Transaction Processing & Concurrency Control

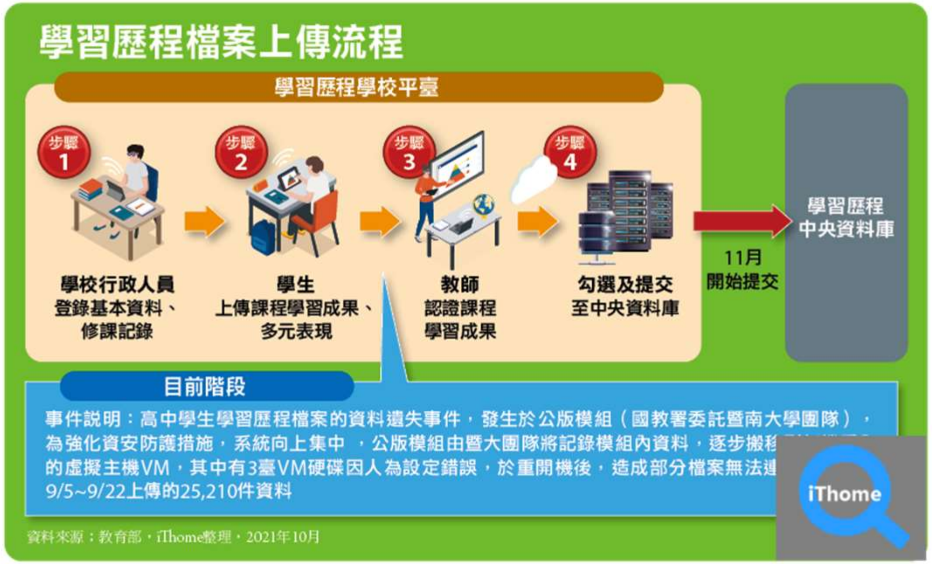
莊 裕 澤
Yuh-Jzer Joung
Dept. of Information Management
National Taiwan University

高鐵售票系統設計出錯 超賣 800餘個座位

高鐵試營運將於明天上午七時發出首班車，不過台灣高鐵公司執行長歐晉德今天表示，由於售票系統出現漏洞，板橋站南下的四○一班次超賣二百六十三個座位，類似超賣狀況在六日還有七班次，兩天總計超賣座位八百三十一個，將以開加班車並提供旅客補償因應 (2007.01.04)

高鐵公司表示，售票系統承包商神通電腦的程式設計有漏洞，當旅客在售票機上點選座位時，系統沒有立即鎖住，導致同一個座位重複賣。
高鐵公司表示，為避免再發生超賣，除將每列車保留座位提高為一百個，並將售票程式改為只要有人在選某個座位，座位就會立即鎖住，因此昨天下午以後沒有再出現重複訂位現象。

高中學習歷程檔案遺失 (2021.09.25)



事件說明：高中學生學習歷程檔案的資料遺失事件，發生於公版模組（國教署委託暨南大學團隊），為強化資安防護措施，系統向上集中 ，公版模組由暨大團隊將記錄模組內資料，逐步搬移
的虛擬主機VM，其中有3臺VM硬碟因人為設定錯誤，於重開機後，造成部分檔案無法
9/5~9/22上傳的25,210件資料

資料來源：教育部，iThome整理，2021年10月

source: ithome, 2021.10.08

新課綱上路兩年多，明年將迎來首批畢業生，學習歷程檔案將取代備審資料，作為升學的重要參考依據。未料卻傳出學生今年9月5日至9月20日期間上傳校內學習歷程平台的學習歷程檔案不見了，教育部證實暨南大學團隊~~研發的「公板模組」出包~~，並對外致歉，而經過一上午的盤點，初步評估受影響的範圍為學校81校、學生7854人、資料2萬5210件 (UDN 2021.09.25)

高中學習歷程檔案遺失 (2)

- 問題剖析
  - 技術面： VM設定失誤，工程師在新機房建立VM時，誤選了「測試環境設定」樣板，內建還原機制，重開機會自動清理硬碟
  - 維運面：移機至新機房，但尚未建立完整備份即開放系統供外界使用
    - 時程壓縮，新機房建置落後，導致系統搬遷延誤，但學生及學校端的作業時程已經排定
  - 制度面：受經費限制，由學校實驗室團隊長期承包教育部相關資訊系統

Ref. 【深入剖析學習歷程檔案事故4大原因】不只設定和維運問題，更是時程管理和制度面的考驗, iThome 2021.10.08

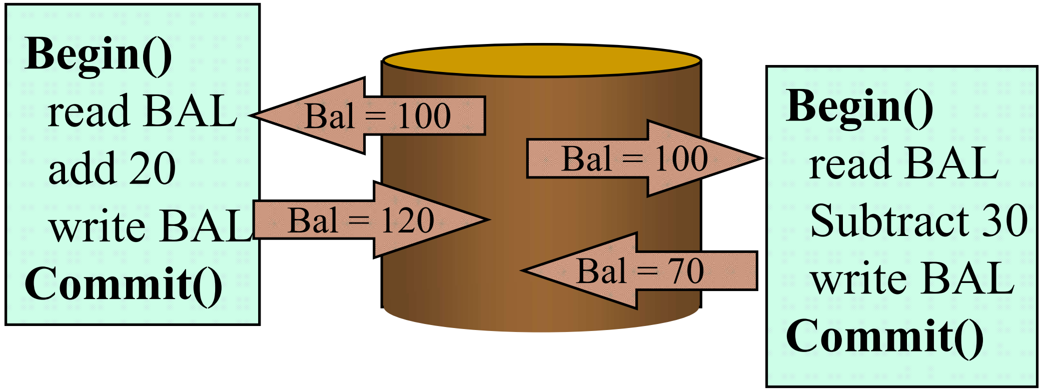
Outline

- Basics of Transactions
- Concurrency Control
  - Locking
  - Optimistic Concurrency Control
  - Timestamp Ordering
- Distributed Transactions
- Distributed Concurrency Control
- Commit Protocols
- Recovery Manager

Transactions

- Transaction: a sequence of operations, typically read/write to databases, that must be treated as a whole that either all occur, or nothing ever happen (*all or nothing*)
  - Transactions account for the major business activities.
    - Book a seat
    - Transfer money
    - Sell something
    - …
  - Transactions are typically executed concurrently and thus bring some challenges

Concurrency Brings Problems

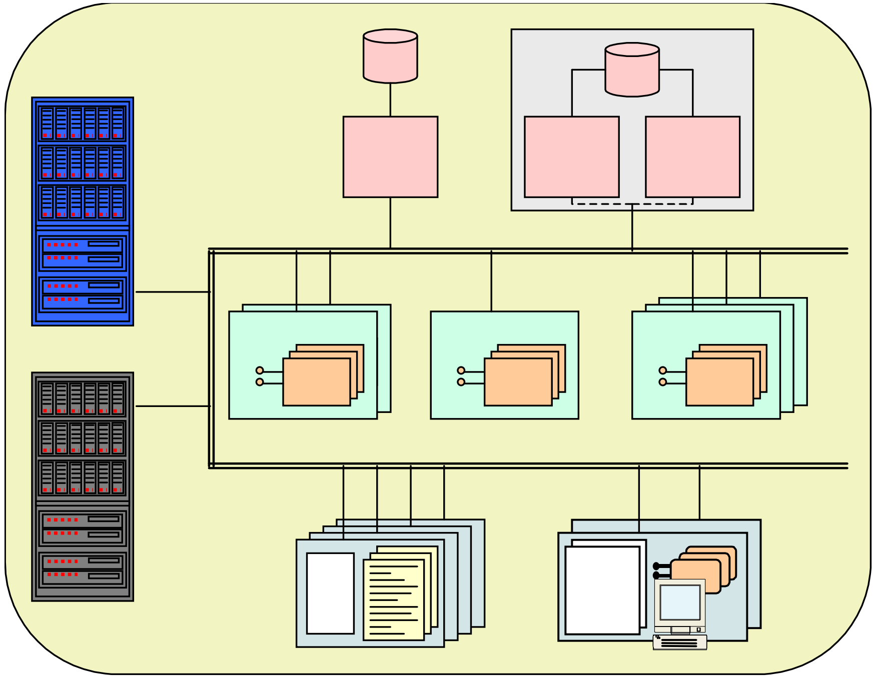


Begin()
read BAL
add 20
write BAL
Commit()

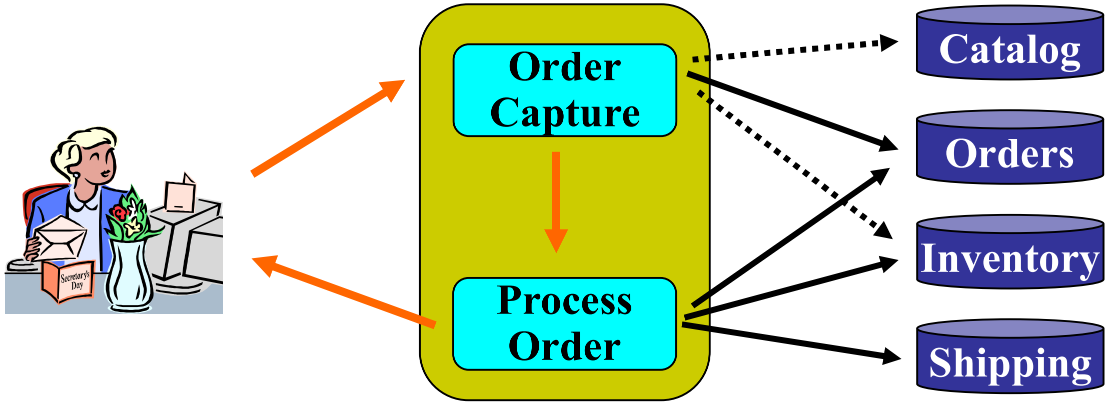
Begin()
read BAL
Subtract 30
write BAL
Commit()

Transaction Systems

- ❑ Efficiently handle high volumes of requests
- ❑ Avoid errors from concurrency
- ❑ Avoid partial results after failure
- ❑ Avoid downtime
- ❑ And … never lose data



Example: Order



Order Capture

Process Order

Catalog

Orders

Inventory

Shipping

Possible transactional operations:

- ✓ create order,
- ✓ update inventory,
- ✓ create shipping record
- ✓ update order status.

application program components

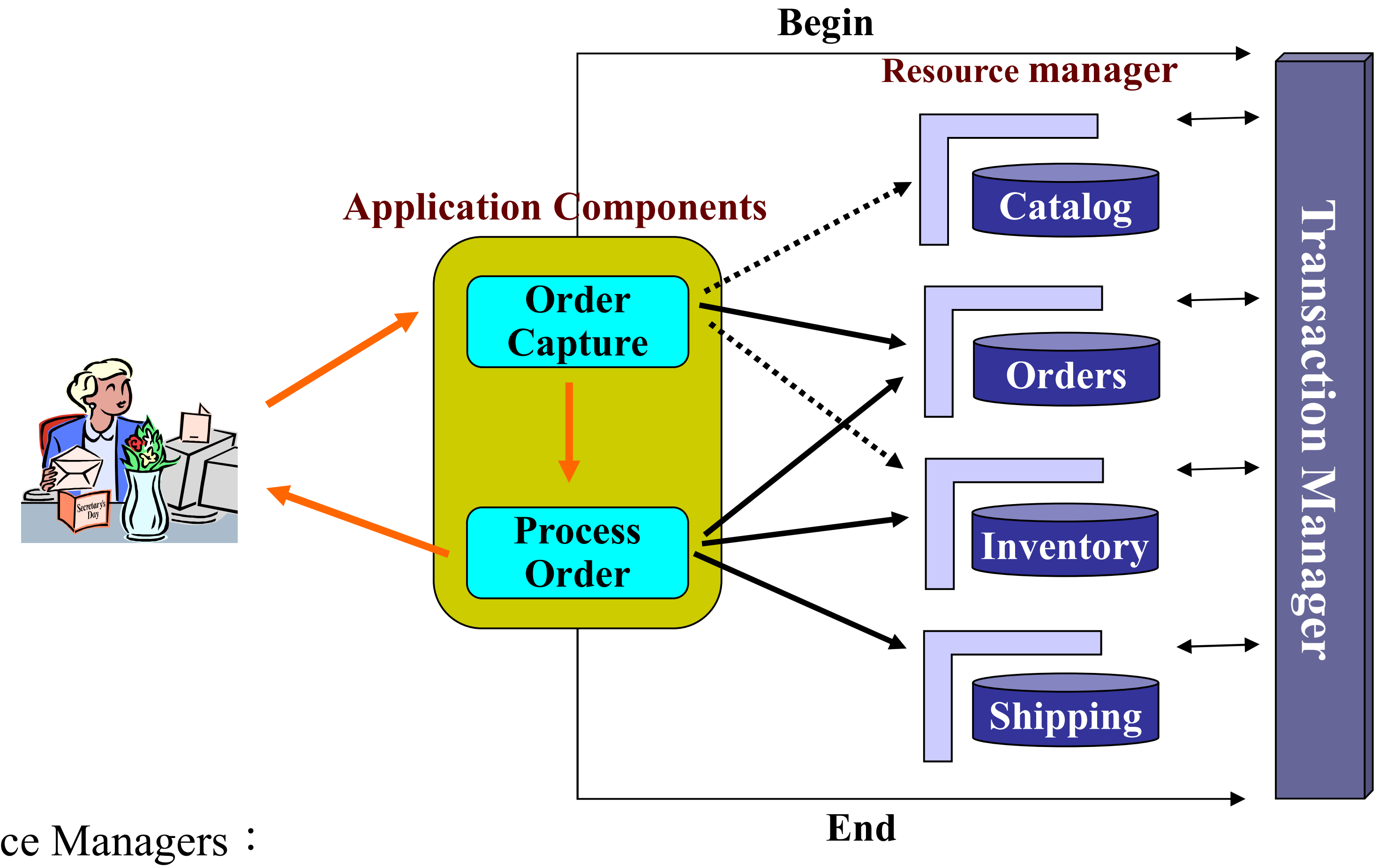
resources

··········→ read only

———→ read/write

But, lots of issues to be addressed, e.g., …

An Architecture for the Example



Resource Managers：

- manage persistent and stable data storage system
- participate in the two phase commit and recovery protocols with the transaction manager.

Transaction Manager

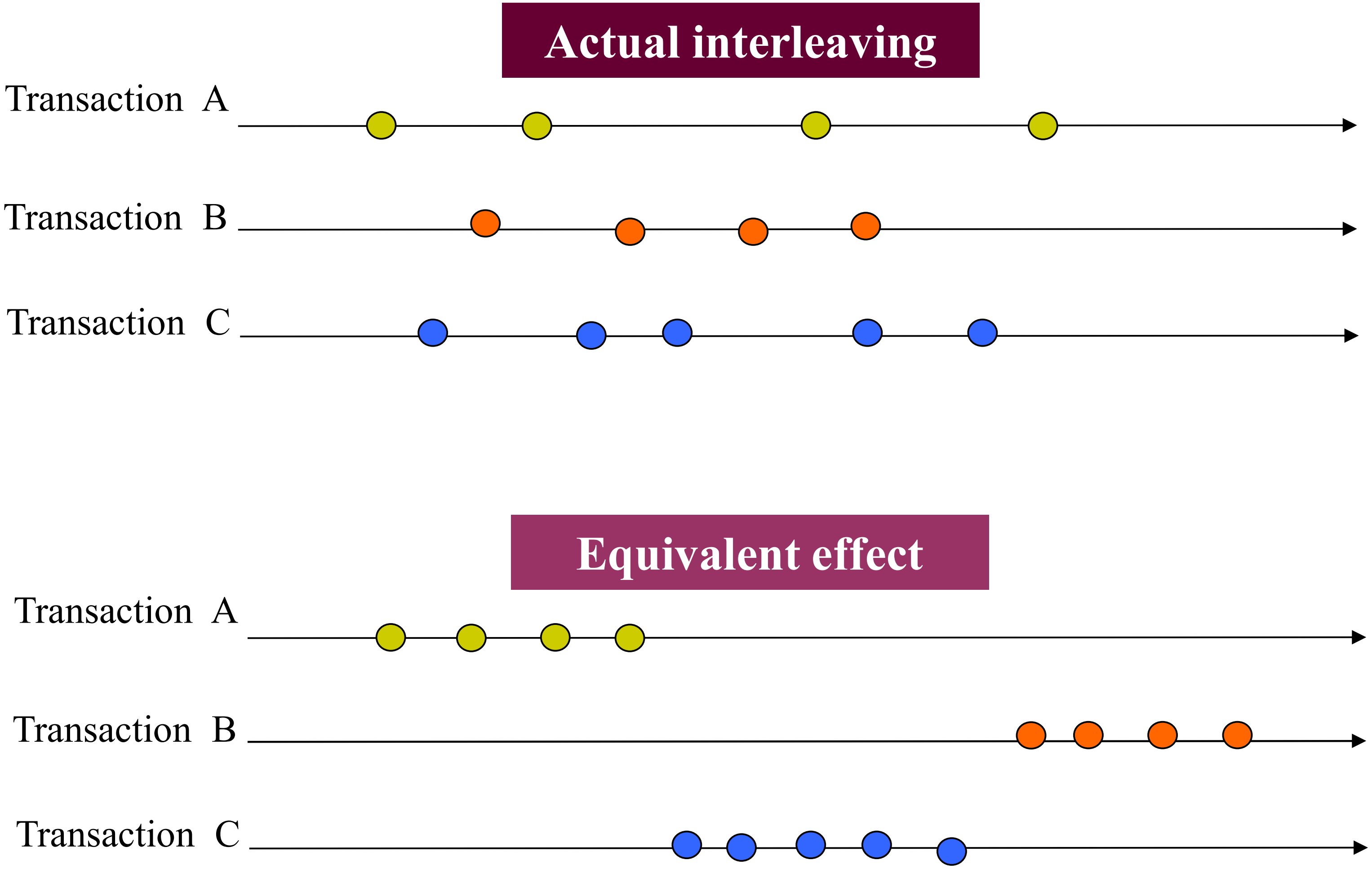
- Core component of a transaction processing system
- Main responsibilities:
  - Establish and maintain transaction context
  - Maintain association between a transaction and the participating resources
  - Concurrency control and maintain atomicity of transactions
  - Initiate and conduct two-phase commit and recovery protocol with the resource managers.
  - Make synchronization calls to the application components before beginning and after end of two-phase commit and recovery process

The ACID property

Properties that are used to guarantee the set of operations of a transaction to be done in such a way that either all of them complete successfully, or none of them has been executed (**all or nothing**), and once a transaction is complete (i.e., committed), its affect is permanent.

- **Atomicity**: a transaction is an atomic unit of processing and it is either performed entirely or not at all
- **Consistency**: a transaction’s correct execution must take the database from one correct state to another
- **Isolation**: each transaction should appear to execute independently of other concurrent transactions.
- **Durability**: the effects of a completed transaction should always be persistent.

Atomicity Implies Serializability



Note about the ACID property

- ACID property rules the RDBMS world where data objects are "monolithic" and replicas of them typically do not spread across wide area networks.
- Database systems that rely on ACID transactions are usually slower at read and write operations.
- For a high throughput system (e.g., Netflix, Facebook, or other big data applications) or for Internet-scale services (e.g., shopping cart+order+payment service), in which data (replicate or part of "data objects") are across several nodes or servers, it is impractical to adopt the "all-or-nothing" concept of transactions to operations of data objects in the services.

The design & implementation of very large distributed storage systems will be discussed in later sessions.

Transaction Life Histories

Successful	Aborted by client	Aborted by server
openTransaction	***openTransaction***	***openTransaction***
operation	operation	operation
operation	operation	operation
•	•	server aborts •
•	•	transaction → •
operation	operation	operation ERROR
closeTransaction	***abortTransaction***	reported to client

Challenges for Transaction Processing

- Typical problems resulting from bad concurrency control:
  - Lost update
  - Inconsistent retrievals
  - Dirty reads (which may lead to irrecoverability!)
  - Cascading aborts
  - Premature writes
- **Concurrency control** ensures concurrent transactions be executed so that the combined effect is the same as if they are executed sequentially in some order; i.e., to guarantee **serializability**.

Some Scenarios

Some Scenarios: Lost Updates

Transaction T:
balance = b.getBalance();
b.setBalance(balance + 20);
a.withdraw(balance - 20)
$a \xrightarrow{20} b$
Original value:
a=100, b=200, c=300
		Transaction U:
		balance = b.getBalance();
		b.setBalance(balance + 20);
		c.withdraw(balance -20)
		$c \xrightarrow{20} b$
		Original value:
		a=100, b=200, c=300	

balance = b.getBalance();	b= $200		
		balance = b.getBalance();	b=$200
		b.setBalance(balance + 20);	b=$220
b.setBalance(balance + 20);	b=$220		
a.withdraw(balance - 20)	a=$80		
		c.withdraw(balance - 20)	c=$280

final result: a=$80, b=$220, c=$280

correct result: a=$80, b=$240, c=$280

A Serializable Interleaving

Transaction T:
balance = b.getBalance();
b.setBalance(balance + 20);
a.withdraw(balance - 20)
$a \xrightarrow{20} b$
Original value:
a=100, b=200, c=300
	Transaction U:
	balance = b.getBalance();
	b.setBalance(balance + 20);
	c.withdraw(balance - 20)
	$c \xrightarrow{20} b$

balance = b.getBalance(); $200	
b.setBalance(balance + 20); b= $220	
	balance = b.getBalance(); $220
	b.setBalance(balance + 20); b=$240
a.withdraw(balance - 20) a=$80	
	c.withdraw(balance - 20) c=$320

Inconsistent Retrievals

Transaction V:
a.withdraw(100)
b.deposit(100)
$a \xrightarrow{100} b$
Original value:
a=100, b=200, c=300
	Transaction W:
	aBranch.branchTotal()

a.withdraw(100) \$100
b.deposit(100) \$300
	total = a.getBalance(); t=\$0
	total = total + b.getBalance(); t=\$200
	total = total + c.getBalance(); t=\$500
	(\$600)
	•
	•

A Serializable Interleaving

Transaction V:
a.withdraw(100)
b.deposit(100)
 	
Original value:
a=100, b=200, c=300
	Transaction W:
	aBranch.branchTotal()

a.withdraw(100) $100
b.deposit(100) $300
	total = a.getBalance();
	t=$100
	total = total + b.getBalance();
	t=$300
	total = total + c.getBalance();
	t=$600
	• • •

A Non-serializable Example

Transaction T:	Transaction U:
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read(i)

w.r.t. j, T should be after U

w.t.t. i, T should be prefer U

Dirty Read

Transaction T:
a.getBalance();
a.setBalance(balance + 10);
 	
Original value:
a=100, b=200, c=300
	Transaction U:
	a.getBalance();
	a.setBalance(balance + 20);

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

Read a temp value that is later aborted

Premature Write

Transaction T: a.setBalance(105) (before transaction) a=100	Transaction U: a.setBalance(110)

balance = a.getBalance();
a=$100
a.setBalance(balance + 5)
a=$105
Commit
	balance = a.getBalance();
	a=$100
	a.setBalance(110)
	a=$110
	Abort (restore a=100)

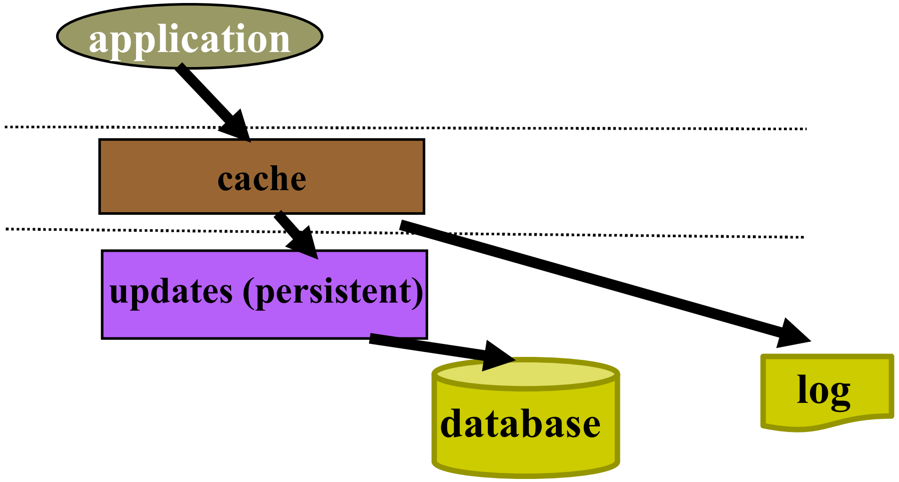
Some database implement abort transaction by restoring the before image of the transaction. So, if U is aborted, $100 may be restored, thus overwriting T's effect (**Premature write**).

Read/Write Conflict Rules

Operations of different transactions		Conflict	Reason
read	*read*	No	The effect of a pair of *read* operations does no depend on the order in which they are executed
read	*write*	Yes	The effect of a *read* and a *write* operations depends on the order of their executions
write	*write*	Yes	The effect of a pair of *write* operations depends on the order of their executions

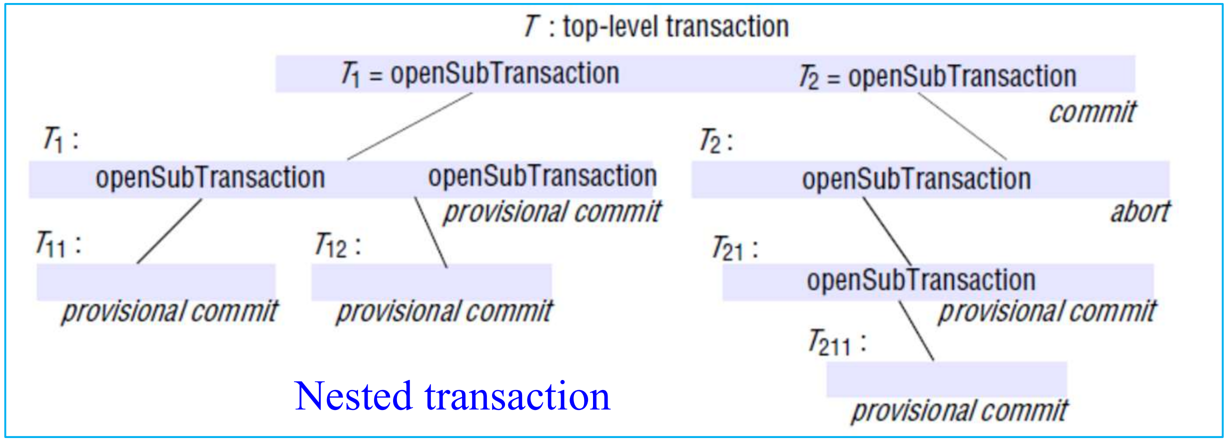
Structure of a Transactional System

- To avoid aborted transactions from causing any effect, each transaction must be given a tentative version of the data items to work on.
- The execution of transactions are strict if the service delays both Read and Write operations on a data item until all transactions that previously wrote the data item have either committed or aborted.



Nested Transactions

- Transactions can be nested (vs. flat), involving sub-transactions.
  - Sub-transactions may run concurrently
- Depending on applications, the parent (top-level) transaction may decide to commit even if some of its sub-transactions may fail.



Nested transaction

Source: Distributed Systems: Concepts and Design 5th ed., C. Coulouris et al., 5th ed., 2011.

Concurrency Control Mechanisms

- Locking
- Optimistic concurrency control
- Timestamps

Locks

- Lock data before using it
  - Set read lock before reading
  - Set write lock before writing
  - Wait if lock cannot be granted
  - Locks only granted if no conflicts
    - Read locks conflict with write locks
    - Write locks conflict with both read and write locks
    - (Read locks do not conflict among themselves)

		Lock requested	
		read	*write*
Lock granted	*none*	ok	ok
	read	ok	wait
	write	wait	wait

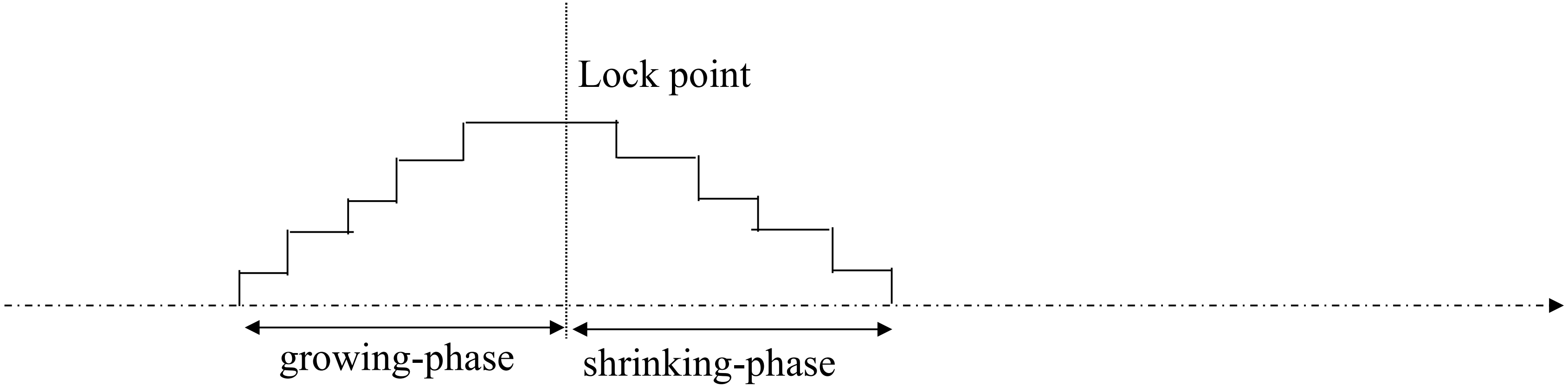
Original value:
a=100, b=200, c=300

Transaction T:
balance = b.getBalance();
b.setBalance(balance + 20);
a.withdraw(balance - 20)
$a \xrightarrow{20} b$
	Transaction U:
	balance = b.getBalance();
	b.setBalance(balance + 20);
	c.withdraw(balance -20)
	$c \xrightarrow{20} b$

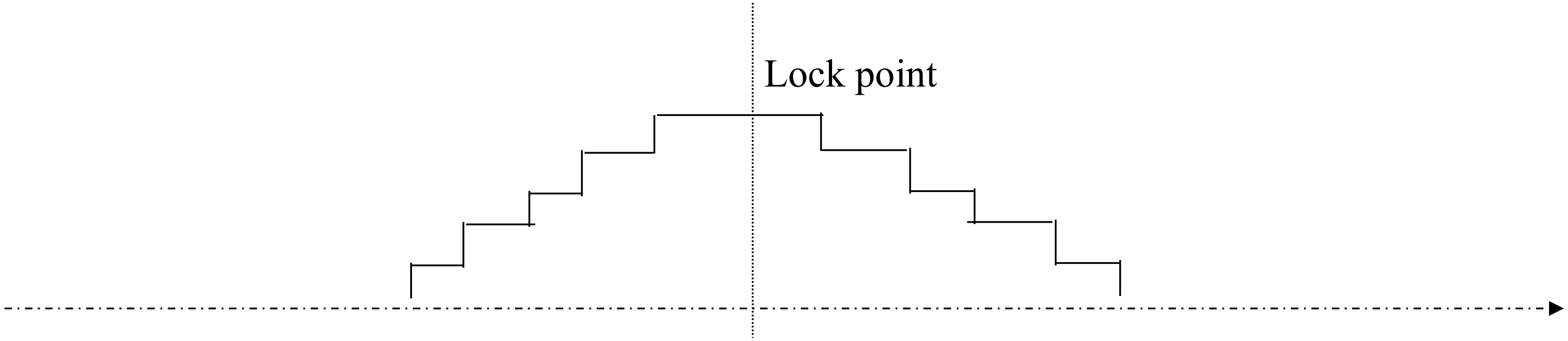
openTransaction
balance = b.getBalance();
b.setBalance(balance + 20);
a=100, b=220, c=300
a.withdraw(balance - 20)
a=80, b=220, c=300
closeTransaction
b locked
a locked
	openTransaction
	balance = b.getBalance();
	wait for locking b
	b.setBalance(balance + 20);
	a=80, b=240, c=300
	c.withdraw(balance - 20)
	a=80, b=240, c=280
	closeTransaction
	b locked
	c locked

Two-Phase Locking

To ensure serializability, **two-phase locking** is usually adopted:
A transaction is not allowed any new locks after it has released a lock.



Why does it guarantee serializability?

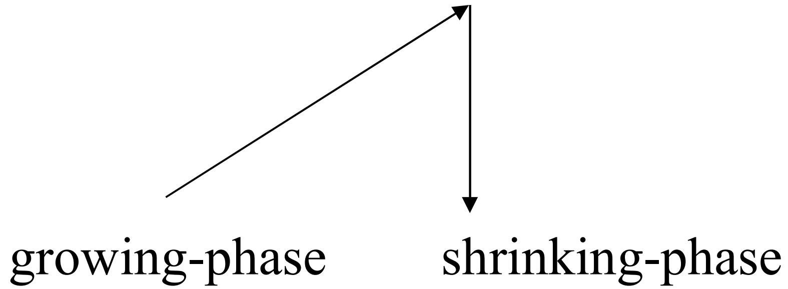


Dirty Reads

- Can two-phase locking prevent dirty reads and premature writes?

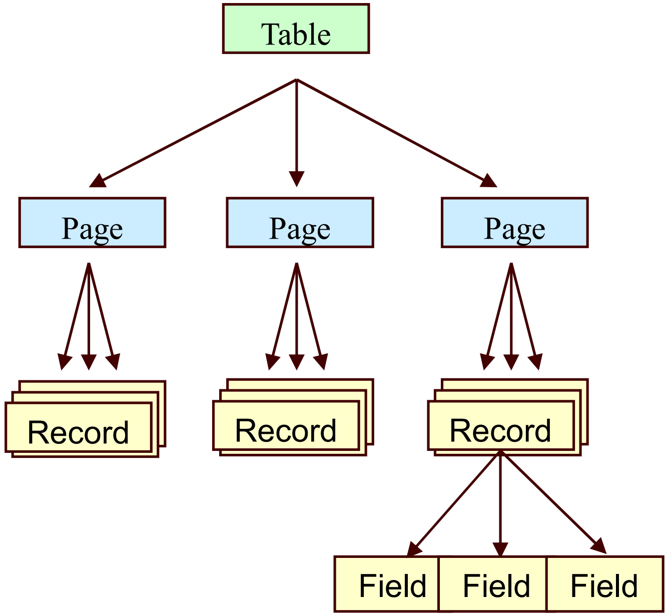
Strict Two-Phase Locking

- To prevent dirty reads and premature writes, we can impose strict two-phase locking for which any lock applied during the progress of a transaction are held until the transaction commits or aborts.



Lock Granularity

- What is locked?
  - Whole database
  - Whole table?
  - Page of data?
  - Individual record?
  - Field in a record?



- Trade-off between concurrency and ease of management
- Fine grain locks give less contention and better performance
  - Fine grain using lots of locks and are more expensive to manage

Lock Managers

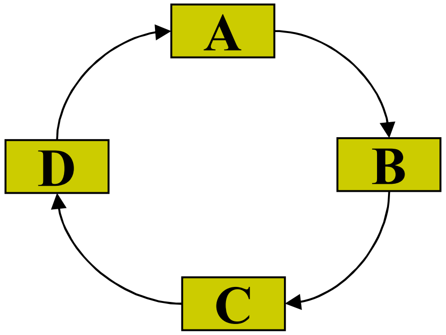
- Code that manages locking
  - Maintains a lock table
    - Keeps track of all locks in the database
    - Waiting requests and granted locks
  - Lock operations are atomic
    - Protected by low-level locks (mutex, spin)

	Locks granted	Locks requested
x	T1(read), T2(read)	T3(write)
y	T2(write)	T4(read), T1(read)
z	T1(read)	

Deadlock

The use of locks can lead to deadlock.

- What constitutes a deadlock?
  - Deadlock is a state in which each member of a group of transactions is *waiting* for some other member to release a lock.
- How to represent a deadlock?
  - A *wait-for graph* can be used to represent the waiting relationships between concurrent transactions at a server.
- Possible solutions:
  - Deadlock Prevention
  - Deadlock Detection
  - Timeout



Deadlock Prevention Techniques

- Lock (in one atomic step) all data items used by a transaction when it starts.
- Every transaction requests locks on data items in a predefined order.
- Each lock is given a limited period, after which it becomes vulnerable.

Deadlock Detection

- Deadlocks may be detected by finding cycles in the wait-for graph.
- Two design issues:
  - Frequency of checking the existence of a cycle
  - Choice of transactions to be aborted

Drawbacks of Locking

- Overhead of lock maintenance
- Locks affect performance
  - Concurrent transactions waiting for access to the same resource
  - holding locks until the end of a transaction (to avoid cascade aborts)
  - deadlock prevention
- Distributed systems can have interesting locking problems

In some applications, the likelihood of conflict is low.

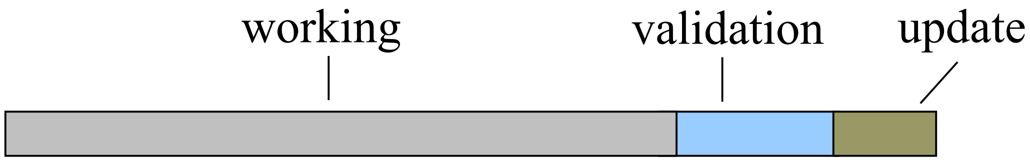
Optimistic Concurrency Control

- Observation
  - in most applications the likelihood of two clients' transactions accessing the same data is low.
- Approach
  - transactions are allowed to proceed as though there were no possibility of conflict until they attempt to close the transactions.
  - Upon closing transaction, checks for conflict; if so, some transactions aborted.

Optimistic Concurrency Control

- Each transaction has the following three phases:
  - **Working phase:** each transaction has a private workspace to work on. Read data are given by the most recently committed version (and so there is no dirty reads.)
  - **Validation phase:** when a transaction is to close, it is validated to see if any conflict has occurred. If it does not pass the validation, then either it is aborted or some other conflicting transaction will be aborted.
  - **Update phase:** all of the changes recorded in its tentative workspace are now made permanent if the transaction pass the validation phase.

A committed transaction's life cycle

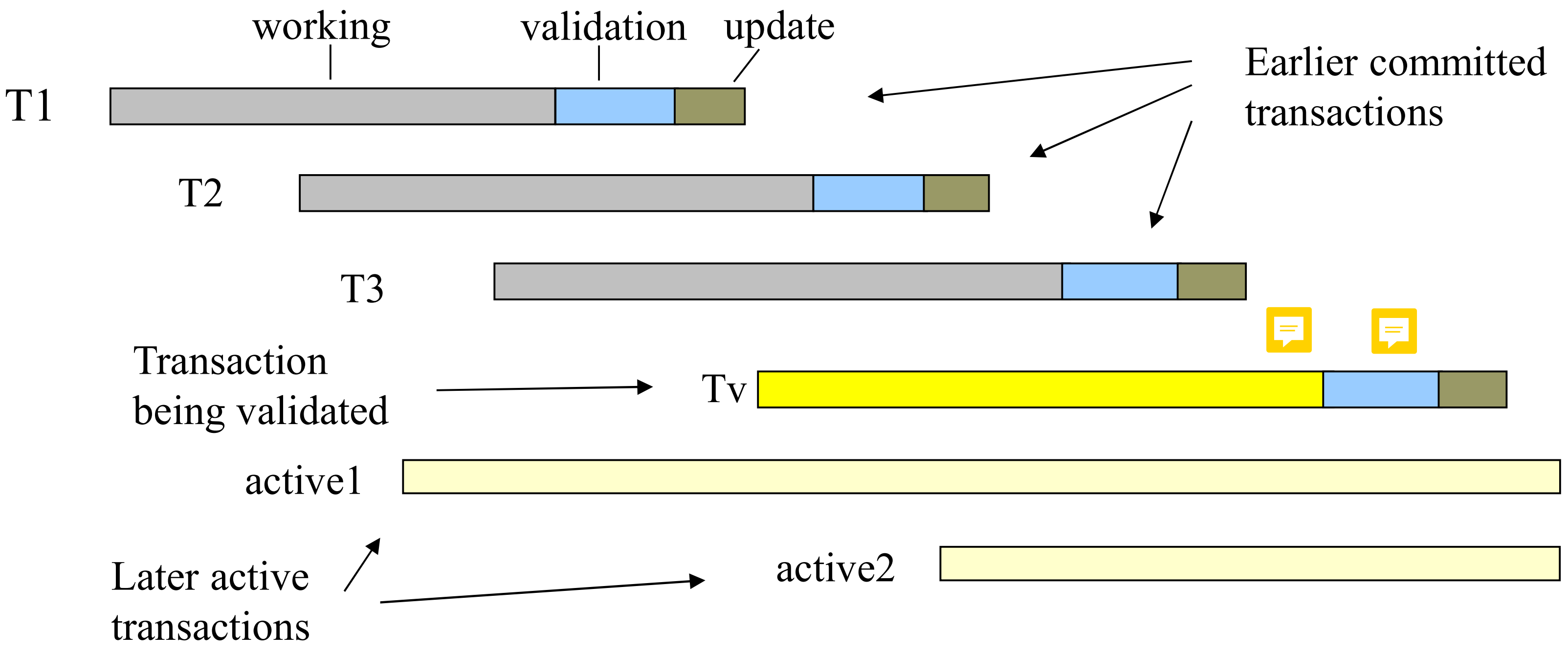


Validation of Transactions

- operated in a mutually exclusive style
  - to reduce the (write-write) conflict situations because the validation and update phases of a transaction are generally short
- assigned a monotonically increasing number to each transaction entering the validation phase.
  - transaction numbers then define the transactions' ordering
    - a transaction always finishes its working phase after all transactions with lower numbers

Two Types of Validation

- Backward:
  - the validated transaction's read set must not overlap with any write sets of earlier overlapping transactions.
    - Tv cannot read what T2 and T3 have written
- Forward:
  - the validated transaction's write set must not overlap with any read sets (which could vary dynamically!) of active overlapping transactions.
    - Tv cannot write what Active1 and Active2 have read


working
validation
update
T1
T2
T3
Earlier committed transactions
Transaction being validated
Tv
active1
active2
Later active transactions

Forward vs. Backward Validation

Backward

- abort the transaction being validated
- must retain the write sets of committed transactions that may conflict with active transactions

Forward

- two options for abortion:
  - abort all the conflicting active transactions
  - abort the transaction being validated
- has to allow for new transactions during the validation process?

- Forward validation has more flexibility to resolve the conflicts, and don’t need to store old read/write sets.
- However, they both may cause starvation.

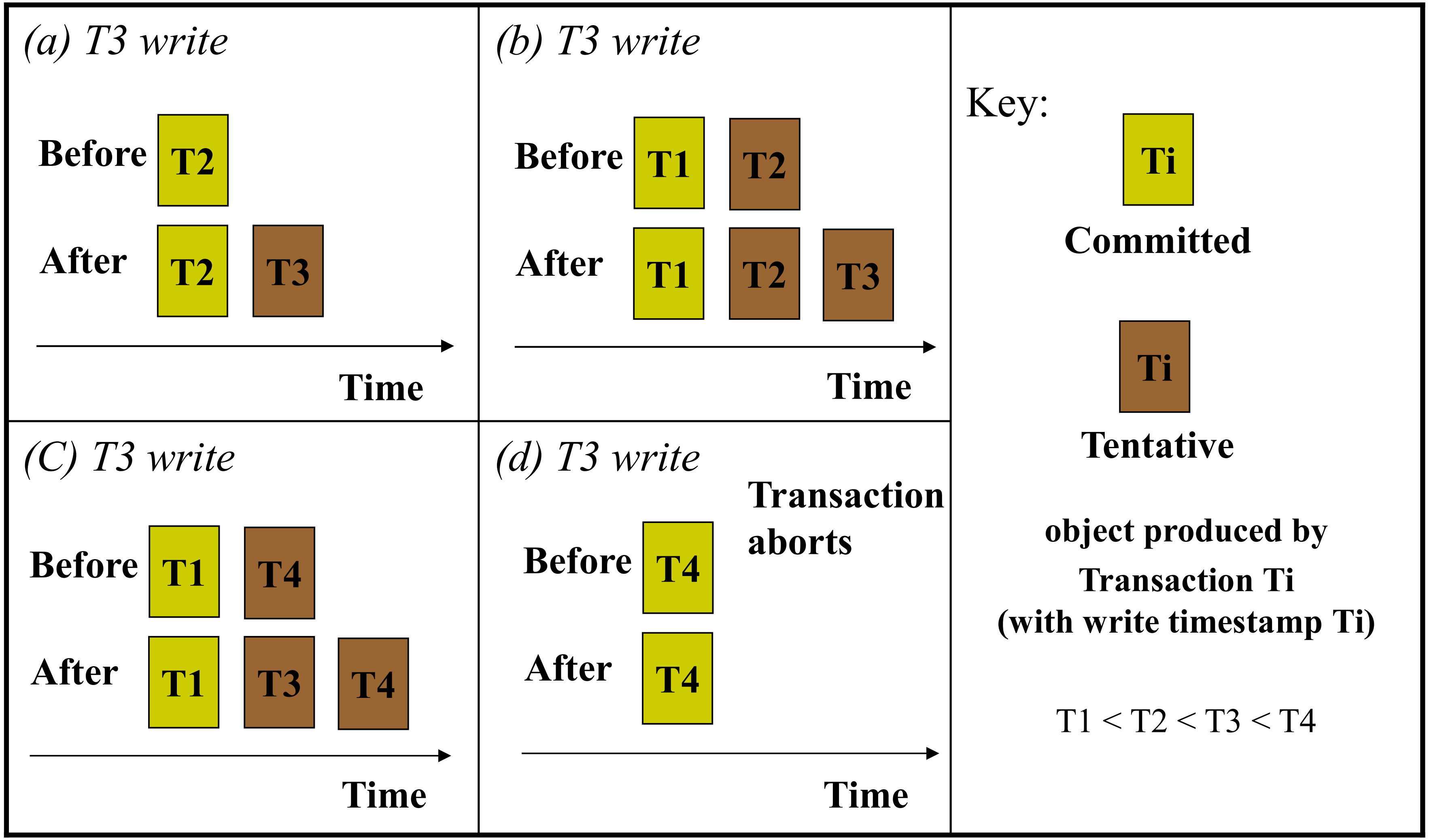
Timestamp Ordering

- Each transaction is assigned a unique timestamp when it starts; the timestamps are used to order transactions.
- Each operation of a transaction is validated when it is carried out based on the following rule:
  - A transaction's request to write a data item is valid only if that item's last read and written was by earlier transactions.
  - A transaction's request to read a data item is valid only if that item's last written was by an earlier transaction.
- If an operation cannot be validated, the transaction is aborted immediately.

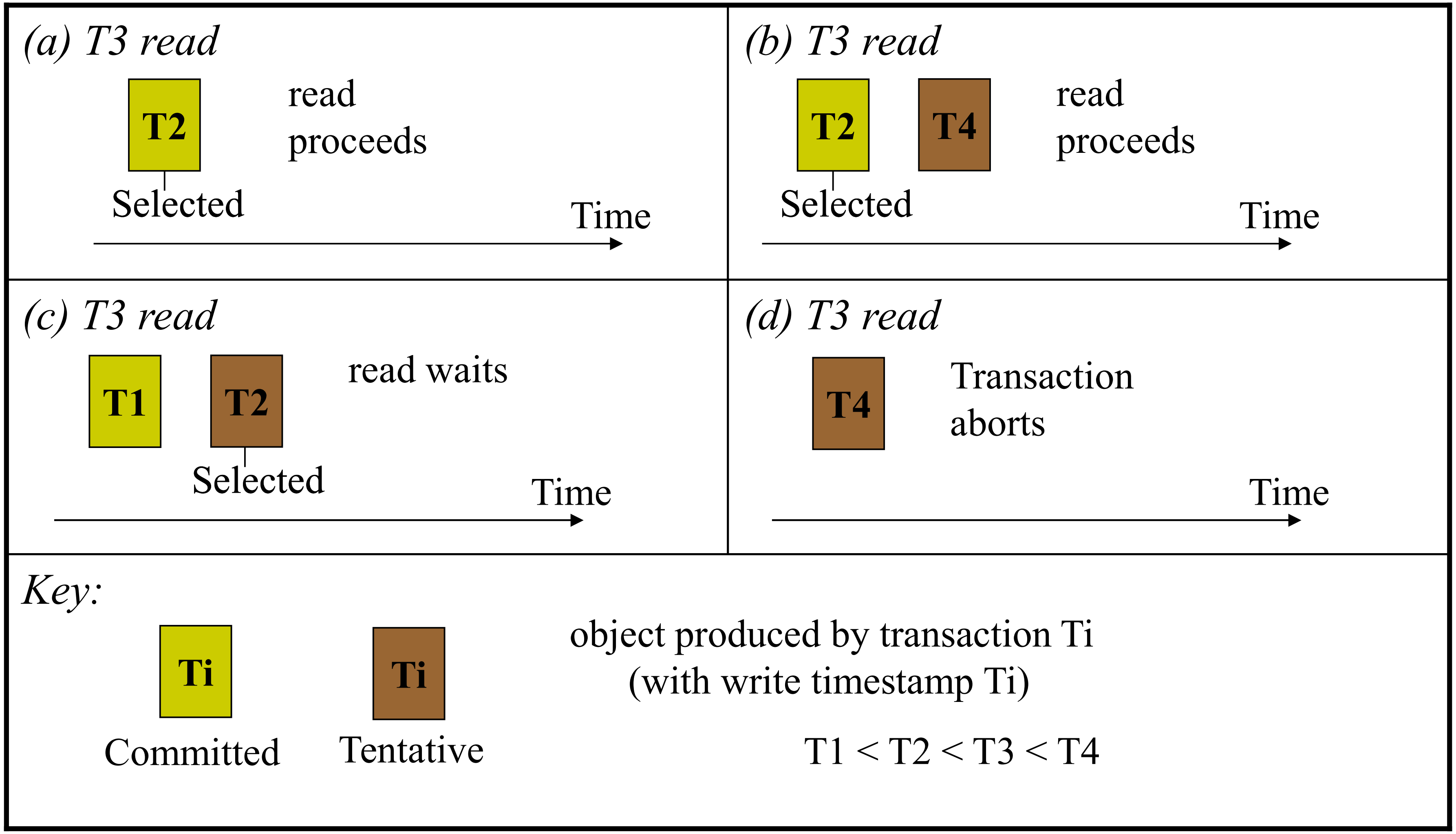
Read/Write Rules

Rule	Tc	Ti	
1.	*write*	*read*	Tc must not *write* an object that has been *read* by any Ti where Ti > Tc , this requires that Tc ≥ the maximum read timestamp of the object
2.	*write*	*write*	Tc must not *write* an object that has been *written* by any Ti where Ti > Tc , this requires that Tc > write timestamp of the committed object.
3.	*read*	*write*	Tc must not *read* an object that has been *written* by any Ti where Ti > Tc , this requires that Tc > write timestamp of the committed object.

Writes and Timestamp Ordering



Reads and Timestamp Ordering



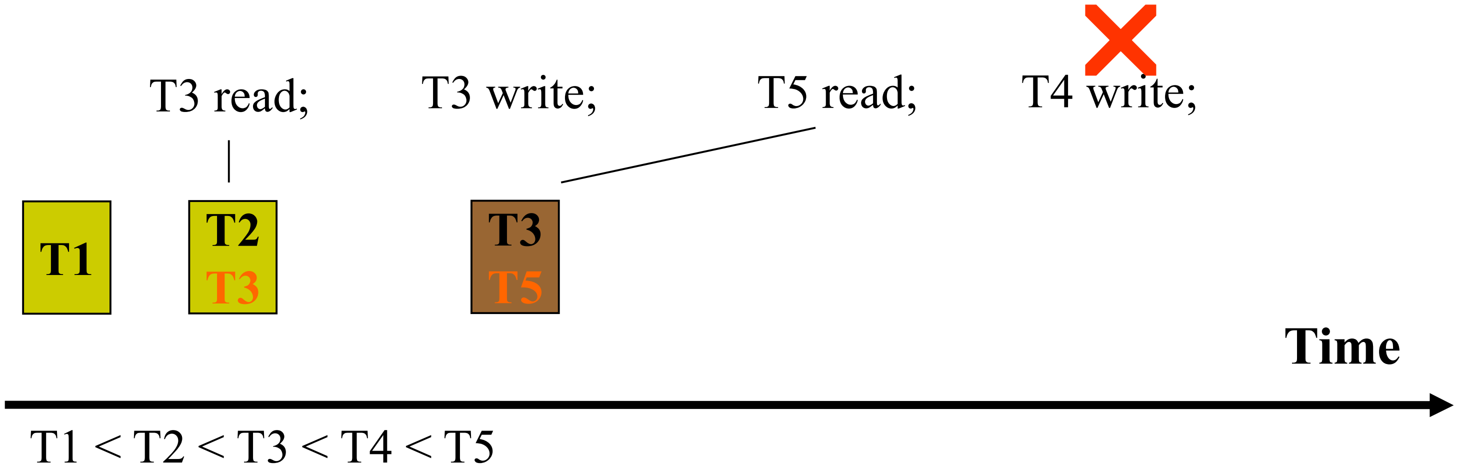
Relaxing the Write Rule

- In dealing with write/write conflicts, if a write is too late, it can be ignored instead of aborting the transaction, because if it had arrived in time its effects would have been overwritten anyway.
- However, if another transaction has read the data item, the transaction with the late write fails due to the read timestamp on the item.

Multi-version Timestamp Ordering

- The server keeps old committed versions as well as tentative versions in its list of versions of data items.
- With the list, *Read* operations that arrive too late need not be rejected.
- A *Read* operation of a transaction is directed to the version with the largest write timestamp less than the transaction timestamp.
- A transaction *Tc* must not write objects that have been read by any *Ti* where *Ti* > *Tc*.

A Late Write Invalidating a Read



T3 read;

T3 write;

T5 read;

T4 write;

T1

T2
T3

T3
T5

Time

T1 < T2 < T3 < T4 < T5

Key:

Ti
Tk

Committed

Ti
Tk

Tentative

Object produced by transaction Ti (with write timestamp Ti and **read** timestamp **Tk**)

Comparison: Locking vs. Timestamps

	Locking	**Timestamps**
"Attitude"	pessimistic	optimistic
Order Decision	dynamic	static
Benefiting Transactions	More writes than reads	Read dominant
Conflict Resolution	Wait, may abort later	Abort immediately

Comparison of the Three Schemes

- Locking
  - Lock maintenance is sometimes an overhead
  - May cause deadlock.
  - To avoid cascading aborts, locks cannot be released until the end of the transaction. So concurrency may be decreased.
- Optimistic concurrency control
  - More efficient when conflicts are rare
    - No deadlock if validation is done sequentially
- Timestamps
  - Deadlock is not possible.
  - Prone to restart.

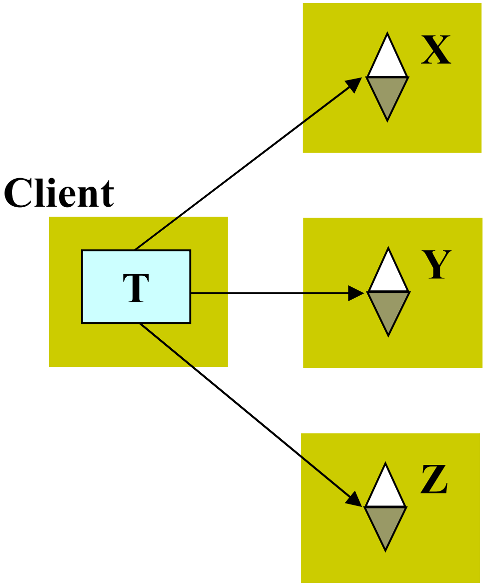
Distributed Transactions

- Any transaction whose activities involve multiple servers is a distributed transaction.
  - Two types of distributed transactions: flat and nested.
  - One of the servers involved in a distributed transaction must act as a coordinator to take care of commit or abort request.
- **Two-phase commit** protocol is commonly used to guarantee atomic commitment.

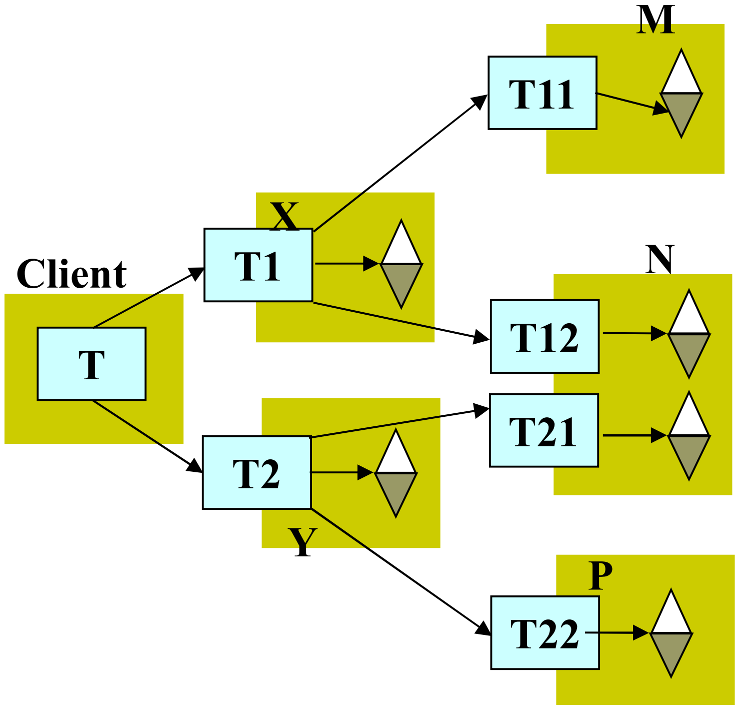
Structures of Distributed Transactions

- ❑ A flat transaction accesses servers' objects *sequentially*.
- ❑ The subtransactions of a nested transaction can run *in parallel* (*concurrently*).

(a) Flat transaction

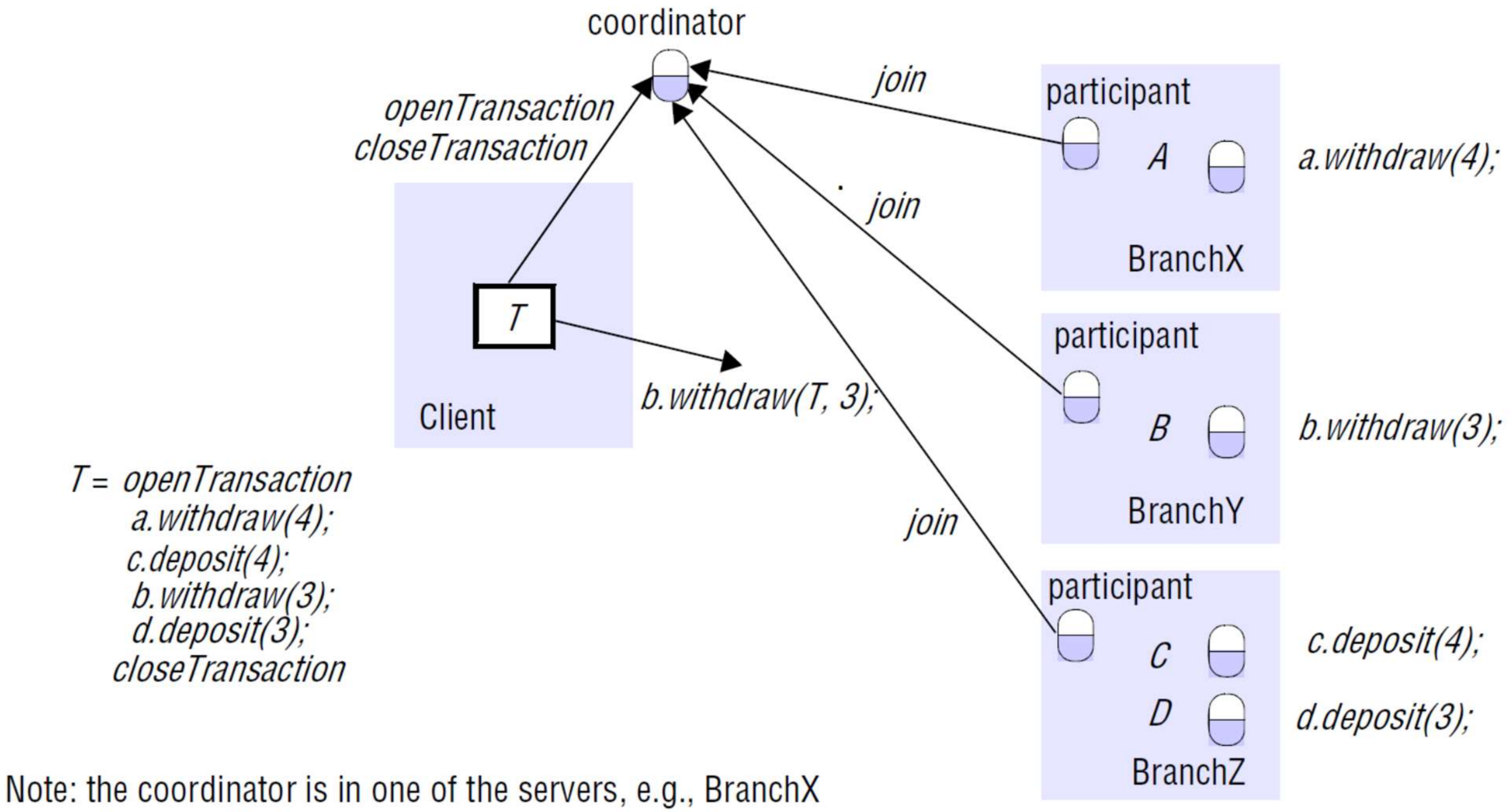


(b) Nested transactions



Example: A distributed banking transaction

A coordinator is selected to take care of the transaction, i.e., to **commit** or **abort**.



Source: Distributed Systems: Concepts and Design 5th ed., C. Coulouris et al., 5th ed., 2011.

Atomic Commitment

- When a distributed (flat) transaction comes to an end, either all or none of its operations (all-or-nothing) are carried out.
- Due to atomicity, if one part of a transaction is aborted, then the whole transaction must also be aborted.
- Depending on the applications, some nested transaction may allow part of its sub-transactions to abort, while others to commit.

Two-phase Commit Protocol

- The **two-phase commit protocol (2PC)** is designed to allow any server to abort its part of a transaction.
  - In the first phase (voting), each server votes for the transaction to be committed or aborted.
  - In the second phase (decision), every server carries out the joint decision.
- Why not one-phase commit protocol?

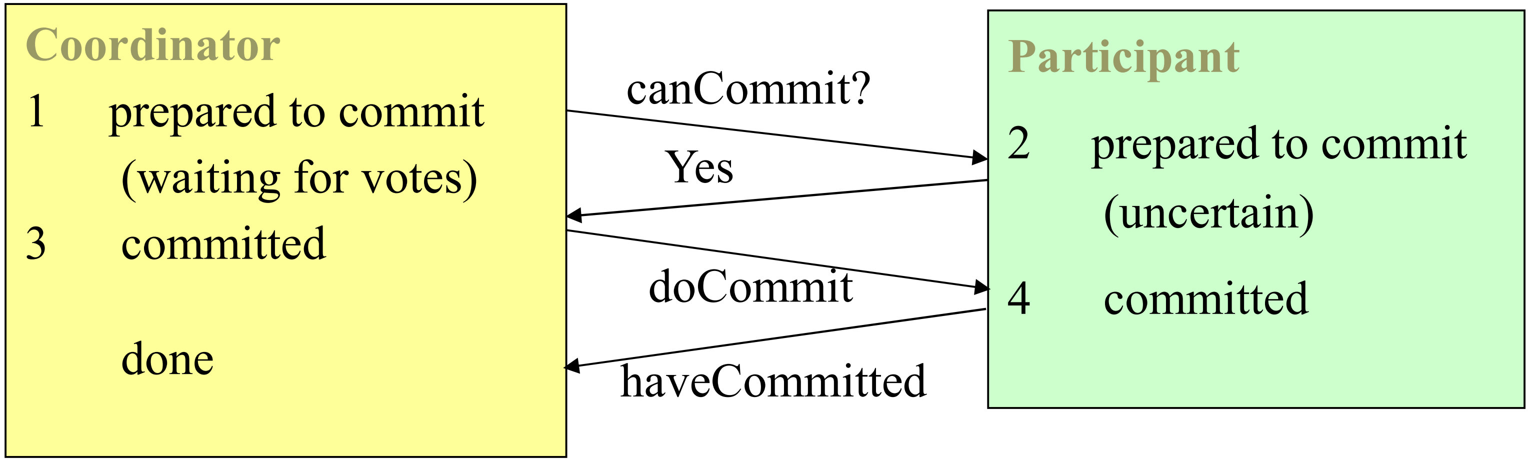


Illustration: A Normal Case

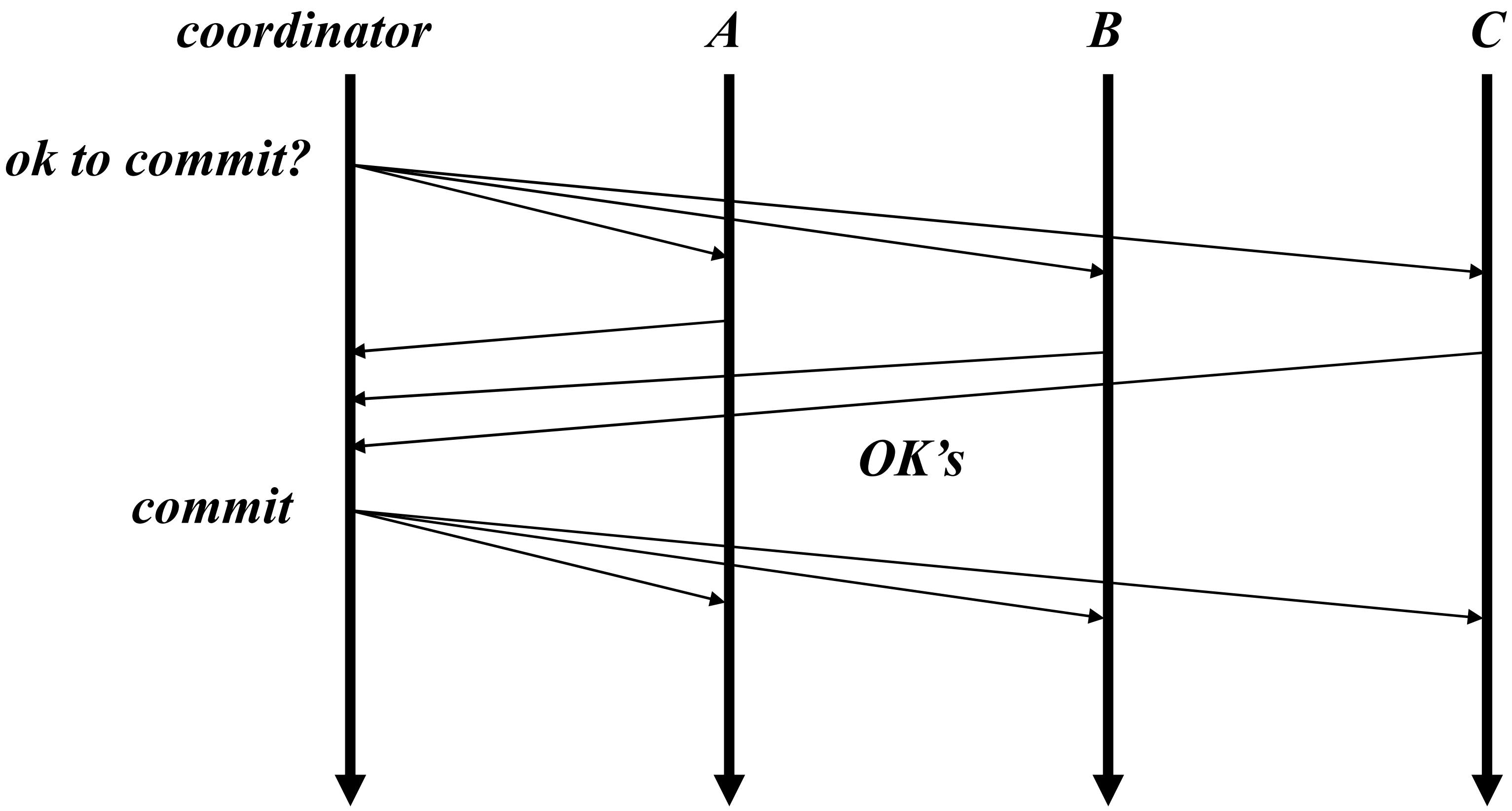


Illustration: Failures before Decision

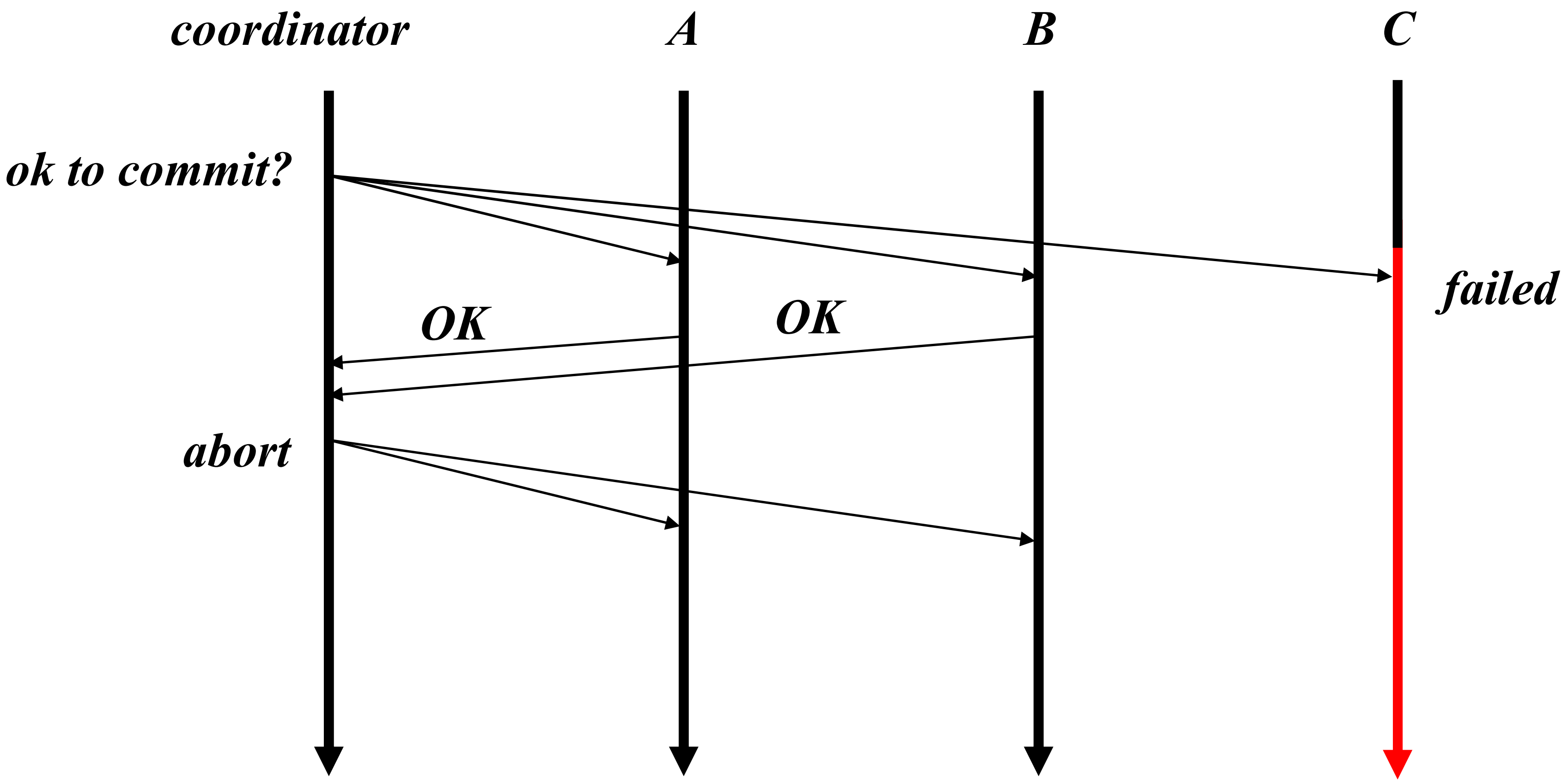
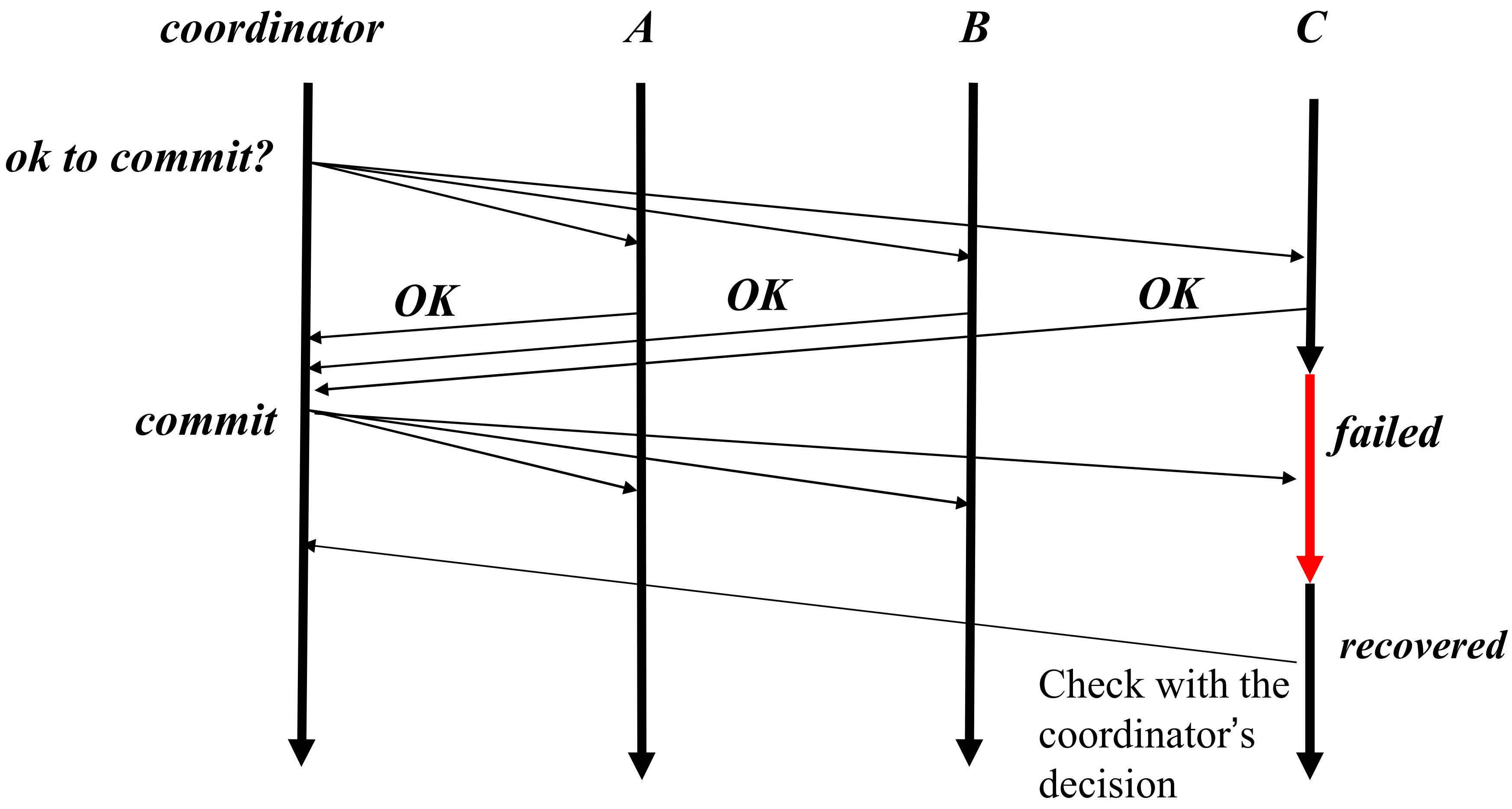
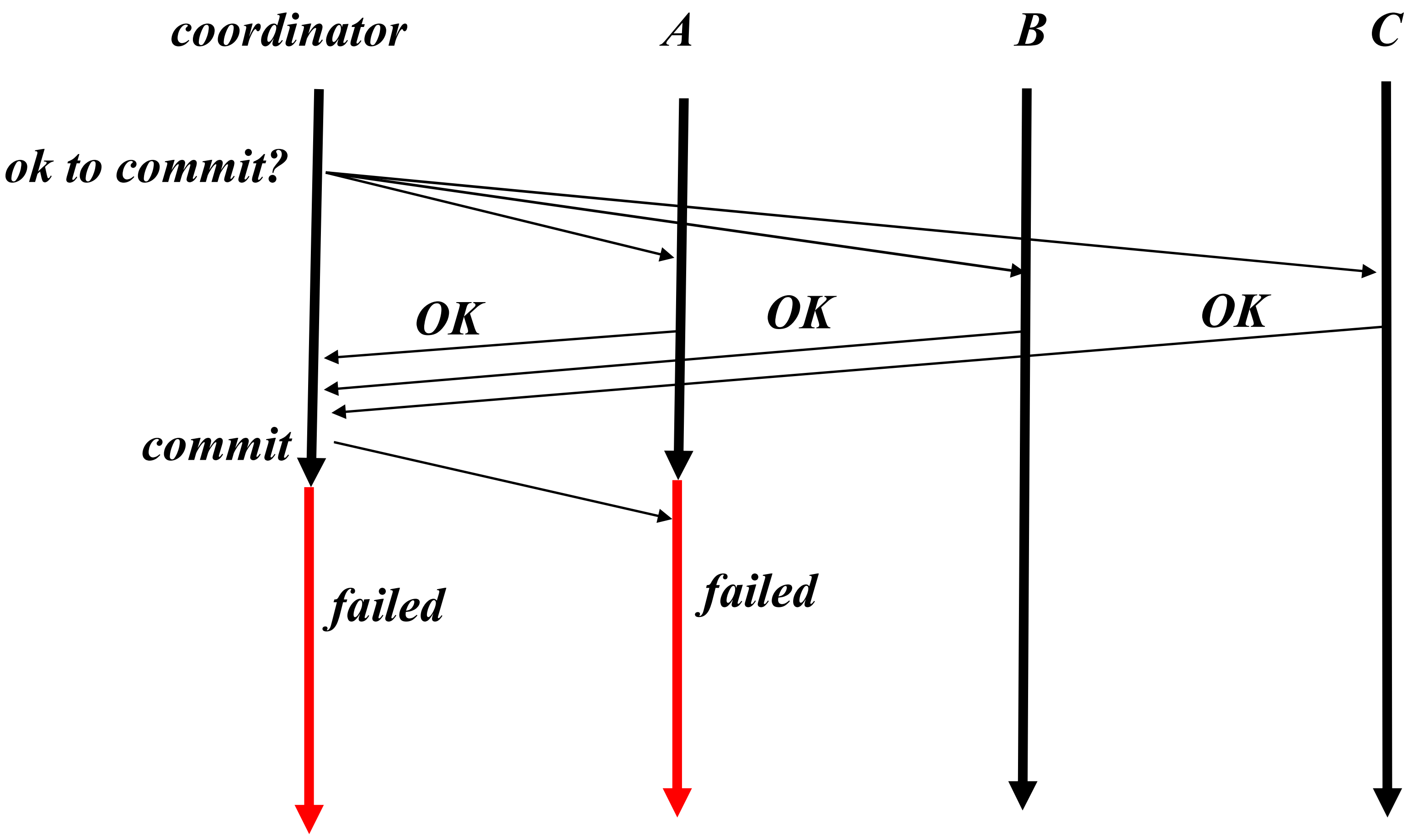


Illustration: Failures after Decision



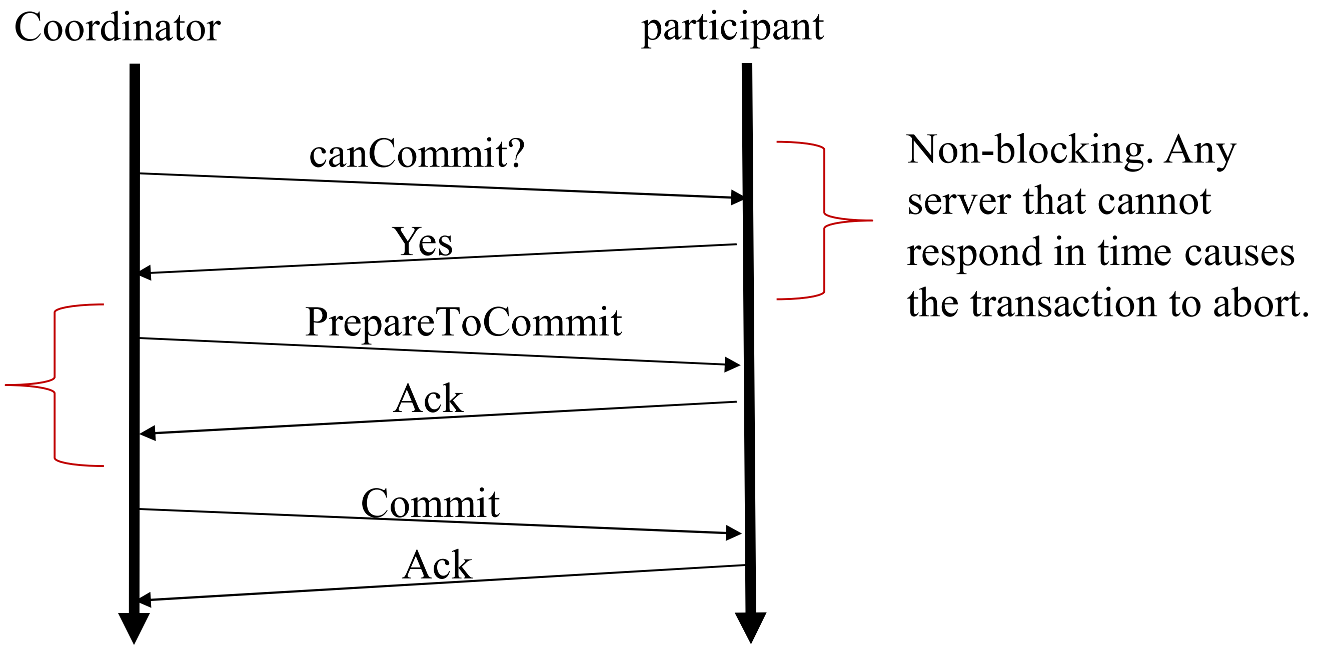
2PC is Blocking



B, C can’t commit or abort, as they don’t know what *A* responded or what the coordinator’s decision was.
Ps. Coordinator is usually one of the participants.

Three-Phase Commit Protocol

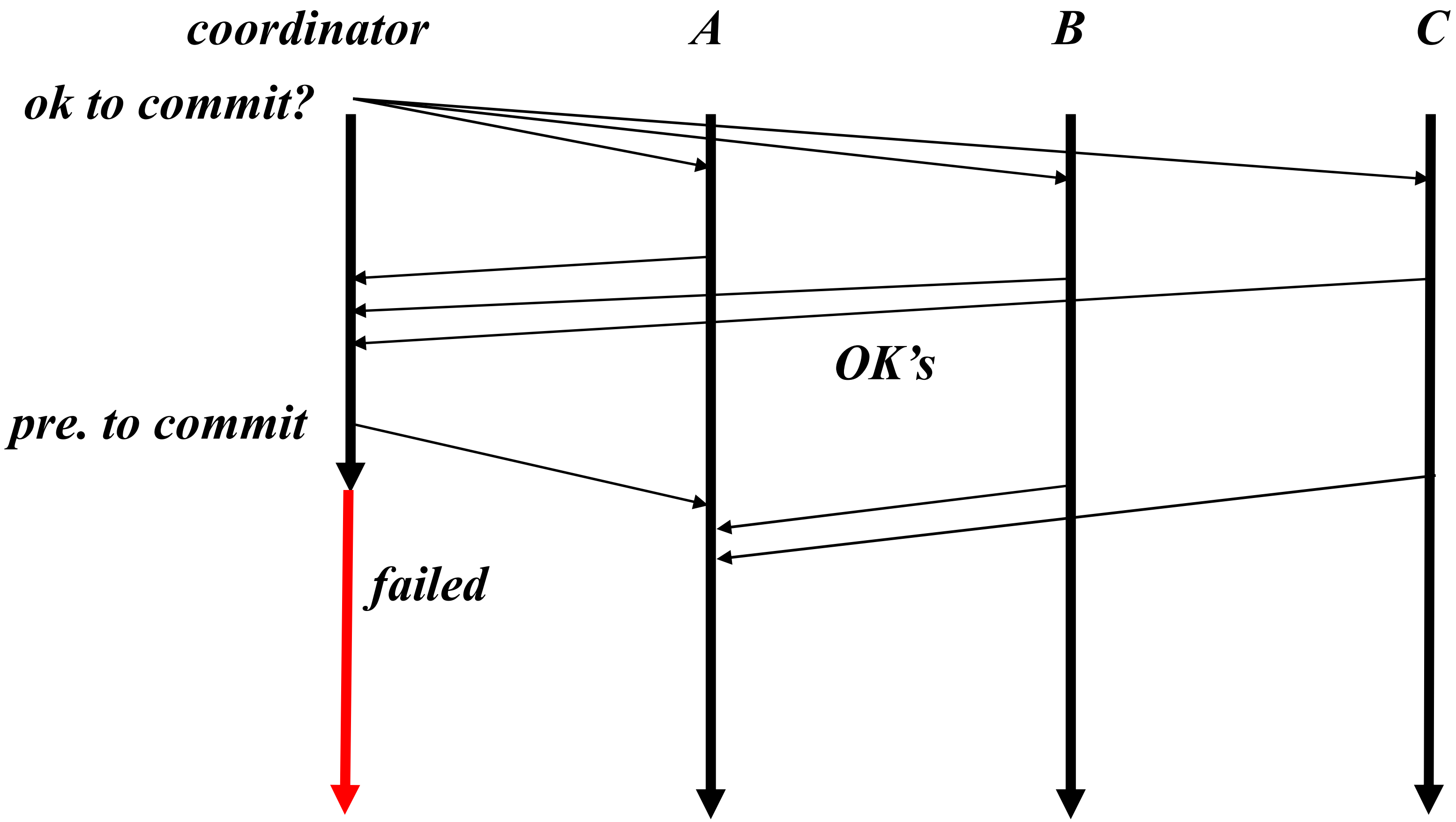
- The **three-phase commit protocol (3PC)** is designed to eliminate the blocking issues in 2PC:
  - In the 1st phase (voting), each server votes for the transaction to be committed or aborted. Unable to respond is considered a failure.
  - In the 2nd phase (pre-commit), the coordinator authorizes the servers to prepare commit.
  - In the 3rd phase (commit), the coordinator asks the servers to commit



Non-blocking. Any server that cannot respond in time causes the transaction to abort.

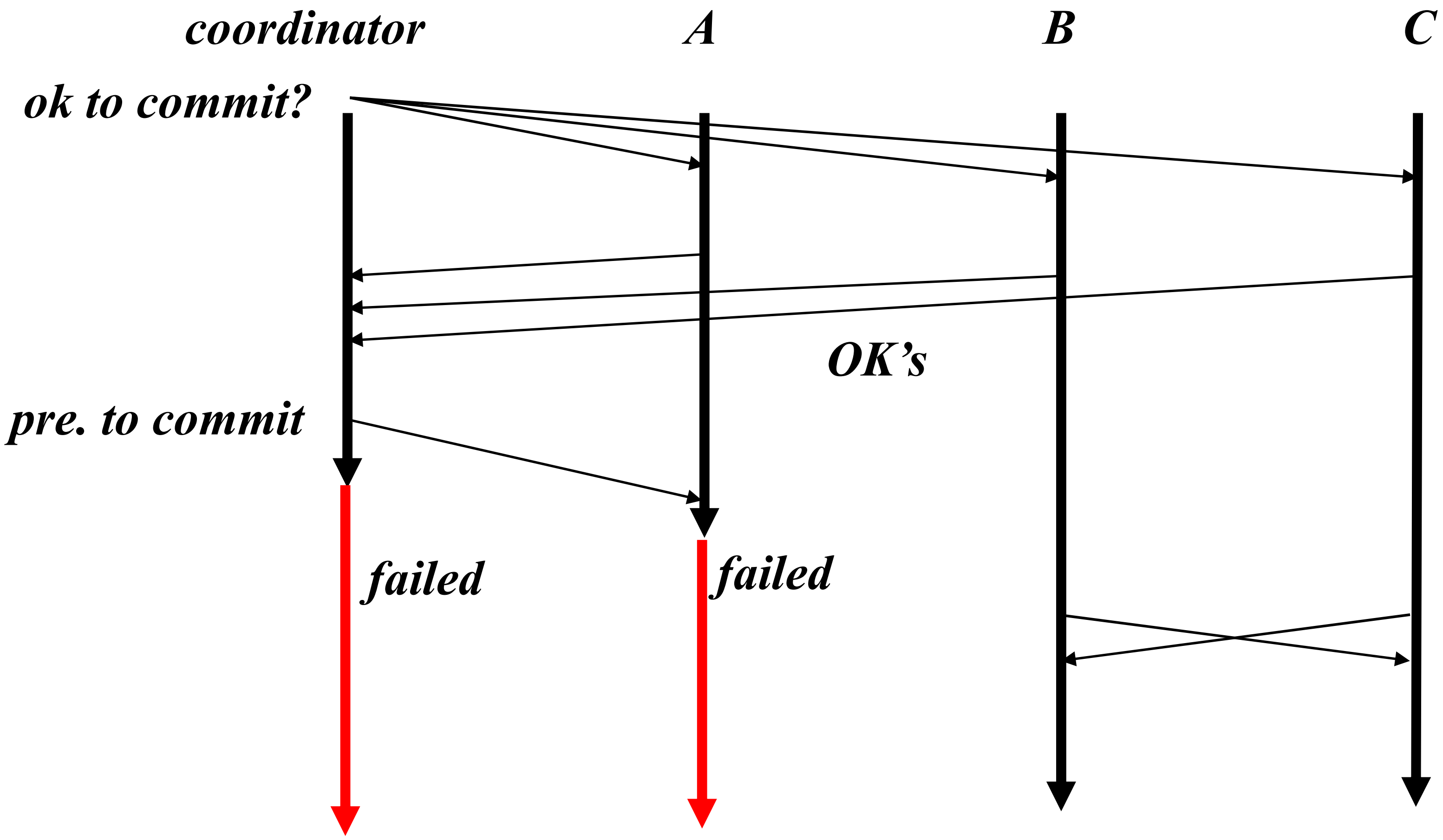
Non-blocking. Any server that does not receive PrepareToCommit from the coordinator can check with other participants if they have received, and commit/abort accordingly.

Illustration of 3PC



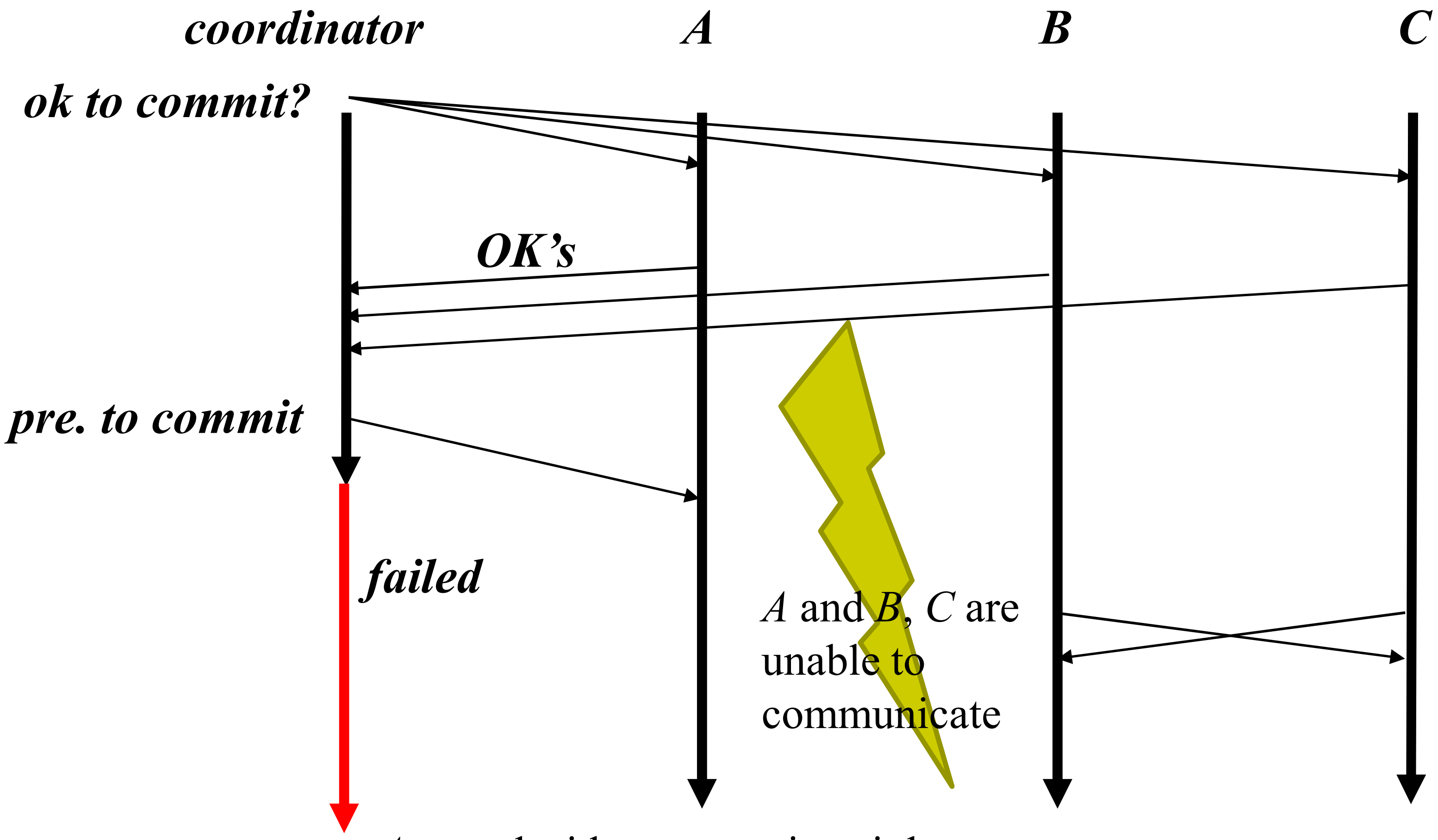
By checking with *A* (or a new elected coordinator), *B* and *C* know that all participants can commit in phase 1, and so can decide to commit.

Illustration of 3PC (2)



Since none of *B* and *C* have received PrepareToCommit, they can abort the transaction.

3PC can’t tolerate network partition

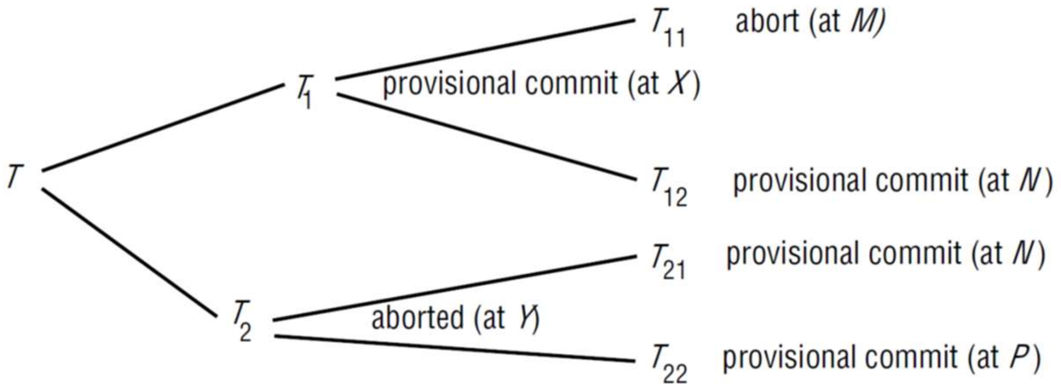


A may decide to commit as it has received PrepareToCommit, and believes that other participants are able to commit when they recover.

B, *C* may decide to abort due to unable to find a participant that has received PrepareToCommit.

Atomic Commitment in Nested Transactions

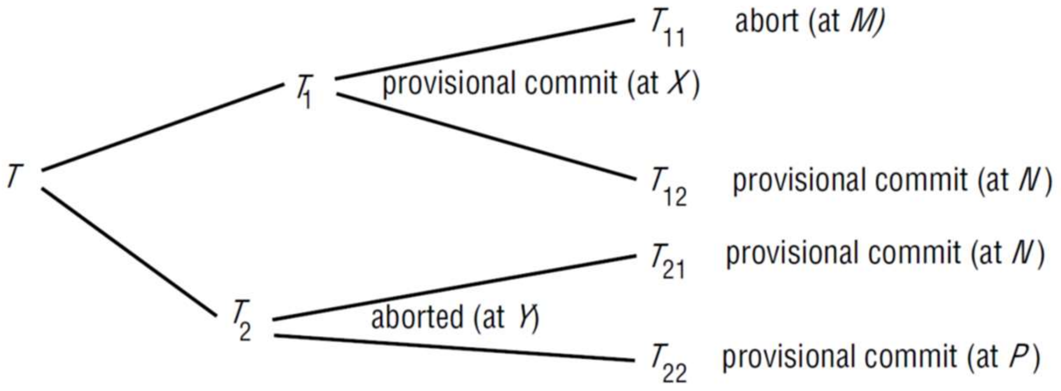
- When a subtransaction completes, it makes an independent decision either to commit *provisionally* or to abort.
- A parent transaction may commit even if one of its child transactions has aborted (depending on the semantics of the transaction)
- If a parent transaction aborts, then its subtransactions will be forced to abort.
- Subtransactions will not carry out a real commitment unless the entire nested transaction decides to commit.


T_{11} abort (at M)
T_1 provisional commit (at X)
T
T_{12} provisional commit (at N)
T_{21} provisional commit (at N)
T_2 aborted (at Y)
T_{22} provisional commit (at P)

Source: Distributed Systems: Concepts and Design 5th ed., C. Coulouris et al., 5th ed., 2011.

Two-Phase Commit in Nested Transactions

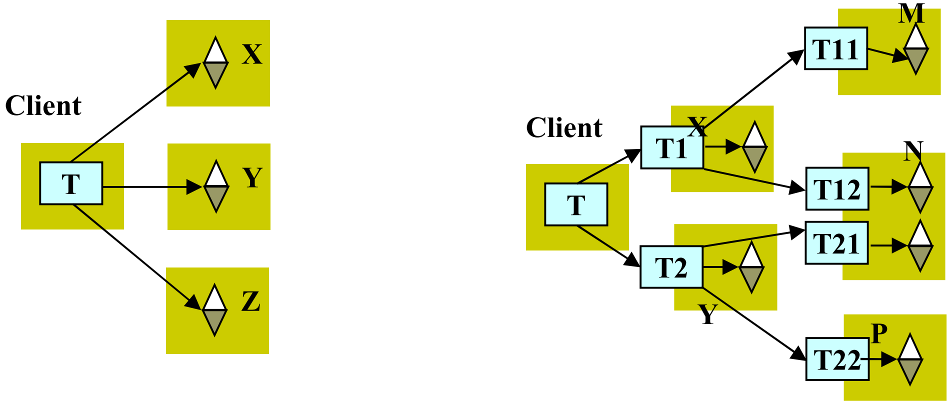
- When a subtransaction provisionally commits, it reports its status and the status of its descendants to its parent.
- When a subtransaction aborts, it just reports abort to its parent.
- Eventually, the top-level transaction receives a list of all subtransactions (except the descendants of an aborted transaction) in the tree, together with the status of each.


T
T_1
provisional commit (at X)
T_{11}
abort (at M)
T_{12}
provisional commit (at N)
T_2
aborted (at Y)
T_{21}
provisional commit (at N)
T_{22}
provisional commit (at P)

Source: Distributed Systems: Concepts and Design 5th ed., C. Coulouris et al., 5th ed., 2011.

Two-Phase Commit in Nested Transactions (2)

- The top-level coordinator sends *canCommit?* to all sub-coordinators in the provisional commit list.
- When a server receives a *canCommit?*
  - If it has provisionally committed substractions
    - prepares those without aborted ancestors for commitment,
    - aborts those with aborted ancestors, and
    - sends a *Yes* vote to the coordinator.
  - Otherwise (it must have failed), sends a *No* vote.



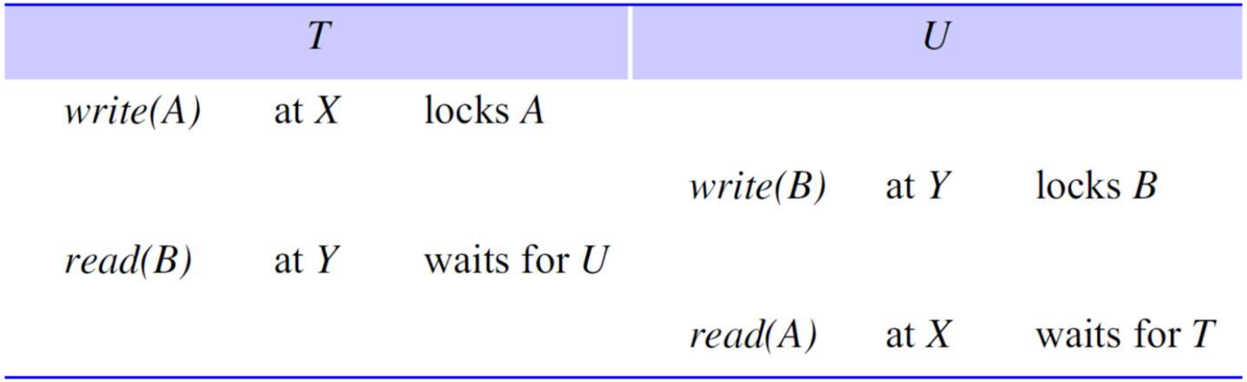
Note: coordination can become hierarchical if the structure of transaction is hierarchical; similarly, it could also be flat.

Concurrency Control in Distributed Transactions

- Each server applies concurrency control to its own objects.
- Every pair of transactions are serializable in the same order at all servers.

Locking in Distributed Transactions

- Locking in Distributed Transactions
  - Each server maintains locks for its own data items.
  - Locks cannot be released until the transaction has been committed or aborted at all servers.
  - *Distributed deadlocks* might occur if different servers impose different orderings on transactions.
    - A simple solution is to use a global deadlock detector.
    - A more sophisticated approach is to adopt a distributed solution.

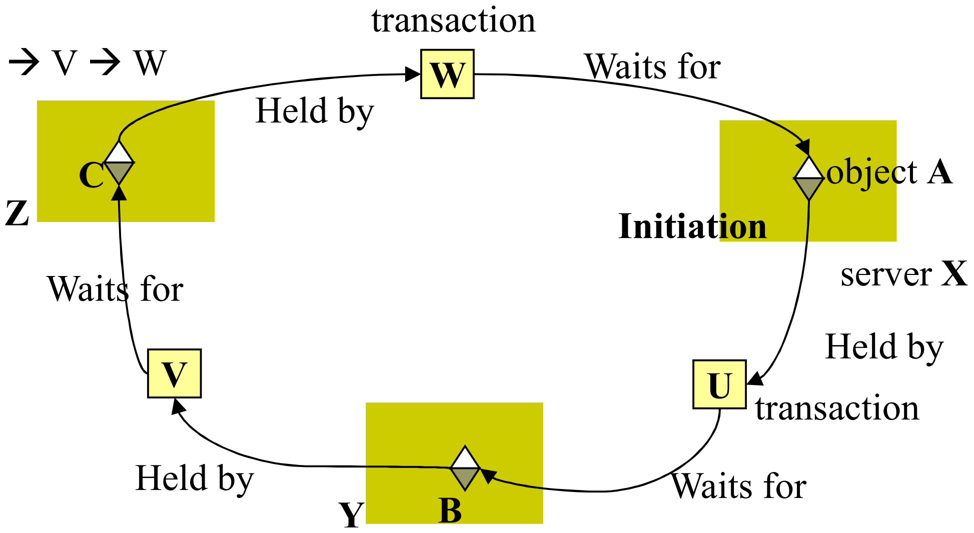


T			*U*		
write(*A*)	at *X*	locks *A*			
			write(*B*)	at *Y*	locks *B*
read(*B*)	at *Y*	waits for *U*			
			read(*A*)	at *X*	waits for *T*

Distributed Deadlocks

- A cycle in the global wait-for graph (but not in any single local one) represents a ***distributed deadlock***.
- A deadlock that is detected but is not really a deadlock is called a ***phantom deadlock***.
- Two-phase locking prevents phantom deadlocks; autonomous aborts may cause phantom deadlocks.
- A simple deadlock detection algorithm uses a technique called **edge chasing** to detect a deadlock. When a deadlock is detected, a transaction in the deadlock is aborted to break the deadlock.

W → U → V → W

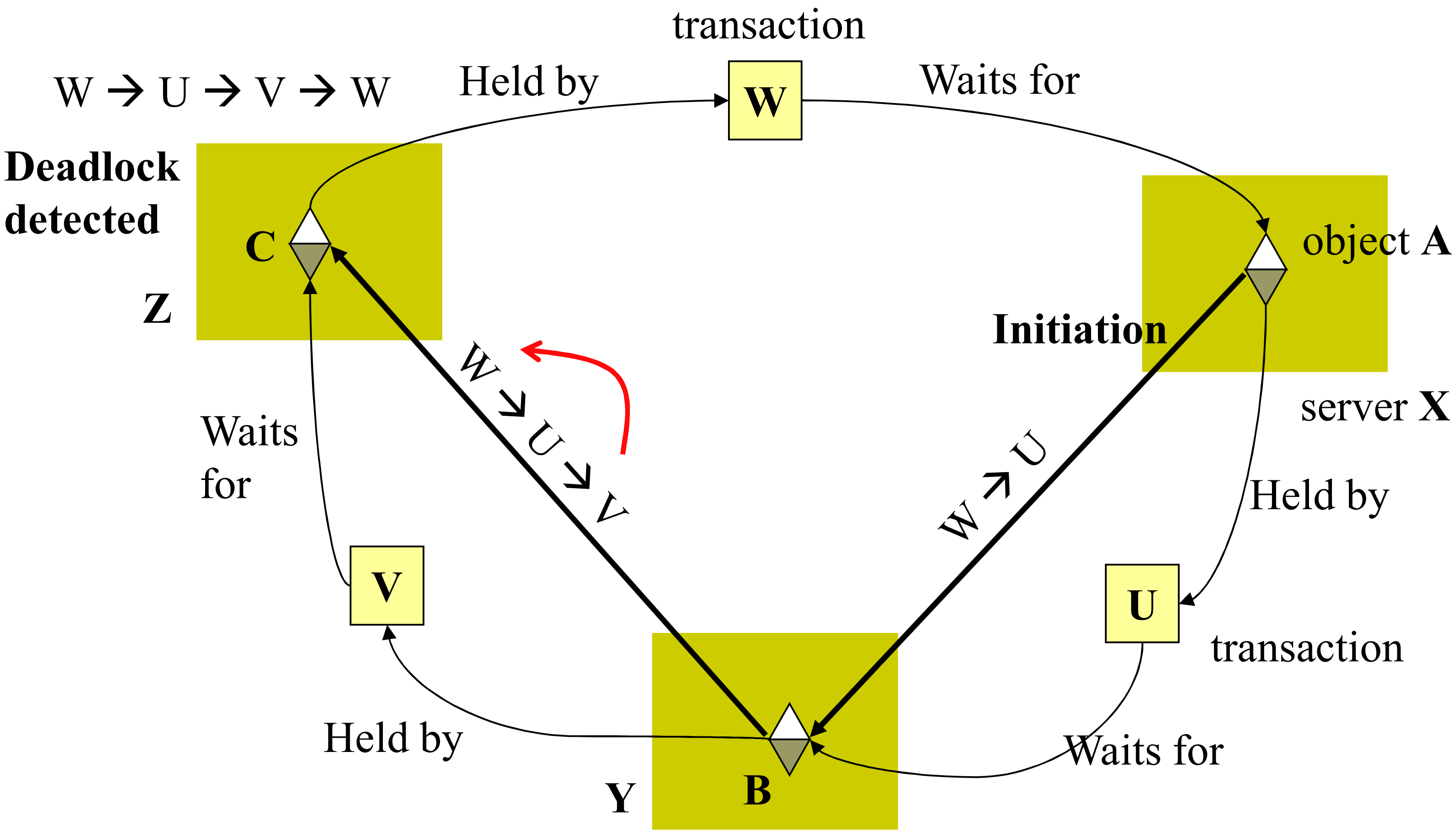


Edge Chasing

- Initiation: when a server notes that a transaction T starts waiting for another transaction U, which is waiting to access an object at another server, it sends a *probe* containing $\langle T\rightarrow U\rangle$ to the server of the object at which transaction U is blocked.
- Detection: receive probes and decide whether deadlock has occurred and whether to forward the probes.

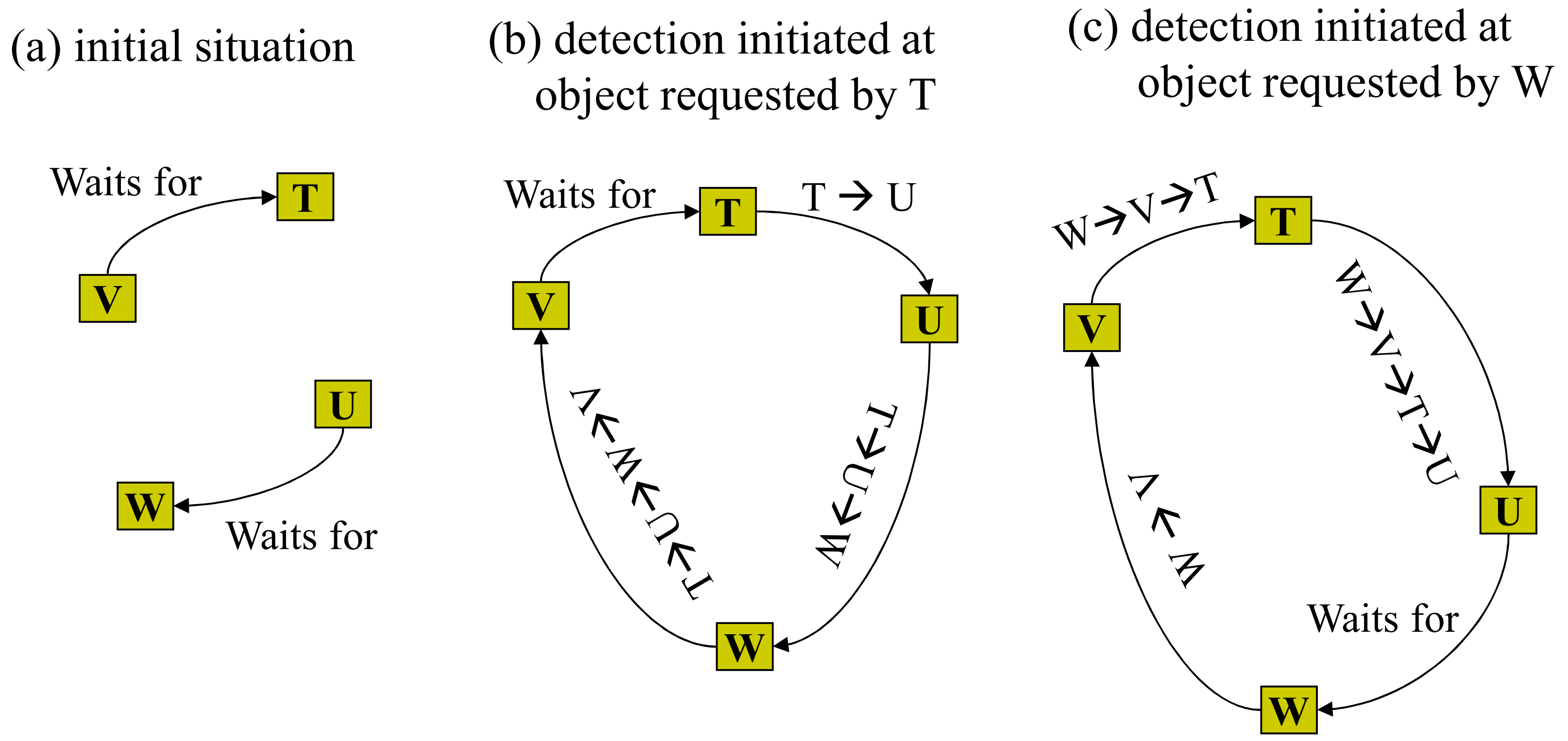
  When a server receives a probe $\langle T\rightarrow U\rangle$ and finds the transaction that U is waiting for, say V, is waiting for another object elsewhere, a probe $\langle T\rightarrow U\rightarrow V\rangle$ is forwarded.
- Resolution: select a transaction in the cycle to abort

Probes for Detecting Deadlocks



Source: G. Coulouris et al., *Distributed Systems: Concepts and Design, Third Edition.*

Independently Initiated Probes



(a) initial situation

(b) detection initiated at object requested by T

(c) detection initiated at object requested by W

Unnecessary abortion may be executed if not carefully coordinated!

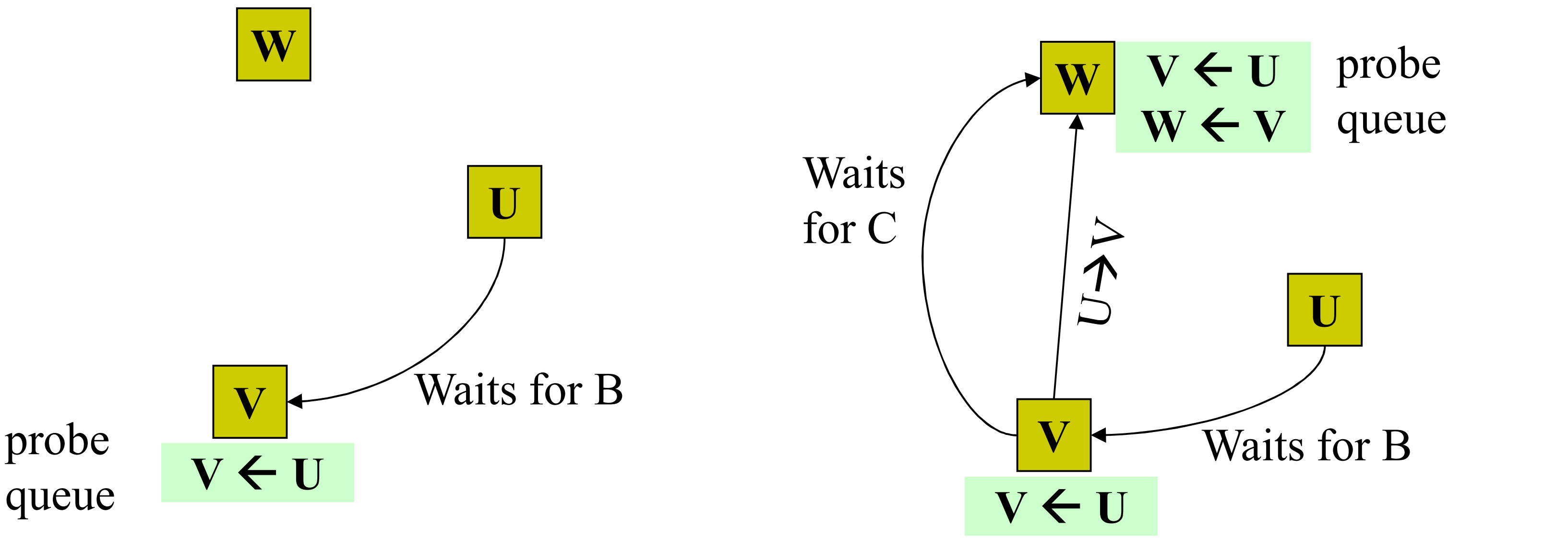
Solutions? Total order of transactions.

Can this be used to reduce number of probe messages?

Probes Traveling Downhill

(a) V stores probe when U starts waiting

(b) Probe is forwarded when V starts waiting



Need to be careful about passing on probes to new holders and to discard probes that refer to transactions that have been committed or aborted:

- If relevant probes are discarded, undetected deadlocks may occur
- If outdated probes are retained, false deadlocks may be detected.

Timestamp Ordering

- A *globally unique* transaction timestamp is issued by the coordinator.
  - How to issue a unique global timestamp?
- Conflicts are resolved as each operation is performed.
- If the resolution of a conflict requires a transaction to be aborted, the coordinator will be informed.

Optimistic Concurrency Control

- If only one transaction may perform validation at a time (per server site), then concurrent validation may cause *commitment deadlocks; parallel validation* does not have the problem.

Transaction T	Transaction U
$Read(A)$ at X $Write(A)$ $Read(B)$ at Y $Write(B)$	$Read(B)$ at Y $Write(B)$ $Read(A)$ at X $Write(A)$

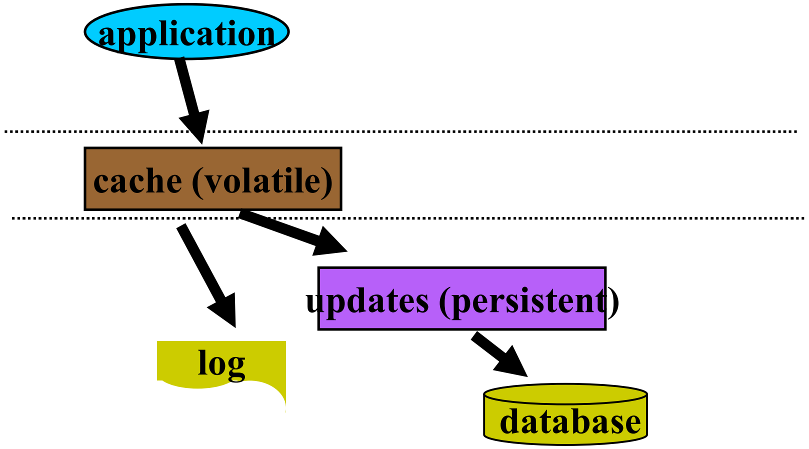
- A parallel validation checks (among other things) conflicts between write operations of the transaction being validated against the write operations of other concurrent transactions.
- To ensure that transactions at different servers are globally serializable, the servers may
  - conduct a global validation (check if there is a cyclic ordering) or
  - use the same globally unique transaction number for the same transaction.

Transaction Recovery

- Recall the ACID property of transactions
  - The effect of committed transactions must be maintained and aborted transactions should not alter the state.
- Typical issues:
  - Durability and redundancy
    - Keep critical information on disk
    - In-memory copies for performance
    - Ensure disk writes complete before continuing at critical times
    - Keep multiple copies of disk data
  - Protecting against …
    - Memory loss when system fails
    - Disk file loss with disk failure
- Recovery is done by the recovery manager.

Recovery Manager

- saves data items (sometimes plus operations to them) in permanent storage (in a recovery file) for committed transactions.
- restores the server's data items after crash.
- **Checkpoint** is a point of time at which all updates have been written onto the database from the buffers so that, in case of a system crash, the recovery manager does not have to redo the transactions that have been committed before checkpoint.
- Checkpoint can be done periodically to speed up recovery.



How Does It Work?

- During a transaction all updates to data items are performed in terms of the transaction’s own private tentative copies of the updated data items. These are kept in an intentions list (per server).
- During the two-phase commit protocol when a transaction:
  - **agrees to commit:** It must have saved its intentions list and the associated data items in the recovery file.
  - **commits:** Its tentative versions as reflected in its intentions list overwrite the committed versions. They are also written to the recovery file.
  - **aborts:** The intentions list is used to delete the tentative versions the transaction created.

Logging

- The recovery file represents a log of all the transactions performed by a server. The history contains:
  - The values of data items (initial snap shot followed by tentative updates), and
  - transaction status entries that in turn contain intentions lists and a linked list pointer to the previous entry and terminated at a checkpoint.
- The recovery manager processes transaction operations as follows:
  - **prepared to commit**: Appends all data items in the intentions list to the log followed by a 'prepared to commit' transaction status entry.
  - **commit**: 'Committed' transaction status entry

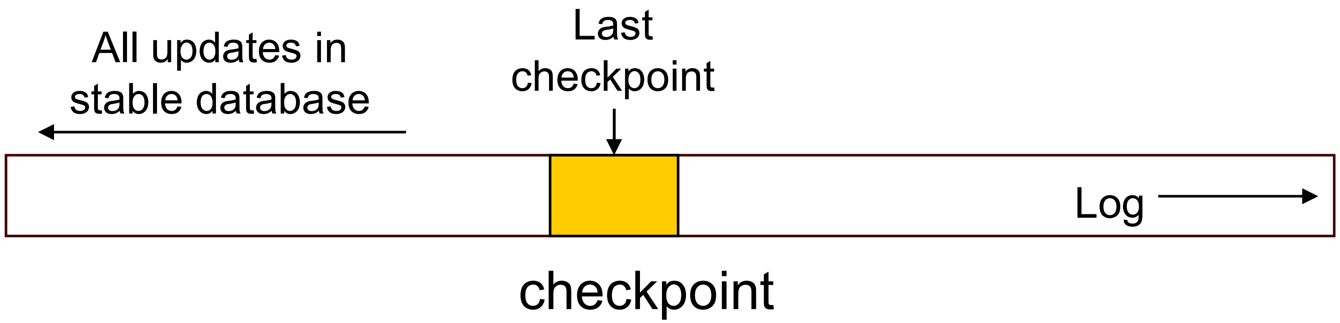
Types of Entry in a Recovery File

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 <objectID, Pi>, where Pi is the position in the recovery file of the value of the object.

the actual layout depends on the application

Recovery in Logging

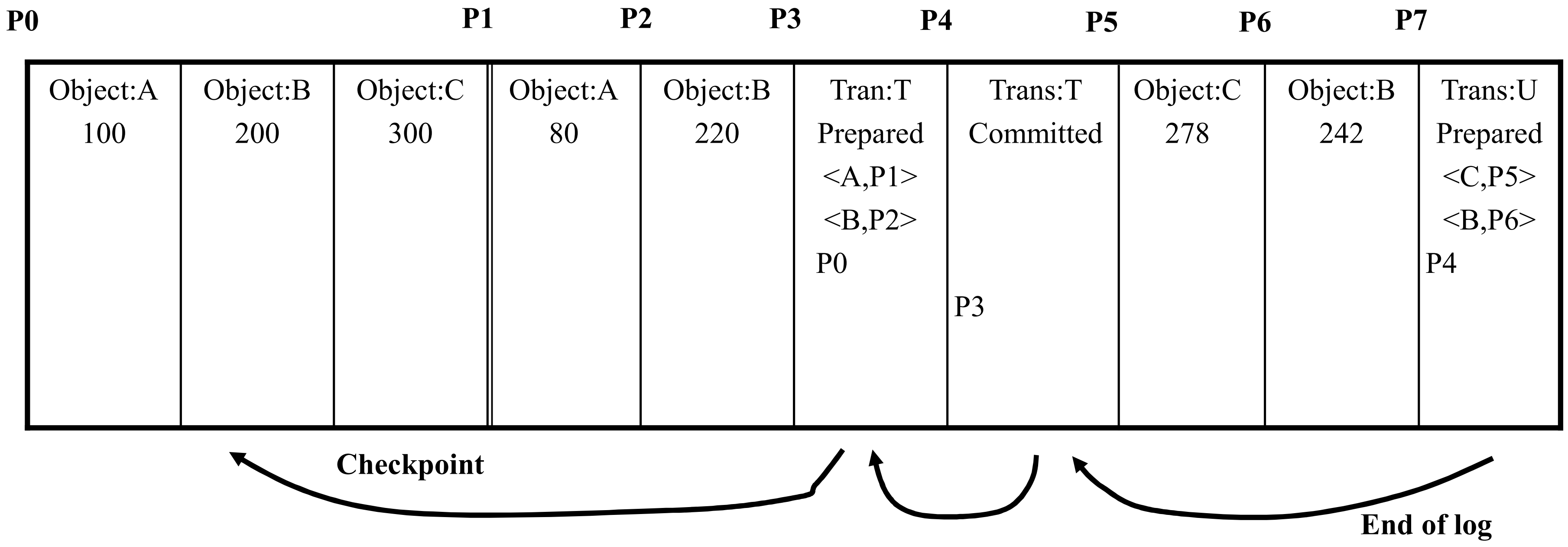
- When a server restarts, it first sets initial default values before handing over to the recovery manager.
- The recovery manager traverses the log file backwards following the linked list of transaction status entries until it reaches the last checkpoint. In doing so it
  - Restores values for committed transactions.
  - Converts prepared transactions to aborted.



Reorganizing the recovery file

- This is done periodically to make recovery faster and to reduce log file storage requirements. It can also be done as part of the recovery process, so called checkpointing
  - Adds a mark to the current recovery file.
  - Writes a checkpoint to the future recovery file.
  - Copies the committed values of a server's data to the future recovery file.
  - Copies across unresolved transactions before the mark and everything after the mark.
  - Switches to the new recovery file.

A Log for Banking Service

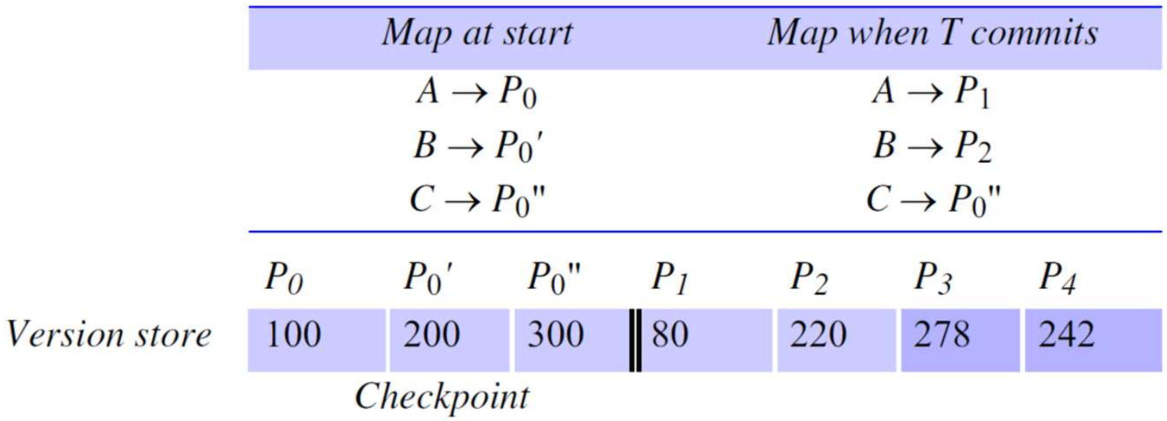


Object:A 100	Object:B 200	Object:C 300	Object:A 80	Object:B 220	Tran:T Prepared <A,P1> <B,P2> P0	Trans:T Committed P3	Object:C 278	Object:B 242	Trans:U Prepared <C,P5> <B,P6> P4

Depending on applications, some may log ‘actions to objects’ along with a checkpoint of the values of objects.

Shadow Versions

- **Shadow versions** uses a map to locate versions of the server's objects in a file (called **version store**) to speed up recovery.
  - The versions written by each transaction are 'shadows' of the previous committed versions.
  - The transaction status entries and intentions lists are stored separately.
  - When a transaction commits, a new map is made by copying the old map and entering the positions of the shadow versions. The new map then replaces the old map.
  - To restore the objects after crash, the recovery manager uses the map to locate the objects in the version store.
  - map can be written in the same file as the version store or separately.



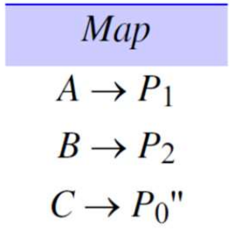
Map at start	*Map when T commits*
$A \rightarrow P_0$	$A \rightarrow P_1$
$B \rightarrow P_0'$	$B \rightarrow P_2$
$C \rightarrow P_0''$	$C \rightarrow P_0''$

	P_0	P_0'	P_0''	P_1	P_2	P_3	P_4
Version store	100	200	300	80	220	278	242

Checkpoint

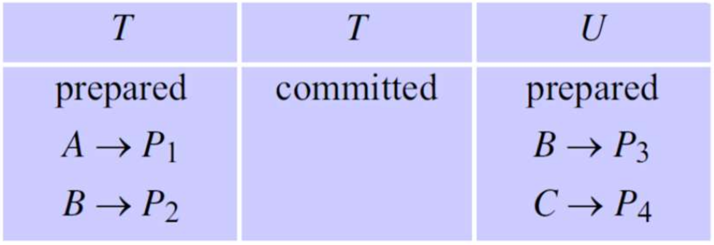
Shadow versions vs. Logging

- The shadow versions method provides faster recovery
- Logging should be faster than shadow versions during the normal activity of the system
  - requires only a sequence of append operations to the same log file, whereas shadow versions require an additional stable storage write (involving two unrelated disk blocks).
- The logging technique records transaction status entries, intentions lists and objects all in the same file.
- Care must be taken in shadow versions as fails may occur between updating the transaction status file and updating the map.



Map
$A \rightarrow P_1$
$B \rightarrow P_2$
$C \rightarrow P_0''$

Transaction status file (stable storage)



T	*T*	*U*
prepared $A \rightarrow P_1$ $B \rightarrow P_2$	committed	prepared $B \rightarrow P_3$ $C \rightarrow P_4$

Concurrency Control in Internet-Based Applications

- Abort is not realistic in Internet-based applications in which shared objects are typically large (e.g., documents) or transactions take some time to complete
  - Dropbox, Wikipedia, e-commerce, …
- Approach:
  - Some forms of optimistic concurrency control followed by conflict resolution