Distributed Systems (5DV020)

Concurrency Control

Fall 2014

Problems with concurrent transactions

- Transactions are carried out concurrently for higher performance
  - Otherwise, painfully slow
- Serial Equivalence
  - Interleaved operations produce same effect as if transactions have been performed one at a time
  - Does not mean to *actually* perform one transaction at a time, as this would lead to horrible performance
- Two operations are in conflict if the final result depends on the order of execution
  - Value set by a *write*
  - Result of a *read*

Read – Read → No conflict
Read – Write (or Write – Read) → Conflict!
Write – Write →Conflict!

Concurrency control

- ❑ Serialize access to objects
  - ❑ Each server is responsible for concurrency control on own objects
  - ❑ All servers are jointly responsible for concurrency control of conflicting transactions
- ❑ Ensure serially equivalent interleavings
- ❑ Maximize concurrency
  - ➢ Locks (wait for access)
  - ➢ Optimistic concurrency control (check for conflicts at the end)
  - ➢ Timestamp ordering (check to delay or reject)

Locks

Locks

- Need an object? Get a lock for it!
  - Read or write locks, or both (exclusive)
- Two-phase locking
  - Accumulate locks gradually, then release locks gradually
- Strict two-phase locking
  - Accumulate locks gradually, keep them all until completion

  Enables “strict” systems
- Granularity and tradeoffs

Locks

Transaction *T*:	**Transaction *U*:**
balance = b.getBalance()	*balance = b.getBalance()*
*b.setBalance(bal*1.1)*	*b.setBalance(bal*1.1)*
a.withdraw(bal/10)	*c.withdraw(bal/10)*

Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*		
		● ● ●	lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 16.14**

Sharing locks

- Read locks can be shared
- Promote read lock to write lock if no other transactions require a lock
- Requesting a write lock when there are already read locks, or a read lock when there is already a write lock?
  - Wait until lock is available

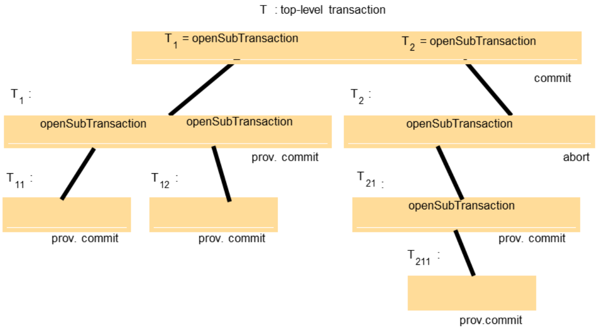
For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

Lock compatibility

Rules for strict two-phase locking

1. When an operation accesses an object within a transaction:
   - (a) If the object is not already locked, 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.

Locks and nested transactions



- ❑ Isolation
  - ➢ From other sets of nested transactions
  - ➢ From other transactions in own set
- ❑ Rules:

➢ Parents do not run concurrently with children

➢ Children can temporarily acquire locks from ancestors

➢ Parent inherits locks when child transactions commit

- ▪ Locks are discarded if child aborts

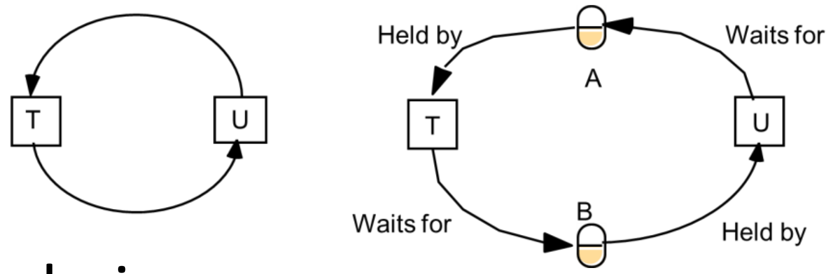
➢ Sub-transactions at each level are treated as flat transactions

There are also rules for acquiring and releasing locks

Big problem: Deadlocks

- Typical deadlock:

  Transaction A waits for B,
  transaction B waits for A



- Deadlocks may arise in long chains
- Conceptually, construct a *wait-for graph*
  - Directed edge between nodes if one waits for the other
  - Cycles indicate deadlocks

    Abort transaction(s) as needed

Handling deadlock

- Deadlock prevention
  - Acquire all locks from the beginning
    Bad performance, not always possible
- Deadlock detection
  - As soon as a lock is requested, check if a deadlock will occur
    Bad performance: avoid checking always
  - Must include algorithm for determining which transaction to abort
- Lock timeouts
  - Locks invulnerable for a certain time, then they are vulnerable
  - Leads to unnecessary aborts
    - Long-running transactions
    - Overloaded system
  - How to decide useful timeout value?

Distributed deadlock

- Local and distributed deadlocks

  Phantom deadlocks

- Simplest solution
  - Manager collects local wait-for information and constructs global wait-for graph
    - Single point of failure, bad performance, does not scale, what about availability, etc.
- Distributed solution – edge chasing or path pushing
  - Don’t construct a global wait-for graph, instead only send *probes*

Optimistic Concurrency control

Locks, drawbacks

- Overhead (even on read-only transactions)
  - Necessary only in the worst case
- Deadlock
  - Prevention reduces concurrency severely
  - Timeouts or detection
- Reduced concurrency in general
  - Locks need to be maintained until transactions end

Enter optimistic concurrency control

Optimistic Concurrency Control

- Assumes that conflicts are rare
  - Probability of multiple accesses to same object is low
  - Only need to worry about real conflicts
- Transaction phases:

Working

- Transaction works with tentative data (*read* and *write* sets)

Validation (Upon completion)

- Check if transaction may commit or abort
- Conflict resolution

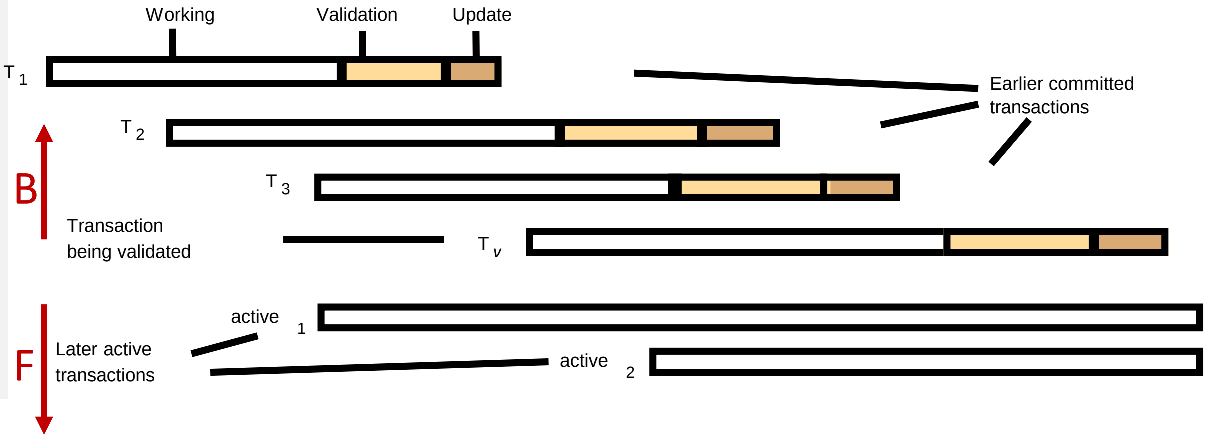
Update

- Write tentative data from committed transactions to permanent storage

Validation

- Use conflict rules from earlier!
  - On overlapping transactions
- Validate one transaction at a time against others
- Transactions are numbered (not to be confused with IDs) as they enter the validation phase
- Only a single transaction at a time in update phase
- Backward or Forward validation

T_v	T_i	Rule
write	read	T_i must not read objects written by T_v
read	write	T_v must not read objects written by T_i
write	write	T_i must not read objects written by T_v and T_i must not read objects written by T_v



Backward validation

- Check *read* set against *write* set of transactions that:
  - were active at the same time as the transaction currently being validated; and
  - have already committed
- Transactions with only *write* set need not be checked
- If overlap is found, then current transaction must be aborted!

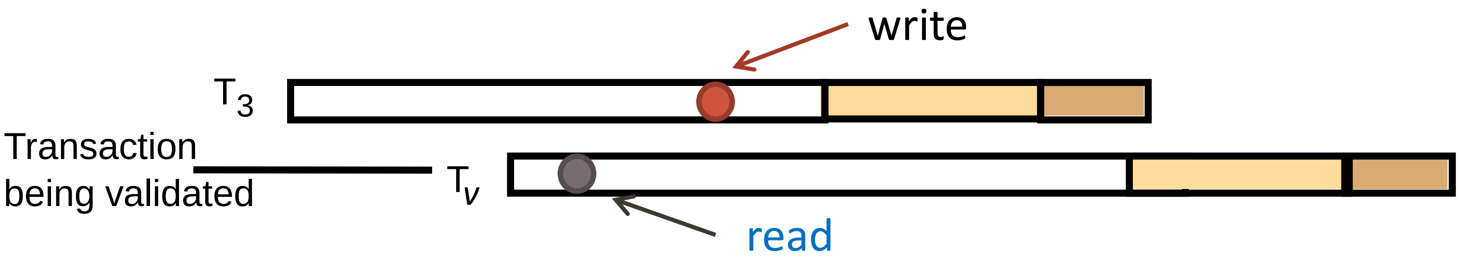
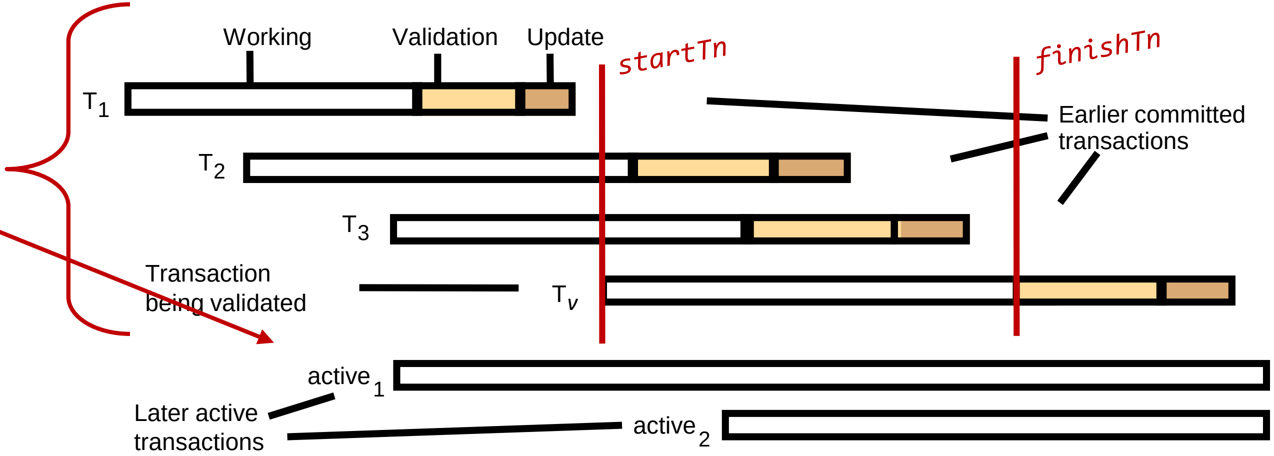


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 16.28**

Backward validation - example

write sets must be saved until active finishes




```
Backward validation of transaction Tv
        boolean valid = true;
        for (int Ti  = startTn+1; Ti <= finishTn; Ti++){
                if (read set of Tv intersects write set of Ti) valid = false;
        }
```

Forward validation

- Check *write* set against *read* set of transactions that are currently active
  - Note that read sets of active transactions may change during validation
- *read-only* transactions need not be checked
- If overlap is found, we can choose which transaction(s) to abort
  - Wait until conflicting transactions have finished
  - Abort conflicting active transactions
  - Abort transactions being validated

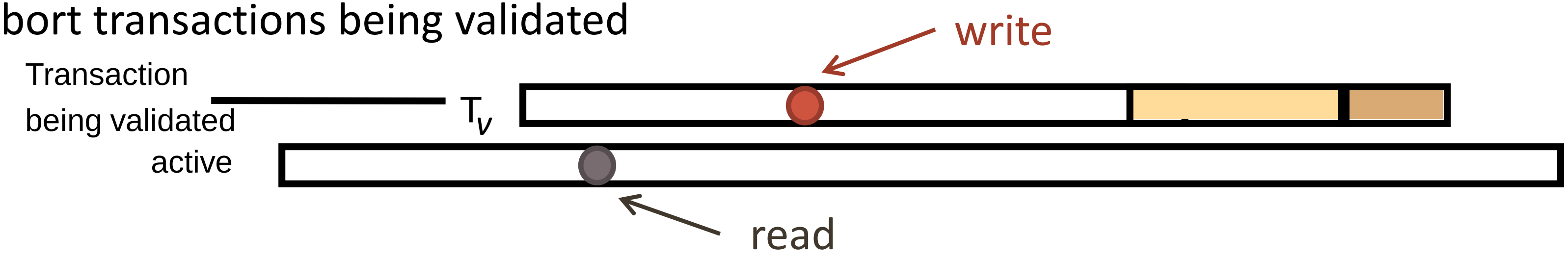
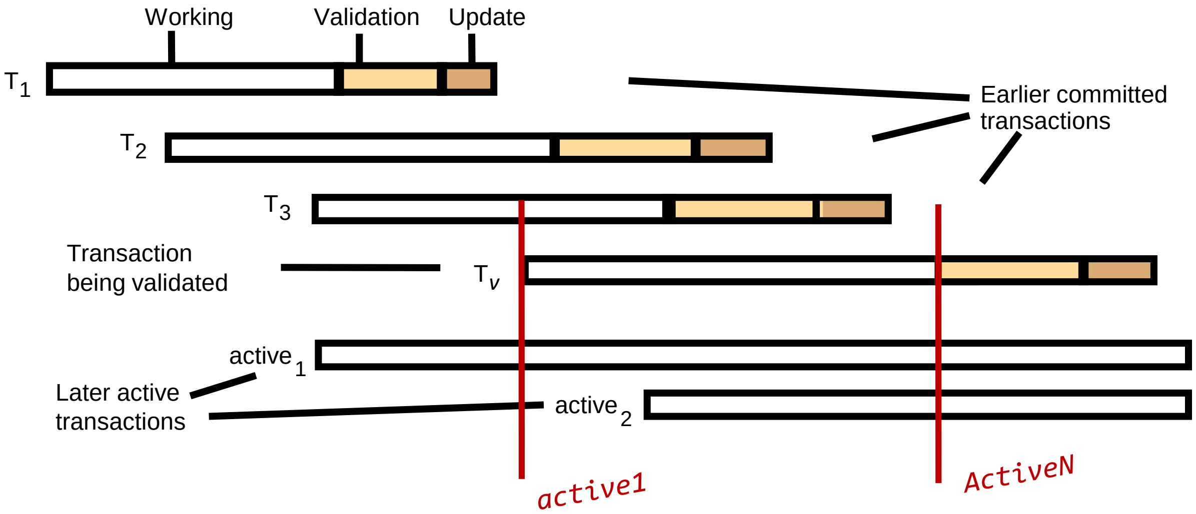


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 18.2**

Forward validation - example




```
Forward validation of transaction Tv
        boolean valid = true;
        for (int Tid = active1; Tid <= activeN; Tid++){
                if (write set of Tv intersects read set of Tid) valid = false;
        }
```

Comparison of validation schemes

- Size of *read/write* sets
  - *Read* sets are usually bigger
  - Forward compares against “growing” *read* sets
- Choice of transaction to abort
  - Backward a single choice, Forward three choices
  - Linked to starvation
- Overhead
  - Backward requires storing old *write* sets
  - Forward may need to re-run each time the *read* set for any active transaction changes and must allow for checking new valid transactions

Timestamp ordering

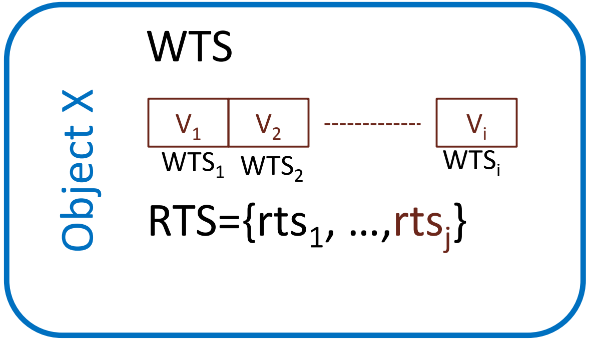
Overview

- Avoids locks, relies on timestamps
- Transactions are assigned timestamps when they start
  - Timestamps are assigned to all read and write accesses that a transaction makes
- Read and write access is granted according to timestamp order – validated when carried out
  - Requests are totally ordered
  - Serial execution of transactions
  - Transactions are aborted if validation is unsuccessful

Ordering rule

- Based on operation conflicts
  - Writes are valid only if the object was last read or written by earlier transactions
  - Reads are valid only if the object was last written by an earlier transactions
- Transactions can access an object concurrently
  - Writes on tentative versions until committed
  - Writes may be performed after *closeTransaction()*
  - Reads must wait for earlier transactions to finish (no deadlock)

Details



- Tentative versions are created when writes are accepted
  - Write timestamp set to transaction timestamp
- Reads are directed to a version according to timestamp
  - The earliest version
  - Transaction timestamp is added to read timestamps
- For commits:
  - Tentative version becomes the object (values)
  - Tentative version timestamps become the objects' timestamps

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 $T_c >$ 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 $T_c >$ write timestamp of the committed object.

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 16.29**

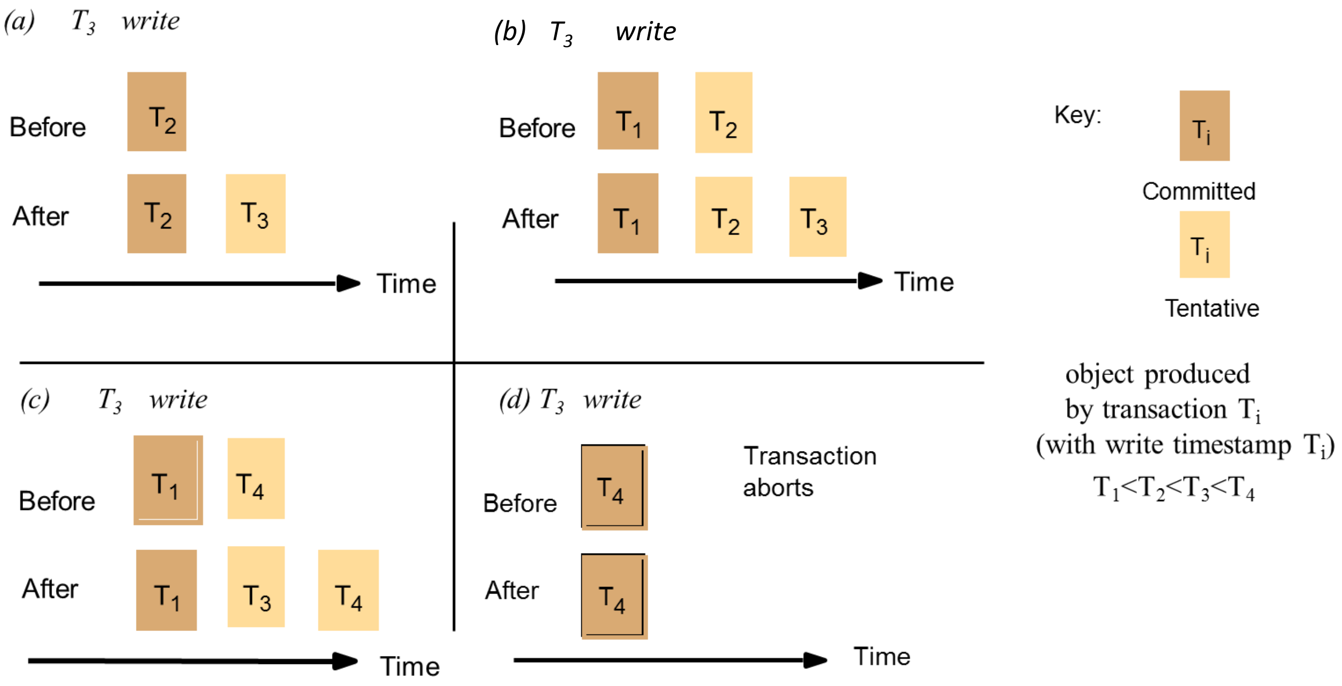
Timestamp ordering write rule

- A write is accepted if (transaction T_c, object D):

```
if (Tc ≥ maximum read timestamp on D && Tc > write timestamp on committed version of D)
        perform write operation on tentative version of D with write timestamp Tc
else /* too late */
        Abort transaction Tc
```

- When a write is accepted a new tentative version is created with timestamp T_c
- Writes that arrive too late are aborted
  - A transaction with a later timestamp has already operated on the object

Example: write operations



Timestamp ordering read rule

- A read is accepted if (transaction T_c, object D):

```
if ( Tc > 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 Tc
```

- Reads that arrive too early need to wait for the earlier transaction to complete (aborts dirty reads)
- Reads that arrive too late are aborted

Example: read operations

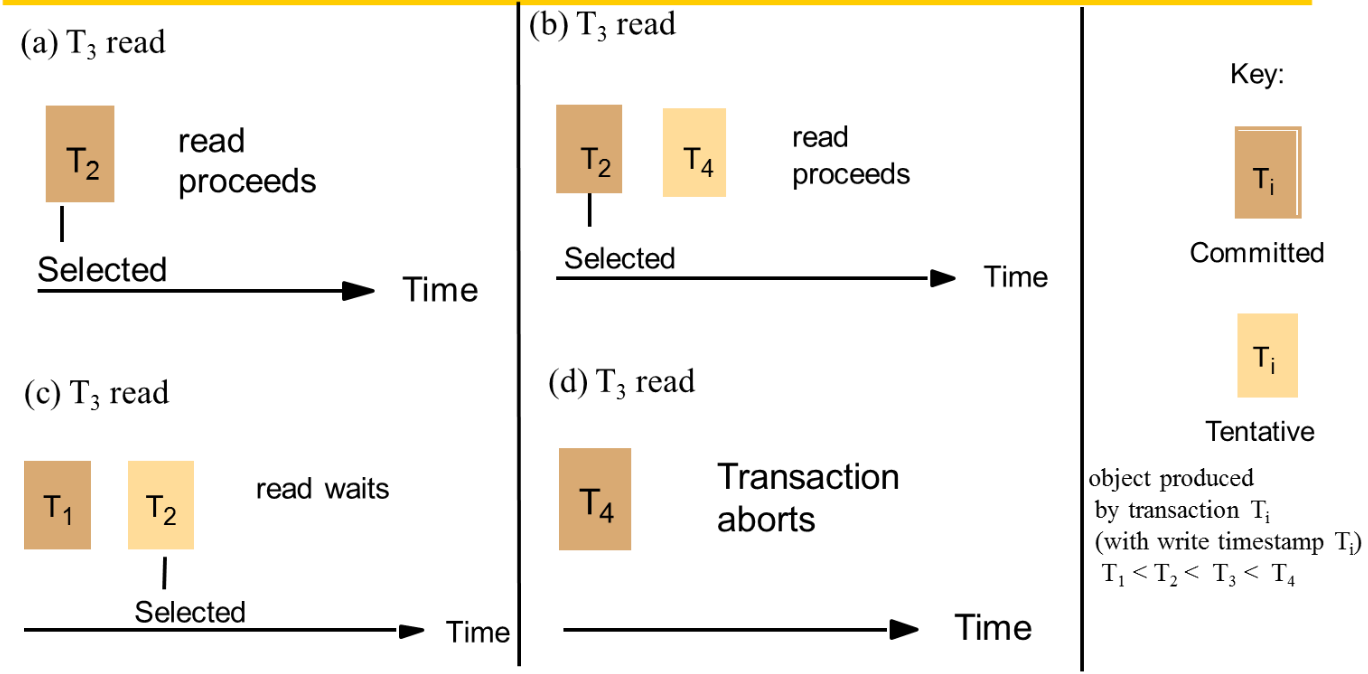


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 16.31**

Combined example

		Timestamps and versions of objects					
T	*U*	*A*		*B*		*C*	
		RTS	*WTS*	*RTS*	*WTS*	*RTS*	*WTS*
		{}	***S***	{}	***S***	{}	***S***
openTransaction							
bal = b.getBalance()				{*T*}			
	openTransaction						
*b.setBalance(bal*1.1)*					***S***, *T*		
	bal = b.getBalance()						
	wait for T						
a.withdraw(bal/10)	●●●		***S***, *T*				
commit	●●●		***T***		***T***		
	bal = b.getBalance()			{*U*}			
	*b.setBalance(bal*1.1)*				***T***, *U*		
	c.withdraw(bal/10)						***S***, *U*

Summary

- Comparison of concurrency control schemes
- Pessimistic CC
  - Two-phase locking – serialization ordering is decided dynamically
  - Transactions need to wait for locks ...and yet, can still be aborted
  - Locking maybe beneficial for transactions with more writes than reads (compared against timestamp ordering)
  - Large overhead (avoided in new systems)

- ❑ Timestamp ordering – serialization ordering is decided statically
  - ❑ Beneficial for transactions with more reads than writes
- ❑ For systems with many CC-r For systems with many CC-related issues
  - ➢ Pessimistic will give a more stable quality of service
  - ➢ Optimistic will abort a large number of transactions and requires substantial work

Advanced DS course (this fall)

http://www8.cs.umu.se/kurser/5DV153/HT14/

Next Lecture

Peer-2-peer
and
explanation of PGcom