Distributed Systems

ID2201

transactions

Johan Montelius

The problem

- 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 crashes

Surviving a crash

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

Failure model

- Permanent storage:
  - omission failures
  - writing the wrong value
  - *but writing to the right location*
- Servers crash:
  - restarted using persistent storage only
- Network:
  - asynchronous
  - omission failures
  - duplicate messages

Requirements - ACID

- Atomic
  - either all or nothing
- Consistent
  - this is an application concern
- Isolation
  - intermediate effects of a transaction are not visible to other transactions
- Durability
  - persistent once acknowledged

The solution - not

- All requirements can be achieved by only allowing sequential access to the transaction server.
  - severe restriction
- Our goal is to provide as much concurrency as possible while preserving the behavior of sequential access.

The solution - not

- Only have one server with persistent storage, if it crashes we only have to wait for it to restart.
  - for how long must we wait
- Our goal is to replicate the server to provide resilience.

Transaction API

- openTransaction() :
  - returns a *transaction identifier*
- closeTransaction(*tid*) :
  - returns success or failure of transaction
- abortTransaction(*tid*) :
  - client explicitly aborts transaction
- *operation*(tid, arg) :
  - operations that belong to a transaction
  - read, write, append, deposit, ...
  - *we will write operations with implicit tid*

Bank transaction examples

- Operations
  - getBalance(account)
  - setBalance(account)
  - withdraw(account, amount)
  - deposit(account, amount)

Lost update

```
bal = getBalance(b);



setBalance(b, bal*1.1);

withdraw(a, bal*0.1);
```

```

bal = getBalance(b);
setBalance(b, bal*1.1);

withdraw(c, bal*0.1);
```

Inconsistent retrievals

```
withdraw(a,100);




deposit(b,100);
```

```
ta = getBalance(a);
tb = getBalance(b);

Total = ta + tb;
```

Conflicting operations

- Which operations are order sensitive?
  - read – read
  - read - write
  - write – write
- Two *transactions* are *serially equivalent* iff all pair of conflicting operations of the transactions are executed *in the same order*.

Lost update revisited

```
bal = getBalance(b);

setBalance(b, bal*1.1);

withdraw(a, bal*0.1);
```

```
bal = getBalance(b);

setBalance(b, bal*1.1);

withdraw(c, bal*0.1);
```

Lost update revisited

```
bal = getBalance(b);

setBalance(b, bal*1.1);



withdraw(a, bal*0.1);
```

```



bal = getBalance(b);

setBalance(b, bal*1.1);

withdraw(c, bal*0.1);
```

Inconsistent retrievals revisited

```
withdraw(a,100);



deposit(b,100);
```

```
ta = getBalance(a);
tb = getBalance(b);

Total = ta + tb;
```

Inconsistent retrievals revisited

```
withdraw(a,100);

deposit(b,100);
```

```
ta = getBalance(a);

tb = getBalance(b);

Total = ta + tb;
```

Problems with abort

- Even if our operations are done in a serial equivalent order the isolation requirement can be violated.

```
bal = getBalance(a);
setBalance(a, bal +10);




abortTransaction();
```

```


bal = getBalance(a);
setBalance(a, bal +10);

commitTransaction();
```

Dirty read

- To be *recoverable* a transaction must suspend its commit operation if it has performed a *dirty read*.
- If a transaction abort, 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.
  - This might be a bit too strong.
  - How dangerous is cascading abort?

Premature writes

- Similar problem with write operations. How do we recover?
- Write operations must be delayed.

```
setBalance(a,105);



abortTransaction();
```

```
setBalance(a,110);

commitTransaction();
```

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

- 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?

Increase concurrency

- Two-version locking
  - read, write and commit locks
- Hierarchical locks
  - smaller locks increase concurrency but increase overhead
  - structure locks in a hierarchy, taking a higher lock prevents someone from taking any lock in the group

Read and write locks

- Read operations do not have to be serialized.
- Use different locks for read and write access
- Multiple transactions can take read locks but only if the write lock is not taken.
- Only one transaction can take a write lock but only if the read lock is not taken.
- Read locks can be *promoted* to write locks
  - why not release and take?

Deadlock

- The obvious danger when using locks is to land in a deadlock situation.

```
deposit(a, 100);

withdraw(b, 100);

commit;
```

```
deposit(b, 200);

withdraw(a, 100);
```

Handle deadlock

- Prevention
  - take locks *all at once* in advance or
  - in *predefined order*
  - reduces concurrency!
- Detection
  - check for cyclic dependencies as a lock is taken
  - large overhead
  - which lock should be removed?

Handle deadlock

- Timeout
  - A taken lock is made *vulnerable* after a timeout.
  - If other transactions are waiting the lock must be *released*, this normally results in a aborted transaction.
  - Timeout can be a result of overload, aborted transactions will increase load.

Why locking s*ks

- 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 control

- Perform transaction in a copy of objects without locks hoping that no other transaction will interfere.
- When performing a commit operation the validity is controlled
- If transaction is valid the objects are updated and (if write operations where involved) values written to permanent storage.

Working phase

- Keeps a tentative version of each object.
- Read operations performed only if a committed value exists or if a value exists in the tentative version.
- Write operations are only visible in tentative version.

Validation phase

- A transaction will check *overlapping* transactions for conflicting operations.
  - transactions not yet committed at the start of the transaction
- A transaction is given a sequence number when entering the validation phase.
- T_v is *serializable with respect to* T_i if
  - T_v does not read what T_i wrote
  - T_i does not read what T_v wrote
  - T_v and T_i do not write the same object

Let's be optimistic

- If we are lucky, and we are, many transactions do not have any conflicts with overlapping transaction.
- Test will be quick and successful
- If successful move on to the *update-phase.*

Backward validation

- T_{start} is sequence number when transaction enters the working phase.
- T_{end} is sequence number when entering the validation phase.
- Validate a transaction by comparing all read operations with write operations of (commited) transactions with sequence number:
  - $T_{start} < T_i < T_{end}$
- if conflicting
  - abort

Forward validation

- Validate a transaction by comparing all write operations with read operations of overlapping active (uncommitted) transactions.
- Why does this work?
- if conflict
  - abort the transaction
  - abort the other transaction
  - try later... let the conflicting transaction commit, hope for the best

Optimistic pros and cons

- Works well if no conflicts.
- Backward validation
  - need to save all write operations
- Forward validation
  - flexible if not successful
  - transactions active while we do validation
- How do we guarantee liveness?

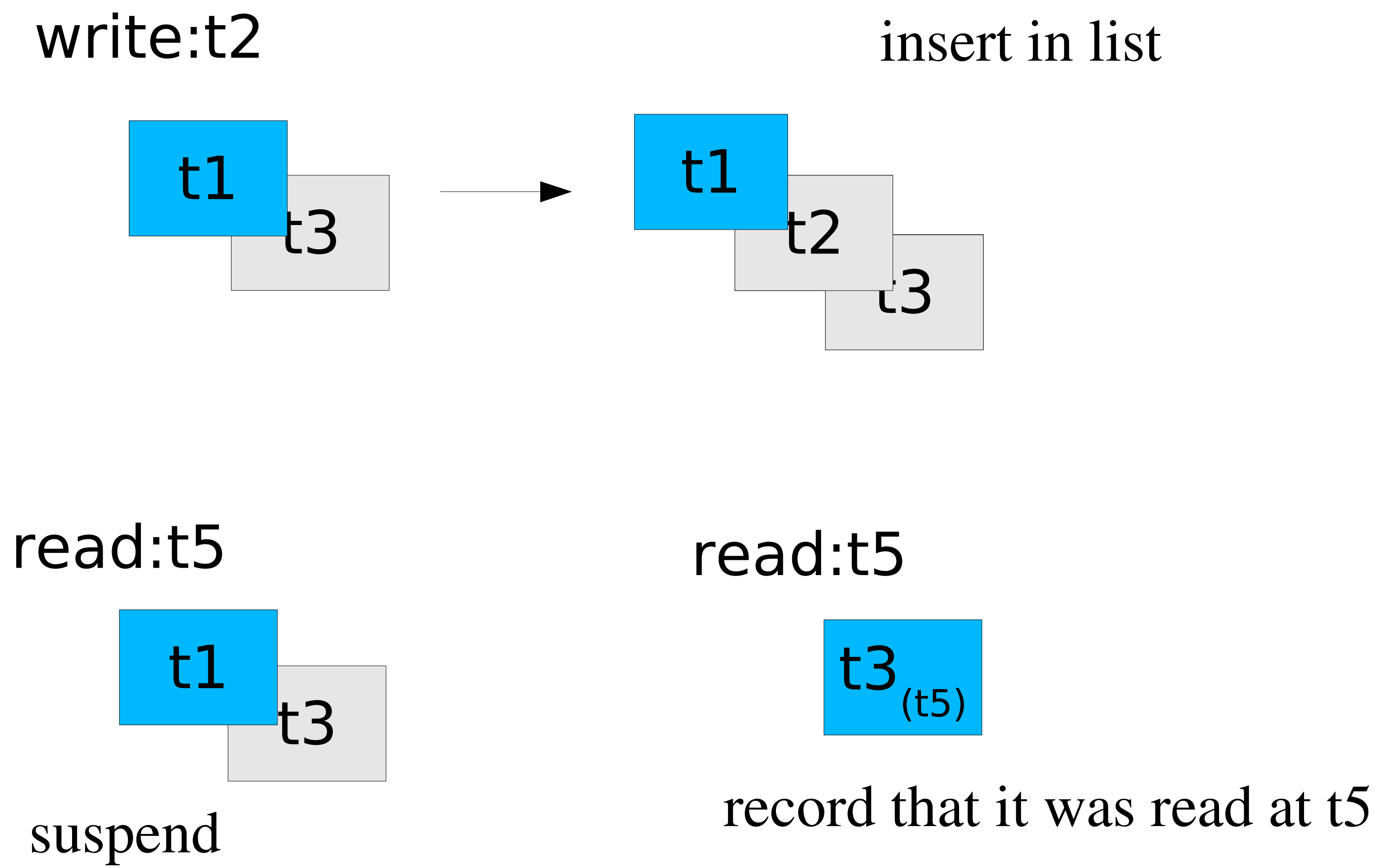
Timestamp ordering

- Each transaction is given a time stamp when started.
- There is a total order of active transactions.
- Operations are validated when performed:
  - writing only if *no later* transaction has read or written
  - reading only if *no later* transaction has written

Timestamp implementation

- 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 only waits for tentative writes
- If a operation arrives too late the transaction is aborted.

Timestamp implementation



Summary

- Transactions group sequences of operations into a ACID operation.
- Problem is how to increase concurrency.
- Need to preserve serial equivalence.
- Aborting transactions is a problem.
- Implementations:
  - locking
  - optimistic concurrency control
  - timestamps