Transactions

Distributed Software Systems

Transactions

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

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);

Operations in *Coordinator* interface

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.

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*		

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

The lost update problem

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

The inconsistent retrievals problem

Transaction ***V*** *a.withdraw(100)* *b.deposit(100)*		**Transaction** ***W*:** *aBranch.branchTotal()*	
a.withdraw(100);	\$100		
		total = a.getBalance()	\$100
		total = total+b.getBalance()	\$300
		total = total+c.getBalance()	
b.deposit(100)	\$300	:	

A serially equivalent interleaving of *T* and *U*

Transaction ***T*** *balance = b.getBalance()* *b.setBalance(balance*1.1)* *a.withdraw(balance/10)*		**Transaction** ***U*** *balance = b.getBalance()* *b.setBalance(balance*1.1)* *c.withdraw(balance/10)*	
balance = b.getBalance()	\$200		
*b.setBalance(balance*1.1)*	\$220		
		balance = b.getBalance()	\$220
		*b.setBalance(balance*1.1)*	\$242
a.withdraw(balance/10)	\$80		
		c.withdraw(balance/10)	\$278

A serially equivalent interleaving of *V* and *W*

Transaction *V*: *a.withdraw(100); b.deposit(100)*		**Transaction *W*:** *aBranch.branchTotal()*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$400
		total = total+c.getBalance()	
		...	

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			

Serializability

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```

(b)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```

(c)

Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	Illegal

(d)

a) – c) Three transactions T_1, T_2, and T_3
d) Possible schedules

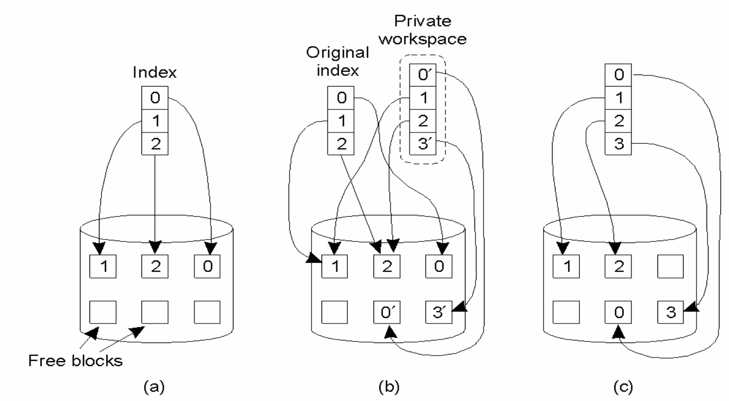
Read and *write* operation conflict rules

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

A non-serially equivalent interleaving of operations of transactions *T* and *U*

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

Implementing Transactions: Private Workspace



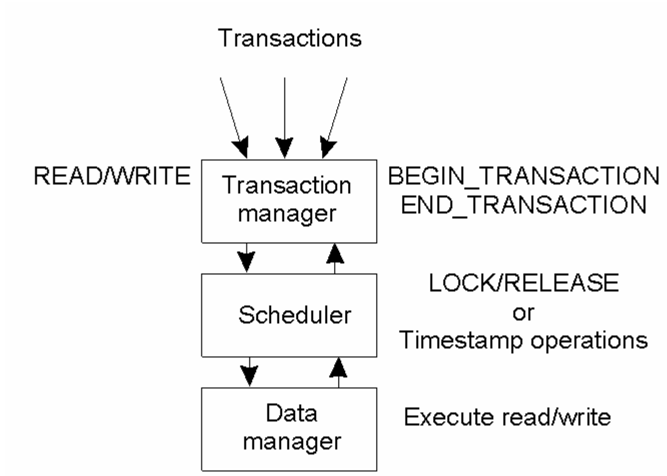
a) The file index and disk blocks for a three-block file
b) The situation after a transaction has modified block 0 and appended block 3
c) After committing

Implementing Transactions: Writeahead Log

x = 0;	Log	Log	Log
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

a) A transaction
b) – d) The log before each statement is executed

Concurrency Control



General organization of managers for handling transactions.

Transactions *T* and *U* with exclusive locks

Transaction *T*:		Transaction *U*:	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

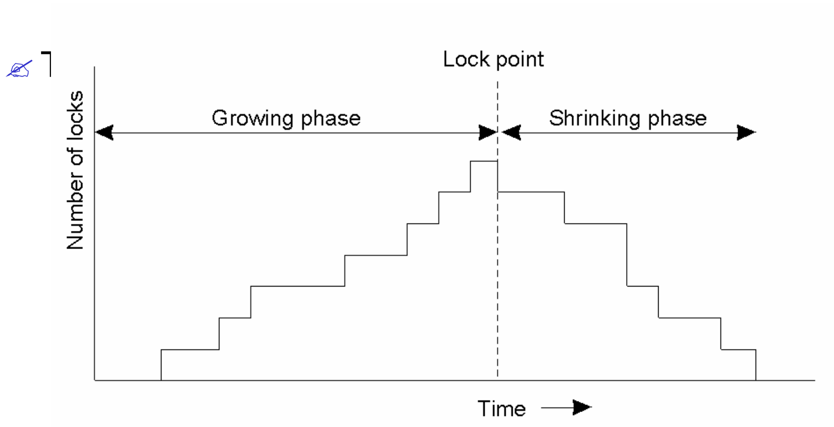
Lock compatibility

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

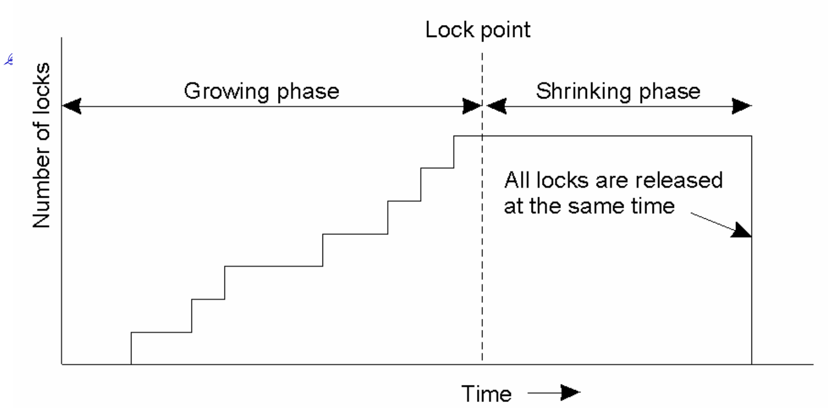
Use of locks in strict two-phase locking

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.

Two-Phase Locking (1)



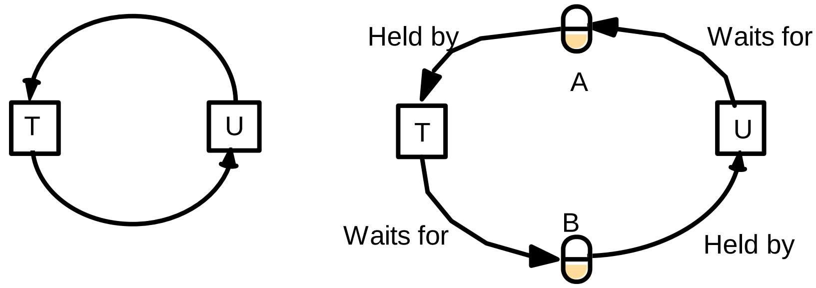
Strict Two-Phase Locking (2)



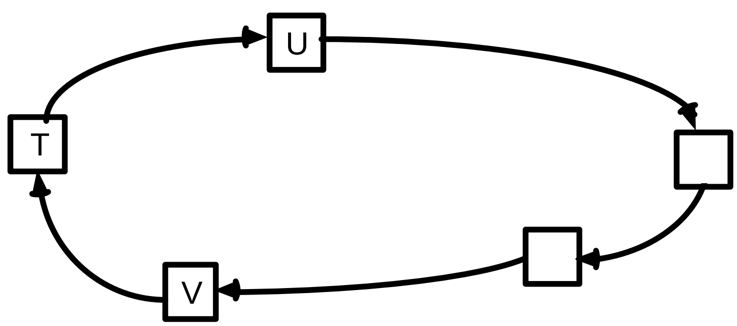
Deadlock with write locks

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*
•••		•••	
•••		•••	
		•••	

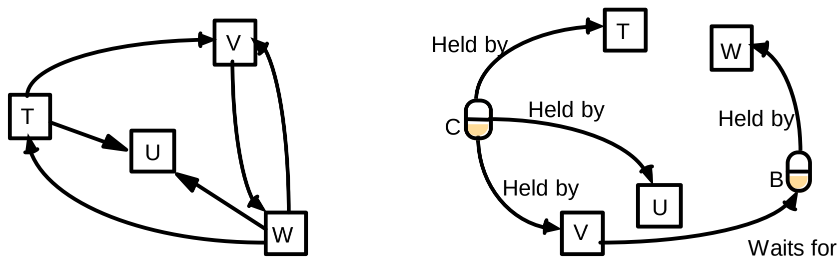
The wait-for graph



A cycle in a wait-for graph



Another wait-for graph



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*

Optimistic Concurrency Control

- Drawbacks of locking
  - Overhead of lock maintenance
  - Deadlocks
  - Reduced concurrency
- Optimistic Concurrency Control
  - In most applications, likelihood of conflicting accesses by concurrent transactions is low
  - Transactions proceed as though there are no conflicts
  - Three phases
    - Working Phase – transactions read and write private copies of objects
    - Validation Phase – each transaction is assigned a transaction number when it enters this phase
    - Update Phase

Optimistic Concurrency Control: Serializability of transaction T_v with respect to transaction T_i

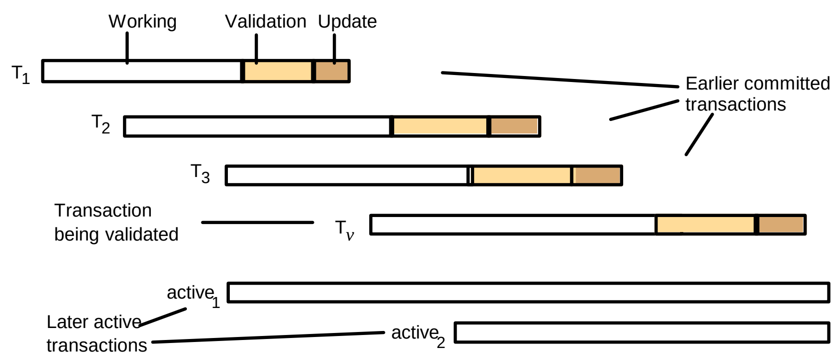
T_v and T_i are overlapping transactions

For T_v to be serializable wrt T_i the following rules must hold

T_v	T_i	Rule
write	*read*	1. T_i must not read objects written by T_v
read	*write*	2. T_v must not read objects written by T_i
write	*write*	3. T_i must not write objects written by T_v and T_v must not write objects written by T_i

If simplification is made that only one transaction may be in its validation or write phases at one time, then third rule is always satisfied

Validation of transactions



Validation of Transactions

Backward validation of transaction T_v

boolean valid = true;
for (int T_i = *startTn*+1; T_i <= *finishTn*; T_i++){
if (read set of T_v intersects write set of T_i) valid = false;
}

Forward validation of transaction T_v

boolean valid = true;
for (int T_{id} = *active1*; T_{id} <= *activeN*; T_{id}++){
if (write set of T_v intersects read set of T_{id}) valid = false;
}

Timestamp based concurrency control

- Each timestamp is assigned a unique timestamp at the moment it starts
  - In distributed transactions, Lamport's timestamps can be used
- Every data item has a timestamp
  - Read timestamp = timestamp of transaction that last read the item
  - Write timestamp = timestamp of transaction that most recently changed an item

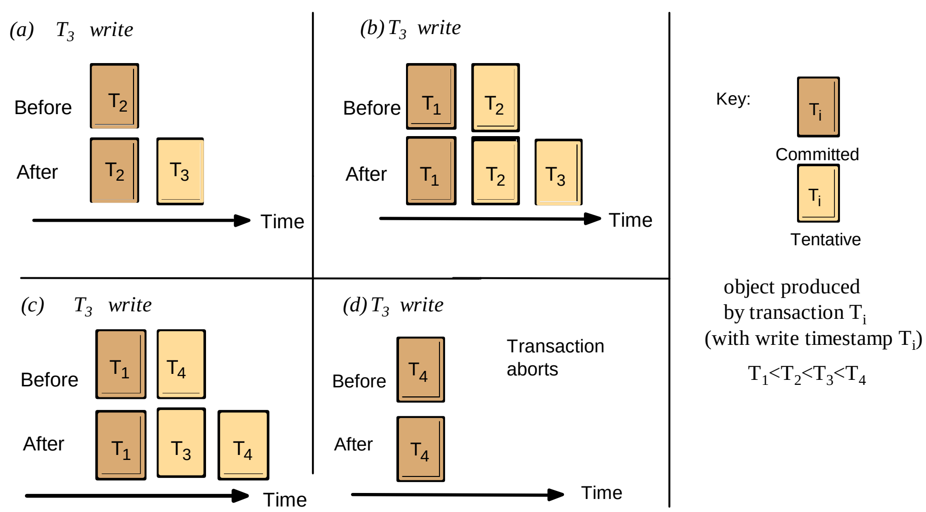
Operation conflicts for timestamp ordering

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.
2.	*write*	*write*	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$ this requires that T_c > write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$ this requires that T_c > write timestamp of the committed object.

Timestamp ordering write rule

if (T_c = maximum read timestamp on D &&
    T_c > write timestamp on committed version of D)
        perform write operation on tentative version of D with write timestamp T_c
else /* write is too late */
    Abort transaction T_c

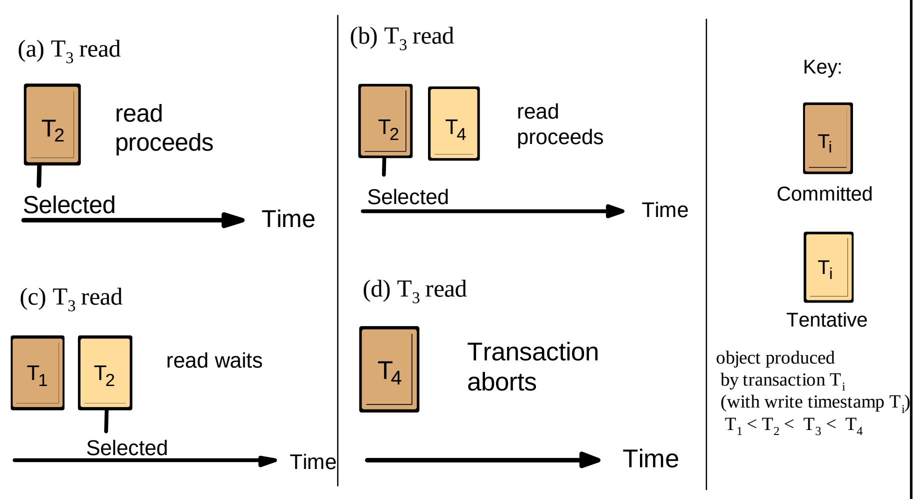
Write operations and timestamps



Timestamp ordering read rule

if (T_c > write timestamp on committed version of *D*) {
    let $D_{selected}$ be the version of *D* with the maximum write timestamp = T_c
    if ($D_{selected}$ is committed)
        perform *read* operation on the version $D_{selected}$
    else
        Wait until the transaction that made version $D_{selected}$ commits or aborts
        then reapply the *read* rule
} else
    Abort transaction T_c

Read operations and timestamps



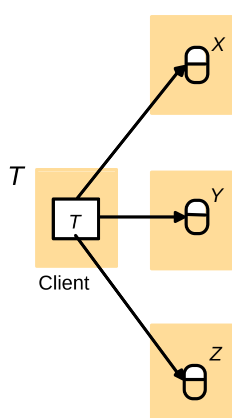
Timestamps in transactions *T* and *U*

T	*U*	*A* RTS	*A* WTS	*B* RTS	*B* WTS	*C* RTS	*C* WTS
		{}	***S***	{}	***S***	{}	***S***
openTransaction							
bal = b.getBalance()				{*T*}			
	openTransaction						
*b.setBalance(bal*1.1)*					***S***, *T*		
	bal = b.getBalance()						
	wait for T						
a.withdraw(bal/10)	•••		***S***, *T*				
commit	•••		***T***		***T***		
	bal = b.getBalance()			{*U*}			
	*b.setBalance(bal*1.1)*				***T***, *U*		
	c.withdraw(bal/10)						***S***, *U*

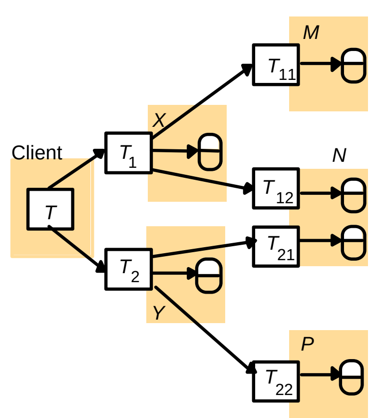
Timestamps and versions of objects

Distributed transactions

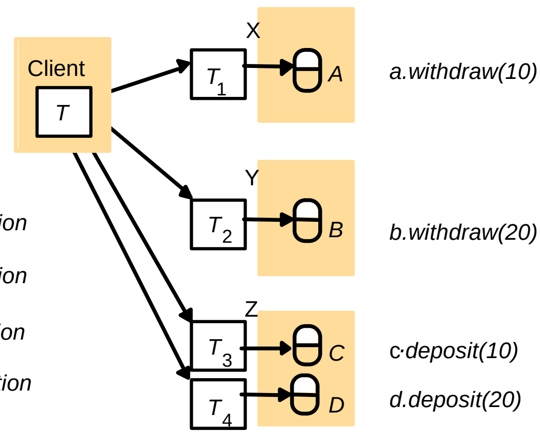
(a) Flat transaction



(b) Nested transactions

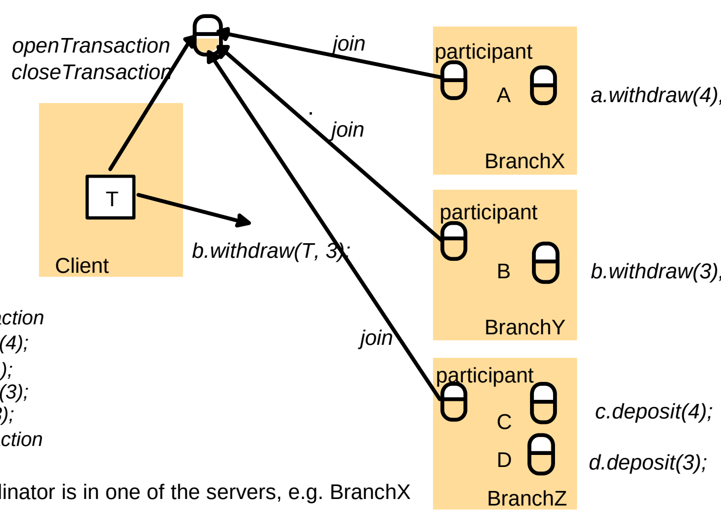


Nested banking transaction



T = openTransaction
openSubTransaction
a.withdraw(10);
openSubTransaction
b.withdraw(20);
openSubTransaction
c.deposit(10);
openSubTransaction
d.deposit(20);
closeTransaction

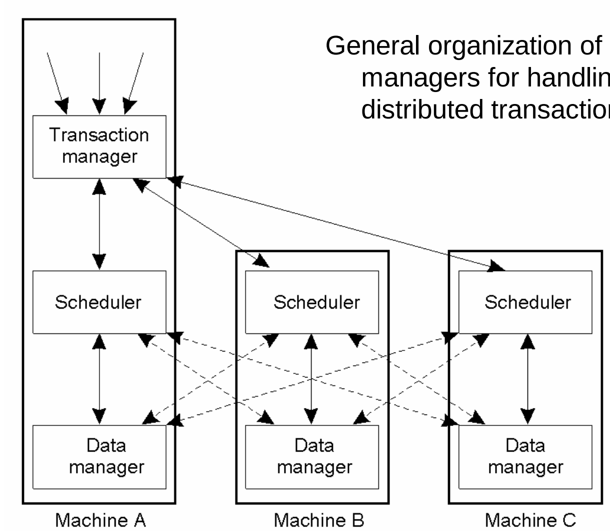
A distributed banking transaction



T = openTransaction
a.withdraw(4);
c.deposit(4);
b.withdraw(3);
d.deposit(3);
closeTransaction

Note: the coordinator is in one of the servers, e.g. BranchX

Concurrency Control for Distributed Transactions



General organization of managers for handling distributed transactions.

Concurrency Control for Distributed Transactions

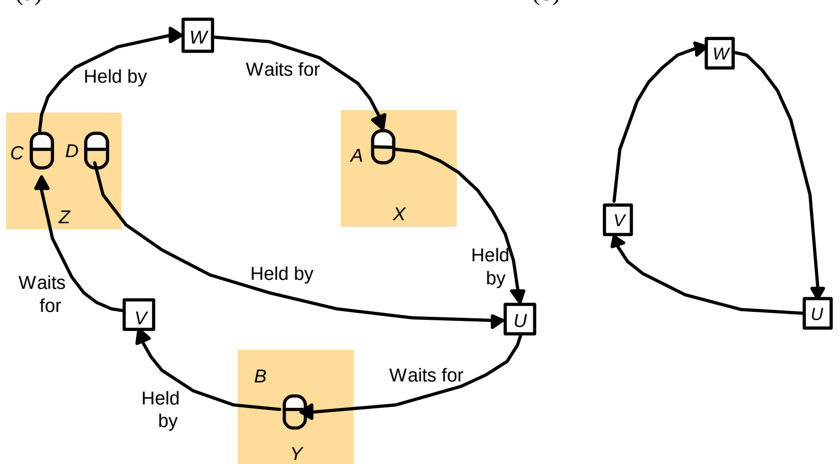
- Locking
  - Distributed deadlocks possible
- Timestamp ordering
  - Lamport time stamps
    - for efficiency it is required that timestamps issued by coordinators be roughly synchronized

Interleavings of transactions *U*, *V* and *W*

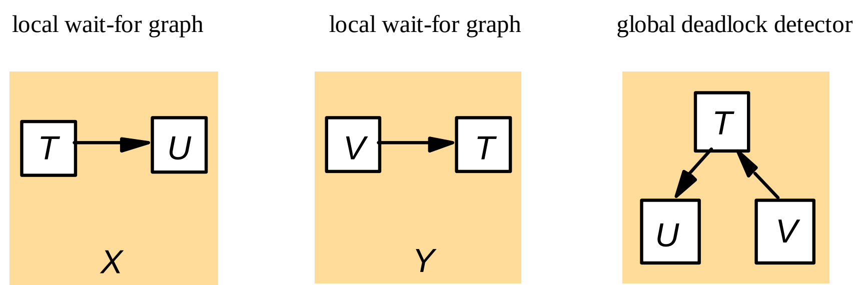
U		*V*		*W*	
d.deposit(10)	lock *D*				
		b.deposit(10)	lock *B* at *Y*		
a.deposit(20)	lock *A* at *X*				
				c.deposit(30)	lock *C* at *Z*
b.withdraw(30)	wait at *Y*				
		c.withdraw(20)	wait at *Z*		
				a.withdraw(20)	wait at *X*

Distributed deadlock

(a) (b)



Local and global wait-for graphs



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)

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.

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.

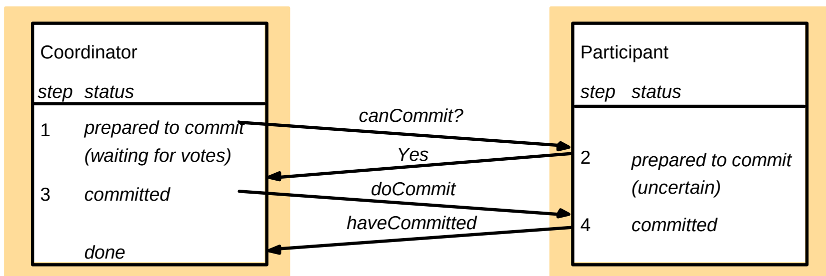
doCommit(trans)
Call from coordinator to participant to tell participant to commit its part of a transaction.

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

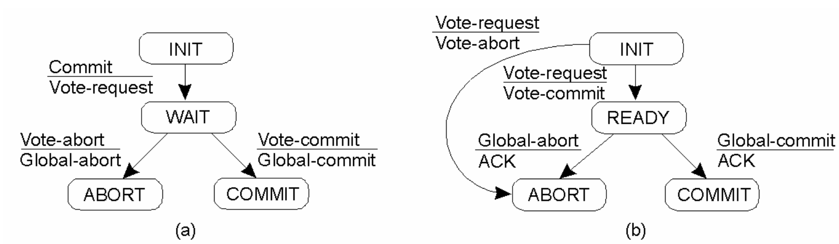
haveCommitted(trans, participant)
Call from participant to coordinator to confirm that it has committed the transaction.

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.

Communication in two-phase commit protocol



Two-Phase Commit (1)



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

Two-Phase Commit (2)

State of Q	**Action by P**
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Actions taken by a participant *P* when residing in state *READY* and having contacted another participant *Q*.

Two-Phase Commit (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;
}
```

Outline of the steps taken by the coordinator in a two phase commit protocol

Two-Phase Commit (4)

Steps taken by participant process in 2PC.

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;
}
```


Two-Phase Commit (5)

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

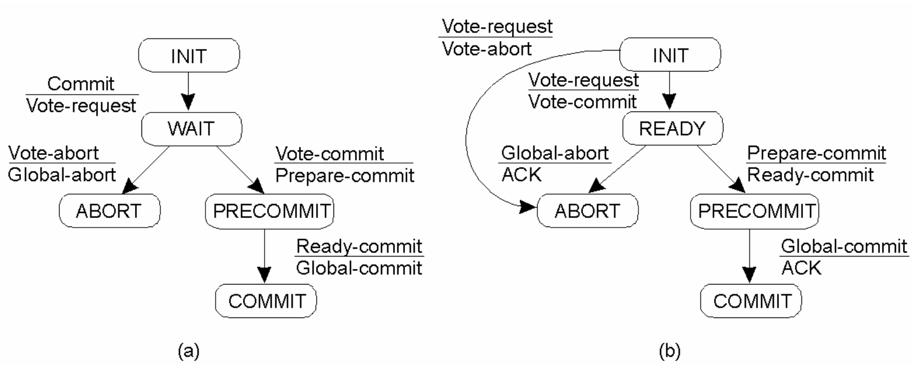
```
while true {
   wait until any incoming DECISION_REQUEST is received; /* remain blocked */
   read most recently recorded STATE from the local log;
   if STATE == GLOBAL_COMMIT
      send GLOBAL_COMMIT to requesting participant;
   else if STATE == INIT or STATE == GLOBAL_ABORT
      send GLOBAL_ABORT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

Steps taken for handling incoming decision requests.

Three Phase Commit

- Problem with 2PC
  - If coordinator crashes, participants cannot reach a decision, stay blocked until coordinator recovers
- 3PC
  - 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

Three-Phase Commit



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