CMPU4021
Distributed Systems

Transactions and Concurrency Control

Introduction to transactions

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.

ACID properties of transactions

- **A**tomicity:
  - *All or nothing*
  - The transaction happens as a single indivisible action. Everything succeeds or else the entire transaction is rolled back. Others do not see intermediate results.

- **C**onsistency:
  - 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

- **I**solated (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.

- **D**urability
  - 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’
  - 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

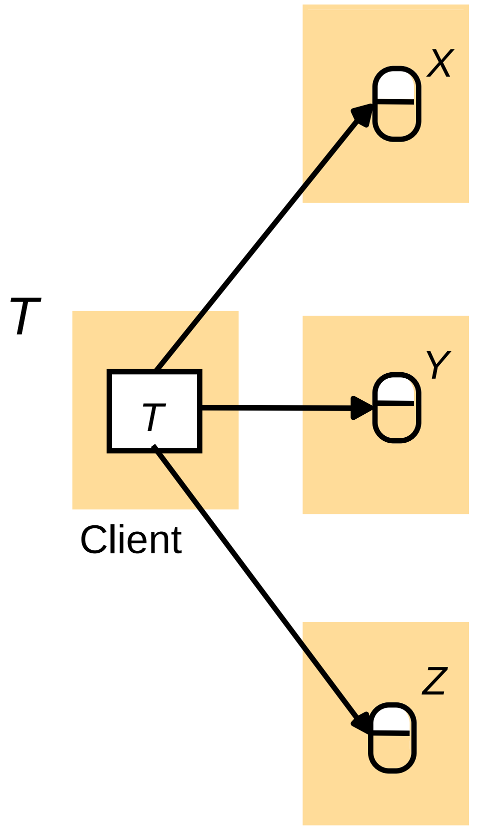
- Hierarchical

- 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.

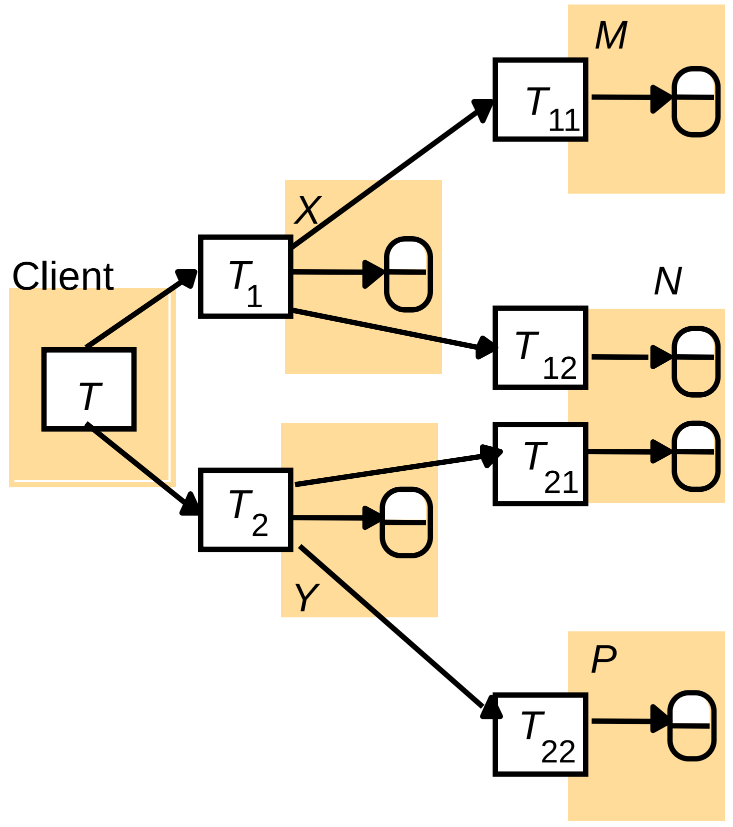
- To a parent, a subtransaction is atomic with respect to failures and concurrent access. Transactions at the same level can run concurrently but access to common objects is serialised - a subtransaction can fail independently of its parent and other subtransactions; when it aborts, its parent decides 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 TRANSACTIONS

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

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.

Failure model for the commit protocols

- Commit protocols are designed to work in an asynchronous system in which
  - servers may crash
  - messages may be lost

- It is assumed that an underlying request-reply protocol removes corrupt and duplicated messages.

- There are no Byzantine faults
  - servers either crash or obey the messages they are sent

Atomic commit protocols

- One coordinator and multiple participants
- Protocols for atomic distributed commit
  - One-phase
    - the coordinator to communicate the commit or abort request to all of the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they have carried it out.
  - Two-phase
    - designed to allow any participant to abort its part of a transaction
    - can result in extensive delays for participants in the uncertain state.
  - Three-phase
    - designed to alleviate delays due to participants in the uncertain state.
    - more expensive in terms of the number of messages and the number of rounds
    - required for the normal (failure-free) case.

TWO-PHASE COMMIT PROTOCOL

The two-phase commit protocol

- During the progress of a transaction, the only communication between coordinator and participant is the *join* request
  - The client request to commit or abort goes to the coordinator
    - if client or participant request abort, the coordinator informs the participants immediately
    - if the client asks to commit, the 2PC comes into use

- 2PC
  - *voting phase*: coordinator asks all participants if they can commit
    - if yes, participant records updates in permanent storage and then votes
  - *completion phase*: coordinator tells all participants to commit or abort
  - the next slide shows the operations used in carrying out the protocol

Operations for two-phase commit protocol

canCommit?(trans)-> Yes / No

This is a request with a reply

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

doCommit(trans)

Call from coordinator to participant to tell participant to commit its part of a transaction.

These are asynchronous requests to avoid delays

doAbort(trans)

Call from coordinator to participant to tell participant to abort its part of a transaction.

Asynchronous request

haveCommitted(trans, participant)

Call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) -> Yes / No

Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

- participant interface- *canCommit?, doCommit, doAbort*
- coordinator interface- *haveCommitted, getDecision*

The two-phase commit protocol

Phase 1 (voting phase):

1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

The Voting Rules

1. Each participant has one vote which can be either 'commit‘ or 'abort’;

2. Having voted, a participant cannot change its vote;

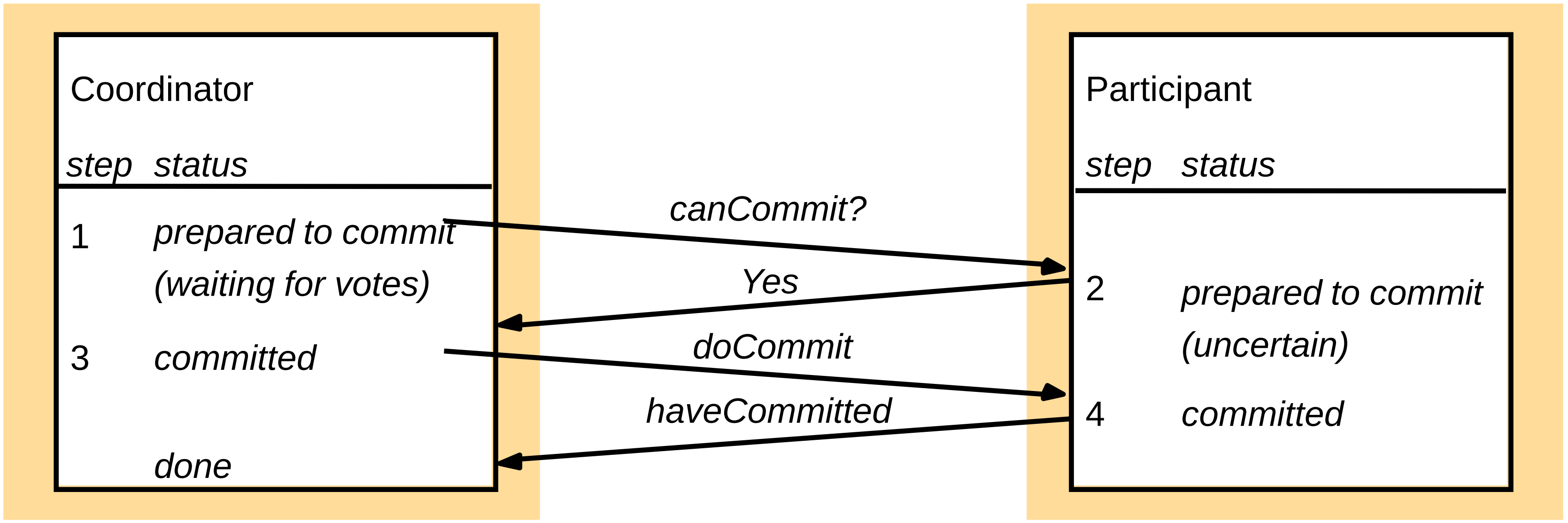
3. If a participant votes 'abort' then it is free to abort the transaction immediately; any site is in fact free to abort a transaction at any time up until it records a 'commit' vote. Such a transaction abort is known as a unilateral abort.

4. If a participant votes ‘commit’, then it must wait for the co-ordinator to broadcast either the 'global-commit' or 'global-abort' message;

5. If all participants vote 'commit' then the global decision by the co-ordinator must be 'commit';

6. The global decision must be adopted by all participants.

Communication in two-phase commit protocol



- Time-out actions in the 2PC
    - to avoid blocking forever when a process crashes or a message is lost
  - *uncertain* participant (step 2) has voted yes. it can't decide on its own
    - it uses `getDecision` method to ask coordinator about outcome
  - participant has carried out client requests, but has not had a `Commit?` from the coordinator. It can abort unilaterally
  - coordinator delayed in waiting for votes (step 1). It can abort and send `doAbort` to participants.

Communication in two-phase commit protocol

- A participant may be delayed
  - Carried out all its client requests, but has not yet received a *`canCommit`*? Call from the coordinator.
  - As the client sends the *`closeTransaction`* to the `Coordinator`, a participant can only detect such a situation if it notices that it has not had a request in a particular transaction for a long time, e.g.
    - By a timeout period on a lock.
  - As no decision has been made at this stage, the participant can decide to *abort* unilaterally after some period of time.

Communication in two-phase commit protocol

- The coordinator may be delayed:
  - When it is waiting for votes from the participants.
  - As it has not yet decided the fate of the transaction, it may decide to abort the transaction after some period of time.
  - It must then announce *doAbort* to the participants who have already sent their votes.
  - Some participant may try to vote *Yes* after this, but their votes will be ignored and they will enter the *uncertain* state.

Performance of the two-phase commit protocol

- If there are no failures, the 2PC involving *N* participants requires
  - *N* *canCommit?* messages and replies, followed by *N* *doCommit* messages.
    - the cost in messages is proportional to 3*N*, and the cost in time is three rounds of messages.
    - The *haveCommitted* messages are not counted

- There may be arbitrarily many server and communication failures

- 2PC is guaranteed to complete eventually, but it is not possible to specify a time limit within which it will be complete
  - delays to participants in uncertain state
  - some 3PCs designed to alleviate such delays
    - they require more messages and more rounds for the normal case

THREE-PHASE COMMIT PROTOCOL

Three-phase commit (3PC) protocol

Phase 1 (the same as for two-phase commit):

- The coordinator sends a `canCommit?` request to each of the participants in the transaction.
- When a participant receives a `canCommit?` request it replies with its vote (`Yes` or `No`) to the coordinator. Before voting Yes, it prepares to commit by saving objects in permanent storage. If the vote is No the participant aborts immediately.

Phase 2:

- The coordinator collects the votes and makes a decision.
  - If it is `No`, it `aborts` and informs participants that voted `Yes`
  - if the decision is `Yes`, it sends a `preCommit` request to all the participants.
  - Participants that voted `Yes` wait for a `preCommit` or `doAbort` request.
  - They acknowledge `preCommit` requests and carry out `doAbort` requests.

Phase 3:

- The coordinator collects the acknowledgements.
- When all are received, it commits and sends `doCommit` requests to the participants.
- Participants wait for a `doCommit` request.
- When it arrives, they `commit`.

Commit protocols: delays handling

Assumptions: communication does not fail:

- Two-phase commit protocol
  - the ‘uncertain/delay’ period occurs because a participant has voted yes but has not yet been told the outcome.
    - It can no longer abort unilaterally
- Three-phase commit protocol
  - The participants ‘uncertain’ period lasts from when the participant votes yes until it receives the *preCommit* request.
  - At this stage, no other participant can have committed. Therefore if a group of participants discover that they are all ‘uncertain’ and the coordinator cannot be contacted, they can decide unilaterally to abort.

GENERAL ASYNCHRONOUS CONSENSUS: PAXOS

Paxos: High Overview

- Paxos is a family of protocols providing distributed consensus
- Goal: agree on a single value even if multiple systems propose different values concurrently
- Common use: provide a consistent ordering of events from multiple clients
  - All machines running the algorithm agree on a proposed value from a client
  - The value will be associated with an event or action
  - Paxos ensures that no other machine associates the value with another event

Paxos: High Overview

- Three classes of “agents“
  - Proposers: proposes a value `v`
  - Acceptors: accepts a value `v` that is proposed
  - Learners: learns that a value `v` was accepted

- These agents may be mapped to the same process, so a process could play all three roles

- Three phases
  - Prepare
    - “What’s the last proposed value?”
  - Accept
    - “Accept my proposal.”
  - Learn
    - “Let’s tell others about the consensus.”

Paxos

- Used in Chubby
  - The central component of the Google infrastructure offering storage and coordination services for other infrastructure services, including GFS and Bigtable.
  - A lock service
  - Enables multiple clients to share a lock and coordinate
  - Locks are supposed to be held for hours and days, not seconds.
  - In addition, it can store small files.

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 and 17: Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 5thEd.