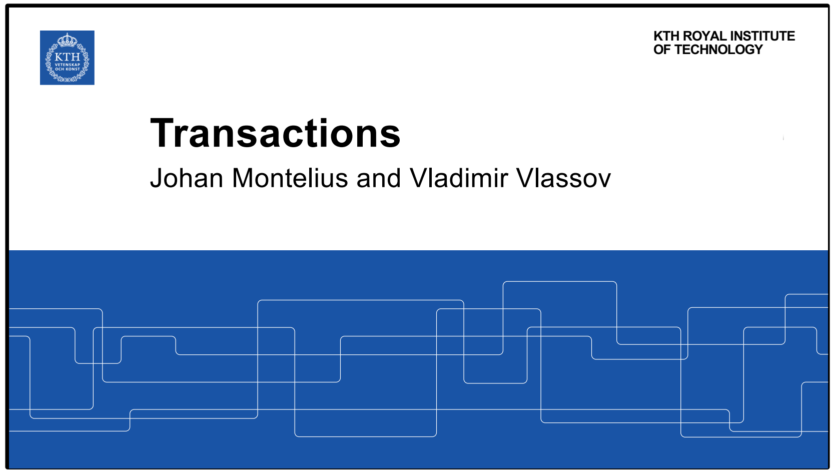
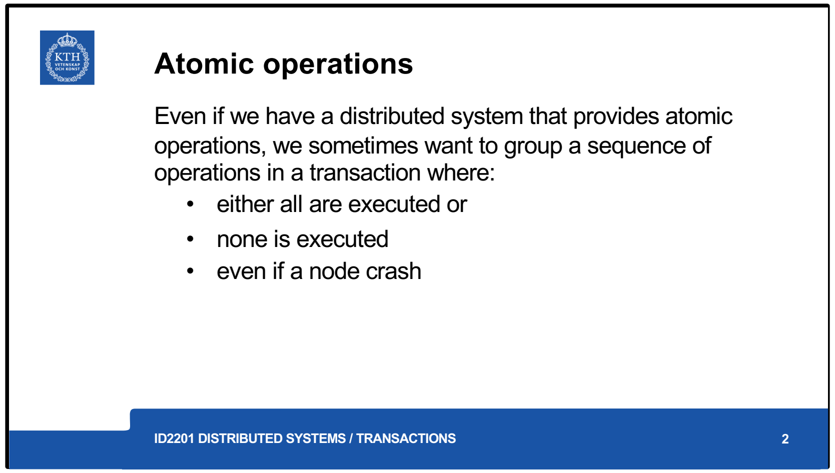


KTH ROYAL INSTITUTE
OF TECHNOLOGY

Transactions

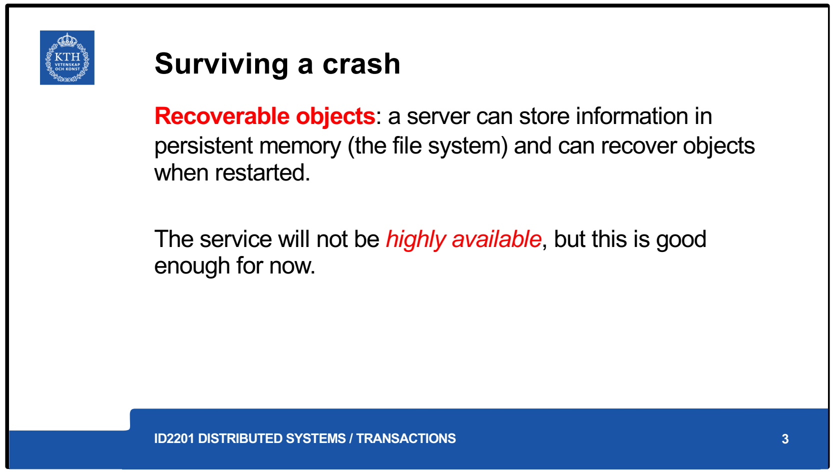
Johan Montelius and Vladimir Vlassov



Atomic operations

Even if we have a distributed system that provides atomic operations, we sometimes want to group a sequence of operations in a transaction where:

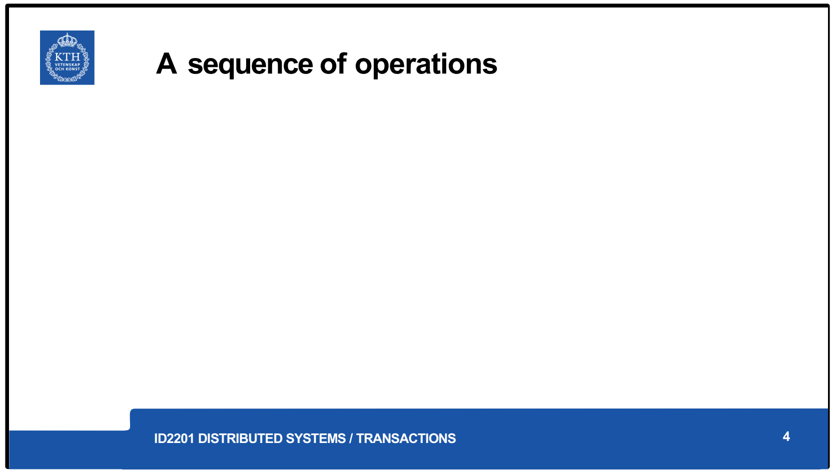
- either all are executed or
- none is executed
- even if a node crash



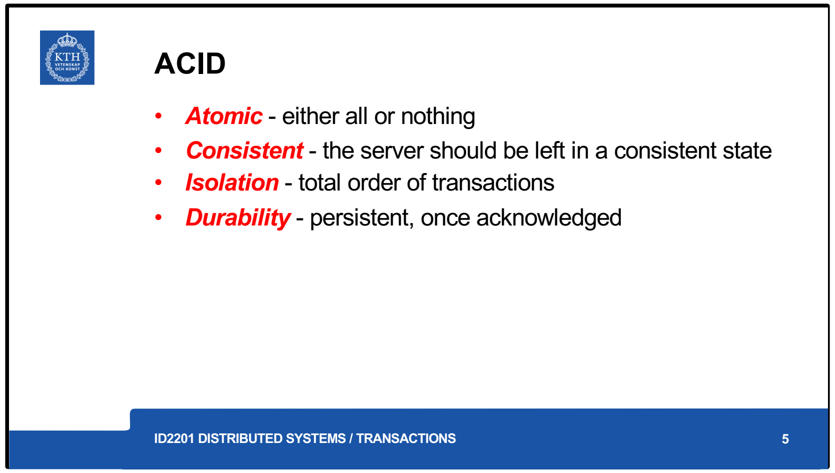
Surviving a crash

Recoverable objects: a server can store information in persistent memory (the file system) and can recover objects when restarted.

The service will not be *highly available*, but this is good enough for now.

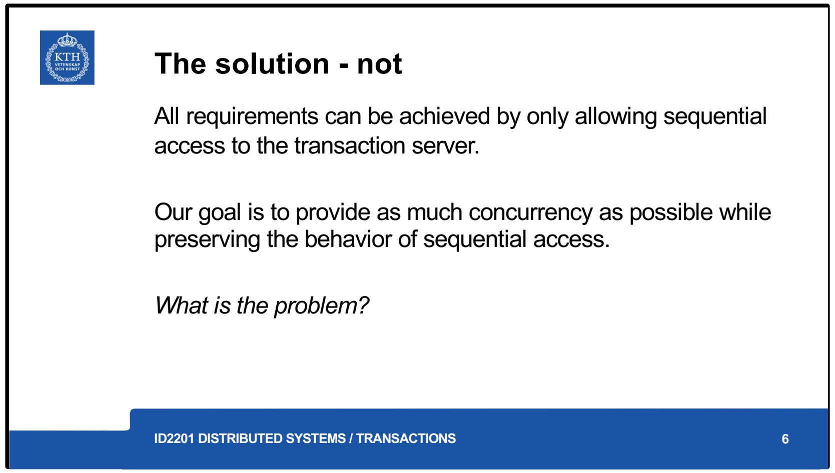


A sequence of operations



ACID

- ***Atomic*** - either all or nothing
- ***Consistent*** - the server should be left in a consistent state
- ***Isolation*** - total order of transactions
- ***Durability*** - persistent, once acknowledged


The solution - not

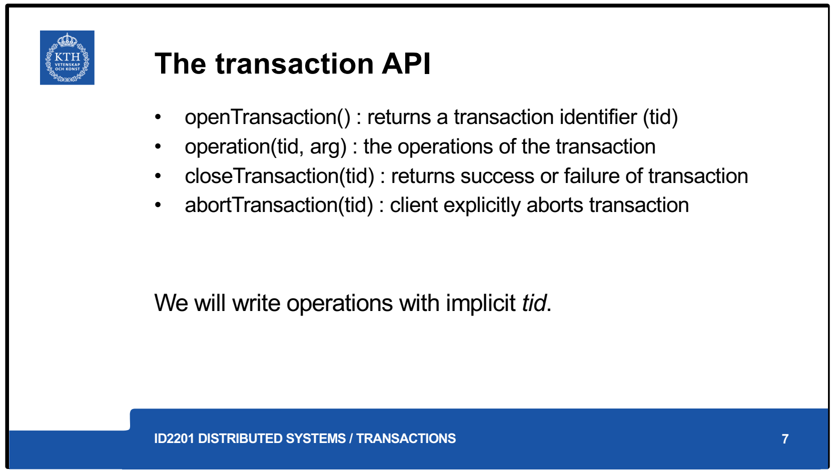
All requirements can be achieved by only allowing sequential access to the transaction server.

Our goal is to provide as much concurrency as possible while preserving the behavior of sequential access.

What is the problem?

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 6

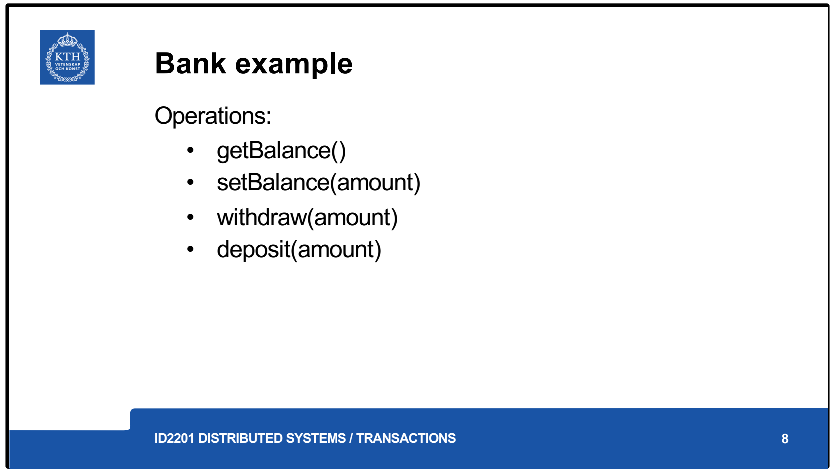
Concurrent transactions = interleaved trasactions



The transaction API

- openTransaction() : returns a transaction identifier (tid)
- operation(tid, arg) : the operations of the transaction
- closeTransaction(tid) : returns success or failure of transaction
- abortTransaction(tid) : client explicitly aborts transaction

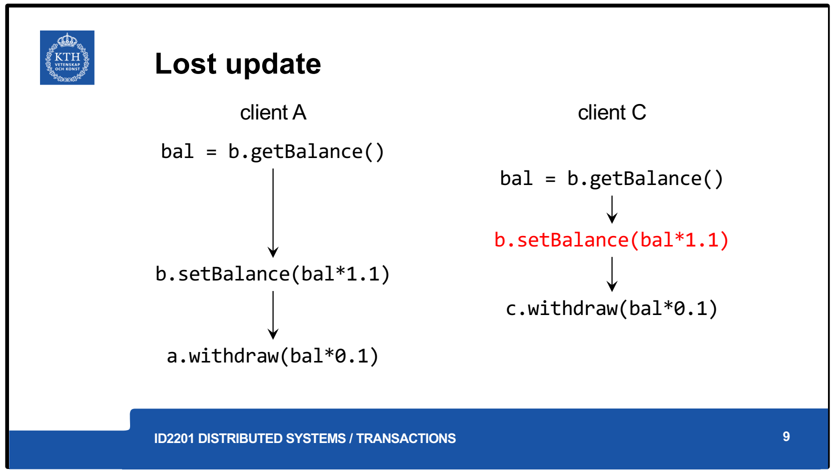
We will write operations with implicit *tid*.



Bank example

Operations:

- getBalance()
- setBalance(amount)
- withdraw(amount)
- deposit(amount)

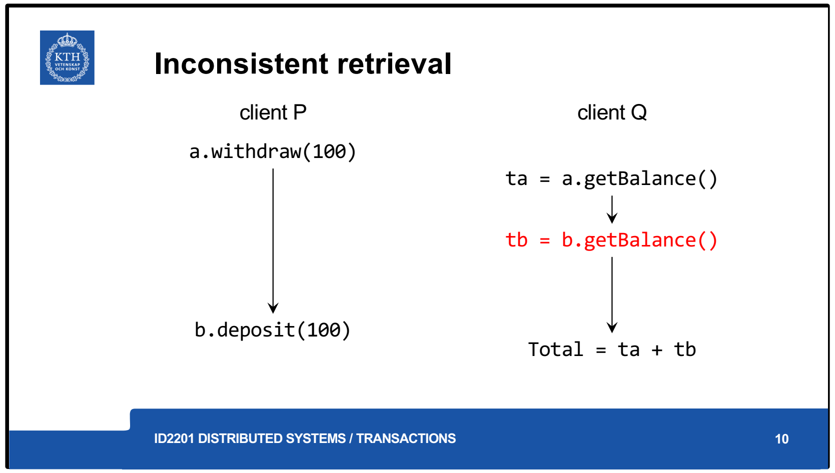


Lost update

```
         client A                              client C

bal = b.getBalance()
          |                           bal = b.getBalance()
          |                                     ↓
          |                           b.setBalance(bal*1.1)
          ↓                                     ↓
b.setBalance(bal*1.1)                  c.withdraw(bal*0.1)
          ↓
 a.withdraw(bal*0.1)
```

Assume that all operations are synchronized, i.e. Atomic.
Client A: increase **b** by 10% from **a**
Client C: increase **b** by 10% from **c**
The net effects on account **b** of executing the transactions of both clients should be to increase the balance of account **b by 10% twice**. However it might be not the case if the transactions are executed concurrently. This is an illustration of the **'lost update' problem**. C's update is lost because client A overwrites it without seeing it. Both transactions have read the old value before either writes the new value.
A concurrent program of n processes each of m atomic actions can produce (n*m)!/(m!)^n different histories. 20 in this example.

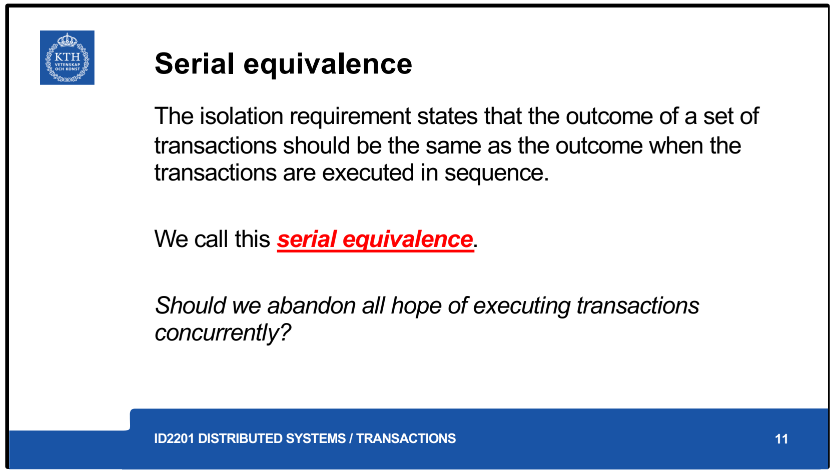


Inconsistent retrieval

```
client P                         client Q

a.withdraw(100)
       |                         ta = a.getBalance()
       |                                 ↓
       |                         tb = b.getBalance()
       ↓                                 |
b.deposit(100)                           ↓
                                 Total = ta + tb
```

Client P: trasfer 100 from A to B
Client Q: get total balance of A and B
The balances of the two bank accounts, A and B, are both initially $200. The result of branch Total includes the sum of A and B as $300, which is wrong. This is an illustration of the '***inconsistent retrievals***' problem. q's retrievals are inconsistent because p has performed only the withdrawal part of a transfer at the time the sum is calculated.
Also **unrepeatable read (read-write conflict)**



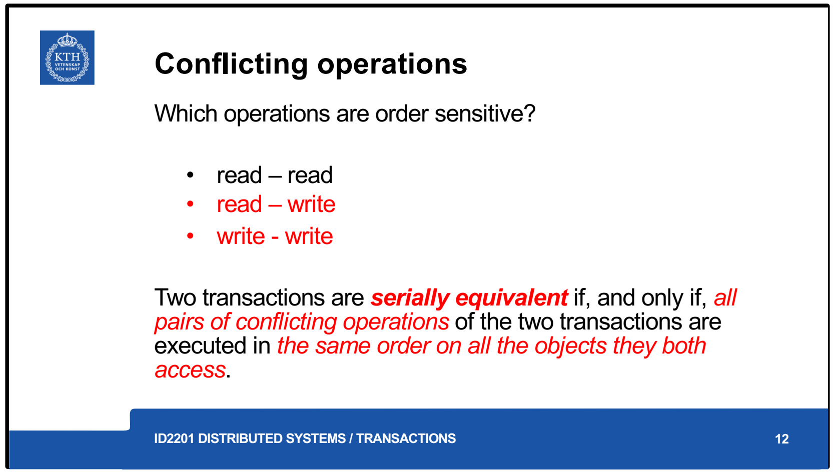
Serial equivalence

The isolation requirement states that the outcome of a set of transactions should be the same as the outcome when the transactions are executed in sequence.

We call this ***serial equivalence***.

Should we abandon all hope of executing transactions concurrently?

If each of several transactions is known to have the correct effect when it is done on its own, then we can infer that if these transactions are done one at a time in some order the combined effect will also be correct. **An interleaving of the operations of transactions** in which the combined effect is the same as if the transactions had been performed one at a time in some order **is a *serially equivalent* interleaving.**
The use of serial equivalence as a criterion for correct concurrent execution prevents the occurrence of lost updates and inconsistent retrievals.



Conflicting operations

Which operations are order sensitive?

- read – read
- read – write
- write - write

Two transactions are ***serially equivalent*** if, and only if, *all pairs of conflicting operations* of the two transactions are executed in *the same order on all the objects they both access*.

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 12

A***pair of operations conflicts,*** when their ***combined effect depends on the order in which they are executed***. To simplify matters we consider a pair of operations, read and write. The conflict rules for read and write are as follows:

read – read: No conflict because the effect of a pair of reads does not depend on the order in which they are executed

read – write: Conflict because the effect of read and write depends on the order of their execution

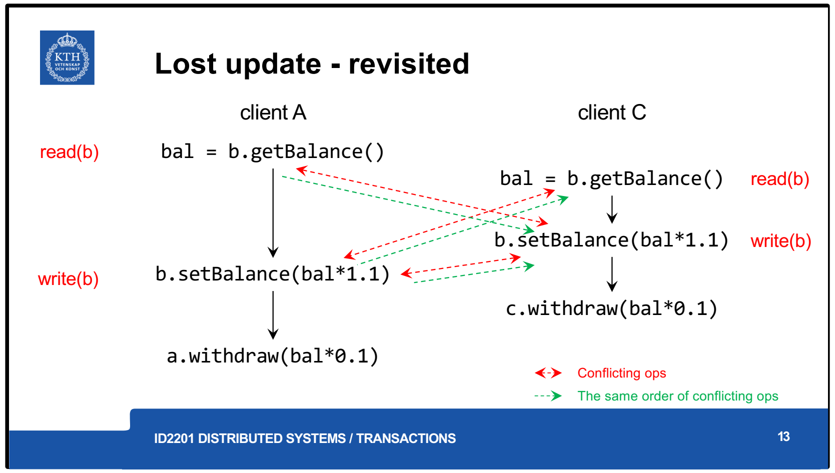
write – write: Conflict because the effect of a pair of writes depends on the order of their execution

When execution of transactions is interleaved, we can have 3 different violations:

write-read: conflict (dirty read)

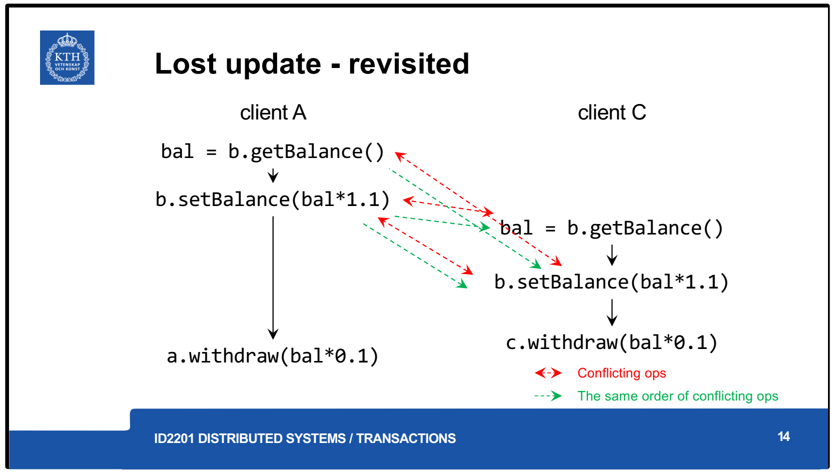
read-write: conflict (unrepeatable read)

write-write: conflict (overwriting uncommitted data)



Lost update - revisited

The lost update problem occurs when two transactions read the old value of a variable and then use it to calculate the new value. This cannot happen if one transaction is performed before the other, because the later transaction will read the value written by the earlier one. **The same order**: if **A** reads first (getBalance) and **C** write second (SetBalance) then **A** writes first (SetBalance) and **C** reads second (getBalance); and **A** writes first (SetBalace) and **C** writes second.



Lost update - revisited

client A

```
bal = b.getBalance()
b.setBalance(bal*1.1)
a.withdraw(bal*0.1)
```

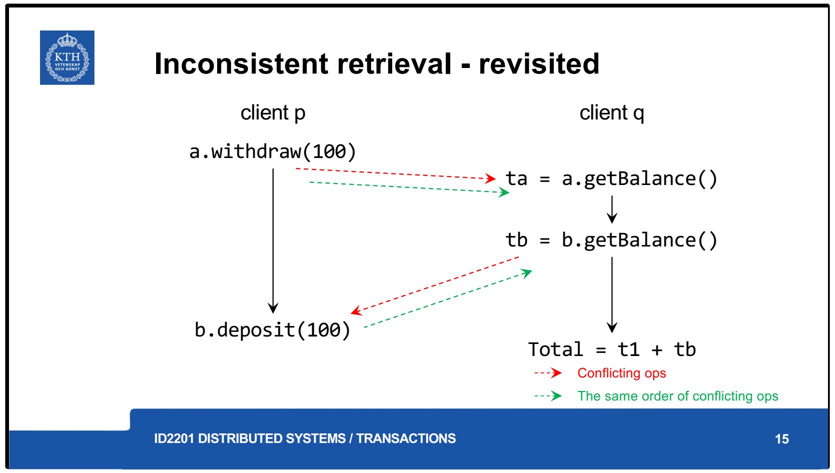
client C

```
bal = b.getBalance()
b.setBalance(bal*1.1)
c.withdraw(bal*0.1)
```

<-> Conflicting ops

---> The same order of conflicting ops

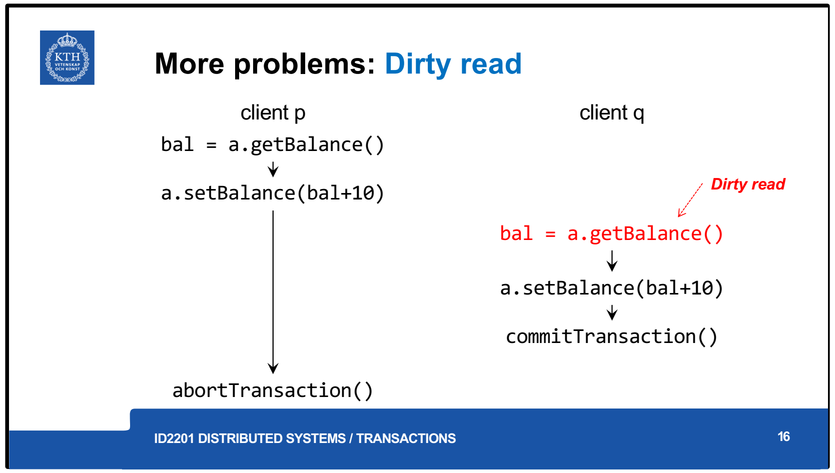
The lost update problem occurs when two transactions read the old value of a variable and then use it to calculate the new value. This cannot happen if one transaction is performed before the other, because the later transaction will read the value written by the earlier one. **The same order**: if **A** reads first (getBalance) and **C** write second (SetBalance) then **A** writes first (SetBalance) and **C** reads second (getBalance); and **A** writes first (SetBalace) and **C** writes second.



Inconsistent retrieval - revisited

The inconsistent retrievals problem can occur when a retrieval transaction runs concurrently with an update transaction. It cannot occur if the retrieval transaction is performed before or after the update transaction. **A serially equivalent interleaving** of a retrieval transaction and an update transaction will prevent inconsistent retrievals occurring.

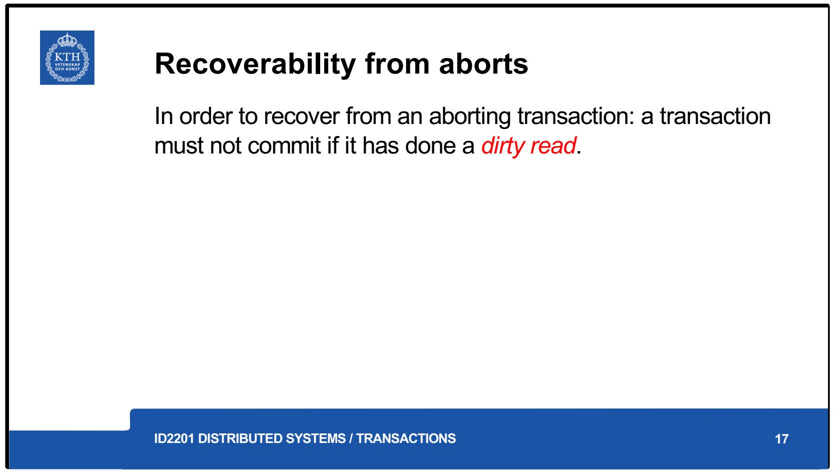
Serial equivalence is used as a criterion for the derivation of concurrency control protocols. ***These protocols attempt to serialize transactions in their access to objects. Three alternative approaches to concurrency control are commonly used: locking, optimistic concurrency control and timestamp ordering. However, most practical systems use locking***



More problems: Dirty read

```
client p                          client q
bal = a.getBalance()
a.setBalance(bal+10)
                                  bal = a.getBalance()
                                  a.setBalance(bal+10)
                                  commitTransaction()
abortTransaction()
```

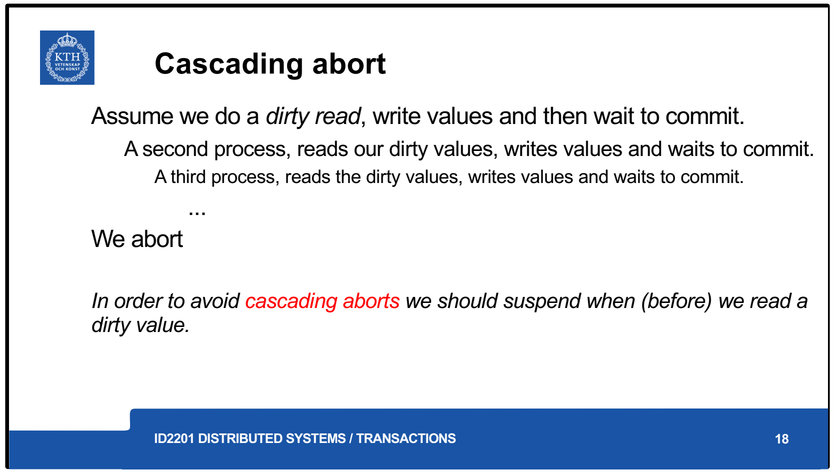
Servers must record all the effects of committed transactions and none of the effects of aborted transactions. **They must therefore allow for the fact that a transaction may abort by preventing it affecting other concurrent transactions if it does so. This section illustrates two problems associated with aborting transactions in the context of the banking example: dirty reads and premature writes.**
The isolation property of transactions requires that transactions do not see the uncommitted state of other transactions. ***The 'dirty read' problem is caused by the interaction between a read operation in one transaction and an earlier write operation in another transaction on the same object. In the case that p aborts, then q must abort as well.***



Recoverability from aborts

In order to recover from an aborting transaction: a transaction must not commit if it has done a *dirty read*.

If a transaction (like q) has committed after it has seen the effects of a transaction that subsequently aborted, the situation is not recoverable. To ensure that such situations will not arise, any transaction (like q) ***that is in danger of having a dirty read delays its commit operation***. The strategy for recoverability is to delay commits until after the commitment of any other transaction whose uncommitted state has been observed. In the case that p aborts, then q must abort as well.



Cascading abort

Assume we do a *dirty read*, write values and then wait to commit.

A second process, reads our dirty values, writes values and waits to commit.

A third process, reads the dirty values, writes values and waits to commit.

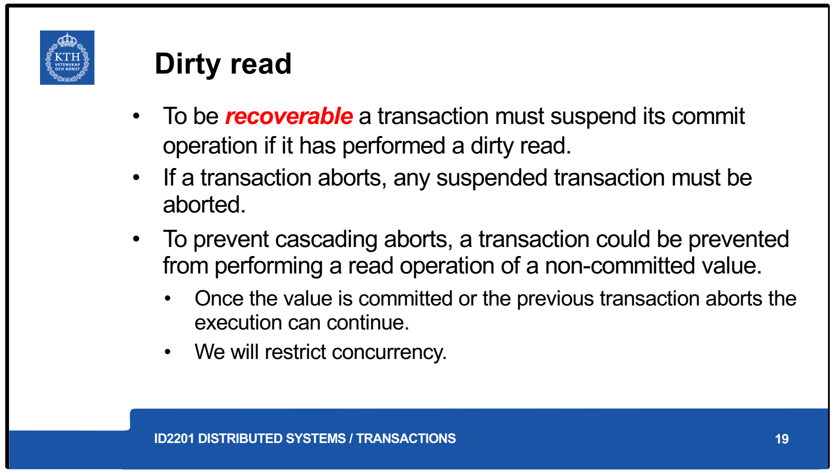
...

We abort

In order to avoid cascading aborts we should suspend when (before) we read a dirty value.

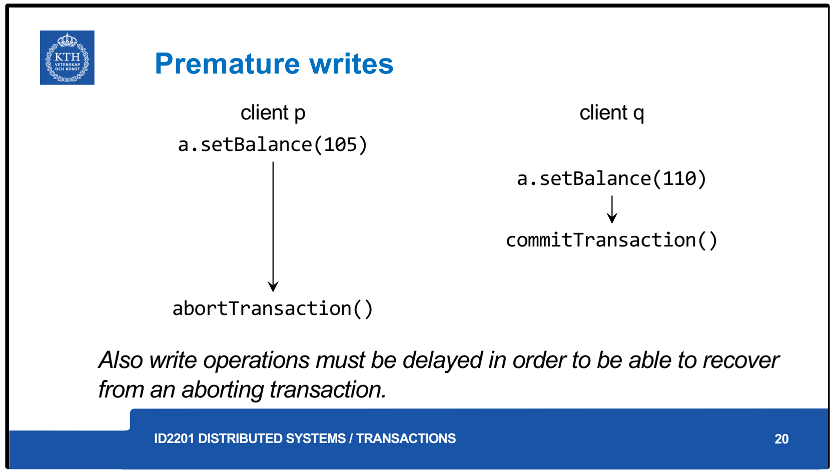
ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 18

To avoid cascading aborts, transactions are only allowed to read objects that were written by committed transactions. ***To ensure that this is the case, any read operation must be delayed until other transactions that applied a write operation to the same object have committed or aborted.*** The avoidance of cascading aborts is a stronger condition than recoverability.



Dirty read

- To be ***recoverable*** a transaction must suspend its commit operation if it has performed a dirty read.
- If a transaction aborts, any suspended transaction must be aborted.
- To prevent cascading aborts, a transaction could be prevented from performing a read operation of a non-committed value.
  - Once the value is committed or the previous transaction aborts the execution can continue.
  - We will restrict concurrency.



Premature writes

client p	client q
`a.setBalance(105)`	
	`a.setBalance(110)`
	`commitTransaction()`
`abortTransaction()`	

Also write operations must be delayed in order to be able to recover from an aborting transaction.

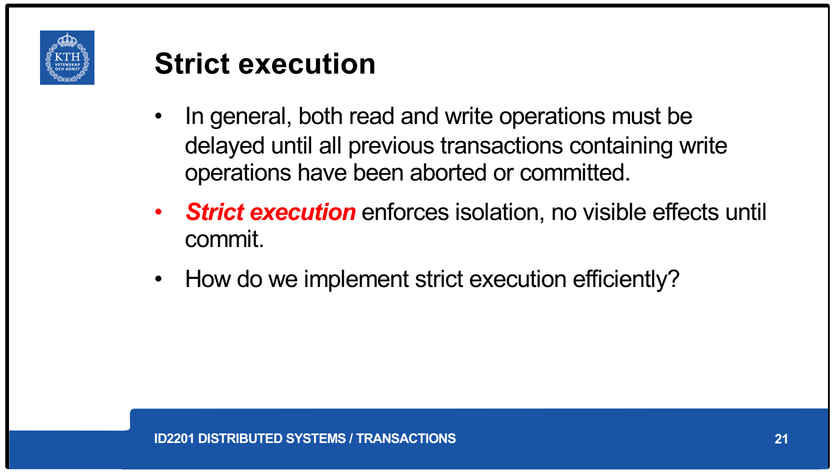
Consider another implication of the possibility that a transaction may abort. This one is related to ***the interaction between write operations on the same object belonging to different transactions***.

Assume a = 100 before; serial effect of p then q is 110;

after abortTransaction the state should be 110 but it is set to 100, i.e. the state (“ before image”) before p started.

Some database systems implement the action of abort by restoring ‘before images’ of all the writes of a transaction.

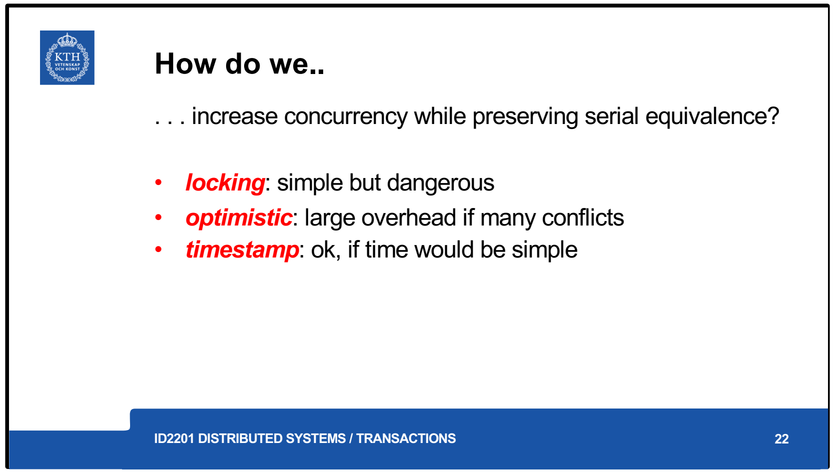
To ensure correct results in a recovery scheme that uses before images, write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted.



Strict execution

- In general, both read and write operations must be delayed until all previous transactions containing write operations have been aborted or committed.
- ***Strict execution*** enforces isolation, no visible effects until commit.
- How do we implement strict execution efficiently?

For a server of recoverable objects to participate in transactions, it must be designed so that any updates of objects can be removed if and when a transaction aborts. To make this possible, ***all of the update operations performed during a transaction are done in tentative versions of objects in volatile memory***. Each transaction is provided with its own private set of tentative versions of any objects that it has altered. ***The tentative versions are transferred to the objects only*** when a transaction commits, by which time they will also have been recorded in permanent storage. When a transaction aborts, its tentative versions are deleted.

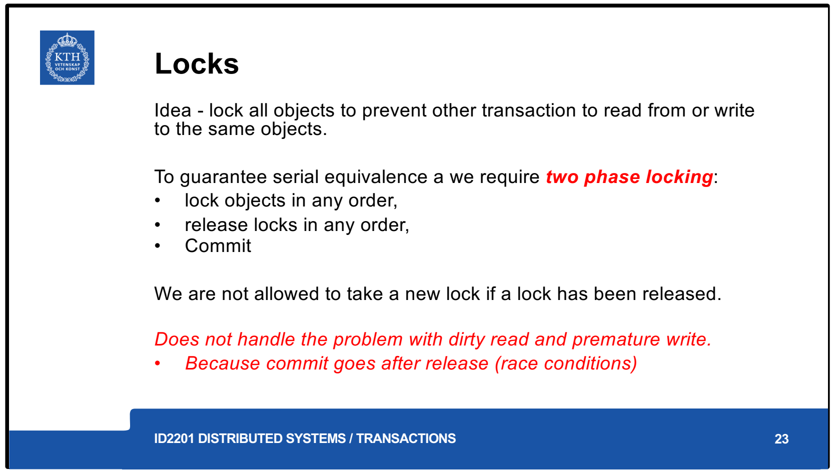


How do we..

. . . increase concurrency while preserving serial equivalence?

- ***locking***: simple but dangerous
- ***optimistic***: large overhead if many conflicts
- ***timestamp***: ok, if time would be simple

Serial equivalence is used as a criterion for the derivation of concurrency control protocols. These protocols attempt to serialize transactions in their access to objects. Three alternative approaches to concurrency control are commonly used: locking, optimistic concurrency control and timestamp ordering. Basically, concurrency control can be achieved either by clients' transactions waiting for one another or by restarting transactions after conflicts between operations have been detected, or by a combination of the two.



Locks

Idea - lock all objects to prevent other transaction to read from or write to the same objects.

To guarantee serial equivalence a we require ***two phase locking***:

- lock objects in any order,
- release locks in any order,
- Commit

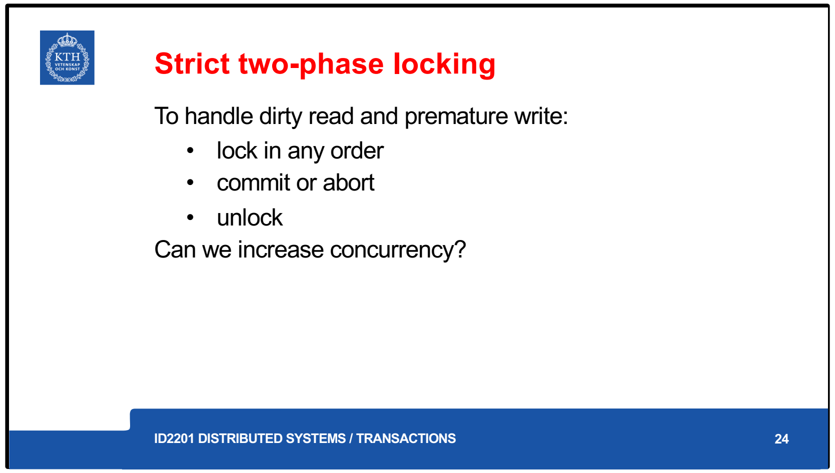
We are not allowed to take a new lock if a lock has been released.

Does not handle the problem with dirty read and premature write.

- *Because commit goes after release (race conditions)*

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 23

Transactions must be scheduled so that their effect on shared data is serially equivalent. A server can achieve serial equivalence of transactions ***by serializing access*** to the objects.
A simple example of a serializing mechanism is the use of ***exclusive locks***.
All pairs of conflicting operations of two transactions should be executed in the same order. To ensure this, a transaction is not allowed any new locks after it has released a lock. The first phase of each transaction is a '***growing phase***', during which new locks are acquired. In the second phase, the locks are released (a '***shrinking phase***'). This is called ***two-phase locking.***



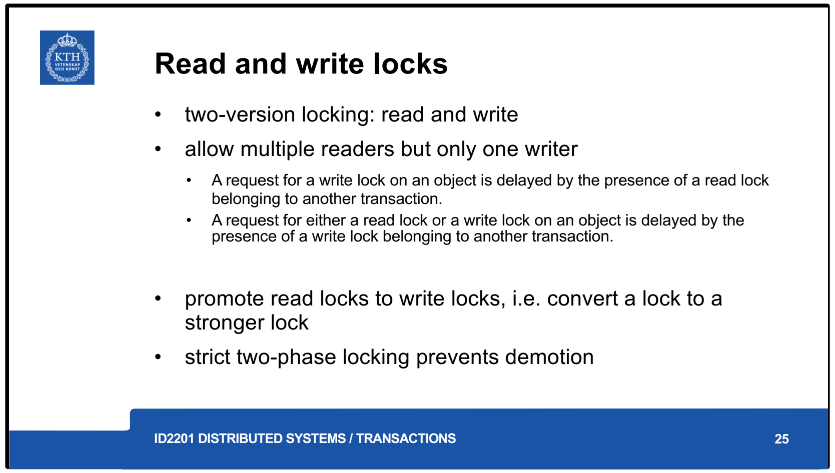
Strict two-phase locking

To handle dirty read and premature write:

- lock in any order
- commit or abort
- unlock

Can we increase concurrency?

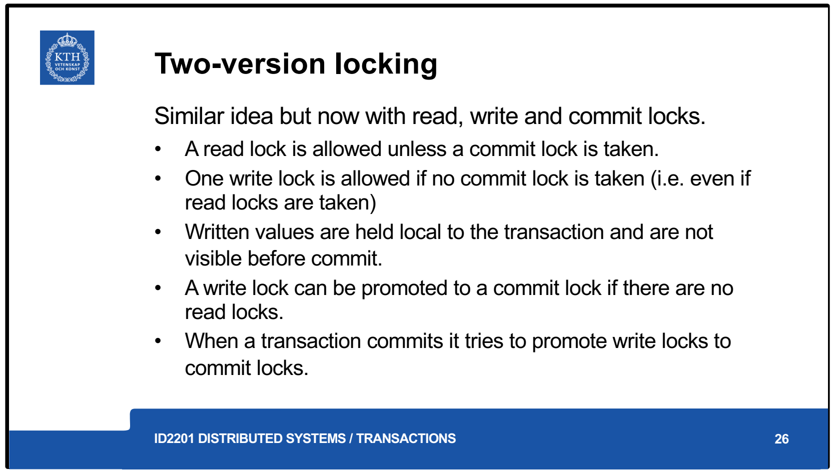
because transactions may abort, strict executions are needed to prevent dirty reads and premature writes. Under a strict execution regime, a transaction that needs to read or write an object must be delayed until other transactions that wrote the same object have committed or aborted. To enforce this rule, any locks applied during the progress of a transaction are held until the transaction commits or aborts. This is called ***strict two-phase locking***. The presence of the locks prevents other transactions reading or writing the objects. When a transaction commits, to ensure recoverability, the locks must be held until all the objects it updated have been written to permanent storage.



Read and write locks

- two-version locking: read and write
- allow multiple readers but only one writer
  - A request for a write lock on an object is delayed by the presence of a read lock belonging to another transaction.
  - A request for either a read lock or a write lock on an object is delayed by the presence of a write lock belonging to another transaction.
- promote read locks to write locks, i.e. convert a lock to a stronger lock
- strict two-phase locking prevents demotion

pairs of read operations from different transactions on the same object do not conflict. Therefore, a simple exclusive lock that is used for both read and write operations reduces concurrency more than is necessary.
A request for a write lock on an object is delayed by the presence of a read lock belonging to another transaction.
A request for either a read lock or a write lock on an object is delayed by the presence of a write lock belonging to another transaction.



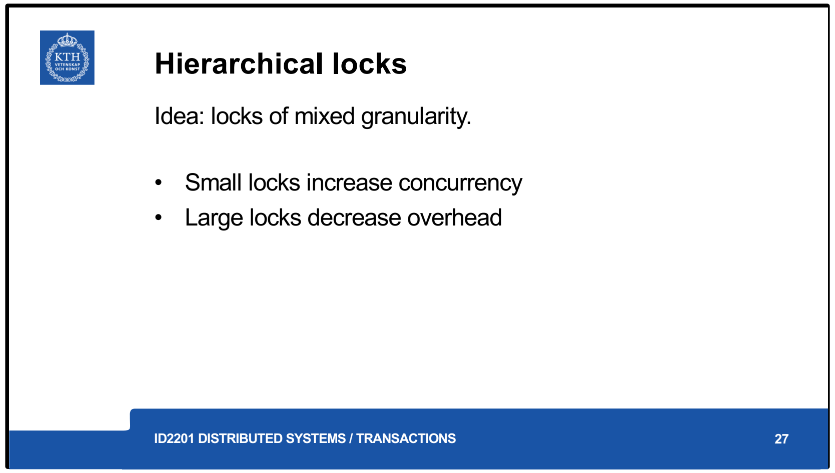
Two-version locking

Similar idea but now with read, write and commit locks.

- A read lock is allowed unless a commit lock is taken.
- One write lock is allowed if no commit lock is taken (i.e. even if read locks are taken)
- Written values are held local to the transaction and are not visible before commit.
- A write lock can be promoted to a commit lock if there are no read locks.
- When a transaction commits it tries to promote write locks to commit locks.

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 26

This is an optimistic scheme that allows one txn to write tentative versions of objects while other txns read from the committed versions of the same objects. Reads only wait if another txn is currently committing the same object. This scheme allows more concurrency than read-write locks but writing txns risk waiting or even rejection when they attempt to commit. Txns cannot commit their writes immediately if other uncompleted txns have read the same objects. Therefore, txns that request to commit in such situation wait until the reading txns have completed. Deadlocks may occur when txns are waiting to commit. Therefore, txns may need to be aborted when they are waiting to commit, to resolve deadlocks. **This variation on strict two-phase locking uses three types of lock: a read lock, a write lock and a commit lock.** When the txn coordinator receives a request to commit a txn, it attempts to convert all that transaction's write locks to commit locks. If any of the objects have outstanding read locks, the txn must wait until the txns that set these locks have completed and the locks are released.

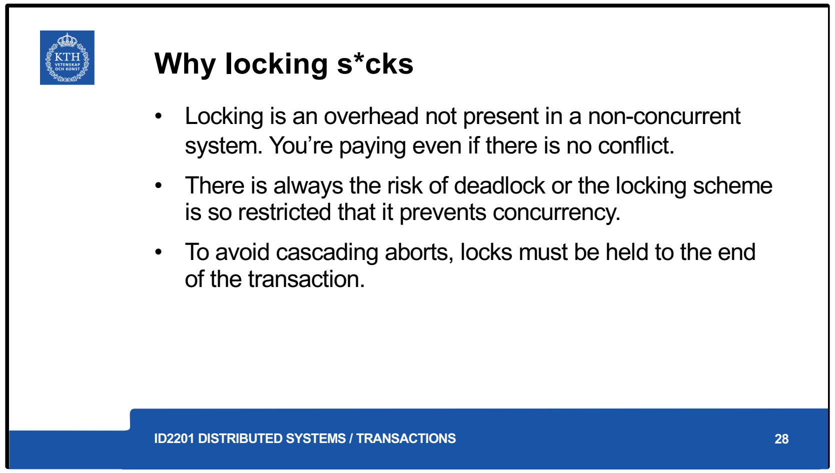

Hierarchical locks

Idea: locks of mixed granularity.

- Small locks increase concurrency
- Large locks decrease overhead

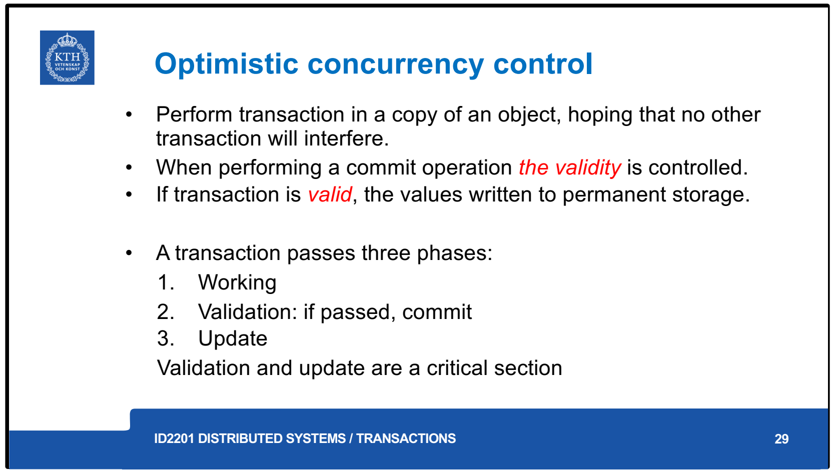
ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 27

the use of a ***hierarchy of locks with different granularities***. At each level, the setting of ***a parent lock*** has the same effect as setting ***all the equivalent child locks***. This economizes on the number of locks to be set. In our banking ***example***, ***the branch is the parent and the accounts are children***



Why locking s*cks

- Locking is an overhead not present in a non-concurrent system. You’re paying even if there is no conflict.
- There is always the risk of deadlock or the locking scheme is so restricted that it prevents concurrency.
- To avoid cascading aborts, locks must be held to the end of the transaction.



Optimistic concurrency control

- Perform transaction in a copy of an object, hoping that no other transaction will interfere.
- When performing a commit operation *the validity* is controlled.
- If transaction is *valid*, the values written to permanent storage.

- A transaction passes three phases:
  1. Working
  2. Validation: if passed, commit
  3. Update

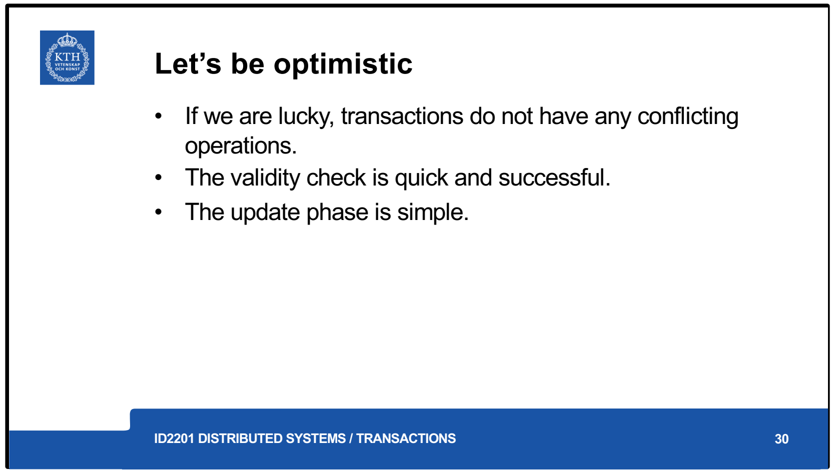
  Validation and update are a critical section

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 29

based on the observation that, in most applications, the likelihood of two clients' transactions accessing the same object is low.

During the ***working phase***, each transaction has a ***tentative version*** of each of the objects that it updates. When there are several concurrent transactions, several different tentative values of the same object may coexist.

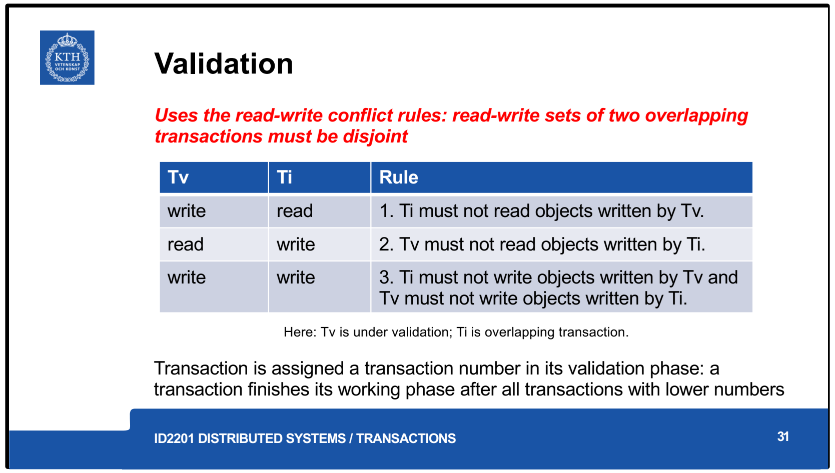
Update phase: If a transaction is validated, all of the changes recorded in its tentative versions are made permanent. Read-only transactions can commit immediately after passing validation. Write transactions are ready to commit once the tentative versions of the objects have been recorded in permanent storage.



Let's be optimistic

- If we are lucky, transactions do not have any conflicting operations.
- The validity check is quick and successful.
- The update phase is simple.

Validation uses the read-write conflict rules to ensure that the scheduling of a particular transaction is serially equivalent with respect to all other overlapping transactions – that is, any transactions that had not yet committed at the time this transaction started. To assist in performing validation, each transaction is assigned a transaction number when it enters the validation phase (that is, when the client issues a closeTransaction). If the transaction is validated and completes successfully, it retains this number; if it fails the validation checks and is aborted, or if the transaction is read only, the number is released for reassignment. Transaction numbers are integers assigned in ascending sequence; the number of a transaction therefore defines its position in time – a transaction always finishes its working phase after all transactions with lower numbers.



Validation

Uses the read-write conflict rules: read-write sets of two overlapping transactions must be disjoint

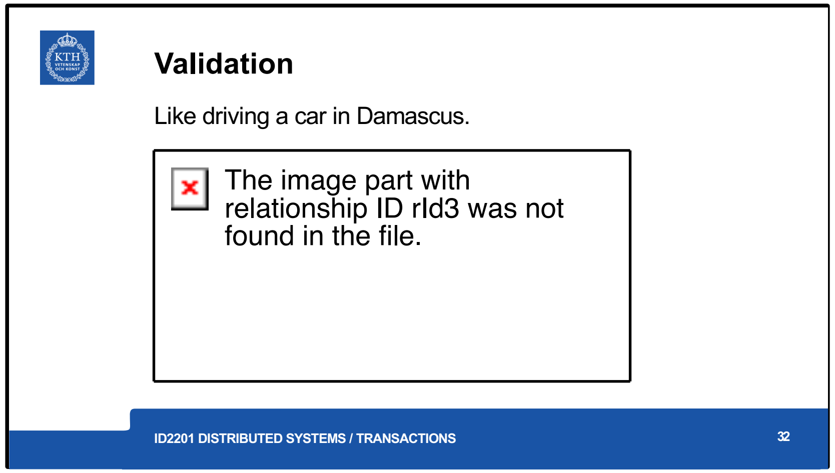
Tv	Ti	Rule
write	read	1. Ti must not read objects written by Tv.
read	write	2. Tv must not read objects written by Ti.
write	write	3. Ti must not write objects written by Tv and Tv must not write objects written by Ti.

Here: Tv is under validation; Ti is overlapping transaction.

Transaction is assigned a transaction number in its validation phase: a transaction finishes its working phase after all transactions with lower numbers

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 31

Tv is under validation; Ti is overlapping transaction.
Each transaction is assigned a transaction number when it enters the validation phase (that is, when the client issues a closeTransaction).
If the transaction number were to be assigned at the beginning of the working phase, then a transaction that reached the end of the working phase before one with a lower number would have to wait until the earlier one had completed before it could be validated.
As the validation and update phases of a transaction are generally short in duration compared with the working phase, a simplification can be achieved by ***making the rule that only one transaction may be in the validation and update phase at one time. When no two transactions may overlap in the update phase, rule 3 is satisfied.***



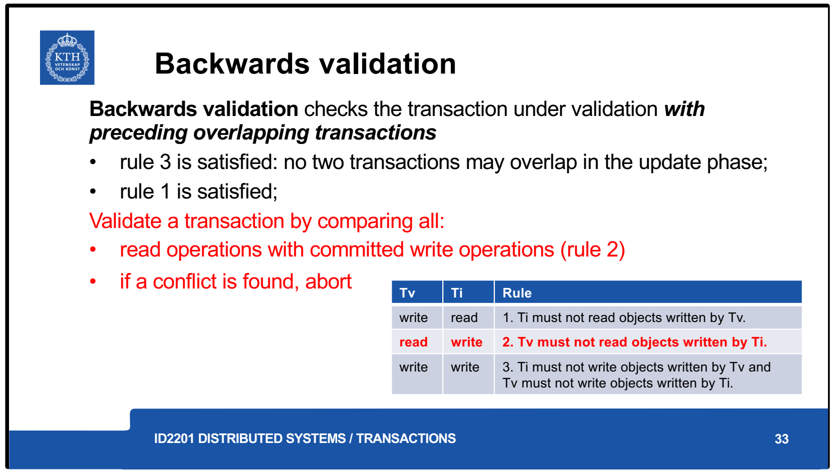
Validation

Like driving a car in Damascus.

The image part with relationship ID rId3 was not found in the file.

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 32

As the validation and update phases of a transaction are generally short in duration compared with the working phase, a simplification can be achieved by ***making the rule that only one transaction may be in the validation and update phase at one time. When no two transactions may overlap in the update phase, rule 3 is satisfied.***
The validation of a transaction must ensure that rules 1 and 2 are obeyed by testing for overlaps between the objects of pairs of transactions Tv and Ti**. There are two forms of validation – backward and forward. Backward validation** checks the transaction undergoing validation **with other preceding overlapping transactions** – those that entered the validation phase before it. **Forward validation** checks the transaction undergoing validation **with other later transactions, which are still active.**



Backwards validation

Backwards validation checks the transaction under validation ***with preceding overlapping transactions***

- rule 3 is satisfied: no two transactions may overlap in the update phase;
- rule 1 is satisfied;

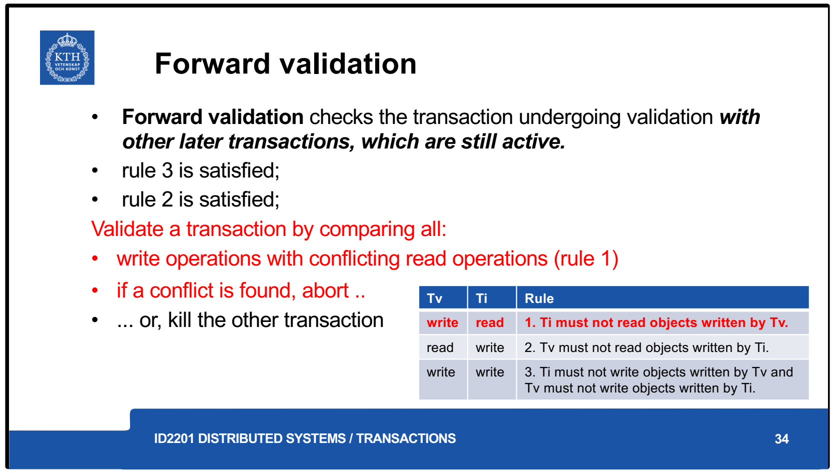
Validate a transaction by comparing all:

- read operations with committed write operations (rule 2)
- if a conflict is found, abort

Tv	Ti	Rule
write	read	1. Ti must not read objects written by Tv.
read	**write**	**2. Tv must not read objects written by Ti.**
write	write	3. Ti must not write objects written by Tv and Tv must not write objects written by Ti.

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 33

Backward validation checks the transaction undergoing validation ***with other preceding overlapping transactions, i.e. earlier overlapping transactions. Validation with past transations – backward.***
As all the *read* operations of earlier overlapping transactions were performed before the validation of *Tv* started, they cannot be affected by the *writes* of the current transaction (and **rule 1 is satisfied**). The validation of transaction *Tv* checks whether its read set (the objects affected by the *read* operations of *Tv*) overlaps with any of the write sets of earlier overlapping transactions, *Ti* (rule 2). If there is any overlap, the validation fails.



Forward validation

- **Forward validation** checks the transaction undergoing validation ***with other later transactions, which are still active.***
- rule 3 is satisfied;
- rule 2 is satisfied;

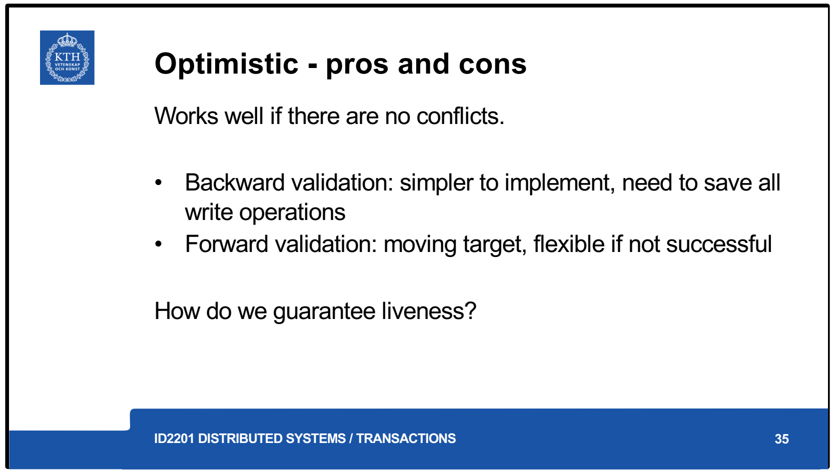
Validate a transaction by comparing all:

- write operations with conflicting read operations (rule 1)
- if a conflict is found, abort ..
- ... or, kill the other transaction

Tv	Ti	Rule
write	read	**1. Ti must not read objects written by Tv.**
read	write	2. Tv must not read objects written by Ti.
write	write	3. Ti must not write objects written by Tv and Tv must not write objects written by Ti.

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 34

Forward validation checks the transaction undergoing validation ***with other later transactions, which are still active. Validation with future transactions – forward.***
In forward validation of the transaction Tv, the write set of Tv is compared with the read sets of all overlapping active transactions – those that are still in their working phase (rule 1). Rule 2 is automatically fulfilled because the active transactions do not write until after Tv has completed.


Optimistic - pros and cons

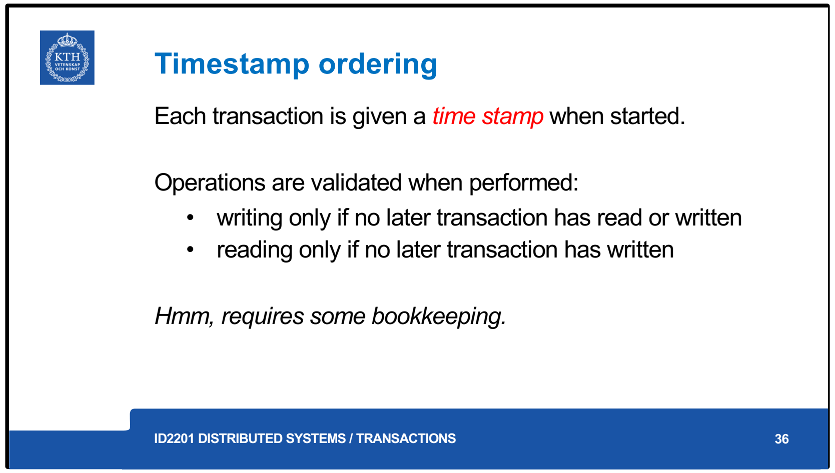
Works well if there are no conflicts.

- Backward validation: simpler to implement, need to save all write operations
- Forward validation: moving target, flexible if not successful

How do we guarantee liveness?

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 35

Backward: check my reads with committed writes. **Forward**: check later reads with my (uncommitted) writes. Moving target: read set may changed; Flexible: defer validation until a later time; abort; or kill the other transactions. **Liveness:** When a transaction is aborted, it will normally be restarted by the client program. But in schemes that rely on aborting and restarting transactions, there is no guarantee that a particular transaction will ever pass the validation checks, for it may come into conflict with other transactions for the use of objects each time it is restarted. **The prevention of a transaction ever being able to commit is called *starvation*.** Occurrences of starvation are likely to be rare, but a server that uses optimistic concurrency control must ensure that a client does not have its transaction aborted repeatedly. Kung and Robinson suggest that this could be done if the server detects a transaction that has been aborted several times. They suggest that when the server detects such a transaction it should be given exclusive access by the use of a critical section protected by a semaphore.



Timestamp ordering

Each transaction is given a *time stamp* when started.

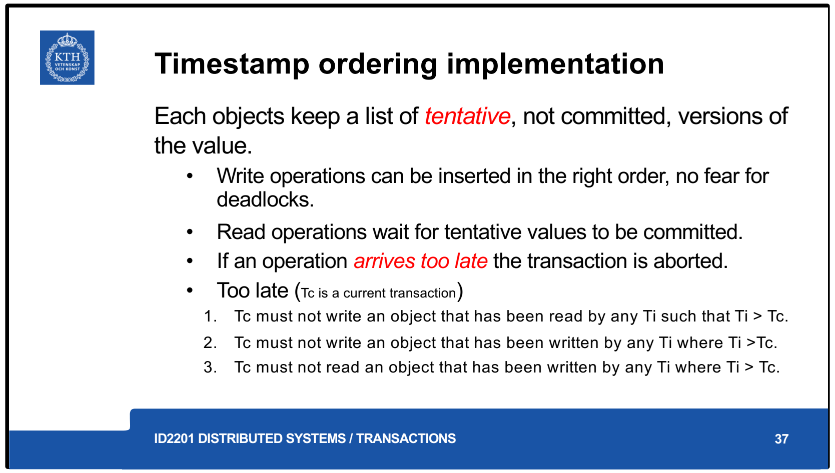
Operations are validated when performed:

- writing only if no later transaction has read or written
- reading only if no later transaction has written

Hmm, requires some bookkeeping.

In concurrency control schemes based on timestamp ordering, each operation in a transaction is validated when it is carried out. If the operation cannot be validated, the transaction is aborted immediately and can then be restarted by the client. Each transaction is assigned a unique timestamp value when it starts. The timestamp defines its position in the time sequence of transactions. Requests from transactions can be totally ordered according to their timestamps. The basic timestamp ordering rule is based on operation conflicts and is very simple: **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 transaction. This rule assumes that there is only one version of each object and restricts access to one transaction at a time.



Timestamp ordering implementation

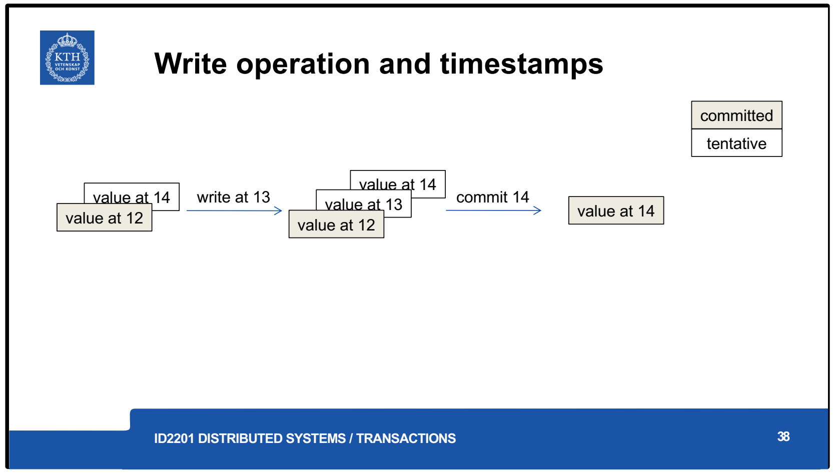
Each objects keep a list of *tentative*, not committed, versions of the value.

- Write operations can be inserted in the right order, no fear for deadlocks.
- Read operations wait for tentative values to be committed.
- If an operation *arrives too late* the transaction is aborted.
- Too late (Tc is a current transaction)
  1. Tc must not write an object that has been read by any Ti such that Ti > Tc.
  2. Tc must not write an object that has been written by any Ti where Ti >Tc.
  3. Tc must not read an object that has been written by any Ti where Ti > Tc.

Tc is current; Ti is later than Tc

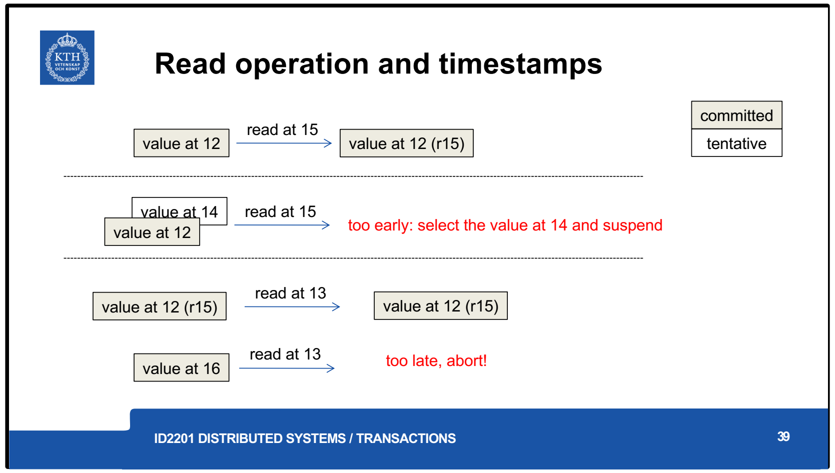
Too late:

1) Tc must not write an object that has been read by any Ti such that Ti > Tc. (This requires that Tc >= the maximum read timestamp of the object.)
2) Tc must not write an object that has been written by any Ti where Ti >Tc. (This requires that Tc > the write timestamp of the committed object.)
3) Tc must not read an object that has been written by any Ti where Ti > Tc. (This requires that Tc > the write timestamp of the committed object.)


Write operation and timestamps

Too late:

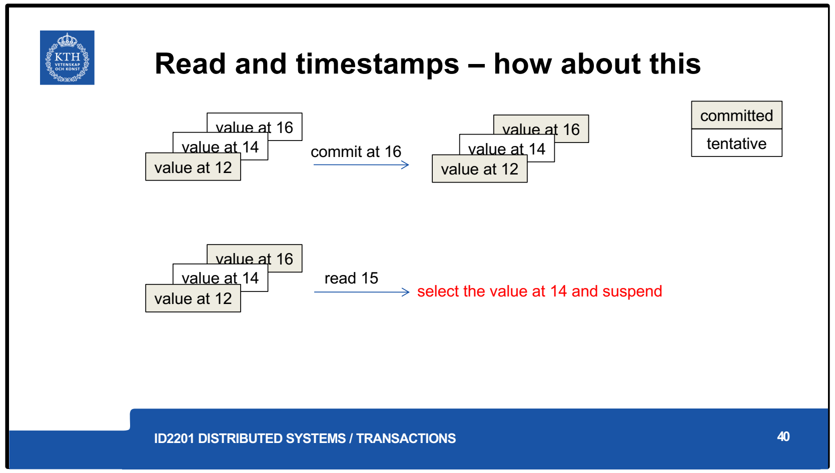
1) Tc must not write an object that has been read by any Ti such that Ti > Tc. (This requires that Tc >= the maximum read timestamp of the object.)
2) Tc must not write an object that has been written by any Ti where Ti >Tc. (This requires that Tc > the write timestamp of the committed object.)
3) Tc must not read an object that has been written by any Ti where Ti > Tc. (This requires that Tc > the write timestamp of the committed object.)


Read operation and timestamps

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 39

Value at 16 → read at 13 – too late, abort!
Too late:

1) Tc must not write an object that has been read by any Ti such that Ti > Tc. (This requires that Tc >= the maximum read timestamp of the object.)
2) Tc must not write an object that has been written by any Ti where Ti >Tc. (This requires that Tc > the write timestamp of the committed object.)
3) Tc must not read an object that has been written by any Ti where Ti > Tc. (This requires that Tc > the write timestamp of the committed object.)



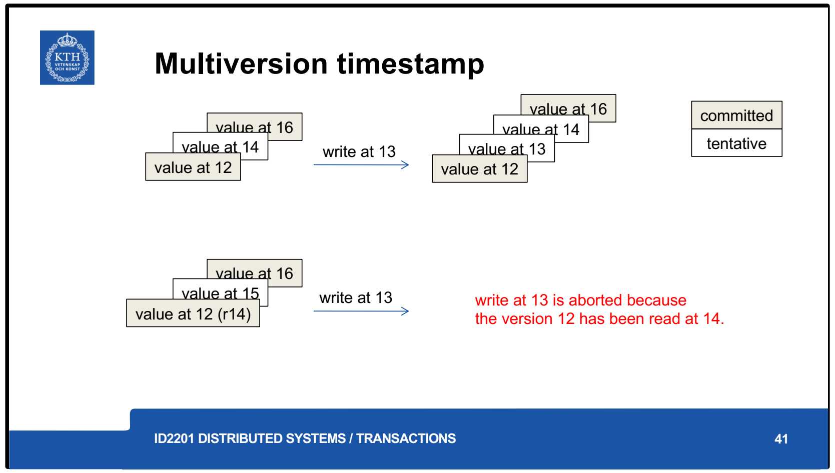
Read and timestamps – how about this

ID2201 DISTRIBUTED SYSTEMS / TRANSACTIONS 40

Value at 16 → read at 13 – too late, abort!

Too late:

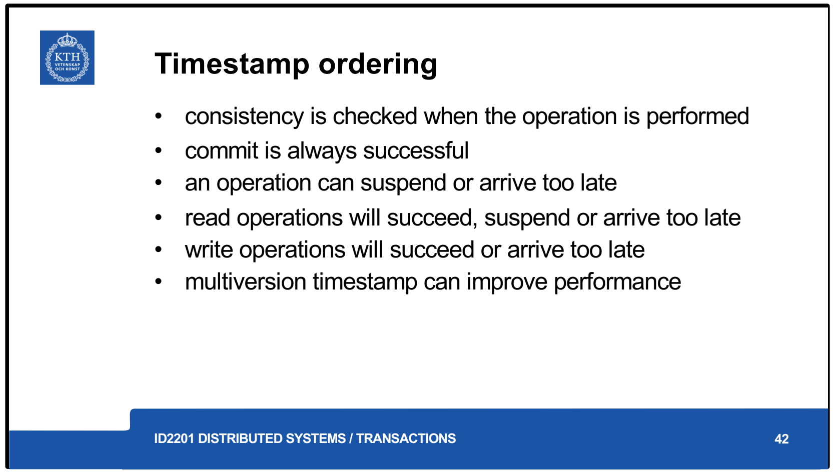
1) Tc must not write an object that has been read by any Ti such that Ti > Tc. (This requires that Tc >= the maximum read timestamp of the object.)
2) Tc must not write an object that has been written by any Ti where Ti >Tc. (This requires that Tc > the write timestamp of the committed object.)
3) Tc must not read an object that has been written by any Ti where Ti > Tc. (This requires that Tc > the write timestamp of the committed object.)



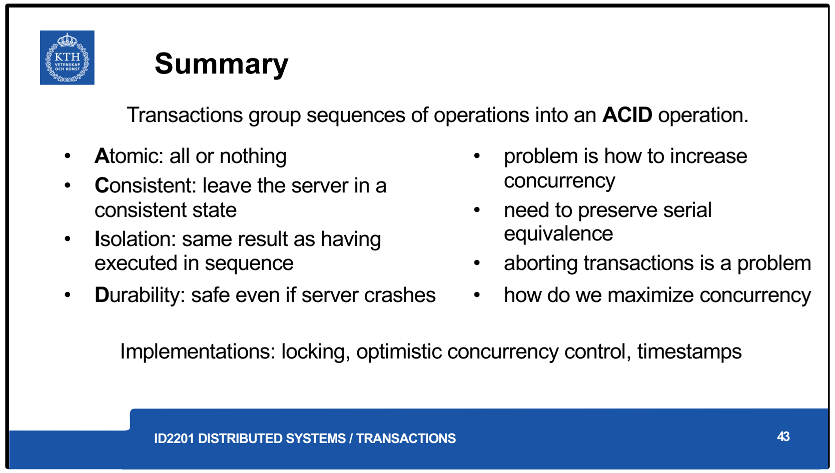
Multiversion timestamp

In multiversion timestamp ordering, a list of old committed versions as well as tentative versions is kept for each object. This list represents the history of the values of the object. **The benefit of using multiple versions is that read operations that arrive too late need not be rejected.** Each version has a read timestamp recording the largest timestamp of any transaction that has read from it in addition to a write timestamp. As before, whenever a write operation is accepted, it is directed to a tentative version with the write timestamp of the transaction. Whenever a read operation is carried out, it is directed to the version with the largest write timestamp less than the transaction timestamp. If the transaction timestamp is larger than the read timestamp of the version being used, the read timestamp of the version is set to the transaction timestamp. **When a read arrives late, it can be allowed to read from an old committed version, so there is no need to abort late read operations. In multiversion timestamp ordering, read operations are always permitted, although they may have to wait for earlier transactions to complete**

(either commit or abort), which ensures that executions are recoverable. There is no conflict between write operations of different transactions, because each transaction writes its own committed version of the objects it accesses. This removes rule 2 in the conflict rules for timestamp ordering.
In the last case: write at 13 should be aborted becasue the version 12 has been read at 14.


Timestamp ordering

- consistency is checked when the operation is performed
- commit is always successful
- an operation can suspend or arrive too late
- read operations will succeed, suspend or arrive too late
- write operations will succeed or arrive too late
- multiversion timestamp can improve performance



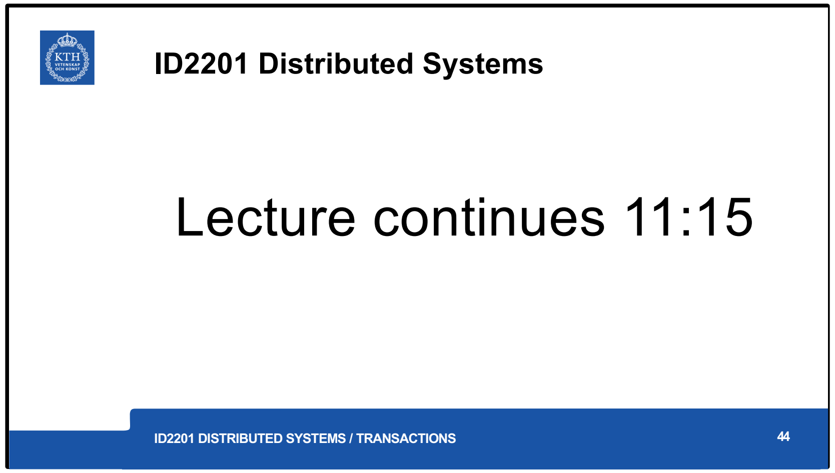
Summary

Transactions group sequences of operations into an **ACID** operation.

- **A**tomic: all or nothing
- **C**onsistent: leave the server in a consistent state
- **I**solation: same result as having executed in sequence
- **D**urability: safe even if server crashes

- problem is how to increase concurrency
- need to preserve serial equivalence
- aborting transactions is a problem
- how do we maximize concurrency

Implementations: locking, optimistic concurrency control, timestamps



ID2201 Distributed Systems

Lecture continues 11:15