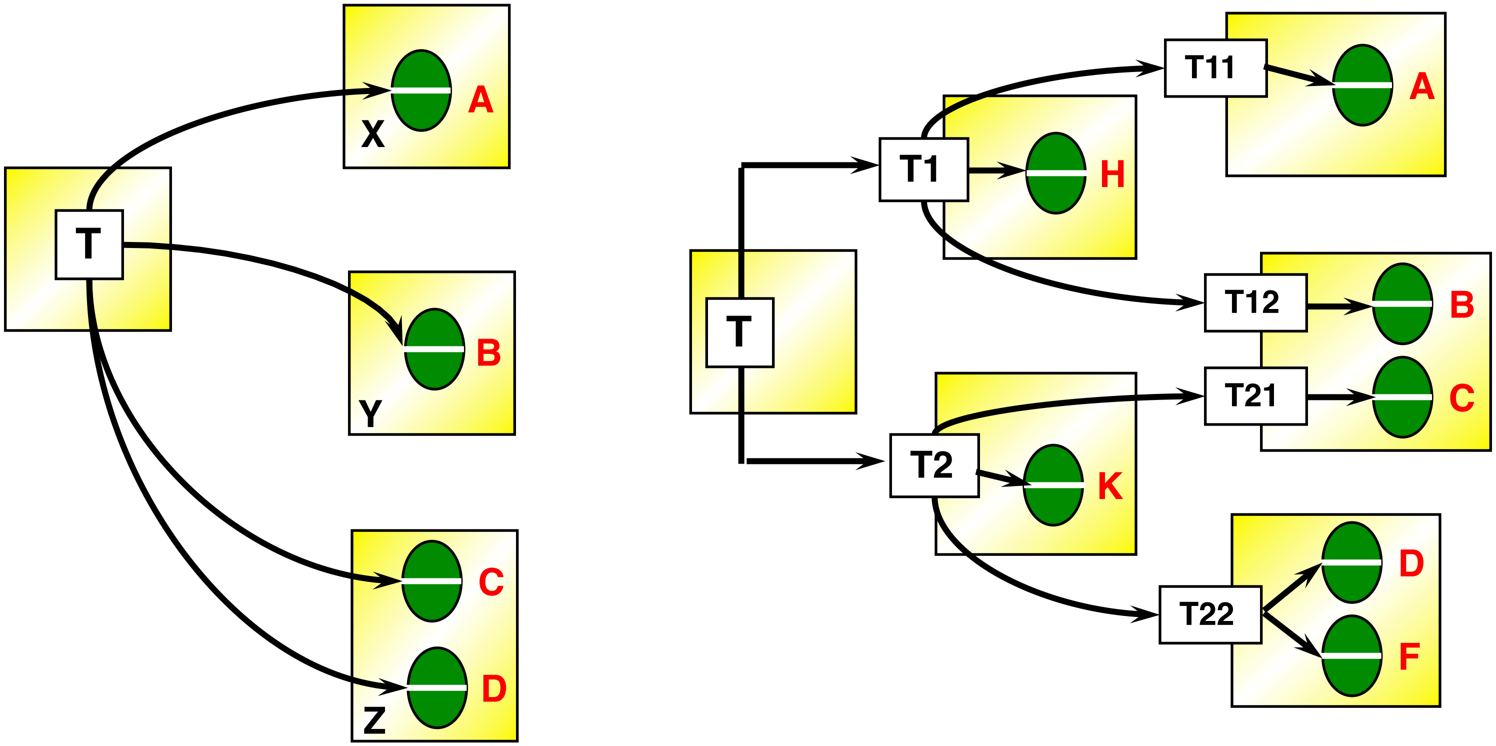
Distributed Transactions

CS425 /CSE424/ECE428 – Distributed Systems – Fall 2011

Material derived from slides by I. Gupta, M. Harandi, J. Hou, S. Mitra, K. Nahrstedt, N. Vaidya

Distributed Transactions

- A transaction that invokes operations at several servers.

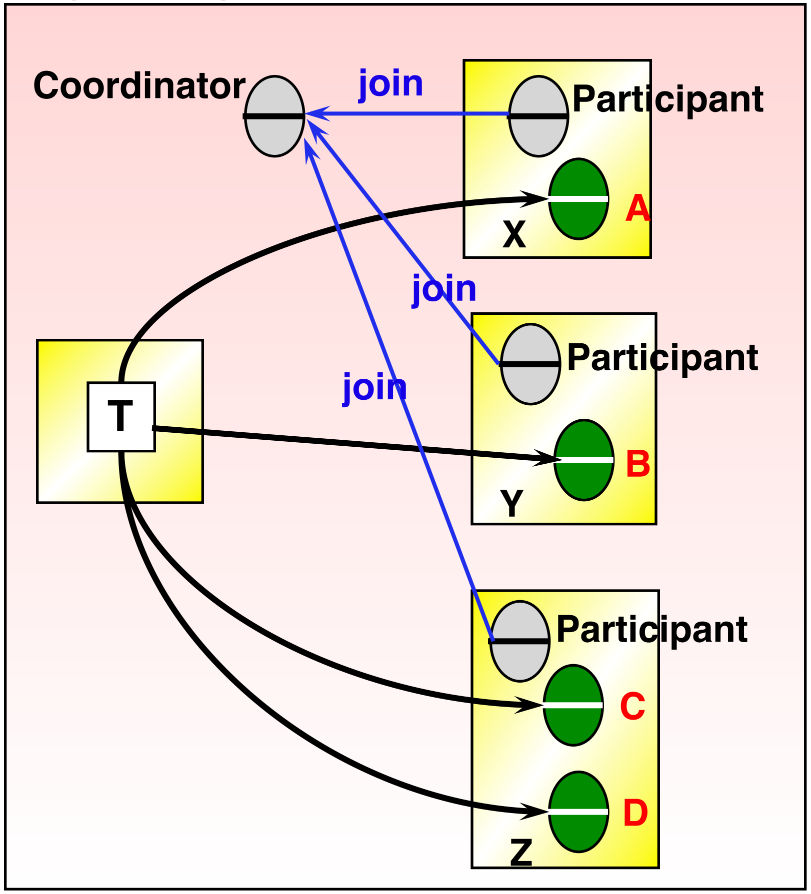


Flat Distributed Transaction

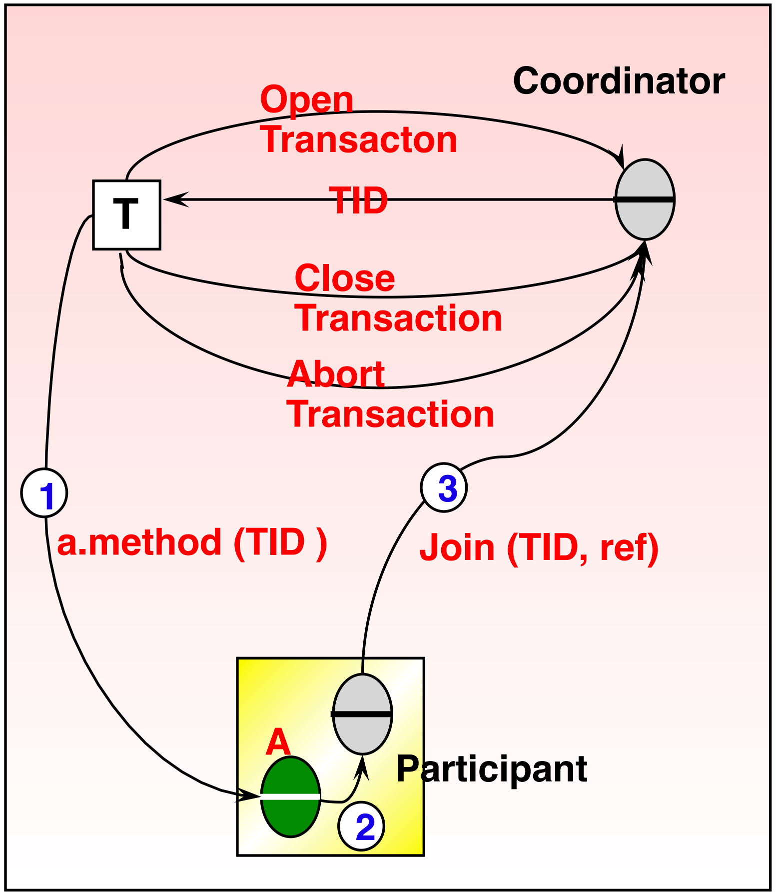
Nested Distributed Transaction

Coordination in Distributed Transactions

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

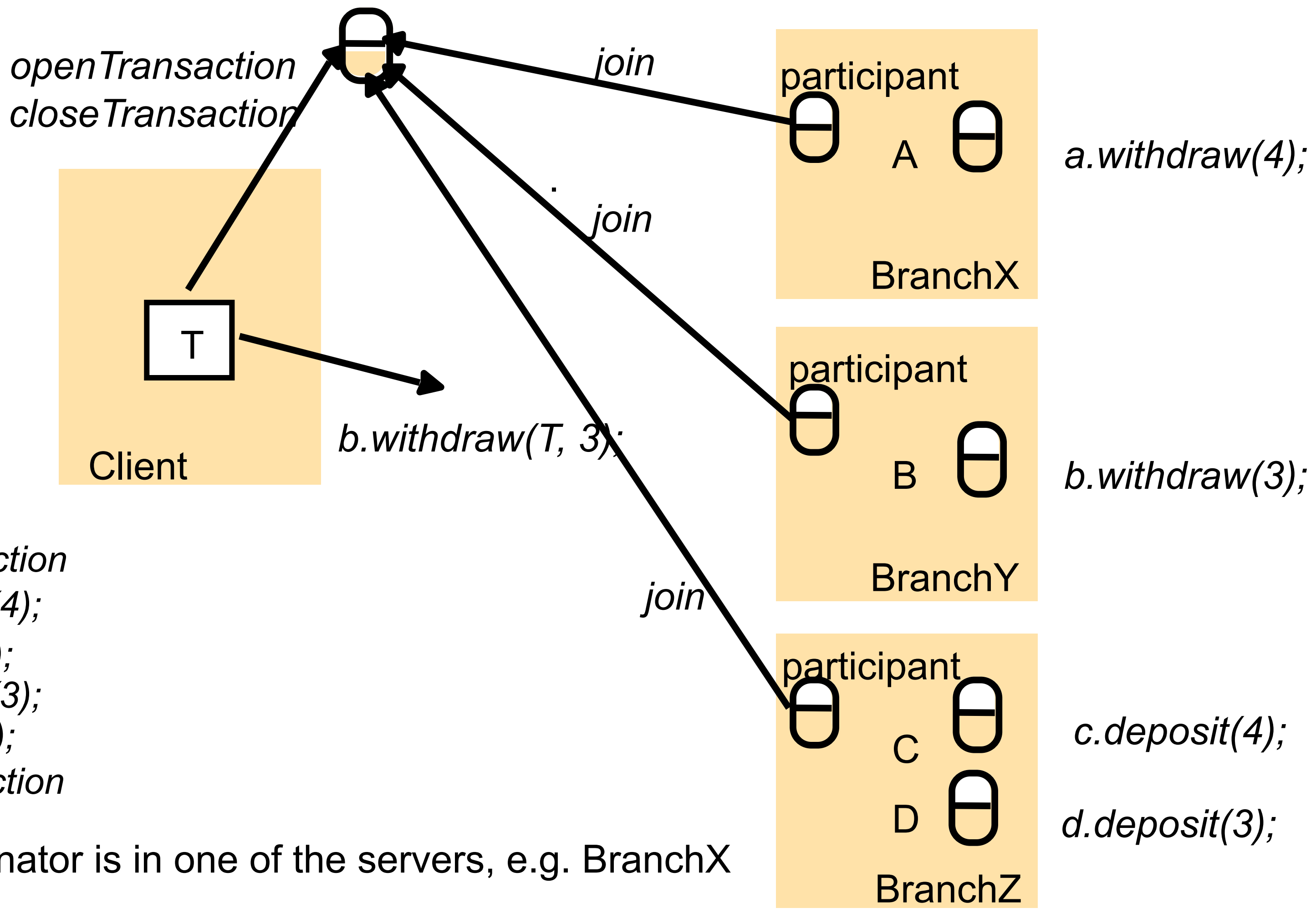


Coordinator & Participants



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

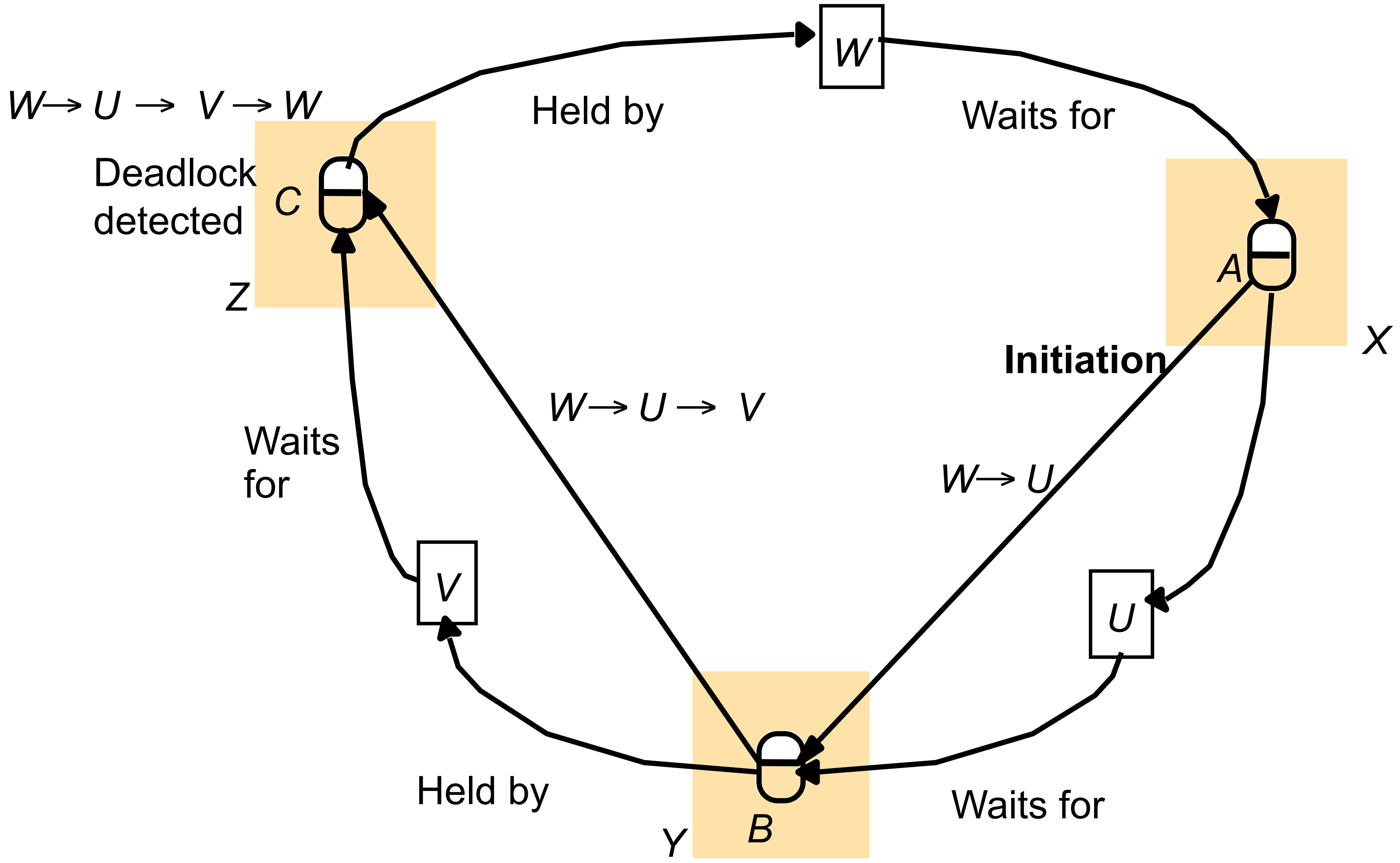
I. Locks in Distributed Transactions

- **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 (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.**

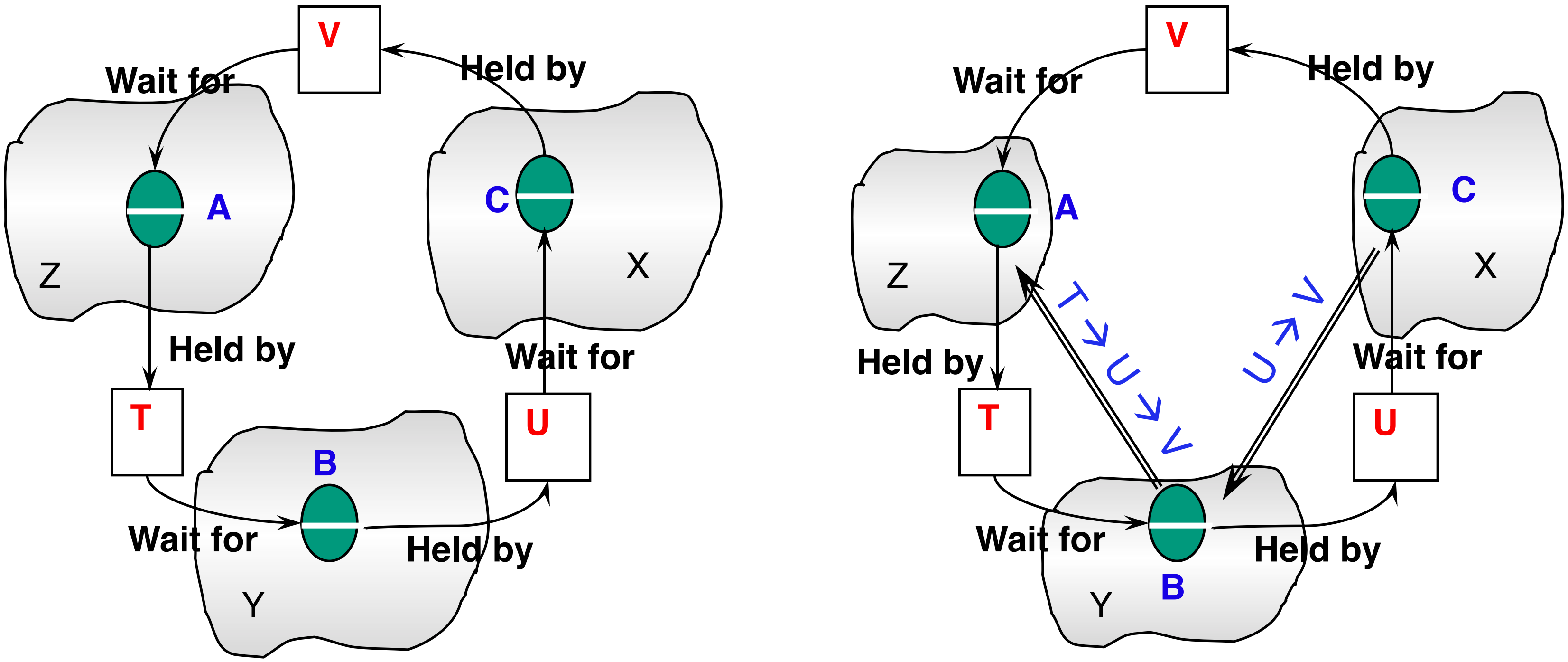
Probes Transmitted to Detect Deadlock



Edge Chasing

- Initiation: When a server S_1 notices 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, one or more transactions in the cycle is/are aborted to break the deadlock.
- Phantom deadlocks=false detection of deadlocks that don't actually exist
  - Edge chasing messages contain stale data (Edges may have disappeared in the meantime). So, all edges in a "detected" cycle may not have been present in the system all at the same time.

Reverse Edge Chasing



LOCAL Wait-for GRAPHS	X: U → V
	Y: T → U
	Z: V → T

X: U → V

Y: T → U → V

Z: T→ U →V → T deadlock detected

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, (then after a while) 1->2, 2->3. Deadlock never detected.
  - Fix: whenever an edge is created, tell everyone (broadcast) about this edge. May be inefficient.

Deadlock Prevention

- **Give objects unique integer identifiers**
- **Restrict transactions to acquire locks only in increasing order of object ids**
- **Prevents deadlock – why?**
  - **Which of the necessary conditions for deadlock does it violate?**
    - **Exclusive Locks**
    - **No preemption**
    - **Circular Wait**

II. 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 (remember these are on the server side) 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. Never have some agreeing to commit, and others agreeing to abort. Err on the side of safety.**
- **First cut: *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.**
- **Alternative: *Two-phase commit* protocol**
  - **First phase involves coordinator collecting a vote (commit or abort) from each participant (which stores partial results in permanent storage before voting).**
  - **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**

RPCs 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. (May not be required if getDecision() is used – see below)

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):

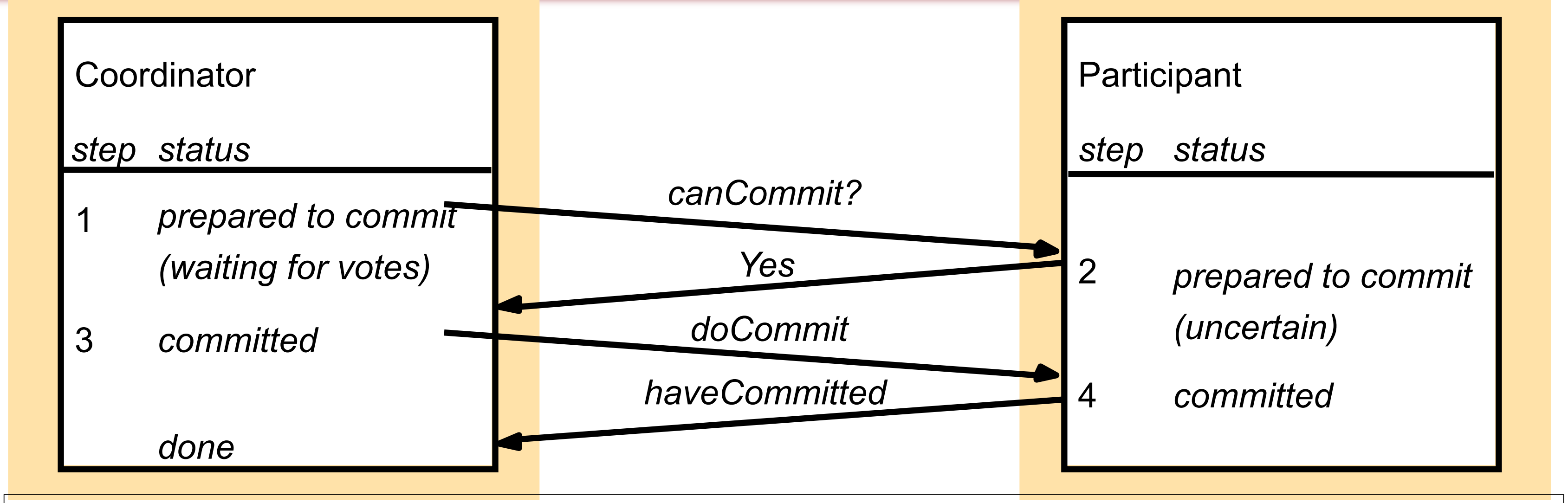
1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If its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



Coordinator

step	*status*
1	*prepared to commit* *(waiting for votes)*
3	*committed*
	done

canCommit?

Yes

doCommit

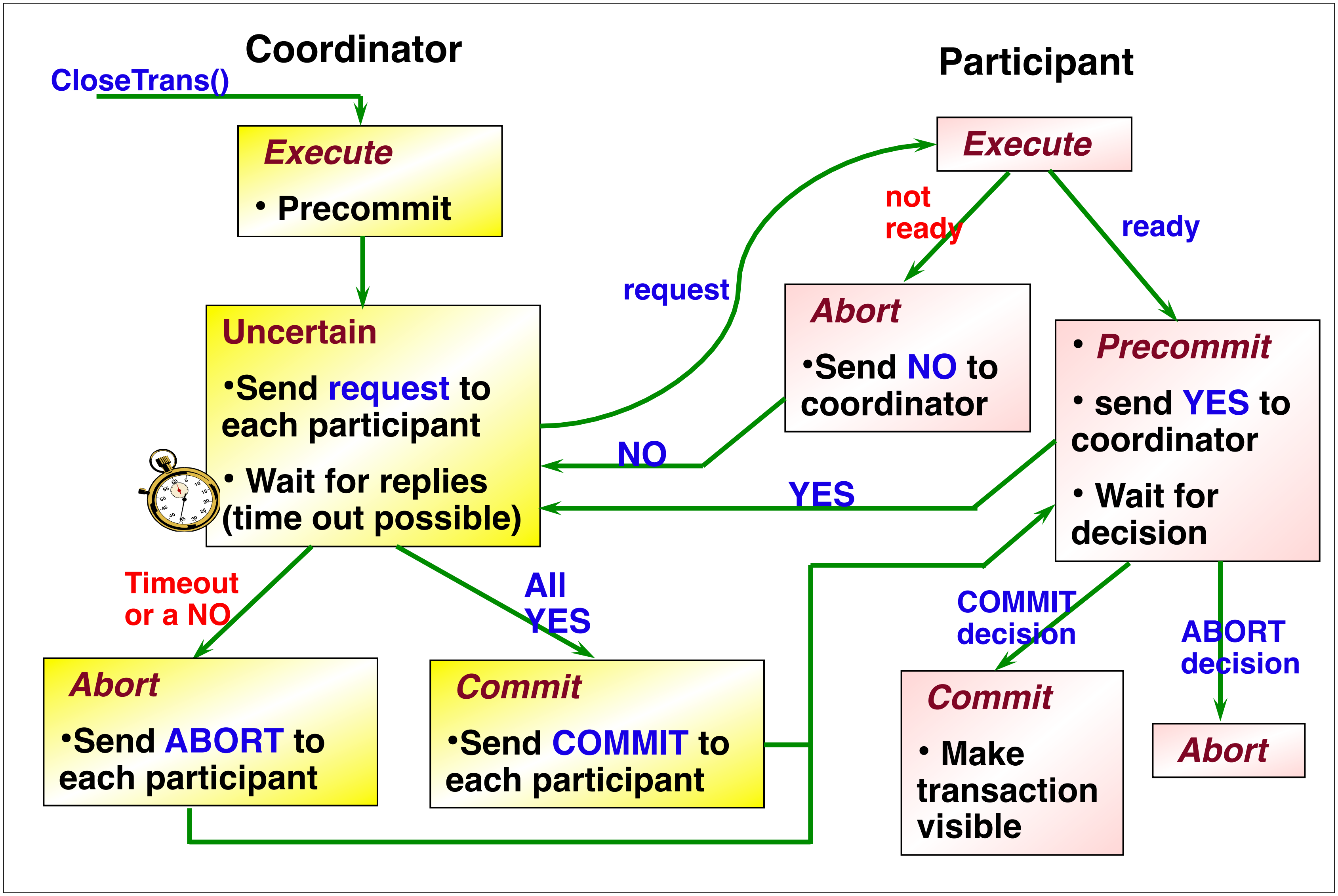
haveCommitted

Participant

step	*status*
2	*prepared to commit* *(uncertain)*
4	*committed*

- **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 abort after having voted Yes but before receiving doCommit/doAbort!**

Two Phase Commit (2PC) Protocol



Coordinator

CloseTrans()

Execute

- **Precommit**

Uncertain

•Send request to each participant

• Wait for replies (time out possible)

Timeout or a NO

All YES

Abort

•Send ABORT to each participant

Commit

•Send COMMIT to each participant

request

NO

YES

Participant

Execute

not ready

ready

Abort

•Send NO to coordinator

- ***Precommit***
- **send YES to coordinator**
- **Wait for decision**

COMMIT decision

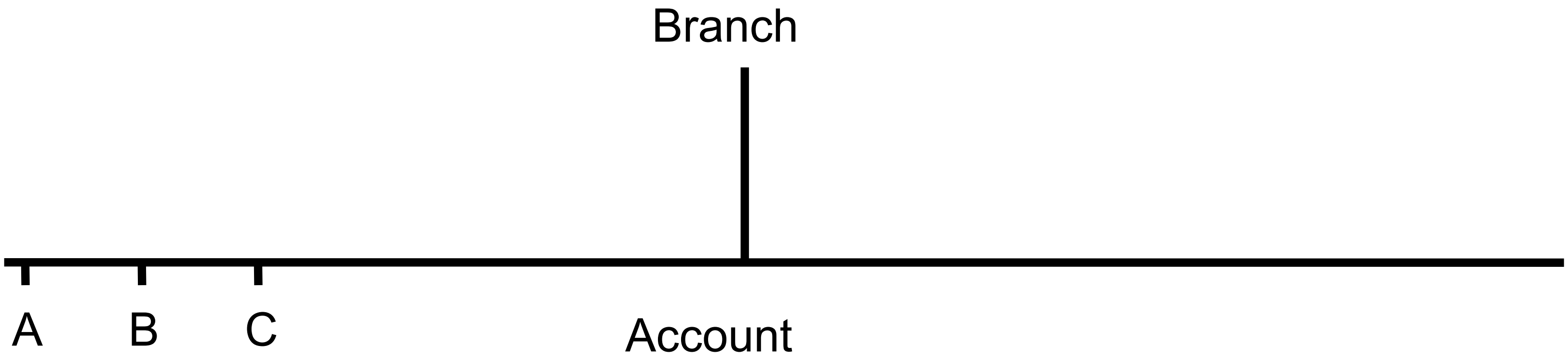
ABORT decision

Commit

- **Make transaction visible**

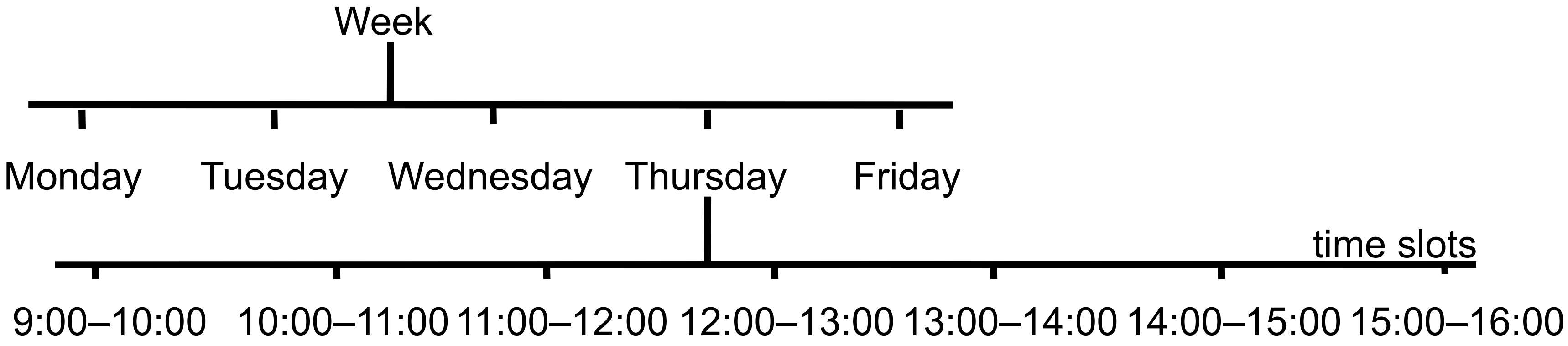
Abort

Lock Hierarchy for the Banking Example



- Deposit and withdrawal operations require locking at the granularity of an account.
- branchTotal operation acquires a read lock on all of the accounts.

Lock Hierarchy for a Diary



Week

Monday Tuesday Wednesday Thursday Friday

time slots

9:00–10:00 10:00–11:00 11:00–12:00 12:00–13:00 13:00–14:00 14:00–15:00 15:00–16:00

At each level, the setting of a parent lock has the same effect as setting all the equivalent child locks.

Hierarchical Locking

- If objects are in a "part-of" hierarchy, a lock at a higher node implicitly applies to children objects.
- Before a child node (in the object hierarchy) gets a read/write lock, an intention lock (I-read/I-write) is set for <u>all</u> ancestor nodes. The intention lock is compatible with other intention locks but conflicts with read/write locks according to the usual rules.

Lock set	Lock requested			
	read	write	I-read	I-write
none	OK	OK	OK	OK
read	OK	WAIT	OK	WAIT
write	WAIT	WAIT	WAIT	WAIT
I-read	OK	WAIT	OK	OK
I-write	WAIT	WAIT	OK	OK

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
  - Hierarchical locking