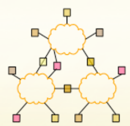
15-446 Distributed Systems Spring 2009



L-16 Transactions

1

Today's Lecture

- Transaction basics
- Locking and deadlock
- Distributed transactions

2

Transactions

- A ***transaction*** is a sequence of server operations that is guaranteed by the server to be atomic in the presence of multiple clients and server crashes.
  - 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

3

Transactions – The ACID Properties

- Are the four desirable properties for reliable handling of concurrent transactions.
- Atomicity
  - The “All or Nothing” behavior.
- C: stands for either
  - Concurrency: Transactions can be executed concurrently
  - … or Consistency: Each transaction, if executed by itself, maintains the correctness of the database.
- Isolation (Serializability)
  - Concurrent transaction execution should be equivalent (in effect) to a *serialized* execution.
- Durability
  - Once a transaction is *done*, it stays done.

4

Bank Operations

Operations of the Account interface

deposit(amount)
deposit amount in the account
withdraw(amount)
withdraw amount from the account
getBalance() -> amount
return the balance of the account
setBalance(amount)
set the balance of the account to amount

Operations of the Branch interface

create(name) → account
create a new account with a given name
lookUp(name) → account
return a reference to the account with the given name
branchTotal() → amount
return the total of all the balances at the branch

A client's banking transaction

Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

5

The transactional model

- Applications are coded in a stylized way:
  - *begin* transaction
  - Perform a series of *read*, *update* operations
  - Terminate by *commit* or *abort*.
- Terminology
  - The application is the transaction manager
  - The data manager is presented with operations from concurrently active transactions
  - It schedules them in an interleaved but serializable order

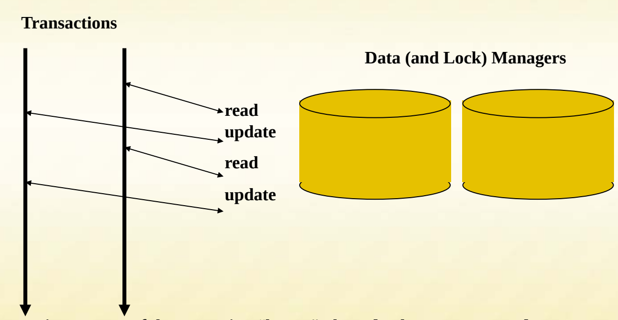
6

A side remark

- Each transaction is built up incrementally
  - Application runs
  - And as it runs, it issues operations
  - The data manager sees them one by one
- But often we talk as if we knew the whole thing at one time
  - We're careful to do this in ways that make sense
  - In any case, we usually don't need to say anything until a "commit" is issued

7

Transaction and Data Managers



transactions are stateful: transaction "knows" about database contents and updates

8

Typical transactional program

```
begin transaction;
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
commit transaction;
```


Transaction life histories

Successful	Aborted by client		Aborted by server
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
:	:	server aborts transaction →	:
operation	*operation*		*operation ERROR reported to client*
closeTransaction	*abortTransaction*		

- *openTransaction() → trans;*
  - starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.
- *closeTransaction(trans) → (commit, abort);*
  - ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.
- *abortTransaction(trans);*
  - aborts the transaction.

Transactional Execution Log

- As the transaction runs, it creates a history of its actions. Suppose we were to write down the sequence of operations it performs.
- Data manager does this, one by one
- This yields a "schedule"
  - Operations and order they executed
  - Can infer order in which transactions ran
- Scheduling is called "concurrency control"

Concurrency control

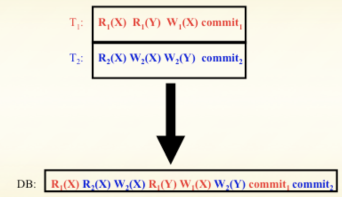
- Motivation: without concurrency control, we have lost updates, inconsistent retrievals, dirty reads, etc. (see following slides)
- Concurrency control schemes are designed to allow two or more transactions to be executed correctly while maintaining serial equivalence
  - Serial Equivalence is correctness criterion
    - Schedule produced by concurrency control scheme should be equivalent to a serial schedule in which transactions are executed one after the other
- Schemes:
  - locking,
  - optimistic concurrency control,
  - time-stamp based concurrency control

Serializability

- Means that effect of the interleaved execution is indistinguishable from some possible serial execution of the committed transactions
- For example: *T1 and T2 are interleaved but it "looks like" T2 ran before T1*
- Idea is that transactions can be coded to be correct if run in isolation, and yet will run correctly when executed concurrently (and hence gain a speedup)

13

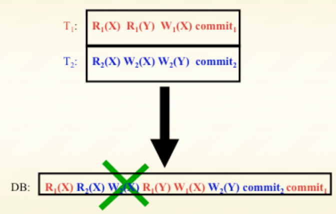
Need for serializable execution



Data manager interleaves operations to improve concurrency

14

Non serializable execution

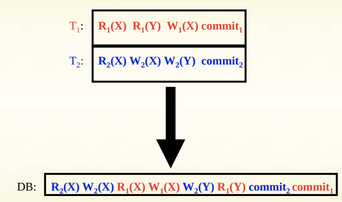


Unsafe! Not serializable

Problem: transactions may "interfere". Here, T_2 changes x, hence T_1 should have either run first (read and write) or after (reading the changed value).

15

Serializable execution



Data manager interleaves operations to improve concurrency but schedules them so that it looks as if one transaction ran at a time. This schedule "looks" like T_2 ran first.

16

Read and *write* operation conflict rules

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

17

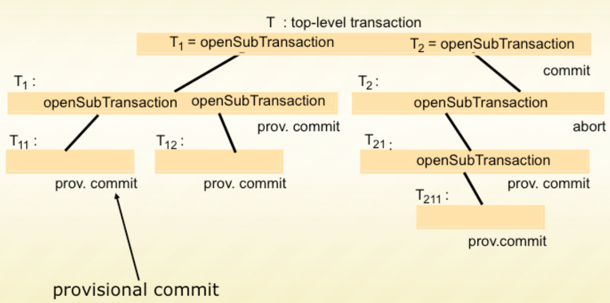
A dirty read when transaction *T* aborts

Transaction *T*: *a.getBalance()* *a.setBalance(balance + 10)*		**Transaction *U*:** *a.getBalance()* *a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

uses result of uncommitted transaction!

18

Nested transactions



19

Committing Nested Transactions

- A transaction may commit or abort only after its child transactions have completed
- When a sub-transaction completes, it makes an independent decision either to commit provisionally or to abort. Its decision to abort is final.
- When a parent aborts, all of its sub-transactions are aborted
- When a sub-transaction aborts, the parent can decide whether to abort or not
- If a top-level transaction commits, then all of the sub-transactions that have provisionally committed can commit too, provided that non of their ancestors has aborted.

20

Today's Lecture

- Transaction basics
- Locking and deadlock
- Distributed transactions

Schemes for Concurrency control

- Locking
  - Server attempts to gain an exclusive 'lock' that is about to be used by one of its operations in a transaction.
  - Can use different lock types (read/write for example)
  - Two-phase locking
- Optimistic concurrency control
- Time-stamp based concurrency control

What about the locks?

- Unlike other kinds of distributed systems, transactional systems typically *lock* the data they access
- They obtain these locks as they run:
  - Before accessing "x" get a lock on "x"
  - Usually we assume that the application knows enough to get the right kind of lock. It is not good to get a read lock if you'll later need to update the object
- In clever applications, one lock will often cover many objects

Locking rule

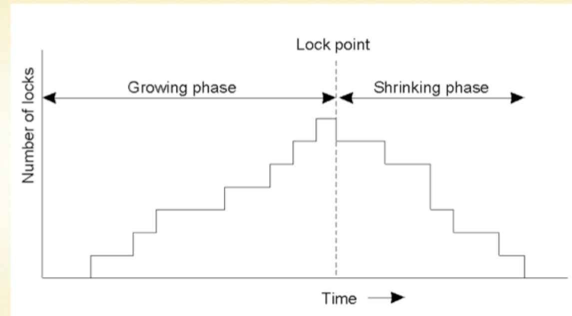
- Suppose that transaction *T* will access object *x*.
  - We need to know that first, *T* gets a lock that "covers" *x*
- What does coverage entail?
  - We need to know that if any other transaction *T'* tries to access *x* it will attempt to get the *same lock*

Examples of lock coverage

- We could have one lock per object
- … or one lock for the whole database
- … or one lock for a category of objects
  - In a tree, we could have one lock for the whole tree associated with the root
  - In a table we could have one lock for row, or one for each column, or one for the whole table
- All transactions must use the same rules!
- And if you will update the object, the lock must be a "write" lock, not a "read" lock

25

Two-Phase Locking (1)

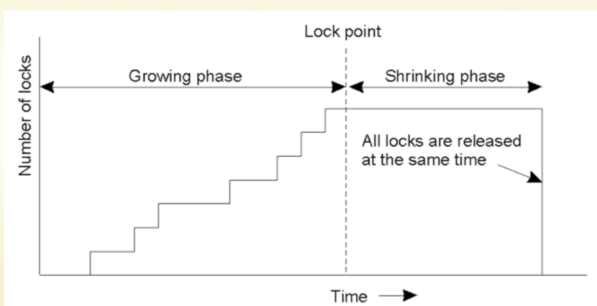


In two-phase locking, a transaction is not allowed to acquire any new locks after it has released a lock

26

Strict Two-Phase Locking (2)

- Strict two-phase locking.



27

Use of locks in 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.

28

Lock compatibility

For one object		*Lock requested* *read*	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

Operation Conflict rules:

1. If a transaction T has already performed a read operation on a particular object, then a concurrent transaction U must not write that object until T commits or aborts
2. If a transaction T has already performed a read operation on a particular object, then a concurrent transaction U must not read or write that object until T commits or aborts

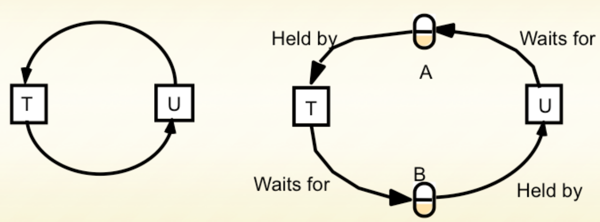
29

Deadlock with write locks

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*
•••		•••	
•••		•••	

30

The wait-for graph



31

Dealing with Deadlock in two-phase locking

- Deadlock prevention
  - Acquire all needed locks in a single atomic operation
  - Acquire locks in a particular order
- Deadlock detection
  - Keep graph of locks held. Check for cycles periodically or each time an edge is added
  - Cycles can be eliminated by aborting transactions
- Timeouts
  - Aborting transactions when time expires

32

Resolution of deadlock

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*
		•••	
(timeout elapses) *T*'s lock on *A* becomes vulnerable, unlock *A*, abort T		•••	
		a.withdraw(200);	write locks *A* unlock *A*, *B*

33

Contrast: Timestamped approach

- Using a fine-grained clock, assign a "time" to each transaction, uniquely. E.g. T1 is at time 1, T2 is at time 2
- Now data manager tracks temporal history of each data item, responds to requests as if they had occured at time given by timestamp
- At commit stage, make sure that commit is consistent with serializability and, if not, abort

34

Example of when we abort

- T1 runs, updates x, setting to 3
- T2 runs concurrently but has a larger timestamp. It reads x=3
- T1 eventually aborts
- ... T2 must abort too, since it read a value of x that is no longer a committed value
  - Called a cascaded abort since abort of T_1 triggers abort of T_2

35

Pros and cons of approaches

- Locking scheme works best when conflicts between transactions are common and transactions are short-running
- Timestamped scheme works best when conflicts are rare and transactions are relatively long-running

36

Today's Lecture

- Transaction basics
- Locking and deadlock
- Distributed transactions

37

Distributed Transactions

- Motivation
  - Provide distributed atomic operations at multiple servers that maintain shared data for clients
  - Provide recoverability from server crashes
- Properties
  - Atomicity, Consistency, Isolation, Durability (ACID)
- Concepts: commit, abort, distributed commit

38

Concurrency Control for Distributed Transactions

- Locking
  - Distributed deadlocks possible
- Timestamp ordering
  - Lamport time stamps
    - for efficiency it is required that timestamps issued by coordinators be roughly synchronized

39

Transactions in distributed systems

- Notice that client and data manager might not run on same computer
  - Both may not fail at same time
  - Also, either could timeout waiting for the other in normal situations
- When this happens, we normally abort the transaction
  - Exception is a timeout that occurs while commit is being processed
  - If server fails, one effect of crash is to break locks even for read-only access

40

Transactions in distributed systems

- Main issue that arises is that now we can have multiple database servers that are touched by one transaction
- Reasons?
  - Data spread around: each owns subset
  - Could have replicated some data object on multiple servers, e.g. to load-balance read access for large client set
  - Might do this for high availability

41

Atomic Commit Protocols

- The atomicity of a transaction requires that when a distributed transaction comes to an end, either all of its operations are carried out or none of them
- One phase commit
  - Coordinator tells all participants to commit
    - If a participant cannot commit (say because of concurrency control), no way to inform coordinator
- Two phase commit (2PC)

42

The two-phase commit protocol - 1

Phase 1 (voting phase):

1. The coordinator sends a *canCommit? (VOTE_REQUEST)* request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote *Yes (VOTE_COMMIT)* or *No* (*VOTE_ABORT*) 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.

43

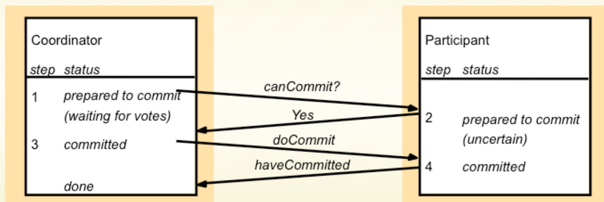
The two-phase commit protocol - 2

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 (GLOBAL_COMMIT)* request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort (GLOBAL_ABORT)* 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.

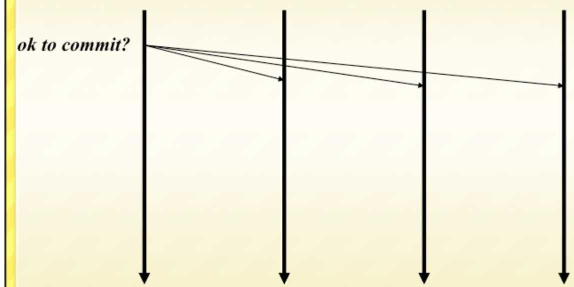
44

Communication in two-phase commit protocol



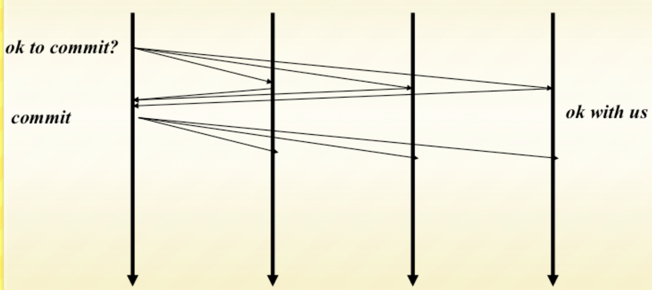
45

Commit protocol illustrated



46

Commit protocol illustrated



Note: garbage collection protocol not shown here

47

Operations for two-phase commit protocol

canCommit?(trans)-> Yes / No
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.

doAbort(trans)
Call from coordinator to participant to tell participant to abort its part of a transaction.

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.

48

Two-Phase Commit protocol - 3

- **actions by coordinator:**

```
while START _2PC to local log;
  multicast VOTE_REQUEST to all participants;
  while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
      write GLOBAL_ABORT to local log;
      multicast  GLOBAL_ABORT to all participants;
      exit;
    }
    record vote;
  }
  if all participants sent VOTE_COMMIT and coordinator votes COMMIT{
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;
  } else {
    write GLOBAL_ABORT  to local log;
    multicast GLOBAL_ABORT to all participants;
  }
```

49

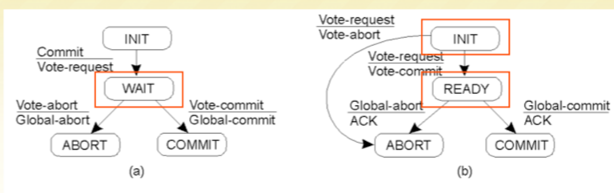
Two-Phase Commit protocol - 4

- **actions by participant:**

```
write INIT to local log;
  wait for VOTE_REQUEST from coordinator;
  if timeout {
    write VOTE_ABORT to local log;
    exit;
  }
  if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
      multicast DECISION_REQUEST to other participants;
      wait until DECISION is received; /* remain blocked */
      write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
      write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
      write GLOBAL_ABORT to local log;
  } else {
    write VOTE_ABORT to local log;
    send  VOTE ABORT to coordinator;
  }
```

50

Two-Phase Commit protocol - 5



a) The finite state machine for the coordinator in 2PC.
b) The finite state machine for a participant.

- If a failure occurs during a 'blocking' state (red boxes), there needs to be a recovery mechanism.

51

Two Phase Commit Protocol - 6

Recovery

- 'Wait' in Coordinator – use a time-out mechanism to detect participant crashes. Send GLOBAL_ABORT
- 'Init' in Participant – Can also use a time-out and send VOTE_ABORT
- 'Ready' in Participant P – abort is not an option (since already voted to COMMIT and so coordinator might eventually send GLOBAL_COMMIT). Can contact another participant Q and choose an action based on its state.

State of Q	Action by P
COMMIT	Transition to COMMIT
ABORT	Transition to ABORT
INIT	Both P and Q transition to ABORT (Q sends VOTE_ABORT)
READY	Contact more participants. If all participants are 'READY', must wait for coordinator to recover

52

Two-Phase Commit protocol - 7

- **actions for handling decision requests:** /* executed by separate thread */

```
while true {
      wait until any incoming DECISION_REQUEST is
   received; /* remain blocked */
      read most recently recorded STATE from the local log;
      if STATE == GLOBAL_COMMIT
         send GLOBAL_COMMIT to requesting participant;
      else if STATE == INIT or STATE == GLOBAL_ABORT
         send GLOBAL_ABORT to requesting participant;
      else
         skip;  /* participant remains blocked */
```

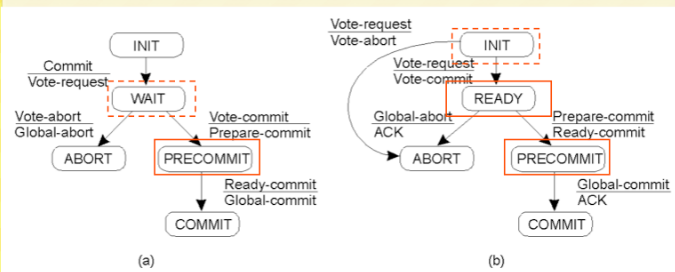
53

Three Phase Commit protocol - 1

- Problem with 2PC
  - If coordinator crashes, participants cannot reach a decision, stay blocked until coordinator recovers
- Three Phase Commit3PC
  - There is no single state from which it is possible to make a transition directly to either COMMIT or ABORT states
  - There is no state in which it is not possible to make a final decision, and from which a transition to COMMIT can be made

54

Three-Phase Commit protocol - 2



a) Finite state machine for the coordinator in 3PC
b) Finite state machine for a participant

55

Three Phase Commit Protocol - 3

Recovery

- 'Wait' in Coordinator – same
- 'Init' in Participant – same
- 'PreCommit' in Coordinator – Some participant has crashed but we know it wanted to commit. GLOBAL_COMMIT the application knowing that once the participant recovers, it will commit.
- 'Ready' or 'PreCommit' in Participant P – (i.e. P has voted to COMMIT)

State of Q	Action by P
PRECOMMIT	Transition to PRECOMMIT. If all participants in PRECOMMIT, can COMMIT the transaction
ABORT	Transition to ABORT
INIT	Both P (in READY) and Q transition to ABORT (Q sends VOTE_ABORT)
READY	Contact more participants. If can contact a majority and they are in 'Ready', then ABORT the transaction. If the participants contacted in 'PreCommit' it is safe to COMMIT the transaction

Note: if any participant is in state PRECOMMIT, it is impossible for any other participant to be in any state other than READY or PRECOMMIT.

56