Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for the materials: Indy Gupta and Nikita Borisov

Logistics

- MP3 has been (will soon be?) released..
- No class on Friday: EOH and midterm 2
  - I'll hold online OH on Friday, will announce details over Campuswire.

Agenda for today

- Transaction Processing and Concurrency Control
  - Chapter 16
    - Transaction semantics: ACID
    - Isolation and serial equivalence
    - Conflicting operations
    - Two-phase locking
    - Deadlocks
    - Timestamped ordering (wrap up)

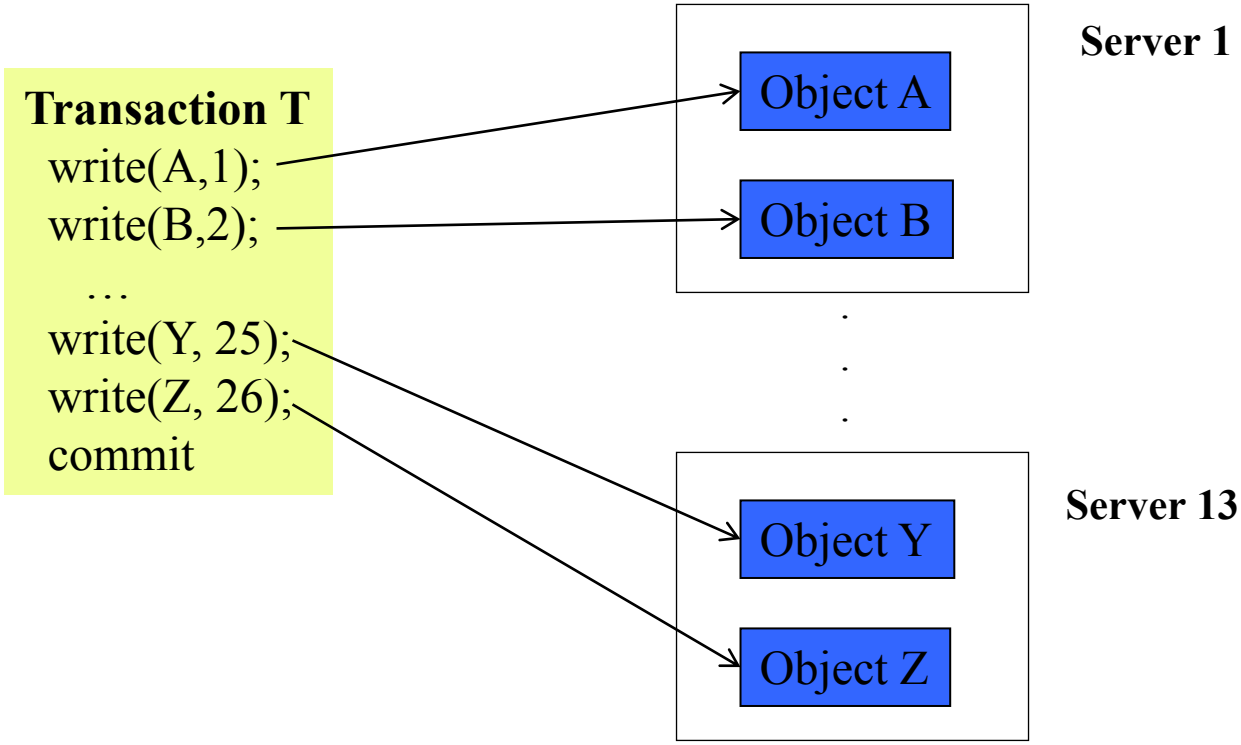
- Distributed Transactions

Distributed Transactions

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

Transactions with Distributed Servers

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



Distributed Transaction Challenges

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

Distributed Transaction Challenges

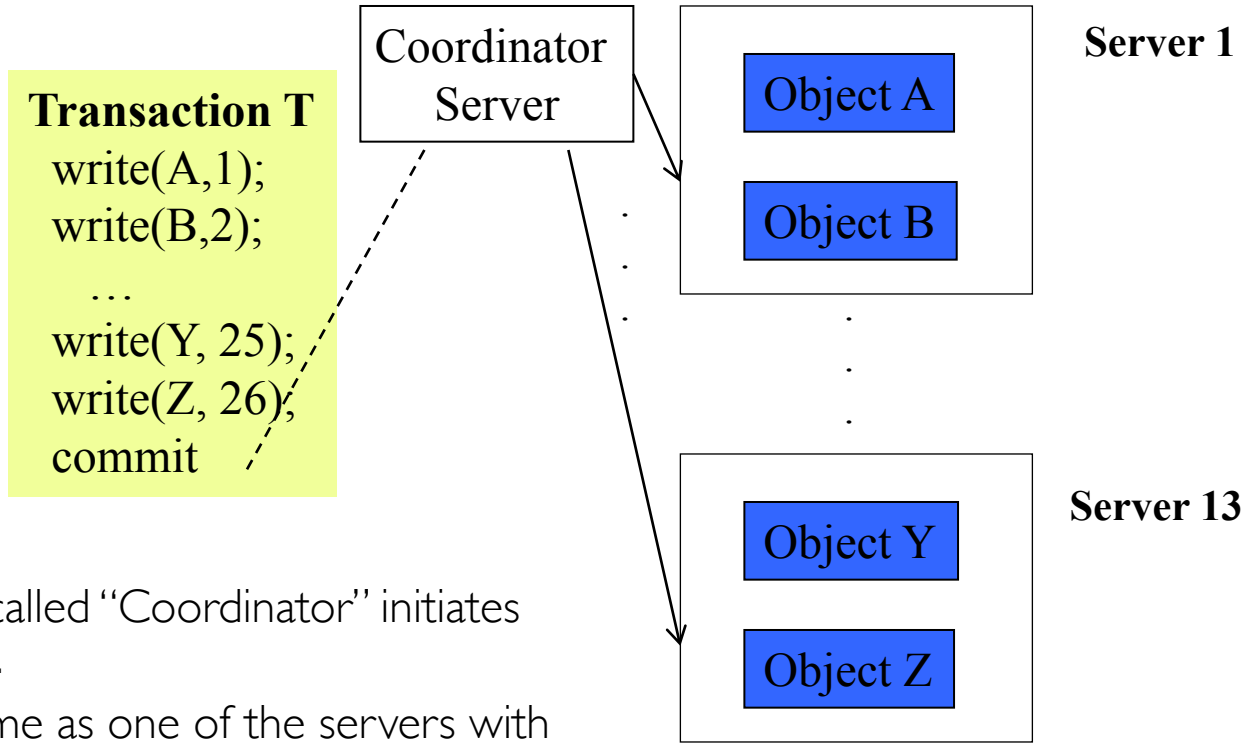
- **Atomic: all-or-nothing**
  - ***Must ensure atomicity across servers.***
- **C**onsistent: rules maintained
  - *Generally done locally, but may need to check non-local invariants at commit time.*
- **Isolation: multiple transactions do not interfere with each other**
  - ***Locks at each server.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

Distributed Transaction Atomicity

- When transaction T tries to commit, need to ensure
  - all these 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)

Coordinator Server



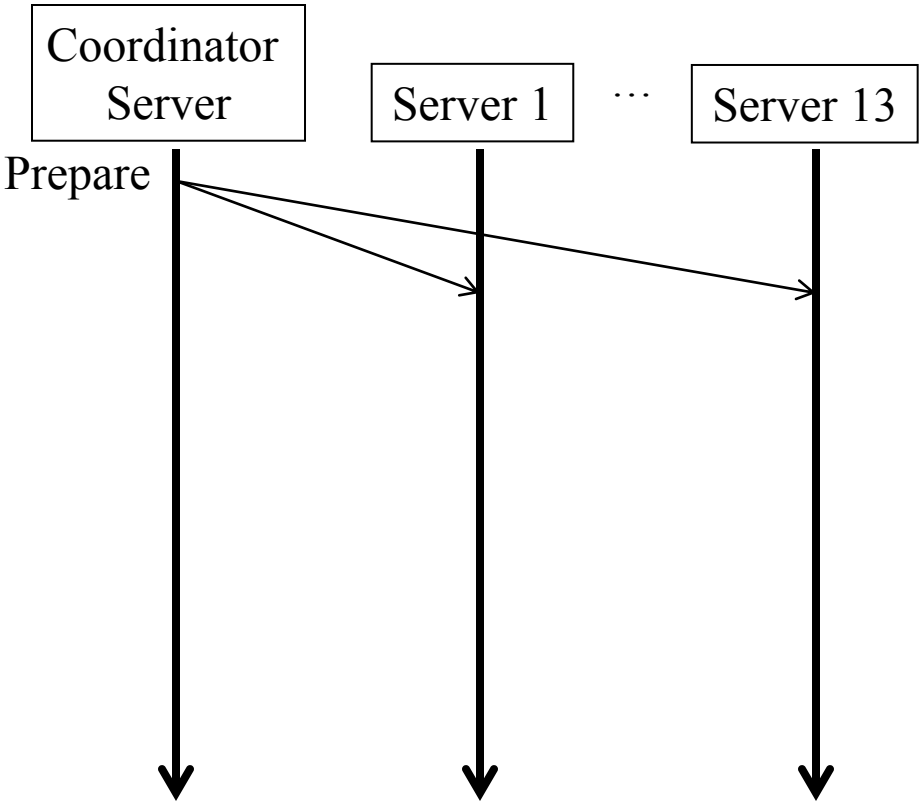
- Special server called "Coordinator" initiates atomic commit.
  - can be same as one of the servers with objects.
- Different transactions may have different coordinators.

One-phase commit

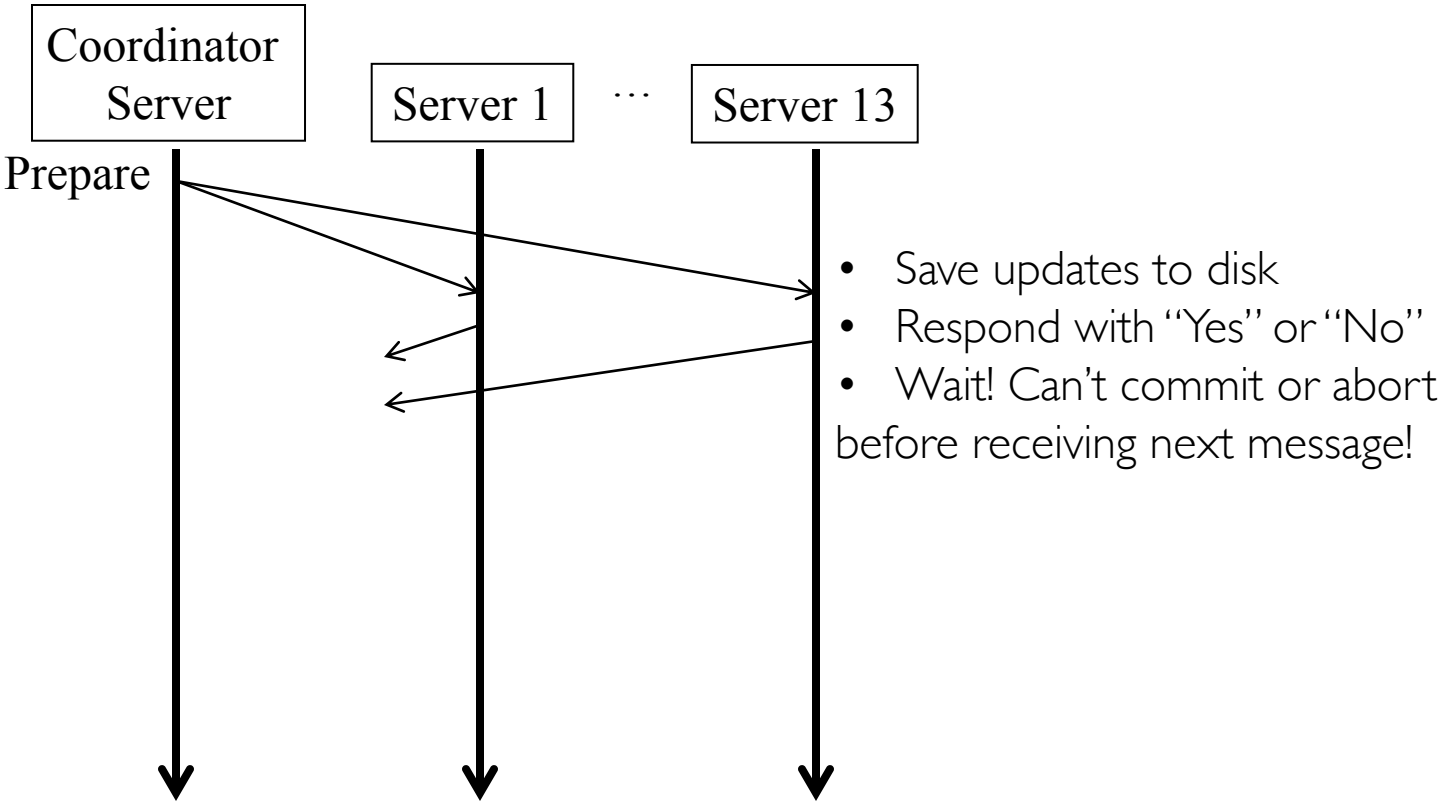
- Client relays the "commit" or "abort" command to the coordinator.
  - Coordinator tells other servers to commit / abort.

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

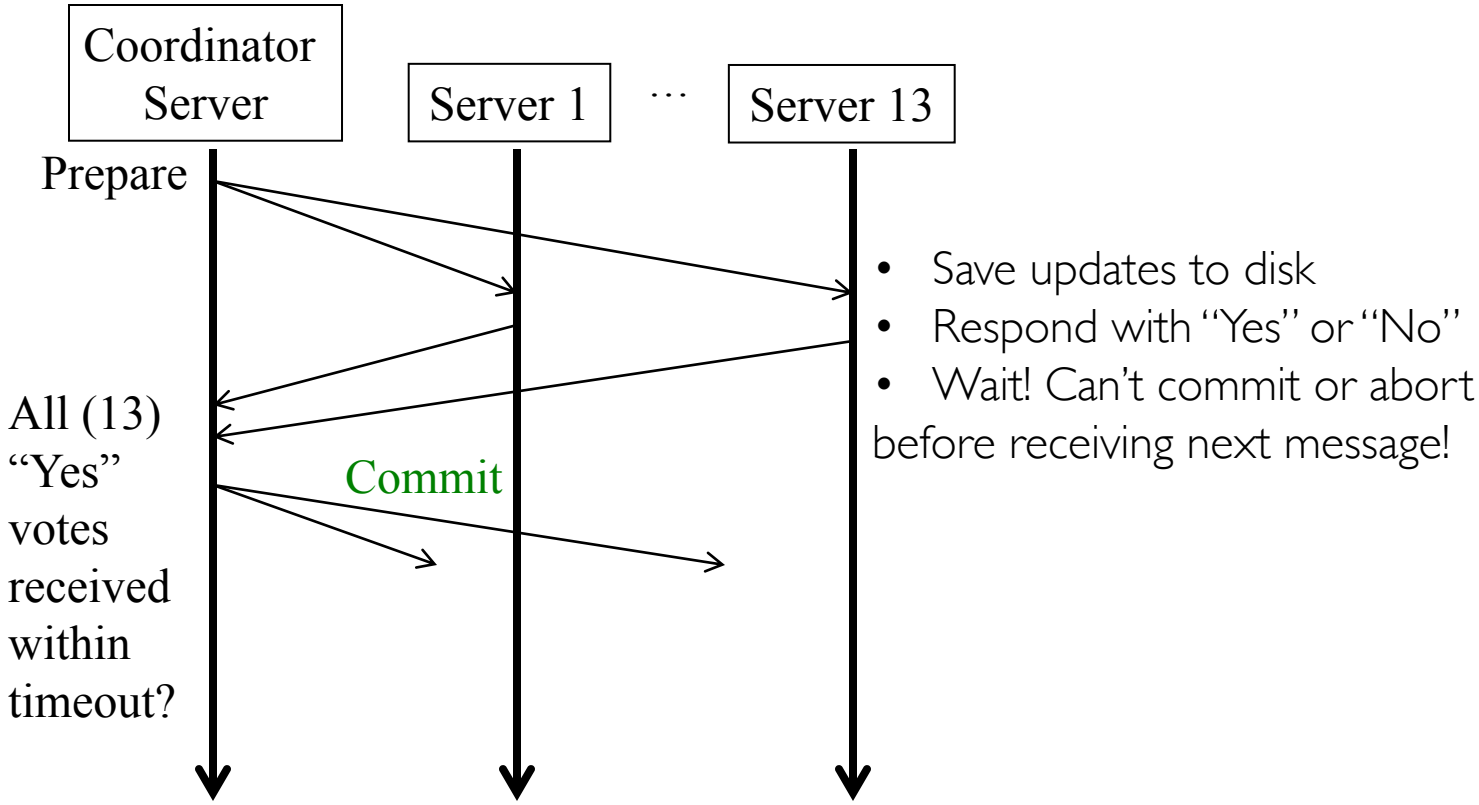
Two-phase commit



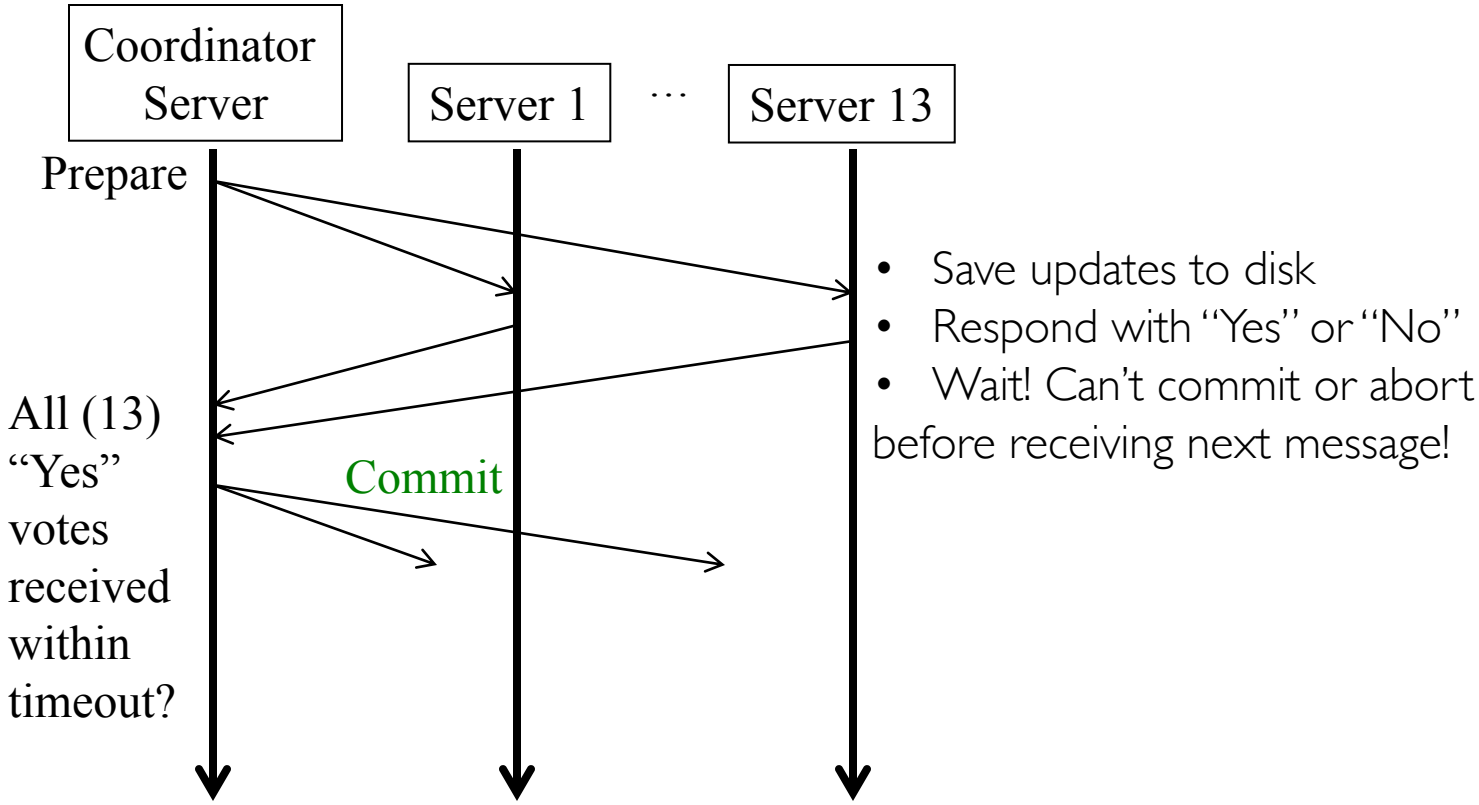
Two-phase commit



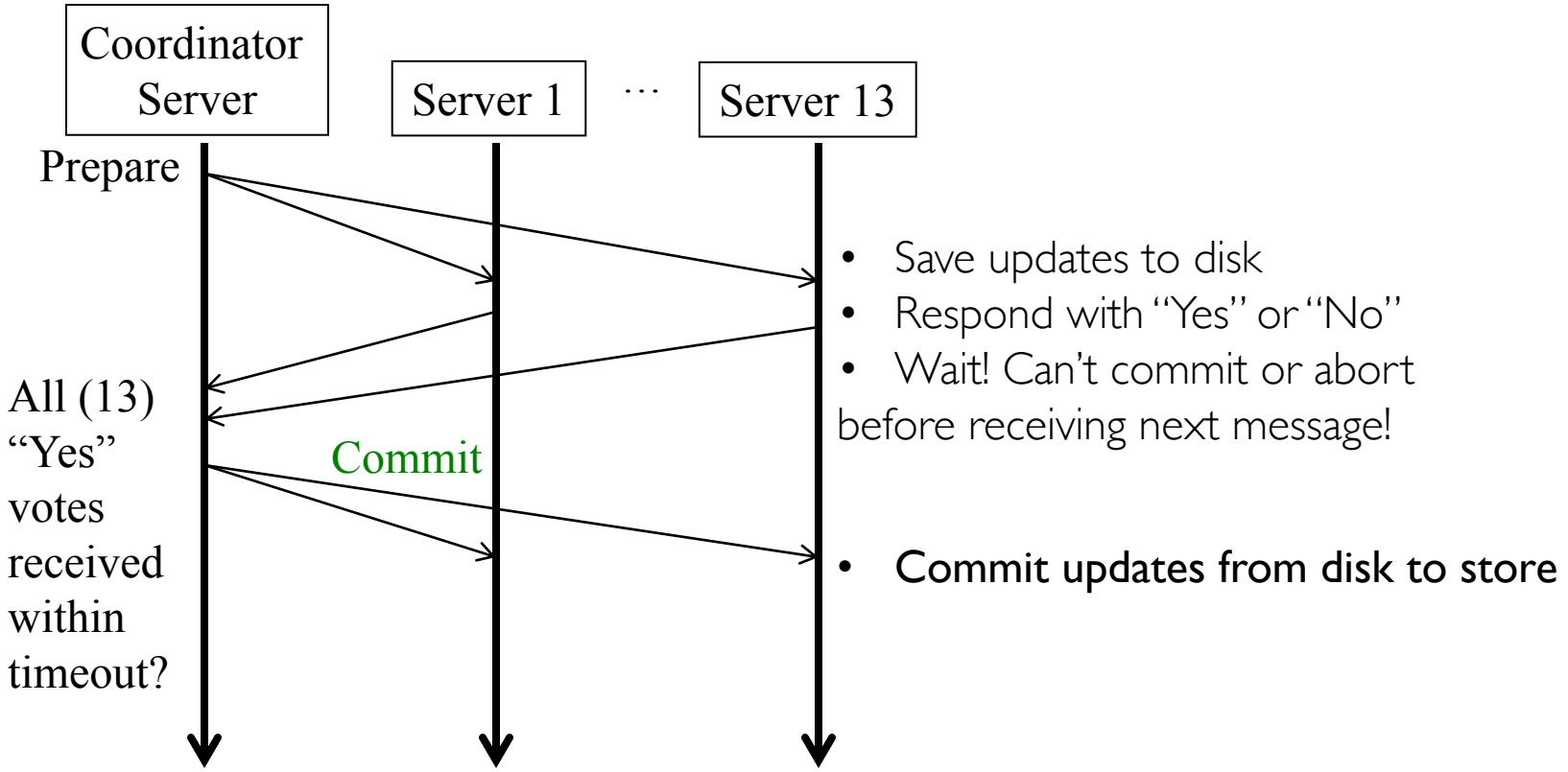
Two-phase commit



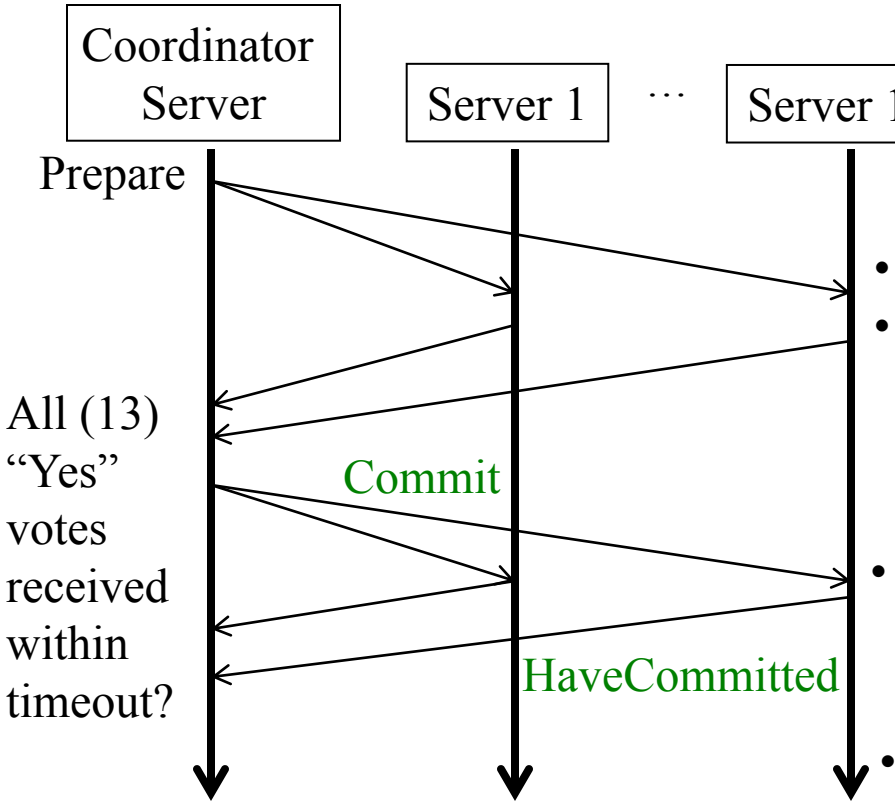
Two-phase commit



Two-phase commit



Two-phase commit

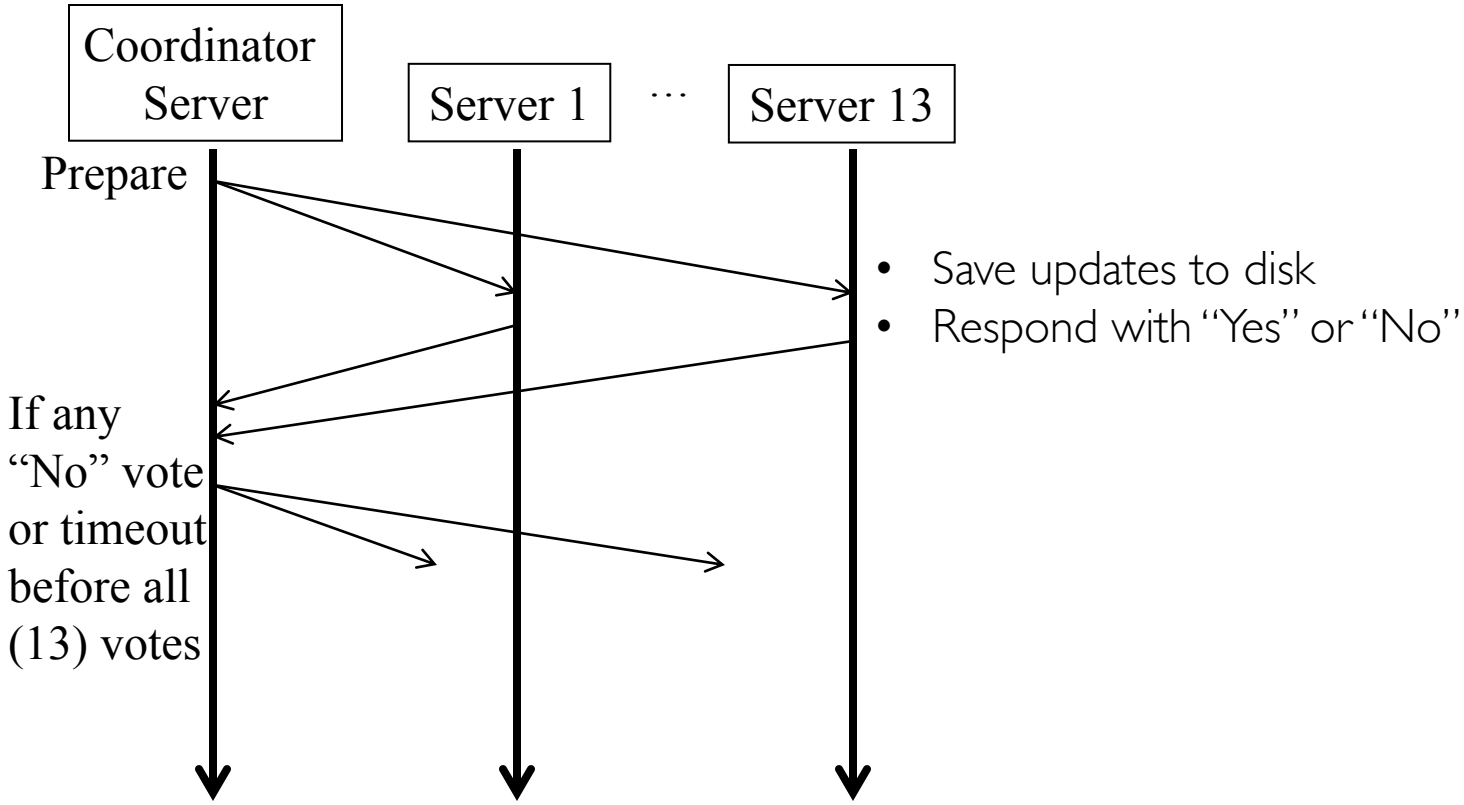


- Save updates to disk
- Respond with "Yes" or "No"

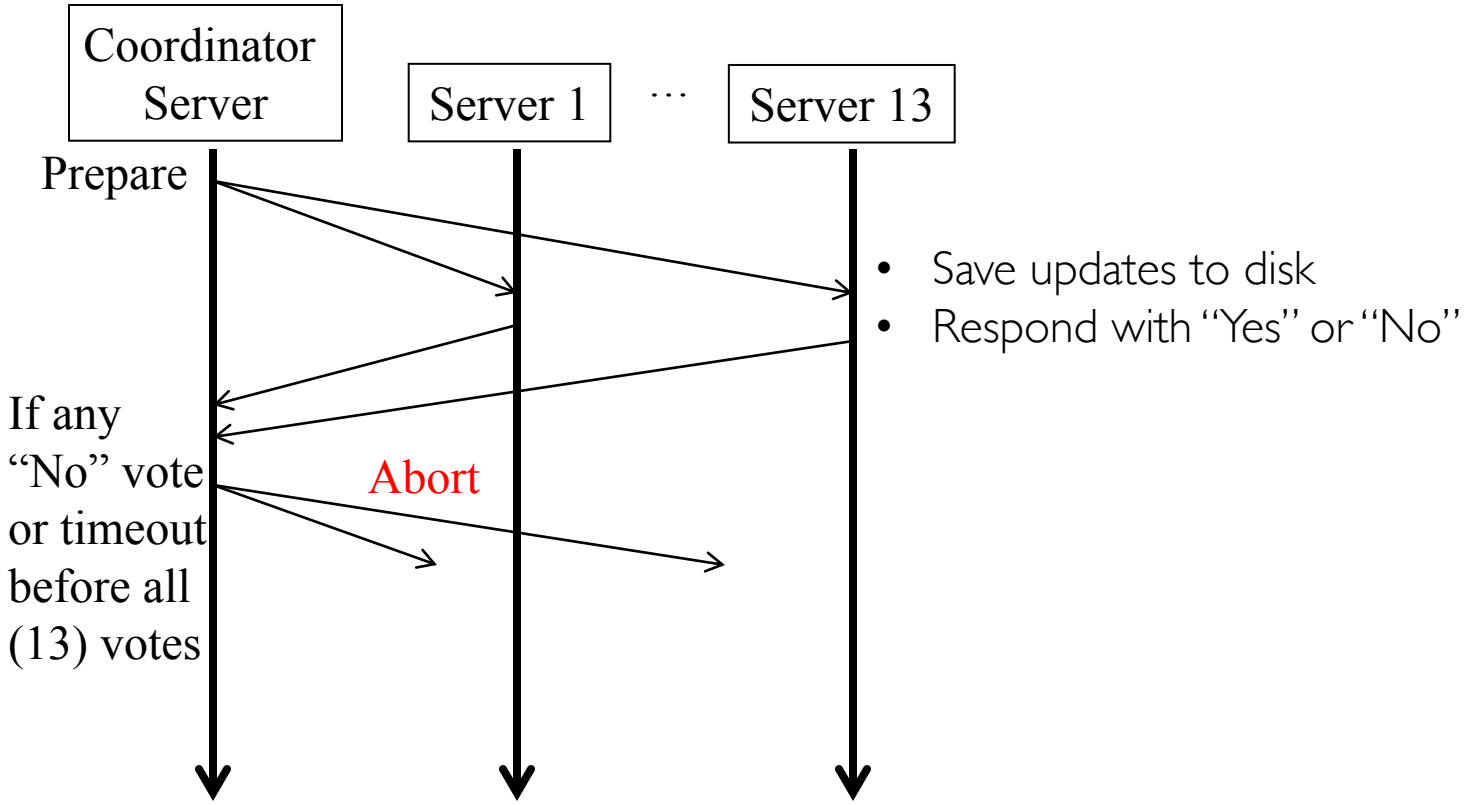
- Commit updates from disk to store

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

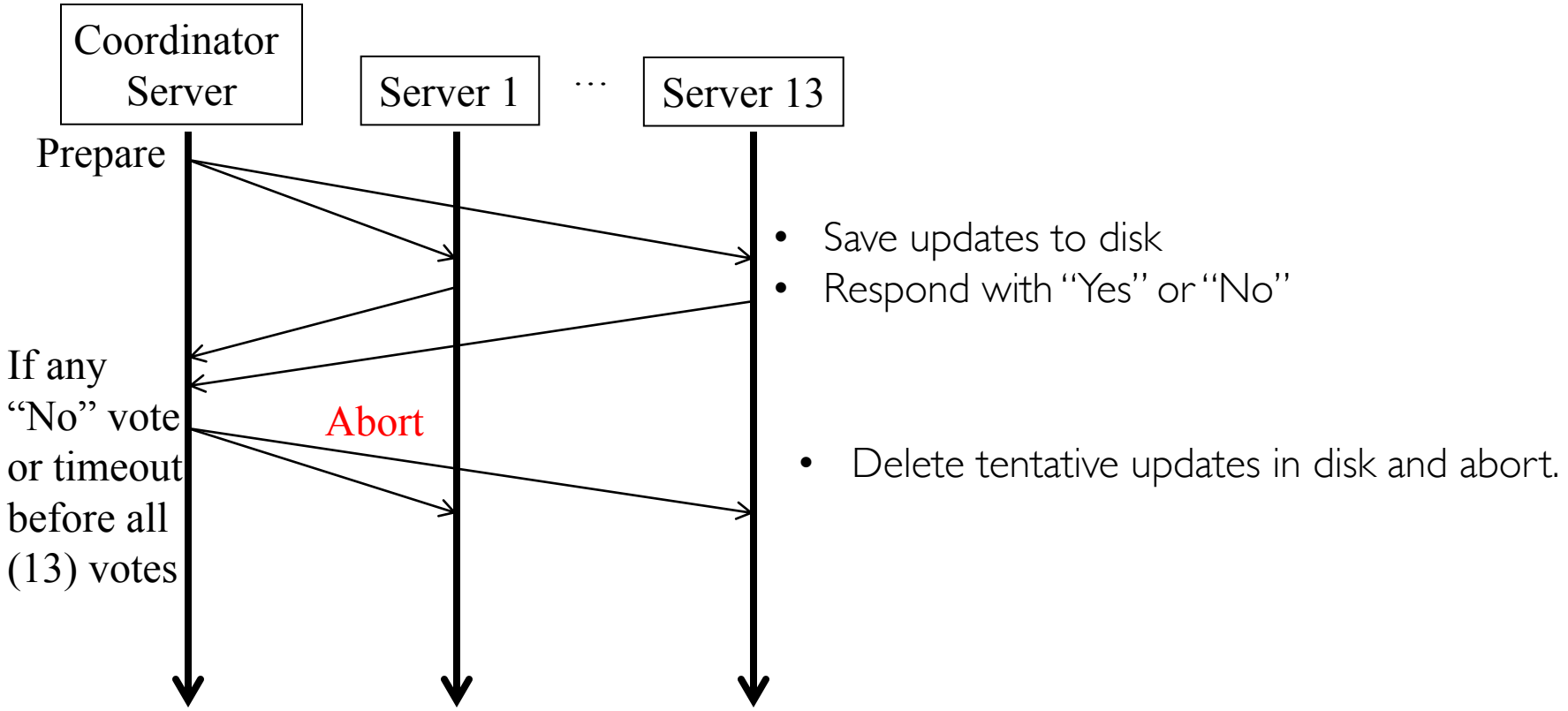
Two-phase commit



Two-phase commit



Two-phase commit



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 first phase. Retrievable after crash recovery.
- To deal with coordinator crashes
  - Coordinator logs all decisions and received/sent messages on disk.
  - After recovery => retrieve the logged state.

Failures in Two-phase Commit (contd)

- To deal with Prepare message loss
  - Coordinator will not receive yes/no from servers that didn't receive prepare message. It will therefore timeout and abort the transaction.
- 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, need to ensure
  - all these 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)

- Consensus is impossible in asynchronous system.
  - What makes two-phase commit work?
  - Crash failures in 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.*

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.*

Isolation with Distributed Transaction

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

- Servers are collectively responsible for serial equivalence of operations.

Timestamped Ordering with Distributed Transaction

- 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 2PC (two-phase commit) protocol.
- How to handle deadlocks?

Deadlock Detection in Distributed Transactions

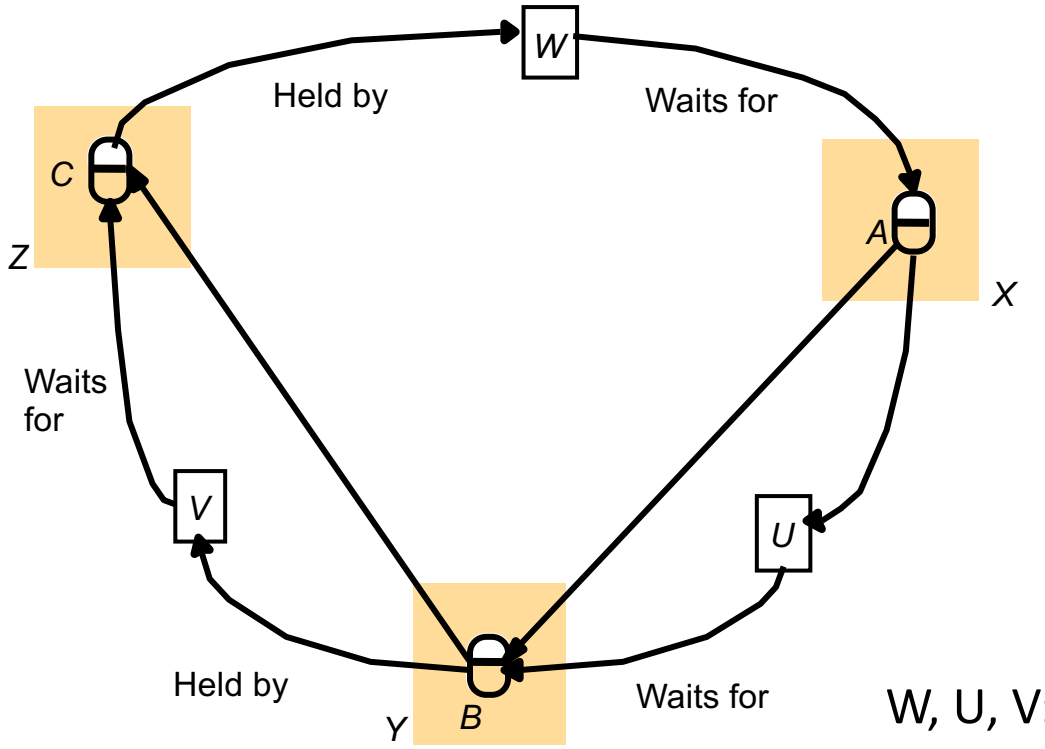
- The wait-for graph in a distributed set of transactions is distributed.

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

- Issues:
  - Single point of failure (can get blocked if the central server fails).
  - Scalability.

Decentralized Deadlock Detection

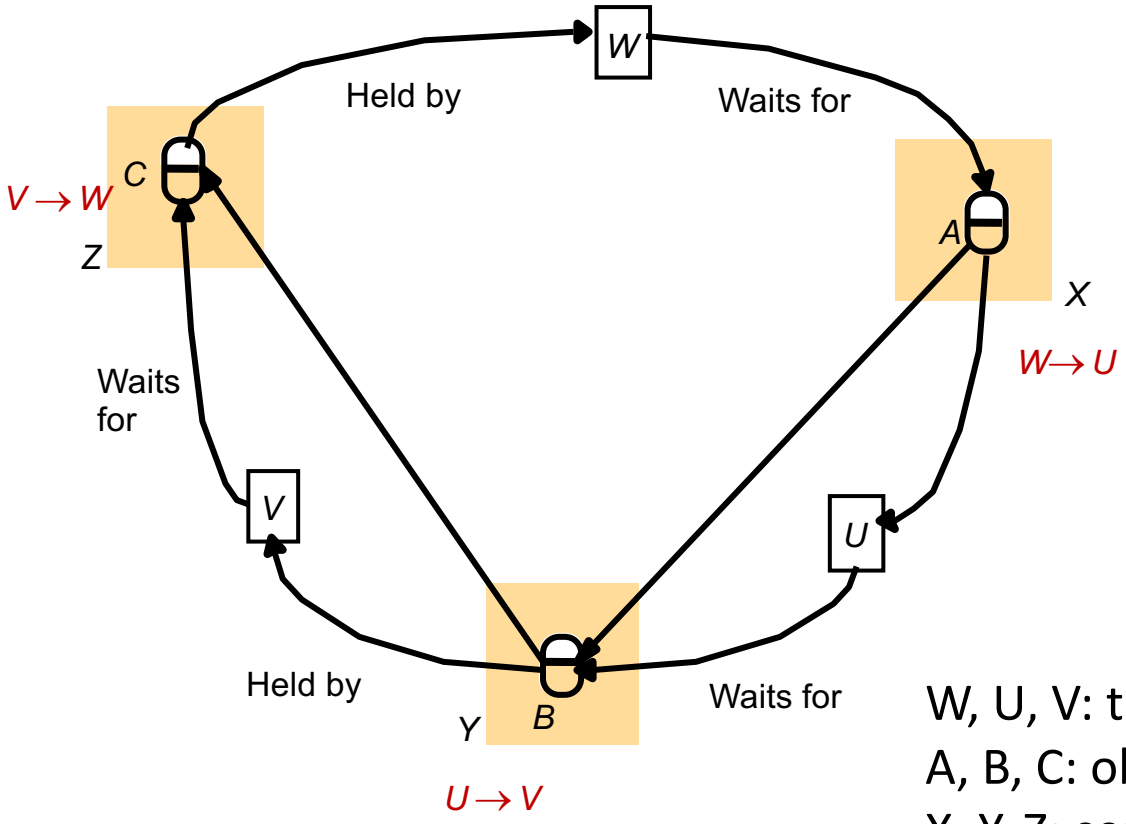
- **Edge chasing:** Forward "probe" messages to servers in the edges of wait-for graph, pushing the graph forward, until cycle is found.



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 cycle is found.



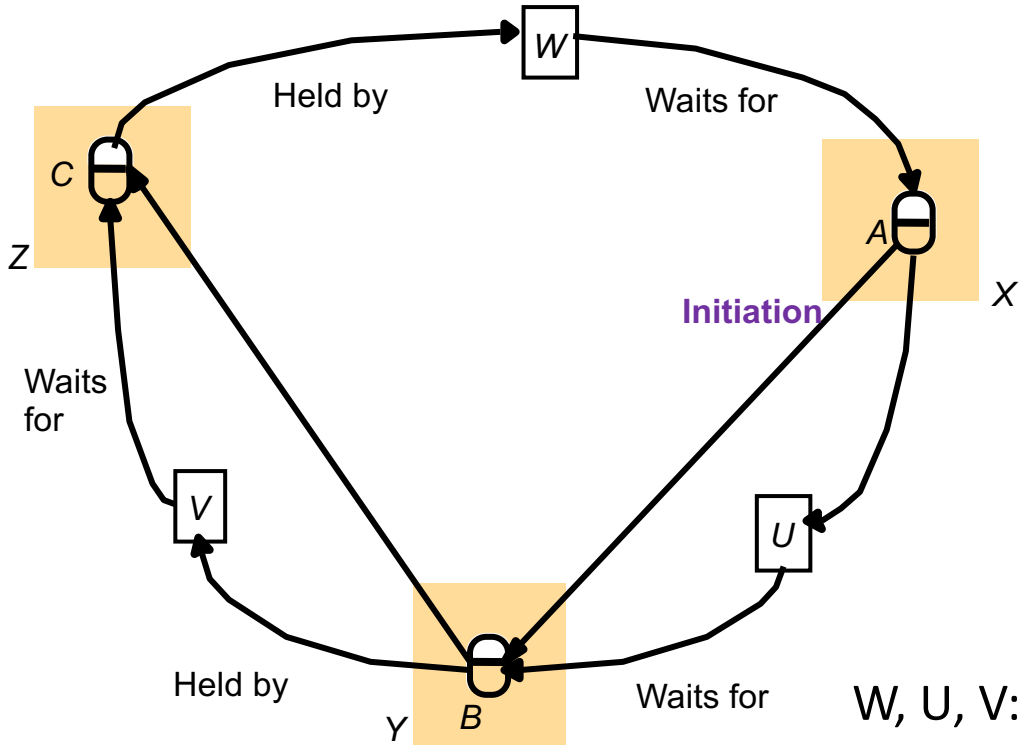
All servers know local wait-for relationships.

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.

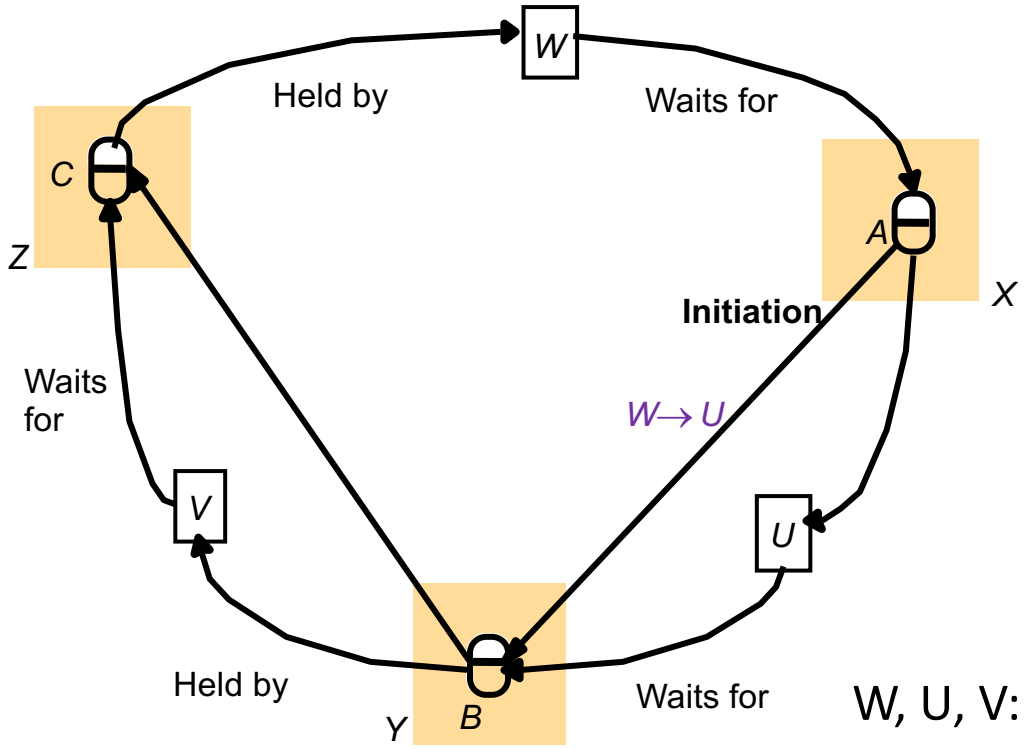


- Server X realizes W is waiting on U (a potential edge in the wait-for graph).
- Ask U's coordinator whether U is waiting on anything, and at which server.

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.

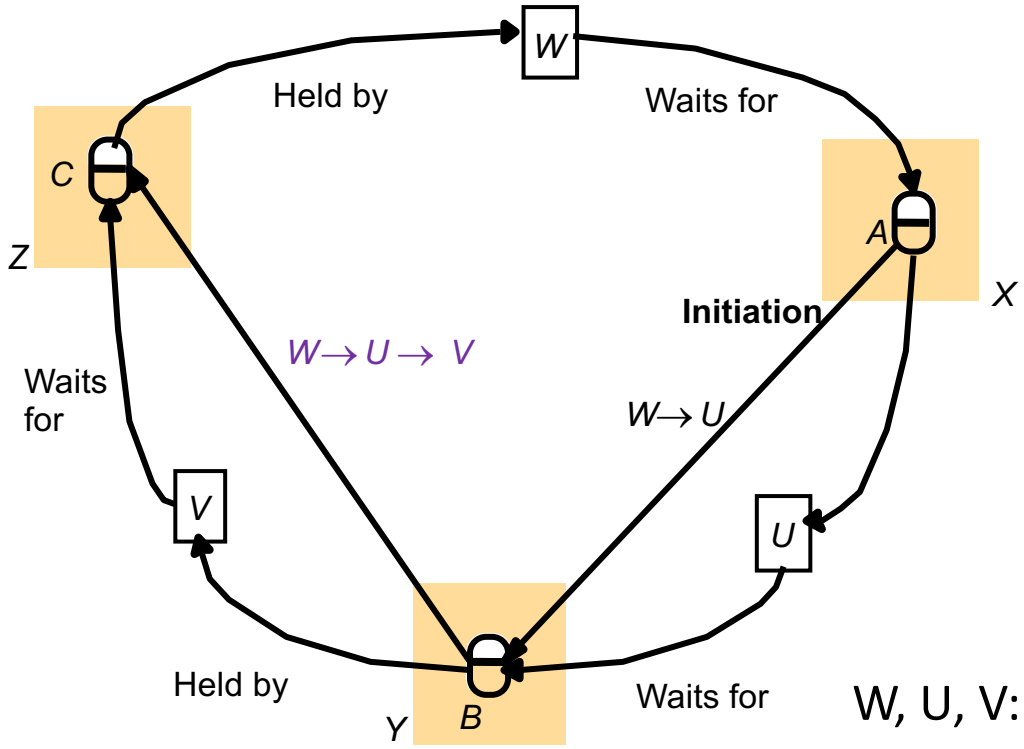


- Server X realizes W is waiting on U (a potential edge in the wait-for graph).
- Ask U's coordinator whether U is waiting on anything, and at which server.
- *Send a probe to the next server.*

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.

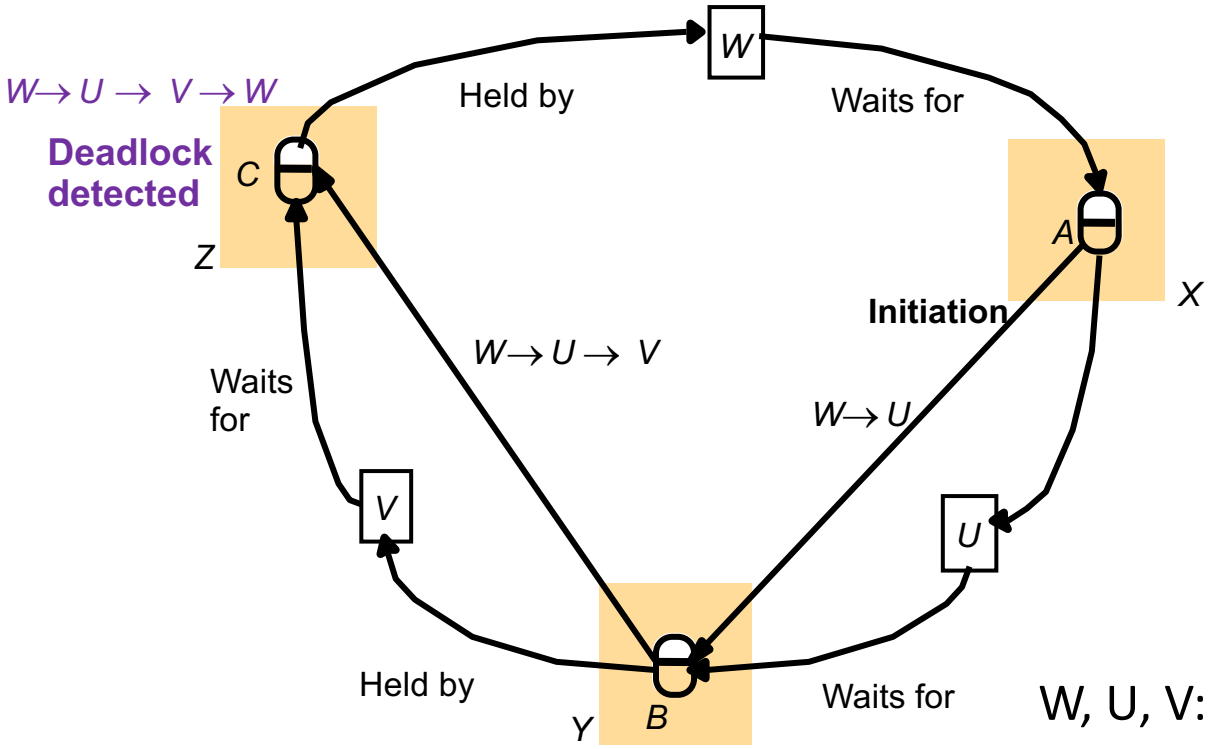


- Y adds another edge, and forwards the probe to the next server.

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.
- A 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
  - 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.
- Leads to spurious aborts.

Transaction Priority

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

Summary

- Distributed Transaction: Different objects that a transaction touches are stored on different servers.
  - One server process marked out as coordinator
  - Atomic Commit: 2PC
  - Deadlock detection: Centralized, Edge chasing

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

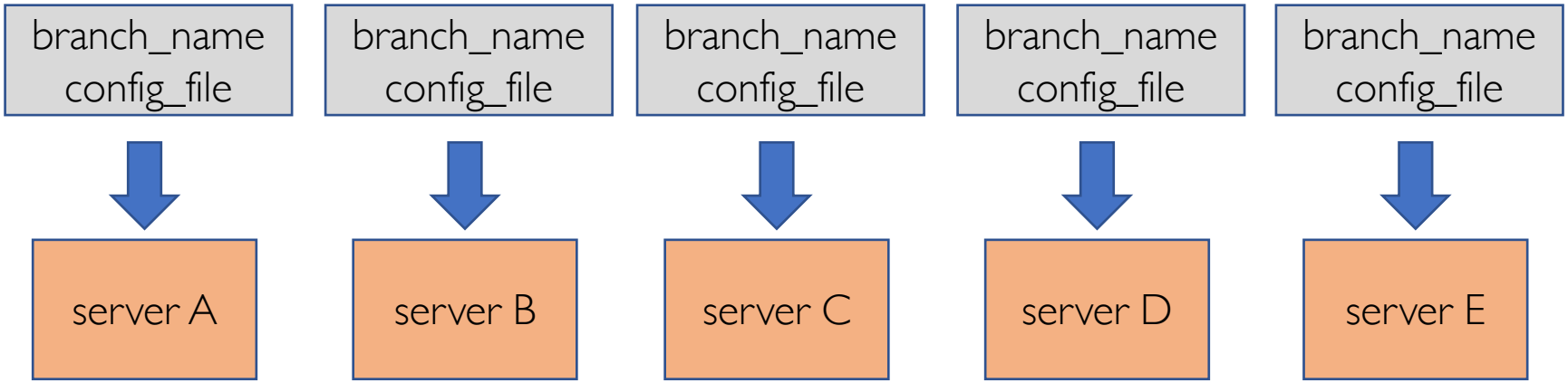
MP3: Distributed Transactions

- https://courses.grainger.illinois.edu/ece428/sp2026/mps/mp3.html
- Lead TA: Yu Li

- Task:
  - Build a distributed transaction system that satisfies ACI properties (you do not need to handle Durability).

- Objective:
  - Think through and implement algorithms for achieving atomicity and consistency with distributed transactions (two-phase commit), concurrency control (two-phase locking / timestamped ordering), deadlock detection.

MP3: Distributed Transactions

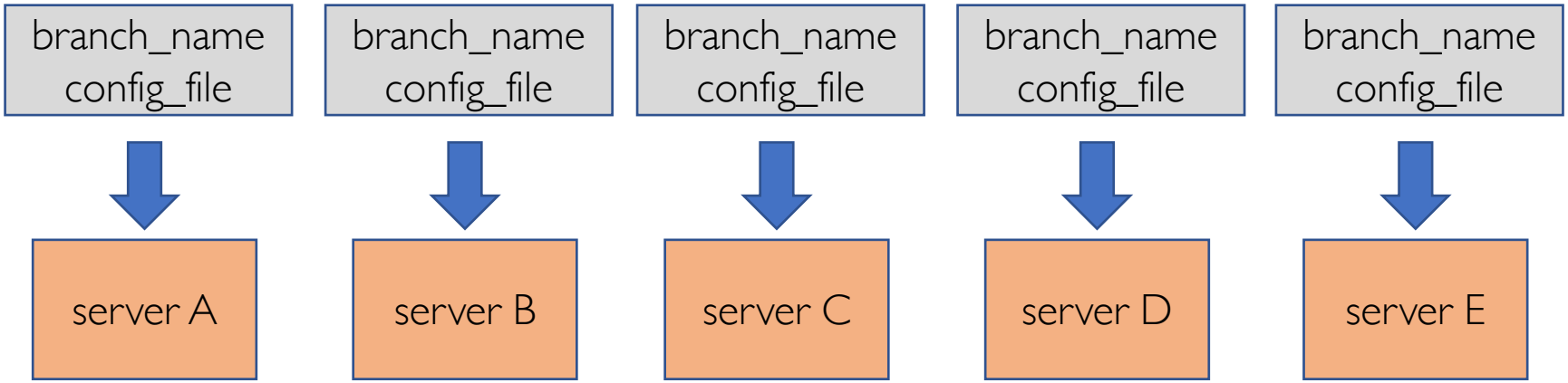


sample config_file

```
A sp23-cs425-0101.cs.illinois.edu 1234
B sp23-cs425-0101.cs.illinois.edu 1234
C sp23-cs425-0101.cs.illinois.edu 1234
D sp23-cs425-0101.cs.illinois.edu 1234
E sp23-cs425-0101.cs.illinois.edu 1234
```

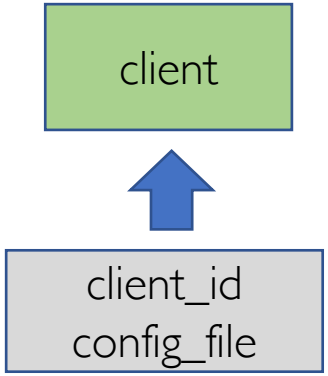
Use this information to establish communication across servers.

MP3: Distributed Transactions



sample config_file

```
A sp23-cs425-0101.cs.illinois.edu 1234
B sp23-cs425-0101.cs.illinois.edu 1234
C sp23-cs425-0101.cs.illinois.edu 1234
D sp23-cs425-0101.cs.illinois.edu 1234
E sp23-cs425-0101.cs.illinois.edu 1234
```

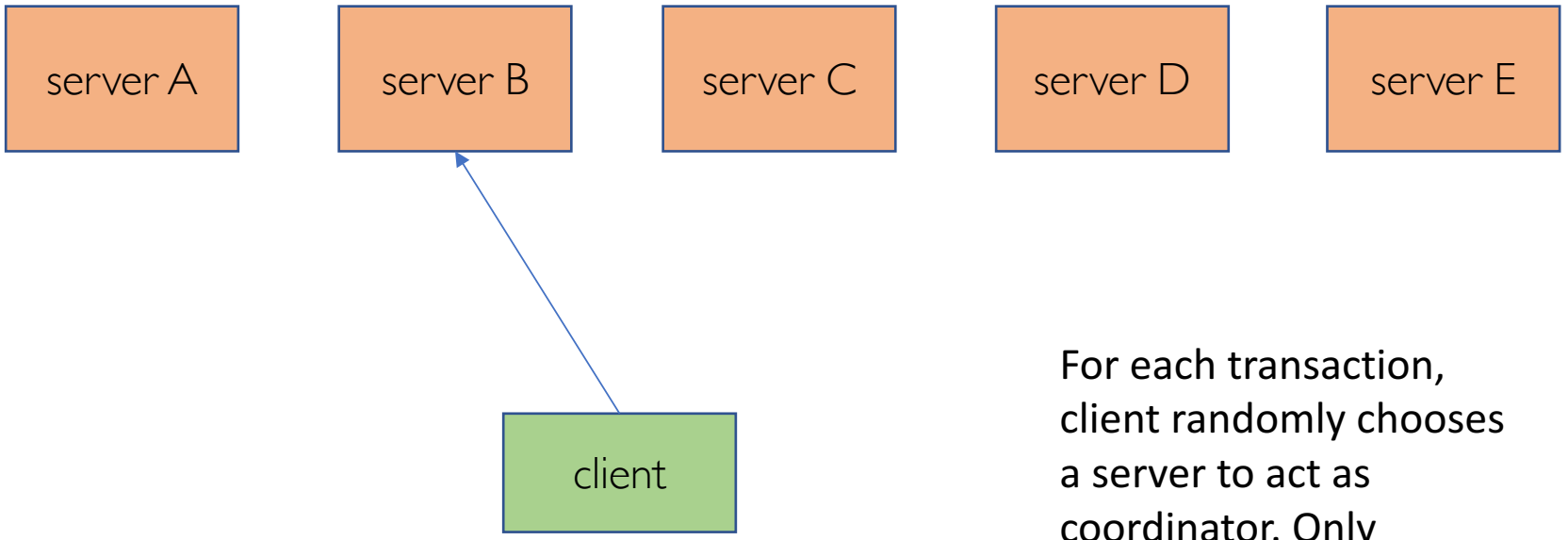


MP3: Distributed Transactions

Receives user input (command) from stdin.
Prints output of the command to stdout.

< BEGIN //start a new transaction

MP3: Distributed Transactions

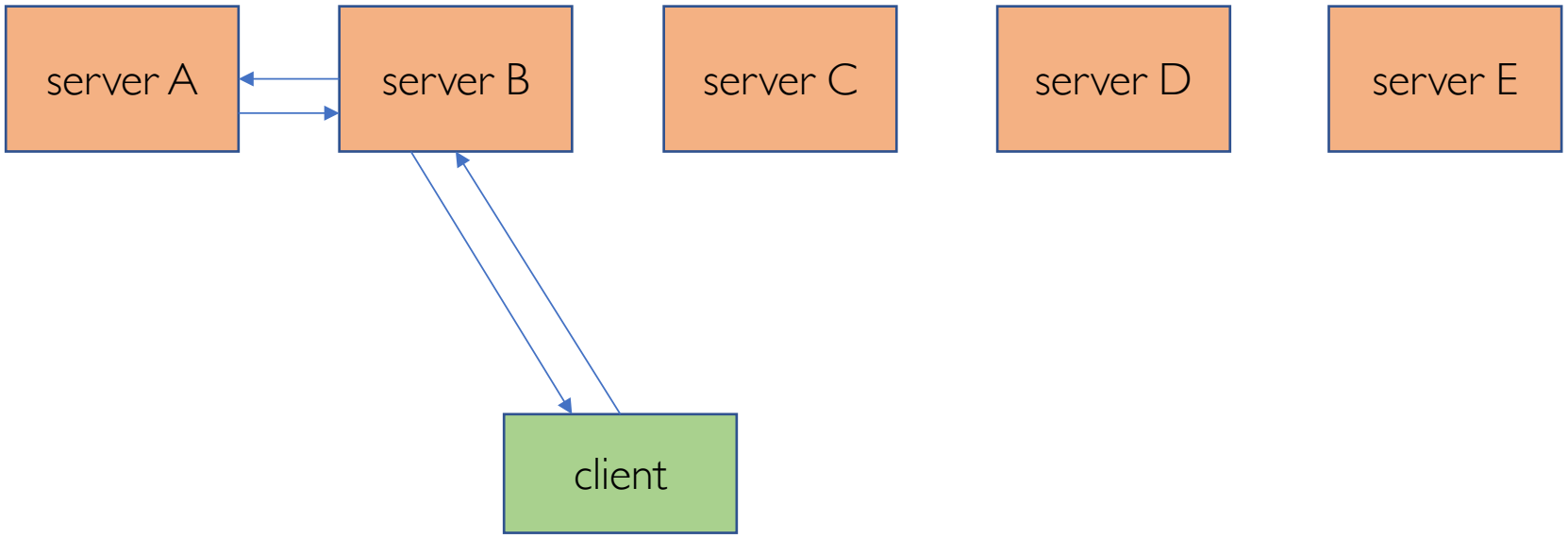


Receives user input (command) from stdin.
Prints output of the command to stdout.

For each transaction, client randomly chooses a server to act as coordinator. Only communicates with the coordinator

< BEGIN //start a new transaction
> OK
< DEPOSIT A.foo 10 //deposit 10 units in account foo at branch A

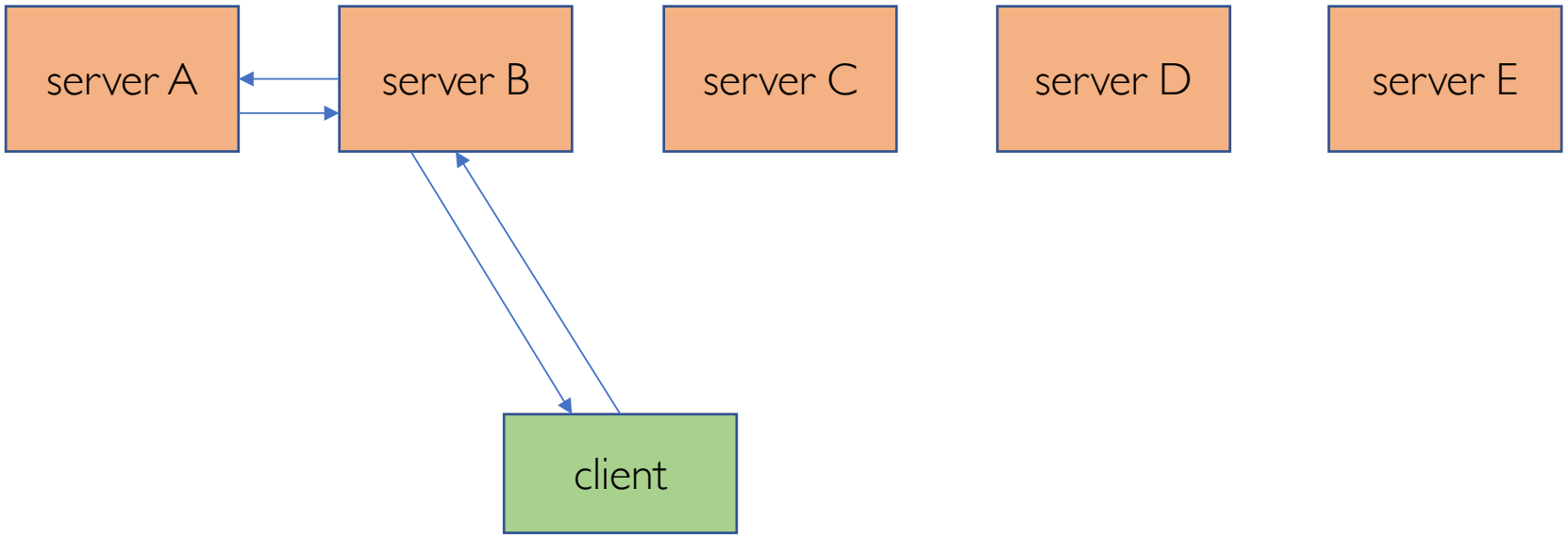
MP3: Distributed Transactions



Receives user input (command) from stdin.
Prints output of the command to stdout.

< BEGIN //start a new transaction
> OK
< DEPOSIT A.foo 10 //deposit 10 units in account foo at branch A
> OK

MP3: Distributed Transactions

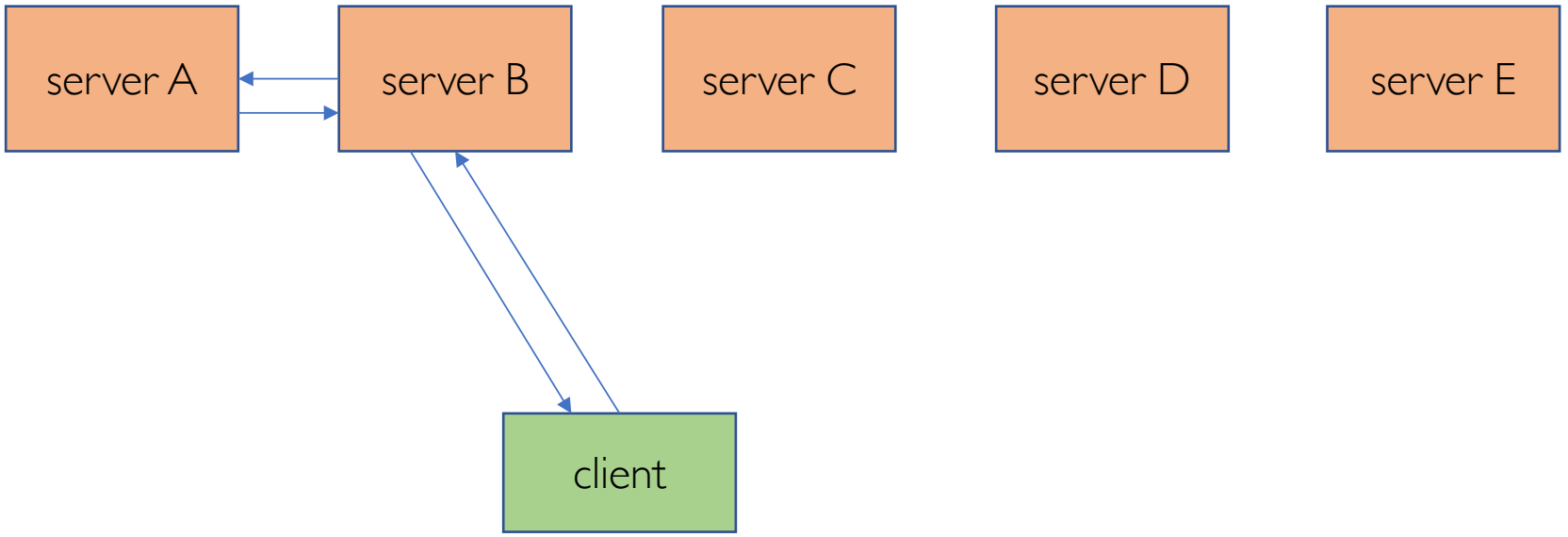


Receives user input (command) from stdin.
Prints output of the command to stdout.

```
< BEGIN //start a new transaction
> OK
< DEPOSIT A.foo 10 //deposit 10 units in account foo at branch A
> OK
```

Other possible commands: WITHDRAW and BALANCE (only applicable if the account exists)

MP3: Distributed Transactions



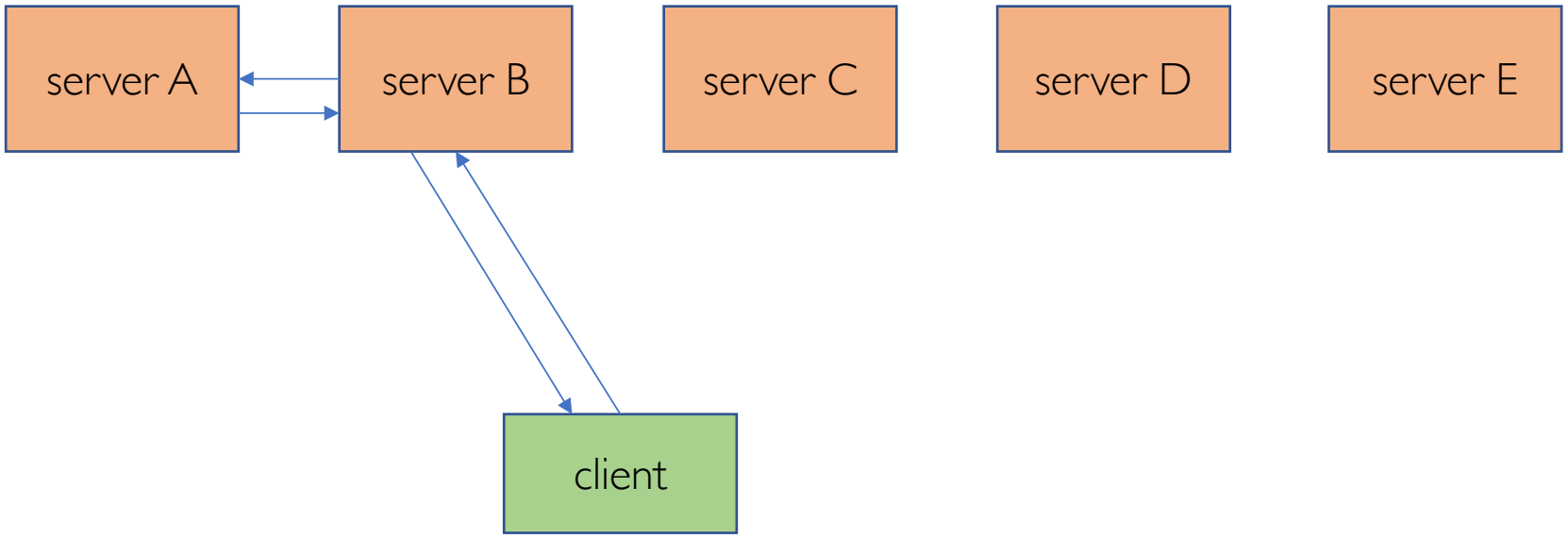
Receives user input (command) from stdin.
Prints output of the command to stdout.

User enters COMMIT or ABORT to end the transaction.

A server may also choose to ABORT a transaction (e.g. if consistency violated, or if needed for concurrency control).

Changes made by one transaction visible to others only after it successful commits.

MP3: Distributed Transactions

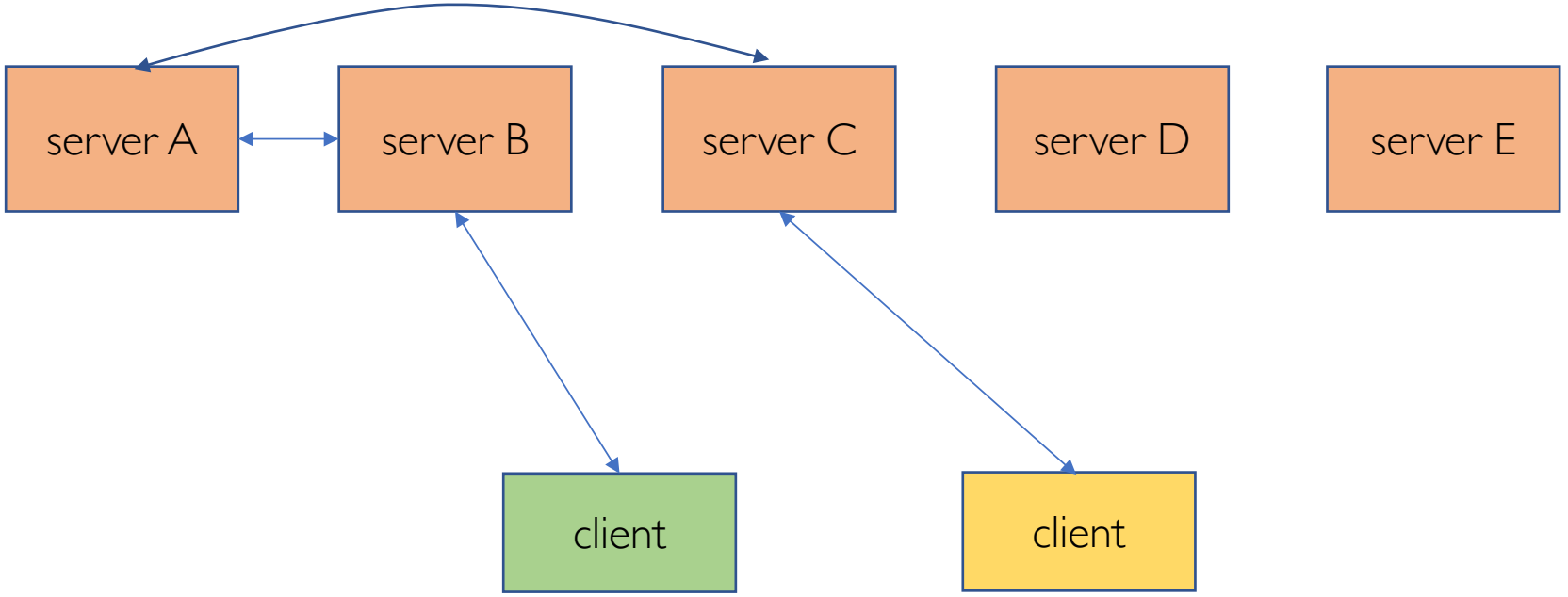


Receives user input (command) from stdin.
Prints output of the command to stdout.

Required properties:

- Atomicity:
  - all servers commit the entire transaction, or all rollback the entire transaction.
- Consistency:
  - cannot withdraw from or read balance of a non-existent account.
  - a transaction cannot result in a negative account balance.

MP3: Distributed Transactions



Receives user input (command) from stdin.
Prints output of the command to stdout.

Required properties:

- Isolation:
  - multiple clients may concurrently issue commands on the object.
  - Must provide serial equivalence.
- Deadlock avoidance.

MP3: Distributed Transactions

- Due on May 1st.
  - Late policy: Can use remainder of your 168hours of grace period accounted per student over the entire semester.
- Read the specification fully and carefully.
  - Required semantics discussed more completely there.
- Start early!