Transactions

Distributed Software Systems

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
- Properties(ACID)
  - Atomicity – all or nothing
  - Consistency – a transaction must move the system from one consistent state to another
  - Isolation – intermediate effects must be invisible to other clients
  - Durability
- Concepts: commit, abort

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);

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

Lost Update Problem - 1

Consider the following transactions T and U with the initial account balances for a, b, and c as 100, 200, and 300 respectively.

T: *balance = b.getbalance()*
*b.setbalance(balance*1.1)*
a.withdrawn(balance/10)

U: *balance = b.getbalance()*
*b.setbalance(balance*1.1)*
c.withdrawn(balance/10)

600

If we execute T followed by U, we get balances of 80, 242 and 278

If we execute U followed by T, we get balances of 78, 242 and 280

Lost Update Problem - 2

Transaction *T*:	**Transaction *U*:**
balance = b.getBalance();	*balance = b.getBalance();*
*b.setBalance(balance*1.1);*	*b.setBalance(balance*1.1);*
a.withdraw(balance/10)	*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

In the above, we get balances of 80, 220 and 280 – U's update of b was lost

Lost Update Problem - 3

T:	**U:**
balance = b.getbalance()	*balance = b.getbalance()*
*b.setbalance(balance*1.1)*	*b.setbalance(balance*1.1)*
a.withdrawn(balance/10)	*c.withdrawn(balance/10)*

balance = b.getbalance()	
*b.setbalance(balance*1.1)*	
	balance = b.getbalance()
	*b.setbalance(balance*1.1)*
a.withdrawn(balance/10)	
	c.withdrawn(balance/10)

We get balances of 80, 242 and 278, same as T followed by U

The inconsistent retrievals problem

Transaction *V*		**Transaction *W*:**	
a.withdraw(100)			
b.deposit(100)		*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 *V* and *W*

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

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*

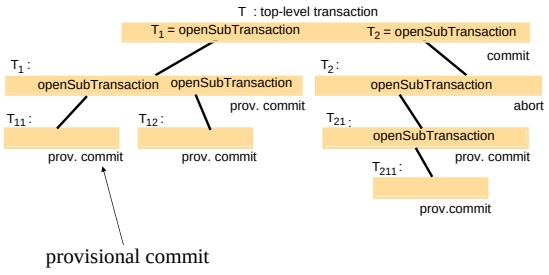
Transaction *T*:	**Transaction *U*:**
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

A dirty read when transaction *T* aborts

Transaction *T*:		**Transaction *U*:**	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*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!

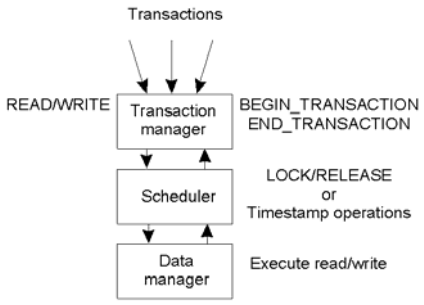
Nested transactions



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.

Concurrency Control



General organization of managers for handling transactions.

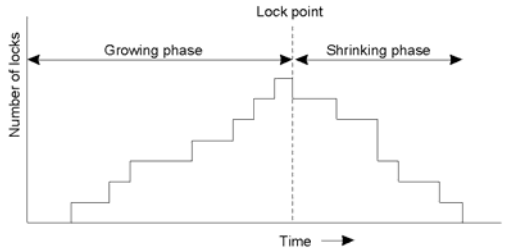
Schemes for Concurrency control

- 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

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*

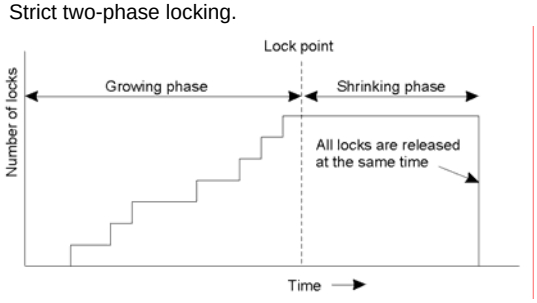
Two-Phase Locking (1)



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

Strict Two-Phase Locking (2)

- Strict two-phase locking.



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.

Lock compatibility

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

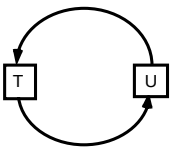
Operation Conflict rules:

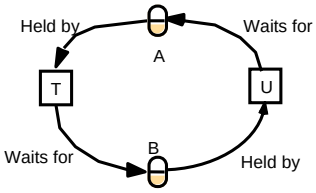
1. If a transaction T has already performed a read operation on a particular object, then a concurrent transaction U must not write that object until T commits or aborts
2. If a transaction T has already performed a read operation on a particular object, then a concurrent transaction U must not read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

Deadlock with write locks

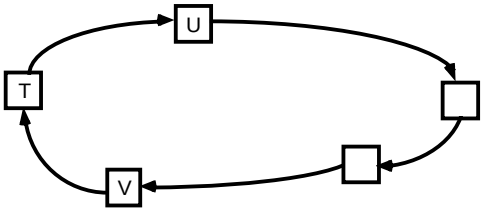
Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s	*a.withdraw(200);*	waits for *T*'s
	lock on *B*	•••	lock on *A*
•••		•••	
•••		•••	

The wait-for graph

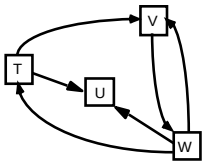


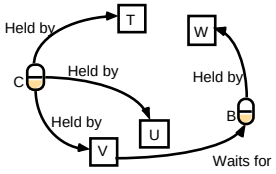


A cycle in a wait-for graph



Another wait-for graph





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

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*

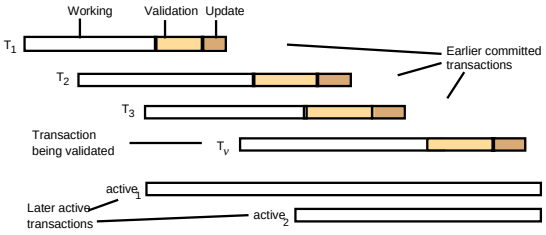
Optimistic Concurrency Control

- 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

Optimistic Concurrency Control

- Three phases:
  - **Working Phase** – transactions read and write private copies of objects (most recently committed)
  - **Validation Phase** – Once transaction is done, the transaction is validated to establish whether or not its operations on objects conflict with operations of other transactions on the same object. If not conflict, can commit; else some form of conflict resolution is needed and the transaction may abort.
  - **Update Phase** – if commit, private copies are used to make permanent change.

Validation of transactions



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

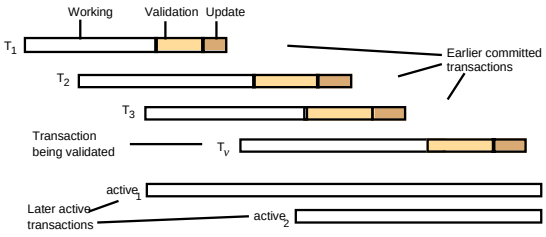
Backward validation of transaction T_v

```
boolean valid = true;
for (int Ti = startTn+1; Ti <= finishTn; Ti++){
        if (read set of Tv intersects write set of Ti) valid = false;
}
```

Forward validation of transaction T_v

```
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++){
        if (write set of Tv intersects read set of Tid) valid = false;
}
```


Validation of transactions



Schemes for Concurrency control

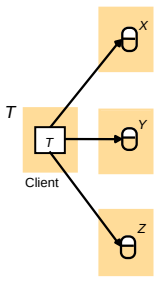
- Locking
- Optimistic concurrency control
- Time-stamp based concurrency control
  - Each timestamp is assigned a unique timestamp at the moment it starts
    - In distributed transactions, Lamport's timestamps can be used
  - Every data item has a timestamp
    - Read timestamp = timestamp of transaction that last read the item
    - Write timestamp = timestamp of transaction that most recently changed an item
  - A request to write to an object is valid only if that object was last read and written by earlier (wrt timestamps) transactions.
  - A request to read an object is valid only if that object was last written by an earlier transaction.

Distributed Transactions

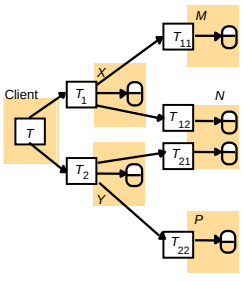
- Motivation
  - Provide distributed atomic operations at multiple servers that maintain shared data for clients
  - Provide recoverability from server crashes
- Properties
  - Atomicity, Consistency, Isolation, Durability (ACID)
- Concepts: commit, abort, distributed commit

Distributed Transactions

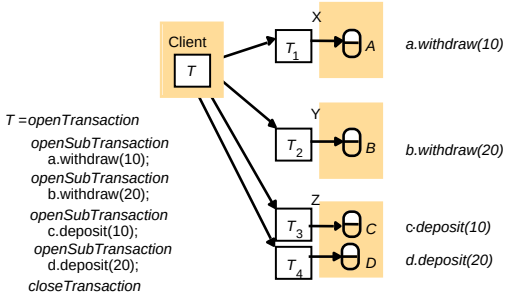
(a) Flat transaction



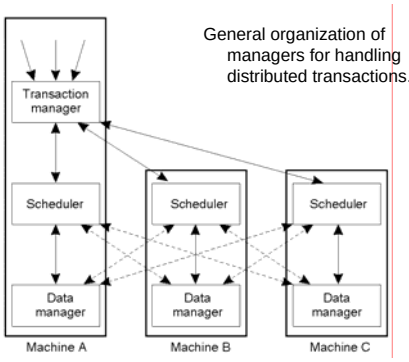
(b) Nested transactions



Nested banking transaction

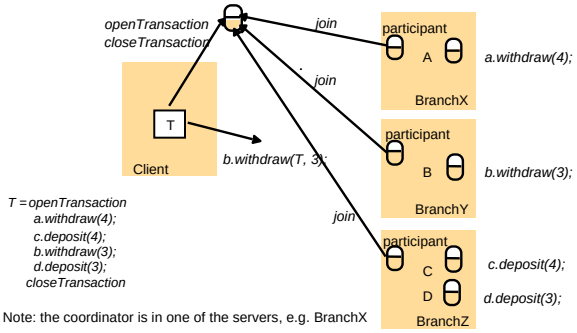


Concurrency Control for Distributed Transactions



General organization of managers for handling distributed transactions.

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

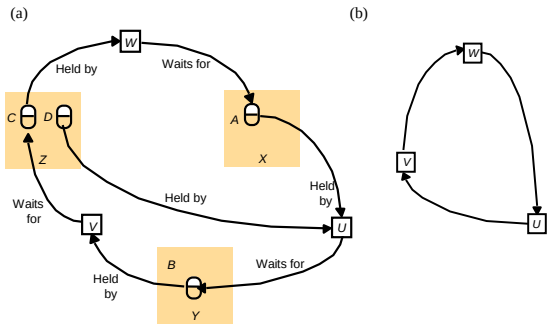
Concurrency Control for Distributed Transactions

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

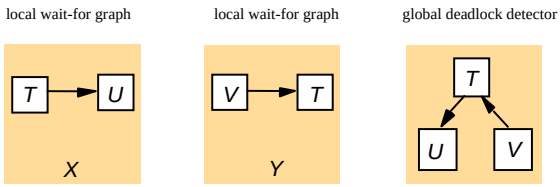
Interleavings of transactions *U*, *V* and *W*

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



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 - 1

Phase 1 (voting phase):

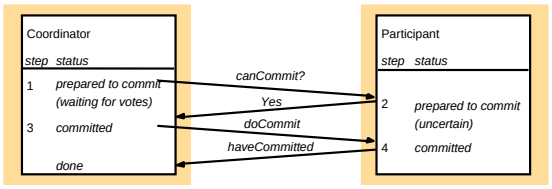
1. The coordinator sends a *canCommit? (VOTE_REQUEST)* request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote *Yes (VOTE_COMMIT)* or *No* (*VOTE_ABORT*) to the coordinator. Before voting *Yes*, it prepares to commit by saving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

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.

Communication in two-phase commit protocol



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.

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;
}
```

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

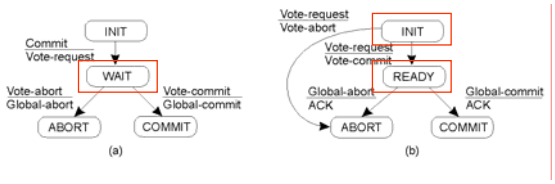
Two-Phase Commit protocol - 4

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 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.

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

Two-Phase Commit protocol - 7

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

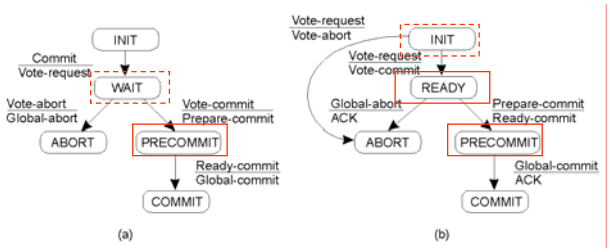
```
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send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

Steps taken for handling incoming decision requests.

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

Three-Phase Commit protocol - 2



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

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.