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(amount)
- 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

client P

```
a.withdraw(100)

b.deposit(100)
```

client Q

```
ta = a.getBalance()

tb = b.getBalance()

Total = ta + tb
```

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 pairs of conflicting operations* of the two transactions are executed in *the same order on all the objects they both access*.

Lost update - revisited

	client A	client C	
read(b)	`bal = b.getBalance()`		
		`bal = b.getBalance()`	read(b)
		`b.setBalance(bal*1.1)`	write(b)
write(b)	`b.setBalance(bal*1.1)`		
		`c.withdraw(bal*0.1)`	
	`a.withdraw(bal*0.1)`		

Conflicting ops

The same order of conflicting ops

Lost update - revisited

client A

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

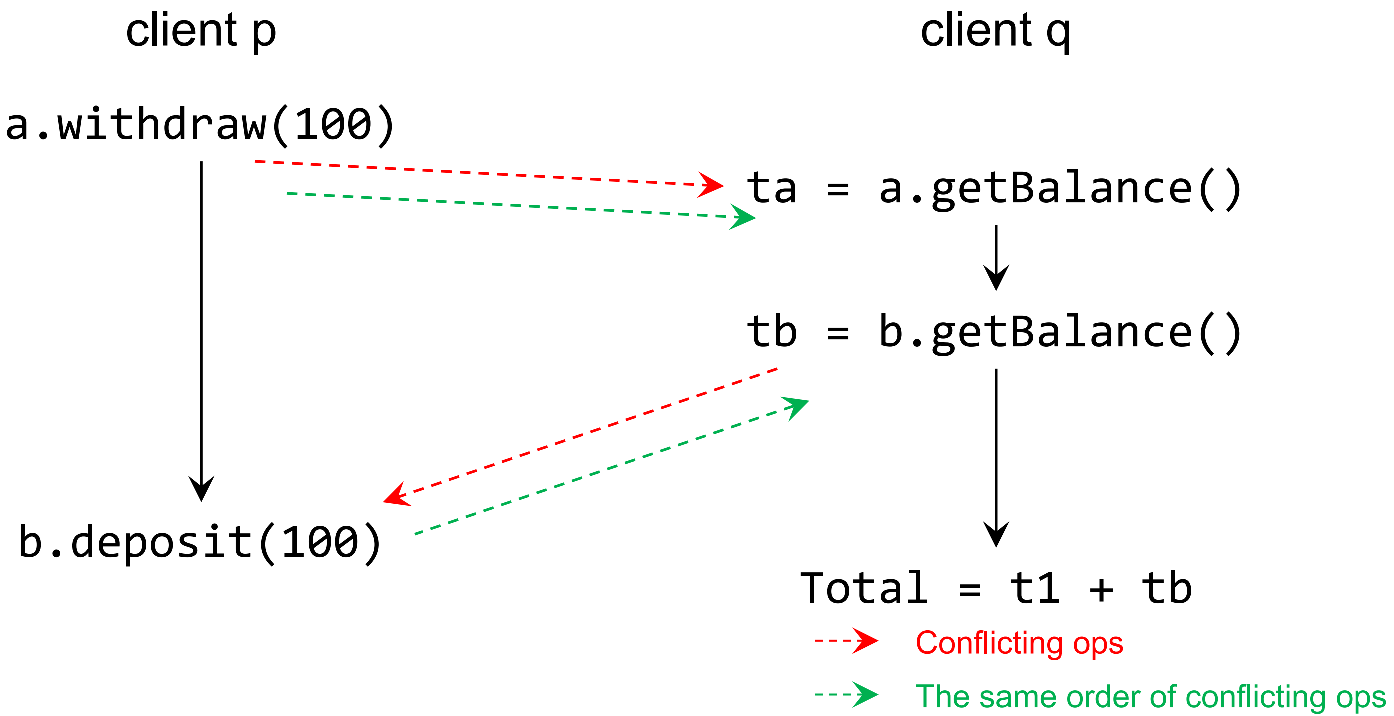
client C

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

Conflicting ops

The same order of conflicting ops

Inconsistent retrieval - revisited



More problems: Dirty read

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

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 (before) 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

client p

`a.setBalance(105)`

`abortTransaction()`

client q

`a.setBalance(110)`

`commitTransaction()`

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 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)*

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
  - 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

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.

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

Validation

Like driving a car in Damascus.

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

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.

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.

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.

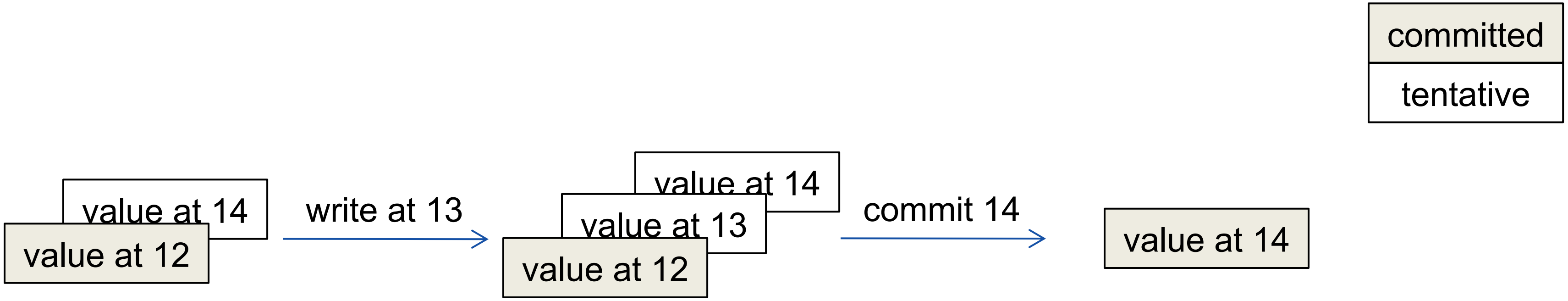
Timestamp ordering implementation

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.
- 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.

Write operation and timestamps



Read operation and timestamps

committed

tentative

value at 12 $\xrightarrow{\text{read at 15}}$ value at 12 (r15)

---

value at 14

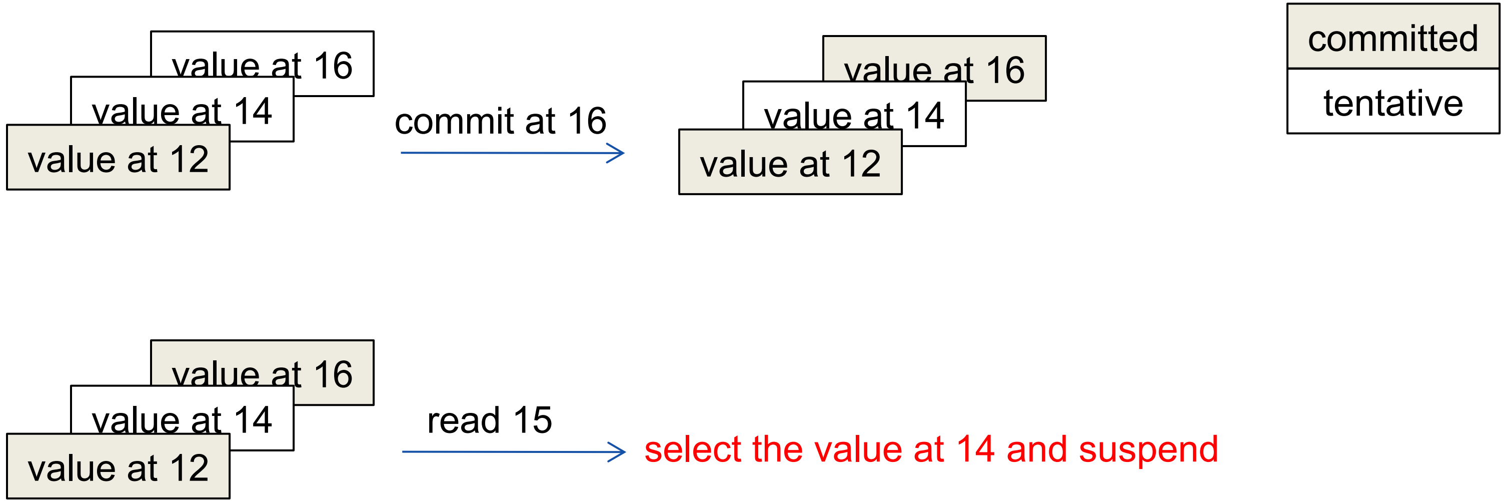
value at 12 $\xrightarrow{\text{read at 15}}$ too early: select the value at 14 and suspend

---

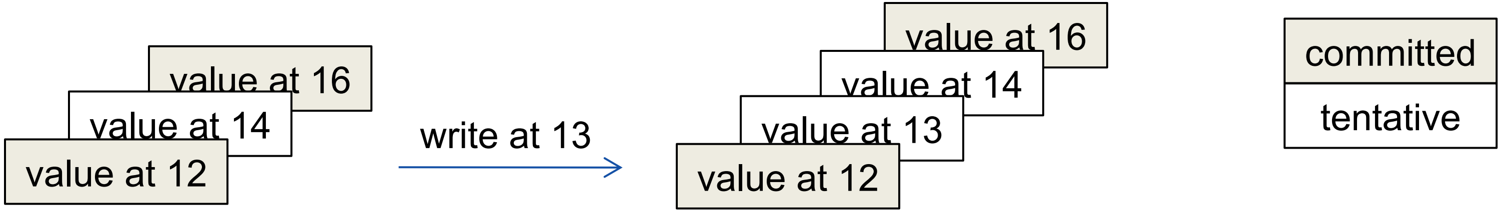
value at 12 (r15) $\xrightarrow{\text{read at 13}}$ value at 12 (r15)

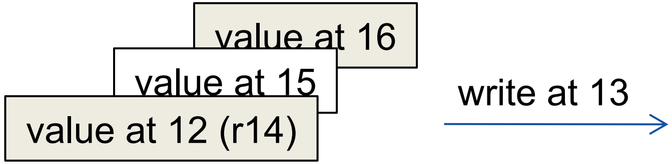
value at 16 $\xrightarrow{\text{read at 13}}$ too late, abort!

Read and timestamps – how about this



Multiversion timestamp





write at 13 is aborted because
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