

# Outline

---

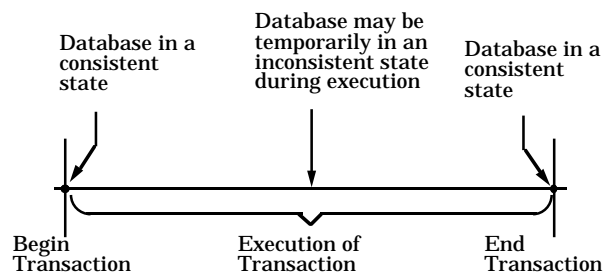
- Introduction
- Background
- Distributed DBMS Architecture
- Distributed Database Design
- Semantic Data Control
- Distributed Query Processing
- Distributed Transaction Management
  - Transaction Concepts and Models
  - Distributed Concurrency Control
  - Distributed Reliability
- Distributed Database Operating Systems
- Open Systems and Interoperability
- Parallel Database Systems
- Distributed Object Management
- Concluding Remarks

## Transaction

---

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

- concurrency transparency
- failure transparency



## Transaction Example – A Simple SQL Query

---

```
Transaction BUDGET_UPDATE
begin
    EXEC SQL UPDATE J
              SET      BUDGET = BUDGET*1.1
              WHERE   JNAME = "CAD/CAM"
end.
```

## 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 – SQL Version

---

```
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")
endif
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}
```

## Example Transaction – Reads & Writes

---

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);
    temp ← Read(flight_no(date).stsold);
    if temp = flight(date).cap then
        begin
            output("no free seats");
            Abort
        end
    else begin
        Write(flight(date).stsold, temp + 1);
        Write(flight(date).cname, customer_name);
        Write(flight(date).special, null);
        Commit;
        output("reservation completed")
    end
end . {Reservation}
```

## Characterization

---

- 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)
  - ➡  $RS \cup WS$

## Formalization

---

Let

- ➡  $O_{ij}(x)$  be some operation  $O_j$  of transaction  $T_i$  operating on entity  $x$ , where  $O_j \in \{\text{read}, \text{write}\}$  and  $O_j$  is atomi.c
- ➡  $OS_i = \cup_j O_{ij}$
- ➡  $N_i \in \{\text{abort}, \text{commit}\}$

Transaction  $T_i$  is a partial order  $T_i = \{\Sigma_i, <_i\}$  where

- ❶  $\Sigma_i = OS_i \cup \{N_i\}$
- ❷ For any two operations  $O_{ij}, O_{ik} \in OS_i$ , if  $O_{ij} = R(x)$  and  $O_{ik} = W(x)$  for any data item  $x$ , then either  $O_{ij} <_i O_{ik}$  or  $O_{ik} <_i O_{ij}$
- ❸  $\forall O_{ij} \in OS_i, O_{ij} <_i N_i$

## Example

---

Consider a transaction  $T$ :

Read( $x$ )  
Read( $y$ )  
 $x \leftarrow x + y$   
Write( $x$ )  
Commit

Then

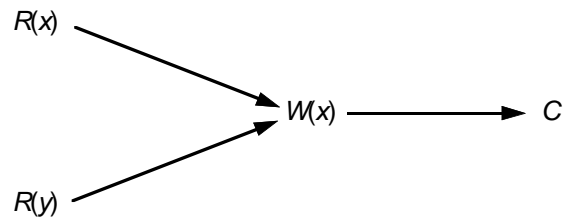
$\Sigma = \{R(x), R(y), W(x), C\}$   
 $< = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$

## DAG Representation

---

Assume

$\leq = \{(R(x), W(x)), (R(y), W(x)), (R(x), C), (R(y), C), (W(x), C)\}$



## Properties of Transactions

---

### **A** TOMICITY

⇒ all or nothing

### **C**ONSISTENCY

⇒ no violation of integrity constraints

### **I**SOLATION

⇒ concurrent changes invisible ÷ serializable

### **D**URABILITY

⇒ committed updates persist

## Atomicity

---

- Either **all or none** of the transaction's operations are performed.
- Atomicity requires that if a transaction is interrupted by a failure, its partial results must be **undone**.
- The activity of preserving the transaction's atomicity in presence of transaction aborts due to input errors, system overloads, or deadlocks is called **transaction recovery**.
- The activity of ensuring atomicity in the presence of system crashes is called **crash recovery**.

## Consistency

---

- Internal consistency
  - ⇒ A transaction which executes *alone* against a *consistent* database leaves it in a consistent state.
  - ⇒ Transactions do not violate database integrity constraints.
- Transactions are **correct** programs

## Isolation

---

- **Serializability**

- If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order.

- **Incomplete results**

- An incomplete transaction cannot reveal its results to other transactions before its commitment.
- Necessary to avoid **cascading aborts**.

## Durability

---

- Once a transaction commits, the system must guarantee that the results of its operations will never be lost, in spite of subsequent failures.
- Database recovery



# Characterization of Transactions

---

## Based on

- ➡ Application areas
  - ◆ non-distributed vs. distributed
  - ◆ compensating transactions
  - ◆ heterogeneous transactions
- ➡ Timing
  - ◆ on-line (short-life) vs batch (long-life)
- ➡ Structure
  - ◆ flat (or simple) transactions
  - ◆ nested transactions
- ➡ Organization of read and write actions
  - ◆ two-step
  - ◆ restricted
  - ◆ action model

# Transaction Structure

---

## ■ Flat transaction

- ➡ Consists of a sequence of **primitive** operations embraced between a **begin** and **end** markers.

```
Begin_transaction Reservation
...
end.
```

## ■ Nested transaction

- ➡ The operations of a transaction may themselves be transactions.

```
Begin_transaction Reservation
...
  Begin_transaction Airline
    - ...
  end. {Airline}
  Begin_transaction Hotel
    ...
  end. {Hotel}
end. {Reservation}
```

## Nested Transactions

---

- Have the same properties as their parents  $\Rightarrow$  may themselves have other nested transactions.
- Introduces concurrency control and recovery concepts to within the transaction.
- Types
  - ➡ Closed nesting
    - ◆ Subtransactions begin *after* their parents and finish *before* them.
    - ◆ Commitment of a subtransaction is conditional upon the commitment of the parent (commitment through the root).
  - ➡ Open nesting
    - ◆ Subtransactions can execute and commit independently.
    - ◆ Compensation may be necessary.

## 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

---

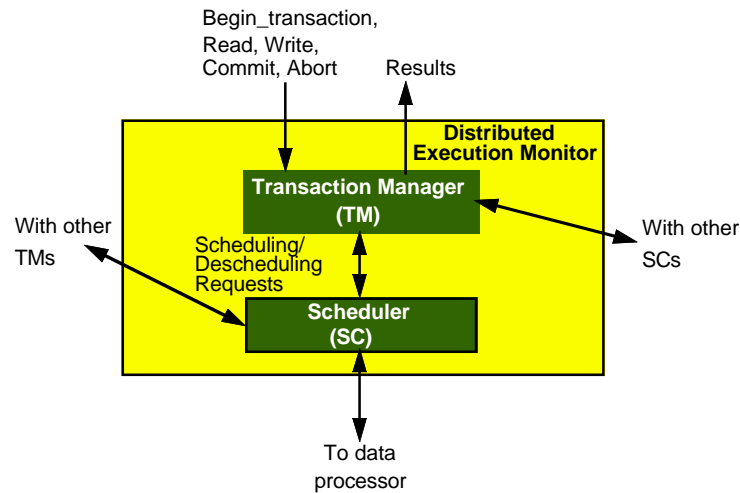
- Transaction structure (usually called transaction model)
  - ➡ Flat (simple), nested
- Internal database consistency
  - ➡ Semantic data control (integrity enforcement) algorithms
- Reliability protocols
  - ➡ Atomicity & Durability
  - ➡ Local recovery protocols
  - ➡ Global commit protocols

## Transaction Processing Issues

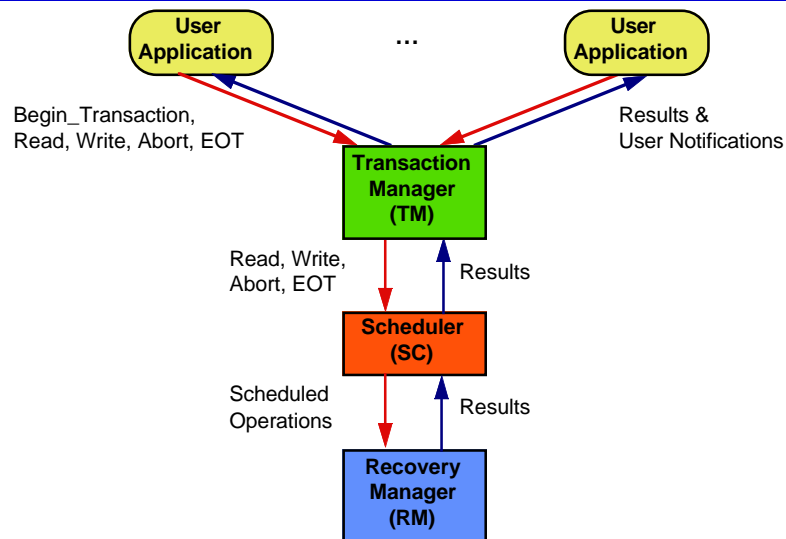
---

- 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

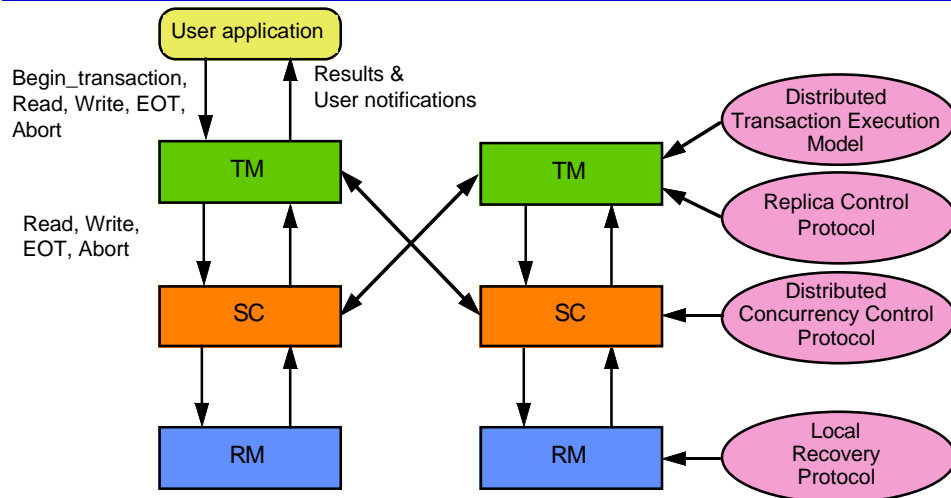
## Architecture Revisited



## 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.

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

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

## Formalization of Schedule

---

A **complete schedule**  $SC(T)$  over a set of transactions  $T = \{T_1, \dots, T_n\}$  is a partial order  $SC(T) = \{\Sigma_T, <_T\}$  where

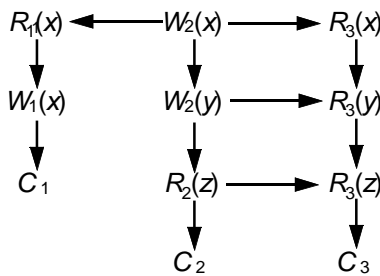
- 1  $\Sigma_T = \cup_i \Sigma_i$  , for  $i = 1, 2, \dots, n$
- 2  $<_T \supseteq \cup_i <_i$ , for  $i = 1, 2, \dots, n$
- 3 For any two conflicting operations  $O_{ij}, O_{kl} \in \Sigma_T$ , either  $O_{ij} <_T O_{kl}$  or  $O_{kl} <_T O_{ij}$

## Complete Schedule – Example

Given three transactions

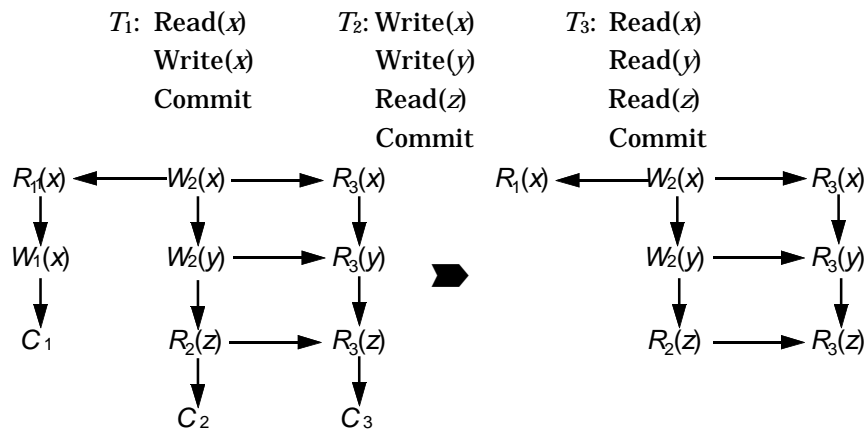
$T_1$ : Read( $x$ )	$T_2$ : Write( $x$ )	$T_3$ : Read( $x$ )
Write( $x$ )	Write( $y$ )	Read( $y$ )
Commit	Read( $z$ )	Read( $z$ )
	Commit	Commit

A possible complete schedule is given as the DAG



## Schedule Definition

A **schedule** is a prefix of a complete schedule such that only some of the operations and only some of the ordering relationships are included.



## 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 unaborting 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.



## Serializable History

---

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

The following are not conflict equivalent

$$H_3 = \{W_2(x), W_2(y), R_2(z), C_2, R_1(x), W_1(x), C_1, R_3(x), R_3(y), R_3(z), C_3\}$$

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

The following are conflict equivalent; therefore

$H_2$  is *serializable*.

$$H_3 = \{W_2(x), W_2(y), R_2(z), C_2, R_1(x), W_1(x), C_1, R_3(x), R_3(y), R_3(z), C_3\}$$

$$H_2 = \{W_2(x), R_1(x), W_1(x), C_1, R_3(x), W_2(y), R_3(y), R_2(z), C_2, R_3(z), C_3\}$$

## 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

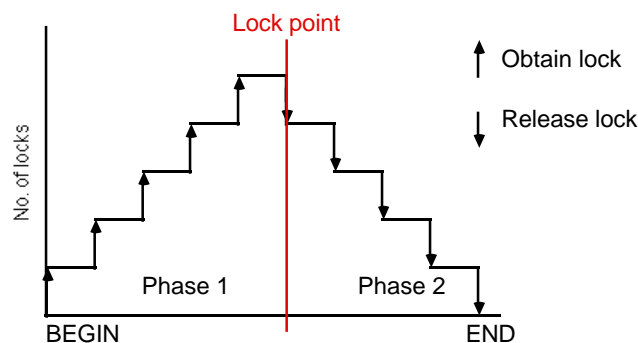
- Transactions indicate their intentions by requesting locks from the scheduler (called **lock manager**).
- Locks are either **read lock** (*rl*) [also called **shared lock**] or **write lock** (*wl*) [also called **exclusive lock**]
- Read locks and write locks conflict (because Read and Write operations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

	<i>rl</i>	<i>wl</i>
<i>rl</i>	yes	no
<i>wl</i>	no	no

- Locking works nicely to allow concurrent processing of transactions.

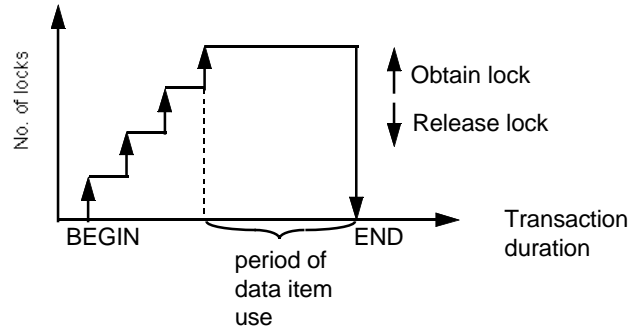
## Two-Phase Locking (2PL)

- 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.



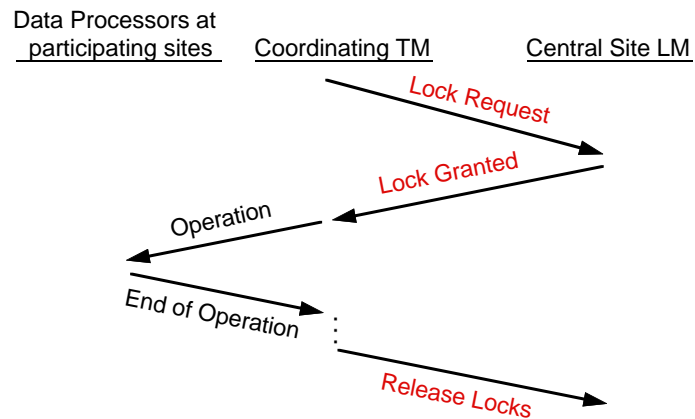
## Strict 2PL

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.



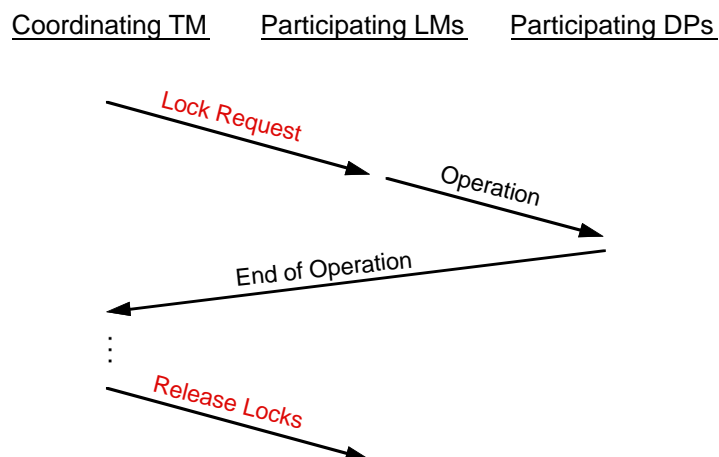
## Distributed 2PL

---

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

## 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 Timestamp Ordering

---

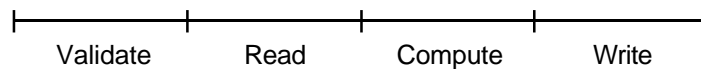
- Do not modify the values in the database, create new values.
- A  $R_i(x)$  is translated into a read on one version of  $x$ .
  - ➡ Find a version of  $x$  (say  $x_v$ ) such that  $ts(x_v)$  is the largest timestamp less than  $ts(T_i)$ .
- A  $W_i(x)$  is translated into  $W_i(x_w)$  and accepted if the scheduler has not yet processed any  $R_j(x_r)$  such that

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

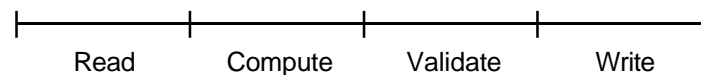
## Optimistic Concurrency Control Algorithms

---

Pessimistic execution



Optimistic execution



# Optimistic Concurrency Control Algorithms

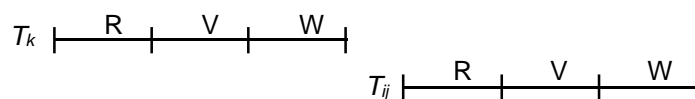
---

- 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

---

- 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

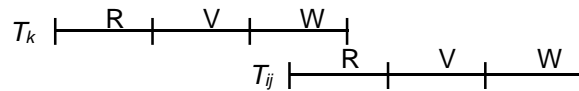




## Optimistic CC Validation Test

- ② 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  $WS(T_k) \cap RS(T_{ij}) = \emptyset$

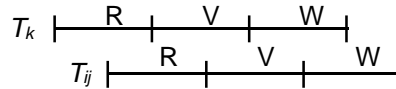
➡ Read and write phases overlap, but  $T_{ij}$  does not read data items written by  $T_k$



## Optimistic CC Validation Test

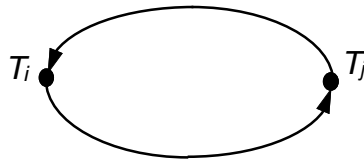
- ③ If there is any transaction  $T_k$  such that  $ts(T_k) < ts(T_{ij})$  and which completes its read phase before  $T_{ij}$  completes its read phase, then validation succeeds if  $WS(T_k) \cap RS(T_{ij}) = \emptyset$  and  $WS(T_k) \cap WS(T_{ij}) = \emptyset$

➡ 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

---

## ■ Ignore

- ➡ Let the application programmer deal with it, or restart the system

## ■ 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 Avoidance – Wait-Die & Wound-Wait Algorithms

---

**WAIT-DIE Rule:** If  $T_i$  requests a lock on a data item which is already locked by  $T_j$ , then  $T_i$  is permitted to wait iff  $ts(T_i) < ts(T_j)$ . If  $ts(T_i) > ts(T_j)$ , then  $T_i$  is aborted and restarted with the same timestamp.

- ➡ if  $ts(T_i) < ts(T_j)$  then  $T_i$  waits else  $T_i$  dies
- ➡ non-preemptive:  $T_i$  never preempts  $T_j$
- ➡ prefers younger transactions

**WOUND-WAIT Rule:** If  $T_i$  requests a lock on a data item which is already locked by  $T_j$ , then  $T_i$  is permitted to wait iff  $ts(T_i) > ts(T_j)$ . If  $ts(T_i) < ts(T_j)$ , then  $T_j$  is aborted and the lock is granted to  $T_i$ .

- ➡ if  $ts(T_i) < ts(T_j)$  then  $T_j$  is wounded else  $T_i$  waits
- ➡ preemptive:  $T_i$  preempts  $T_j$  if it is younger
- ➡ prefers older transactions

## Deadlock Detection

---

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

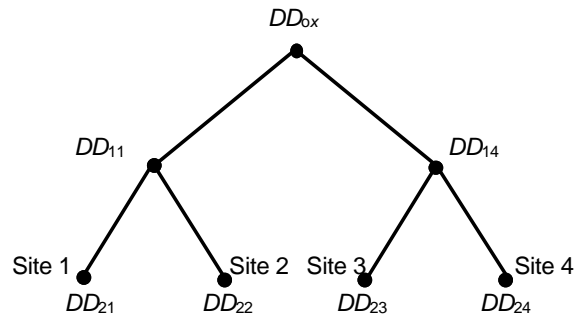
## Centralized Deadlock Detection

---

- 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:
  - ➡ The local WFGs are formed at each site and passed on to other sites. Each local WFG is modified as follows:
    - ❶ Since each site receives the potential deadlock cycles from other sites, these edges are added to the local WFGs
    - ❷ 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.

# Reliability

---

Problem:

How to maintain

atomicity

durability

properties of transactions

## Fundamental Definitions

---

### ■ Reliability

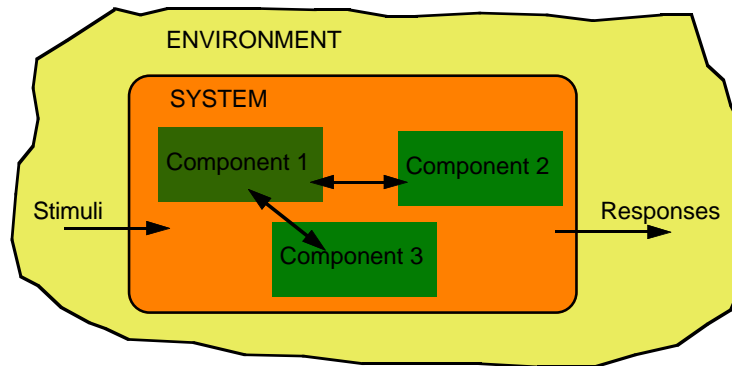
- ➡ A measure of success with which a system conforms to some authoritative specification of its behavior.
- ➡ Probability that the system has not experienced any failures within a given time period.
- ➡ Typically used to describe systems that cannot be repaired or where the continuous operation of the system is critical.

### ■ Availability

- ➡ The fraction of the time that a system meets its specification.
- ➡ The probability that the system is operational at a given time  $t$ .

## Basic System Concepts

---



**External state**

**Internal state**

## Fundamental Definitions

---

### ■ Failure

- ➡ The deviation of a system from the behavior that is described in its specification.

### ■ Erroneous state

- ➡ The internal state of a system such that there exist circumstances in which further processing, by the normal algorithms of the system, will lead to a failure which is not attributed to a subsequent fault.

### ■ Error

- ➡ The part of the state which is incorrect.

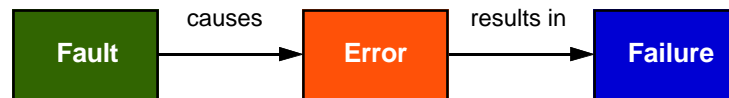
### ■ Fault

- ➡ An error in the internal states of the components of a system or in the design of a system.



## Faults to Failures

---

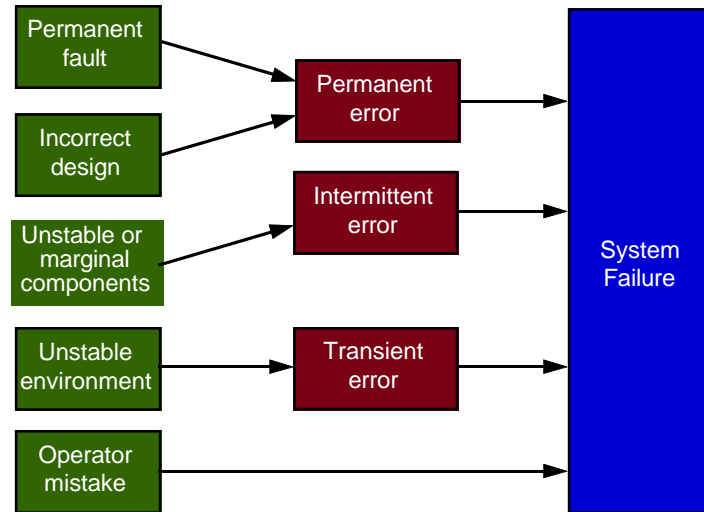


## Types of Faults

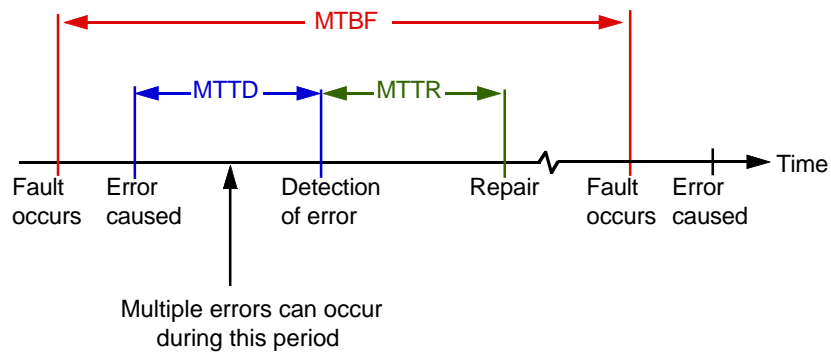
---

- **Hard faults**
  - ➡ Permanent
  - ➡ Resulting failures are called hard failures
- **Soft faults**
  - ➡ Transient or intermittent
  - ➡ Account for more than 90% of all failures
  - ➡ Resulting failures are called soft failures

## Fault Classification



## Failures



## Fault Tolerance Measures

---

### Reliability

$$R(t) = \Pr\{0 \text{ failures in time } [0, t] \mid \text{no failures at } t=0\}$$

If occurrence of failures is Poisson

$$R(t) = \Pr\{0 \text{ failures in time } [0, t]\}$$

Then

$$\Pr(k \text{ failures in time } [0, t]) = \frac{e^{-m(t)} [m(t)]^k}{k!}$$

where  $m(t)$  is known as the *hazard function* which gives the time-dependent failure rate of the component and is defined as

$$m(t) = \int_0^t z(x) dx$$

## Fault-Tolerance Measures

---

### Reliability

The mean number of failures in time  $[0, t]$  can be computed as

$$E[k] = \sum_{k=0}^{\infty} k \frac{e^{-m(t)} [m(t)]^k}{k!} = m(t)$$

and the variance can be computed as

$$\text{Var}[k] = E[k^2] - (E[k])^2 = m(t)$$

Thus, reliability of a single component is

$$R(t) = e^{-m(t)}$$

and of a system consisting of  $n$  non-redundant components as

$$R_{\text{sys}}(t) = \prod_{i=1}^n R_i(t)$$

## Fault-Tolerance Measures

---

### Availability

$$A(t) = \Pr\{\text{system is operational at time } t\}$$

Assume

- ♦ Poisson failures with rate  $\lambda$
- ♦ Repair time is exponentially distributed with mean  $1/\mu$

Then, steady-state availability

$$A = \lim_{t \rightarrow \infty} A(t) = \frac{\mu}{\lambda + \mu}$$

## Fault-Tolerance Measures

---

### MTBF

Mean time between failures

$$\text{MTBF} = \int_0^{\infty} R(t) dt$$

### MTTR

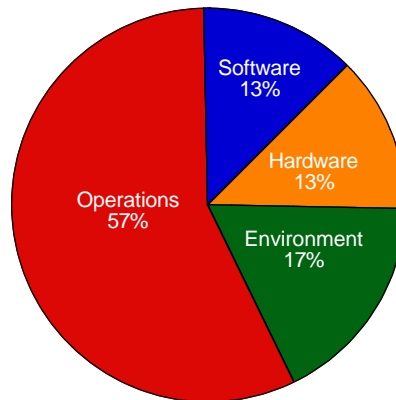
Mean time to repair

### Availability

$$\frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

## Sources of Failure – SLAC Data (1985)

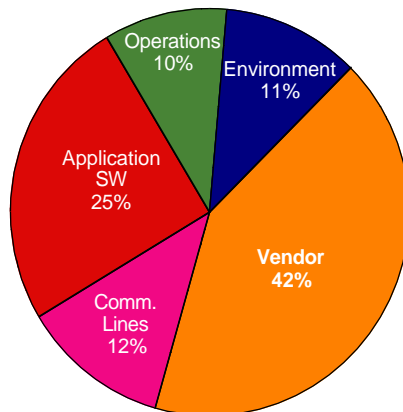
---



S. Mourad and D. Andrews, "The Reliability of the IBM/XA Operating System", *Proc. 15th Annual Int. Symp. on FTCS*, 1985.

## Sources of Failure – Japanese Data (1986)

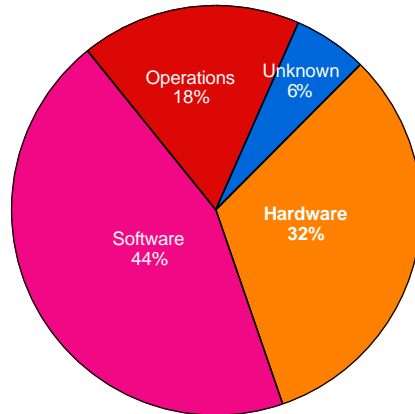
---



"Survey on Computer Security", Japan Info. Dev. Corp., 1986.

## Sources of Failure – 5ESS Switch (1987)

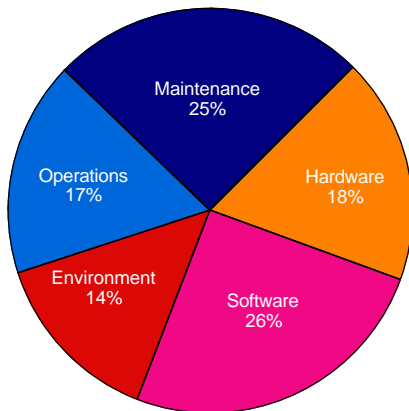
---



D.A. Yaeger. 5ESS Switch Performance Metrics. *Proc. Int. Conf. on Communications, Volume 1*, pp. 46-52, June 1987.

## Sources of Failures – Tandem Data (1985)

---



Jim Gray, *Why Do Computers Stop and What can be Done About It?*, Tandem Technical Report 85.7, 1985.

## Types of Failures in D-DBMS

### ■ 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

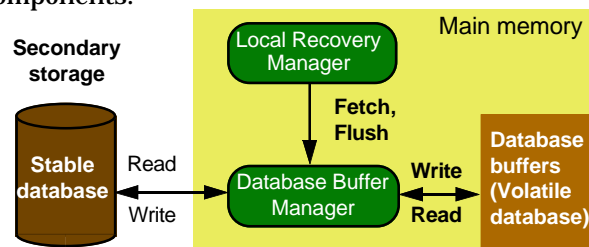
## Local Recovery Management – Architecture

### ■ 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

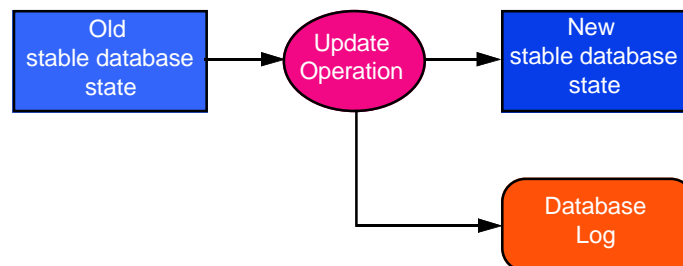
- ➡ Each update causes the new value(s) of data item(s) to be stored separate from the old value(s)

## 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

---

The log contains information used by the recovery process to restore the consistency of a system. This information may include

- transaction identifier
- type of operation (action)
- items accessed by the transaction to perform the action
- old value (state) of item (**before image**)
- new value (state) of item (**after image**)

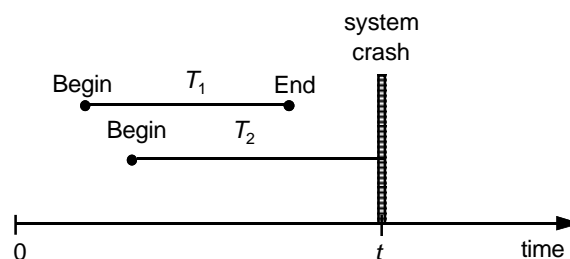
...

## Why Logging?

---

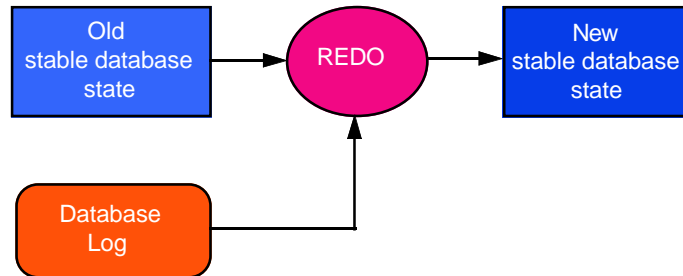
Upon recovery:

- all of  $T_1$ 's effects should be reflected in the database (REDO if necessary due to a failure)
- none of  $T_2$ 's effects should be reflected in the database (UNDO if necessary)



## REDO Protocol

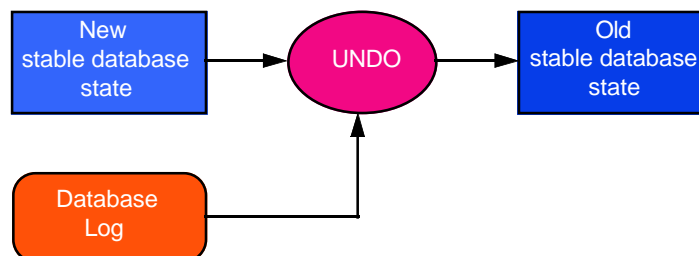
---



- 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.

## When to Write Log Records Into Stable Store

---

Assume a transaction  $T$  updates a page  $P$

■ **Fortunate case**

- ➡ System writes  $P$  in stable database
- ➡ System updates stable log for this update
- ➡ SYSTEM FAILURE OCCURS!... (before  $T$  commits)

We can recover (undo) by restoring  $P$  to its old state by using the log

■ **Unfortunate case**

- ➡ System writes  $P$  in stable database
- ➡ SYSTEM FAILURE OCCURS!... (before stable log is updated)

We cannot recover from this failure because there is no log record to restore the old value.

■ **Solution: Write-Ahead Log (WAL) protocol**

## Write-Ahead Log Protocol

---

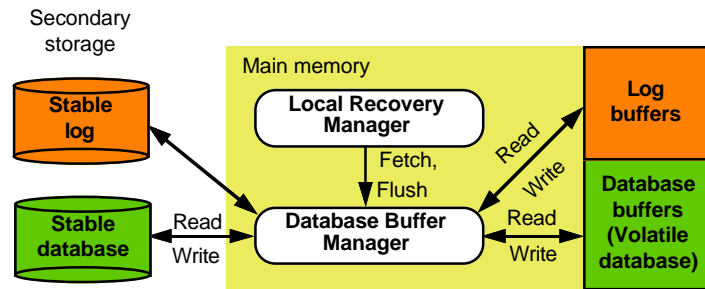
■ **Notice:**

- ➡ 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.

## Logging Interface



## Out-of-Place Update Recovery Information

### ■ Shadowing

- When an update occurs, don't change the old page, but create a shadow page with the new values and write it into the stable database.
- Update the access paths so that subsequent accesses are to the new shadow page.
- The old page retained for recovery.

### ■ Differential files

- For each file  $F$  maintain
  - ◆ a read only part  $FR$
  - ◆ a differential file consisting of insertions part  $DF+$  and deletions part  $DF-$
  - ◆ Thus,  $F = (FR \cup DF+) - DF-$
- Updates treated as delete old value, insert new value

## Execution of Commands

---

Commands to consider:

begin\_transaction  
read  
write  
commit  
abort  
recover

Independent of execution  
strategy for LRM

## Execution Strategies

---

### ■ Dependent upon

- Can the buffer manager decide to write some of the buffer pages being accessed by a transaction into stable storage or does it wait for LRM to instruct it?
  - ◆ fix/no-fix decision
- Does the LRM force the buffer manager to write certain buffer pages into stable database at the end of a transaction's execution?
  - ◆ flush/no-flush decision

### ■ Possible execution strategies:

- no-fix/no-flush
- no-fix/flush
- fix/no-flush
- fix/flush

## No-Fix/No-Flush

---

### ■ Abort

- ➡ Buffer manager may have written some of the updated pages into stable database
- ➡ LRM performs **transaction undo** (or **partial undo**)

### ■ Commit

- ➡ LRM writes an "end\_of\_transaction" record into the log.

### ■ Recover

- ➡ For those transactions that have both a "begin\_transaction" and an "end\_of\_transaction" record in the log, a partial redo is initiated by LRM
- ➡ For those transactions that only have a "begin\_transaction" in the log, a **global undo** is executed by LRM

## No-Fix/Flush

---

### ■ Abort

- ➡ Buffer manager may have written some of the updated pages into stable database
- ➡ LRM performs transaction undo (or partial undo)

### ■ Commit

- ➡ LRM issues a **flush** command to the buffer manager for all updated pages
- ➡ LRM writes an "end\_of\_transaction" record into the log.

### ■ Recover

- ➡ No need to perform redo
- ➡ Perform global undo

## Fix/No-Flush

---

### ■ Abort

- ➡ None of the updated pages have been written into stable database
- ➡ Release the **fixed** pages

### ■ Commit

- ➡ LRM writes an "end\_of\_transaction" record into the log.
- ➡ LRM sends an **unfix** command to the buffer manager for all pages that were previously **fixed**

### ■ Recover

- ➡ Perform partial redo
- ➡ No need to perform global undo

## Fix/Flush

---

### ■ Abort

- ➡ None of the updated pages have been written into stable database
- ➡ Release the **fixed** pages

### ■ Commit (the following have to be done atomically)

- ➡ LRM issues a **flush** command to the buffer manager for all updated pages
- ➡ LRM sends an **unfix** command to the buffer manager for all pages that were previously **fixed**
- ➡ LRM writes an "end\_of\_transaction" record into the log.

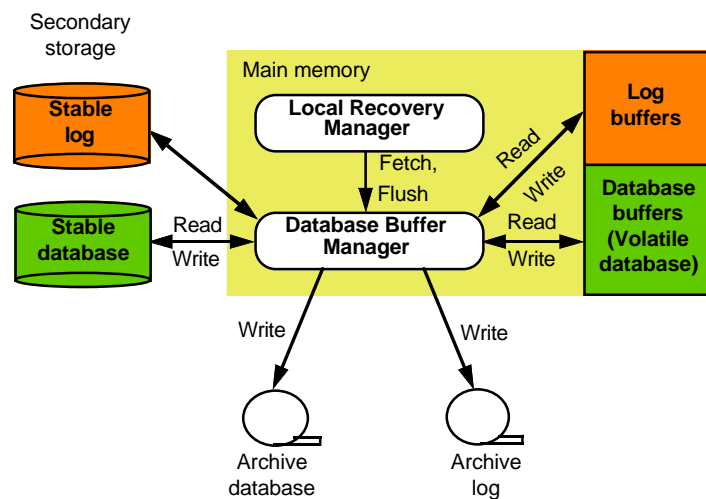
### ■ Recover

- ➡ No need to do anything

## Checkpoints

- Simplifies the task of determining actions of transactions that need to be undone or redone when a failure occurs.
- A checkpoint record contains a list of active transactions.
- Steps:
  - ❶ Write a begin\_checkpoint record into the log
  - ❷ Collect the checkpoint data into the stable storage
  - ❸ Write an end\_checkpoint record into the log

## Media Failures – Full Architecture





## 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

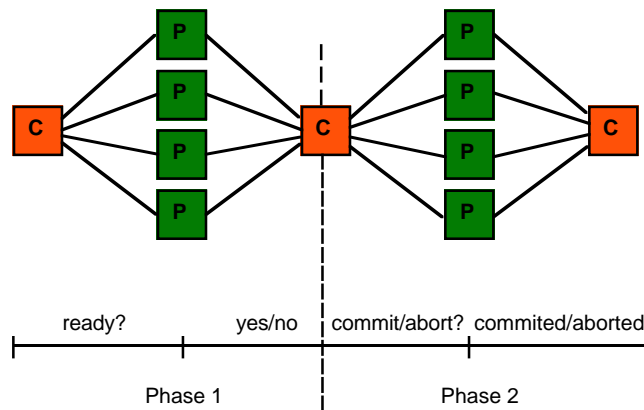
*Phase 2*: Everybody writes the results into the database

- ➡ **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

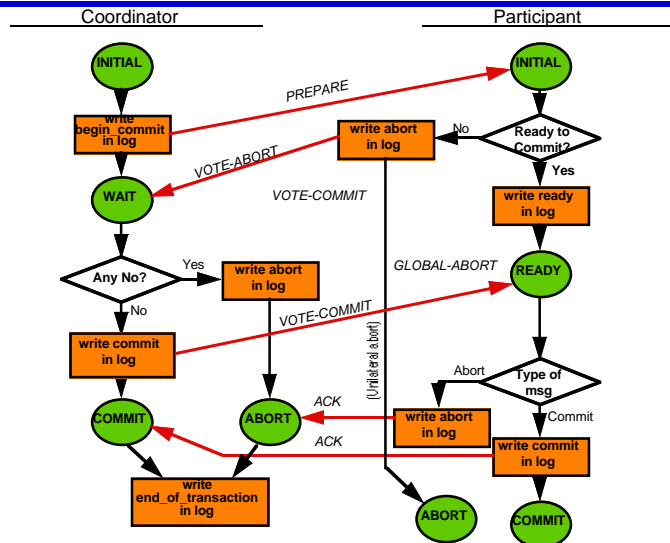
### Global Commit Rule:

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

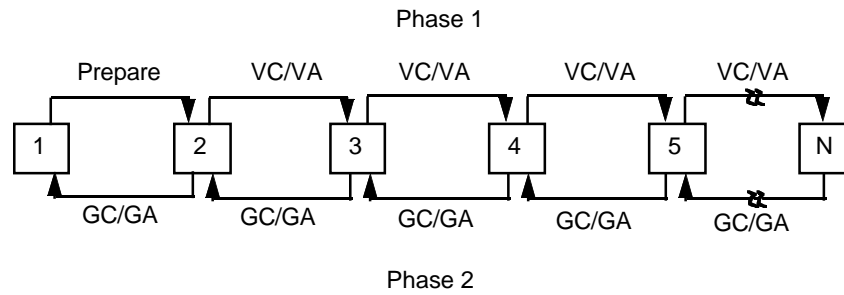
## Centralized 2PC



## 2PC Protocol Actions

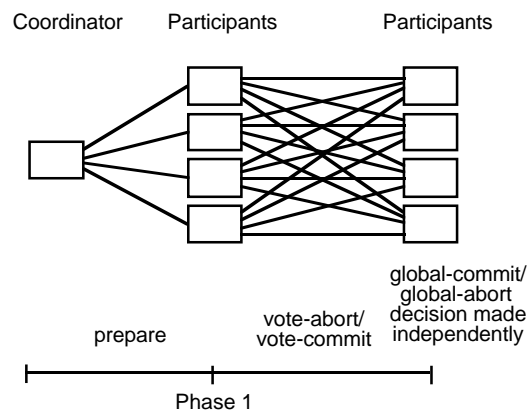


## Linear 2PC

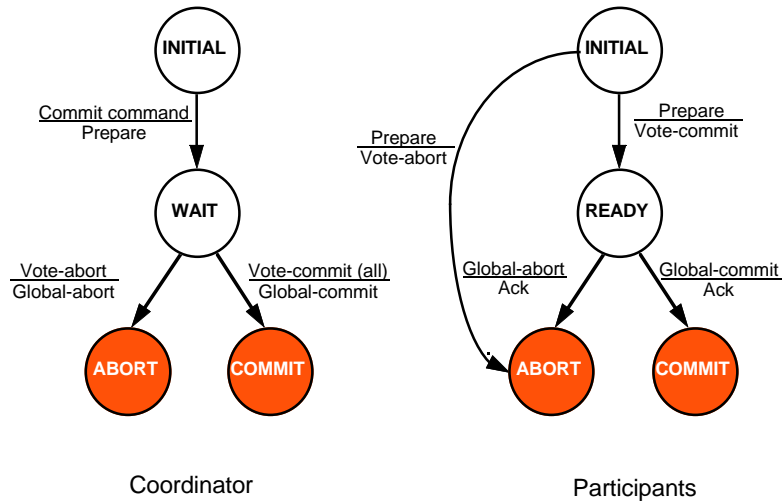


VC: Vote-Commit, VA: Vote-Abort, GC: Global-commit, GA: Global-abort

## Distributed 2PC

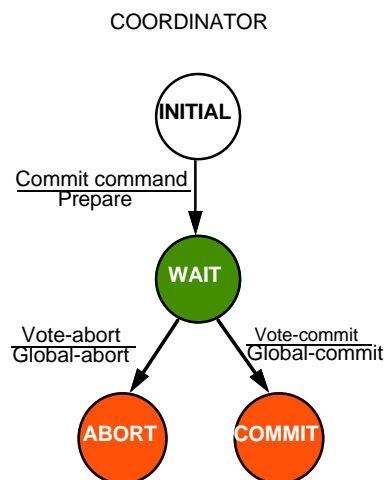


## State Transitions in 2PC



## 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



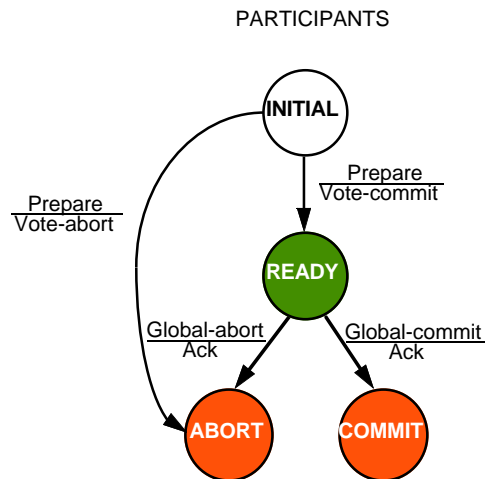
## Site Failures - 2PC Termination

### ■ Timeout in INITIAL

- Coordinator must have failed in INITIAL state
- Unilaterally abort

### ■ Timeout in READY

- Stay blocked



## Site Failures - 2PC Recovery

### ■ Failure in INITIAL

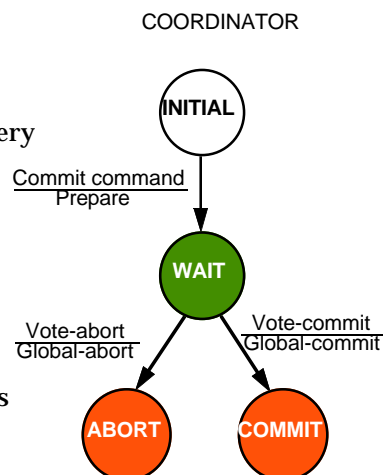
- Start the commit process upon recovery

### ■ Failure in WAIT

- Restart the commit process upon recovery

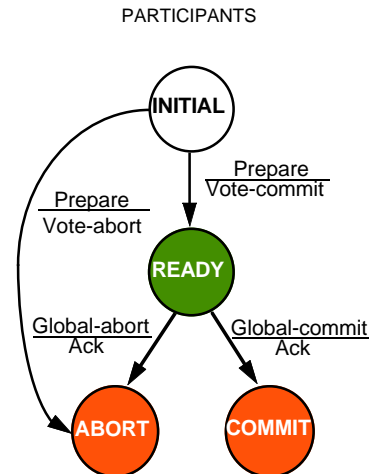
### ■ Failure in ABORT or COMMIT

- 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



## 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

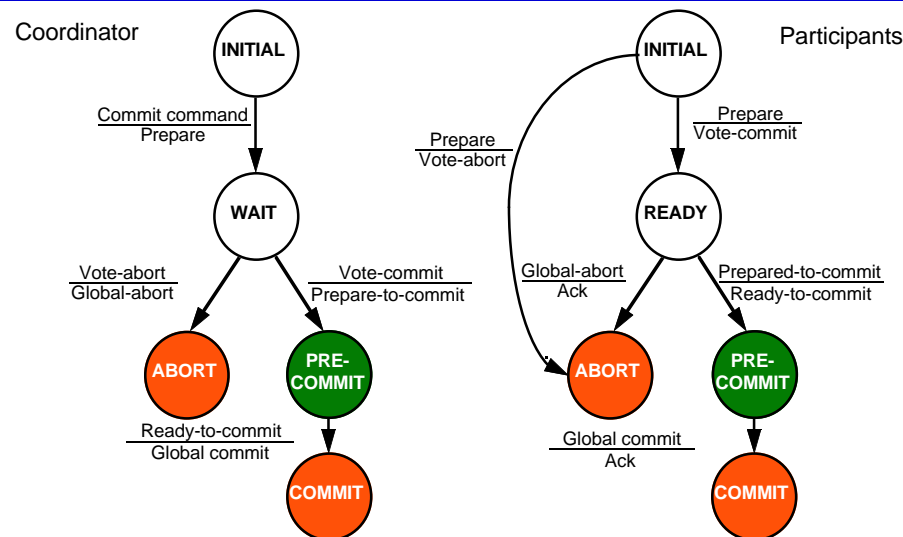
---

- 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

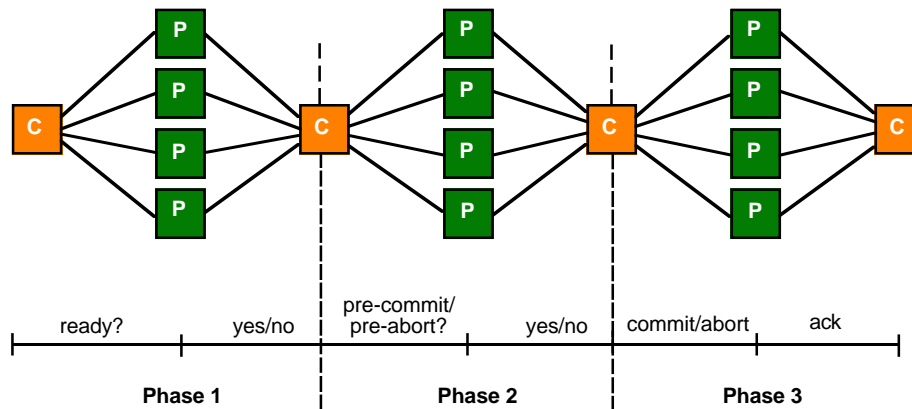
- 3PC is non-blocking.
- A commit protocols is non-blocking iff
  - it is synchronous within one state transition, and
  - its state transition diagram contains
    - ◆ no state which is "adjacent" to both a commit and an abort state, and
    - ◆ no non-committable state which is "adjacent" to a commit 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

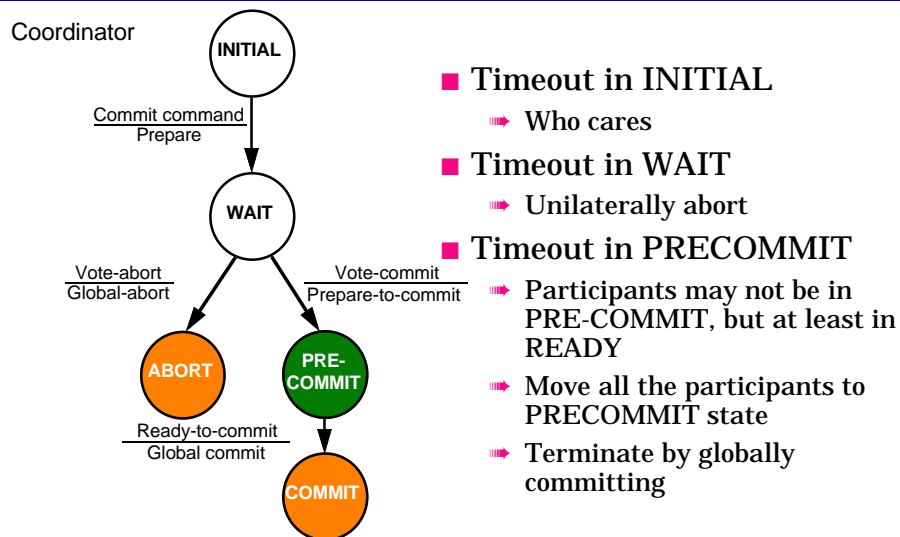




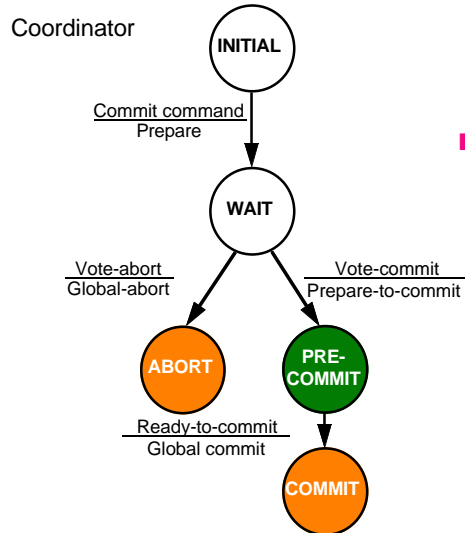
## Communication Structure



## Site Failures – 3PC Termination



## Site Failures – 3PC Termination



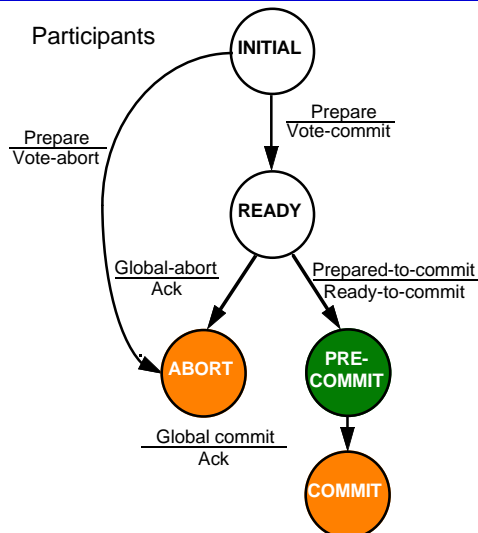
### ■ Timeout in ABORT or COMMIT

- ➡ Just ignore and treat the transaction as completed
- ➡ participants are either in PRECOMMIT or READY state and can follow their termination protocols

Distributed Database Management

© 1994 M. Tamer Özsu & Patrick Valduriez Page 7. 115

## Site Failures – 3PC Termination



### ■ Timeout in INITIAL

- ➡ Coordinator must have failed in INITIAL state
- ➡ Unilaterally abort

### ■ Timeout in READY

- ➡ Voted to commit, but does not know the coordinator's decision
- ➡ Elect a new coordinator and terminate using a special protocol

### ■ Timeout in PRECOMMIT

- ➡ Handle it the same as timeout in READY state

Distributed Database Management

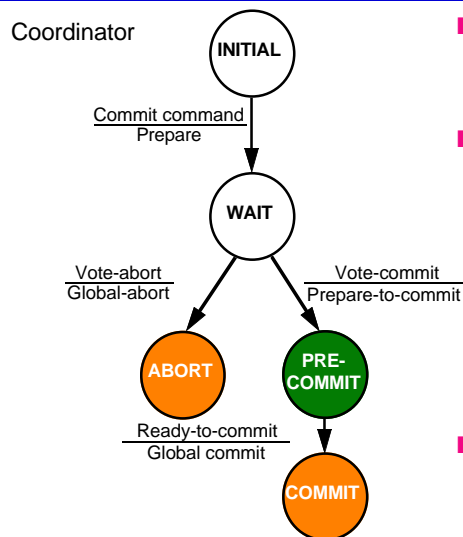
© 1994 M. Tamer Özsu & Patrick Valduriez Page 7. 116

## Termination Protocol Upon Coordinator Election

New coordinator can be in one of four states: WAIT, PRECOMMIT, COMMIT, ABORT

- ❶ Coordinator sends its state to all of the participants asking them to assume its state.
- ❷ Participants “back-up” and reply with appropriate messages, except those in ABORT and COMMIT states. Those in these states respond with “Ack” but stay in their states.
- ❸ Coordinator guides the participants towards termination:
  - ◆ If the new coordinator is in the WAIT state, participants can be in INITIAL, READY, ABORT or PRECOMMIT states. New coordinator globally aborts the transaction.
  - ◆ If the new coordinator is in the PRECOMMIT state, the participants can be in READY, PRECOMMIT or COMMIT states. The new coordinator will globally commit the transaction.
  - ◆ If the new coordinator is in the ABORT or COMMIT states, at the end of the first phase, the participants will have moved to that state as well.

## Site Failures – 3PC Recovery



### ■ Failure in INITIAL

- ➡ start commit process upon recovery

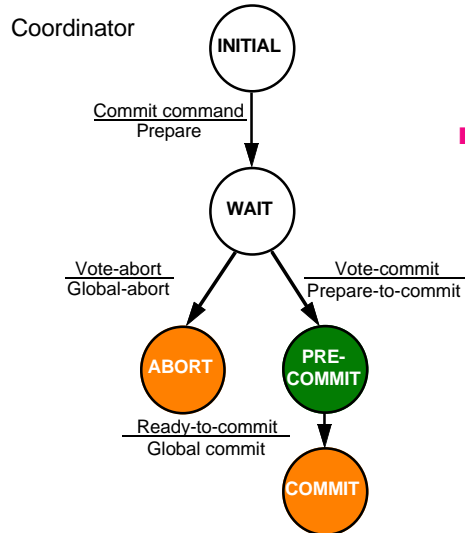
### ■ Failure in WAIT

- ➡ the participants may have elected a new coordinator and terminated the transaction
- ➡ the new coordinator could be in WAIT or ABORT states ⇒ transaction aborted
- ➡ ask around for the fate of the transaction

### ■ Failure in PRECOMMIT

- ➡ ask around for the fate of the transaction

## Site Failures – 3PC Recovery



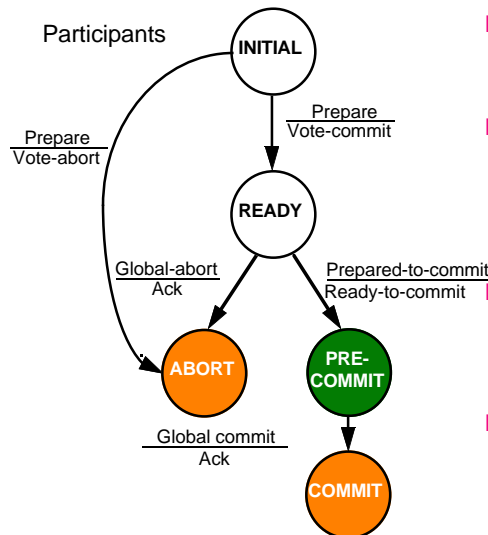
### ■ Failure in COMMIT or ABORT

- Nothing special if all the acknowledgements have been received; otherwise the termination protocol is involved

Distributed Database Management

© 1994 M. Tamer Özsu & Patrick Valduriez Page 7. 119

## Site Failures – 3PC Recovery



### ■ Failure in INITIAL

- unilaterally abort upon recovery

### ■ Failure in READY

- the coordinator has been informed about the local decision
- upon recovery, ask around

### ■ Failure in PRECOMMIT

- ask around to determine how the other participants have terminated the transaction

### ■ Failure in COMMIT or ABORT

- no need to do anything

Distributed Database Management

© 1994 M. Tamer Özsu & Patrick Valduriez Page 7. 120

## 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

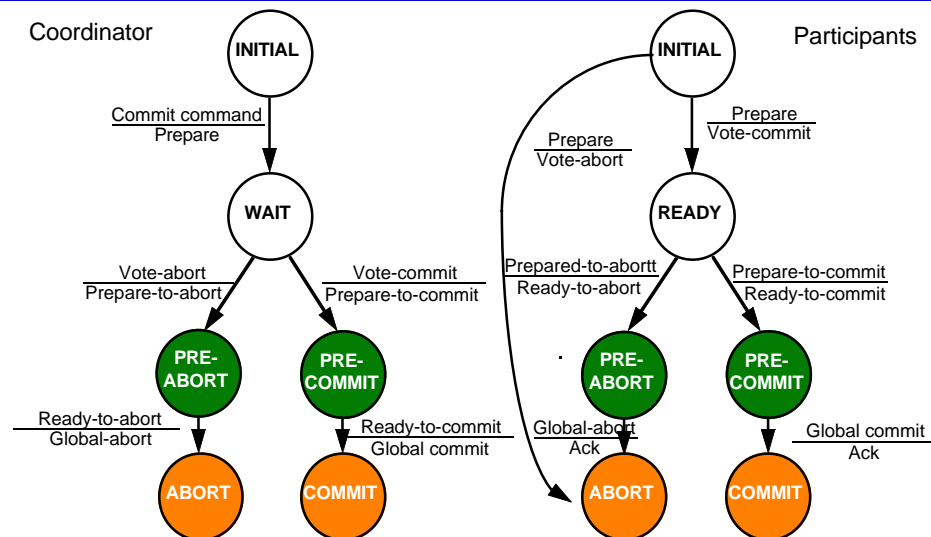
---

- 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  $V_i$ .
- 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  $V_a$

## State Transitions in Quorum Protocols



## 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  $\sum_i V_i = V$
  - 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

## Open Problems

---

- Replication protocols
  - ⇒ experimental validation
  - ⇒ replication of computation and communication
- Transaction models
  - ⇒ changing requirements
    - ◆ cooperative sharing vs. competitive sharing
    - ◆ interactive transactions
    - ◆ longer duration
    - ◆ complex operations on complex data
  - ⇒ relaxed semantics
    - ◆ non-serializable correctness criteria

## Transaction Model Design Space

---

