Timestamped Concurrency & Distributed Transactions

Pessimistic Concurrency Review

Transactions use locks to ensure isolation

- Per-object locks to increase parallelism
- Read/write locks to support read-only / read-mostly workloads
- Two-phase locking to ensure serial equivalence

Issues:

1. Deadlocks can arise
2. Concurrency pessimistically limited

Deadlock Strategies

- Timeout
  - Abort tx if it waits for locks too long
- Deadlock prevention
  - Lock all objects at transaction start
  - Use lock ordering
- Deadlock Detection
  - Maintain wait-for graph, look for cycle
  - Abort one transaction in cycle

Timeout issues:

- False positives / ~~false negatives~~
- Long detection delays

Prevention issues:

- Must know all locks at start of tx
- Reduced parallelism

Detection issues:

- High overhead
- Phantom cycles

Identify all conflicts

T1	T2
~~read X~~ lock X	~~read X~~ lock Y
write Y	write Y
read Z	read A
read W	read B
write V	write C
unlock all	unlock all

Timestamp Ordering

Assign each transaction a unique *timestamp (ts)*

- Serialize transactions according to timestamps

Keep track of timestamp last transaction to read and write an object

Invariants

1. If T reads O, last write timestamp must be lower than T
2. If T writes O, last read and write timestamp must be lower than T's

If T tries to read/write object with higher timestamp, abort and rollback

T (1)	U (2)	V (3)
read X (X.rts=1)		
write Y(Y.wts=1)		
	read X (X.rts=2)	
		read Y (Y.rts = 3)
		write X (X.wts=3)
	read Y (Y.rts=3)	
write X: abort!		

Timestamp Ordering Invariants

Let T1 and T2 have timestamps 1 and 2

Invariants enforce order T1 ; T2

I1: If T reads O, last write timestamp must be lower than T

- If T1 reads O after T2 writes O, T1 sees T2's write

T1 (1)	T2 (2)
	write O
read O	

Timestamp Ordering Invariants

Let T1 and T2 have timestamps 1 and 2

Invariants enforce order T1 ; T2

I1: If T reads O, last write timestamp must be lower than T

- If T1 reads O after T2 writes O, T1 sees T2's write

I2: If T writes O, last read and write timestamp must be lower than T's

- If T2 reads O before T1 writes O, T2 missed T1's write

T1	T2
	read O
write O	

Timestamp Ordering Invariants

Let T1 and T2 have timestamps 1 and 2

Invariants enforce order T1 ; T2

I1: If T reads O, last write timestamp must be lower than T

- If T1 reads O after T2 writes O, T1 sees T2's write

I2: If T writes O, last read and write timestamp must be lower than T's

- If T2 reads O before T1 writes O, T2 missed T1's write
- If T1 writes O after T2 writes O, T2's write has been lost

T1	T2
	write O
write O	

Thomas Write Rule

Let T1 and T2 have timestamps 1 and 2

Invariants enforce order T1 ; T2

I1: If T reads O, last write timestamp must be lower than T

- If T1 reads O after T2 writes O, T1 sees T2's write

I2: If T writes O, last read and write timestamp must be lower than T's

- If T2 reads O before T1 writes O, T2 missed T1's write
- If T1 writes O after T2 writes O, T2's write has been lost

If T writes O and last write timestamp > T's, skip write!

T1	T2
	write O = 7
write O = 5	

Should we abort or skip here?

T (1)	U (2)	V (3)
read X (X.rts=1)		
write Y(Y.wts=1)		
	read X (X.rts=2)	
		read Y (Y.rts = 3) V.deps.add(1)
		write X (X.wts=3)
	read Y (Y.rts=3) U.deps.add(1)	
write X: ??? abort()	abort()	abort()

Dependency Tracking

Start with X=0, Y=0, Z=0

T1	T2
Read X -> 0	
Write Y = 1	
	Read Y -> 1
	Write Z = 2
Read Z—ABORT!	

Dependency Tracking

Start with X=0, Y=0, Z=0

T2 has read value that was produced by aborted transaction!

T1	T2
~~Read X -> 0~~	
~~Write Y = 1~~	
	Read Y -> 1
	Write Z = 2
~~Read Z—ABORT!~~	

Dependency Tracking

Start with X=0, Y=0, Z=0

T2 has read value that was produced by aborted transaction!

When reading object O, add its ~~RTS~~ WTS to dependency list

At commit time, check dependency list

- If tx in dependency list has aborted, abort
- If tx in dependency list is still active, wait

T1	T2
~~Read X -> 0~~	
~~Write Y = 1~~ Y.WTS=1	
	Read Y -> 1 deps.add (Y.WTS)
	Write Z = 2
~~Read Z—ABORT!~~	

Commit

Timestamp Summary

```
read(X)
  if WTS(X) > myTS:
    abort()
  myDEPS.add(WTS(X))
  RTS(X) =
    max(RTS(X), myTS)
```

```
write(X)
  if RTS(X) > myTS:
    abort()
  if WTS(X) > myTS:
    return # skip write
  WTS(X) = myTS
```

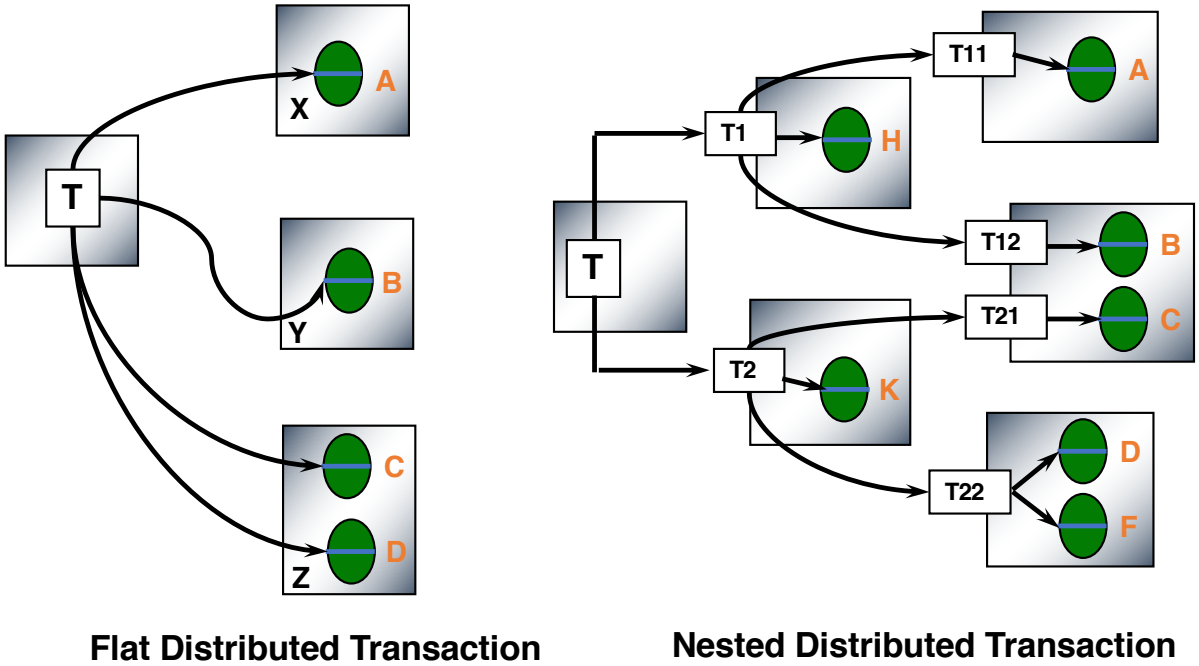
At commit time, wait for myDEPS to complete, abort if any has aborted

Concurrency Control: Summary

- *How to prevent transactions from affecting one another?*
- Goal: increase concurrency and transaction throughput while maintaining correctness (ACID).
- Target *serial equivalence.*
- Two approaches:
  - Pessimistic concurrency control: locking based.
    - read-write locks with two-phase locking and deadlock detection.
  - Optimistic concurrency control: abort if not serially equivalent.
    - timestamped ordering.

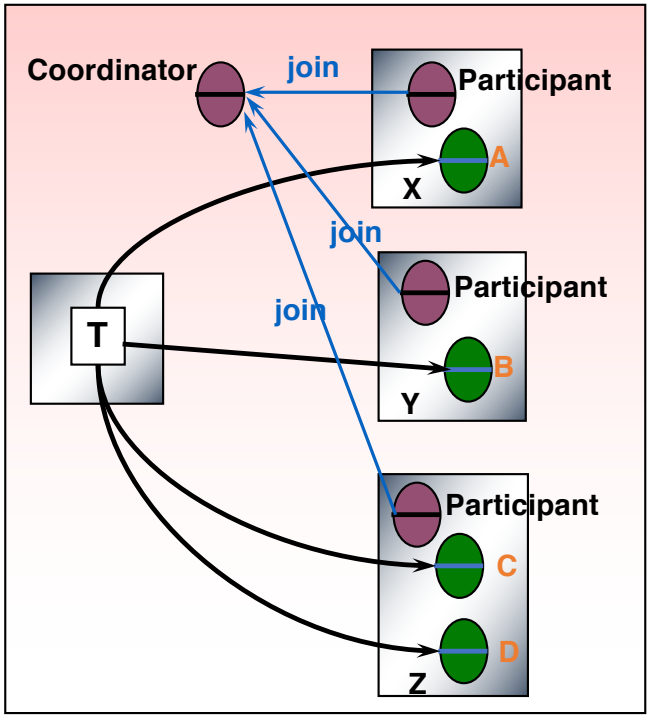
Distributed Transactions

- A transaction that invokes operations at several servers.

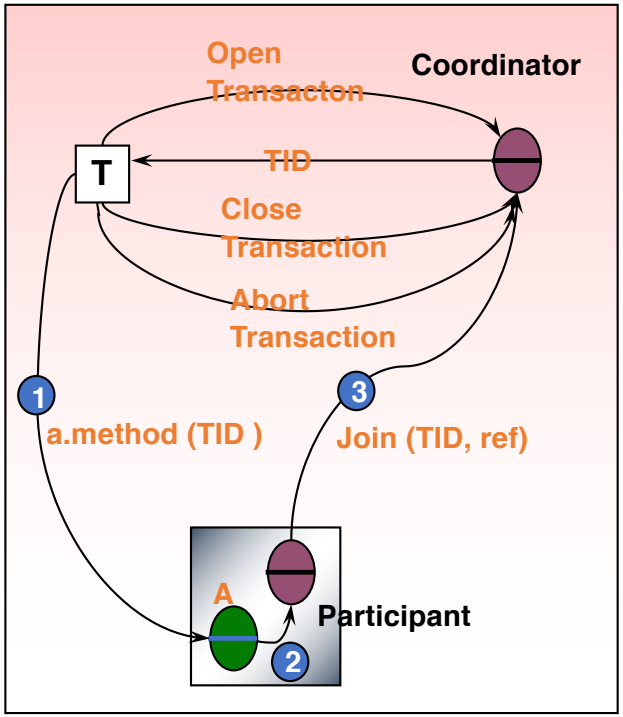


Flat Distributed Transaction **Nested Distributed Transaction**

Coordination in Distributed Transactions

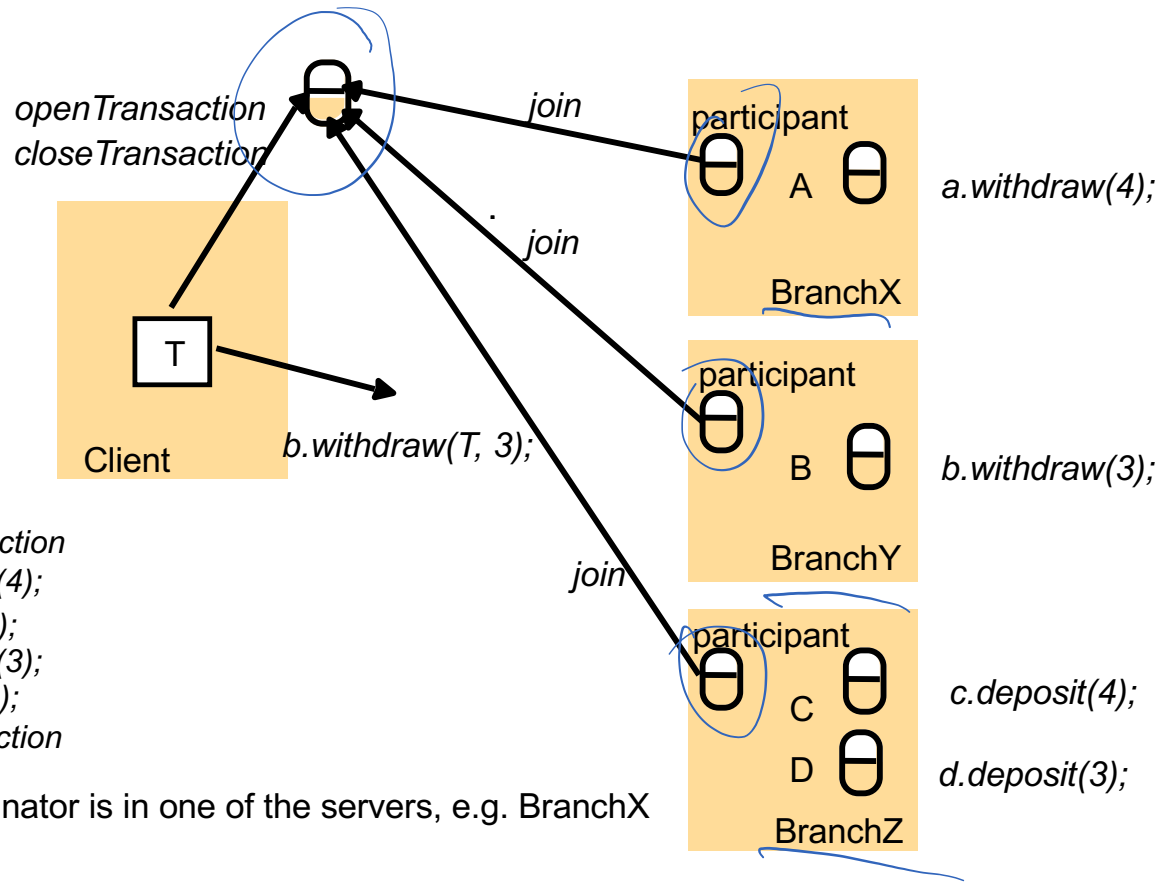


Coordinator & Participants



The Coordination Process

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

Distributed Transaction Challenges

- **Atomicity**: all-or-nothing
  - Must ensure atomicity across servers
- **Consistency**: invariants satisfied
  - Generally done locally, but may need to check non-local invariants at commit time
- **Isolation**: concurrent transactions serially equivalent
  - Locks at each server.
- **Durability**: results preserved after crashes
  - Each server keeps local recovery log

I. Locks in Distributed Transactions

- Each server is responsible for applying concurrency control to objects it stores.
- Servers are collectively responsible for serial equivalence of operations.
- Locks are held locally and cannot be released until all servers involved in a transaction have committed or aborted.
- Locks are retained during 2PC protocol.
- Since lock managers work independently, deadlocks are possible (likely?)

II. Atomic Commit Problem

- At some point, client executes closeTransaction()
  - Result -> commit, abort
- Atomicity requires all-or-nothing
  - All operations on all servers are committed, or
  - All operations on all servers are aborted
- What problem statement is this?

Consensus

Paxos / Raft

E.g., “I will grade Q2 on exam”

- Sending commands / update on *replicated state*
- Proposals *accepted* by default
- Proceed as long as majority of nodes live

2PC

E.g., “Can we all meet at 3pm?”

E.g., “Ready to submit MP2?”

- Coordinating distributed action
- Participants can *disagree*
- Wait or abort on missing participant

Atomic Commit Protocols

- First attempt: Coordinator decides
  - Pick commit or abort
  - Send message to all participants
  - (Retransmit until acknowledged)
- Problems?
  - Participant crashes before receiving commit message
  - Participant decides to abort (deadlock, other problems)

Two-phase Commit

- Phase 1: all participants vote to commit or abort
  - If you vote to commit, store partial results in permanent storage
  - If crash after vote to commit, can restore transaction later
- Phase 2:
  - Save result of vote in permanent storage
  - If **all** vote commit, multicast commit message
  - If **any** vote abort, multicast abort message

RPCs for Two-Phase Commit Protocol

Coordinator -> Participant

canCommit?(trans)-> Yes / No

Ask whether participant can commit a transaction. Participant replies with its vote.

doCommit(trans)

Tell participant to commit its part of a transaction.

doAbort(trans)

Tell participant to abort its part of a transaction.

Participant -> Coordinator

haveCommitted(trans, participant)

Confirm that participant has committed the transaction. (May not be required if getDecision() is used – see below)

getDecision(trans) -> Yes / No

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

2PC – Coordinator

- Phase 1:
  - Send *canCommit?* to all participants, tabulate replies
- Phase 2:

  must update log
  - If all votes are yes, send *doCommit* to all participants
  - If any votes are no, or any participant doesn't reply after timeout, send *doAbort* to all participants [who said yes]
  - Store commit decision to stable storage to support recovery

errs on side of safety

- Recovery after crash
  - If commit decision in stable storage, confirm with participants (push) or wait for getDecision (pull)
  - If getDecision called on commit not in log, reply *No*

2PC - Participant

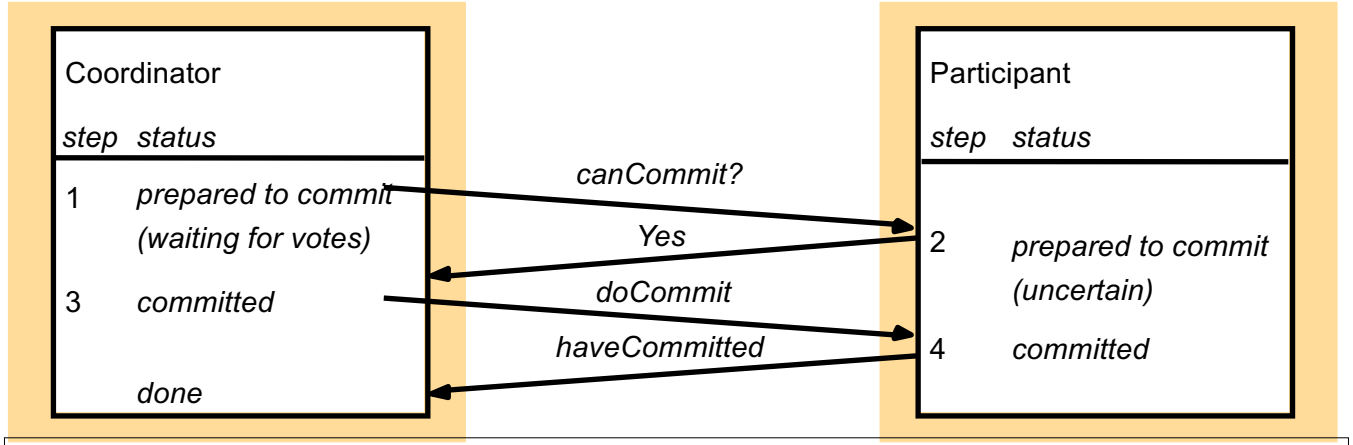
- Phase 1: receive *canCommit?*
  - If OK to commit, reply *Yes* and store transaction in permanent storage
  - If not OK, reply *No* and abort immediately
- Phase 2
  - If receive *doCommit*, commit transaction
  - If receive *doAbort,* abort transaction
  - If timeout, call *getDecision*
- Recovery after crash
  - If crashed after a Yes in Phase 1, call *getDecision*
  - If should commit, recover transaction from permanent storage and commit

The two-phase commit protocol

- *Phase 1 (voting phase):*
  - ordinator sends a *canCommit*? request to each of the participants in the transaction.
  - 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 its vote is *No*, the participant aborts immediately.

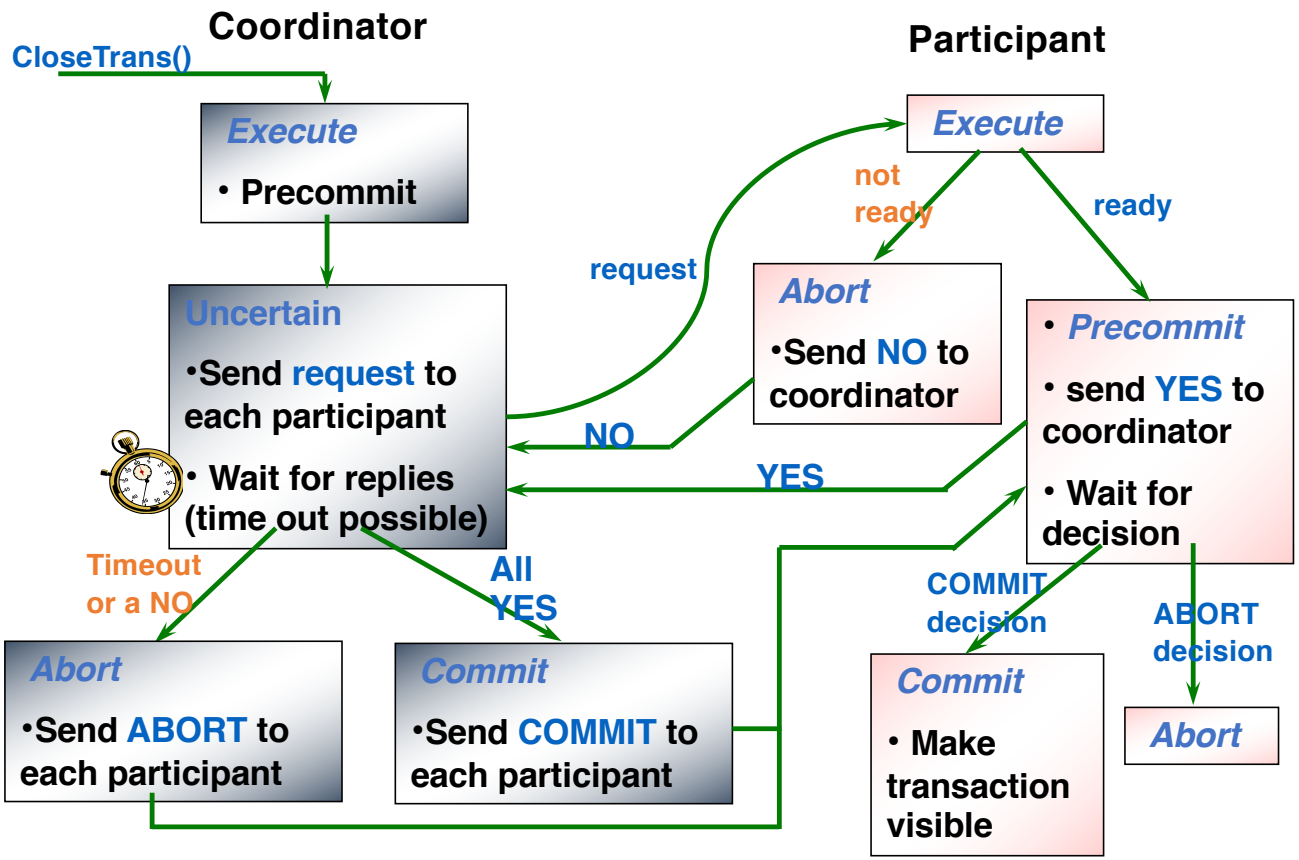
Recall that server may crash

- *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.



- **To deal with server crashes**
  - **Each participant saves tentative updates into permanent storage, right before replying yes/no in first phase. Retrievable after crash recovery.**
- **To deal with canCommit? loss**
  - **The participant may decide to abort unilaterally after a timeout (coordinator will eventually abort)**
- **To deal with Yes/No loss, the coordinator aborts the transaction after a timeout (pessimistic!). It must annouce doAbort to those who sent in their votes.**
- **To deal with doCommit loss**
  - **The participant may wait for a timeout, send a getDecision request (retries until reply received) – cannot unilaterally abort after having voted Yes but before receiving doCommit/doAbort!**

Two Phase Commit (2PC) Protocol



Summary

- Distributed Transactions
  - More than one server process (each managing different set of objects)
  - One server process marked out as coordinator
  - Atomic Commit: 2PC