CMPU4021
Distributed Systems

Distributed Transactions and Concurrency Control

Introduction to transactions

- A group of operations often represent a unit of “work”.
- Transaction
  - An operation composed of a number of discrete steps.
- Free from interference by operations being performed on behalf of other concurrent clients
- Either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

Basic Transaction Operations

- Begin transaction
  - mark the start of a transaction
- End transaction
  - mark the end of a transaction – no more tasks
- Commit transaction
  - make the results permanent
- Abort transaction
  - kill the transaction, restore old values
- Read/write/compute data (modify files or objects)
  - Data needs to be restored if the transaction is aborted.

The transactional model

- Applications are coded as follows:
  - *begin* transaction
  - Perform a series of *read*, *update* operations
  - Terminate by *commit* or *abort*.

ACID properties of transactions

Atomicity:

- The transaction happens as a single indivisible action. Everything succeeds or else the entire transaction is rolled back. Others do not see intermediate results.

Consistency:

- A transaction takes the system from one consistent state to another consistent state.
  - A transaction cannot leave the database in an inconsistent state.
    - E.g., total amount of money in all accounts must be the same before and after a transfer funds' transaction

Isolated (Serializable)

- Each transaction must be performed without interference from other transactions - there must be no observation by other transactions of a transaction's intermediate effects.
  - If transactions run at the same time, the final result must be the same as if they executed in some serial order.

Durability

- After a transaction has completed successfully, all its effects are saved in permanent storage.

Atomicity of transactions

Two aspects

1. All or nothing:
   - It either completes successfully, and the effects of all of its operations are recorded in the objects, or (if it fails or is aborted) it has no effect at all.
   - Two further aspects of its own:
     - failure atomicity:
       - the effects are atomic even when the server crashes;
     - durability:
       - after a transaction has completed successfully, all its effects are saved in permanent storage.

2. Isolation:
   - Each transaction must be performed without interference from other transactions
     - there must be no observation by other transactions of a transaction's intermediate effects

Transactions

- Transactions are carried out concurrently for higher performance
- Two common problems with transactions
  - Lost update
  - Inconsistent retrieval
- Solution
  - Serial equivalence

Lost Update

```
T1: A=read(x), write(x, A*10)
T2: B=read(x), write(x, B*10)
```

If not properly isolated, we could get the following interleaving:

```
A=read(x), B=read(x), write(x, A*10),
write(x, B*10)
```

Executing T1 and T2 should have increased x by ten times twice, but

- we lost one of the updates

Inconsistent retrieval

```
T1: withdraw(x, 10), deposit(y, 10)
T2: sum all accounts
```

- Improper interleaving:

```
(T1)withdraw(x, 10), (T2)sum+=read(x),
(T2)sum+=read(y), ..., (T1)deposit(y, 10)
```

- The sum would be incorrect
  - It doesn't account for the 10 that are 'in transit'
  - neither in x nor in y
    - the retrieval is inconsistent

Serial equivalence

- A *serially equivalent interleaving* is one in which the combined effect is the same as if the transactions had been done one at a time in some order
  - Does not mean to actually perform one transaction at a time, as this would lead to bad performance

- The same effect means
  - the read operations return the same values
  - the instance variables of the objects have the same values at the end

Conflicting operations

- When a pair of operations conflicts we mean that their combined effects depends on the order in which they are executed.
  - e.g. *read* and *write* (whose effects are the result returned by *read* and the value set by *write)*

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Serial equivalence and conflicting operations

- For two transactions to be *serially equivalent*, it is necessary and sufficient that
  - all pairs of conflicting operations of the two transactions be executed in the same order
    - at all of the objects they *both* access

- Consider
  - T and U access i and j

```
T: x = read(i); write(i, 10); write(j, 20);
U: y = read(j); write(j, 30); z = read (i);
```

  - serial equivalence requires that either
    - *T* accesses *i* before *U* and *T* accesses *j* before *U*. or
    - U accesses *i* before *T* and *U* accesses j before *T.*

- Serial equivalence is used as a criterion for designing concurrency control schemes

Aborted transactions

Two problems associated with aborted transactions:

- ‘Dirty reads’
  - A transaction observes a write from a transaction that has not completed yet.
  - An interaction between a *read* operation in one transaction and an earlier *write* operation on the same object
    - by a transaction that then aborts
  - A transaction that committed with a ‘dirty read’ is not recoverable
- ‘Premature writes’
  - interactions between *write* operations on the same object by different transactions, one of which aborts
- Both can occur in serially equivalent executions of transactions

Dirty reads

- T1 reads a value that T2 wrote, then commits and later, T2 aborts
  - The value is “dirty”, since the update to it should not have happened
  - T1 has committed, so it cannot be undone

- Handling dirty reads
  - Transactions are only allowed to read objects that `committed` transactions have written

Premature writes and Strict executions

- Premature writes
  - a problem related to the interaction between write operations on the same object belonging to different transactions.

- Strict executions of transactions
  - The service delays both read and write operations on an object until all transactions that previously wrote that object have either committed or aborted
    - Enforces isolation

Strict executions of transactions

- Curing premature writes:
  - if a recovery scheme uses ‘before images’
    - write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted
- Strict executions of transactions
  - to avoid both ‘dirty reads’ and ‘premature writes’.
    - delay both read and write operations
  - If both *read* and *write* operations on an object are delayed until all transactions that previously wrote that object have either committed or aborted.
  - Enforces the property of isolation
- *Tentative versions* are used during progress of a transaction
  - objects in tentative versions are stored in volatile memory

Locks

- 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.
- Example of a serializing mechanism is the use of
  - exclusive locks

Locking

- Need an object? Get a lock for it!
  - Read or write locks, or both (exclusive)
- Exclusive locks
  - Only one object can read or write at a time.
  - If you can't lock the data you have to wait

Two-phase locking

- Exclusive locks
  - Server locks object it is about to use for a client
  - If a client requests access to an object that is already locked for another clients, the operation is suspended

- Two phase locking
  - Not permitted acquire a new lock after any release
  - Transactions acquire locks in a *growing* phase and release locks in a *shrinking* phase
  - Ensures ***serial equivalence***

Strict Two Phase Locking

- Two Phase Locking
  - Transaction is not allowed any new locks after it has released a lock.

- Strict Two Phase Locking
  - Any locks acquired are not given back until the transaction completed or aborts (ensures durability).
  - Locks must be held until all the objects it updated have been written to permanent storage.

Strict two phase locking

- Locks are only released upon commit / abort
- Extension of two-phase locking that prevents ***dirty reads*** and ***premature writes***

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.

Deadlock

- A state in which each member of a group of transactions is waiting for some other member to release a lock

Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on *A*
•••		•••	
•••		•••	
		•••	

Flat and Nested Transactions

Flat transaction

- Performed atomically on a unit of work

Nested

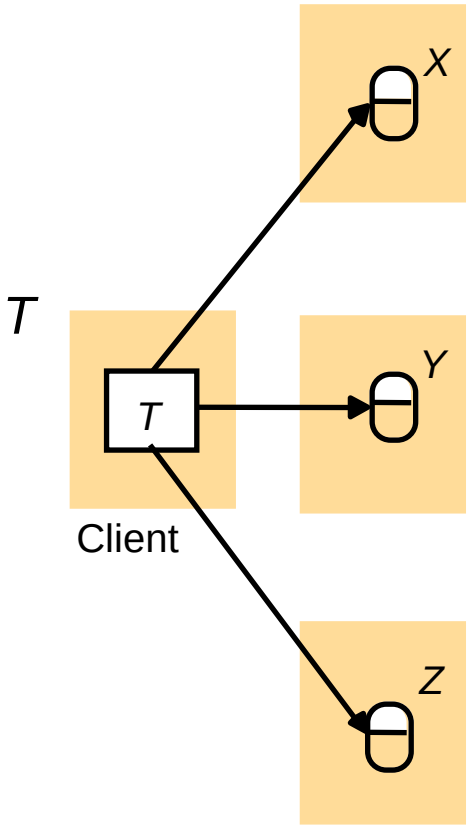
- Hierarchical

- Transactions may be composed of other transactions.
- Several transactions may be started from within a transaction
  - we have a top-level transaction and subtransactions which may have their own subtransactions.

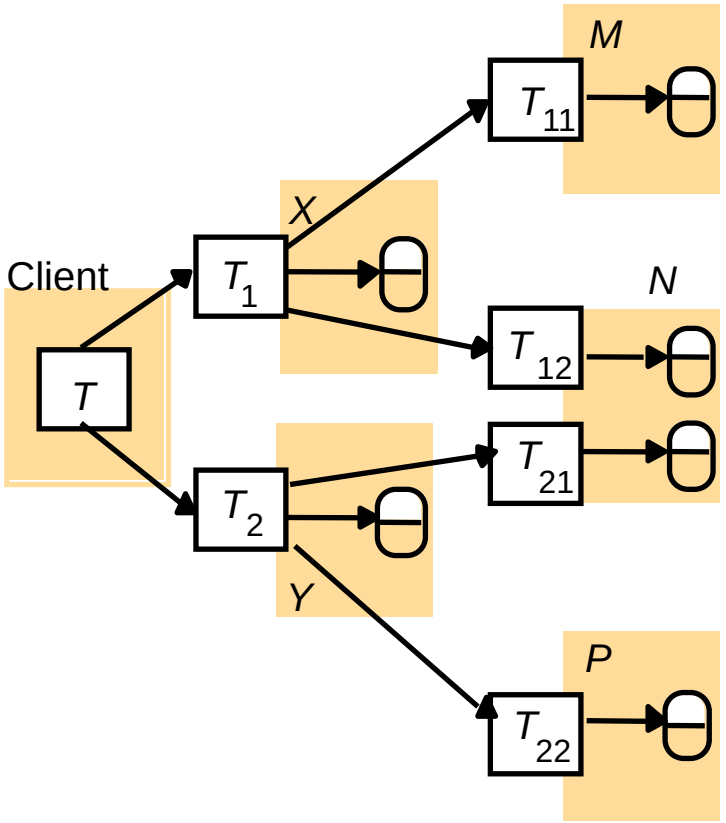
- To a parent, a subtransaction is atomic with respect to failures and concurrent access. Transactions at the same level can run concurrently but access to common objects is serialised - a subtransaction can fail independently of its parent and other subtransactions; when it aborts, its parent decides what to do, e.g. start another subtransaction or give up.

Flat and Nested Transactions

(a) Flat transaction



(b) Nested transactions



Advantages of nested transactions (over *flat* ones)

- Subtransactions may run concurrently with other subtransactions at the same level.
  - this allows additional concurrency in a transaction.
  - when subtransactions run in different servers, they can work in parallel.

- Subtransactions can commit or abort independently.
  - This is potentially more robust
  - A parent can decide on different actions according to whether a subtransaction has aborted or not
  - This is potentially more robust and a parent can decide on different actions according to whether a subtransaction has aborted or not.

Distributed Transaction

- Transaction that updates data on two or more systems
- Implemented as a set of sub-transactions
- Challenge
  - Handle machine, software, & network failures while preserving transaction integrity

Distributed transactions

- A *distributed transaction* refers to a flat or nested transaction that accesses objects managed by
  - *Multiple* servers (processes)
  - All servers need to commit or abort a transaction
- Allows for even better performance
  - At the price of increased complexity

Distributed Transactions

- Each computer runs a transaction manager
  - Responsible for sub-transactions on that system
  - Performs prepare, commit, and abort calls for sub-transactions
- Every sub-transaction must agree to commit changes before the overall transaction can complete

Committing Distributed Transactions

- Transactions may process data at more than one server.
  - Problem: any server may fail or disconnect while a commit for transaction T is in progress.
  - They must agree to commit or abort
    - “Log locally, commit globally.”
- The atomicity property of transactions
  - when a distributed transaction comes to an end, either all of its operations are carried out or none of them.

Summary

- Transaction
  - An operation composed of a number of discrete steps.
- A distributed transaction involves several different servers.
- Atomicity requires that the servers participating in a distributed transaction either all commit it or all abort it
  - Atomic commit protocols are designed to achieve this effect, even if servers crash during their execution

References

- Chapter 16: Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 5thEd.