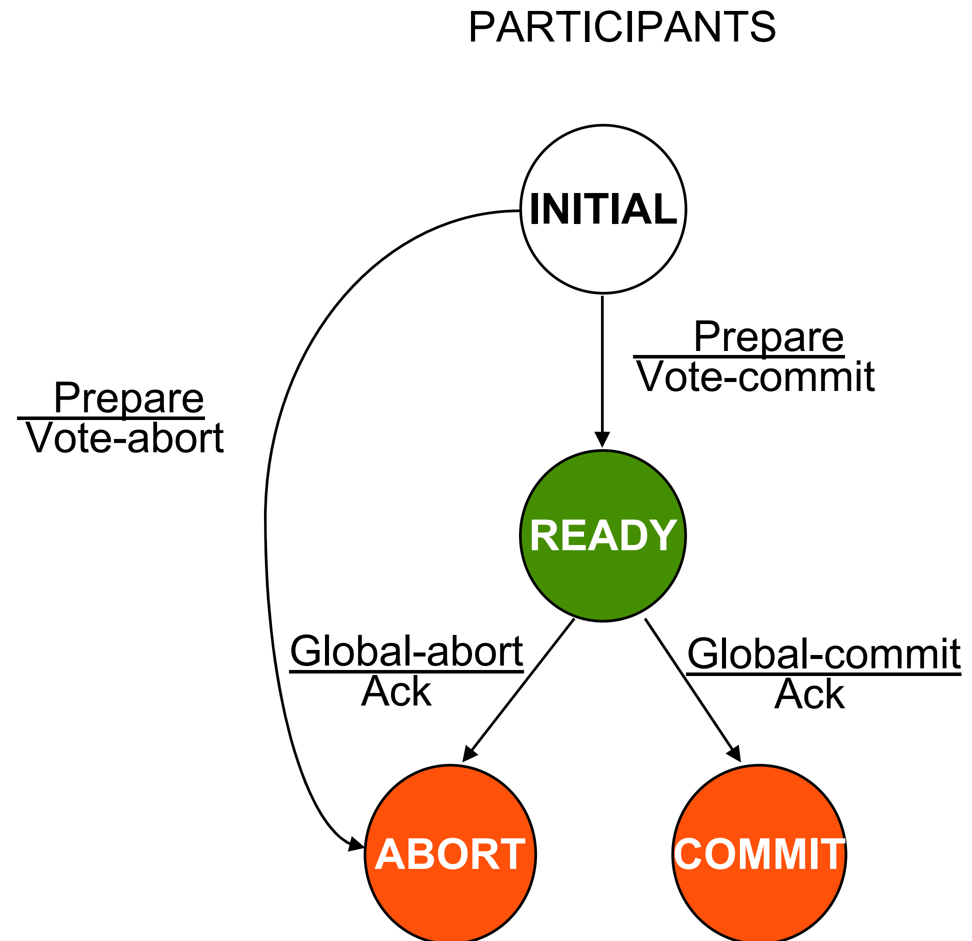
Site Failures - 2PC Termination

- 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



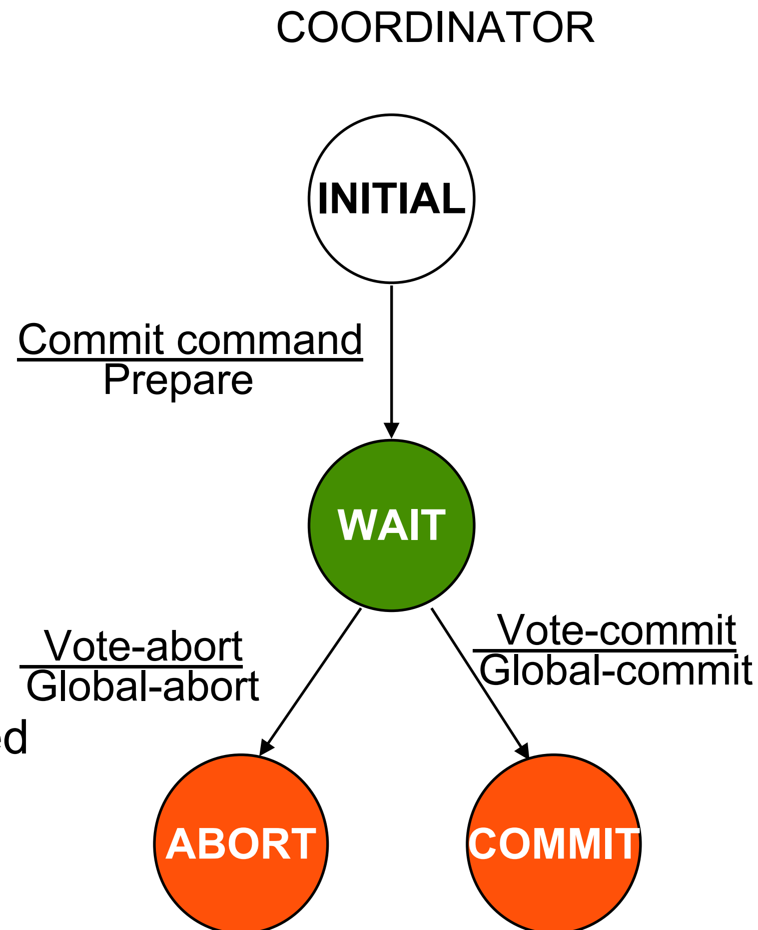
Site Failures - 2PC Termination

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



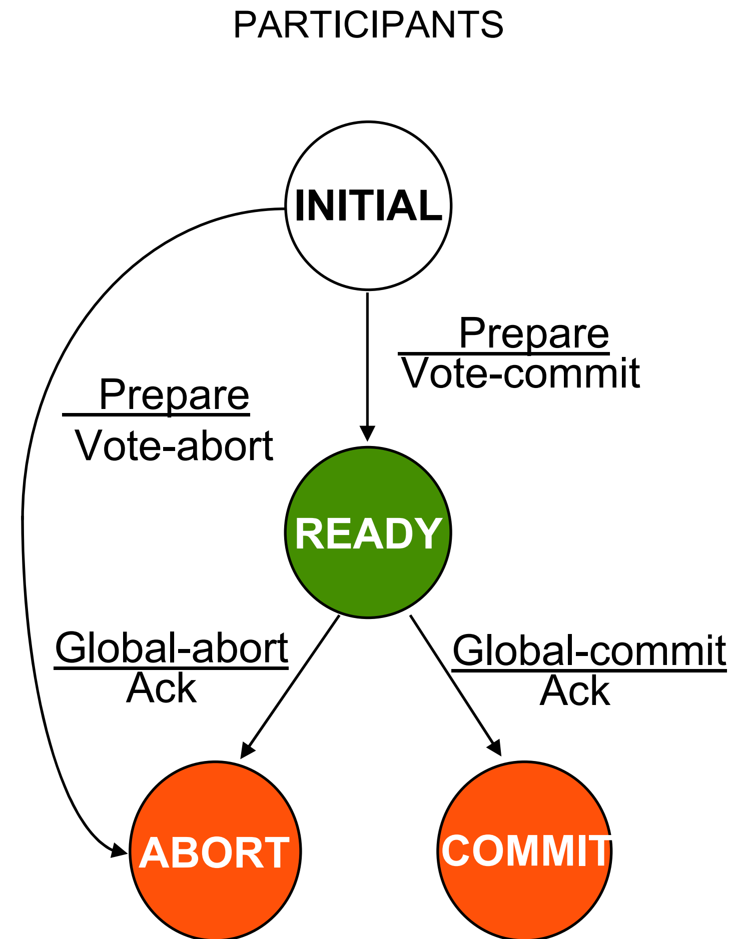
Site Failures - 2PC Recovery

- 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



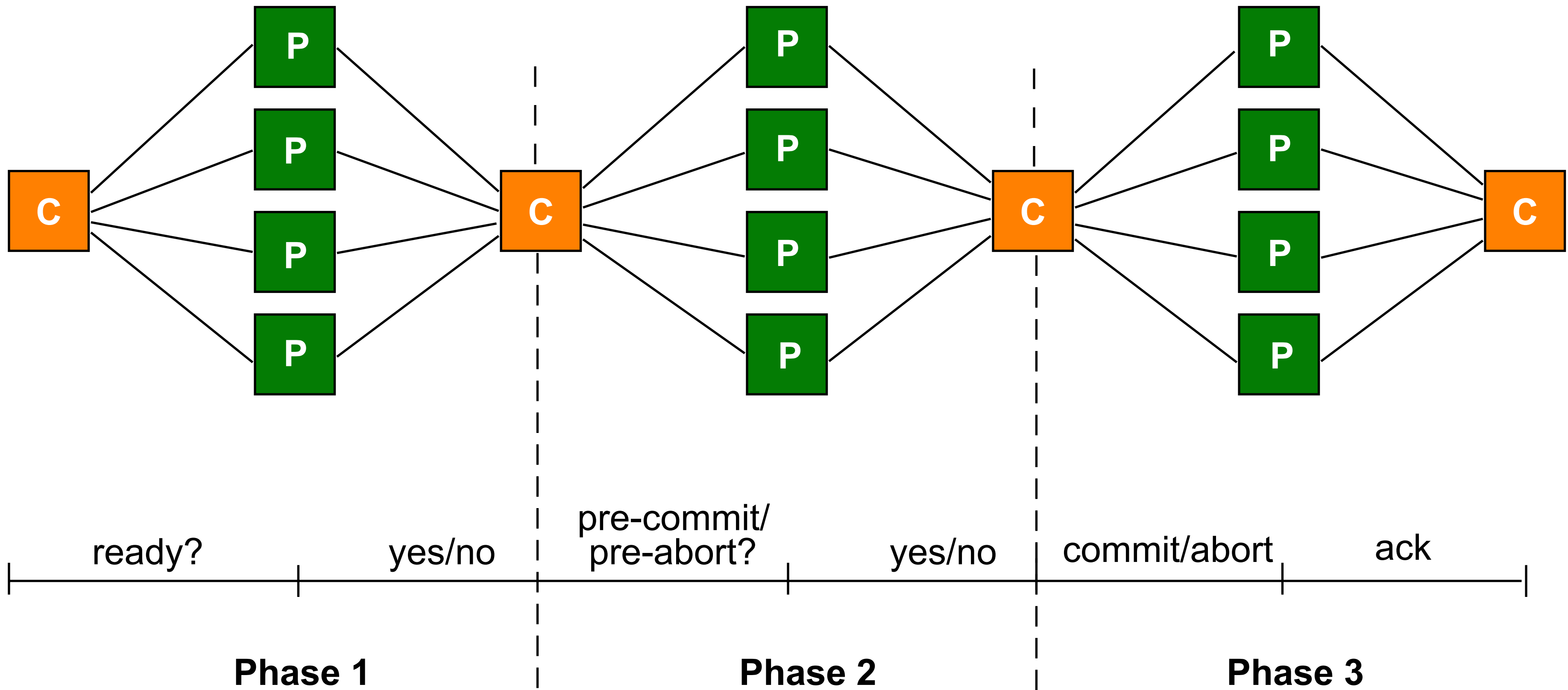
Problem With 2PC

- 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

Communication Structure



Network Partitioning

- Simple partitioning
  - Only two partitions
- Multiple partitioning
  - More than two partitions
- Formal bounds (due to Skeen):
  - 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)
    - different for replicated vs non-replicated databases

Quorum Protocols for Non-Replicated Databases

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

Quorum Protocols for Replicated Databases

- Network partitioning is handled by the replica control protocol.
- One implementation:
  - Assign a vote to each *copy* of a replicated data item (say V_i) such that $\Sigma_i V_i = V$
  - Each operation has to obtain a *read quorum* (V_r) to read and a *write quorum* (V_w) to write a data item
  - Then the following rules have to be obeyed in determining the quorums:
    - $V_r + V_w > V$ — a data item is not read and written by two transactions concurrently
    - $V_w > V/2$ — two write operations from two transactions cannot occur concurrently on the same data item

Use for Network Partitioning

- Simple modification of the ROWA rule:
  - When the replica control protocol attempts to read or write a data item, it first checks if a majority of the sites are in the same partition as the site that the protocol is running on (by checking its votes). If so, execute the ROWA rule within that partition.
- Assumes that failures are “clean” which means:
  - failures that change the network's topology are detected by all sites instantaneously
  - each site has a view of the network consisting of all the sites it can communicate with