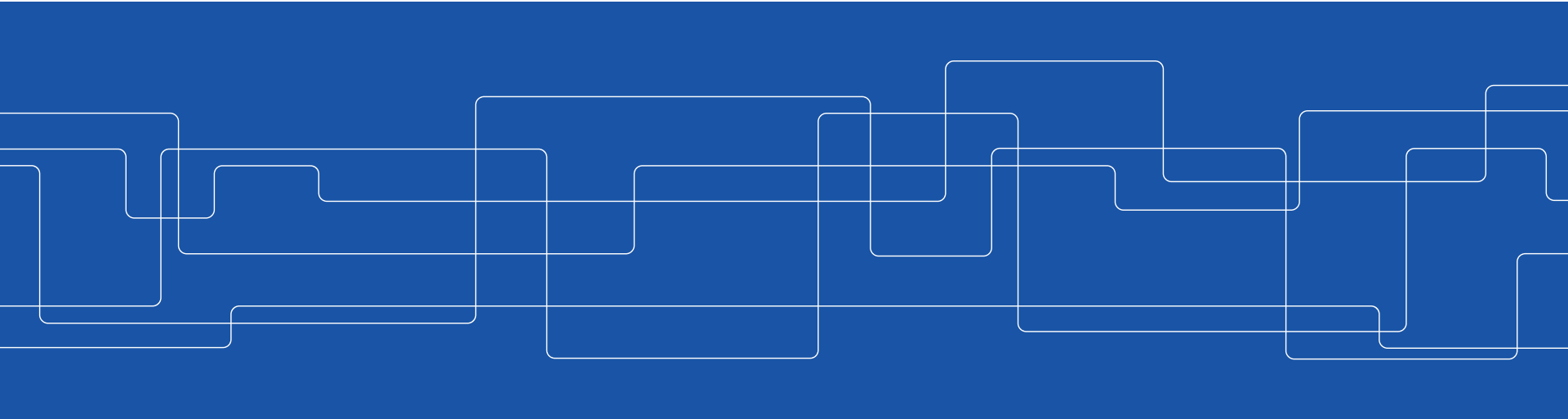

KTH ROYAL INSTITUTE
OF TECHNOLOGY

# 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?*

# 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()
- withdraw(amount)
- deposit(amount)

# Lost update

client a

```
bal = b.getBalance()
```



```
b.setBalance(bal*1.1)
```



```
a.withdraw(bal*0.1)
```

client c

```
bal = b.getBalance()
```

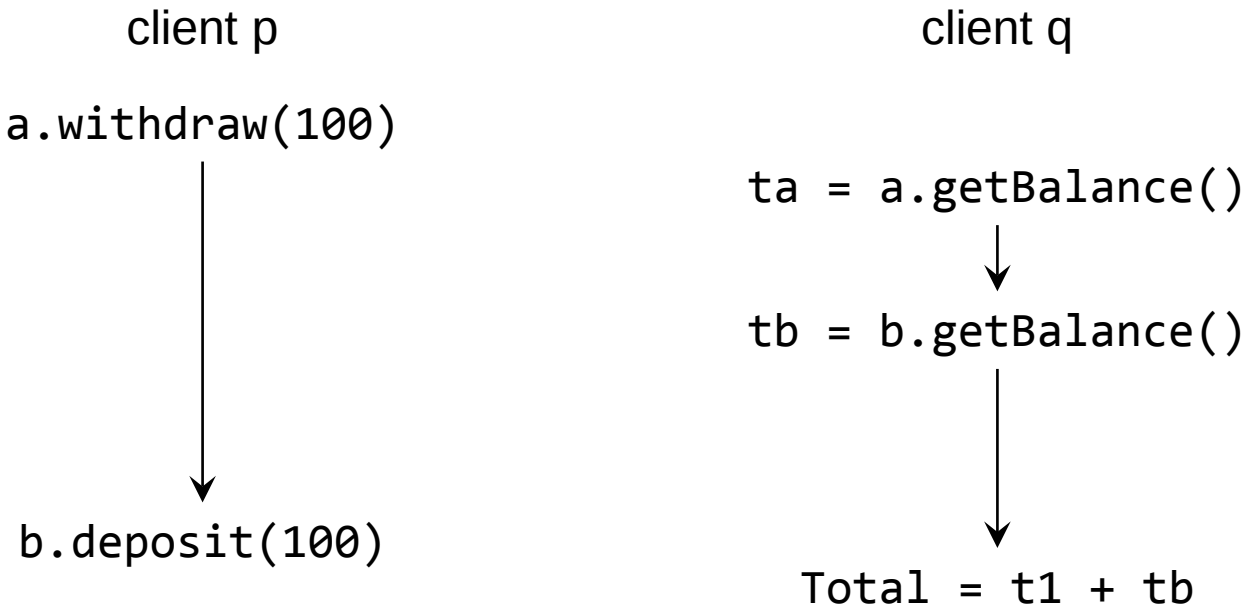


```
b.setBalance(bal*1.1)
```



```
c.withdraw(bal*0.1)
```

# Inconsistent retrieval



# 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?*

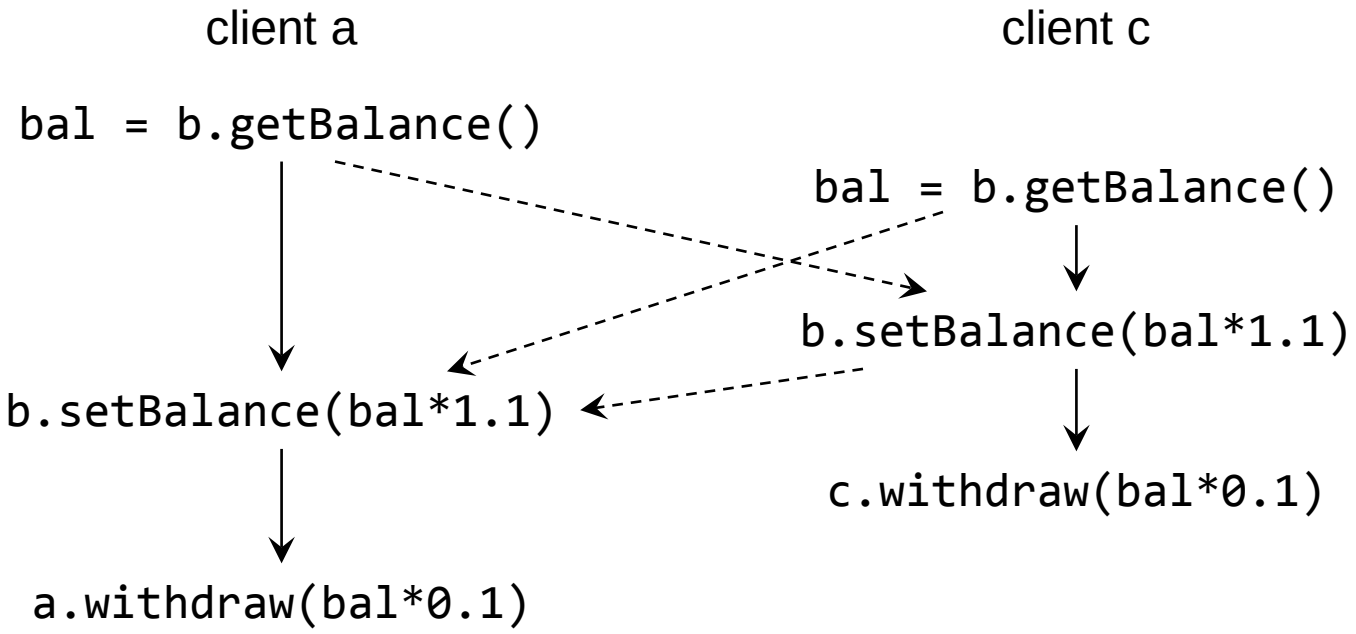
# Conflicting operations

Which operations are order sensitive?

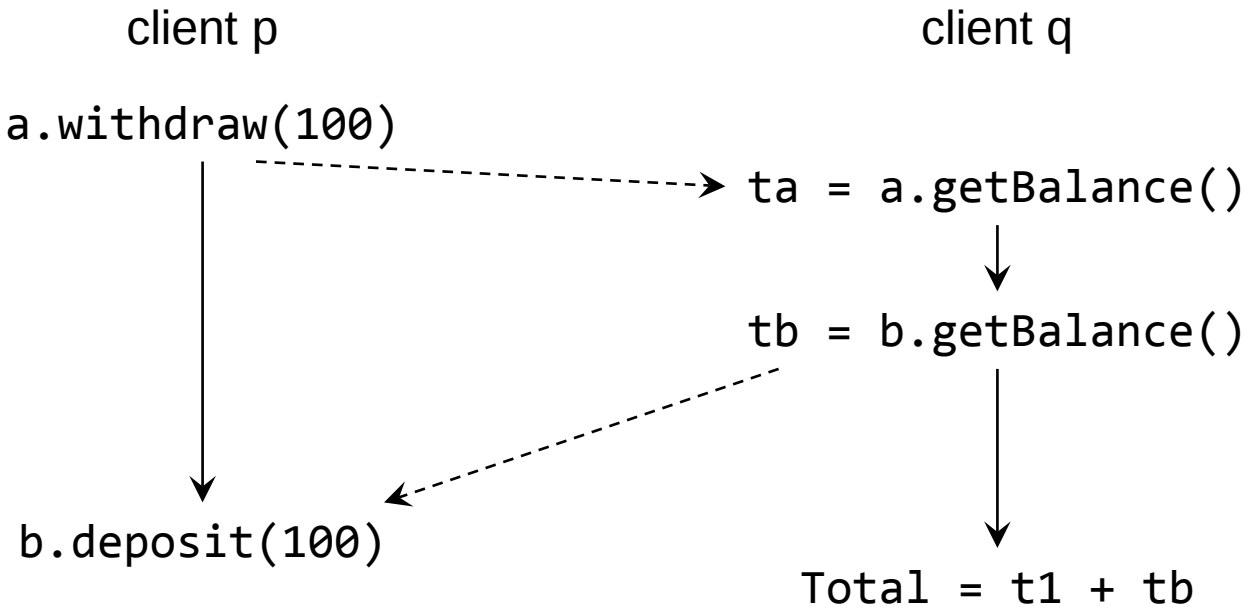
- read – read
- read – write
- write - write

Two transactions are *serially equivalent* if, and only if, all pair of conflicting operations of the transactions are executed in *the same order*.

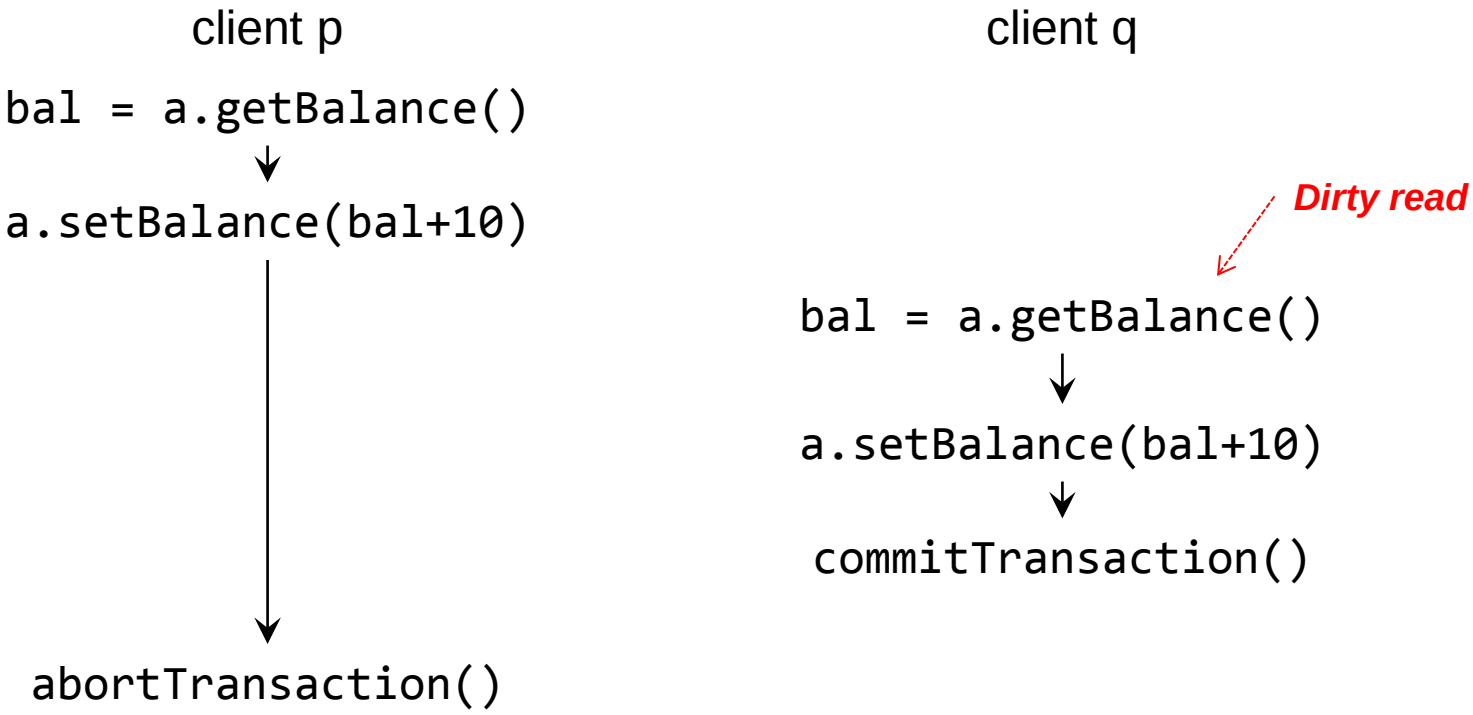
# Lost update - revisited



# Inconsistent retrieval - revisited



# More problems



# Recoverability from aborts

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

# 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.

...

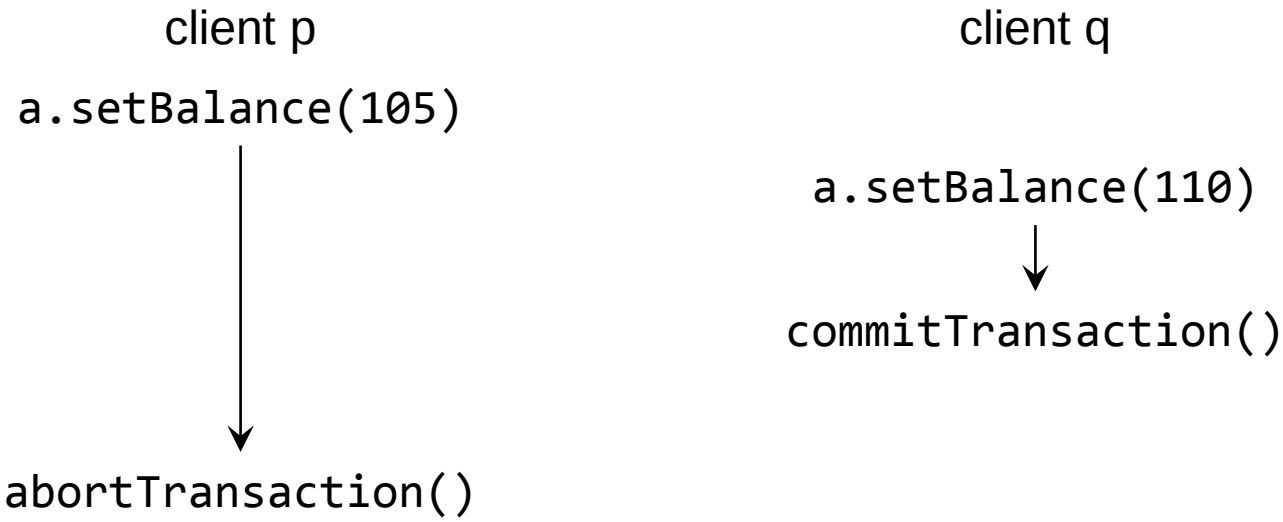
We abort

*In order to avoid cascading aborts we should suspend when we read a dirty value.*

# 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




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

# 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?

# 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

# 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 lock if a lock has been released.

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

# Strict two-phase locking

To handle dirty read and premature write:

- lock in any order
- commit or abort
- unlock

Can we increase concurrency?

# Read and write locks

- two-version locking: read and write
- allow multiple readers but only one writer
- promote read locks to write locks, i.e. convert a lock to a stronger lock
- strict two-phase locking prevents demotion

# 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.

# Hierarchical locks

Idea: locks of mixed granularity.

- Small locks increase concurrency
- Large locks decrease overhead

# 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

# 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

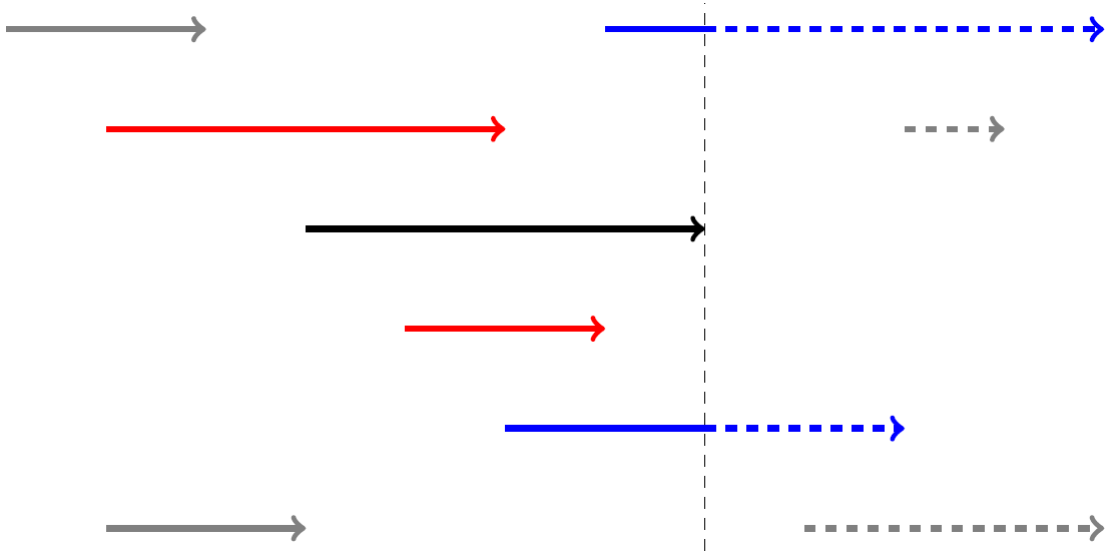
Uses the read-write conflict rules: read-write sets of two overlapping transactions must be disjoint

| 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. |

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

# Validation

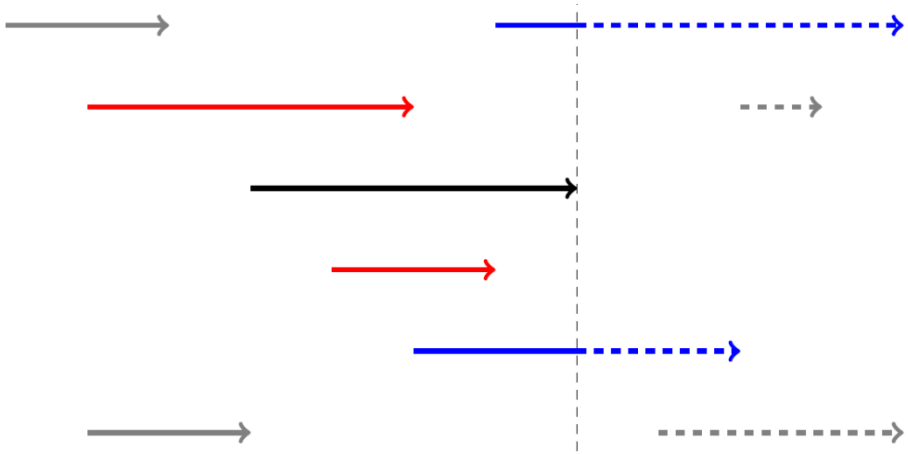
Like driving a car in Damascus.



# Backwards validation

Validate a transaction by comparing all:

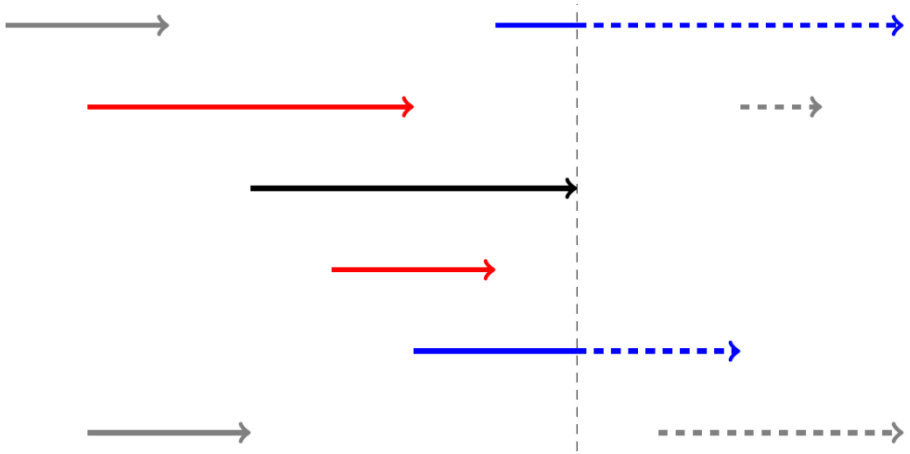
- read operations with committed write operations
- if a conflict is found, abort



# Forward validation

Validate a transaction by comparing all:

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



# 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?

# 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

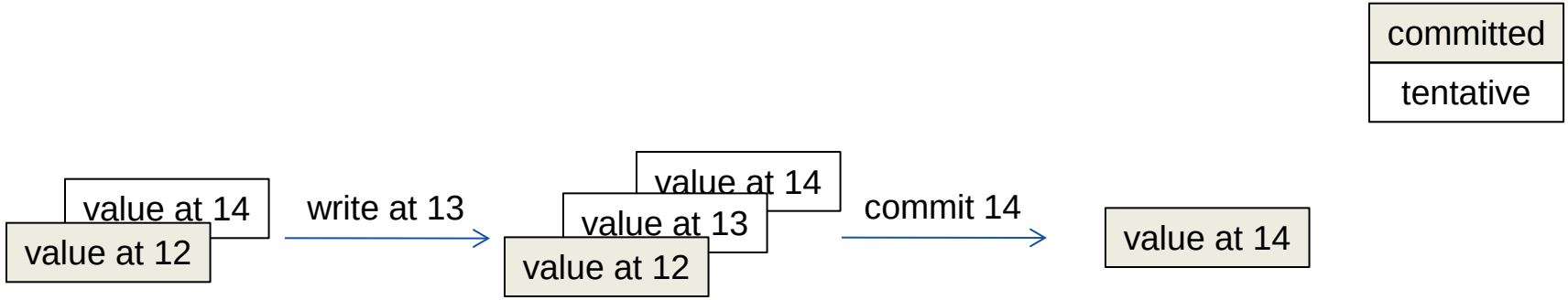
*Hmm, requires some bookkeeping.*

# Timestamp ordering implementation

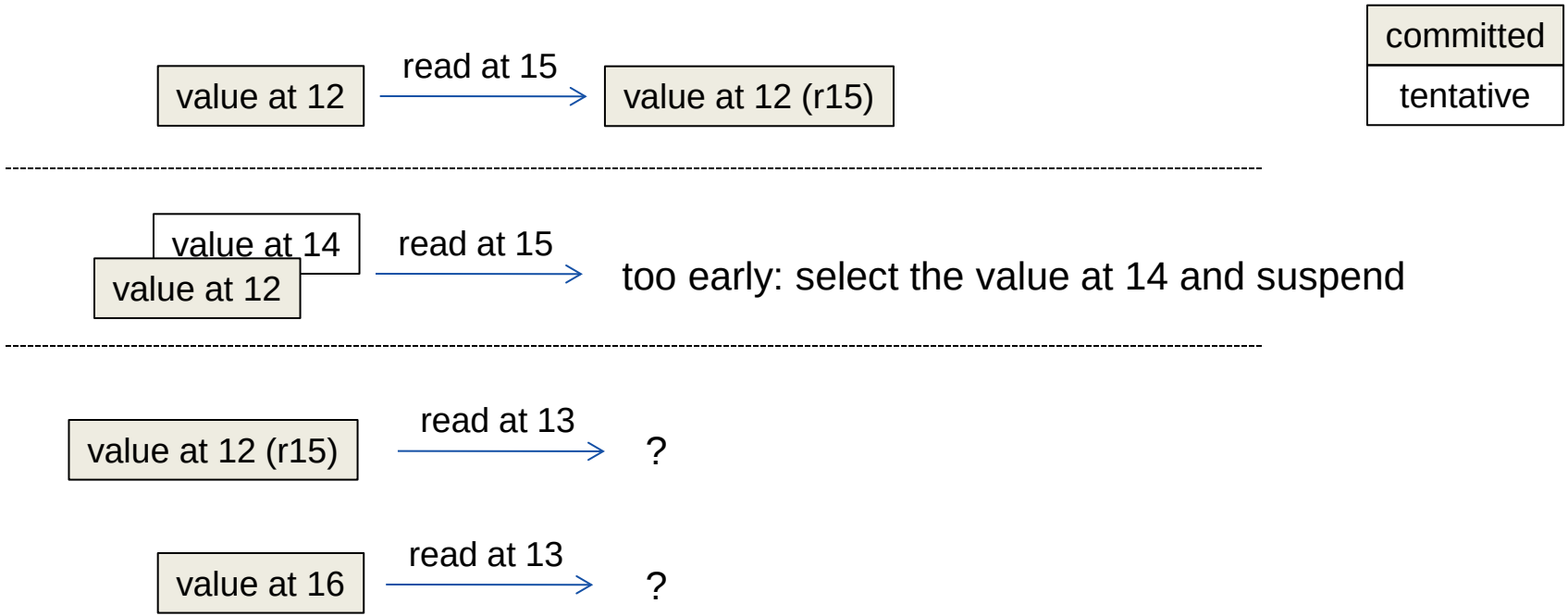
Each objects keep a list of *tentative*, not committed, versions of the value.

- 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.

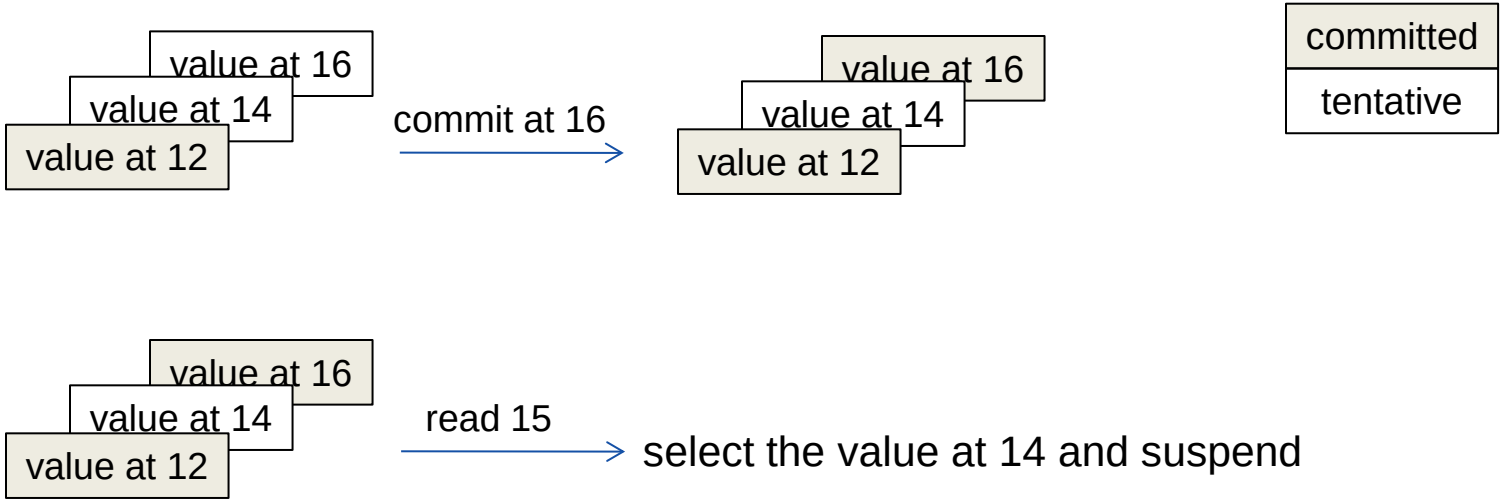
# Write operation and timestamps



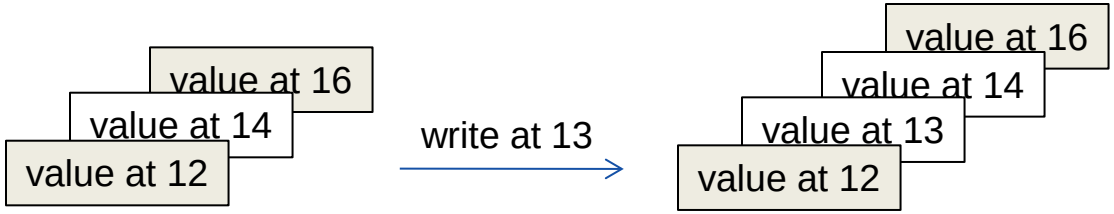
# Read operation and timestamps

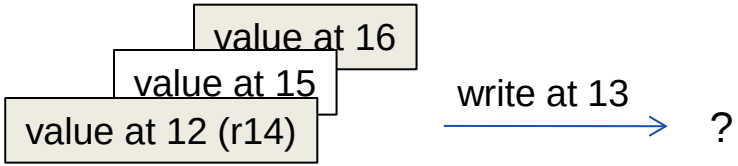


# Read and timestamps – how about this



# Multiversion timestamp



# 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 a ACID operation.

- Atomic: all or nothing
- Consistent: leave the server in a consistent state
- Isolation: same result as having executed in sequence
- Durability: 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