

Transactions and Concurrency Control

CS432: Distributed Systems
Spring 2017

Reading

- Chapter 16, 17 (17.2,17.4,17.5) [Coulouris '11]
- Chapter 12 [Ozsu '10]

Objectives

- Learn about the following:
 - Transactions in distributed systems
 - Techniques that guarantees concurrency control

Outline

- Introduction
- Transactions
- Concurrency Control
 - Two Phase Locking
 - Timestamp Ordering
- Distributed Commit and Recovery and Termination

Introduction

- Distributed machines cooperating / collaborating together are required to synchronize with each other
- Synchronization using clocks:
 - Physical clocks (actual time)
 - Finding a relative order of events (logical time)
- Global states: finding out whether a particular property is true of a distributed system as it executes. Example: detecting deadlock
- Election algorithms to find a coordinator for the distributed system
- Sharing in distributed systems and the necessity for mutual exclusion

Transactions and Concurrency Control

- In a system that has **shared objects** that are managed by servers and accessed by **multiple clients**, concurrency control is required
- **Transaction**: a sequence of server operations that is guaranteed by the server to be atomic in the presence of multiple clients and server crashes
(A: Atomicity, C: Consistency, I: Isolation, D: Durability)
- **Concurrency control**: it deals with the isolation and consistency properties of transactions
 - Isolation: each transaction sees a consistent view of the shared data at all times
 - Consistency of transactions (correctness): a transaction maps from a consistent state to another consistent state

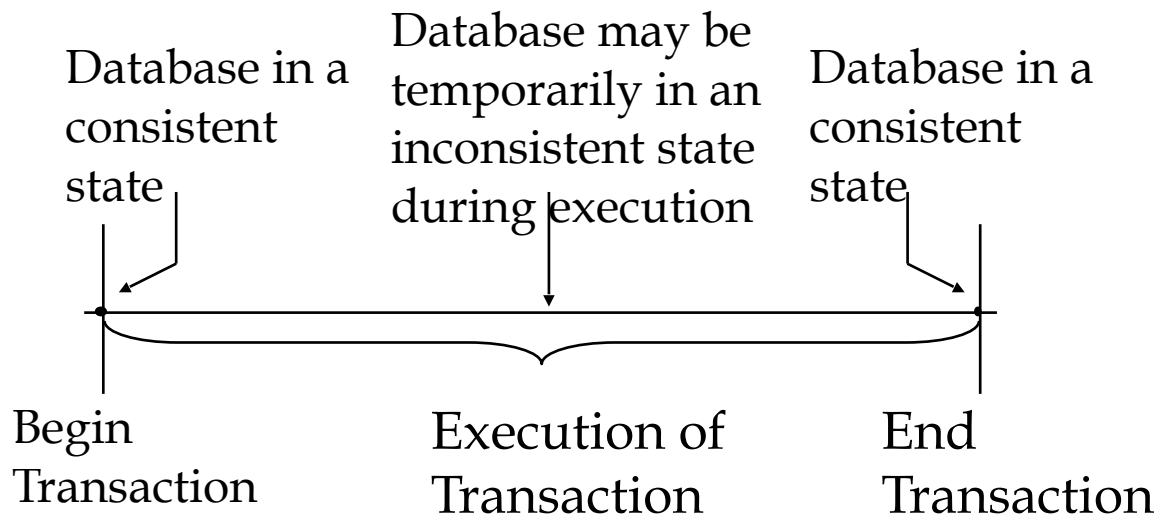
Outline

- Introduction
- **Transactions**
 - Centralized Transactions
 - Principles of Transactions
 - Flat and Nested Transactions
 - Distributed Transactions
- **Concurrency Control**
 - Two Phase Locking
 - Timestamp Ordering
- Distributed Commit and Recovery and Termination

Transactions

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

- concurrency transparency
- failure transparency



Example: Atomic Operations

deposit(amount)

deposit *amount* in the account

withdraw(amount)

withdraw *amount* from the account

getBalance() → *amount*

return the balance of the account

setBalance(amount)

set the balance of the account to *amount*

Operations of the *Branch* interface

create(name) → *account*

create a new account with a given name

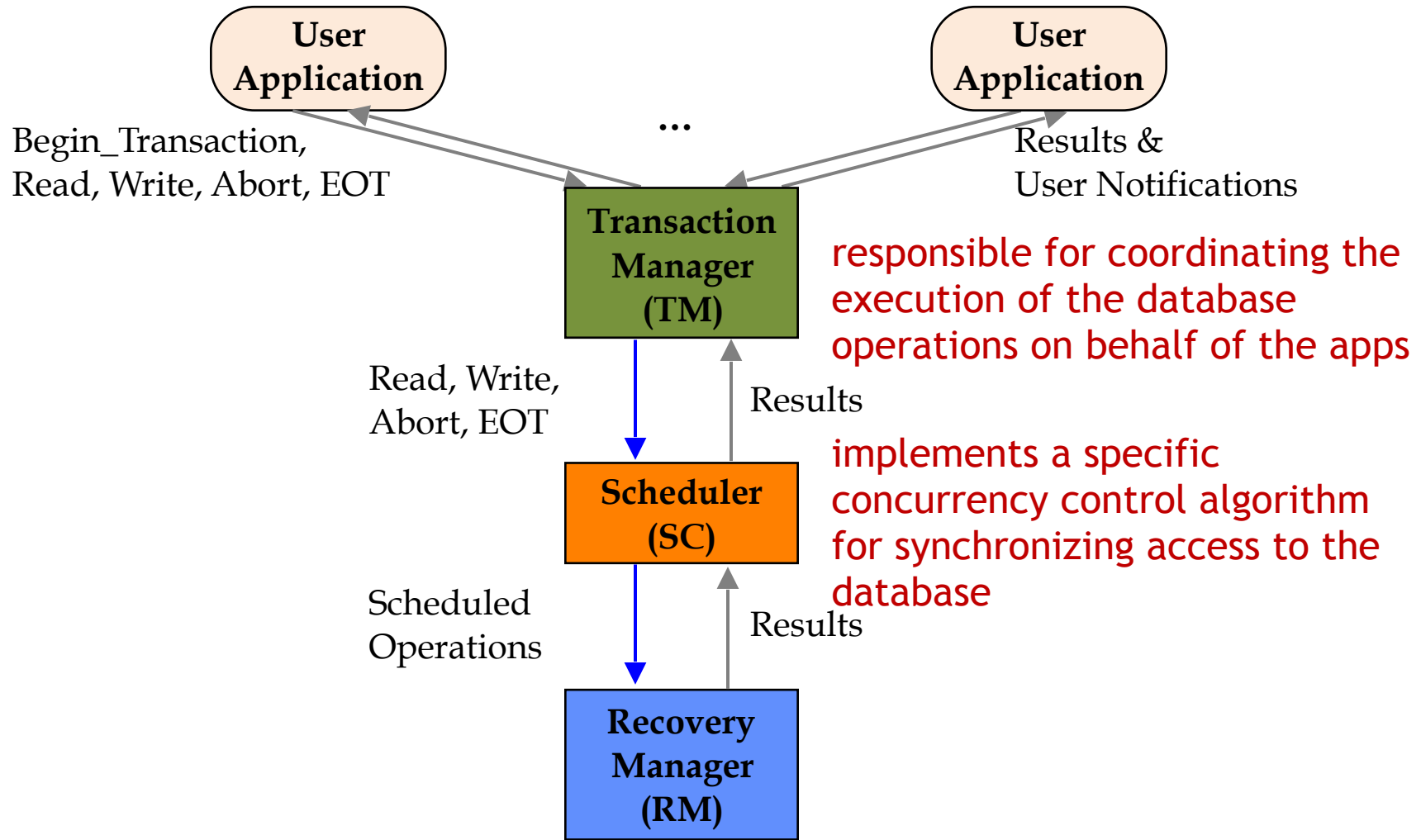
lookUp(name) → *account*

return a reference to the account with the given name

branchTotal() → *amount*

return the total of all the balances at the branch

Centralized Transaction Execution



Principles of Transactions

ATCOMICITY

- All or nothing
- In case of failures, partial results are undone
- Transaction recovery and crash recovery

CONSISTENCY

- No violation of integrity constraints (correctness)

ISOLATION

- Concurrent changes invisible \Rightarrow serializable

DURABILITY

- Committed updates persist
- Database recovery

Example

- Transactions are created and managed by a coordinator (TM)
- Operations in a coordinator interface:

openTransaction() \rightarrow *trans*;

Starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans) \rightarrow (*commit*, *abort*);

Ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);

Aborts the transaction.

Lost Update Problem

- Accounts A, B, and C with initial balances equal \$100, \$200, and \$300, respectively

Transaction T:	Transaction U:
<i>balance = b.getBalance();</i> <i>b.setBalance(balance*1.1);</i> <i>a.withdraw(balance/10)</i>	<i>balance = b.getBalance();</i> <i>b.setBalance(balance*1.1);</i> <i>c.withdraw(balance/10)</i>
<i>balance = b.getBalance();</i> \$200	<i>balance = b.getBalance();</i> \$200
	<i>b.setBalance(balance*1.1);</i> \$220
<i>b.setBalance(balance*1.1);</i> \$220	
<i>a.withdraw(balance/10)</i> \$80	<i>c.withdraw(balance/10)</i> \$280

Example Serial Order

Transaction T:		Transaction U:	
<i>balance = b.getBalance()</i>		<i>balance = b.getBalance()</i>	
<i>b.setBalance(balance*1.1)</i>		<i>b.setBalance(balance*1.1)</i>	
<i>a.withdraw(balance/10)</i>		<i>c.withdraw(balance/10)</i>	
<i>balance = b.getBalance()</i>	\$200	<i>balance = b.getBalance()</i>	\$220
<i>b.setBalance(balance*1.1)</i>	\$220	<i>b.setBalance(balance*1.1)</i>	\$242
<i>a.withdraw(balance/10)</i>	\$80	<i>c.withdraw(balance/10)</i>	\$278

Inconsistent Retrieval Problem

Transaction V:		Transaction W:	
<i>a.withdraw(100)</i>		<i>aBranch.branchTotal()</i>	
<i>b.deposit(100)</i>			
<i>a.withdraw(100);</i>	\$100	<i>total = a.getBalance()</i>	\$100
		<i>total = total + b.getBalance()</i>	\$300
		<i>total = total + c.getBalance()</i>	
<i>b.deposit(100)</i>	\$300	•	
		•	

Example Serial Order

Transaction V:		Transaction W:	
<i>a.withdraw(100);</i> <i>b.deposit(100)</i>		<i>aBranch.branchTotal()</i>	
<i>a.withdraw(100);</i>	\$100		
<i>b.deposit(100)</i>	\$300		
		<i>total = a.getBalance()</i>	\$100
		<i>total = total + b.getBalance()</i>	\$400
		<i>total = total + c.getBalance()</i>	
		...	

Transaction Structure

- Flat transaction
 - Consists of a sequence of **primitive** operations embraced between a **begin** and **end** markers

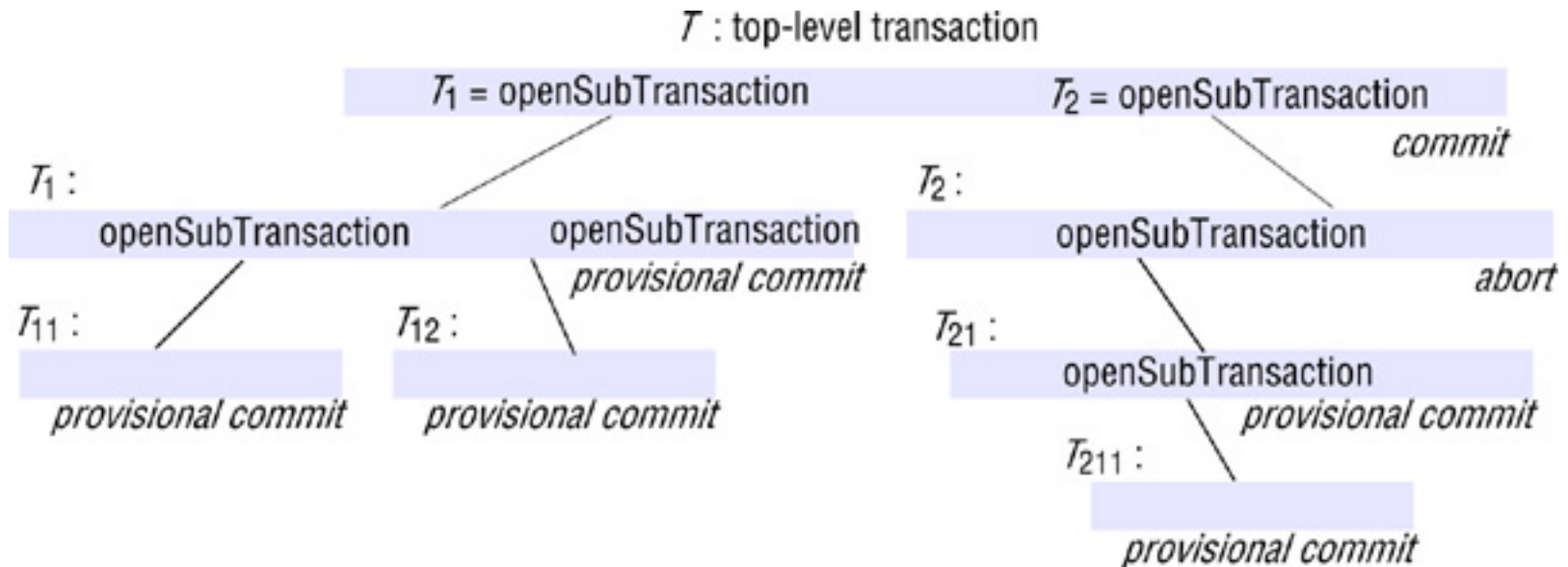
```
Begin_transaction Reservation
...
end
```
- Nested transaction
 - The operations of a transaction may themselves be transactions

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

Nested Transactions

- Have the same properties as their parents and may themselves have other nested transactions
- Introduces concurrency control and recovery concepts to within the transaction
- Types
 - Closed nesting
 - Sub-transactions begin **after** their parents and finish **before** them
 - Committing of a sub-transaction is conditional upon the committing of the parent
 - Open nesting
 - Sub-transactions can execute and commit independently

Nested Transactions Example



- Accessing common objects by sub-transactions is serialized
- Aborting sub-transactions can reflect the top transactions

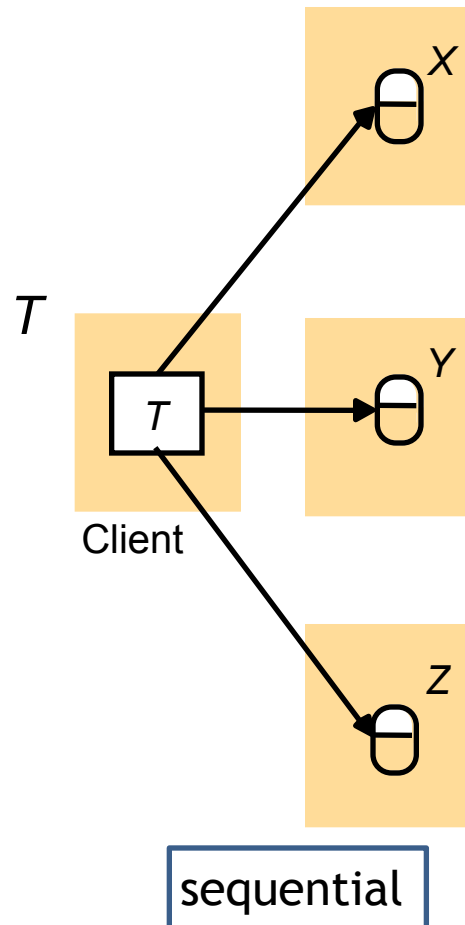
Note: all transactions and sub-transactions are run on a single server

Distributed Transactions

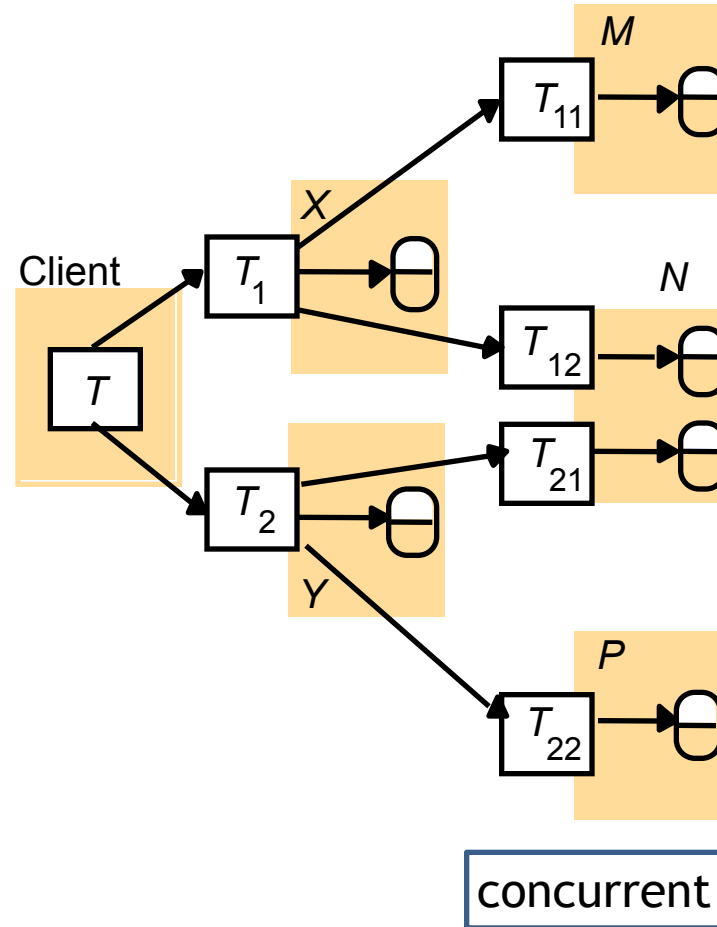
- A client transaction becomes distributed if it invokes operations in several different servers
- Distributed transactions structure:
 - Flat:
 - A client makes requests to more than one server
 - For example to access objects that are available on multiple servers
 - Requests are executed sequentially
 - Nested:
 - Top-level transaction can open sub-transactions, and each sub-transaction can open further sub-transactions down to any depth of nesting
 - Transactions and sub-transactions can run concurrently on different servers

Distributed transactions Example

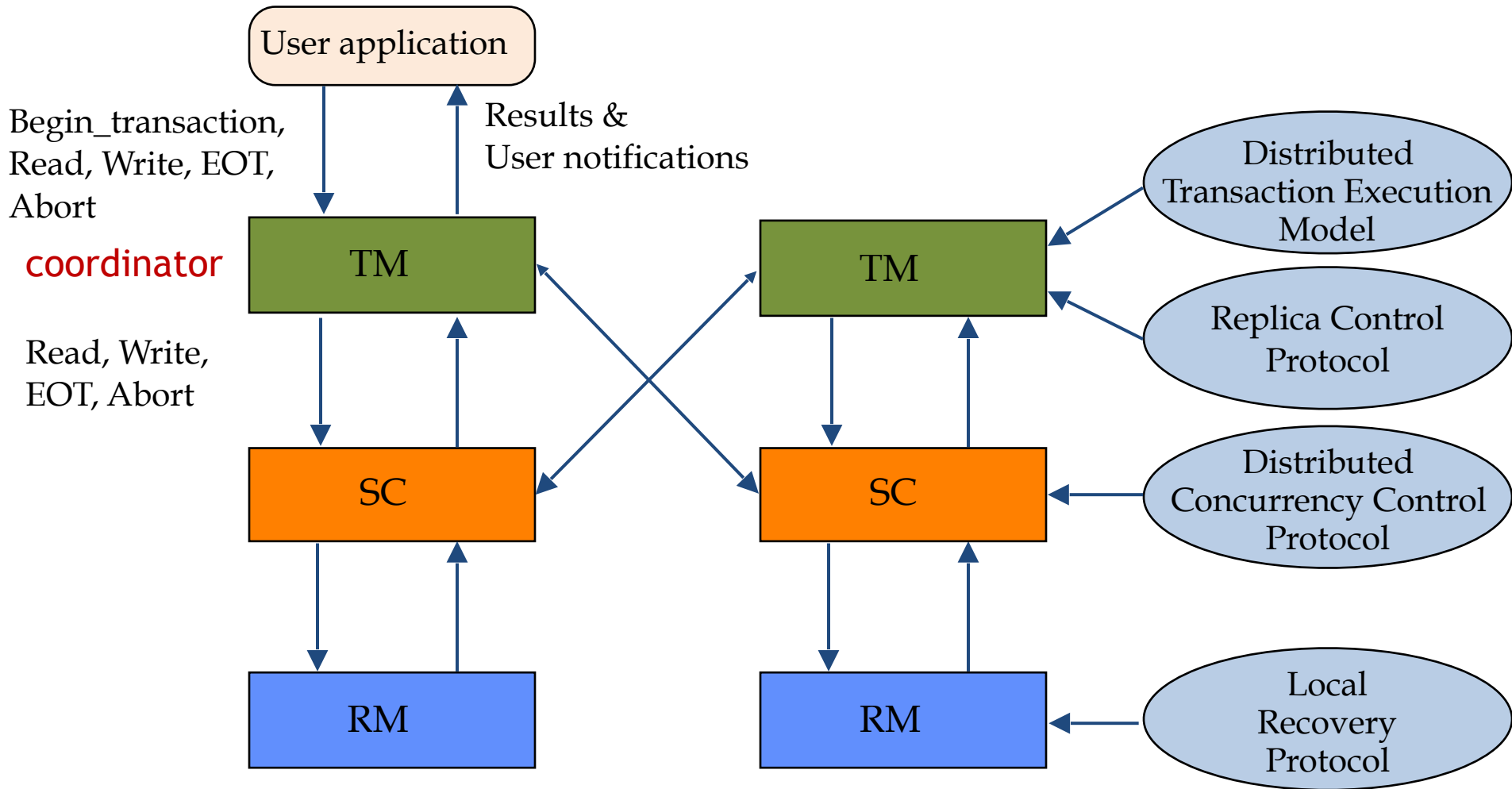
(a) Flat transaction



(b) Nested transactions



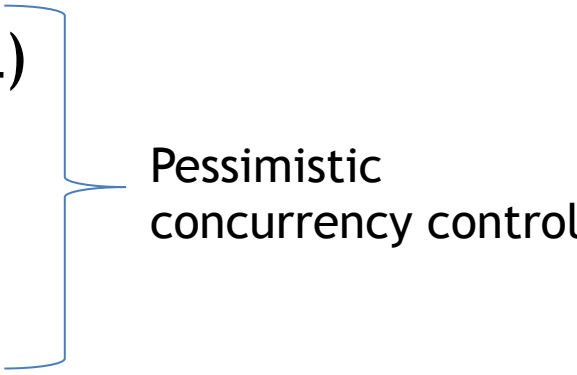
Distributed Transaction Execution



Outline

- Introduction
- Transactions
- **Concurrency Control**
 - Two Phase Locking
 - Timestamp Ordering
- Distributed Commit and Recovery and Termination

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
 - Concurrency Control Algorithms:
 - Two-Phase Locking-based (2PL)
 - Centralized (primary site) 2PL
 - Distributed 2PL
 - Timestamp Ordering (TO)
 - Basic TO
 - Multiversion TO
 - Optimistic Concurrency Control
- 
- Pessimistic
concurrency control

Locking-Based Algorithms

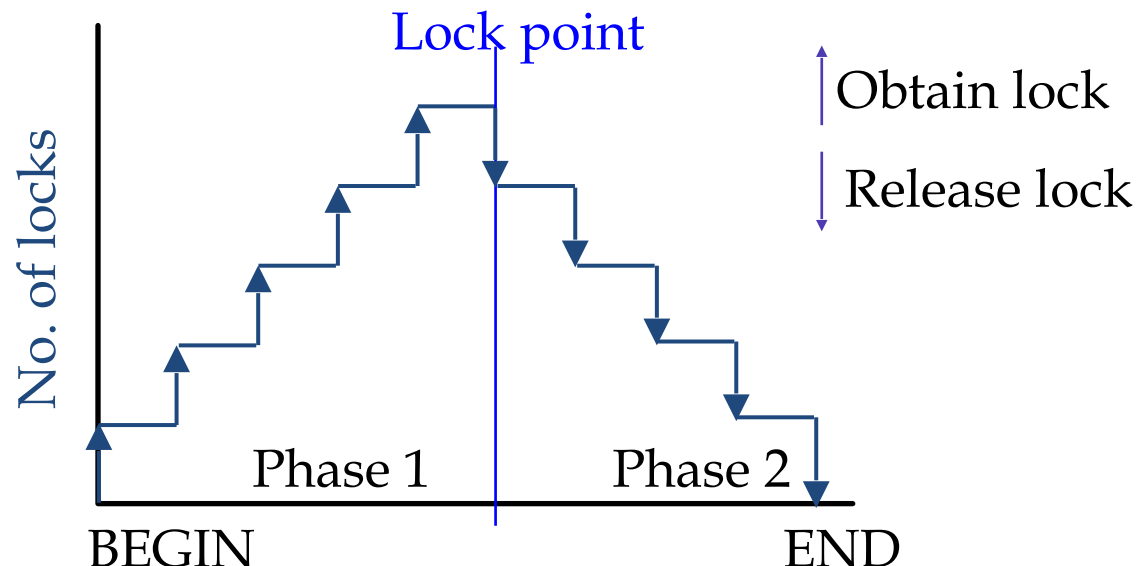
- Transactions indicate their intentions by requesting locks from the scheduler (called **lock manager**)
- Locks are either **read lock** (*rl*) [also called **shared lock**] or **write lock** (*wl*) [also called **exclusive lock**]
- Read locks and write locks conflict (because Read and Write operations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

<i>For one object</i>		<i>Lock requested</i>	
		<i>read</i>	<i>write</i>
<i>Lock already set</i>	<i>none</i>	OK	OK
	<i>read</i>	OK	wait
	<i>write</i>	wait	wait

- Locking works nicely to allow concurrent processing of transactions

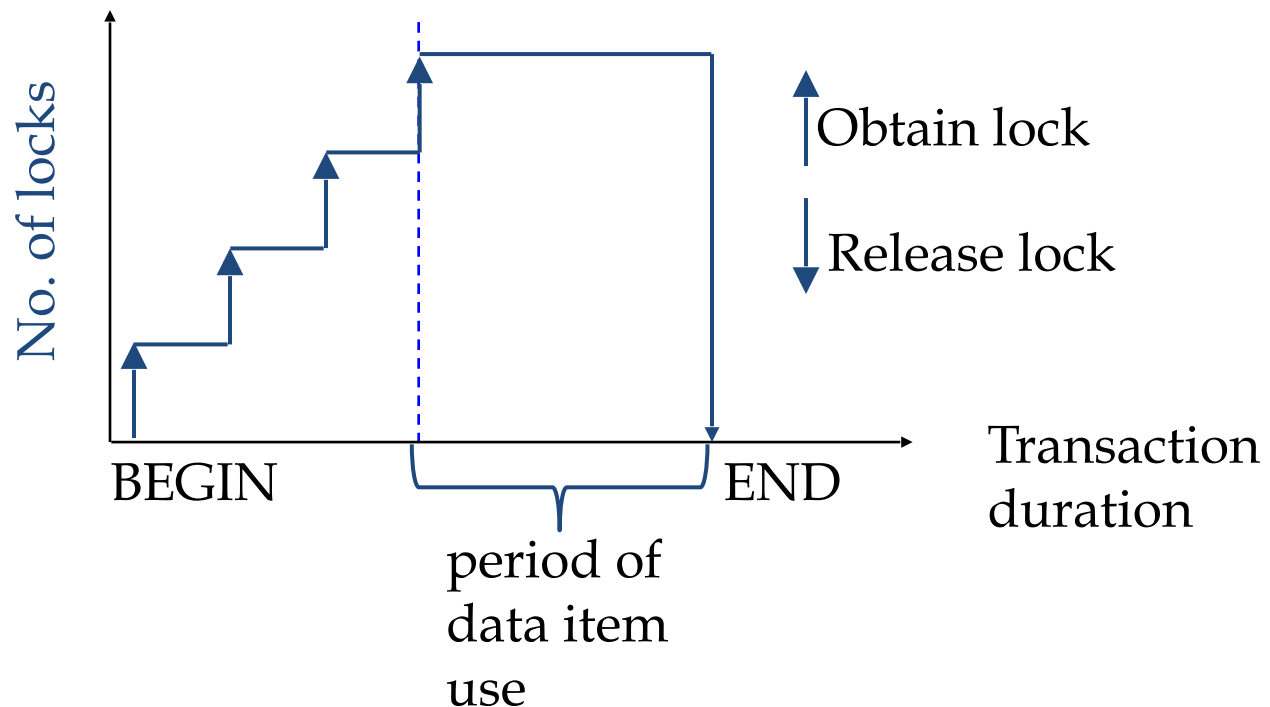
Two-Phase Locking (2PL)

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



Note: Locking-based algorithms may cause deadlocks since they allow exclusive access to resources

Locking Rules for Nested Transactions

- **First Objective:** each set of nested transactions is a single entity that must be prevented from observing the partial effects of any other set of nested transactions
- **Rules:**
 - Every lock that is acquired by a successful sub-transaction is inherited by its parent when it completes
 - Inherited locks are also inherited by ancestors (inheritance passes from child to parent)
- **Reasoning:** ensures that the locks can be held until the top-level transaction has committed or aborted, which prevents members of different sets of nested transactions observing one another's partial effects

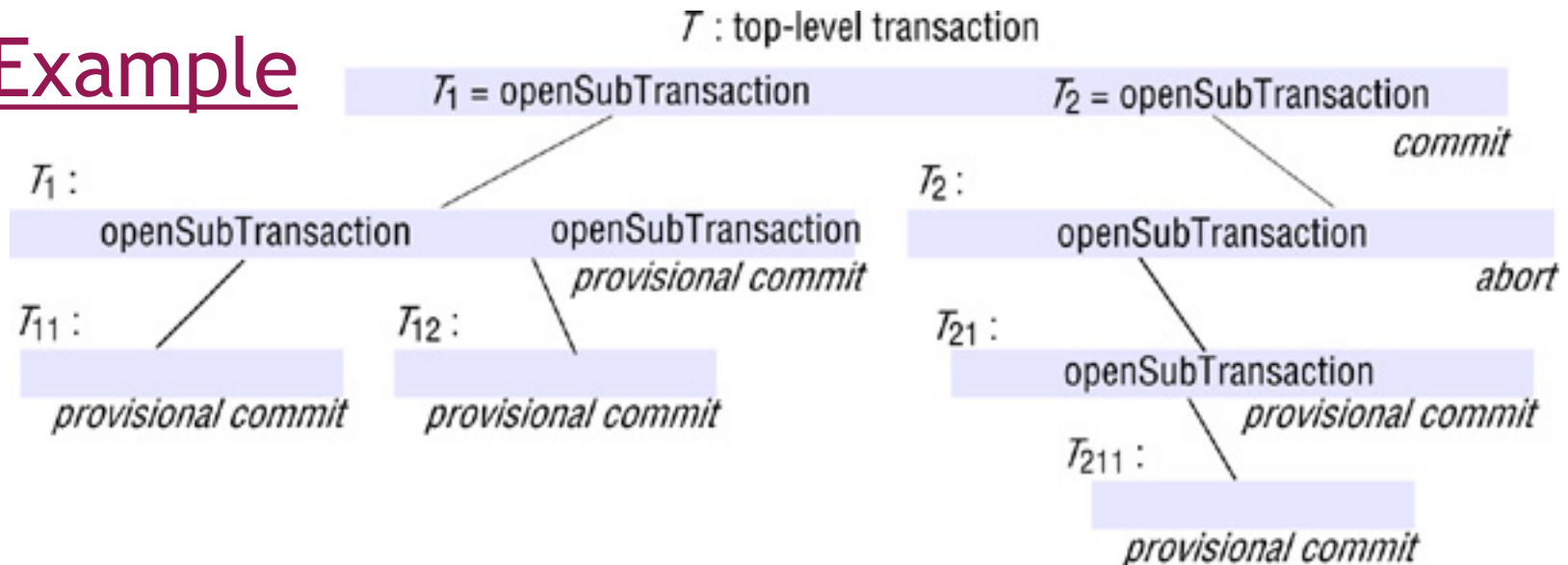
Locking Rules for Nested Transactions

- **Second objective:** each transaction within a set of nested transactions must be prevented from observing the partial effects of the other transactions in the set
- **Rules:**
 - Parent transactions are not allowed to run concurrently with their child transactions. If a parent transaction has a lock on an object, it retains the lock during the time that its child transaction is executing. This means that the child transaction temporarily acquires the lock from its parent for its duration
 - Sub-transactions at the same level are allowed to run concurrently, so when they access the same objects, the locking scheme must serialize their access

Rules: Lock Acquisition and Release

- For a sub-transaction to acquire a read lock on an object, no other active transaction can have a write lock on that object, and the only retainers of a write lock are its ancestors
- For a sub-transaction to acquire a write lock on an object, no other active transaction can have a read or write lock on that object, and the only retainers of read and write locks on that object are its ancestors
- When a sub-transaction commits, its locks are inherited by its parent, allowing the parent to retain the locks in the same mode as the child
- When a sub-transaction aborts, its locks are discarded. If the parent already have the locks, it keeps them

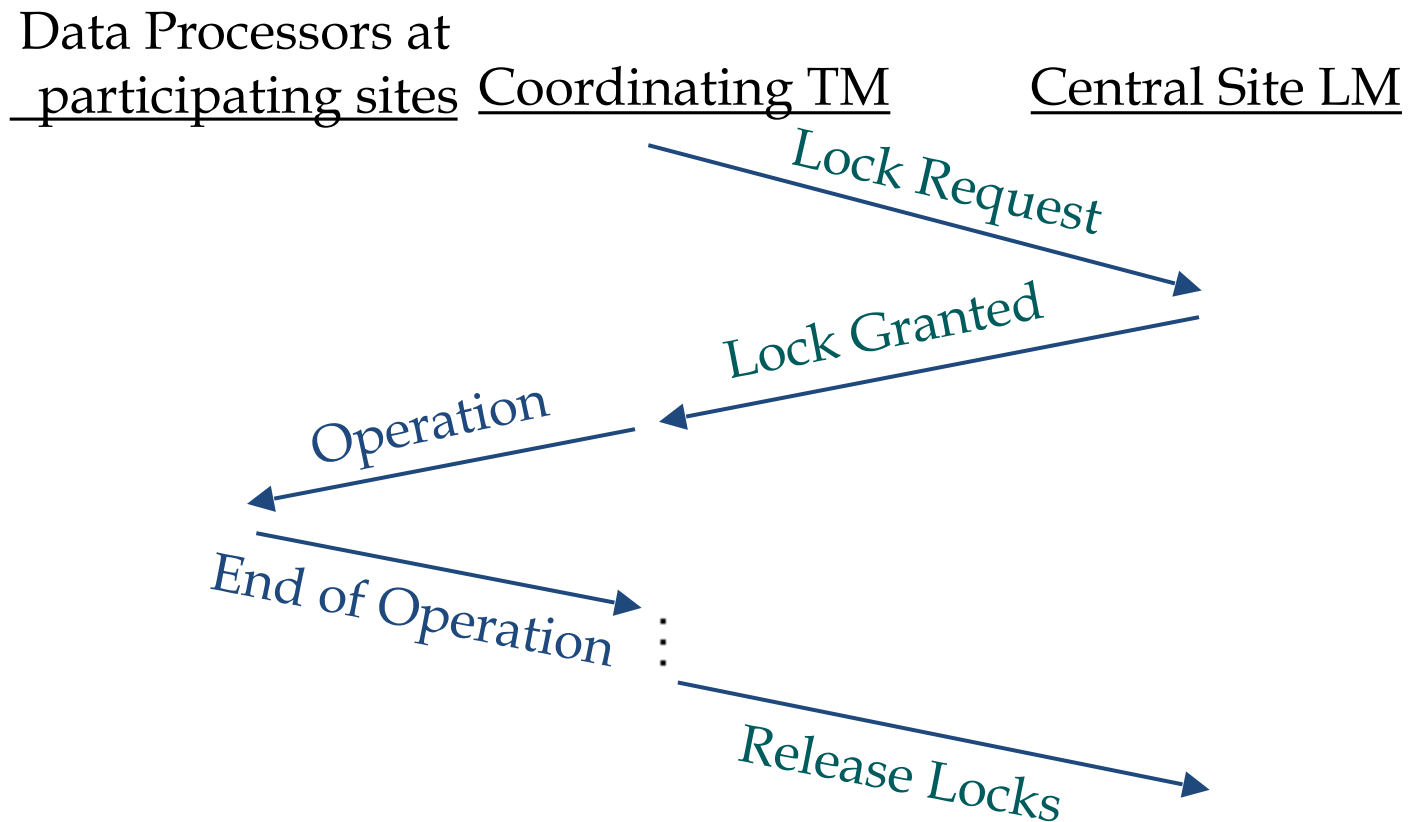
Example



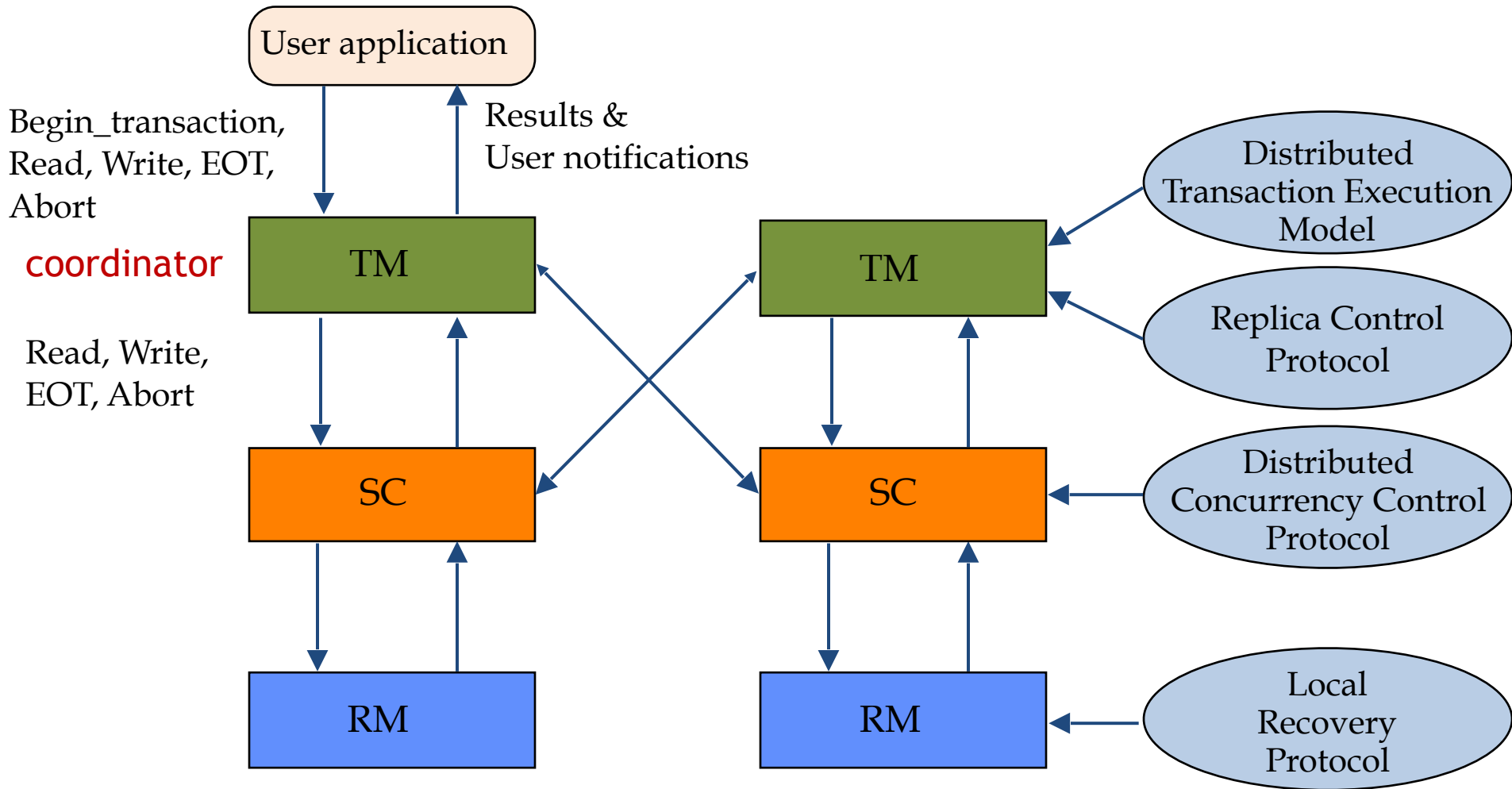
- T_1 , T_2 , and T_{11} access common object, which is not accessed by top transaction T
- T_1 acquire lock
- T_{11} gets the lock from T_1 during each execution and returns it when it completes
- When T_1 completes, T inherits the lock and keeps it until all sub-transactions complete
- T_2 can acquire the lock from T during the period of its execution

Centralized 2PL

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



Distributed Transaction Execution

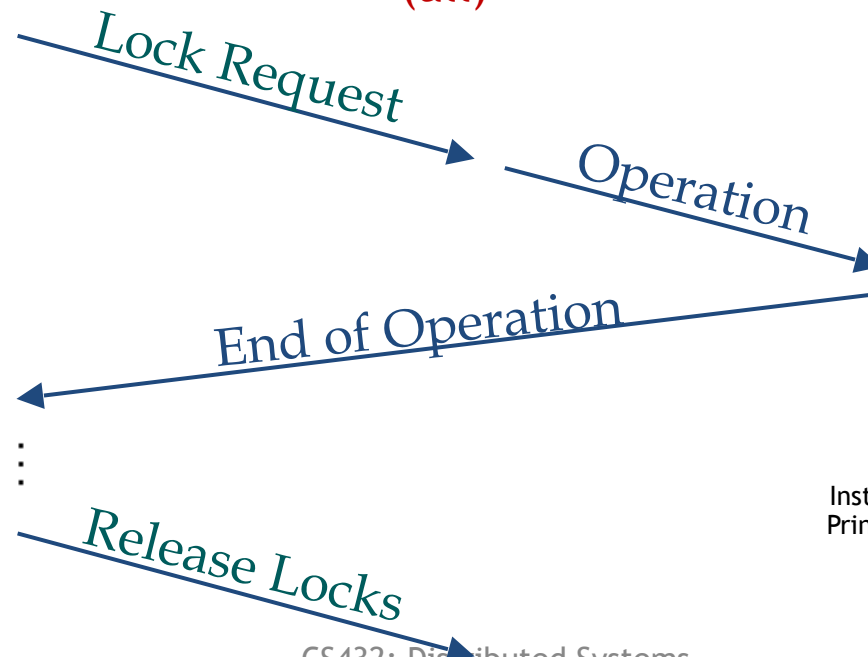


Distributed 2PL

- 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
Writing x requires obtaining write locks for all copies of x

Coordinating TM Participating LMs Participating DPs

(all)



Note: coordinating transaction manager does not wait for a “lock request granted” message

Instructor's Guide for M.T. Ozsü and P.Valduriez,
Principles of Distributed Database Systems, Third
Edition. © Springer 2010

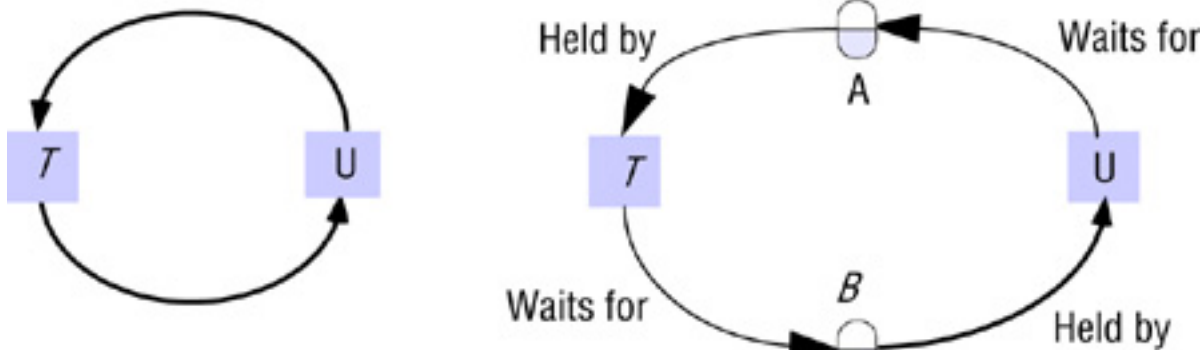
Deadlocks

Transaction <i>T</i>		Transaction <i>U</i>	
Operations	Locks	Operations	Locks
<i>a.deposit(100);</i>	write lock <i>A</i>		
		<i>b.deposit(200)</i>	write lock <i>B</i>
<i>b.withdraw(100)</i>			
...	waits for <i>U</i> 's lock on <i>B</i>	<i>a.withdraw(200);</i>	waits for <i>T</i> 's lock on <i>A</i>
...		...	
...		...	

- Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock
- A wait-for graph can be used to represent the waiting relationships between current transactions

Wait-For Graph

- Nodes represent transactions and the edges represent wait-for relationships between transactions
- There is an edge from node T to U when transaction T is waiting for transaction U to release a lock
- All transactions in a cycle are blocked waiting for locks
- None of the locks are released until a transaction is aborted and hence releases its locks
- A transaction can be involved in more than one cycle



Solution for Deadlocks?

- **Prevention:**

- Lock all needed object at start ==> reduces concurrency, sometimes hard to predict locks that will be needed at start
- Predefined order for acquiring locks ==> premature locking and a reduction in concurrency

- **Upgrade Locks:** introduce an **upgrade lock** for the cases of transactions that acquire a read lock and then upgrade it to write lock (most cases of deadlock)

 **Detection:** for example using wait-for graph, when detected, select a transaction to abort

- **Timeout:** a lock becomes vulnerable after a period of time, if a transaction request it, abort the transaction holding it and release it

Distributed Deadlocks

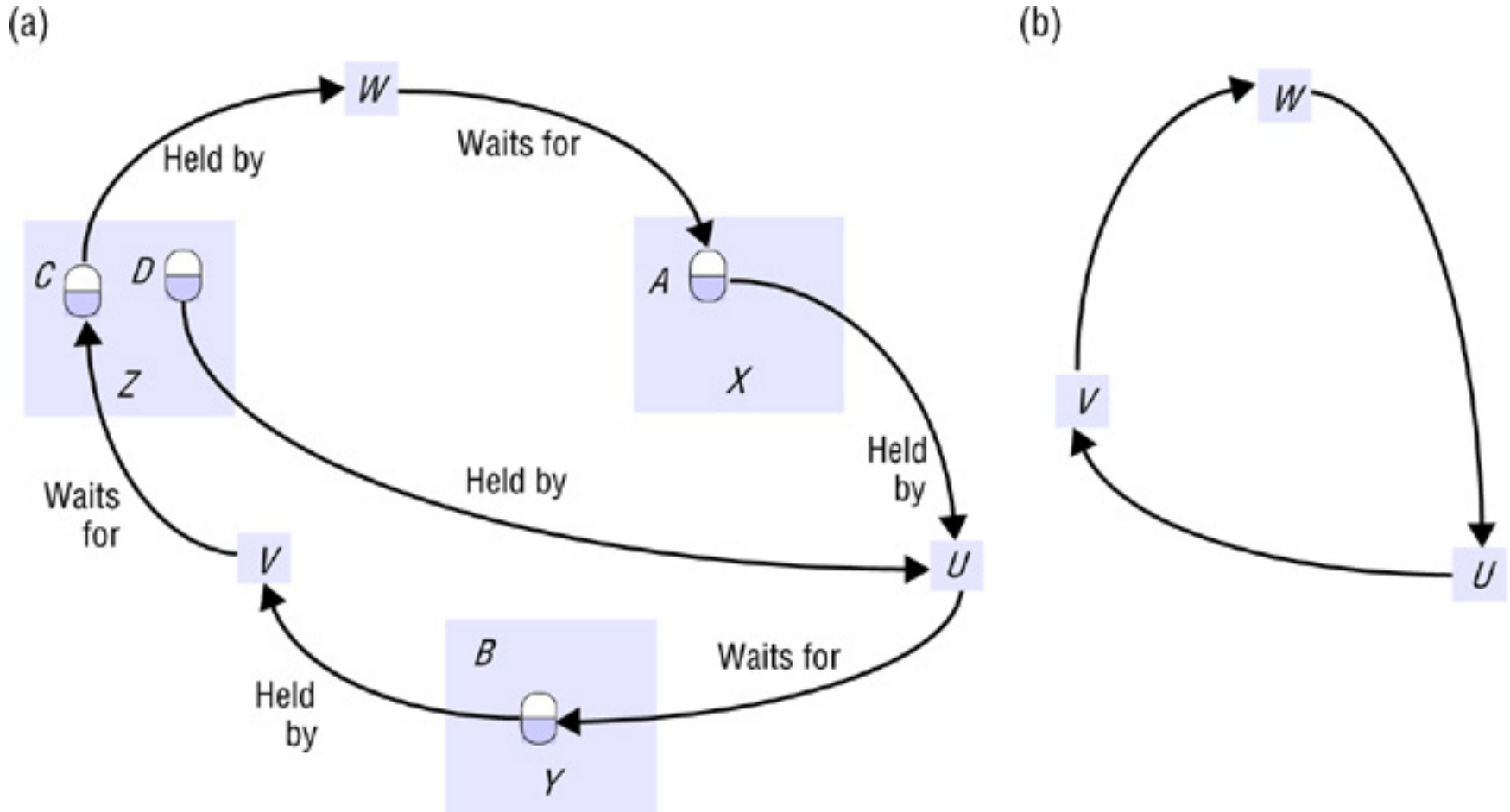
- Need to build a global wait-for graph from local ones
- There can be a cycle in the global wait-for graph that is not in any single local one ==> distributed deadlock
- Use a centralized global deadlock detectors: every interval of time, the local deadlock detectors send their wait-for graphs to it
- Edge chasing:
 - global wait-for graph is not constructed, but each of the servers has knowledge about some of its edges
 - servers find cycles by forwarding messages called probes, which follow the edges of the graph throughout the distributed system

Example: Transactions

U		V		W	
<i>d.deposit(10)</i>	lock <i>D</i>				
		<i>b.deposit(10)</i>	lock <i>B</i>		
<i>a.deposit(20)</i>	lock <i>A</i>		at <i>Y</i>		
	at <i>X</i>				
				<i>c.deposit(30)</i>	lock <i>C</i>
<i>b.withdraw(30)</i>	wait at <i>Y</i>				at <i>Z</i>
		<i>c.withdraw(20)</i>	wait at <i>Z</i>		
				<i>a.withdraw(20)</i>	wait at <i>X</i>

- A managed by X, B managed by Y, C and D managed by Z
- server Y :U → V (added when U requests *b.withdraw(30)*)
- server Z :V → W (added when V requests *c.withdraw(20)*)
- server X :W → U (added when W requests *a.withdraw(20)*)

Example: Wait-For Graph



Timestamp Ordering

- Goal: select a serialization order and execute transactions accordingly
- Advantage: deadlock does not occur
- Trans (T_i) is assigned a globally unique timestamp $ts(T_i)$
- Transaction manager attaches the timestamp to all operations issued by the transaction
- 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 write on x
- **TO Rule.** Given two conflicting operations O_{ij} and O_{kl} belonging, respectively, to transactions T_i and T_k , O_{ij} is executed before O_{kl} if and only if $ts(T_i) < ts(T_k)$

Operation Conflicts for Timestamp Ordering

Rule	T_c	T_i	
1.	write	read	T_c must not <i>write</i> an object that has been <i>read</i> by any T_i where $T_i > T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.
2.	write	write	T_c must not <i>write</i> an object that has been <i>written</i> by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.
3.	read	write	T_c must not <i>read</i> an object that has been <i>written</i> by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.

- 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 $R_i(x)$

$rts(x) \leftarrow \max\{ts(T_i), rts(x)\}$

for $W_i(x)$

if $ts(T_i) < rts(x)$ or $ts(T_i) < wts(x)$

then reject $W_i(x)$

else accept $W_i(x)$

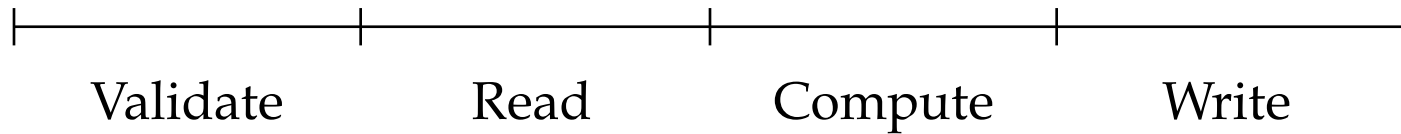
$wts(x) \leftarrow ts(T_i)$

Multiversion Timestamp Ordering

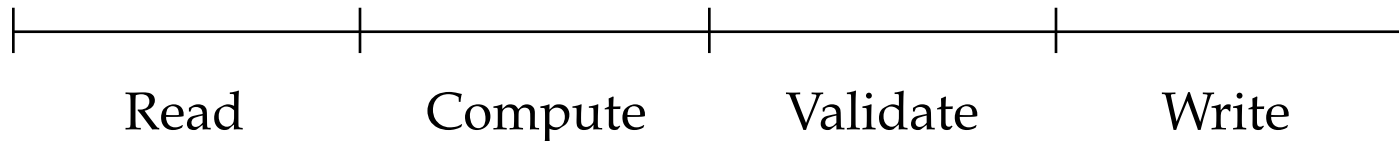
- An attempt to eliminate the restart overhead cost of transactions
- Do not modify the values in the database, create new values
- A $R_i(x)$ is translated into a read on one version of x
 - Find a version of x (say x_v) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$
- A $W_i(x)$ is translated into $W_i(x_w)$ and accepted if the scheduler has not yet processed any $R_j(x_r)$ such that $ts(T_i) < ts(x_r) < ts(T_j)$ (read has been done on older val)

Optimistic Concurrency Control Algorithms

Pessimistic execution



Optimistic execution



Case Study: Dropbox

- Optimistic concurrency control
- Granularity: whole file
- If two users make concurrent updates to the same file, the first write will be accepted and the second rejected
- It provides version history. Users can manually merge updates or restore previous versions

Case Study: Google Apps

- Optimistic concurrency control
- Granularity: character for Google docs, cells for spreadsheet
- A user is aware of others activities, therefore, conflict resolution is left to users
- For multiple users accessing the same cell simultaneously, last update wins

Case Study: Wikipedia

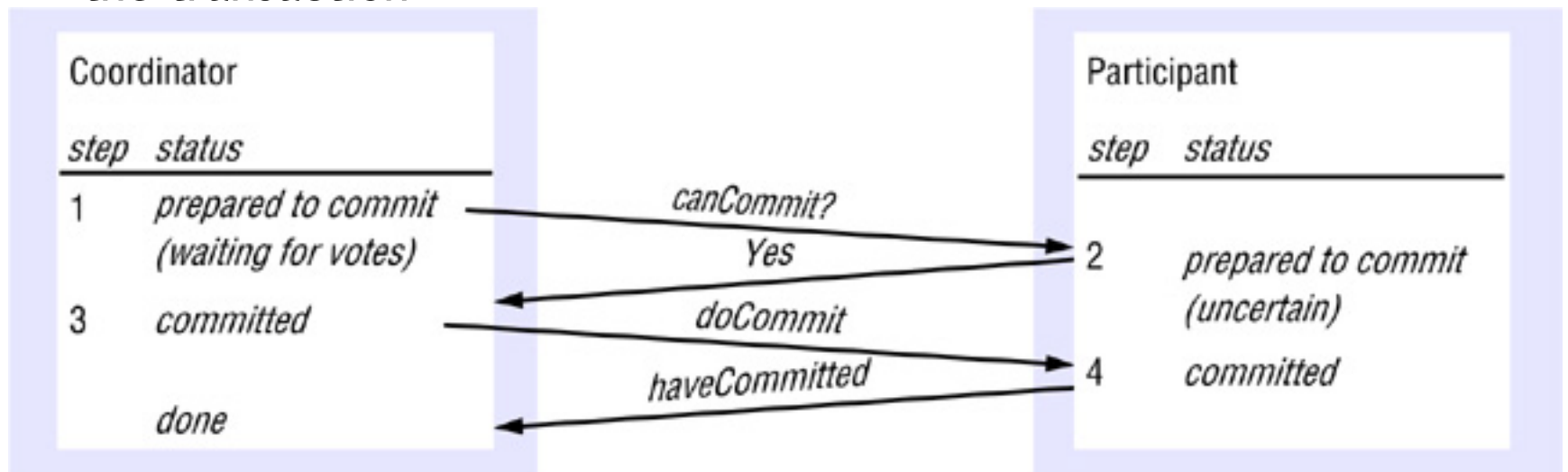
- Optimistic concurrency control
- Users resolve conflicts
- Editors are allowed concurrent access to web pages in which the first write is accepted and a user making a subsequent write is shown an 'edit conflict' screen and asked to resolve the conflicts

Outline

- Introduction
- Transactions
- Concurrency Control
- **Distributed Commit and Recovery and Termination**

Commit Protocol

- Goal: A requirement by the atomicity property. When a distributed transaction comes to an end, either all of its operations are carried out or none of them
- One-Phase Commit: The coordinator communicates the commit or abort request to all of the participants in the transaction and keeps on repeating the request until all of them have acknowledged that they have carried it out
- In Two-Phase commit any participating server can abort its part of the transaction



Distributed Reliability Protocols (1)

- Goal: maintain atomicity and durability of distributed transactions
- Established through a coordinator communicating requests to participants
- Commit protocols
 - How to execute commit command for distributed transactions?
 - Issue: how to ensure atomicity and durability?

Distributed Reliability Protocols (2)

- Termination protocols (unique to distributed systems)
 - 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

Termination vs. Recovery Protocols

- Both are implemented in case of site failures
- Termination protocols address how the operational sites deal with the failure
- Recovery protocols deal with the procedure that the process (coordinator or participant) at the failed site has to go through to recover its state once the site is restarted
- **A non-blocking termination protocol** permits a transaction to terminate at the operational sites without waiting for recovery of the failed site
- **An independent recovery protocol** determines how to terminate a transaction that was executing at the time of a failure without having to consult any other site

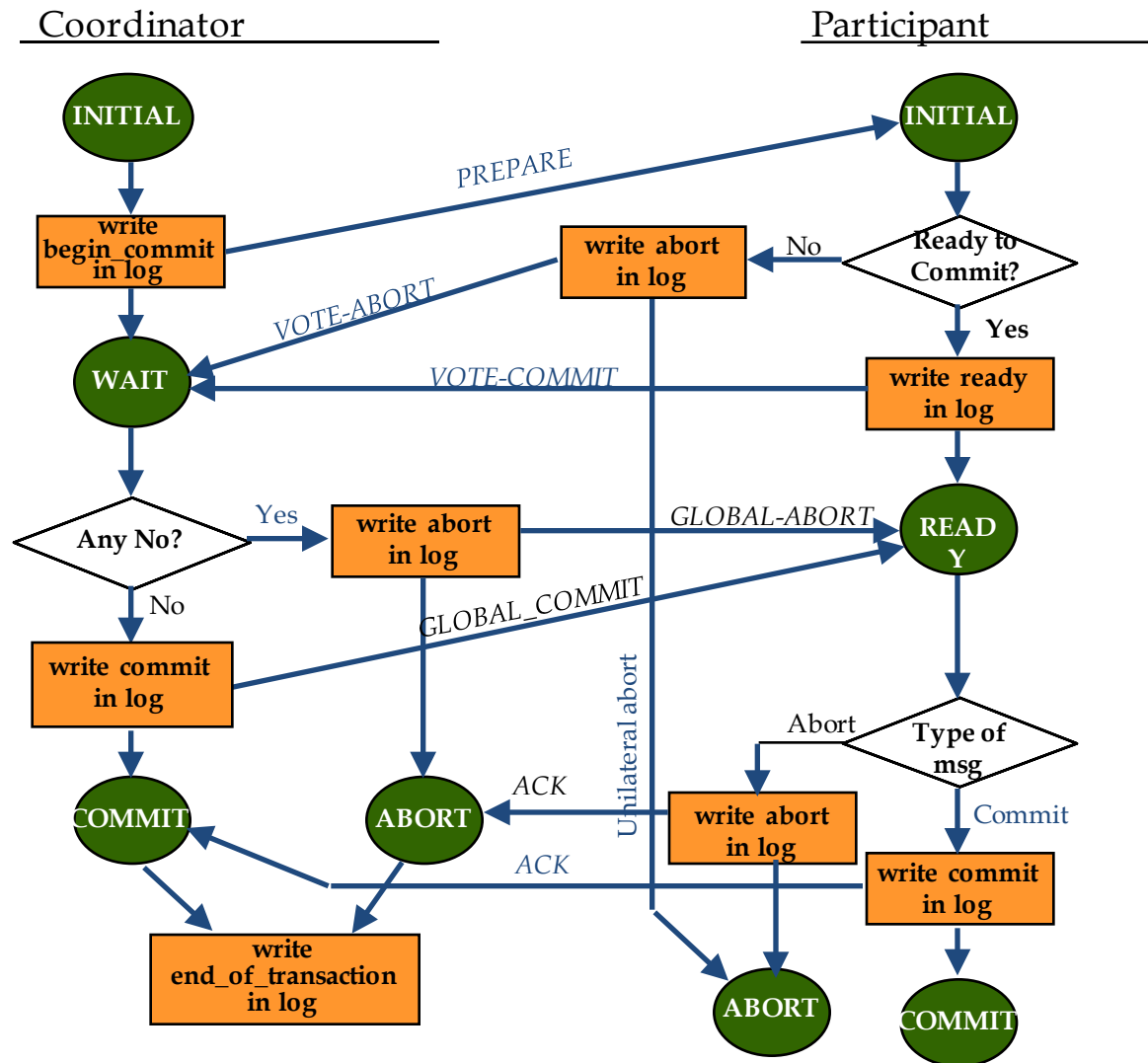
One-Phase Commit

- The coordinator tells all other processes (participants) whether or not to (locally) perform the operation in question
- Disadvantages:
 - If one of the participants cannot actually perform the operation, the coordinator will not know
- Solution: Two-Phase commit
 - All sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent

Two-Phase Commit (2PC)

- **Phase 1:** The coordinator gets the participants ready to commit the transaction at their local sites
- **Phase 2:** Everybody commits
 - 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:**
 - The coordinator aborts a transaction if and only if at least one participant votes to abort it
 - The coordinator commits a transaction if and only if all of the participants vote to commit it

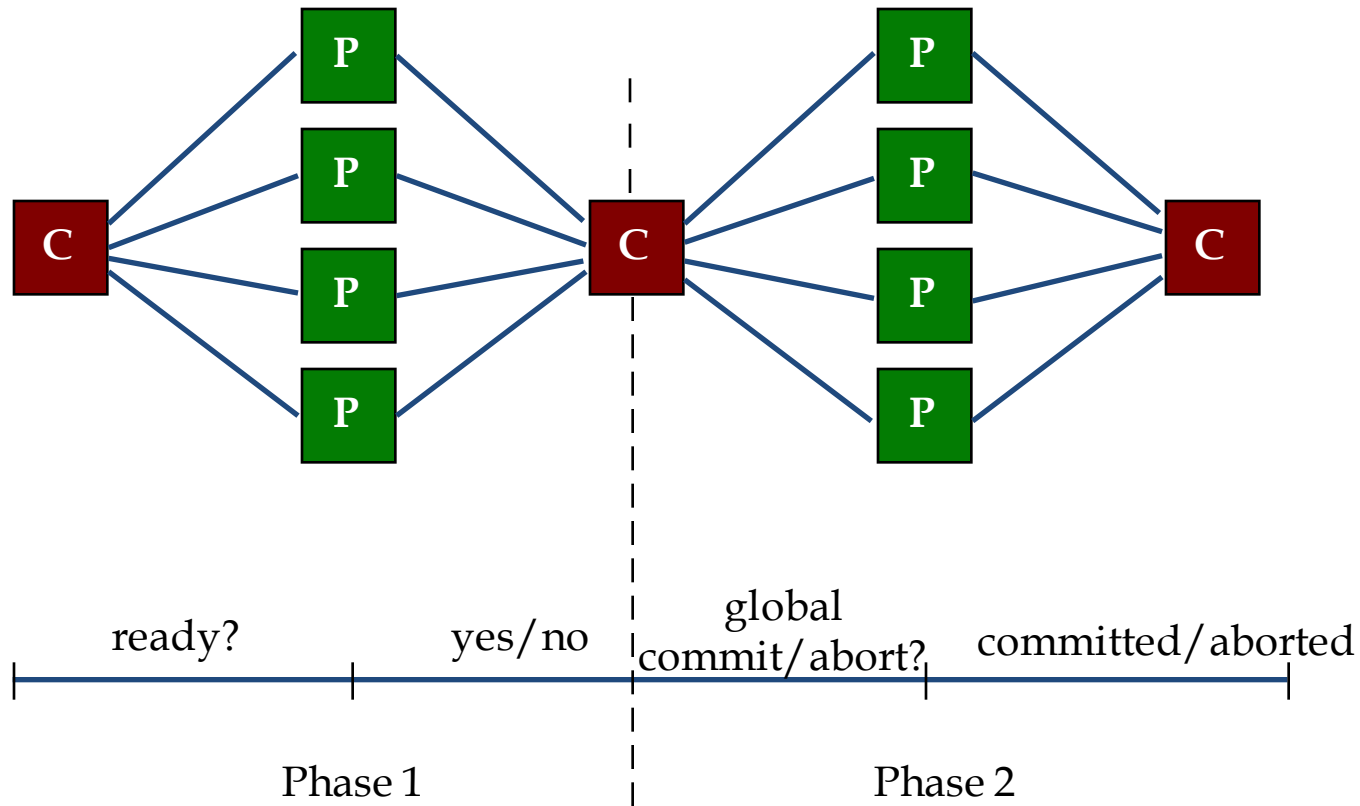
2PC Protocol Actions



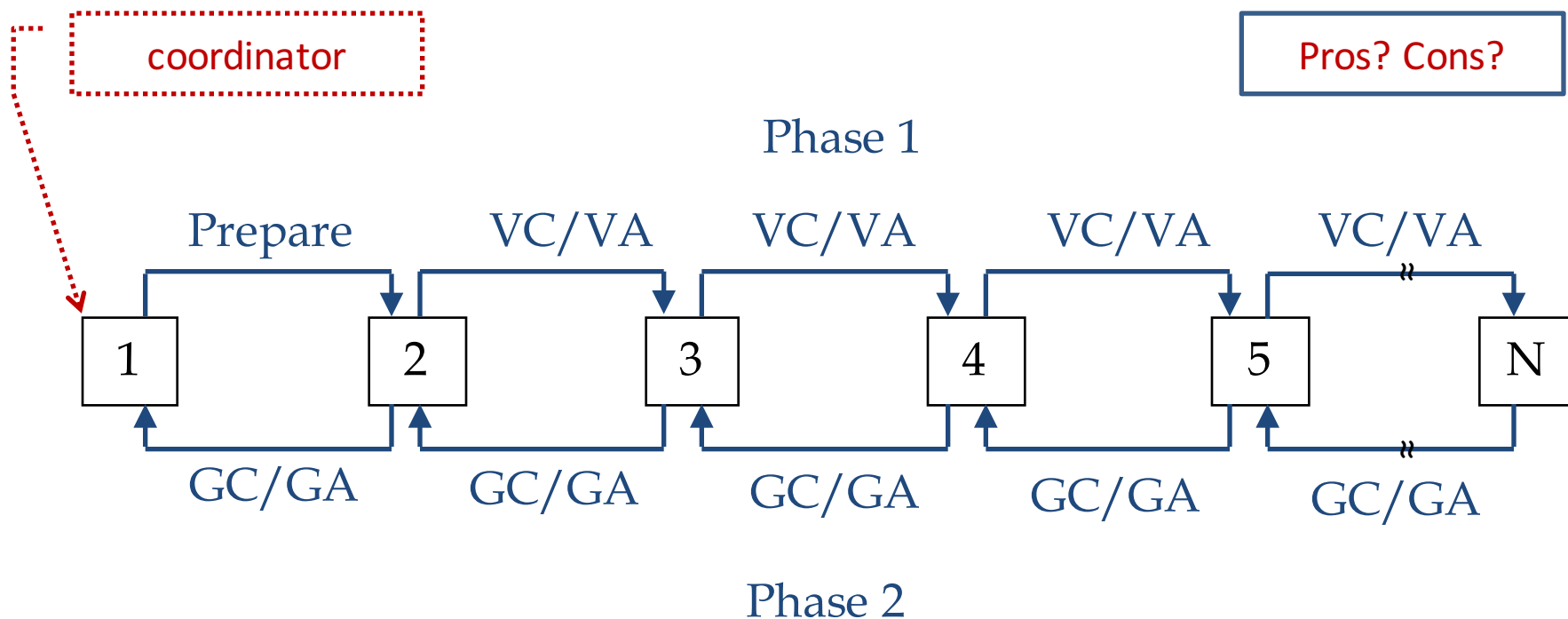
2PC: Observations

- A participant can unilaterally abort a transaction
- Once a participant votes to commit or abort a transaction, it cannot change its vote
- While a participant is in the READY state, it can move either to abort the transaction or to commit it, depending on the nature of the message from the coordinator
- The global termination decision is taken by the coordinator according to the global commit rule
- Coordinator and participants waiting in a state can timeout and invoke a timeout protocol

Centralized 2PC

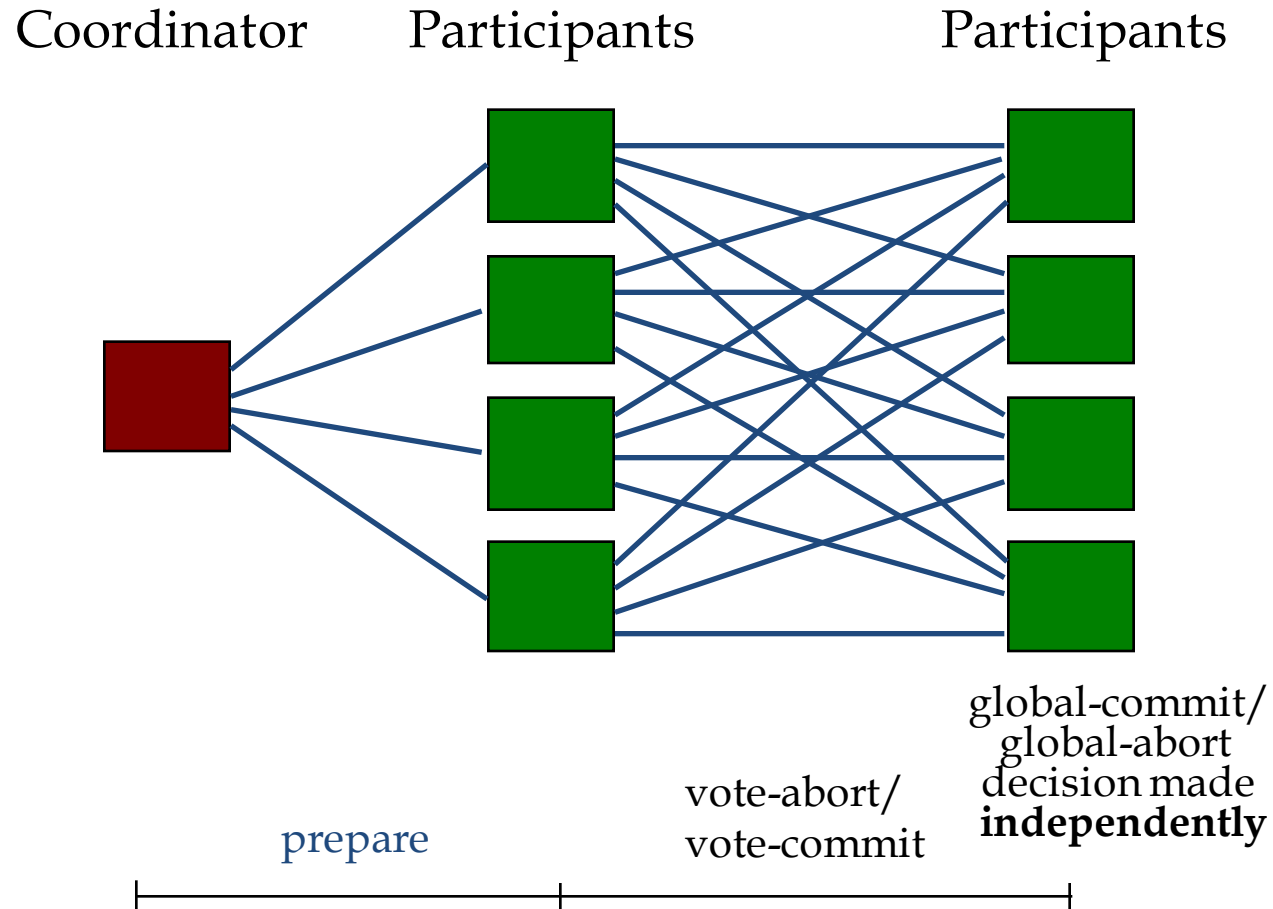


Linear (Nested) 2PC

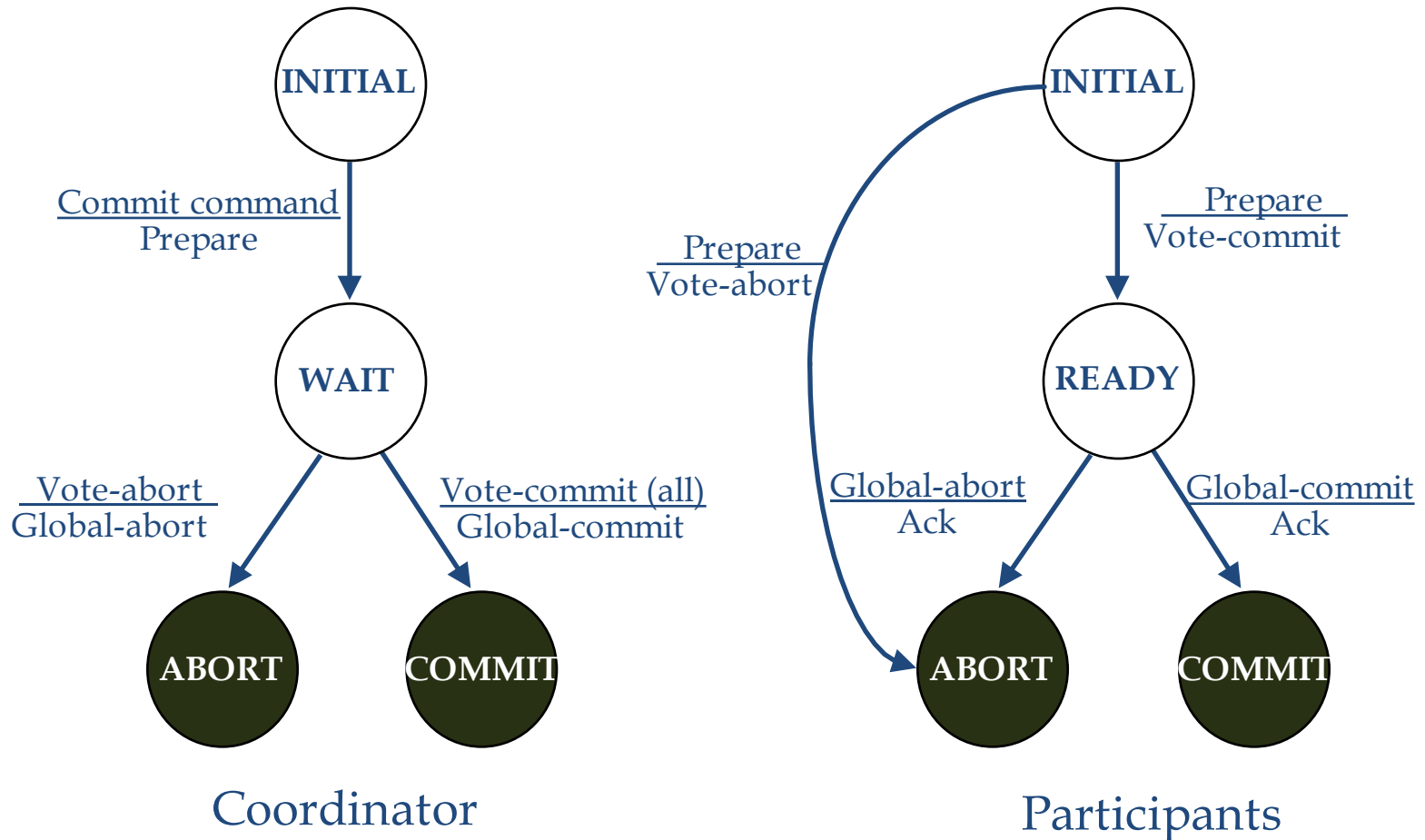


VC: Vote-Commit, VA: Vote-Abort, GC: Global-commit, GA: Global-abort

Distributed 2PC



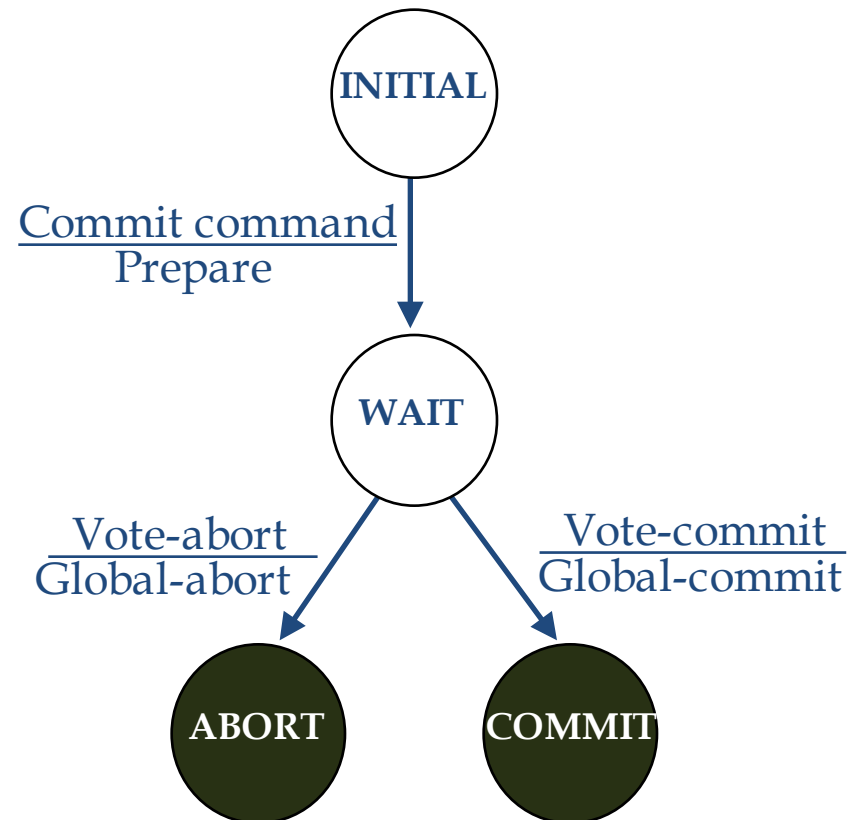
State Transition in 2PC



Site Failure - 2PC Termination Coordinator

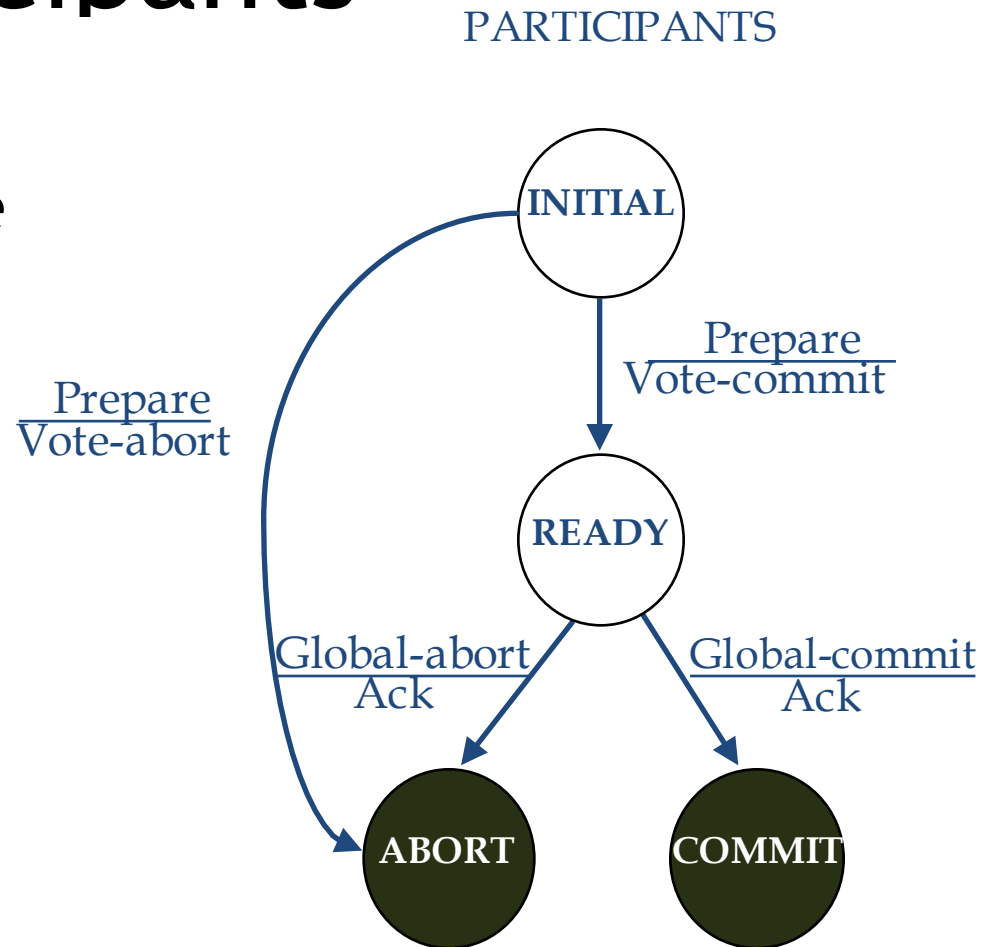
COORDINATOR

- 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
 - May resend Global Abort/Commit commands



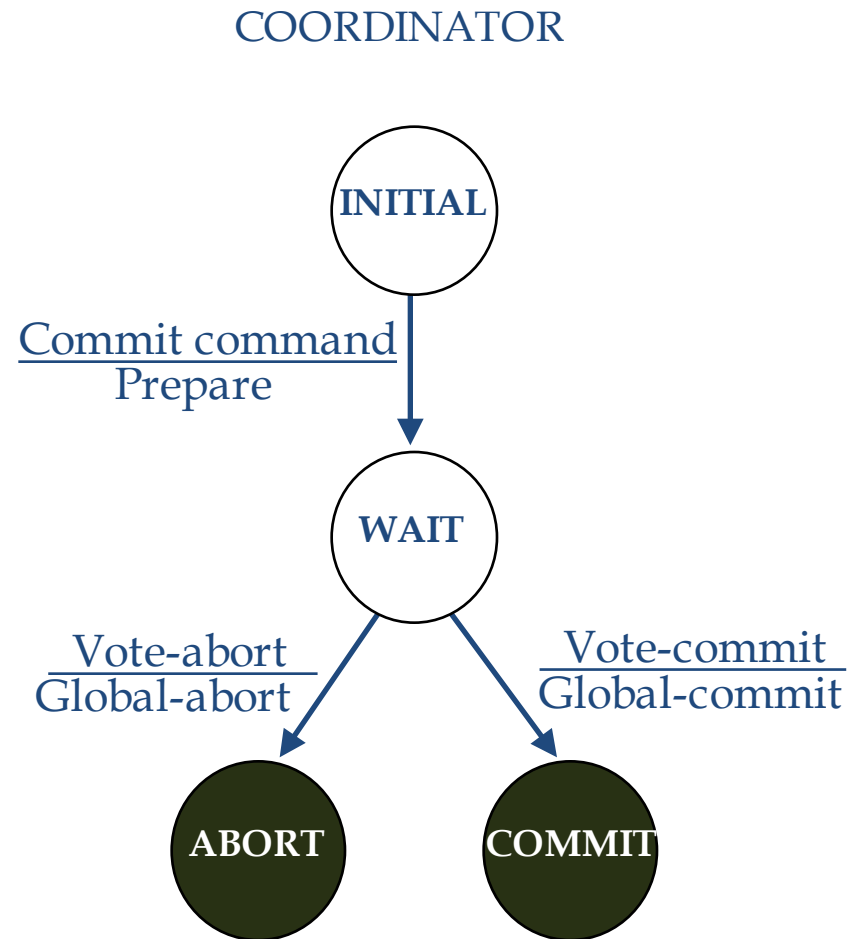
Site Failure - 2PC Termination Participants

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



Site Failure - 2PC Recovery Coordinator

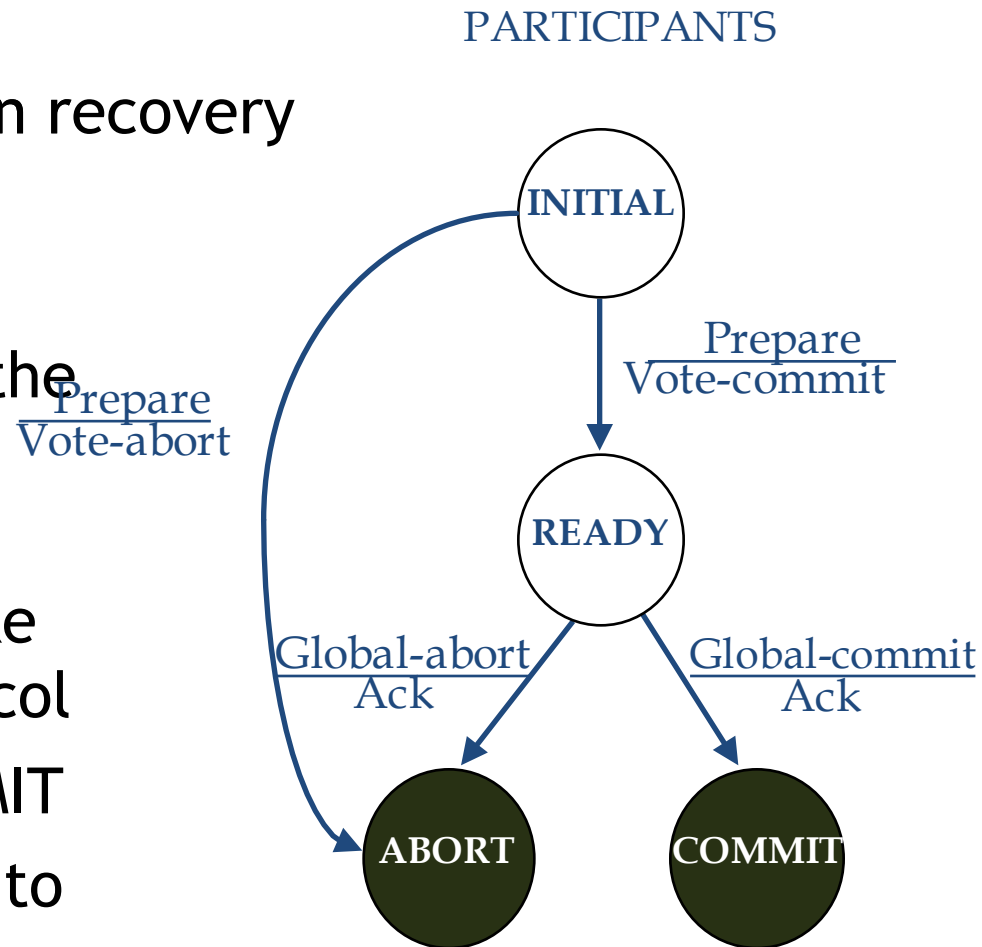
- Failure in INITIAL
 - Start the commit process upon recovery
- Failure in WAIT
 - Restart the commit process upon recovery
- Failure in ABORT or COMMIT
 - Nothing special if all the acks have been received
 - Otherwise the termination protocol is involved



Site Failure - 2PC Recovery

Participants

- 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 Cases

- 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

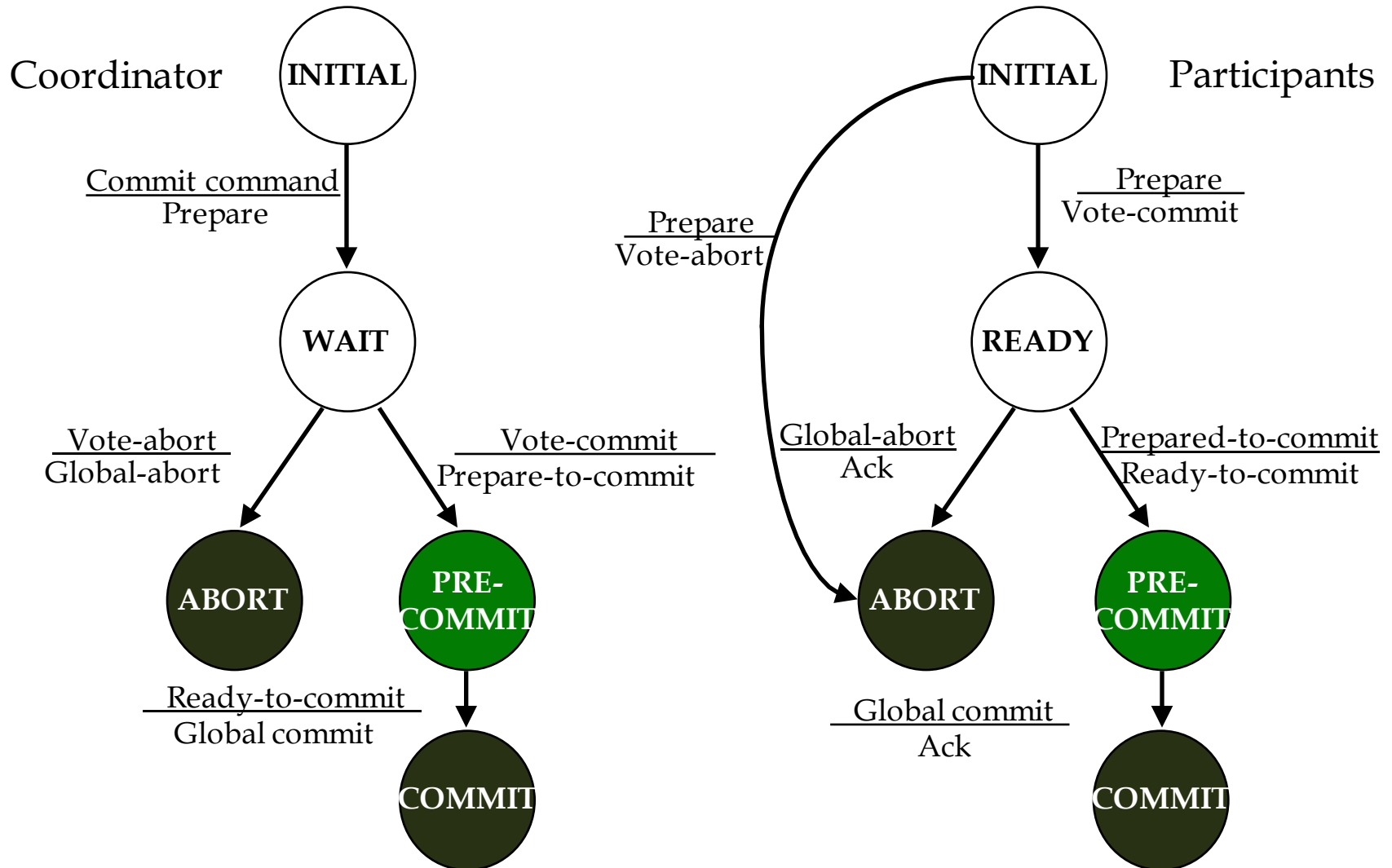
Problems with 2PC

- 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

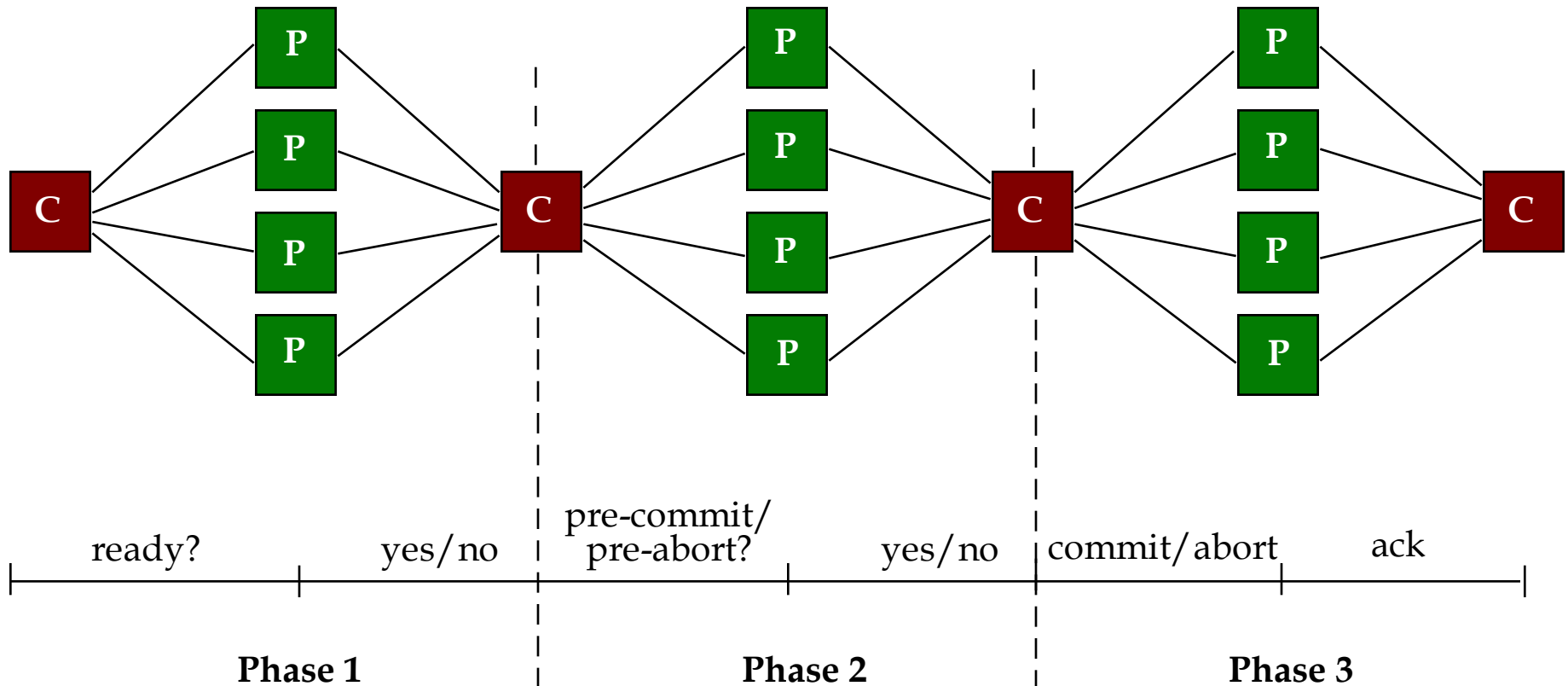
Three-Phase Commit

- 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 state to another with a single state transition
- Committable: all sites have voted to commit a transaction (e.g. COMMIT state)

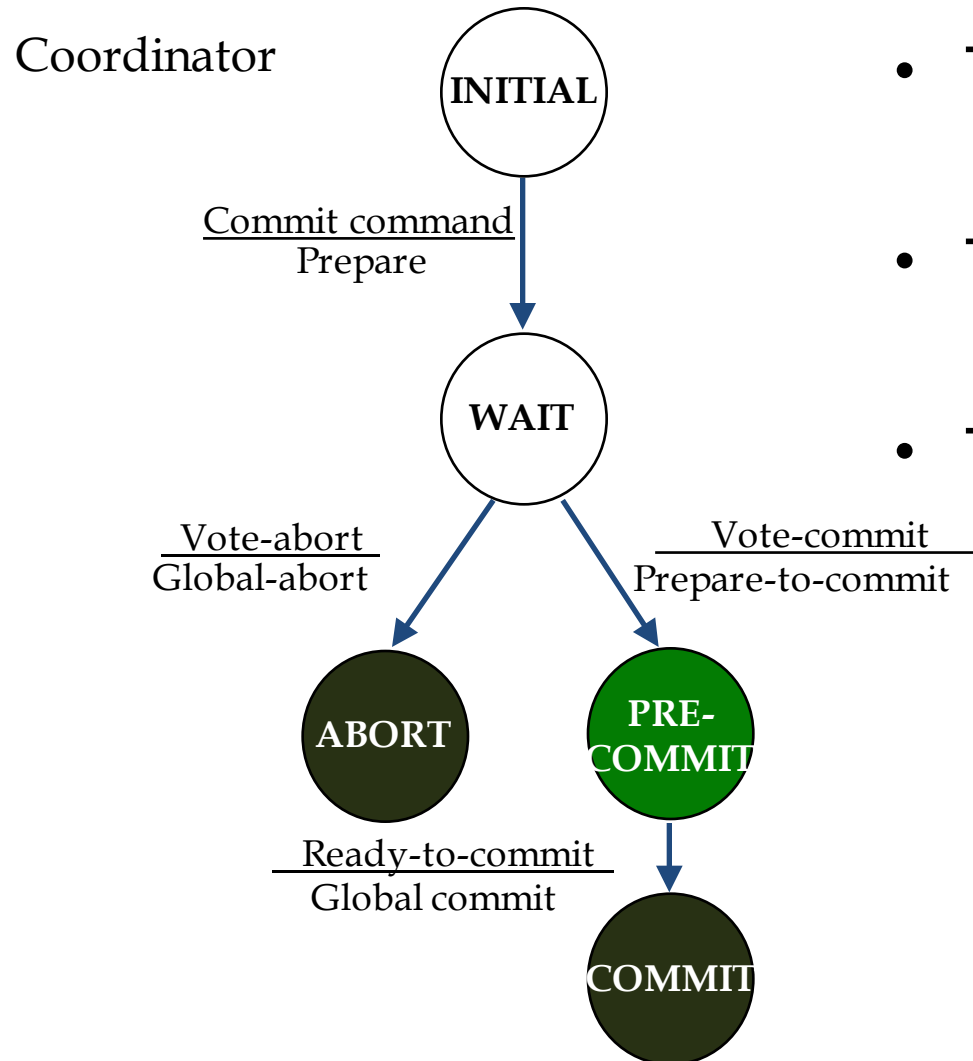
State Transitions in 3PC



Communication Structure

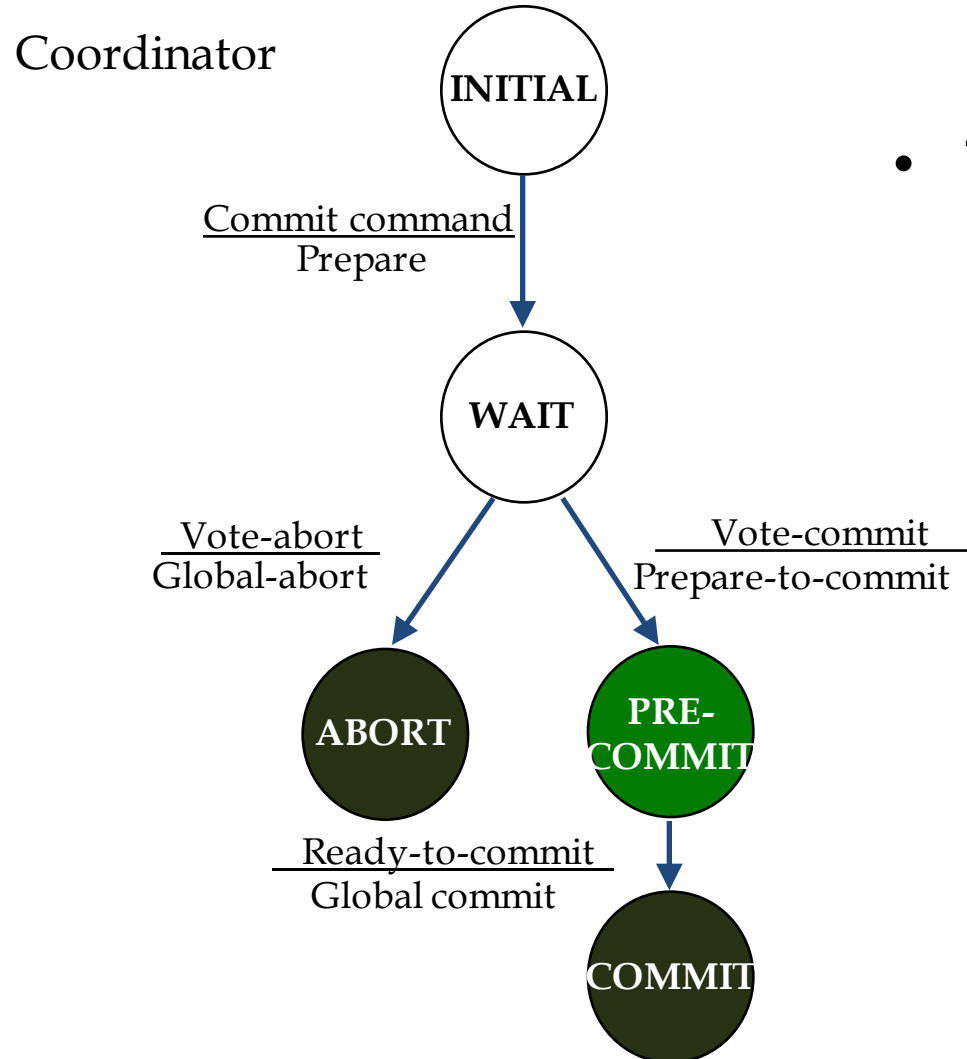


Site Failures - 3PC Termination



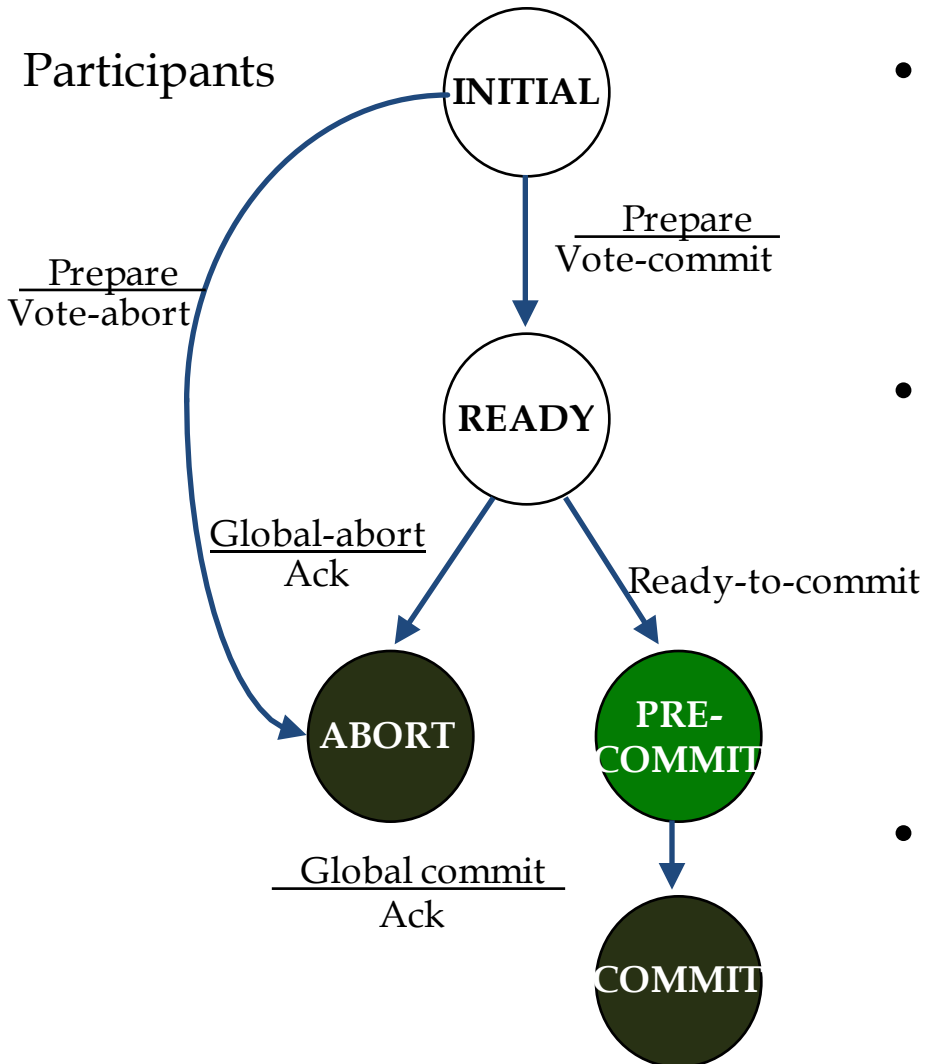
- Timeout in INITIAL
 - Who cares
- Timeout in WAIT
 - Unilaterally abort
- Timeout in PRECOMMIT
 - Participants may not be in PRECOMMIT, but at least in READY
 - Move all the participants to PRECOMMIT state
 - Terminate by globally committing

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

Site Failures - 3PC Termination



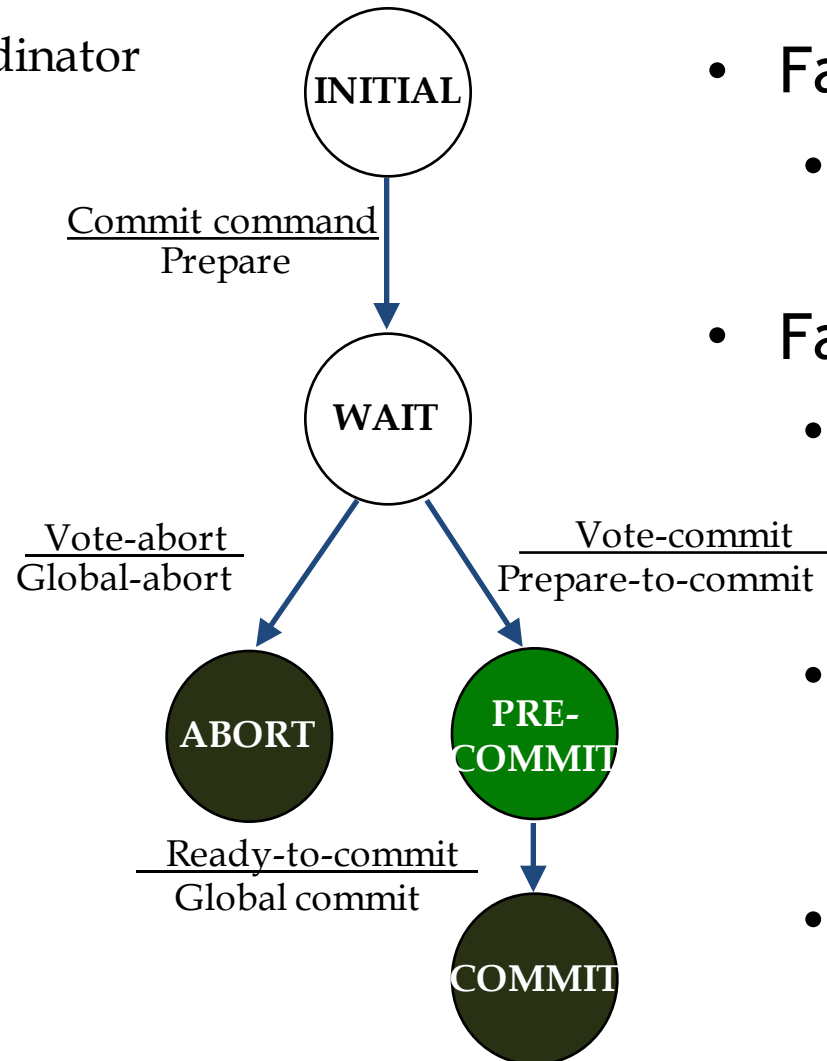
- Timeout in INITIAL
 - Coordinator must have failed in INITIAL state
 - Unilaterally abort
- Timeout in READY
 - Voted to commit, but does not know the coordinator's decision
 - Elect a new coordinator and terminate using a special protocol
- Timeout in PRECOMMIT
 - Handle it the same as timeout in READY state

Termination Protocol Upon Coordinator Election

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

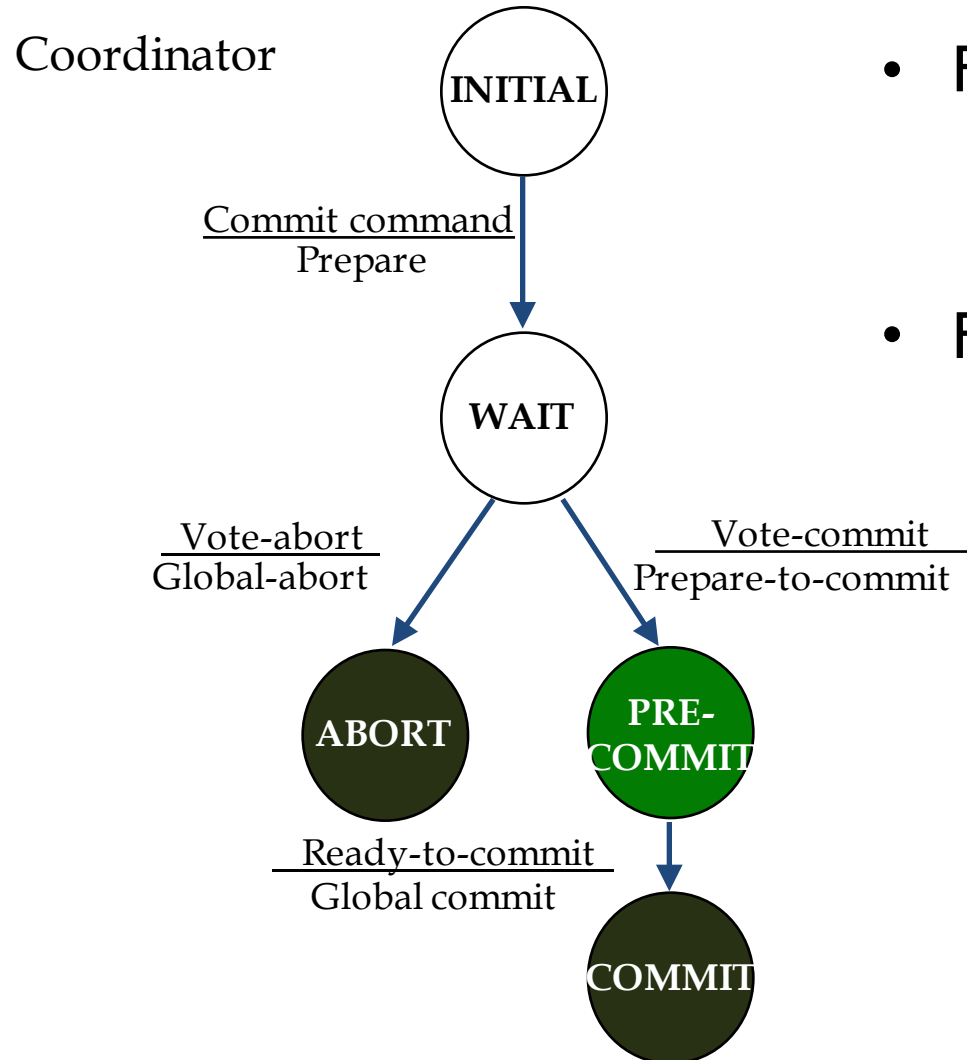
1. Coordinator sends its state to all of the participants asking them to assume its state
2. Participants “back-up” and reply with appropriate messages, except those in ABORT and COMMIT states. Those in these states respond with “Ack” but stay in their states
3. Coordinator guides the participants towards 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



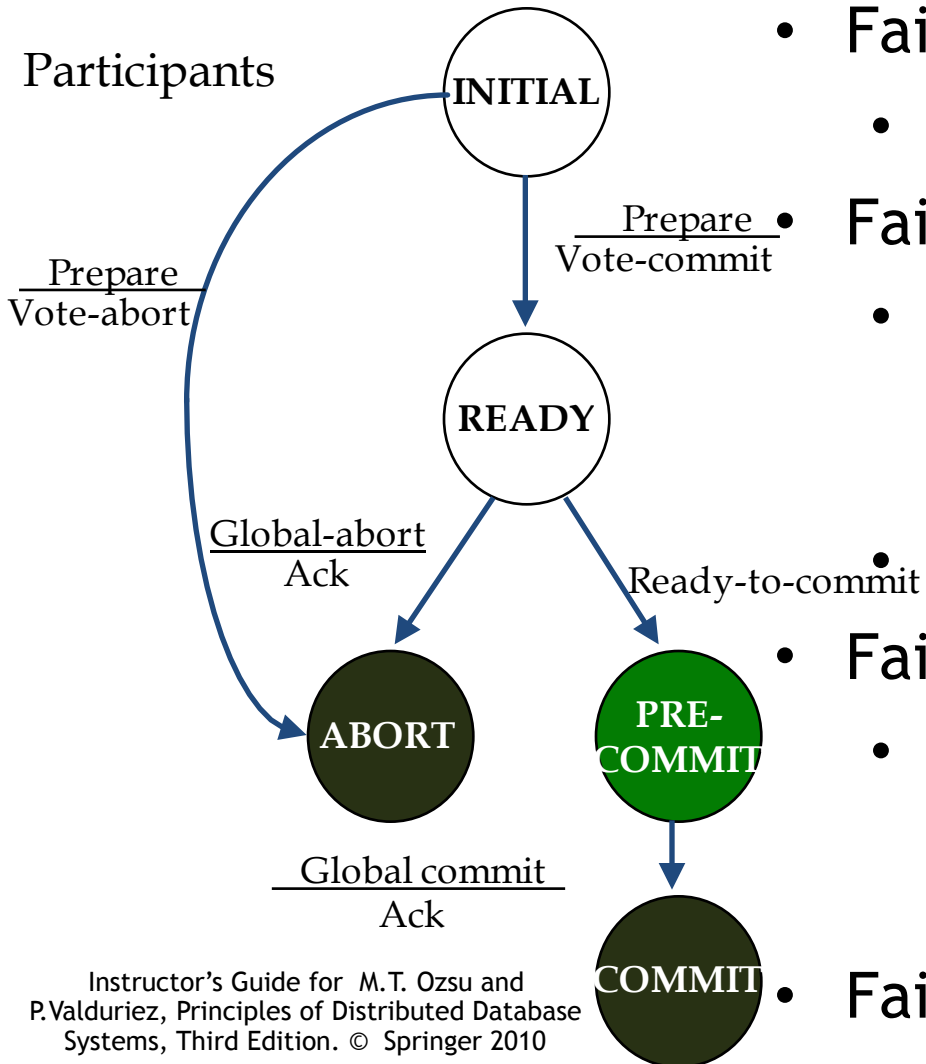
- Failure in INITIAL
 - start commit process upon recovery
- Failure in WAIT
 - the participants may have elected a new coordinator and terminated the transaction
 - the new coordinator could be in WAIT or ABORT states ==> transaction aborted
 - ask around for the fate of the transaction

Site Failures - 3PC Recovery



- Failure in PRECOMMIT
 - ask around for the fate of the transaction
- Failure in COMMIT or ABORT
 - Nothing special if all the acknowledgements have been received
 - otherwise, the termination protocol is involved

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

Thank You