Computer Science 425
Distributed Systems

Lecture 22

Distributed Transactions

Chapter 13.4, Chapter 14

Acknowledgement

- The slides during this semester are based on ideas and material from the following sources:
  - Slides prepared by Professors M. Harandi, J. Hou, I. Gupta, N. Vaidya, Y-Ch. Hu, S. Mitra.
  - Slides from Professor S. Gosh’s course at University o Iowa.

Administrative

- **MP2 posted October 5, 2009, on the course website,**
  - **Deadline November 6 (Friday)**
  - **Demonstration, 4-6pm, 11/6/2009**
    - » **Phone Demo**
    - » **Eclipse setup for Q&A (server/client)**
  - **Sign up !!! Friday 4-6pm in 216 SC**

Plan for Today

- **Exclusive locks**
- **Non-exclusive locks - 2P Locking**
- **Distributed Transactions**
  - **Atomic commit**
  - **Deadlock**
  - **Transaction recovery**

Recall Slide about Conflicting Operations

Transaction T1	Transaction T2
x= a.read()	
a.write(20)	
	y = b.read()
	b.write(30)
b.write(x)	
	z = a.read()

Conflicting Ops.

Non-serially equivalent interleaving of operations

Transaction T1	Transaction T2
x= a.read()	
a.write(20)	
	z = a.read()
b.write(x)	
	y = b.read()
	b.write(30)

Serially equivalent interleaving of operations

(why?)

Concurrent Transactions

- Transaction operations can run concurrently, provided "isolation" principle is not violated (same as interleaving ops.)
- Concurrent operations must be consistent:
  - If transaction *T* has executed a ***read*** operation on object *A*, a concurrent transaction *U* must not ***write*** to *A* until *T* commits or aborts.
  - If transaction *T* has executed a ***write*** operation on object *A*, a concurrent *U* must not ***read or write*** to *A* until *T* commits or aborts.
- How to implement this?
  - First cut: locks

Example: Concurrent Transactions

❖ Exclusive Locks

Transaction T1

OpenTransaction()

balance = b.getBalance() — Lock B

b.setBalance = (balance*1.1)

a.withdraw(balance* 0.1) — Lock A

CloseTransaction() — UnLock B, UnLock A

Transaction T2

OpenTransaction()

balance = b.getBalance() — WAIT on B

...

...

Lock B

b.setBalance = (balance*1.1)

c.withdraw(balance*0.1) — Lock C

CloseTransaction() — UnLock B, UnLock C

Conflict Prevention: Locking

- **Transaction managers set locks on objects they need. A concurrent transactions cannot access locked objects.**
- **Two phase locking:**
  - **In the first (growing) phase, new locks are acquired, and in the second (shrinking) phase, locks are released.**
  - **A transaction is not allowed acquire *any* new locks, once it has released any one lock.**
    - **Serial Equivalence**
- **Strict two phase locking:**
  - **Locking is performed when the requests to read/write are about to be applied.**
  - **Unlocking is performed by the commit/abort operations of the transaction coordinator.**
    - **To prevent dirty reads and premature writes, a transaction waits for another to commit/abort**
- **Use of separate read and write locks is more efficient than a single exclusive lock.**

2P Locking: Non-exclusive locks

non-exclusive lock compatibility

Lock already set	Lock requested: read	Lock requested: write
none	OK	OK
read	OK	WAIT
write	WAIT	WAIT

- A read lock is promoted to a write lock when the transaction needs write access to the same object.
- A read lock shared with other transactions' read lock(s) cannot be promoted. Transaction waits for other read locks to be released.
- Cannot demote a write lock to read lock during transaction – violates the 2P principle ?

Locking Procedure in Strict-2P Locking

- **When an operation accesses an object:**
  - **if the object is not already locked, lock the object & proceed.**
  - **if the object has a conflicting lock by another transaction, wait until object is unlocked.**
  - **if the object has a non-conflicting lock by another transaction, share the lock & proceed.**
  - **if the object has a lower lock by the same transaction,**
    - **if the lock is not shared, promote the lock & proceed**
    - **else, wait until shared locks are released, then lock & proceed**
- **When a transaction commits or aborts:**
  - **release all locks set by the transaction**

Example: Concurrent Transactions

- Non-exclusive Locks

Transaction T1		Transaction T2	
OpenTransaction()			
balance = b.getBalance()	R-Lock B	OpenTransaction()	
		balance = b.getBalance()	R-Lock B
		b.setBalance =balance*1.1	
		Cannot Promote lock on B, Wait	
Commit			
		Promote lock on B	
		…	

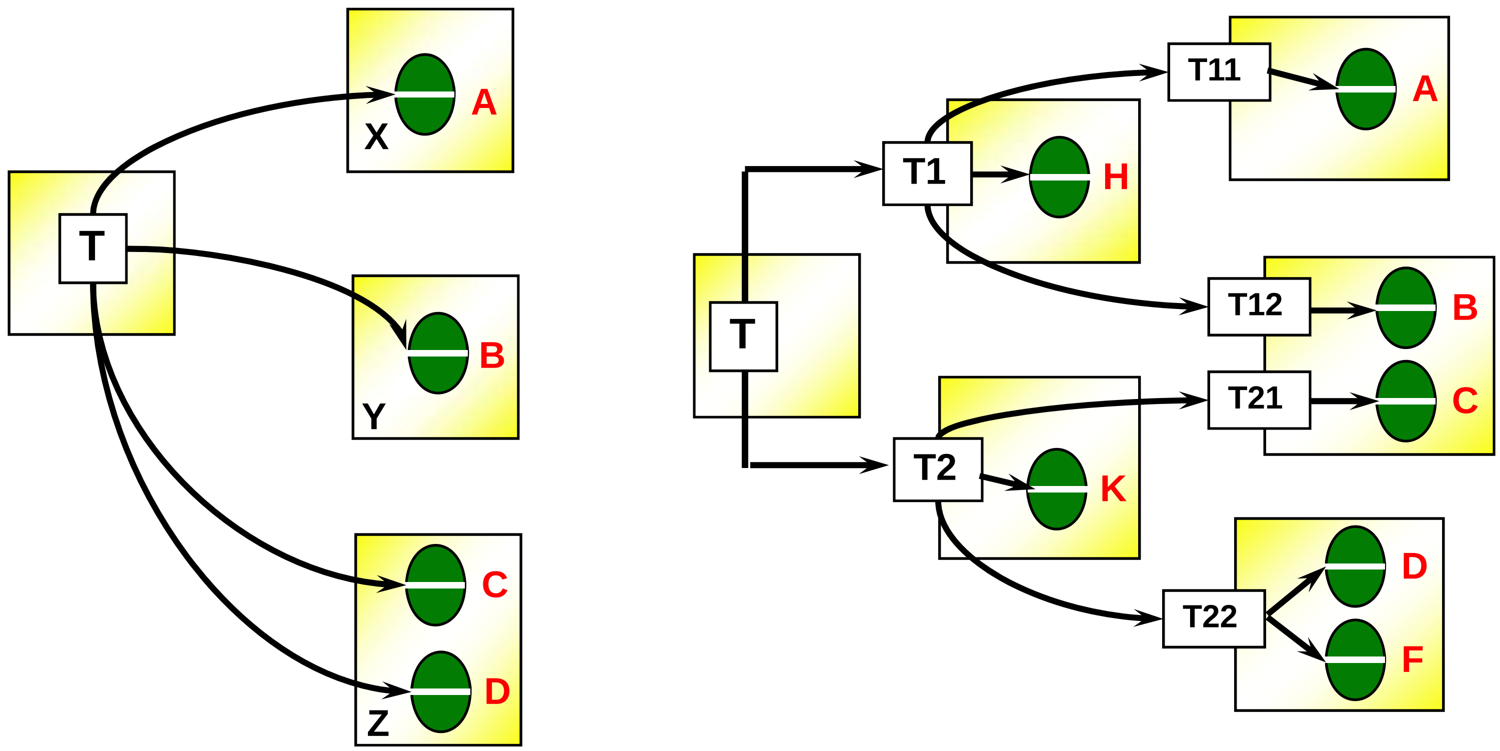
Distributed Transactions

- We have so far looked at:

- Multiple clients and single server
- Locking approaches
- …

Distributed Transactions

- A transaction (flat or nested) that invokes operations in several servers.

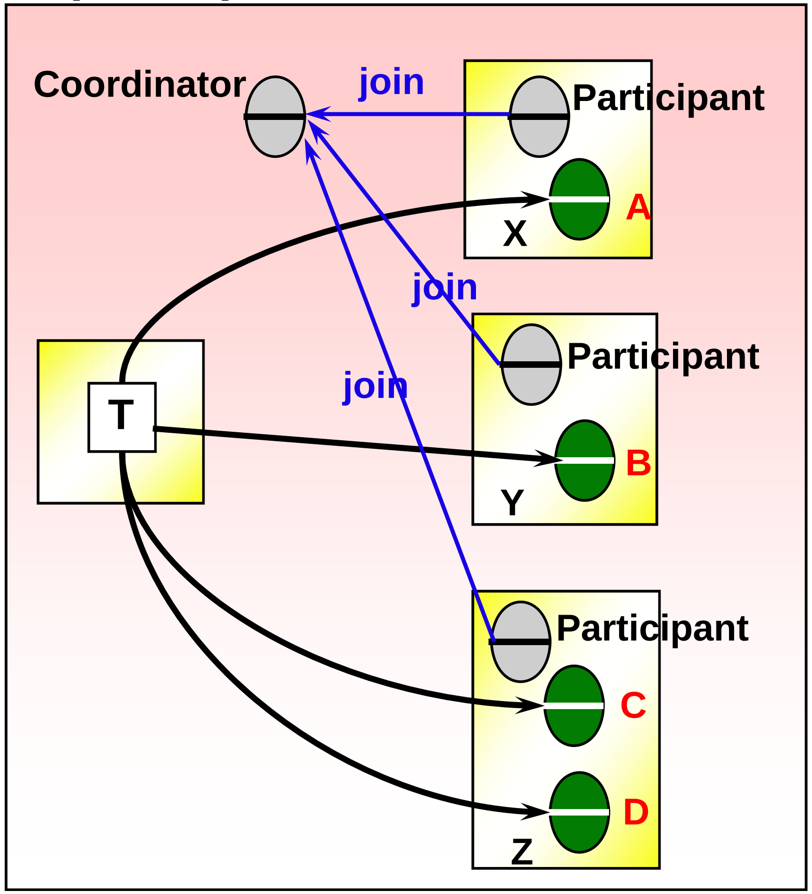


Flat Distributed Transaction

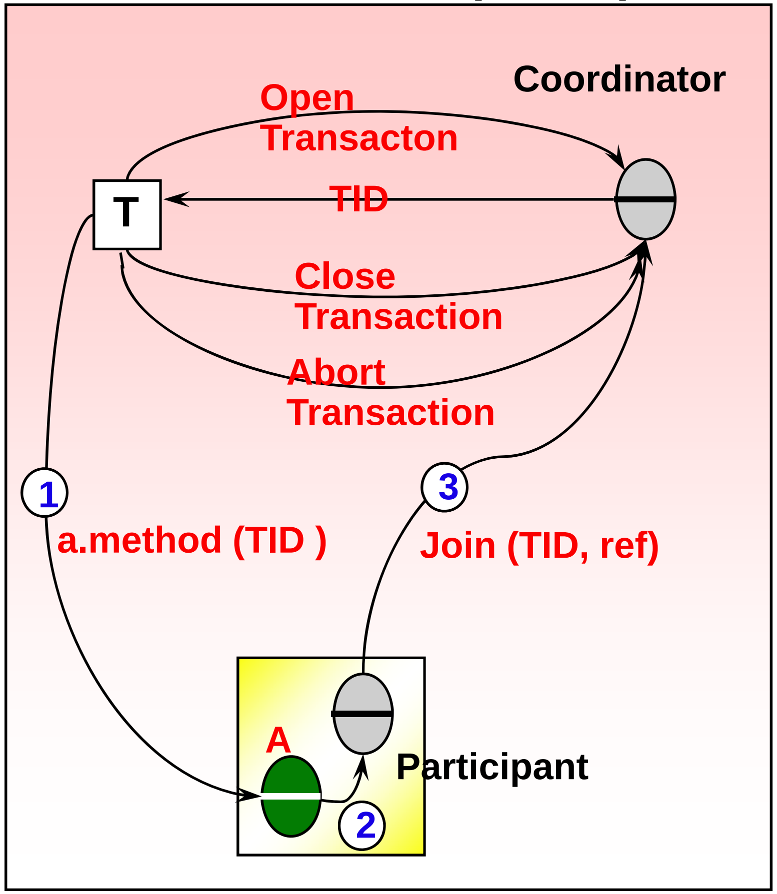
Nested Distributed Transaction

Coordination in Distributed Transactions

Each server has a special *participant* process. Coordinator process (leader) resides in one of the servers, talks to trans. & participants.

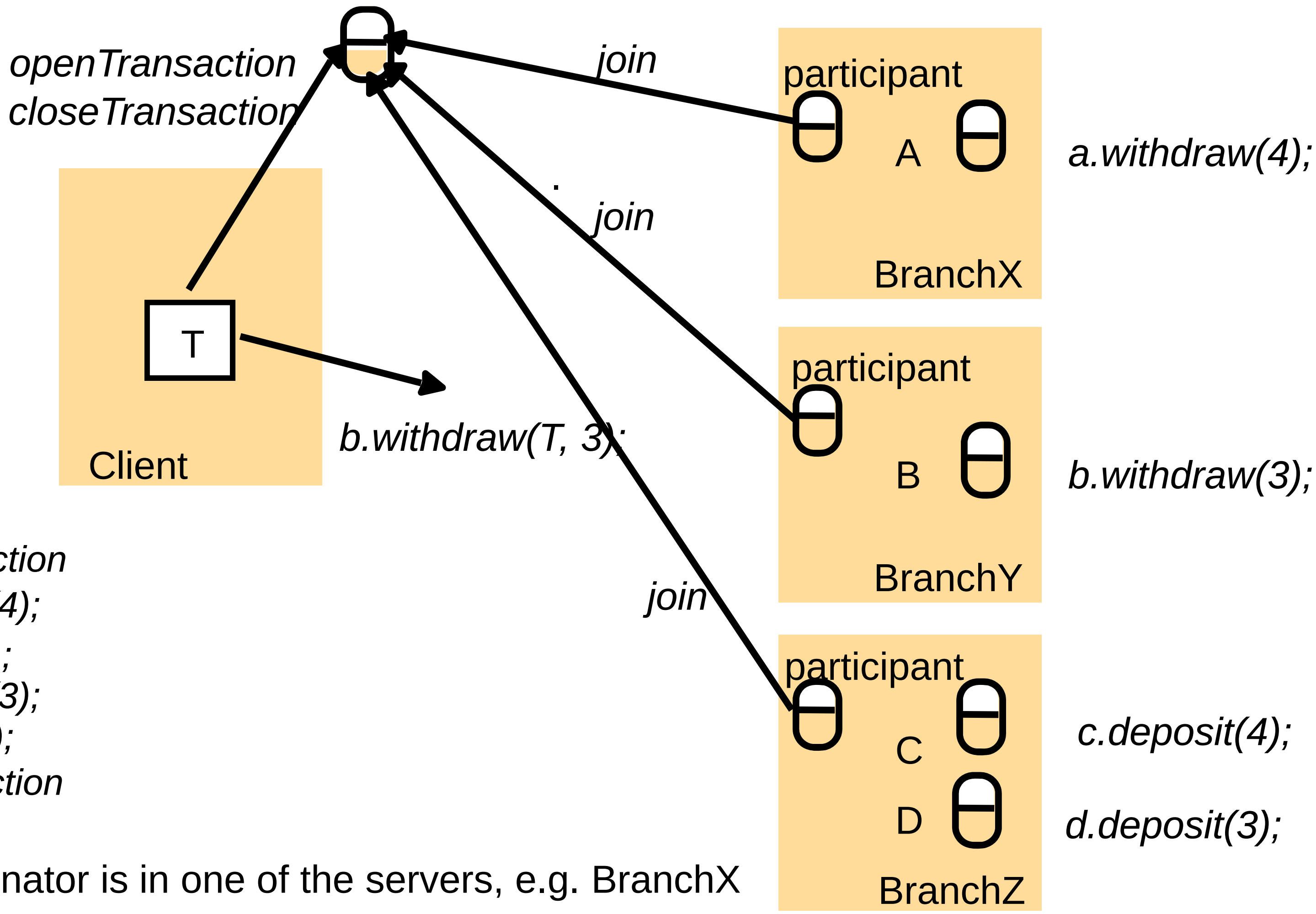


Coordinator & Participants



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

I. Atomic Commit Problem

- **Atomicity principle requires that either all the distributed operations of a transaction complete or all abort.**
- **At some stage, client executes *closeTransaction()*. Now, atomicity requires that either *all* participants and the coordinator commit or *all* abort.**
- **What problem statement is this?**

Atomic Commit Protocols

- Consensus, but it's impossible in asynchronous networks!
- So, need to ensure *safety property* in real-life implementation.
- In a ***one-phase commit*** protocol, the coordinator communicates either commit or abort, to all participants until all acknowledge.
  - Doesn't work when a participant crashes before receiving this message (partial transaction results are lost).
  - Does not allow participant to abort the transaction, e.g., under deadlock.
- In a ***two-phase* protocol**
  - First phase involves coordinator collecting commit or abort vote from each participant (which stores partial results in permanent storage).
  - If all participants want to commit and no one has crashed, coordinator multicasts commit message
  - If any participant has crashed or aborted, coordinator multicasts abort message to all participants

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.

The two-phase commit protocol

Phase 1 (voting phase):

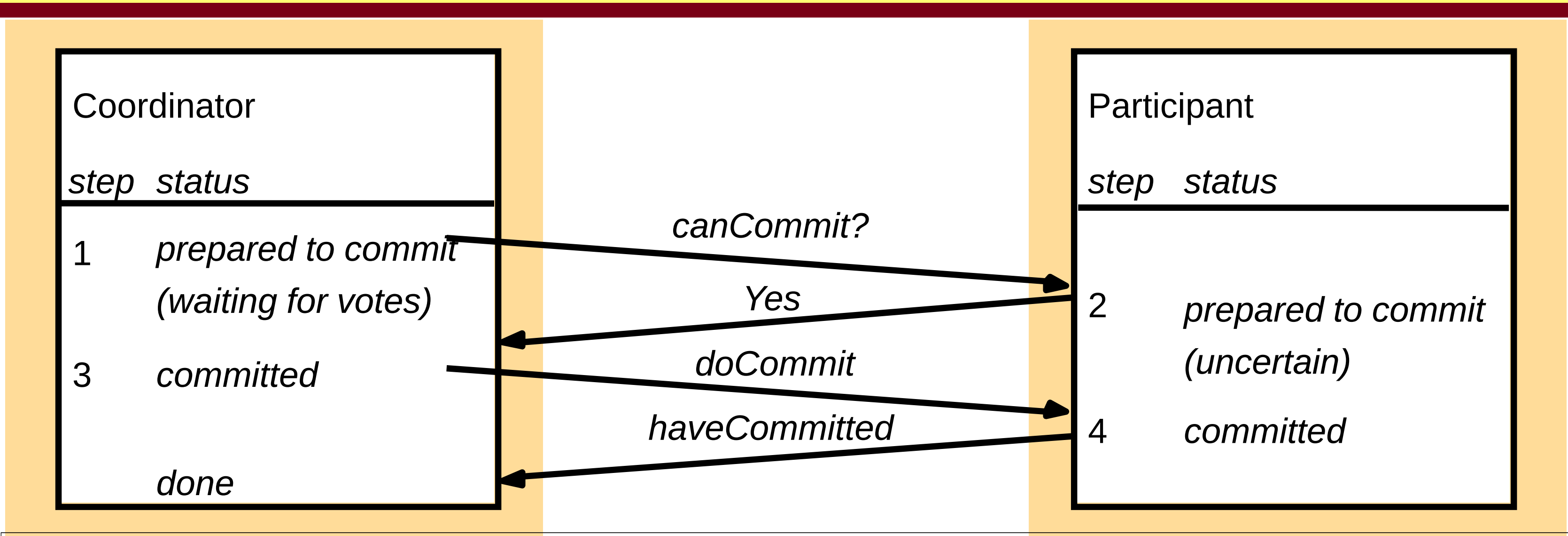
1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the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

Recall that server may crash

Phase 2 (completion according to outcome of vote):

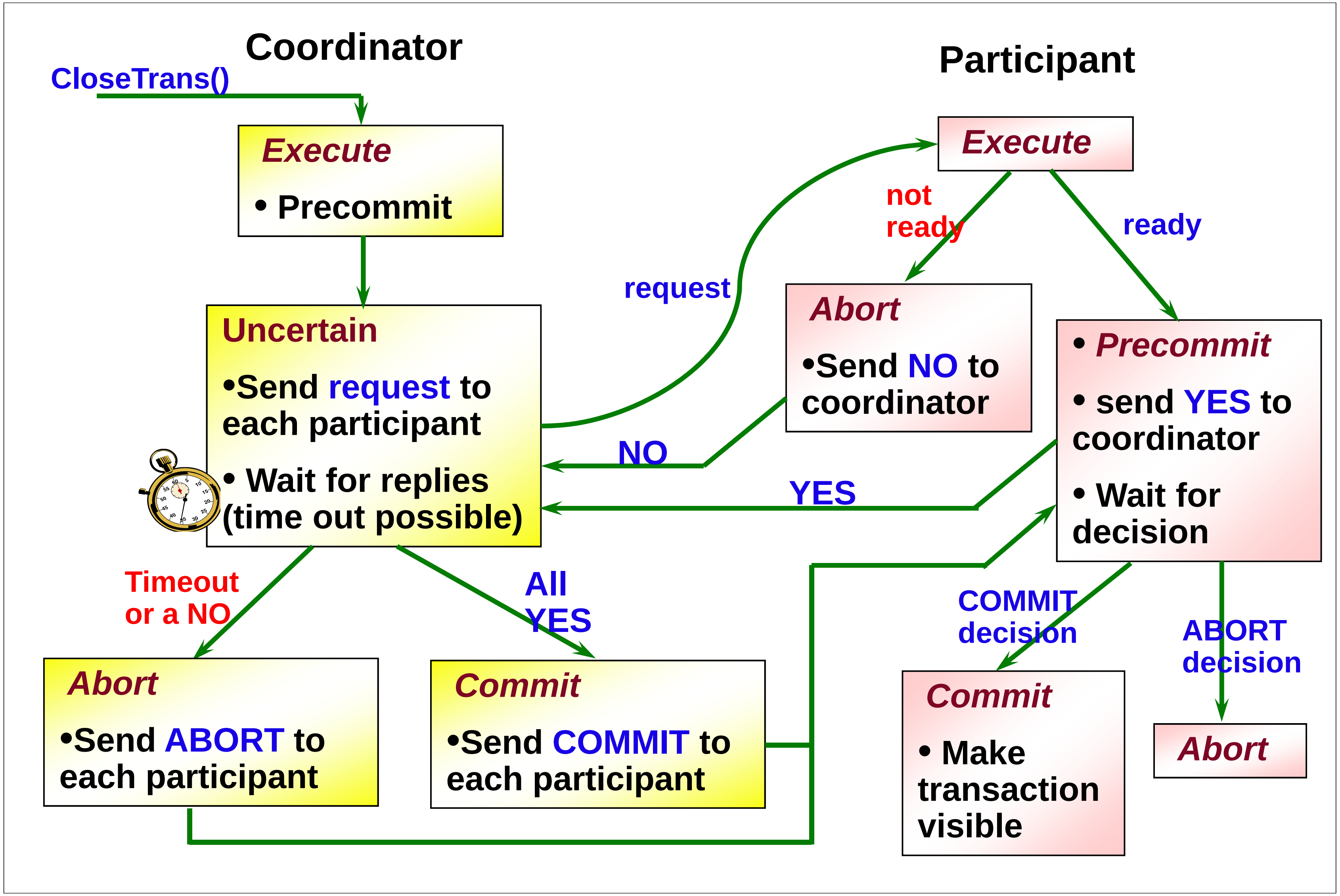
3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are *Yes*, the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends *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.

Communication in Two-Phase Commit Protocol



- To deal with server crashes @ participants
  - 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 abort after having voted Yes but before receiving doCommit/doAbort!

Two Phase Commit (2PC) Protocol



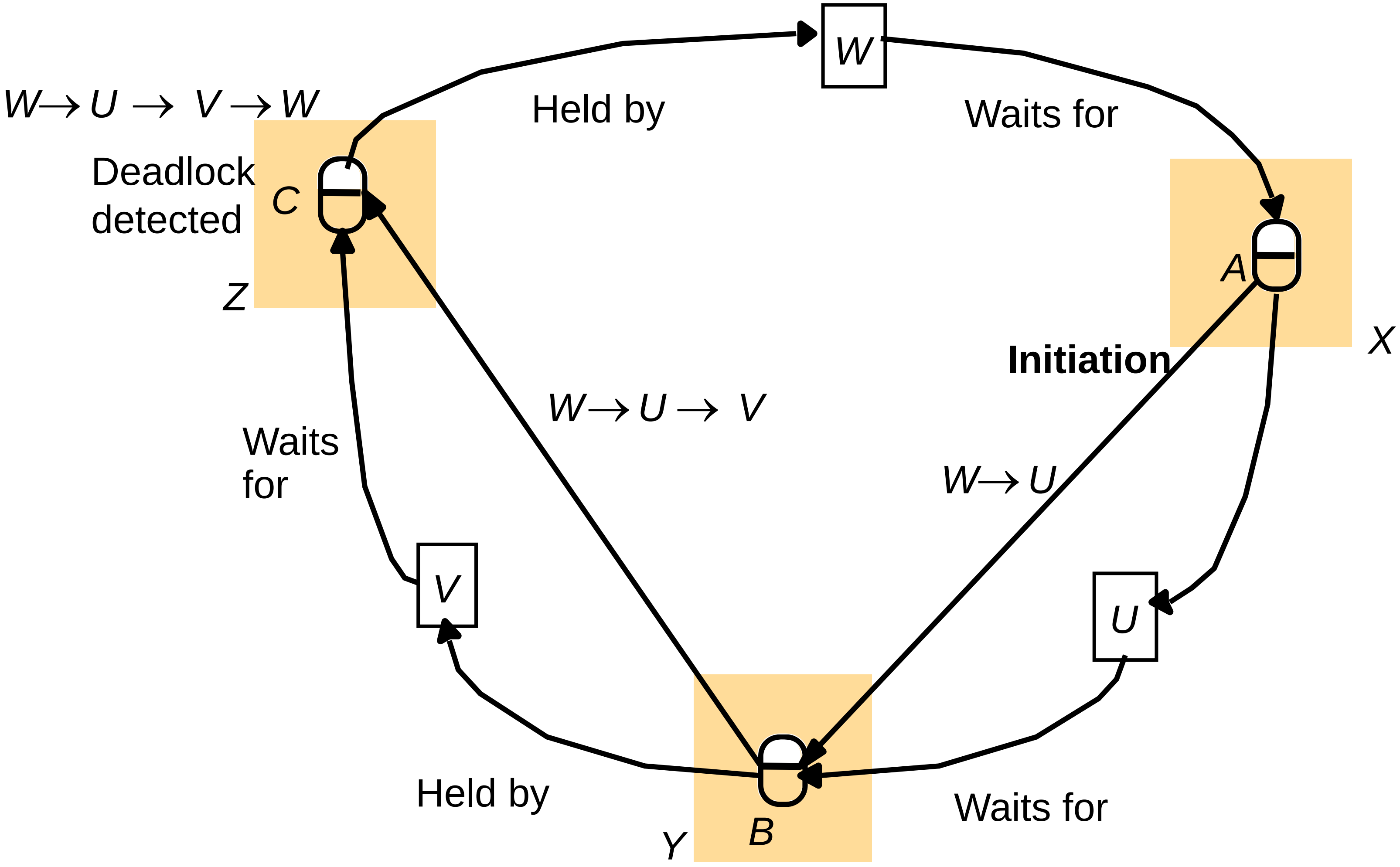
II. Locks in Distributed Transactions

- Each server is responsible for applying concurrency control to its objects.
- 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 (very?) likely.

Distributed Deadlocks

- The **wait-for graph** in a distributed set of transactions is held partially by each server
- To find **cycles** in a distributed wait-for graph, one option is to use a central coordinator:
  - Each server reports updates of its wait-for graph
  - The coordinator constructs a global graph and checks for cycles
- Centralized deadlock detection suffers from usual comm. **overhead + bottleneck** problems.
- In **edge chasing,** servers collectively make the global wait-for graph and detect deadlocks :
  - Servers forward "probe" messages to servers in the edges of wait-for graph, pushing the graph forward, until cycle is found.

Probes Transmitted to Detect Deadlock



Edge Chasing

- **Initiation:** When a server S_1 notes that a transaction T starts waiting for another transaction U, where U is waiting to access an object at another server S_2, it initiates detection by sending <T→U> to S_2.
- **Detection:** Servers receive probes and decide whether deadlock has occurred and whether to forward the probes.
- **Resolution:** When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.
- Phantom deadlocks=false detection of deadlocks that don't actually exist
  - Edges may disappear. So, all edges in a "detected" cycle may not have been present in the system all at the same time.

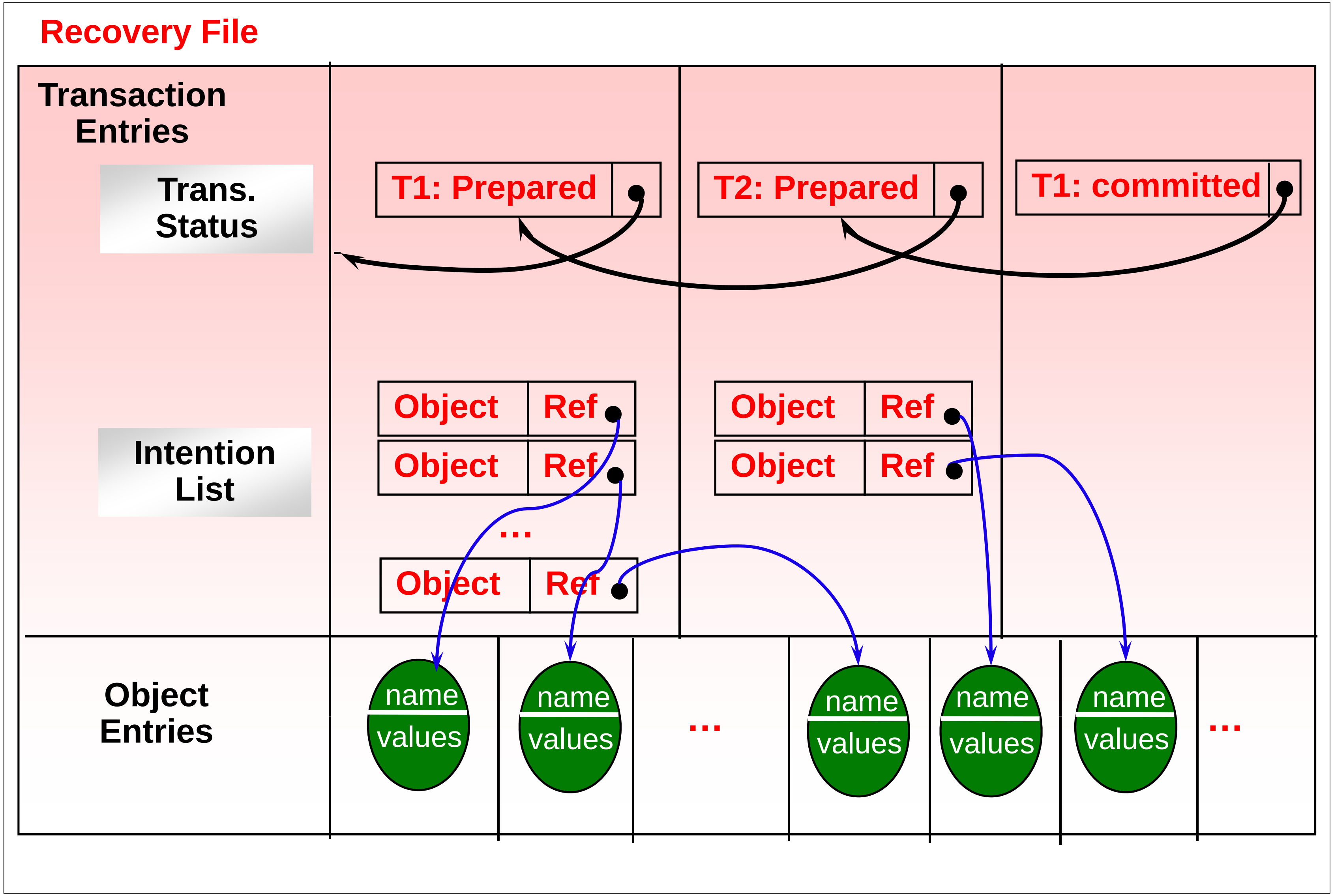
Transaction Priority

- In order to ensure that only one transaction in a cycle is aborted, transactions are given priorities (e.g., inverse of timestamps) in such a way that all transactions are totally ordered.
- When a deadlock cycle is found, the transaction with the lowest priority is aborted. Even if several different servers detect the same cycle, only one transaction aborts.
- Transaction priorities can be used to limit probe messages to be sent only to lower prio. trans. and initiating probes only when higher prio. trans. waits for a lower prio. trans.
  - Caveat: suppose edges were created in order 3->1, 1->2, 2->3. Deadlock never detected.
  - Fix: whenever an edge is created, tell everyone (broadcast) about this edge. May be inefficient.

III. Transaction Recovery

- **Recovery is concerned with:**
  - **Object (data) durability: saved permanently**
  - **Failure Atomicity: effects are atomic even when servers crash**
- **Recovery Manager's tasks**
  - **To save objects (data) on permanent storage for committed transactions.**
  - **To restore server's objects after a crash**
  - **To maintain and reorganize a recovery file for an efficient recovery procedure**
  - **Garbage collection in recovery file**

The Recovery File

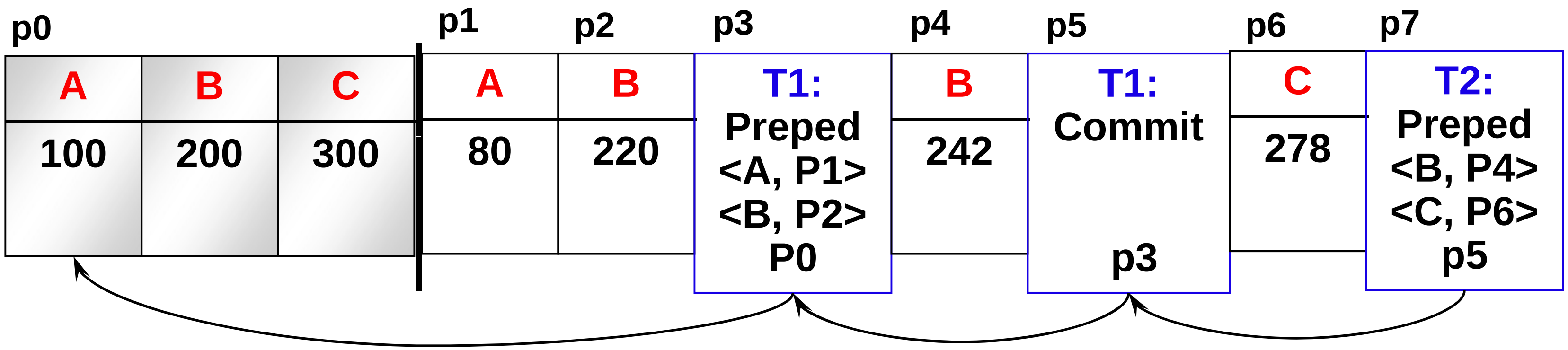


Example: Recovery File

- "Logging"=appending
- Write entries at prepared-to-commit and commit points
- Writes at commit points are ***forced***

a: 100 b: 200 c: 300

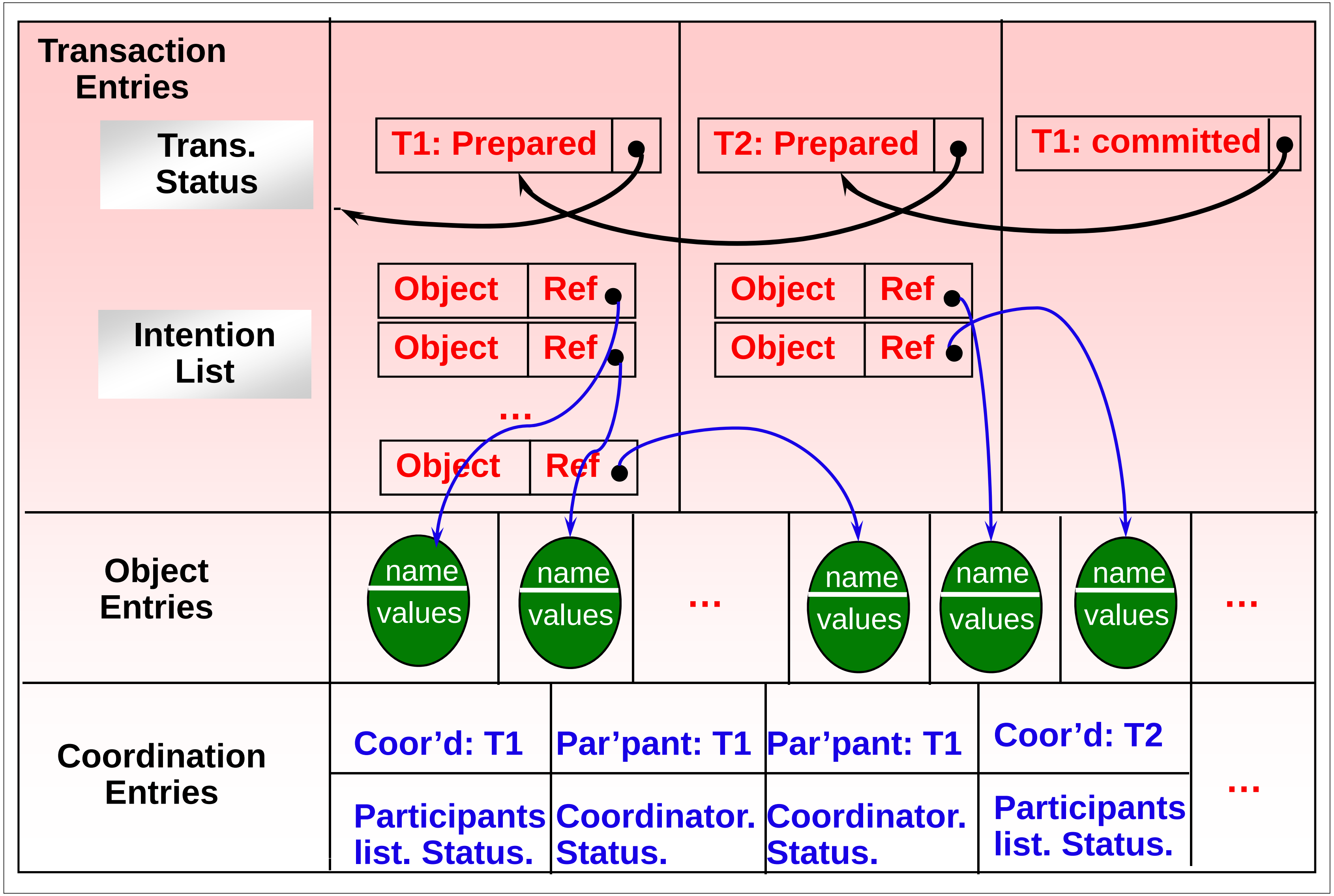
Transaction T1	Transaction T2
balance = b.getBalance()	
b.setBalance = (balance*1.1)	
b: 220	
	balance = b.getBalance()
	b.setBalance(balance*1.1)
	b: 242
a.withdraw(balance* 0.1)	
a: 80	
	c.withdraw(balance*0.1)
	c: 278



Using the Recovery File after a Crash

- When a server recovers, it sets default initial values for objects and then hands over to recovery manager.
- Recovery manager should apply only those updates that are for committed transactions. Prepared-to-commit transactions are aborted.
- Recovery manager has two options:
  1. Read the recovery file forward and update object values
  2. Read the recovery file backwards and update object values
     - Advantage: each object updated exactly once (hopefully)
- Server may crash during recovery
  - Recovery operations needs to be idempotent

The Recovery File for 2PC



Summary

- **Distributed Transactions**
  - **More than one server process (each managing different set of objects)**
  - **One server process marked out as coordinator**
  - **Atomic Commit: 2PC**
  - **Deadlock detection: Edge chasing**
  - **Transaction Recovery: Recovery file**

- **Reading for this lecture was: Chapter 13.4 and Chapter 14**