Distributed Systems

ECE428

Lecture 19

Adopted from Spring 2021

Agenda for today

- Distributed Transactions
  - Chapter 17

Transaction Processing

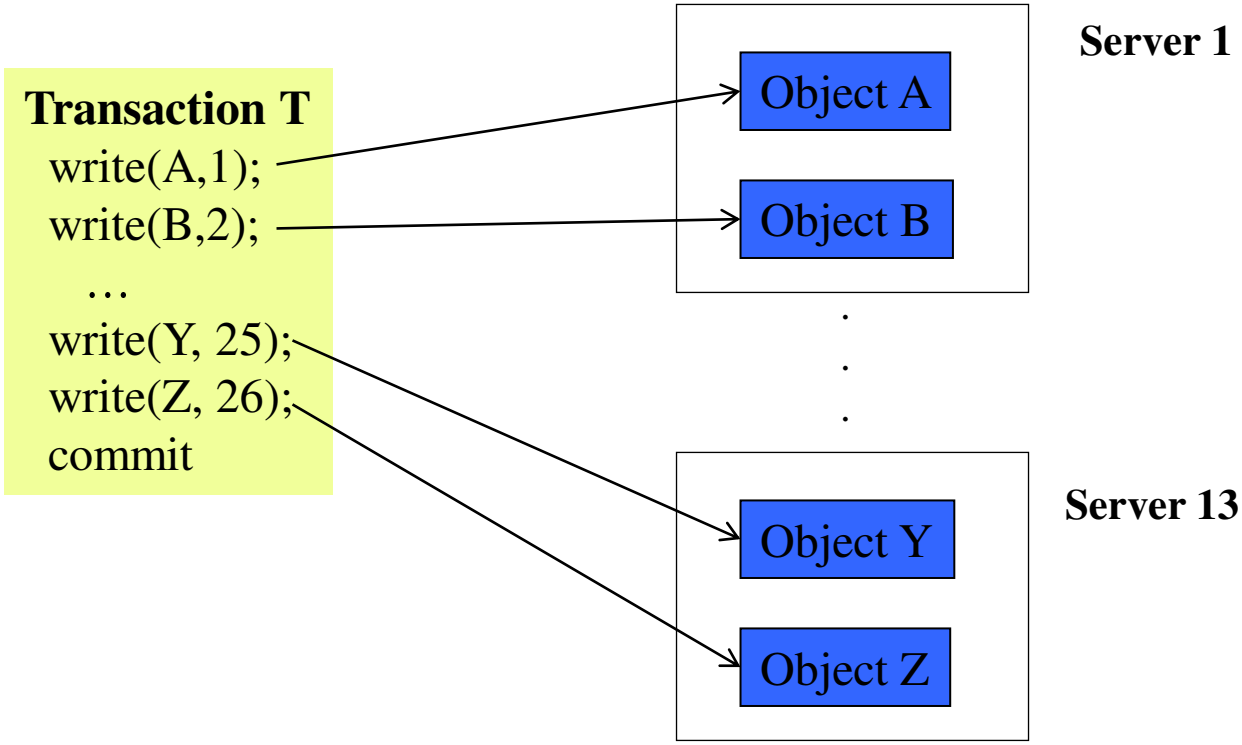
- Required properties: Atomicity, Consistency, Isolation, Durability (ACID).
- *How to prevent transactions from affecting one another?*
- Goal: increase concurrency and transaction throughput while maintaining correctness (ACID).
- Two approaches:
  - Pessimistic concurrency control: locking based.
    - read-write locks with two-phase locking and deadlock detection.
  - Optimistic concurrency control: abort if too late.
    - timestamped ordering.
- Focused on a single server (and multiple clients).

Distributed Transactions

- Transaction processing can be *distributed* across multiple servers.
  - Different objects can be stored on different servers.
    - Our focus today.
  - An object may be replicated across multiple servers.
    - Next class.

Transactions with Distributed Servers

- Different objects touched by a transaction T may reside on different servers.



Distributed Transaction Challenges

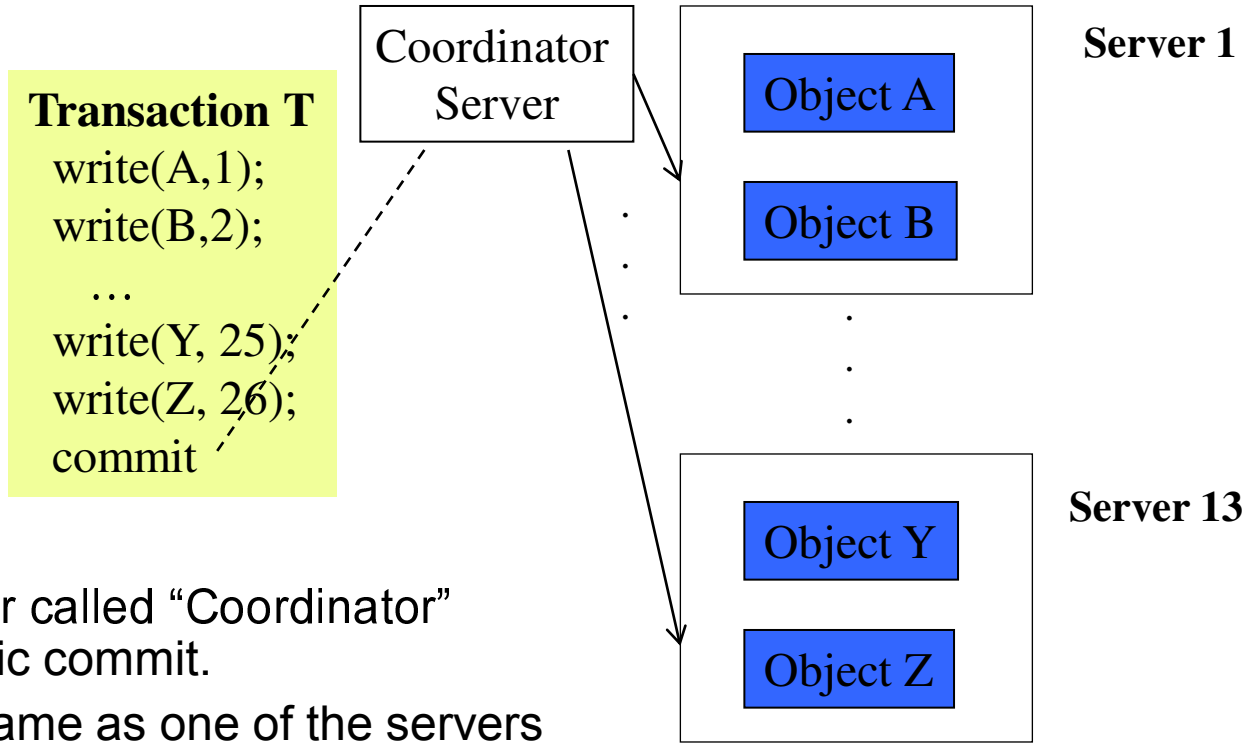
- Atomic: all-or-nothing
  - *Must ensure atomicity across servers.*
- Consistent: rules maintained
  - *Generally done locally, but may need to check non-local invariants at commit time.*
- Isolation: multiple transactions do not interfere with each other
  - *Locks at each server. How to detect and handle deadlocks?*
- Durability: values preserved even after crashes
  - *Each server keeps local recovery log.*

Distributed Transaction Atomicity

- When T tries to commit, need to ensure
  - all servers commit their updates from T, then T will commit
  - Or none of the servers commit, then T is aborted

- What problem is this?
  - Consensus!
  - (It's also called the "Atomic Commit" problem)

Coordinator Server



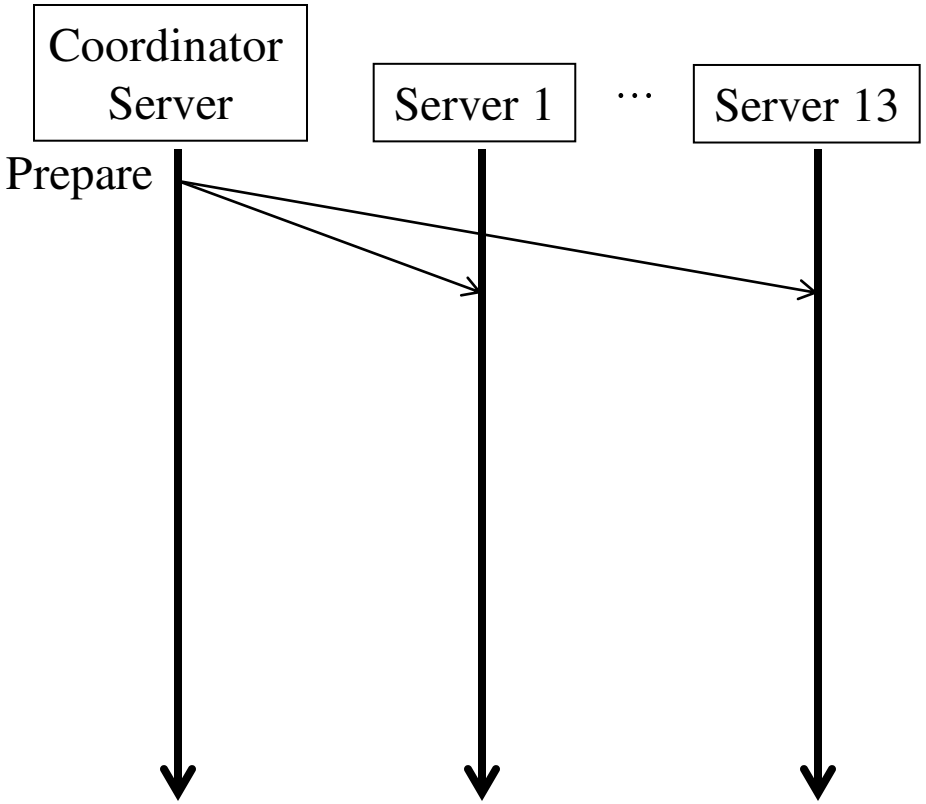
- Special server called “Coordinator” initiates atomic commit.
  - can be same as one of the servers with objects.
- Different transactions may have different coordinators.

One-phase commit

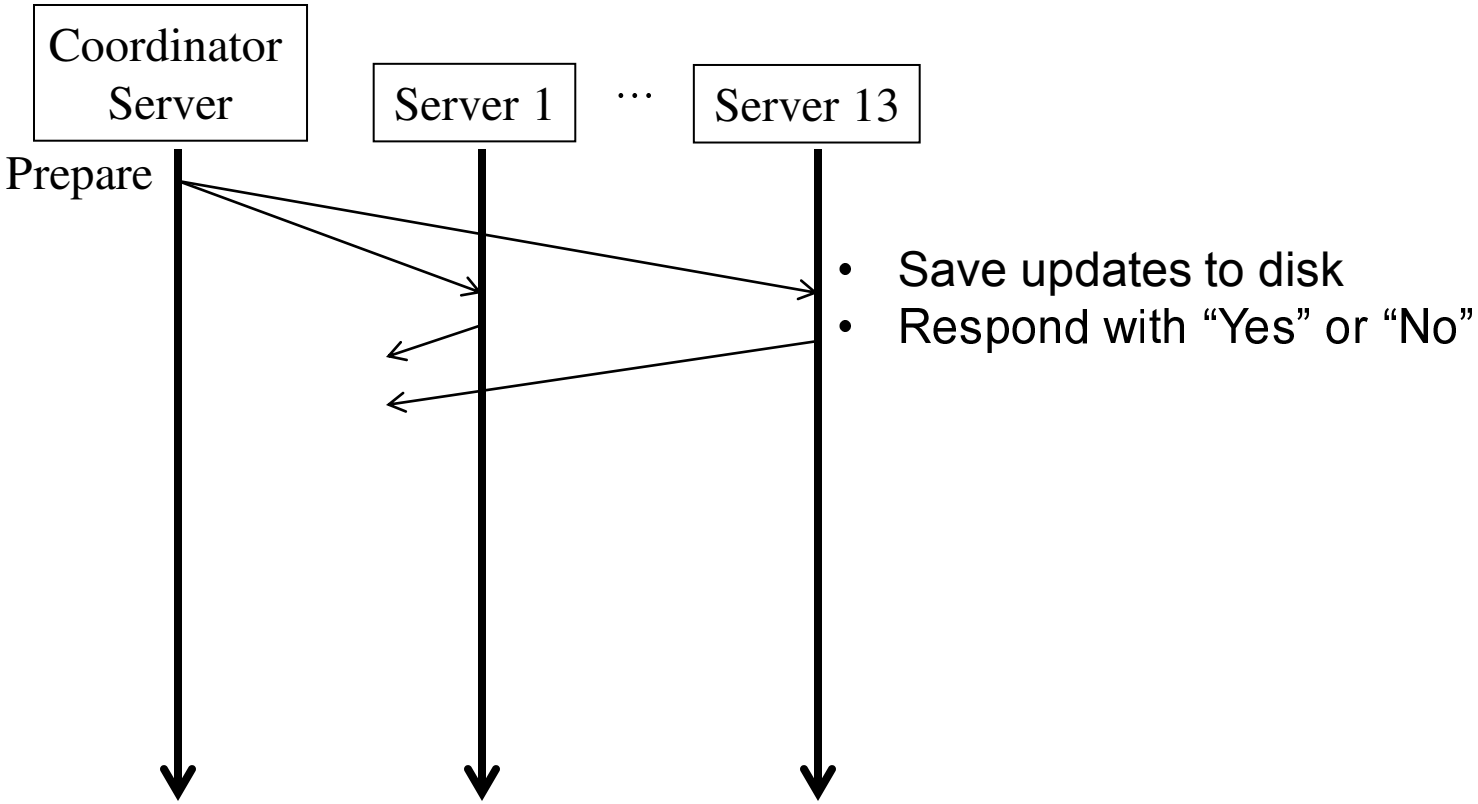
- Client relays the “commit” or “abort” command to coordinator.
  - Coordinator tells other servers to commit / abort.

- *Issues with this?*
  - The server with an object has no say whether the transaction commits or aborts.
    - If a local consistency check fails, it just cannot commit (while other servers may have committed).
  - A server may crash before receiving commit message, with some updates still in memory.

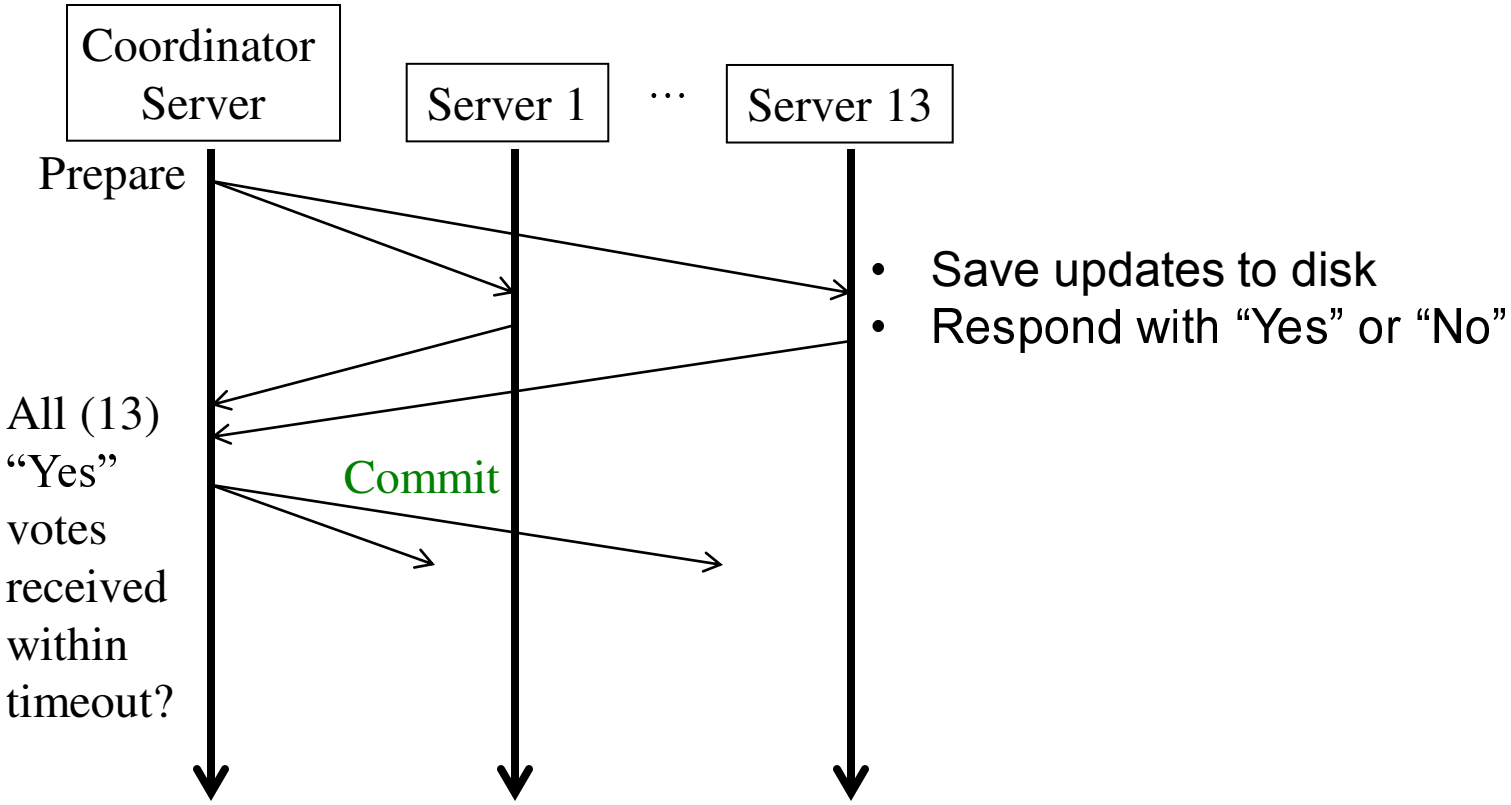
Two-phase commit



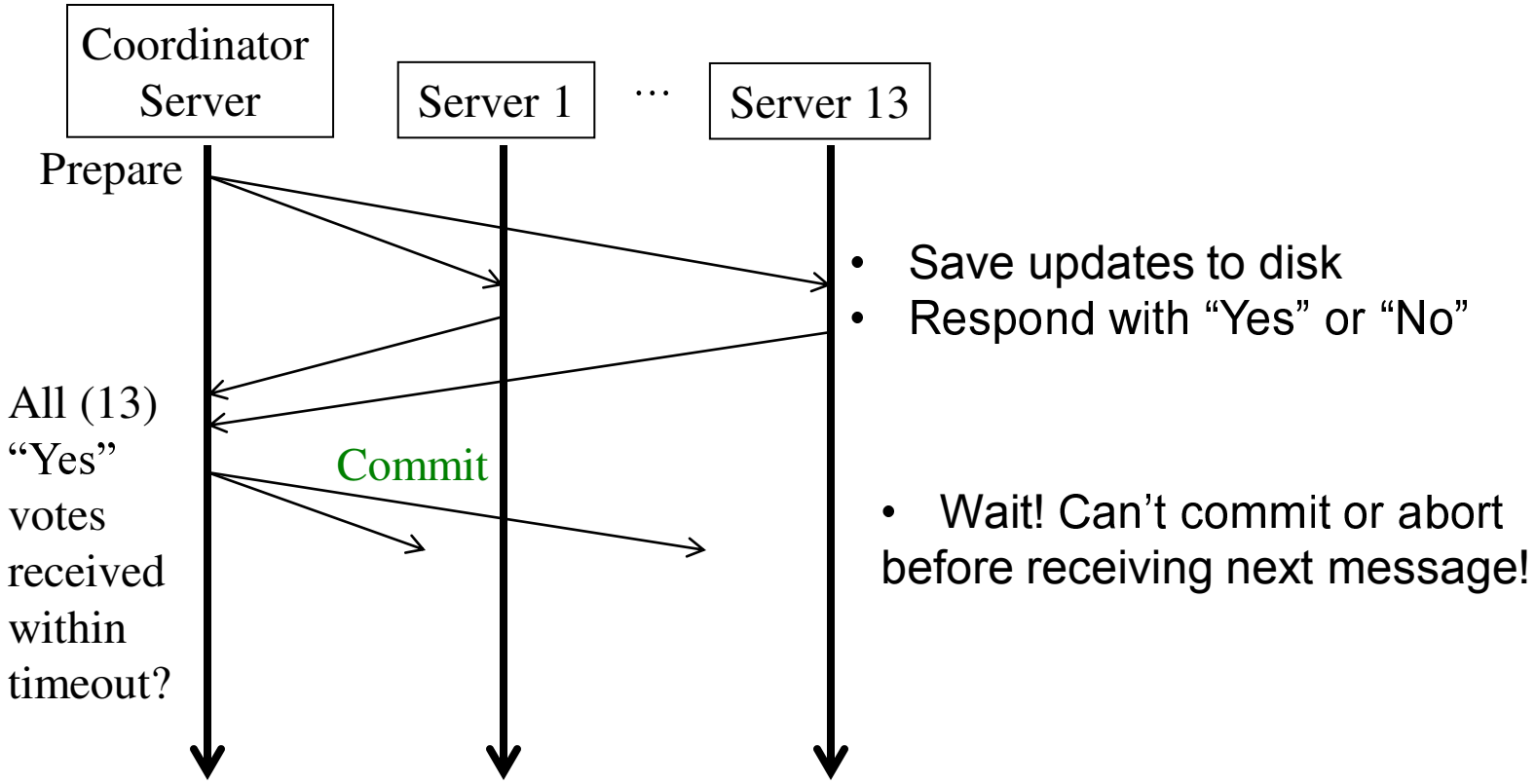
Two-phase commit



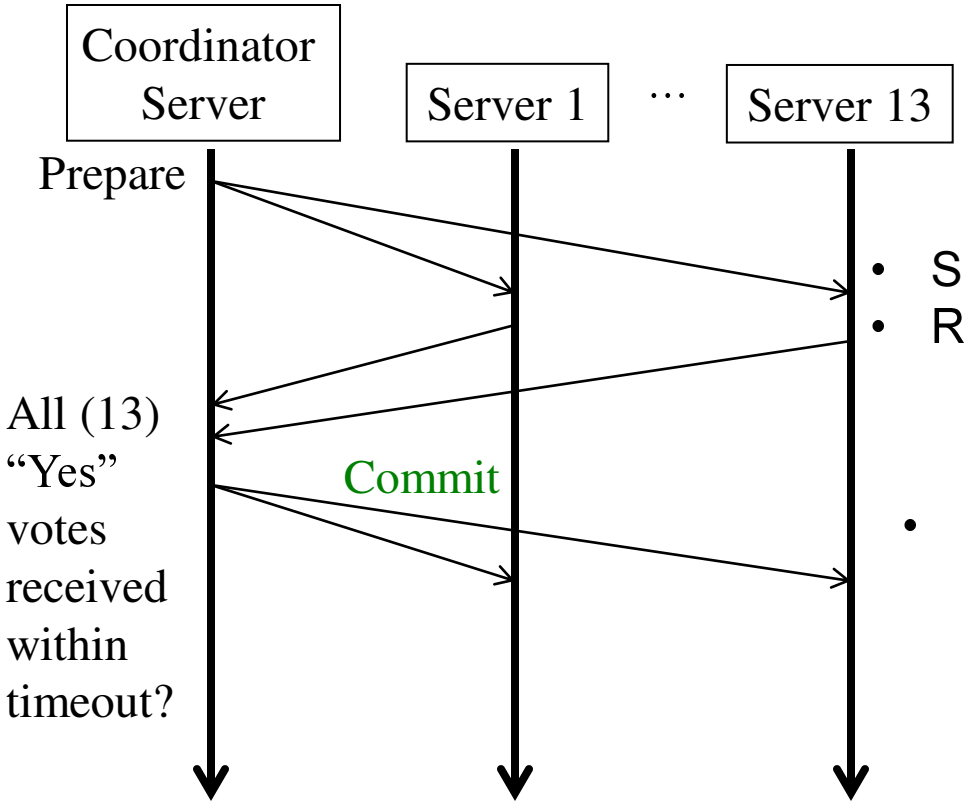
Two-phase commit



Two-phase commit



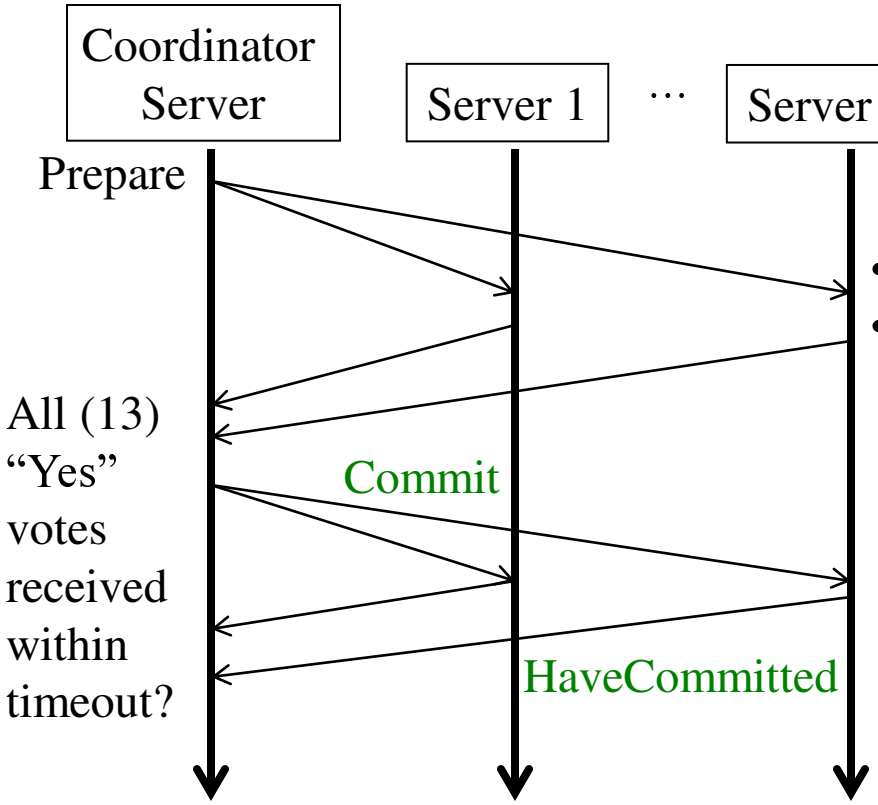
Two-phase commit



- Save updates to disk
- Respond with "Yes" or "No"

- Commit updates from the disk to store

Two-phase commit

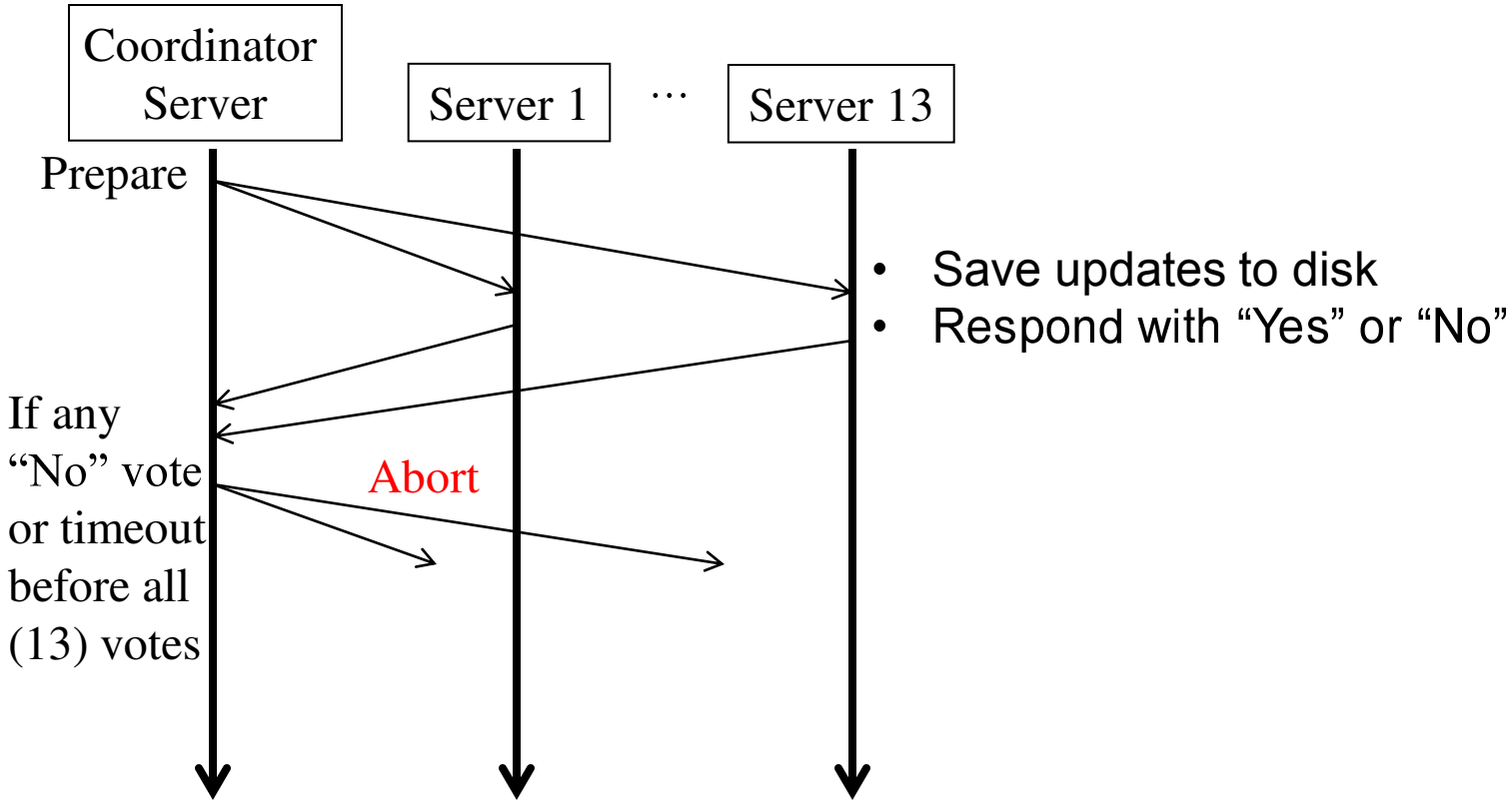


- Save updates to disk
- Respond with "Yes" or "No"

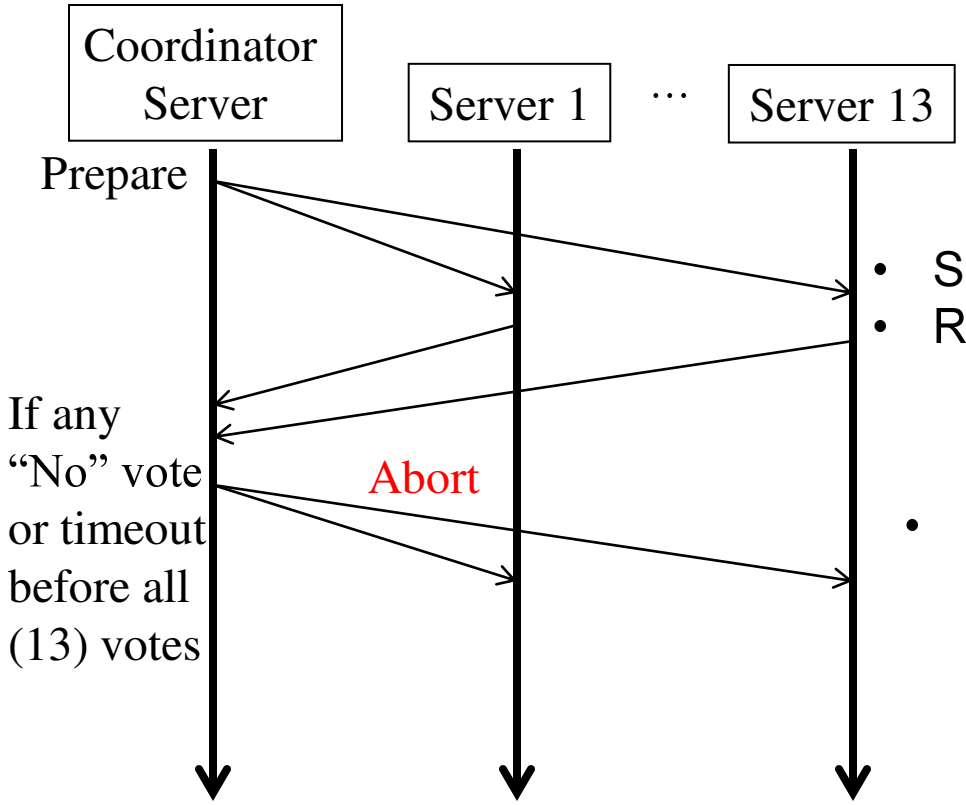
- Commit updates from the disk to store

- Coordinator now knows that all servers have committed and it can delete the associated transaction information.

Two-phase commit



Two-phase commit



- Save updates to disk
- Respond with "Yes" or "No"

- Delete tentative updates on the disk and abort.

Failures in Two-phase Commit

- If server voted Yes, it cannot commit unilaterally before receiving Commit message.
  - Does not know if other servers voted Yes.
- If server voted No, can abort right away.
  - Knows that the transaction cannot be committed.
- To deal with server crashes
  - Each server saves tentative updates into permanent storage, right before replying Yes/No in the first phase, so it is retrievable during crash recovery.
- To deal with coordinator crashes
  - Coordinator logs decisions and received/sent messages on disk.
  - After crash recovery, can retrieve the logged (saved) state.

Failures in Two-phase Commit (contd)

- To deal with Prepare message loss
  - The server may decide to abort unilaterally after a timeout during the first phase (server will vote No, and so coordinator will also eventually abort).
- To deal with Yes/No message loss
  - coordinator aborts the transaction after a timeout (pessimistic!).
  - It must announce Abort message to all.
- To deal with Commit or Abort message loss
  - Server can poll coordinator (repeatedly).

Distributed Transaction Atomicity

- When T tries to commit, needs to ensure
  - all servers commit their updates from T => T will commit
  - Or none of these servers commit => T will abort
- What problem is this?
  - Consensus!
  - (It's also called the "Atomic Commit" problem)
- But consensus is impossible in asynchronous system.
  - What makes two-phase commit work?
  - Failures of processes *masked* by replacing the crashed process with a new process whose state is retrieved from permanent storage.
  - *Two-phase commit is blocked until a failed coordinator recovers.*

ACID for Distributed Transactions

- Atomic: all-or-nothing
  - Must ensure atomicity across servers.
- Consistent: rules maintained
  - *Generally done locally, but may need to check non-local invariants at commit time.*
- Isolation: multiple transactions do not interfere with each other
  - *Locks at each server. How to detect and handle deadlocks?*
- Durability: values preserved even after crashes
  - *Each server keeps local recovery log.*

Isolation with Distributed Transaction

- Each server is responsible for applying concurrency control to objects it stores.

- Servers are collectively responsible for serial equivalence of all operations.

Timestamp Ordering with Distributed Transactions

- Each server is responsible for applying concurrency control to objects it stores.
- Servers are collectively responsible for serial equivalence of operations.
- Timestamped ordering can be applied locally at each server.
  - When a server aborts a transaction, inform the coordinator which will relay the “abort” to other servers.

Locks with Distributed Transaction

- 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 two-phase commit protocol.
- How to handle deadlocks?

Deadlock Detection in Distributed Transactions

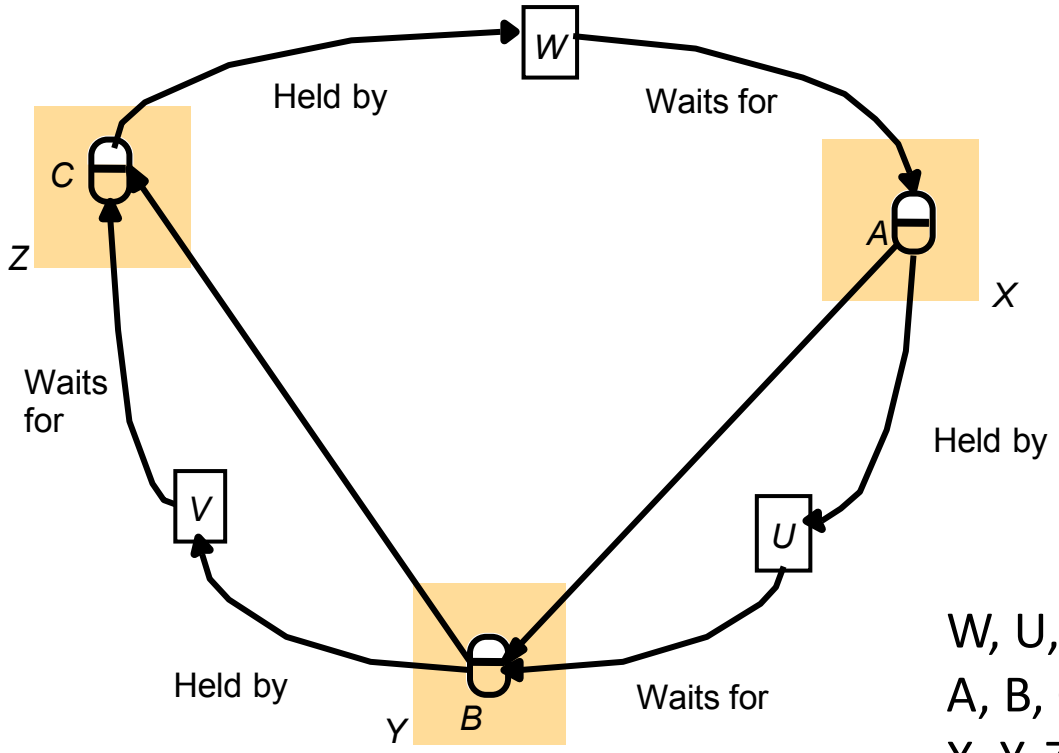
- The wait-for graph in a distributed set of transactions is distributed.

- Centralized detection
  - Each server reports waits-for relationships to central server.
  - Coordinator constructs global graph, checks for cycles.

- Issues:
  - Single point of failure (gets blocked with central server fails).
  - Scalability.

Decentralized Deadlock Detection

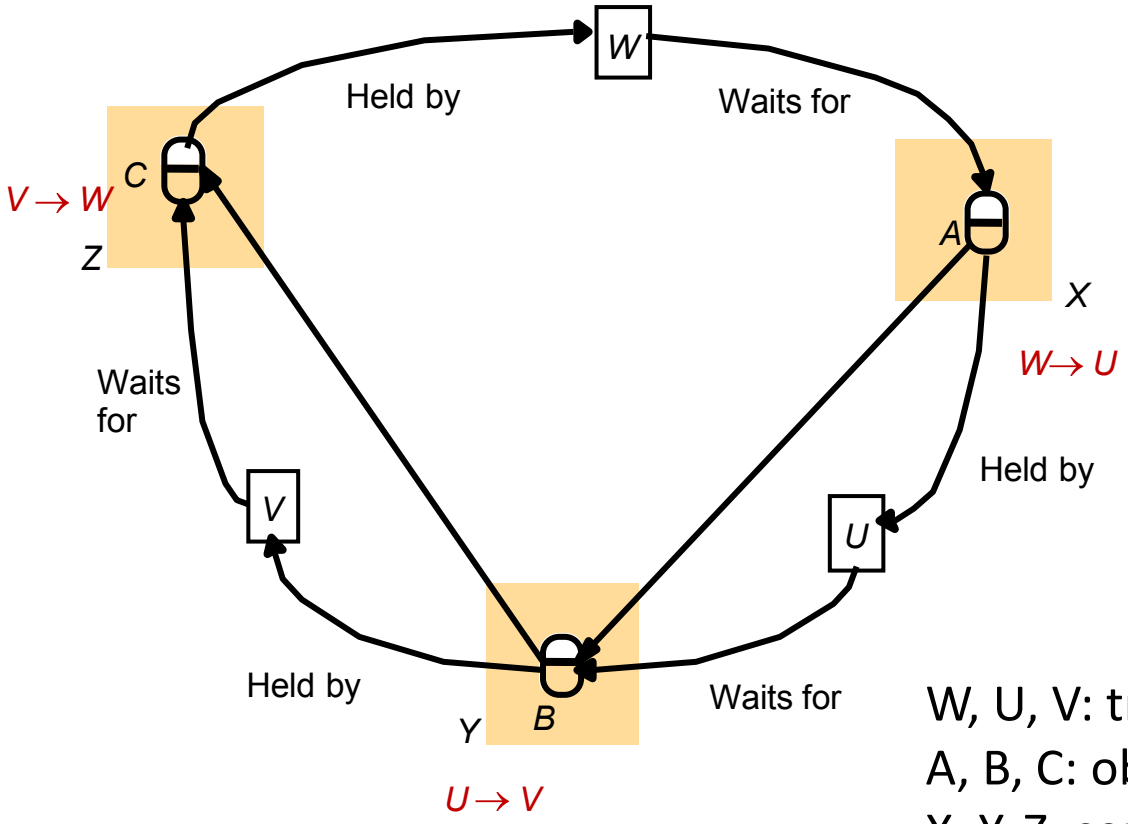
- Edge chasing: Forward “probe” messages to servers in the edges of wait-for graph, pushing the graph forward, until a cycle is found.



W, U, V: transactions
A, B, C: objects
X, Y, Z: servers

Decentralized Deadlock Detection

- Edge chasing: Forward "probe" messages to servers in the edges of wait-for graph, pushing the graph forward, until a cycle is found.



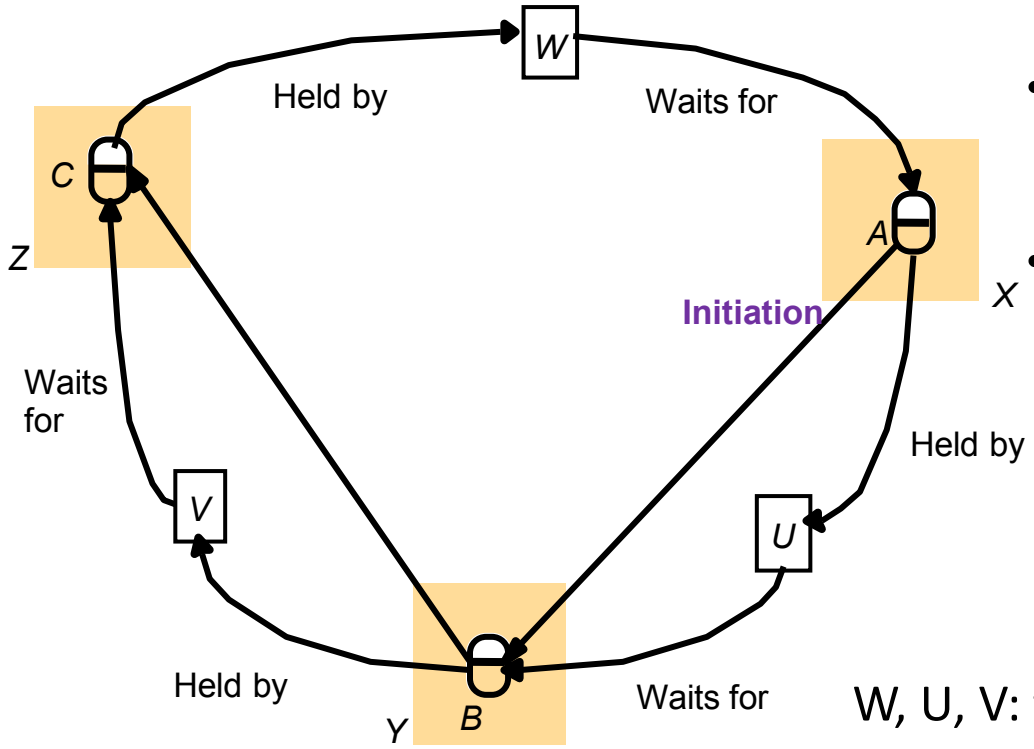
All servers know local wait-for relationships.

Coordinator for each transaction knows whether the transaction is waiting on an object lock, and at which server.

W, U, V: transactions
A, B, C: objects
X, Y, Z: servers

Decentralized Deadlock Detection

- Edge chasing: Forward "probe" messages to servers in the edges of wait-for graph, pushing the graph forward, until cycle is found.

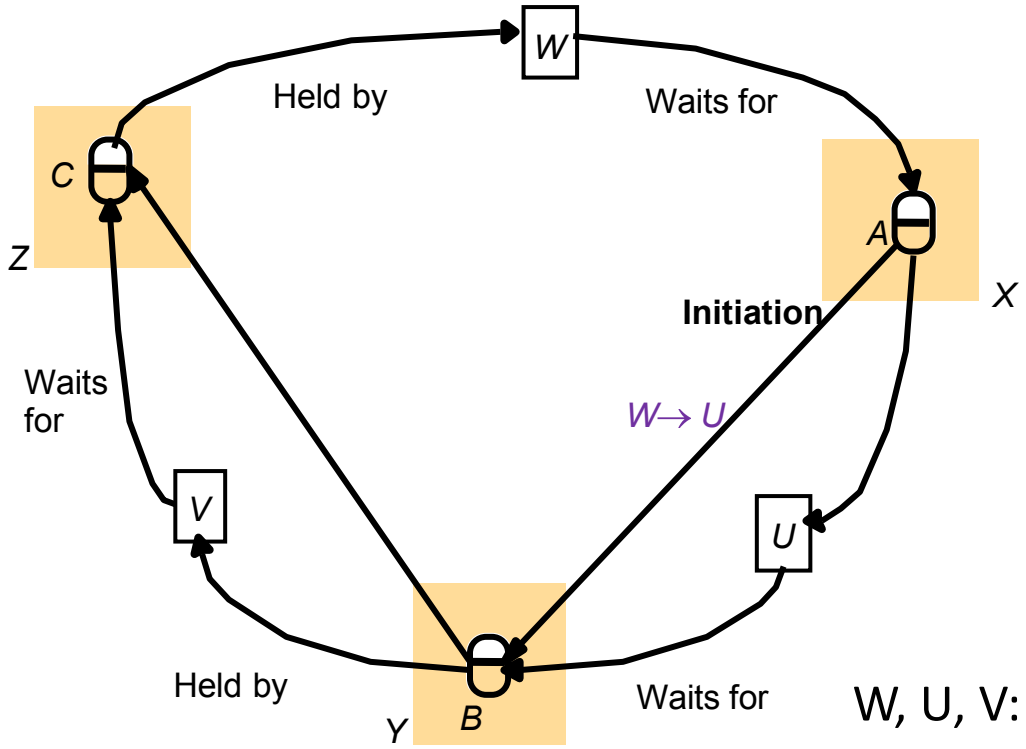


- Server X realizes W is waiting on U (a potential edge in the wait-for graph).
- Asks U's coordinator whether U is waiting on anything, and at which server.

W, U, V: transactions
A, B, C: objects
X, Y, Z: servers

Decentralized Deadlock Detection

- Edge chasing: Forward "probe" messages to servers in the edges of wait-for graph, pushing the graph forward, until cycle is found.

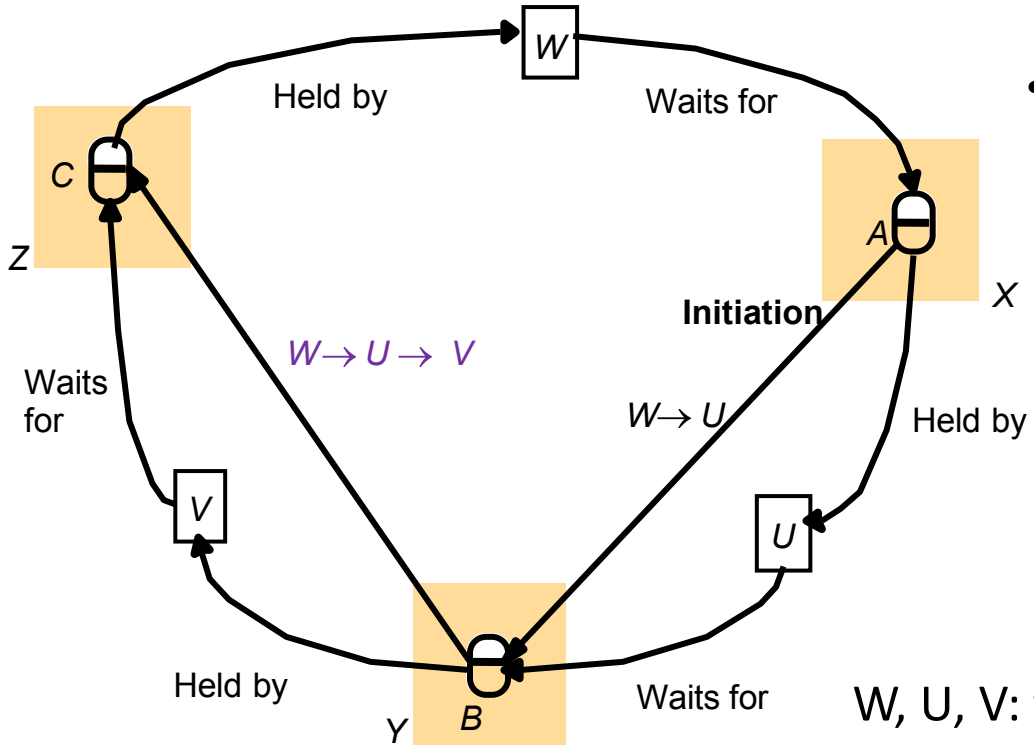


- Server X realizes W is waiting on U (a potential edge in the wait-for graph).
- Asks U's coordinator whether U is waiting on anything, and at which server.
- *Sends a probe to the next server.*

W, U, V: transactions
A, B, C: objects
X, Y, Z: servers

Decentralized Deadlock Detection

- Edge chasing: Forward "probe" messages to servers in the edges of wait-for graph, pushing the graph forward, until cycle is found.

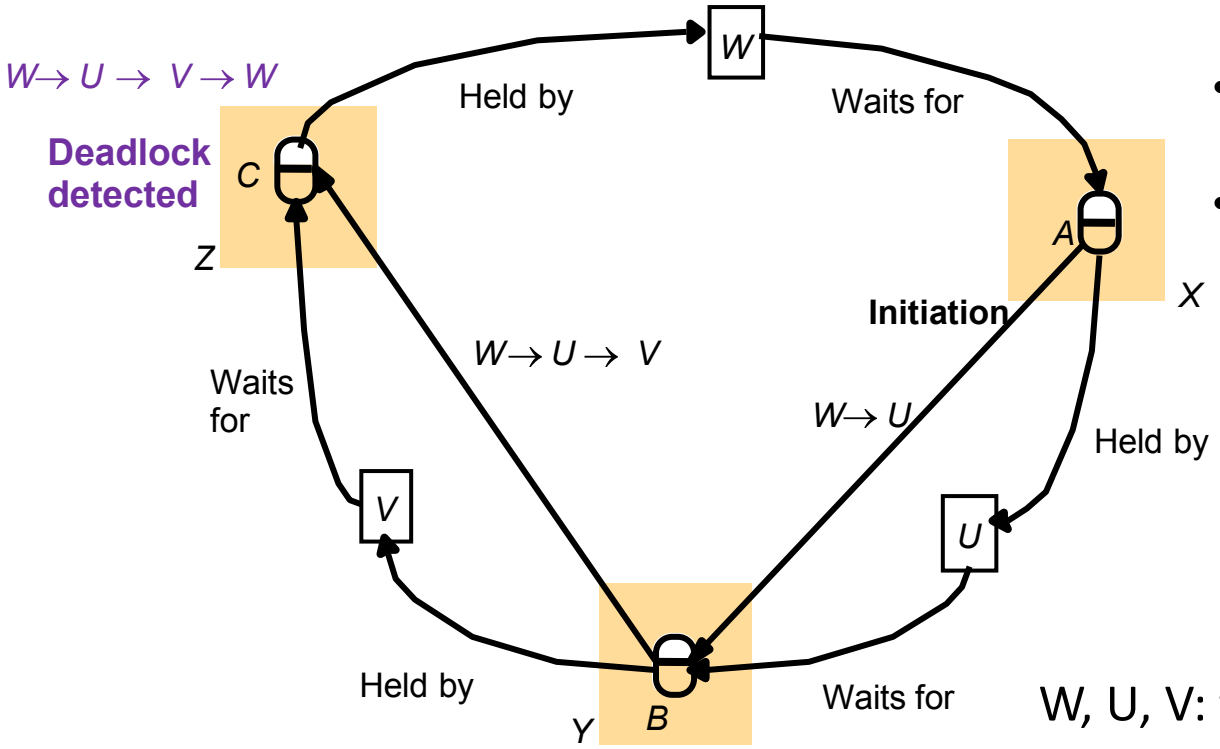


- Y adds another edge, and forwards the probe to the next server.

W, U, V: transactions
A, B, C: objects
X, Y, Z: servers

Decentralized Deadlock Detection

- Edge chasing: Forward "probe" messages to servers in the edges of wait-for graph, pushing the graph forward, until cycle is found.



- Z can now detect a deadlock.
- The transaction in the cycle can now be aborted (by informing its coordinator), and deadlock breaks.

W, U, V: transactions
A, B, C: objects
X, Y, Z: servers

Edge Chasing: Phases

- 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

- Phantom deadlocks = false detection of deadlocks that don't actually exist due to messages containing stale data
  - Edges may have disappeared in meantime
  - Edges in a "detected" cycle may not have been present at the same time
- This leads to spurious aborts of transaction.

Transaction Priority

- Which transaction to abort?
- Transactions may be given a priority.
  - e.g. inverse of their timestamp
- When deadlock cycle is detected, abort the lowest priority transaction.
  - Only one transaction aborted even if several simultaneous probes find a cycle.

Summary

- Distributed Transaction: Different objects that a transaction touches are stored on different servers.
  - One server process marked out as coordinator
  - Atomic Commit: two-phase commit
  - Deadlock detection: Centralized, Edge chasing

- Next class: when objects are *replicated* across multiple servers.