Unit IV: Fault Tolerance, Recovery and Distributed Transaction

References:

1.G. Coulouris, J. Dollimore and T. Kindberg; **Distributed Systems Concepts and Design,4th Edition.**

2.Andrew S. Tanenbaum and Maarten van Steen**; Distributed Systems: Principles and Paradigms, 2nd Edition.**

Fault Tolerance, Recovery and Distributed Transaction

4.1 Fault Models

4.1.1 Crash, Omission and Byzantine failures.

4.1.2 Fault Tolerance Techniques:

4.1.2.1 Replication and Checkpointings

4.1.3 Recovery Mechanisms:

4.1.3.1 Rollback, checkpointing and recovery protocols

4.2 Agreement in faulty system

4.2.1 Distributed consensus

4.2.2 Byzantine Generals Problem

4.3 Distributed Transaction

4.3.1 Concurrency control mechanism

4.3.2 Automic Commitment Protocol

4.4 Distributed Deadlock

4.4.1 Overview of distributed deadlock (Resource and communication deadlock)

4.4.2 Deadlock prevention

4.4.2.1 Timestamp Ordering

4.4.2.2 Prevention Through Resource Allocation

4.4.3 Deadlock detection

4.4.3.1 Centralized Deadlock Detection

4.4.3.2 Distributed Deadlock Detection

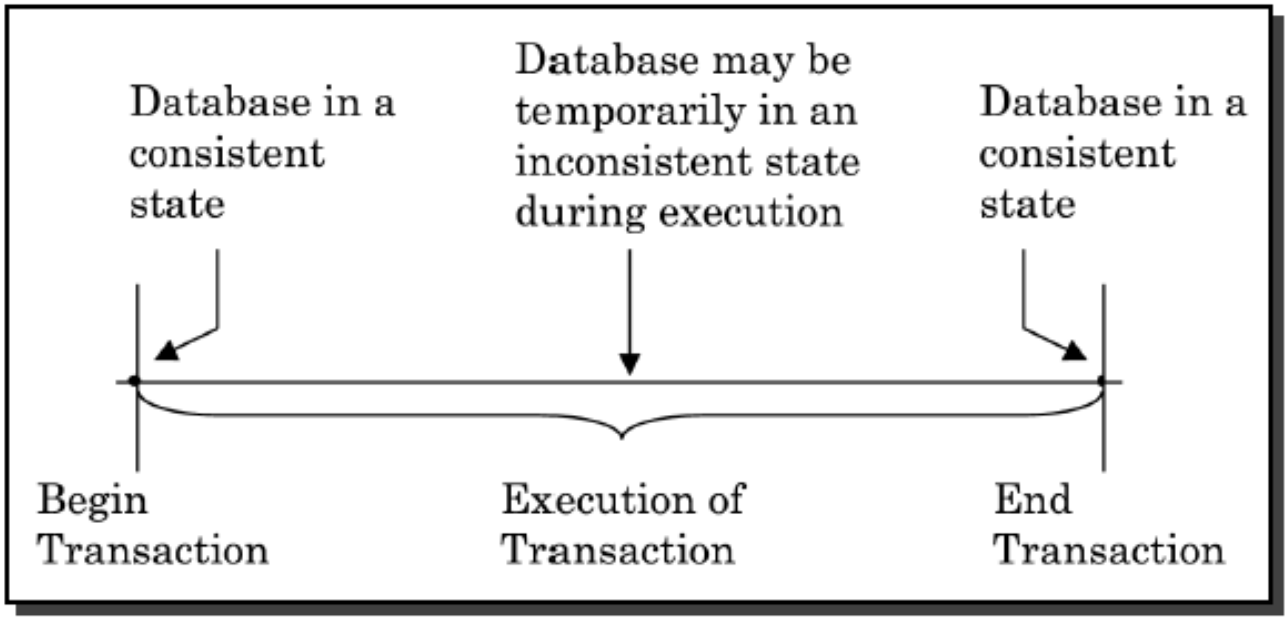
4.4.4 *Resolving deadlock*

TRANSACTION & NESTED TRANSACTION

- Transaction: specified by a client as a set of operations on objects to be performed as an indivisible unit by the servers managed those objects.
- Goal of transaction: ensure all the objects managed by a server remain in a consistent state when accessed by multiple transactions and in the presence of server crashes.
- **States** of a transaction
  - **Active**: Initial state and during the execution
  - **Paritally committed**: After the final statement has been executed
  - **Committed:** After successful completion
  - **Failed:** After the discovery that normal execution can no longer proceed
  - **Aborted:** After the transaction has been rolled back and the DB restored to its state prior to the start of the transaction. Restart it again or kill it.

TRANSACTION & NESTED TRANSACTION

- **Transaction:** A collection of actions that transforms the DB from one consistent state into another consistent state; during the exectuion the DB might be inconsistent.



TRANSACTION & NESTED TRANSACTION

- Transaction: specified by a client as a set of operations on objects to be performed as an indivisible unit by the servers managed those objects.
- Goal of transaction: ensure all the objects managed by a server remain in a consistent state when accessed by multiple transactions and in the presence of server crashes.
- Transaction applies to recoverable objects and intended to be atomic (atomic transaction):
  - Recoverable objects: objects can be recovered after their server crashes.
  - Atomic operations: operations that are free from interference from concurrent operations being performed in the other threads.
- Transaction properties (ACID):
  - **A**tomicity- Transaction must be all or none
  - **C**onsistency- Transaction of the system takes one consistent to another consistent state.
  - **I**solation -one process could not hamper other
  - **D**urability – storing the transaction's log report in non- volatile storage

TRANSACTION & NESTED TRANSACTION

- Maximize concurrency: transactions are allowed to execute concurrently if they would have the same effect as a serial execution → serially equivalent.
- Cases of transaction failing:
  - Service actions related to process crashes.
  - Client actions related to server process crashes.

- **Concurrency control:**
  - Lost update problem
  - Inconsistent retrievals problem.
  - Serial equivalence
  - Conflicting operations

Lost update problem

Occur when two transactions reads the old value of variable and then use it to calculate the new value.

TRANSACTION T		TRANSACTION U	

balance = b.getBalance( );
b.setBalance(balance*1.1);
a.withdraw(balance/10);
		balance = b.getBalance( );
		b.setBalance(balance*1.1);
		c.withdraw(balance/10);	

balance = b.getBalance( );	$200		
		balance = b.getBalance( );	$200
b.setBalance(balance*1.1);	$220	b.setBalance(balance*1.1);	$220
a.withdraw(balance/10);	$80		
		c.withdraw(balance/10) ;	$280

Initially A,B,C has $100,$200,$300 respectively. Transaction T transfer the amount from A to B and Transaction U transfer an amount from account C to an account B. In both cases, the amount transfer is calculated to increase the balance by 10%. The net effects on account B of executing the transactions T and U should be to increase the balance of account B by 10% twice, so its final value is $242.

Inconsistent retrievals problem

This problem occur when retrieval transaction run concurrently with an update transaction. can not occur if retrieval transaction is performed before or after the update transaction.

TRANSACTION V	TRANSACTION W
a.withdraw(100); b.deposit(100);	aBranch.branchTotal( );
a.withdraw(100); $100 b.deposit(100); $300	 total = a.getBalance( ); $100 total = total + b.getBalance( ); $300 total = total + c.getBalance( ); . .

A and b both initially $200. The result of branchTotal includes the sum of A and B as $300, which is wrong(It should be $400).

A serially equivalent interleaving of T and U

- Removes the lost and updates and inconsistent retrieval problem
- An interleaving of the transaction had been performed one at a time in some order is a serially equivalent interleaving.

TRANSACTION T		TRANSACTION U	

balance = b.getBalance( );
b.setBalance(balance*1.1);
a.withdraw(balance/10);
		balance = b.getBalance( );
		b.setBalance(balance*1.1);
		c.withdraw(balance/10);	

balance = b.getBalance( );
b.setBalance(balance*1.1);
 	
a.withdraw(balance/10);
	$200
	$220
	
	$80
		
		balance = b.getBalance( );
		b.setBalance(balance*1.1);
		
		c.withdraw(balance/10) ;
			
			$220
			$242
			
			$278

A serially equivalent interleaving of V and W

TRANSACTION V	TRANSACTION W
a.withdraw(100); b.deposit(100);	aBranch.branchTotal( );
a.withdraw(100); $100 b.deposit(100); $300	total = a.getBalance( ); $100 total = total + b.getBalance( ); $400 total = total + c.getBalance( ); . .

Read and write operation conflict rules

Operations of different transactions		Conflict
Read	Read	NO
Read	Write	YES
Write	Write	YES

- For two transaction to be serially equivalent, it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access.

TRANSACTION & NESTED TRANSACTION

- Recoverability from aborts
  - Problems with aborting:
    - Dirty read
    - Premature writes
  - Solutions:
    - Recoverability of transactions
    - Avoid cascading aborts
    - Strict execution of transactions
    - Tentative versions

A dirty read when transaction T aborts

TRANSACTION T		TRANSACTION U	
a.getBalance( ); a.setBalance(balance+10);		a.getBalance( ); a.setBalance(balance+20);	

balance = a.getBalance( );
a.setBalance(balance+10);
 	
abort transaction;
	$100
	$110
		
		balance = a.getBalance( );
		a.setBalance(balance+20);
		commit transaction;
			
			$110
			$130

- **Recoverability of transactions**: delay commits until after the commitment of any other transaction whose uncommitted state has been observed.
- **Cascading aborts:** the aborting of any transactions may cause further transactions to be aborted → transactions are only allowed to read objects that were written by committed transactions.

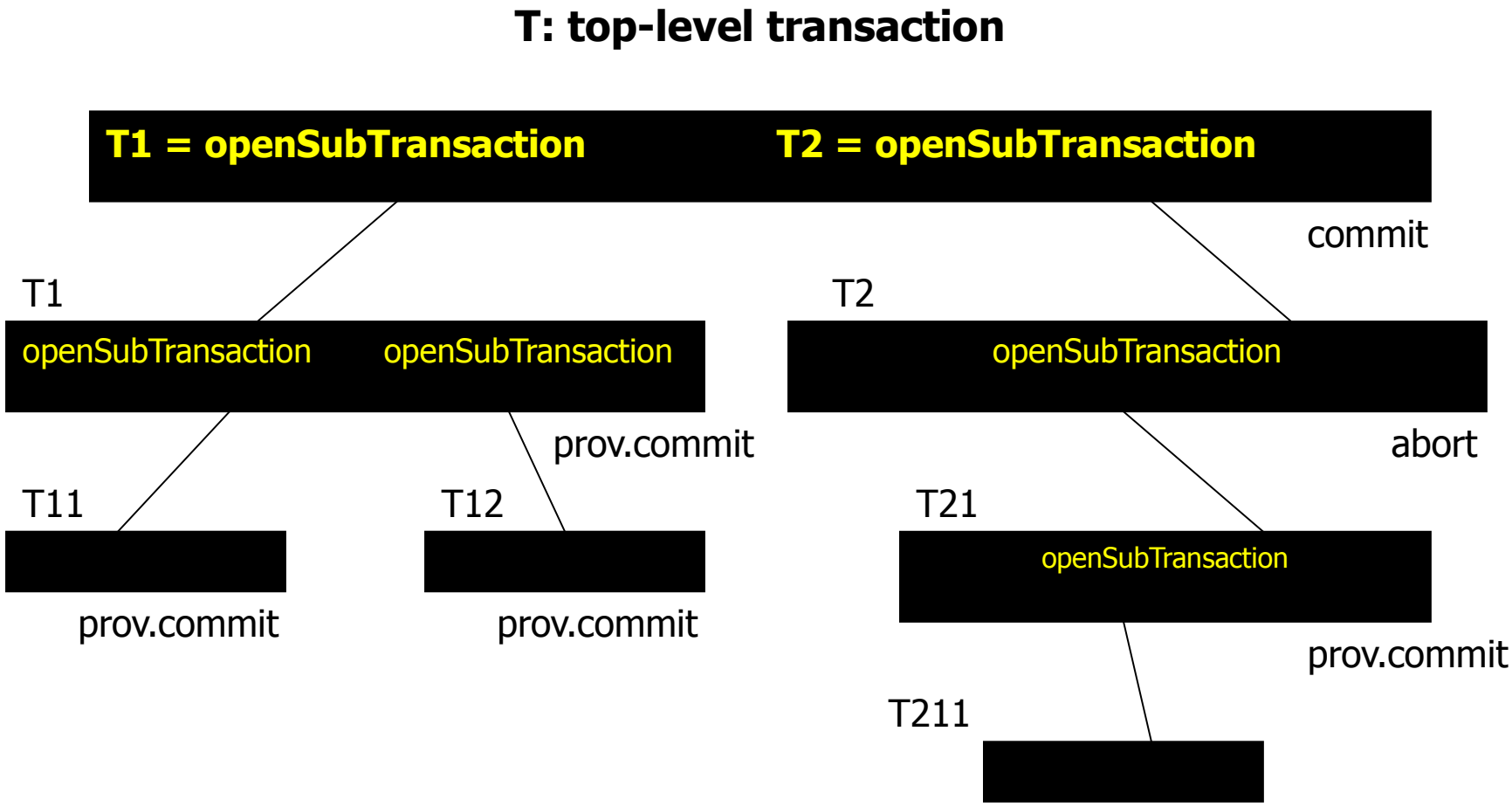
Overwriting uncommitted values

TRANSACTION T		TRANSACTION U	
a.setBalance(105);		a.setBalance(110);	
a.setBalance(105);	$100 $105	a.setBalance(110);	$110

- **Strict execution of transaction**: service delay both read and write operations on an object until all transactions that previously wrote that object have either committed or aborted.

- **Tentative versions**: update operations performed during a transaction are done in tentative versions of objects in volatile memory.

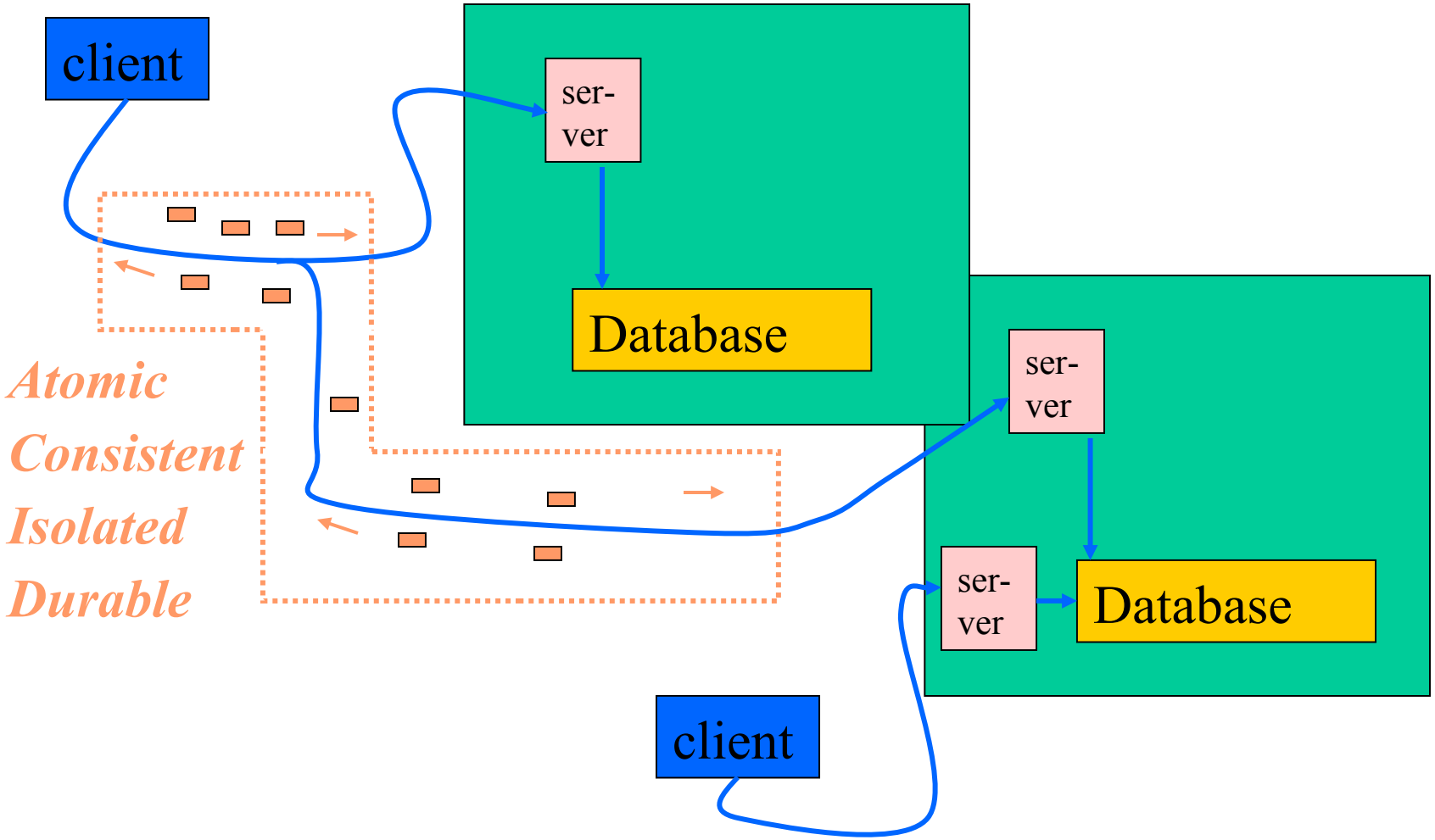
Nested transaction



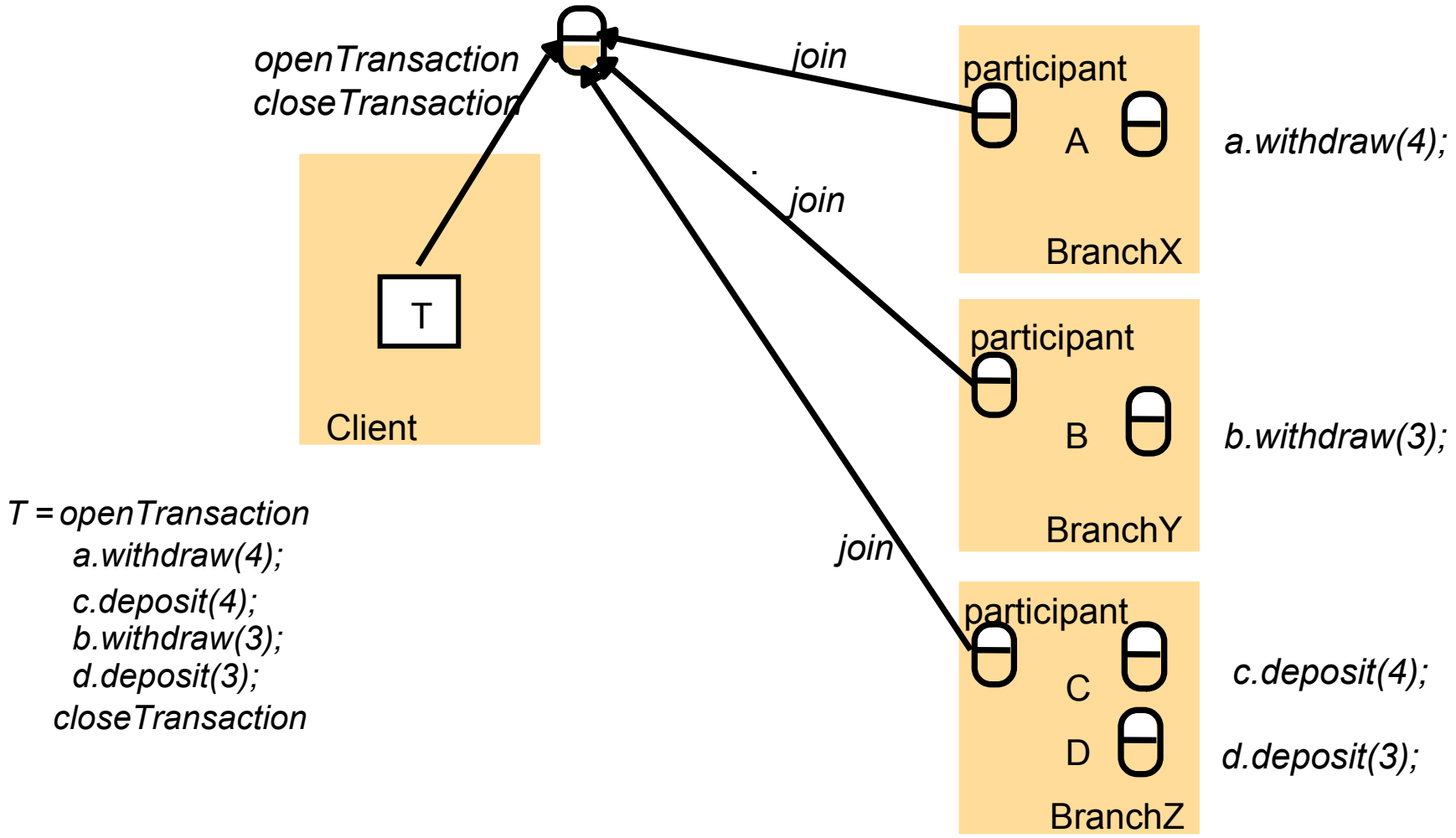
TRANSACTION & NESTED TRANSACTION

- Advantages of nested transaction:
  - Additional concurrency in a transaction
  - Subtransactions can commit or abort independently
- Rules for commitment of nested transactions:
  - Transactions commit/abort only after its child have completed.
  - After completing, subtransaction makes independent decision either to commit provisionally or to abort.
  - When a parent aborts, all of its subtransactions are aborted.
  - When a subtransaction aborts, parent can decide whether to abort or not.
  - Top-level transaction commits→all of the provisionally committed subtransactions can commit too (provided none of their ancestor has aborted).

Distributed Transactions



A Distributed Banking Transaction



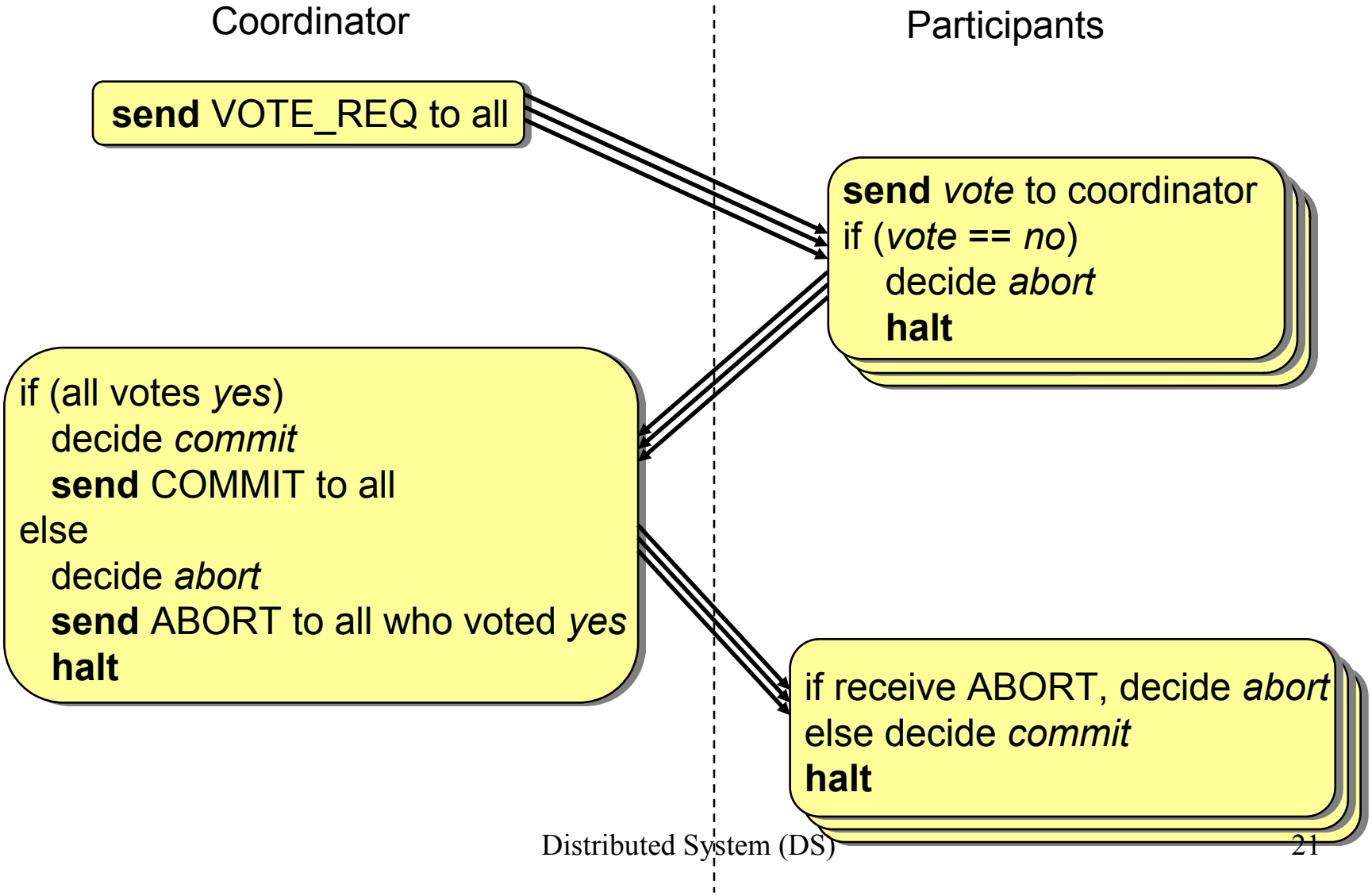
Distributed commit

- Is done by using atomic commitment protocols (ACP)
  1. One-phase Commit
  2. Two-phase Commit
  3. Three-phaseCommit

One-phase Commit

- One-phase commit protocol
  - This scheme however is not fault tolerant
  - One site is designated as a coordinator
  - The coordinator tells all the other processes whether or not to locally perform the operation in question
  - But, what if a process cannot perform the operation? There's no way to tell the coordinator!
  - **The solutions**:
  - The *Two-Phase* and *Three-Phase Commit Protocols*.

Two Phase Commit (2PC)



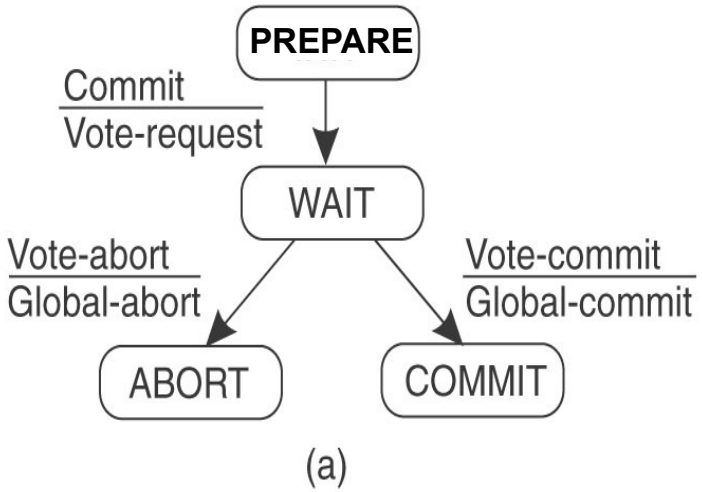
Two-Phase Commit

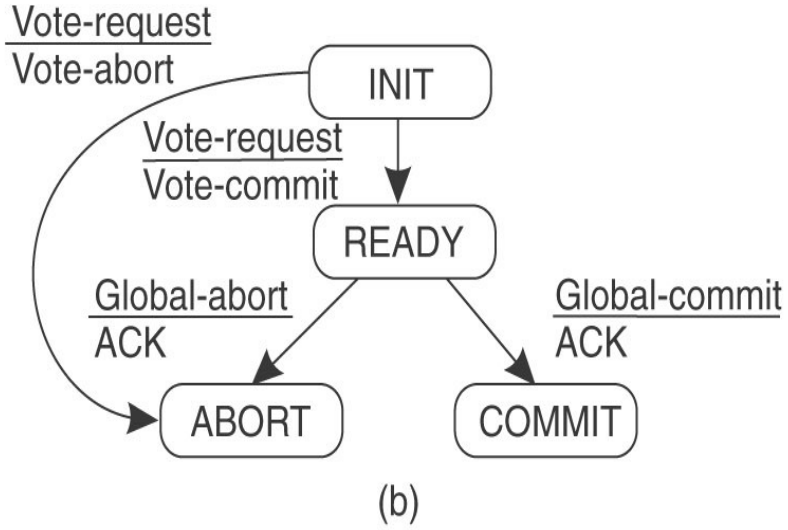
- One site is elected *coordinator* of the transaction *T* (See *Election* algorithms)
- *Phase 1:* When coordinator is ready to commit the transaction
    - Place *Prepare*(*T*) state in log on stable storage
    - Send *Vote_request*(*T*) message to all other participants
    - Wait for replies
- *Phase 2:* Coordinator
  - If any participant replies *Abort*(*T*)
    - Place *Abort*(*T*) state in log on stable storage
    - Send *Global_Abort*(*T*) message to all participants
    - Locally abort transaction *T*
  - If *all* participants reply *Ready_to_commit*(*T*)
    - Place *Commit*(*T*) state in log on stable storage
    - Send *Global_Commit*(*T*) message to all participants
    - Proceed to commit transaction locally

Two-Phase Commit (continued)

- Phase I: Participant gets *Vote_request*(*T*) from coordinator
  - Place *Abort*(*T*) or *Ready*(T) state in local log
  - Reply with *Abort*(*T*) or *Ready_to_commit*(*T*) message to coordinator
  - If *Abort*(*T*) state, locally abort transaction
- Phase II: Participant
  - Wait for *Global_Abort*(*T*) or *Global_Commit*(*T*) message from coordinator
  - Place *Abort*(*T*) or *Commit*(T) state in local log
  - Abort or commit locally per message

Two-Phase Commit States





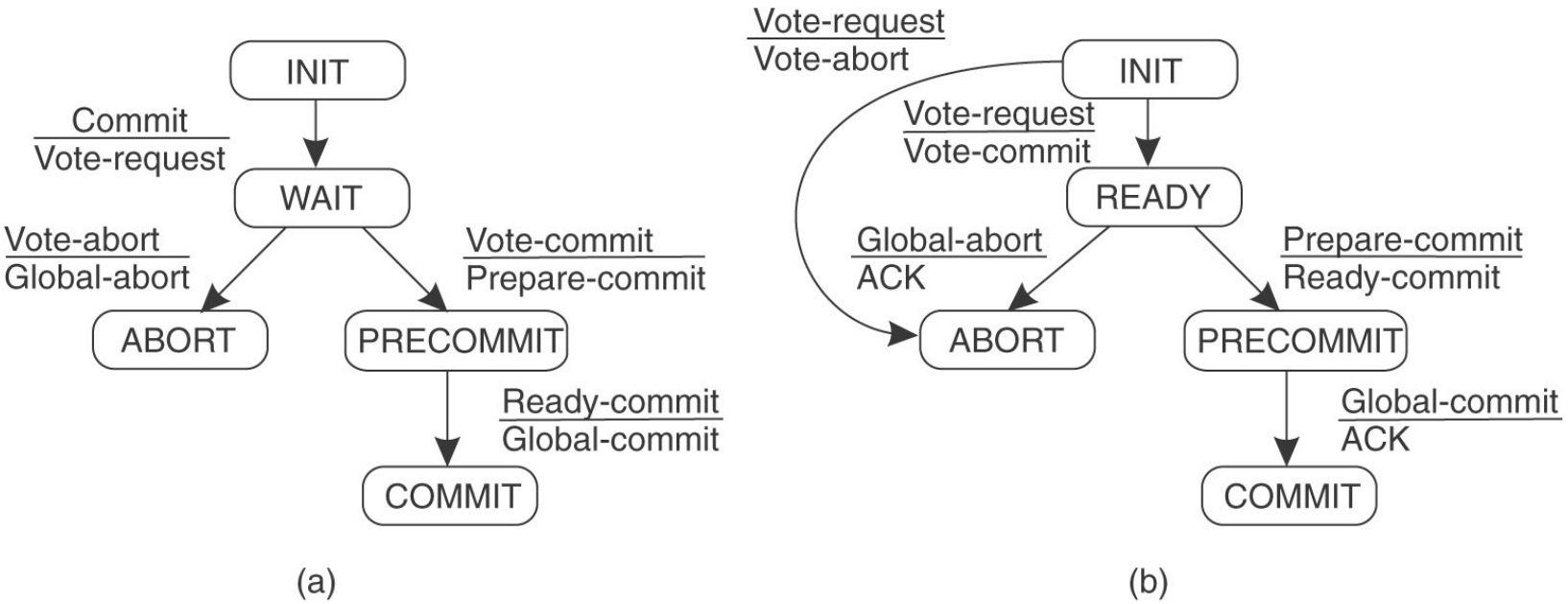
coordinator

participant

Big Problem with Two-Phase Commit

- It can lead to both the coordinator and the group members **blocking**, which may lead to the dreaded *deadlock*.
- If the coordinator crashes, the group members may not be able to *reach a final decision*, and they may, therefore, block until the coordinator *recovers* ...
- Two-Phase Commit is known as a **blocking-commit protocol** for this reason.
- The solution? *The Three-Phase Commit Protocol.*

Three-Phase Commit (1)



- There is no state from which a transition can be made to either *Commit* or *Abort*
- There is no state where it is not possible to make a final decision and from which transition can be made to *Commit*

∀ ⇒ non-blocking commit protocol

Three-Phase Commit

- **Coordinator:**
  - **Phase 1:** The coordinator receives a transaction request. If there is a failure at this point, the coordinator aborts the transaction (i.e. upon recovery, it will consider the transaction aborted). Otherwise, the coordinator sends a **canCommit?** message to the participants and moves to the waiting state.
  - **Phase 2:** If there is a failure, timeout, or if the coordinator receives a **No** message in the waiting state, the coordinator aborts the transaction and sends an **abort** message to all participants. Otherwise the coordinator will receive **Yes** messages from all participants within the time window, so it sends **preCommit** messages to all participants and moves to the prepared state.
  - **Phase 3:** If the coordinator succeeds in the prepared state, it will move to the commit state. However if the coordinator times out while waiting for an acknowledgement from a participant, it will abort the transaction. In the case where all acknowledgements are received, the coordinator moves to the commit state as well.

Three-Phase Commit

- **Participants**
  - **Phase 1:** The participant receives a **canCommit?** message from the coordinator. If the participant agrees it sends a **Yes** message to the coordinator and moves to the prepared state. Otherwise it sends a **No** message and aborts. If there is a failure, it moves to the abort state.
  - **Phase 2:** In the prepared state, if the participant receives an **abort** message from the coordinator, fails, or times out waiting for a commit, it aborts. If the participant receives a **preCommit** message, it sends an ACK message back and awaits a final commit or abort.
  - **Phase 3:** If, after a participant member receives a **preCommit** message, the coordinator fails or times out, the participant member goes forward with the commit.

Three-Phase Commit (continued)

- Coordinator sends *Vote_Request* (as before)
- If all participants respond affirmatively,
  - Put *Precommit* state into log on stable storage
  - Send out *Prepare_to_Commit* message to all
- After all participants acknowledge,
  - Put *Commit* state in log
  - Send out *Global_Commit*

Three-Phase Commit …

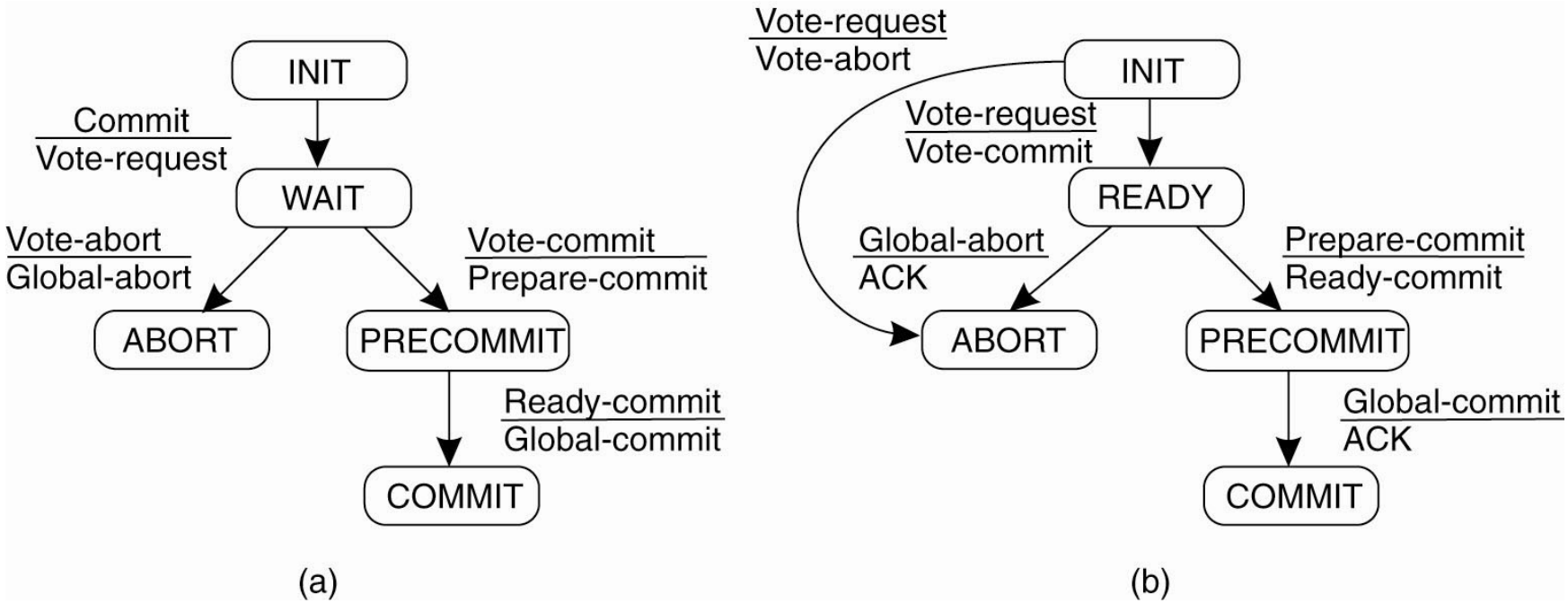


Figure (a) The finite state machine for the coordinator in 3PC. (b) The finite state machine for a participant.

METHOD FOR CONCURRENT CONTROL

- **Lock:**
  - Server attempts to lock any object that is about to use by client's transaction.
  - Requests to lock object's are suspended and wait until the objects are unlocked.
  - Serial equivalence: Serial equivalence requires that all of the transaction's accesses to a particular object be serialized with respect to accesses by other transactions. To achieve serial equivalence transaction is not allowed any new locks after it has release a lock.
    - Two-phase lock: growing phase (new locks are acquired), shrinking phase (locks are released).
  - Strict execution: locks are held until transaction commits/aborts (Strict two-phase locking).
  - Recoverability: locks must be held until all the objects it updated have been written to permanent storage.

METHOD FOR CONCURRENT CONTROL

- Lock:
  - Simple exclusive lock reduces concurrency→ locking schema for multiple transaction reading, single transaction writing: read locks (shared lock) & write locks(exclusive lock).
  - Operation conflict rules:
    - Request for a write lock is delayed by the presence of a read lock belonging to another transaction.
    - Request for either a read/write lock is delayed by the presence of a write lock belonging to another transaction.

Lock compatibility

For one object		Lock requested	
		Read	Write
Lock already set	none	OK	OK
	read	OK	wait
	write	Wait	wait

Transaction T and U with exclusive(write) locks.

Both Read and write operations are blocked under exclusive locks.

TRANSACTION T	TRANSACTION U

balance = b.getBalance( );
b.setBalance(balance*1.1);
a.withdraw(balance/10);
	balance = b.getBalance( );
	b.setBalance(balance*1.1);
	c.withdraw(balance/10);

openTransaction balance = b.getBalance( ); lock B b.setBalance(balance*1.1); a.withdraw(balance/10); lock A closeTransaction unlock A,B	openTransaction balance = b.getBalance( ); wait for T'lock on B … b.setBalance(balance*1.1); lock B c.withdraw(balance/10) ; lock C closeTransaction unlock B,C

Use of lock in two-phase locking:

1. When an operation accesses an objects within a transaction:

(a) if the object is not already lock, it is locked and the operation proceeds.

(b) if the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.

(c) if the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.

(d) if the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used).

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

METHOD FOR CONCURRENT CONTROL

- Locking rule for nested transactions:
  - Locks that are acquired by a successful subtransaction is inherited by its parent & ancestors when it completes. Locks held until top-level transaction commits/aborts.
  - Parent transactions are not allowed to run concurrently with their child transactions.
  - Subtransactions at the same level are allowed to run concurrently.

Deadlock with write lock.

Transaction T		Transaction U	
Operations	Lock	Operations	Lock
a.deposit(100)	write lock A		
		b.deposit(200)	write lock B
b.Withdraw(100)	wait for U's lock on B		
		a.Withdraw(200)	wait for T's lock on A
…			
…		…	
…		…	

METHOD FOR CONCURRENT CONTROL

- **Deadlock**:
  - Definition: A state in which each member of a group of transactions is waiting for some other member to release a lock.
  - **Prevention**:
    - Lock all the objects used by a transaction when it starts → not a good way.
    - Request locks on objects in a predefined order → premature locking & reduction in concurrency.
  - **Detection**: Finding cycle in a wait-for graph→ select a transaction for aborting to break the cycle.
    - Choice of transaction to be aborted is not simple.
  - **Timeouts**: each lock is given a limited period in which it is invulnerable.
    - Transaction is sometimes aborted but actually there is no deadlock.
    - Appropriate length of a timeout.

METHOD FOR CONCURRENT CONTROL

- Increasing concurrency in locking schema: 2 approaches
  - Two-version locking: the setting of exclusive locks is delayed until a transaction commits.
  - Hierarchic locks: mix-granularity locks are used.

Lock compatibility for two-version locking

For one object		Lock to be set		
		Read	write	commit
Lock already set	none	OK	OK	OK
	read	OK	OK	wait
	write	OK	wait	----
	commit	Wait	wait	----

Two-version locking: allows one transaction to write tentative versions of objects when other transactions read from the committed version of the same objects.

- read operations are delayed only while the transactions are being committed rather than during entire execution.
- read operations can cause delay in committing other transactions.

METHOD FOR CONCURRENT CONTROL

- Drawbacks of locking:
  - Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data.
  - Deadlock prevention by locks reduces the concurrency (lock transaction can not use by other until it unlock the transaction). Deadlock detection or timeout not wholly satisfactory for use in interactive programs.
  - To avoid cascading abort, locks can not be release until the end of transaction → reduce potential concurrency.

METHOD FOR CONCURRENT CONTROL

- **Optimistic concurrency control:**
  - Idea: in most applications, the likelihood of two clients transactions accessing the same object is low.
  - Transactions are allowed to proceed as though there were no possibility of conflict with other transactions until the client completes its task and issues a closeTransaction request.
  - **3 phases of a transaction:**
    - **Working phase:** each transaction has a tentative version of each of the objects that it updates.
    - **Validation phase:** when the closeTransaction request is received, the transaction is validated to establish whether or not its operations on objects conflict with other transaction.
    - **Update phase**: changes in tentative versions are made permanent if transaction is validated

METHOD FOR CONCURRENT CONTROL

- **Optimistic concurrency control:**
  - **Validation of transactions:** use the read-write conflict rules to ensure that the scheduling of a transaction is serially equivalent with respect to all other overlapping transactions.
  - **Backward validation:** check the transaction undergoing validation with other preceding overlapping transactions (enter the validation phase before).
    - Read set of the transaction being validated is compared with the write sets of other transactions that have already committed.
  - **Forward validate:** check the transaction undergoing validation with other later transactions
    - Write set of the transaction being validated is compared with the read sets of other overlapping active transactions (still in working phase).

METHOD FOR CONCURRENT CONTROL

- **Timestamp ordering:**
  - Each transaction is assigned a unique timestamp values when it starts.
  - Timestamp defines its position in the time sequence of transaction.
  - Basic timestamp ordering rule:
    - A transaction's request to write an object is valid only if that object was last read and written by earlier transactions. A transaction's request to read an object is valid only if that object was last written by an earlier transactions.
  - No deadlock

METHOD FOR CONCURRENT CONTROL

- **Timestamp ordering write rule:**

if (T_c ≥ maximum read timestamp on D &&

T_c > write timestamp on committed version of D)

perform write operation on tentative version of D with write timestamp T_c

else */*write is too late*/*

abort transaction T_c

METHOD FOR CONCURRENT CONTROL

Figure 16.29 Operation conflicts for timestamp ordering

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* 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 *write* an object that has been *written* by any T_i where $T_i > T_c$. This requires that $T_c >$ the write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$. This requires that $T_c >$ the write timestamp of the committed object.

METHOD FOR CONCURRENT CONTROL

- **Timestamp ordering read rule:**

If (T_c> write timestamp on committed version of D)

{ let $D_{selected}$ be the version of D with the maximum write timestamp ≤ Tc

if ($D_{selected}$ is committed)

perform read operation on the version $D_{selected}$

else

wait until the transaction that made version $D_{selected}$ commits or aborts then reapply the read rule

}

Else

abort transaction T_c

METHOD FOR CONCURRENT CONTROL

- **Multiversion timestamp ordering:**
  - A list of old committed versions as well as tentative versions is kept for each object.
  - Read operations that arrive too late need not be rejected.

- **Multiversion timestamp ordering write rule:**

  If (read timestamp of $D_{maxEarlier} \leq T_c$)

  perform write operation on tentative version of D with write timestamp T_c

  Else

  abort transaction T_c

Comparison of methods for concurrency control

- Three separate method for controlling concurrent access to shared data are strict **two-phase locking**, **optimistic methods** and **timestamp ordering**.
- All of the methods carry some overheads in the time and space they require and they all limit to some extent the potential for concurrent operation.
- Timestamp ordering and two phase locking both use pessimistic approach but timestamp ordering decides the serialization order statically whereas two-phase locking decides the serialization dynamically –according to the order in which objects are accessed.
- Timestamp ordering is better then strict two-phase locking for read-only transactions. Two phase locking is better when operations in transaction are predominantly updates.

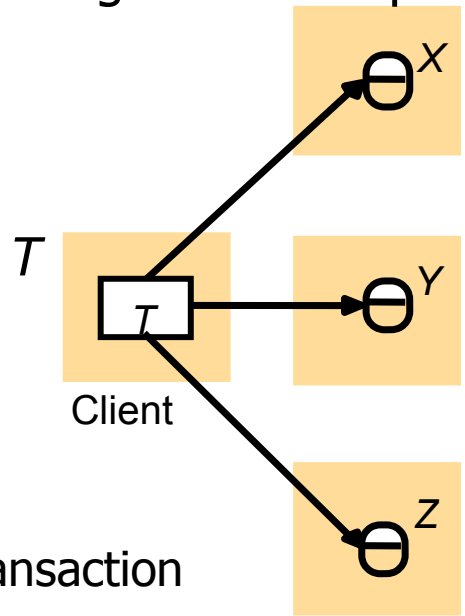
Distributed transactions

A client transaction becomes distributed if it invokes in several servers.

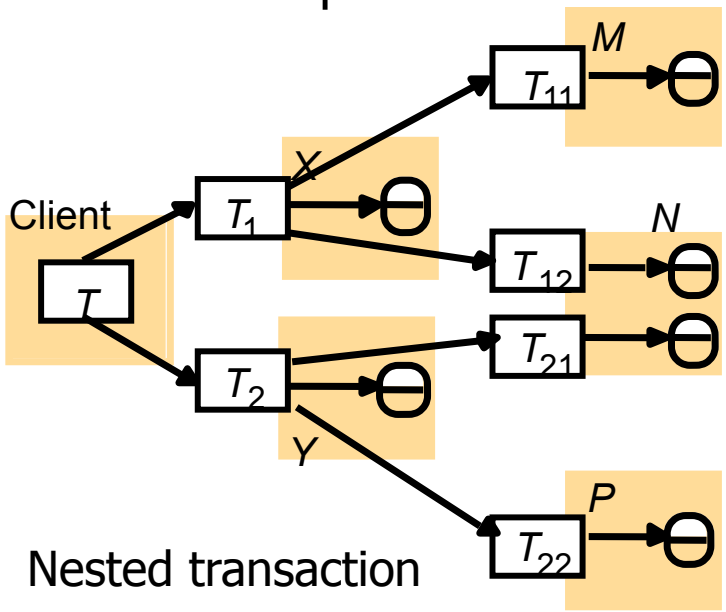
Flat transaction: client request to more then one server. T invokes operation in objects in 3 server x, y, z so T is flat transaction. A flat client transaction completes each its request before going on the next one. Therefore each transaction accesses servers object sequentially. When servers use locking, a transaction can only be waiting for one object at a time.

Nested transactions:

sub transaction at some level can be run concurrently. It gives better performance then in simple transaction.



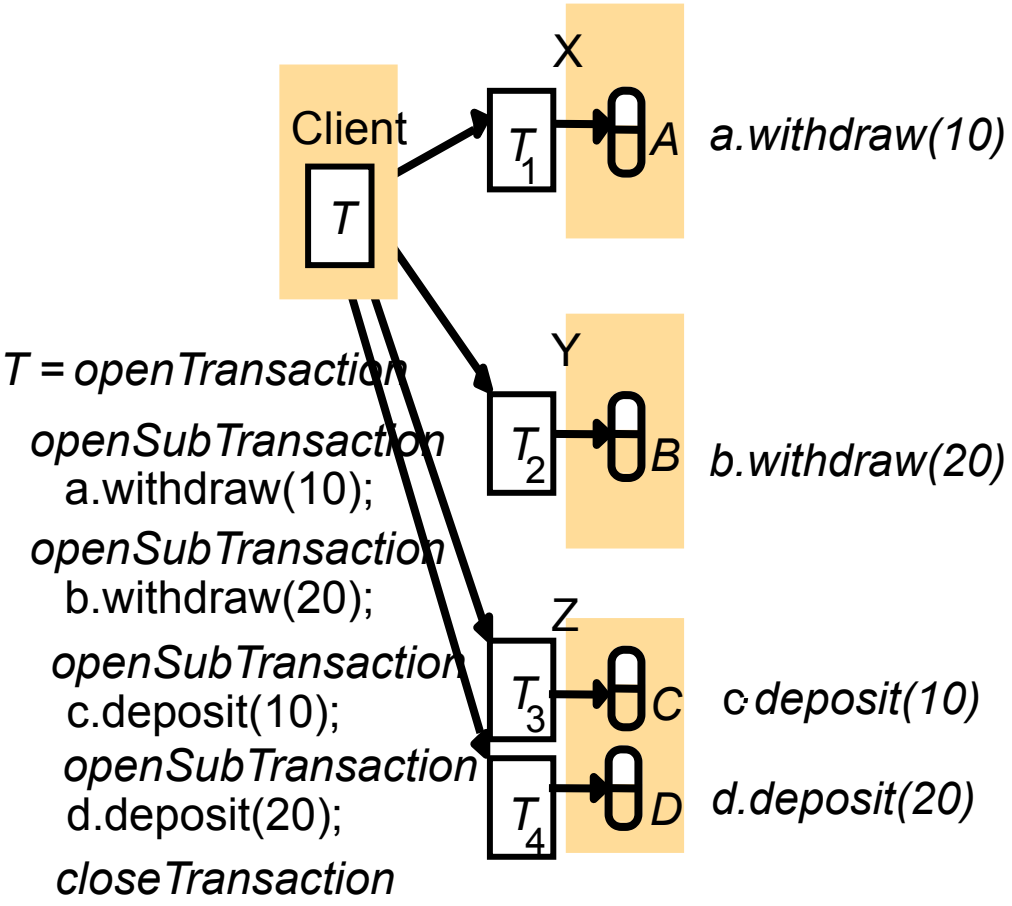
Flat transaction



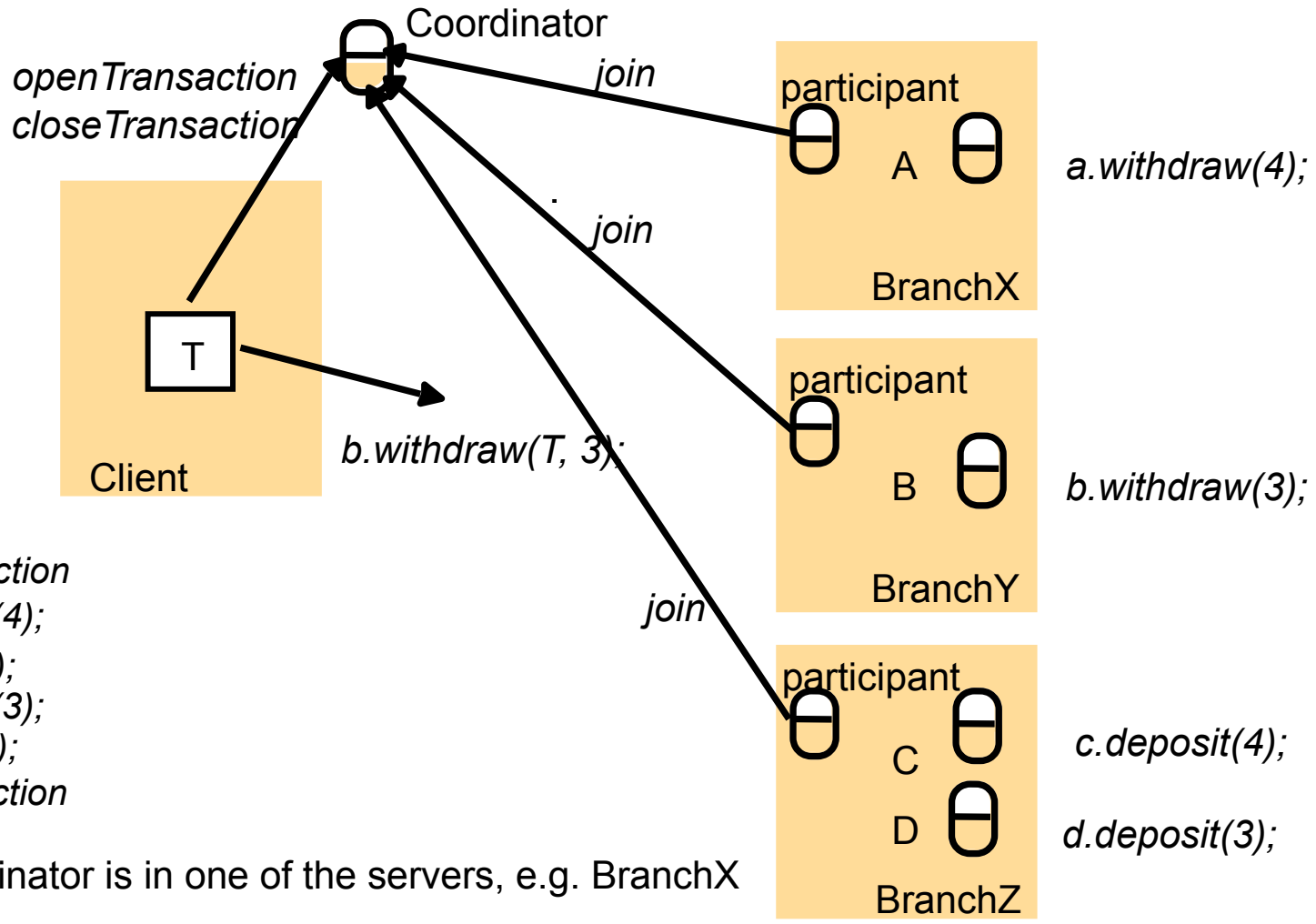
Nested transaction

Nested banking transaction

Consider a distributed transaction in which a client transfer $10 from account A to C and then transfer $20 from B to D. Account A and B are on separate servers X and Y and accounts C and D are at server Z. if this transaction is structured as a set of four nested transactions, as shown in figure, the four requests (two deposit and two withdraw) can run in parallel and overall effect can be achieved with better performance then a simple transaction in which the four operations are invoked sequentially.



A distributed banking(flat) transaction



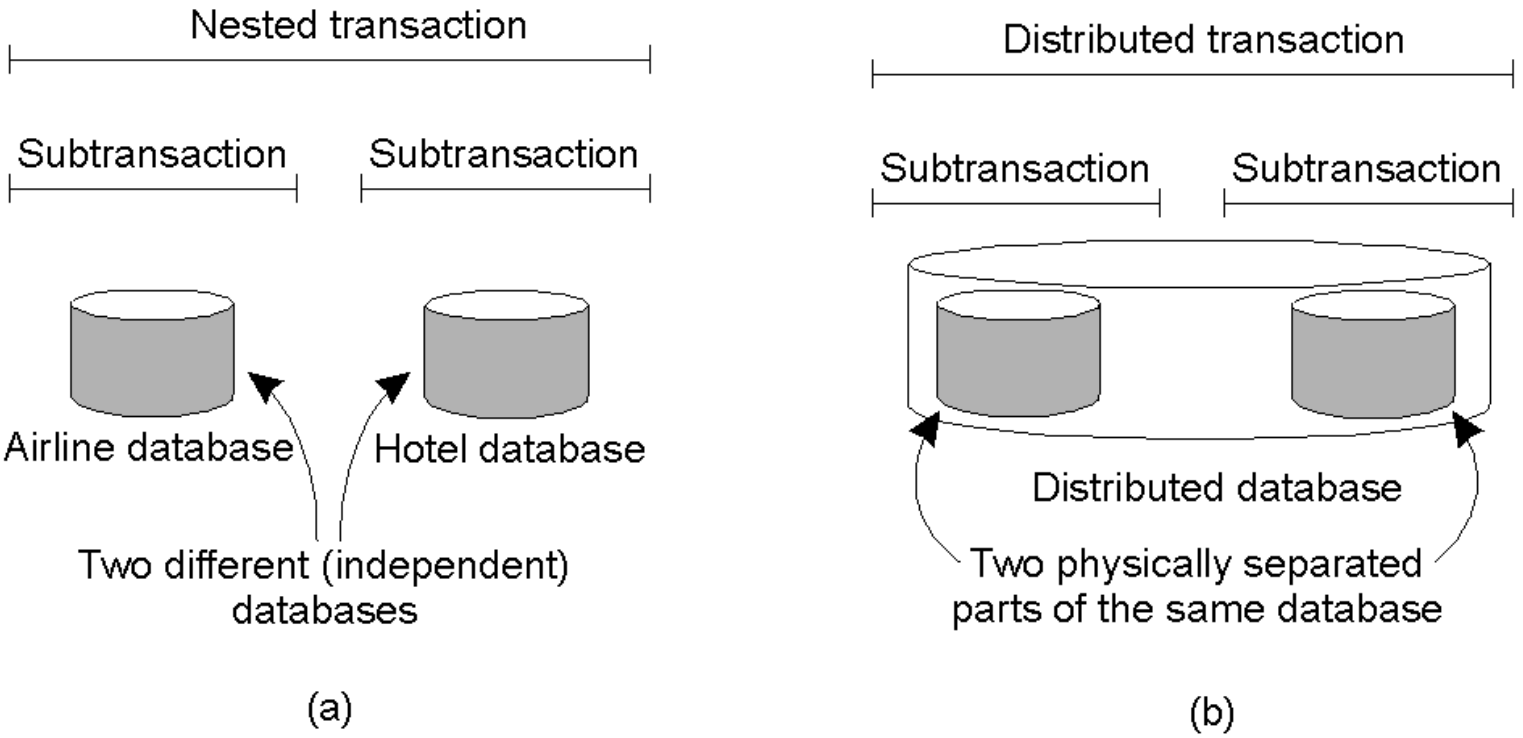
T = openTransaction
a.withdraw(4);
c.deposit(4);
b.withdraw(3);
d.deposit(3);
closeTransaction

Note: the coordinator is in one of the servers, e.g. BranchX

A distributed banking transaction

- A client starts a transaction by sending an *openTransaction* request to the server.
- The coordinator that is Connected carries out the *openTransaction* and returns the resulting transaction identifiers to the client.
- Transaction identifier for distributed transactions must be unique within a distributed system.

Nested vs Distributed Transactions



Atomic commit protocols

- The atomicity of transaction requires that when a distributed transaction comes to end, either all of its operations are carried out or none of them.
- In case of distributed transaction, the client has requested the operations at more then one servers.
- **Atomic commit protocol** are designed to achieve this effect even if server crashes during their execution.
- The two phase commit protocol allows a server to decide to abort unilaterally. It includes timeout actions to deal with delays due to servers crashing.
- The two phase protocol can take an unbounded amount of time to complete but is guaranteed to complete eventually.

Operations for two-phase commit protocol

canCommit?(trans)-> Yes / No

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

doCommit(trans)

Call from coordinator to participant to tell participant to commit its part of a transaction.

doAbort(trans)

Call from coordinator to participant to tell participant to abort its part of a transaction.

haveCommitted(trans, participant)

Call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) -> Yes / No

Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

The two-phase commit protocol

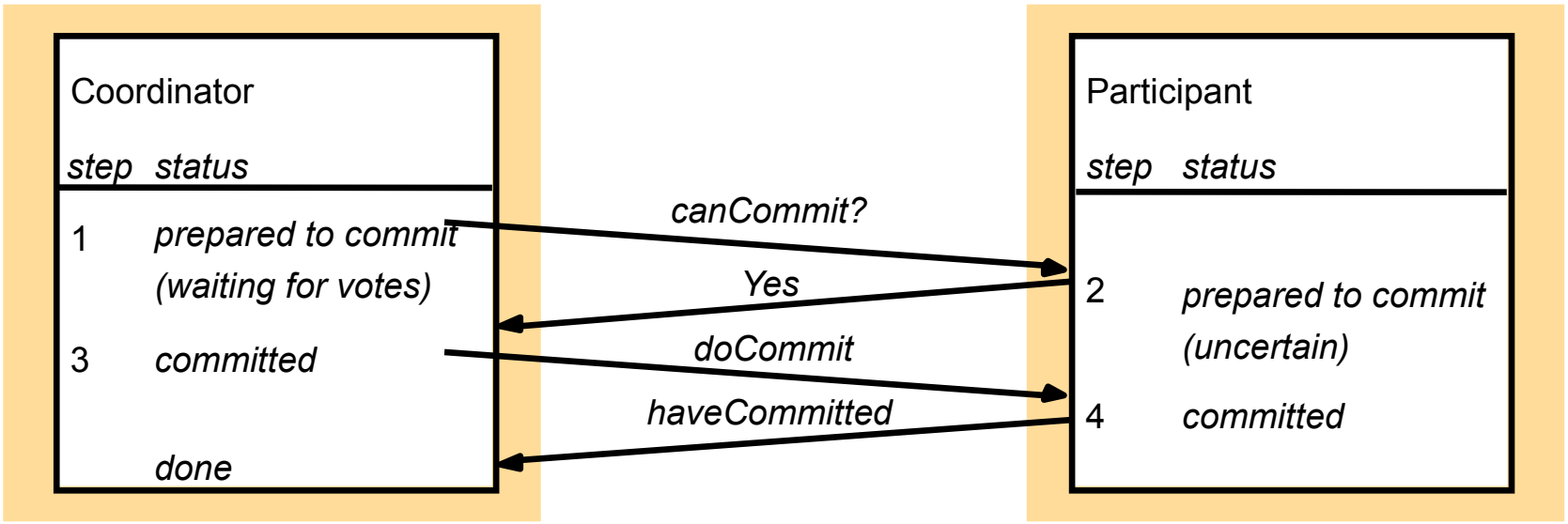
Phase 1 (voting phase):

1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Communication in two-phase commit protocol



Operations in coordinator for nested transactions

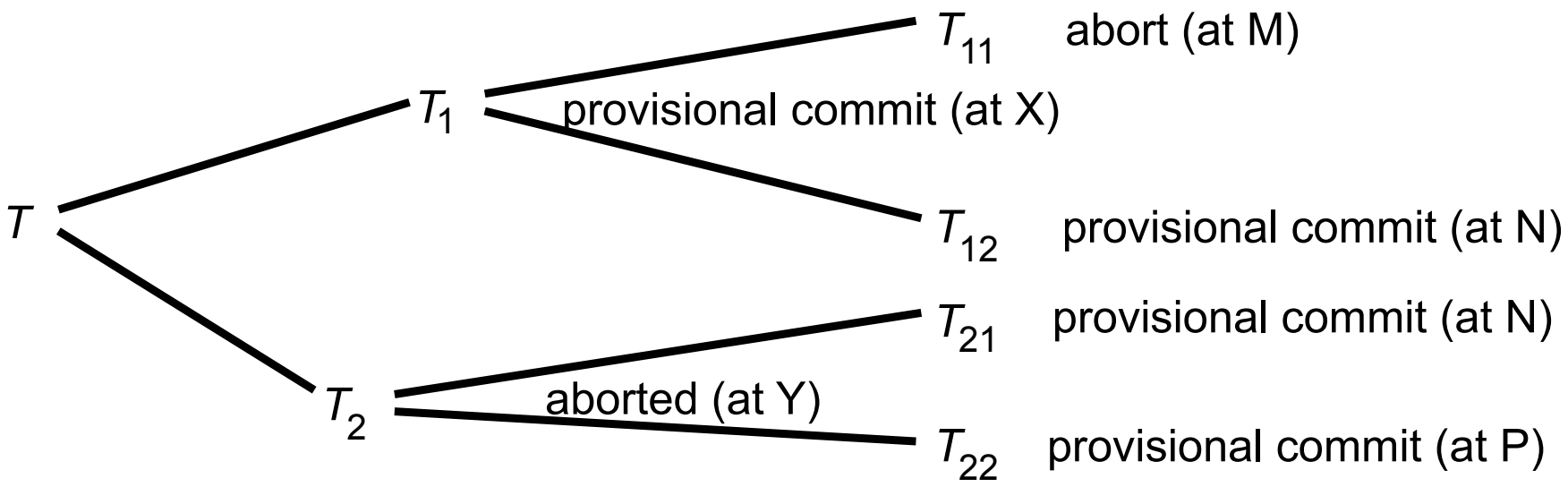
openSubTransaction(trans) -> subTrans

Opens a new subtransaction whose parent is *trans* and returns a unique subtransaction identifier.

getStatus(trans)-> committed, aborted, provisional

Asks the coordinator to report on the status of the transaction *trans*. Returns values representing one of the following: *committed*, *aborted*, *provisional*.

Transaction *T* decides whether to commit



Information held by coordinators of nested transactions

Coordinator of transaction	*Child transactions*	*Participant*	*Provisional commit list*	*Abort list*
T	T_1 , T_2	yes	T_1 , T_{12}	T_{11} , T_2
T_1	T_{11} , T_{12}	yes	T_1 , T_{12}	T_{11}
T_2	T_{21} , T_{22}	no (aborted)		T_2
T_{11}		no (aborted)		T_{11}
T_{12} , T_{21}		T_{12} but not T_{21}	T_{21} , T_{12}	
T_{22}		no (parent aborted)	T_{22}	

*canCommit?*for hierarchic two-phase commit protocol

canCommit?(trans, subTrans) -> Yes / No

->*call from coordinator to participant*

->Call a coordinator to ask coordinator of child subtransaction whether it can commit a subtransaction *subTrans*. The first argument *trans* is the transaction identifier of top-level transaction. Participant replies with its vote *Yes* / *No*.

canCommit? for flat two-phase commit protoco

canCommit?(trans, abortList) -> Yes / No

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote *Yes* / *No*.

Transaction Recovery

a. Logging:

- The recovery file represents a log containing the history of all the transactions performed by a server.

- The history consists of values of objects, transaction status entries and intentions lists of transactions. The order of the entries in the log reflects the order in which transactions have Prepared, committed and aborted at that server.

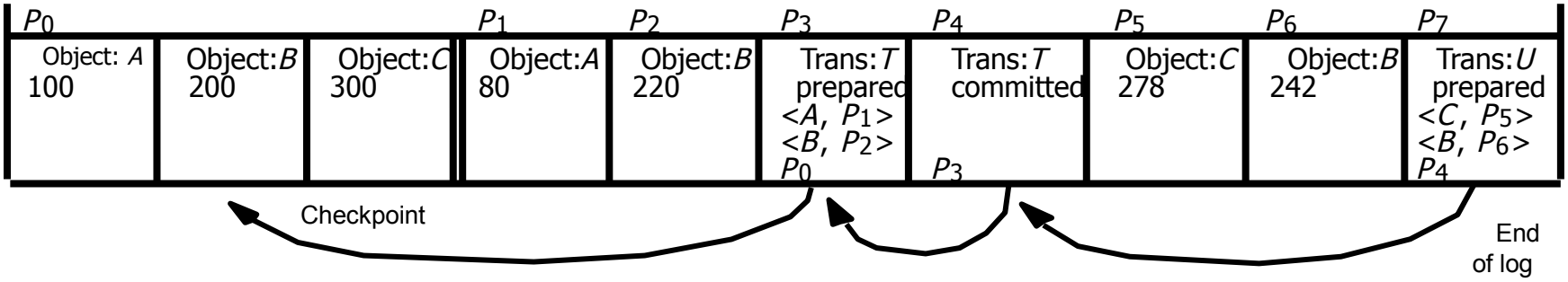
- Recovery file contain a recent snapshot of the values of all the objects in the server followed by a History of transactions after the snapshot.

Transaction Recovery

Types of entry in a recovery file

Type of entry	*Description of contents of entry*
Object	A value of an object.
Transaction status	Transaction identifier, transaction status (*prepared*, *committed*, *aborted*) and other status values used for the two-phase commit protocol.
Intentions list	Transaction identifier and a sequence of intentions, each of which consists of <identifier of object>, <position in recovery file of value of object>.

Log for banking service



TRANSACTION T	TRANSACTION U

balance = b.getBalance( );
b.setBalance(balance*1.1);
a.withdraw(balance/10);
	balance = b.getBalance( );
	b.setBalance(balance*1.1);
	c.withdraw(balance/10);

balance = b.getBalance( );
$200
b.setBalance(balance*1.1);
$220
 	
a.withdraw(balance/10);
$80
	
	balance = b.getBalance( ); $220
	b.setBalance(balance*1.1);
	$242
	
	c.withdraw(balance/10) ;
	$278

Description of Log for banking service

- The log was recently reorganized, and entries to all the left of schedule the double line represent a snapshot of the values of A,B and C before transactions T and U started.
- A ,B and C are unique identifiers for objects.
- We show the situation when transaction T has committed and transaction U has prepared but not committed.
- When transaction prepares to commit, the values of objects A and B are written at positions p1 and p2 in the log, followed by a prepared transaction status entry for T with its intentions list (<A,P1>, <B,P2>).
- When transaction T commits, a committed transaction status entry for T is put at position P4. when transaction U prepares to commit, the values of objects C and B are written as positions P5 and P6 in the log, followed by a prepared transaction status entry for U with its intentions list<C,p5>,<B,P6>)

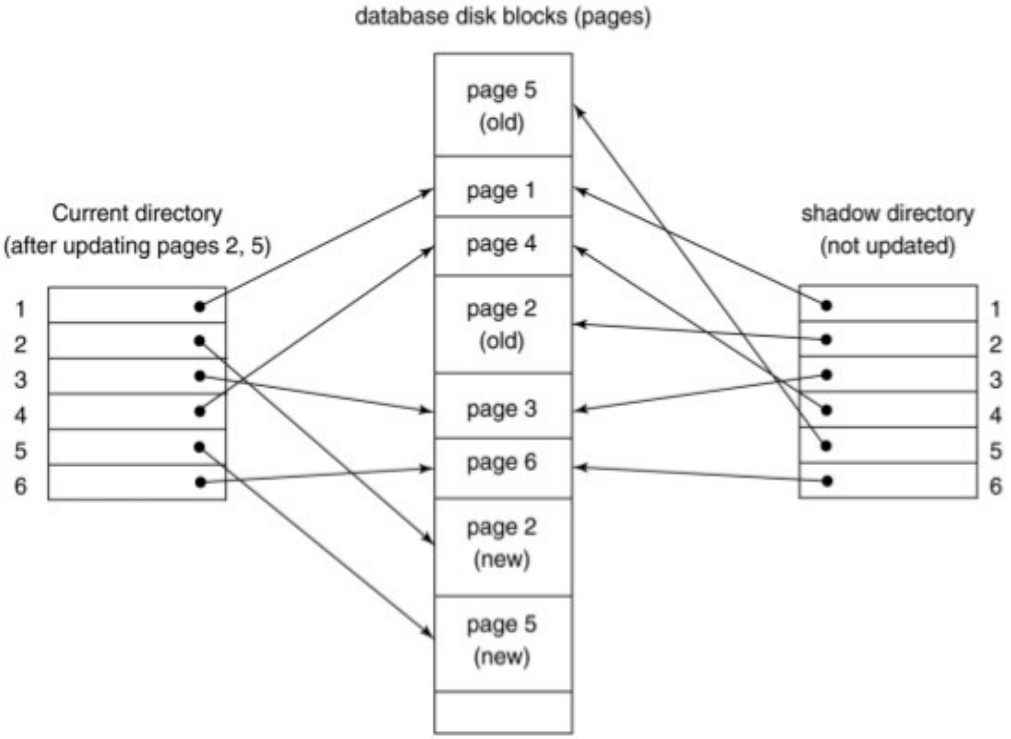
Shadow Paging

- **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- **Idea:** maintain *two* page tables during the lifetime of a transaction – the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
  - Shadow page table is never modified during execution
- To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
- Whenever any page is about to be written for the first time
  - A copy of this page is made onto an unused page.
  - The current page table is then made to point to the copy
  - The update is performed on the copy

Example Shadow paging

In this example the page2 and page 5 are modified.

FIGURE 19.5 An example of shadow paging.



Elmasri/Navathe, Fundamentals of Da

Shadow Paging (Cont.)

- To commit a transaction :
  1. Flush all modified pages in main memory to disk
  2. Output current page table to disk
  3. Make the current page table the new shadow page table, as follows:
     - keep a pointer to the shadow page table at a fixed (known) location on disk.
     - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
- Once pointer to shadow page table has been written, transaction is committed.
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- Pages not pointed to from current/shadow page table should be freed (garbage collected).

Shadow Paging (Cont.)

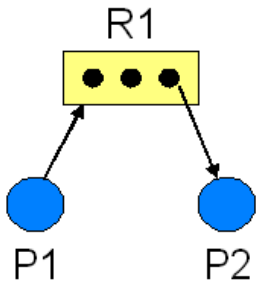
- Advantages of shadow-paging over log-based schemes
  - no overhead of writing log records
  - recovery is trivial
- Disadvantages :
  - Copying the entire page table is very expensive
    - Can be reduced by using a page table structured like a B$^+$-tree
      - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
  - Commit overhead is high even with above extension
    - Need to flush every updated page, and page table
  - Data gets fragmented (related pages get separated on disk)
  - After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
  - Hard to extend algorithm to allow transactions to run concurrently
    - Easier to extend log based schemes

Deadlock

Deadlocks – An Introduction

- What Are DEADLOCKS ?
  - A Blocked Process which can never be resolved unless there is some outside Intervention.

- For Example:-
  - Resource (reusable) R1 is requested by Process P1 but is held by Process P2.



Introduction to Deadlock

Formal definition

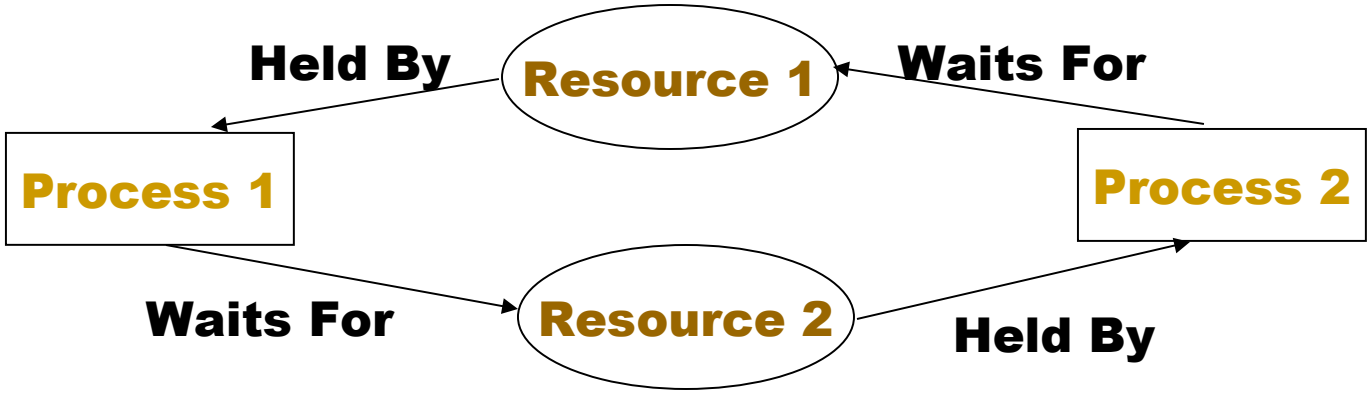
- *"A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause."*
- Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other member in the set, and all the processes continue to wait forever.
- That is, none of the processes can-
  - run
  - release resources
  - be awakened

Distributed deadlock

- Problem definition
  - Permanent blocking of a set of processes that either compete for system resources or communicate with each other
  - No node has complete and up-to-date knowledge of the entire distributed system
  - Message transfers between processes take unpredictable delays
- System state representation with wait-for graphs (WFG)
  - Nodes are processes, *P1*, *P2*, etc.
  - Directed edge from *P1* to *P2* if *P1* blocked and waiting for *P2* to release a resource
  - System is deadlocked if there is a directed cycle

Illustrating A Deadlock

- Wait-For-Graph (WFG)
  - Nodes – Processes in the system
  - Directed Edges – Wait-For blocking relation



- A Cycle represents a Deadlock
- Starvation - A process' execution is permanently halted.

Deadlocks in Distributed Systems

- Resource deadlock
  - Set of deadlocked processes, where each process waits for a resource held by another process (e.g., data object in a database, I/O resource on a server)
  - Most Common.
  - Occurs due to lack of requested Resource

- Communication deadlocks:
  - Set of deadlocked processes, where each process waits to receive messages (communication) from other processes in the set.

Timestamped Deadlock-Prevention Scheme

- Each process P_i is assigned a unique priority number
- Priority numbers are used to decide whether a process P_i should wait for a process P_j; otherwise P_i is rolled back
- The scheme prevents deadlocks
  - For every edge $P_i \rightarrow P_j$ in the wait-for graph, P_i has a higher priority than P_j
  - Thus a cycle cannot exist
- → There's possibility of starvation
  - Can be avoided using timestamps

Wait-Die Scheme

- Based on a nonpreemptive technique
- If P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a smaller timestamp than does P_j (P_i is older than P_j)
  - Otherwise, P_i is rolled back (*dies*)
- Example: Suppose that processes P_1, P_2, and P_3 have timestamps 5, 10, and 15 respectively
  - if P_1 requests a resource held by P_2, then P_1 will wait
  - If P_3 requests a resource held by P_2, then P_3 will be rolled back

Wound-Wait Scheme

- Based on a preemptive technique; counterpart to the wait-die system
- If P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a larger timestamp than does P_j (P_i is younger than P_j). Otherwise P_j is rolled back (P_j is *wounded* by P_i)
- Example: Suppose that processes P_1, P_2, and P_3 have timestamps 5, 10, and 15 respectively
  - If P_1 requests a resource held by P_2, then the resource will be preempted from P_2 and P_2 will be rolled back
  - If P_3 requests a resource held by P_2, then P_3 will wait

Handling Deadlocks in resource allocation

- **Deadlock Prevention**
  - Prioritize processes. Assign resources accordingly.
  - Provide all required resources from start itself.
  - Make Prior Rules:
    - For Ex. – Process P1 cannot request resource R1 unless it releases resource R2.
- **Drawbacks**
  - Inefficient and effects Concurrency.
  - Future resource requirement unpredictable.
  - Starvation possible.

Handling Deadlocks in resource allocation

Deadlock avoidance

- Decision made dynamically, before allocating a resource, the resulting global system state is checked - if safe, allow allocation
- Disadvantages
  - Every site has to maintain global state of system (extensive overhead in storage and communication)
  - Different sites may determine (concurrently) that state is safe, but global state may be unsafe: verification for safe global state by different sites must be mutually exclusive
  - Large overhead to check for every allocation (distributed system may have large number of processes and resources)
  - Requires Prior resource requirement information for all processes.
- Conclusion: Deadlock avoidance impractical in distributed systems

Handling Deadlocks

- **Deadlock Avoidance**
  - Only fulfill those resource requests that won't cause deadlock in the future.
  - Simulate resource allocation and determine if resultant state is safe or not.
- **Drawbacks**
  - Inefficient.
  - Requires Prior resource requirement information for all processes.
  - High Cost of scalability.

Deadlock in resource allocation

Deadlock Detection

- Principle of operation
  - Detection of a cycle in WFG proceeds concurrently with normal operation
- Requirements for the deadlock detection and resolution algorithms
  - Detection
    - The algorithm must detect all existing deadlock in finite time
    - The algorithm should not report non-existent (phantom) deadlock
  - Resolution (recovery)
    - All existing wait-for dependencies in WFG must be removed, i.e. roll-back one or more processes that are deadlocked and give their resources to other blocked processes
- Observation
  - Deadlock detection is the most popular strategy for handling deadlocks in distributed systems

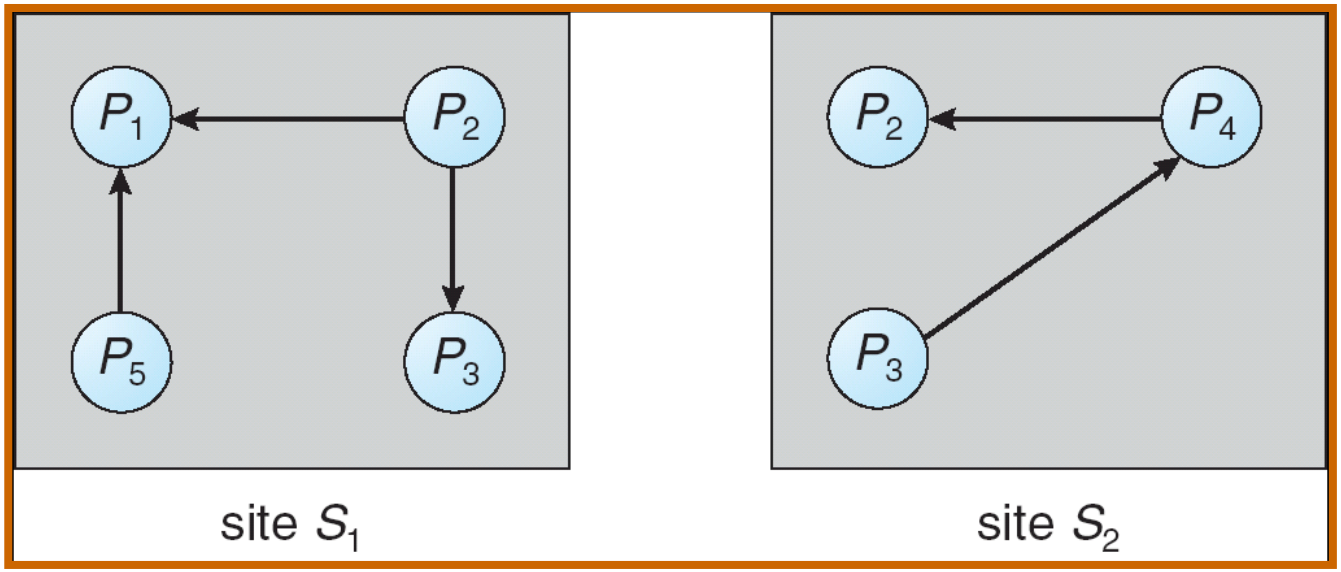
Handling Deadlocks

- **Deadlock Detection**
  - Resource allocation with an optimistic outlook.
  - Periodically examine process status.
  - Detect then break the Deadlock.
- **Resolution** – Roll back 1 or More processes and break dependency.

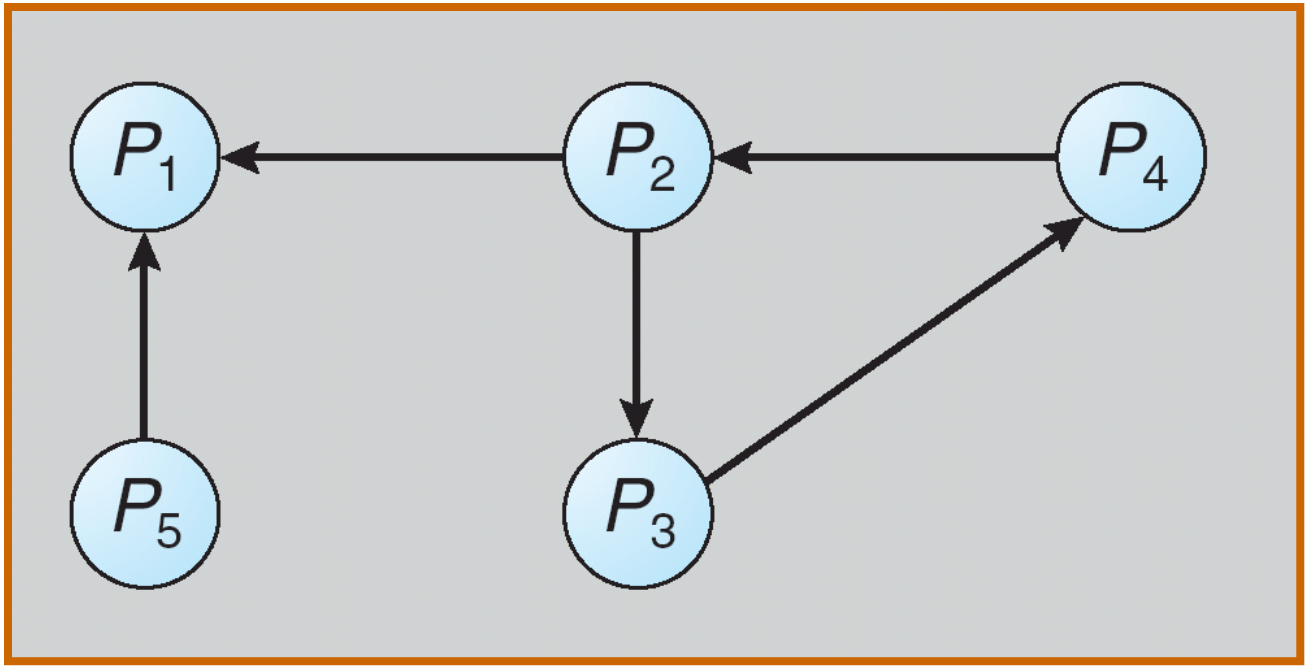
Deadlock Detection

- Use **wait-for** graphs
  - **Local wait-for graphs at each local site**. The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site
  - May also use a **global wait-for graph.** This graph is the union of all local wait-for graphs.

Two Local Wait-For Graphs



Global Wait-For Graph



Deadlock in resource allocation: Algorithms for distributed deadlock detection

Deadlock Detection (cont.)

- Control for distributed deadlock detection can be:
  a. Centralized
  b. Distributed
  c. Hierarchical

1. Centralized deadlock detection algorithms

- Each site keeps a local wait-for graph
- A global wait-for graph is maintained in a single coordination process
- There are three different options (points in time) when the wait-for graph may be constructed:
  1. Whenever a new edge is inserted or removed in one of the local wait-for graphs
  2. Periodically, when a number of changes have occurred in a wait-for graph
  3. Whenever the coordinator needs to invoke the cycle-detection algorithm

Detection Algorithm Based on Option 3

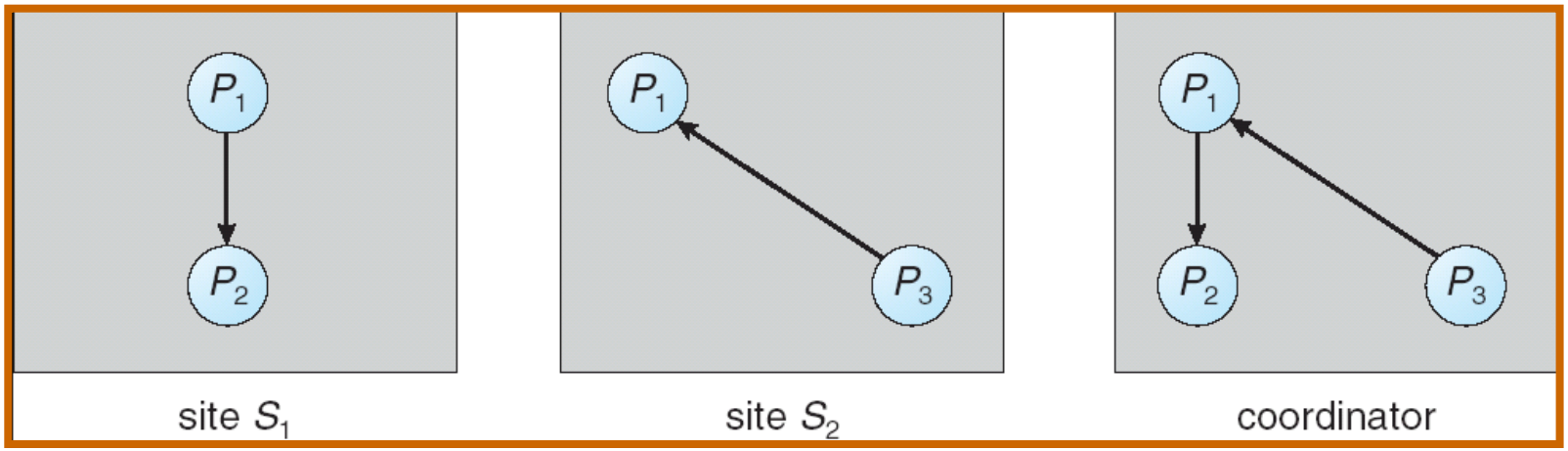
- Append unique identifiers (timestamps) to requests form different sites
- When process P_i, at site A, requests a resource from process P_j, at site B, a request message with timestamp TS is sent
- The edge $P_i \rightarrow P_j$ with the label TS is inserted in the local wait-for of A. The edge is inserted in the local wait-for graph of B only if B has received the request message and cannot immediately grant the requested resource

The Algorithm

1. The controller sends an initiating message to each site in the system
2. On receiving this message, a site sends its local wait-for graph to the coordinator
3. When the controller has received a reply from each site, it constructs a graph as follows:
   (a) The constructed graph contains a vertex for every process in the system
   (b) The graph has an edge Pi → Pj if and only if
      (1) there is an edge Pi → Pj in one of the wait-for graphs,
      (2) Or an edge Pi → Pj with some label TS appears in more than one wait-for graph

If the constructed graph contains a cycle ⇒ deadlock

Local and Global Wait-For Graphs



Deadlock Detection

2. Hierarchical deadlock detection algorithms

- Sites organized in a tree structure with one site at the root of the tree
- Each node (except for leaf nodes) has information about the dependent nodes
- Deadlock is detected by the node that is the common ancestor of all sites which have resource allocations in conflict
- Deadlock is detected at the lowest level

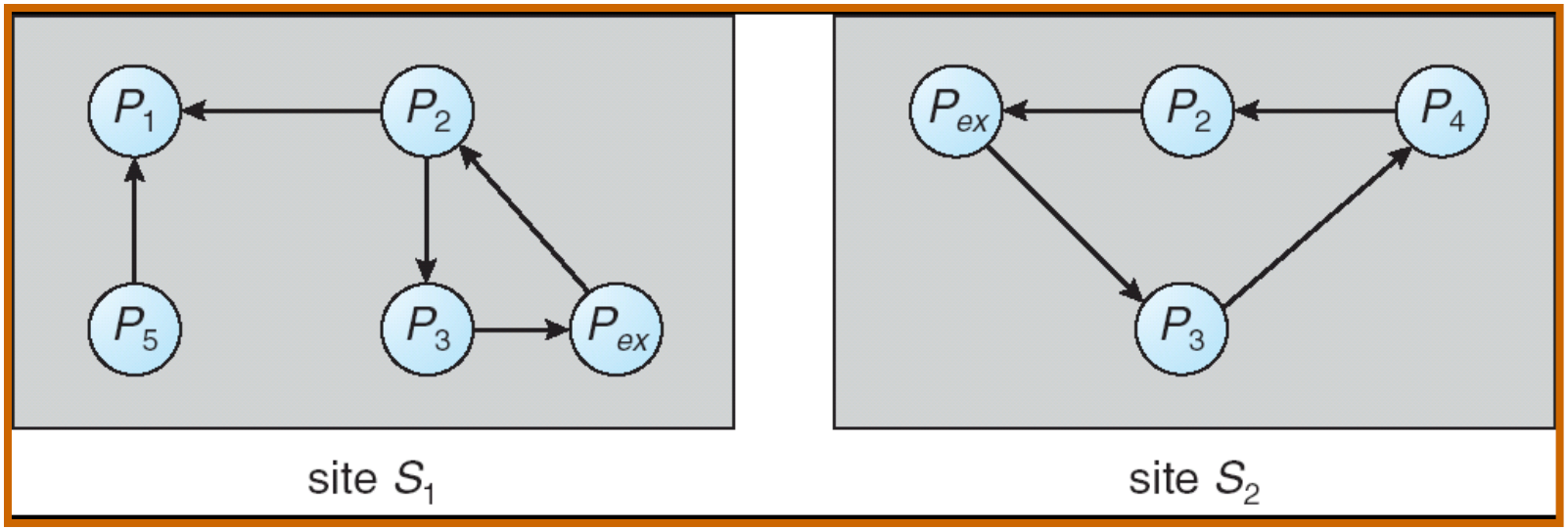
3. Distributed deadlock detection algorithms

- **Principles**
  - All sites responsible for detecting a global deadlock
  - Global state graph distributed over many sites: several of them participate in detection
  - Detection initiated when a process suspected to be deadlocked
  - **Advantages**: No single point of failure, no congestion
  - **Disadvantages**: Difficult to implement
- **Types of algorithms**
  - **Path-pushing algorithms**
    - Each node builds a WFG based on local info & info from other sites
    - Detect and resolves local deadlocks
    - Transmits to other sites deadlock info in form of (waiting path)
  - **Edge-chasing algorithms**
    - Special messages (probes) sent along edges of WFG to detect a cycle
    - When blocked process receives probe, resends it on its outgoing edges of WFG
    - When a process receives a probe it initiated, declares deadlock

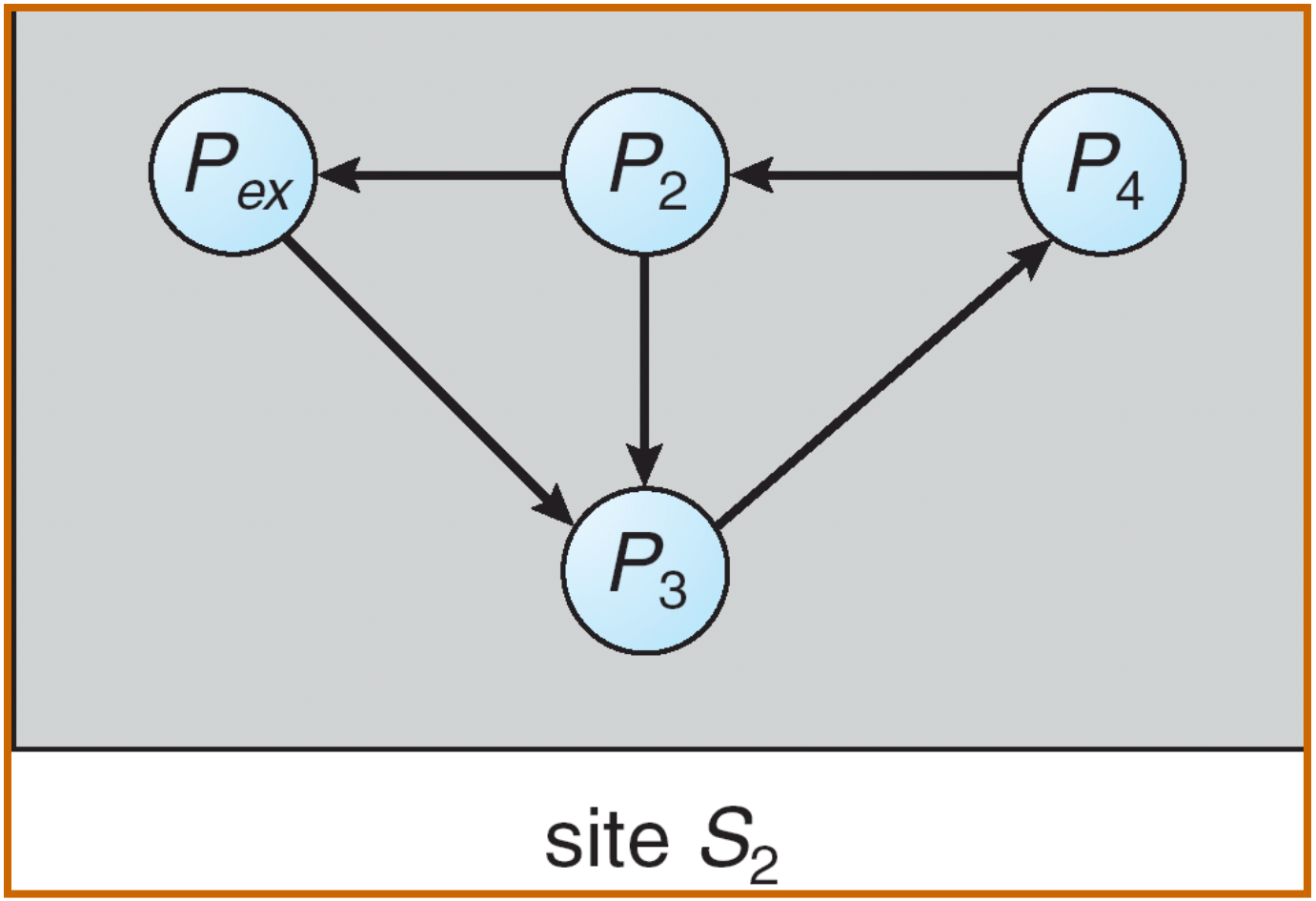
Fully Distributed Approach

- All controllers share equally the responsibility for detecting deadlock
- Every site constructs a wait-for graph that represents a part of the total graph
- We add one additional node P_{ex} to each local wait-for graph
- If a local wait-for graph contains a cycle that does not involve node P_{ex}, then the system is in a deadlock state
- A cycle involving P_{ex} implies the possibility of a deadlock
  - To ascertain whether a deadlock does exist, a distributed deadlock-detection algorithm must be invoked

Augmented Local Wait-For Graphs



Augmented Local Wait-For Graph in Site S2



Deadlock Detection Algorithms

- **Centralized Deadlock Detection**
  - Ho-Ramamoorthy's one and two phase algorithms.
- **Distributed Deadlock Detection**
  - Obermarck's Path Pushing Algorithm.
  - Chandy-Misra-Haas Edge Chasing algorithm.
- **Hierarchical Deadlock Detection**
  - Menasce-Muntz Algorithm.
  - Ho-Ramamoorthy's Algorithm.

Centralized Deadlock Detection

- **Ho-Ramamoorthy's 1-Phase Algorithm**
  - Each site maintains 2 Status Tables:
    - ✓ Process Table.
    - ✓ Resource Table.
  - One of the Sites Becomes the Central Control site.
  - The Central Control site periodically asks for the status tables.

Contd...

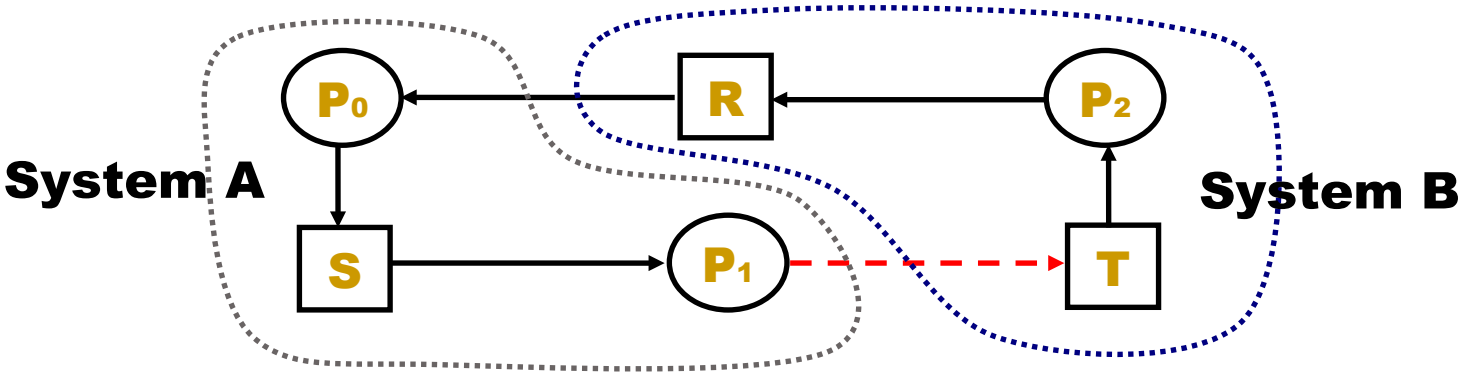
Centralized Deadlock Detection

Ho-Ramamoorthy's 1-Phase Algorithm Contd...

- Control site builds WFG using the status tables.
- Control site analyzes WFG and resolves any present cycles.

- **Shortcomings**
  - Phantom Deadlocks.(detection of false deadlock, a deadlock that doesn't really exist but is falsely detected)
  - High Storage & Communication Costs.

Phantom Deadlocks



- P1 *releases* resource *S* and *asks-for* resource *T*.
- 2 Messages sent to Control Site:
  1. *Releasing* S.
  2. *Waiting-for* T.
- Message 2 arrives at Control Site first. Control Site makes a WFG with cycle, detecting a phantom deadlock.

Centralized Deadlock Detection

- **Ho-Ramamoorthy's 2-Phase Algorithm**
  - Each site maintains a status table for processes.
  - Resources Locked & Resources Awaited.

Phase 1

  - Control Site periodically asks for these Locked & Waited tables.
  - It then searches for presence of cycles in these tables.

Contd...

Centralized Deadlock Detection

Ho-Ramamoorthy's 2-Phase Algorithm

Phase 2

- If cycles are found in phase 1 search, Control site makes 2nd request for the tables.
- The details found common in both table requests will be analyzed for cycle confirmation.

- **Shortcomings**
  - Phantom Deadlocks.

Distributed Deadlock Detection

- **Obermarck's Path-Pushing Algorithm**
  - Individual Sites maintain local WFG
  - A virtual node 'x' exists at each site.
  - Node 'x' represents external processes.
    - **Detection Process**
      - ✓ Case 1: If Site S_n finds a cycle not involving 'x' -> Deadlock exists.
      - ✓ Case 2: If Site S_n finds a cycle involving 'x' -> Deadlock possible.

Contd...

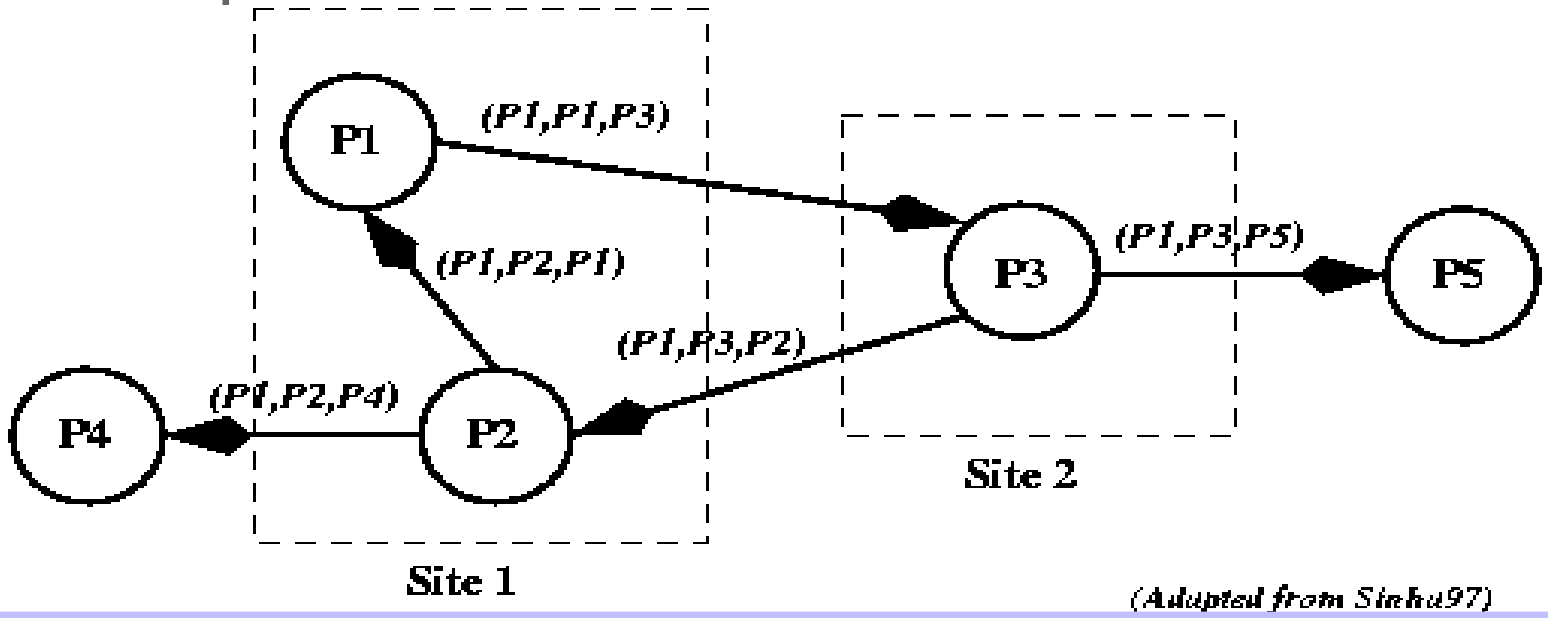
Obermarck's Path-Pushing Algorithm

- If Case 2 ->
  - Site S_n sends a message containing its detected cycles to other sites. All sites receive the message, update their WFG and re-evaluate the graph.
  - Consider Site S_j receives the message:
    - ✓ Site S_j checks for local cycles. If cycle found not involving 'x' (of S_j) -> Deadlock exists.
    - ✓ If site S_j finds cycle involving 'x' it forwards the message to other sites.
  - Process continues till deadlock found.

Distributed Deadlock Detection

- **Chandy-Misra-Haas Edge Chasing algorithm.**
  - The blocked process sends ‘probe’ message to the resource holding process.
  - ‘Probe’ message contains:
    - ✓ ID of blocked process.
    - ✓ ID of process sending the message.
    - ✓ ID of process to which the message was sent.
  - When probe is received by blocked process it forwards it to processes holding the requested resources.
  - If Blocked Process receives its own probe -> Deadlock Exists.

Example…



(Adapted from Sinha97)

- **Message Format: (Initiator, from, to)**
- In this case **P1** initiates the probe message, so that all the messages shown have **P1** as the initiator.
- When the probe message is received by process **P3**, it modifies it and sends it to two more processes.
- Eventually, the probe message returns to process **P1**. **Deadlock!**

Hierarchical Deadlock Detection

- **Menasce-Muntz Algorithm**
  - Sites (controllers) organized in a tree structure.
    - Leaf controllers manage local WFG.
    - Upper controllers handle Deadlock Detection.
      - Each Parent node maintains a Global WFG, union of WFG's of its children. Deadlock detected for its children.
    - Changes propagated upwards in the tree.

Hierarchical Deadlock Detection

- **Ho-Ramamoorthy's Algorithm**
  - Sites grouped into clusters.
  - Periodically 1 site chosen as central control site:
    - Central control site chooses controls site for other clusters.
  - Control site for each cluster collects the status graph there:
    - Ho-Ramamoorthy's 1-phase algorithm centralized DD algorithm used.
  - All control sites forward status report to Central Control site which combines the WFG and performs cycle search.

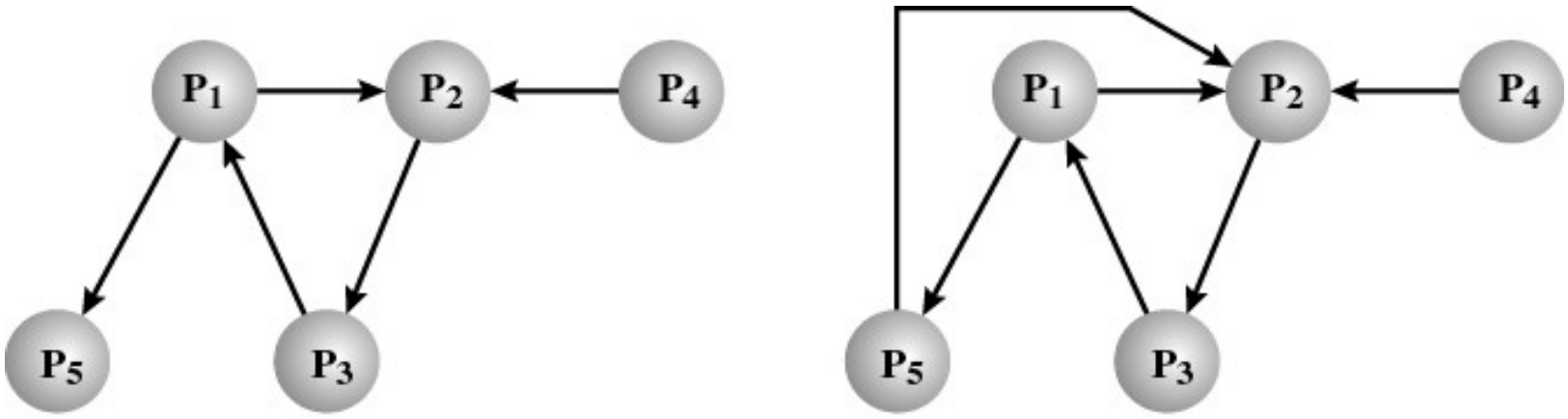
Deadlock in message communication

- **Mutual Waiting**
  - Deadlock conditions
    - Each of a group of processes is waiting for a message from another member of the group and here are no messages in transit
  - Concepts
    - Dependence set *(DS)* of process *Pi* is the set of all processes from which *Pi* is expecting a message
    - *Pi* can proceed when any of the expected messages arrive
    - Deadlock in a set *S* of processes
      - All processes in *S* are stopped, waiting for messages
      - *S* contains the dependence set of all processes in *S*
      - No messages are in transit between processes in *S*

Mutual Waiting (cont.)

- Resource deadlock vs. message deadlock
  - Resource deadlock
    - A deadlock exists if there is a cycle in the WFG
    - A process *Pi* is dependent on process *Pj* if *Pj* holds a resource that *Pi* needs
  - Message deadlock
    - All successors *Pj* of a process *Pi* in *S* are also in *S*
- Example
  - Fig. 14.16a
    - *P1* is waiting for a message from either *P2* or *P5*
    - *P5* is not waiting for any message; sends a message to P1, which is released
    - Links *(P1, P5)* and *(P1, P2)* are removed
    - No deadlock
  - Fig. 14.16b
    - *P5* is now waiting for a message from *P2*
    - *P2* is waiting for a message from *P3*
    - *P3* is waiting for a message from *P1*
    - *P1* is waiting for a message from *P2*
    - Deadlock
- Solution: prevention or detection

Deadlock in message communication: Mutual waiting (cont.)



(a) No deadlock

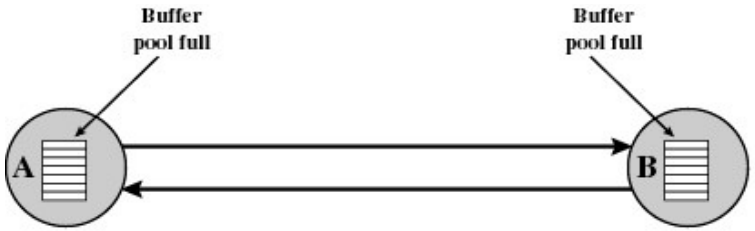
(b) Deadlock

Figure 14.16 Deadlock in Message Communication

Deadlock in message communication

- **Unavailability of Message Buffers**
  - Deadlock can occur in the allocation of buffers for the storage of messages in transit (Example: packet-switching data networks)

1. Direct store-and-forward deadlock
   - Example:
     - Two packet switching nodes, each using a common buffer pool from which buffers are assigned to packets on demand
     - Buffer space for A is filled with packets destined for B
     - Buffer space for B is filled with packets destined for A
     - Neither node can transmit or receive packets: deadlock

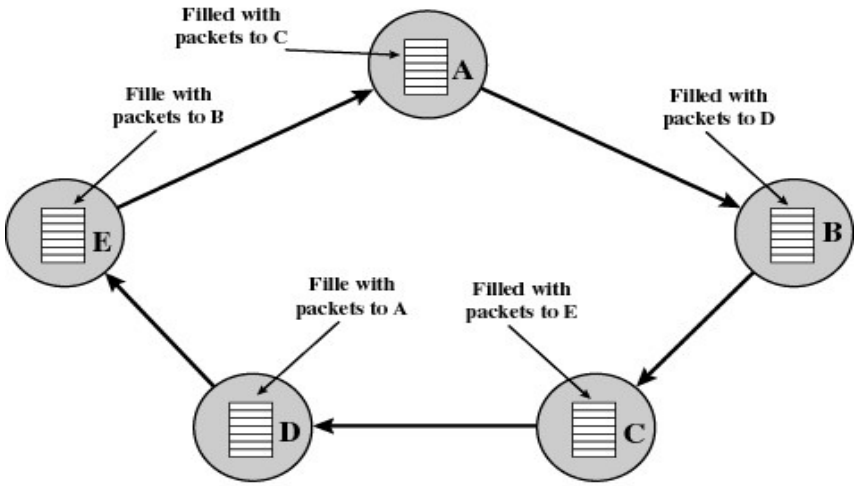


(a) Direct store-and-forward deadlock

   - Solution: Use separate buffers, one for each link

Deadlock in message communication

- **Unavailability of Message Buffers (cont.)**
  2. Indirect store-and-forward deadlock
     - For each node, the queue to the adjacent node in one direction is full with packets destined for the next node beyond



(b) Indirect store-and-forward deadlock

Figure 14.17 Store-and-Forward Deadlock

- **Unavailability of Message Buffers (cont.)**

3. Deadlock in a distributed OS using message passing for inter-process communication
   - A non-blocking *Send* operation requires a buffer to hold outgoing messages
   - Example 1
     - Process *X* has a buffer of size *n*
     - After sending *n* messages, buffer is full
     - When sending message *n+1*, process *X* will block until sufficient buffer is freed
   - Example 2
     - Process *X* has a buffer of size *n*
     - Process *Y* has a buffer of size *m*
     - Both buffers, *n* and *m*, become full and the two processes are blocked: deadlock
   - Possible solutions
     - Prevention: estimate maximum number of messages in transit and allocate corresponding number of buffers
     - Detection: detect deadlock and roll back one of the processes

Summary

Centralized Deadlock Detection Algorithms

- Large communication overhead.
- Coordinator is performance bottleneck.
- Possibility of single point of failure.

Distributed Deadlock Detection Algorithms

- High Complexity.
- Detection of phantom deadlocks possible.

Hierarchical Deadlock Detection Algorithms

- Most Common.
- Efficient.