Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for the materials: Indy Gupta and Nikita Borisov

Logistics

- HW3 due on Wednesday.

- HW4 will be released on Wednesday.

- Midterm 2 next week (April 2-4)
  - See post on Campuswire.
  - Cheatsheet available on Campuswire.
  - Some practice questions will be released on Saturday.

- Class on April 1st: Raft tutorial (by Sarthak) + Q/A with TAs
- My office hour on April 1st will be online (over Zoom).
- Exam syllabus review: last 15mins of Mar 27 lecture.

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

Timestamped ordering

- Assign each transaction an id
- Transaction id determines its position in serialization order.
- Ensure that for a transaction T, both are true:
  1. T's write to object O allowed only if transactions that have read or written O had lower ids than T.
  2. T's read to object O is allowed only if O was last written by a transaction with a lower id than T.
- Implemented by maintaining read and write timestamps for the object
- If rule violated, abort!
- *Never results in a deadlock! Older transaction never waits on newer ones.*

Timestamped ordering: write rule

Transaction T_c requests a write operation on object D

if (Tc ≥ max. read timestamp on D

&& Tc > write timestamp on committed version of D)

Perform a tentative write on D:

If T_c already has an entry in the TW list for D, update it.

Else, add T_c and its write value to the TW list.

else

abort transaction T_c

//too late; a transaction with later timestamp has already read or written the object.

Timestamped ordering: read rule

Transaction T_c requests a read operation on object D

if (T_c > write timestamp on committed version of D) {

D_s = version of D with the maximum write timestamp that is ≤ T_c

//search across the committed timestamp and the TW list for object D.

if (D_s is committed)

read D_s and add T_c to RTS list (if not already added)

else

if D_s was written by T_c, simply read D_s

else

wait until the transaction that wrote D_s is committed or aborted, and reapply the read rule.

// if the transaction is committed, T_c will read its value after the wait.

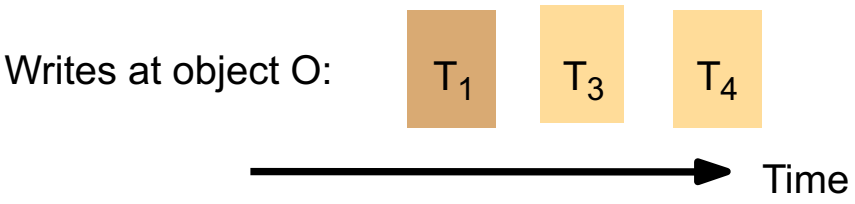
// if the transaction is aborted, T_c will read the value from an older transaction.

} else

abort transaction T_c

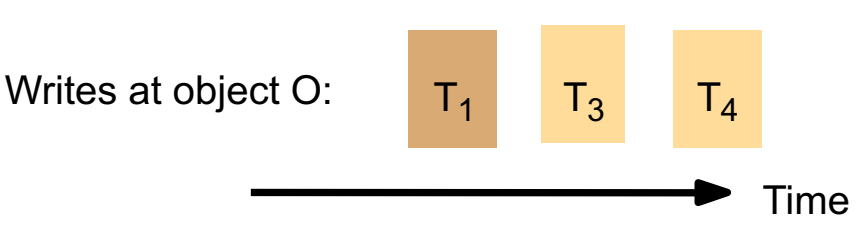
//too late; a transaction with later timestamp has already written the object.

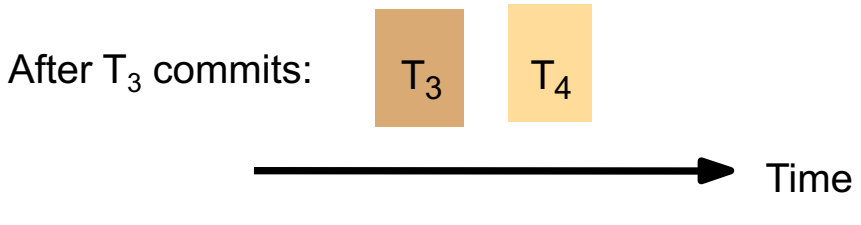
Timestamped ordering: committing



- Suppose T_4 is ready to commit.
- Must wait until T_3 commits or aborts.

- When a transaction is committed, the committed value of the object and associated timestamp are updated, and the corresponding write is removed from TW list.





Lost Update Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);

if(x > 1)

x = x – 1;

write(x, ABC123);

commit

Transaction T2

x = getSeats(ABC123);

if(x > 1)

x = x – 1;

write(x, ABC123);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1, 2
TW:

Abort!

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS:
TW:

ABC789: state
committed value = 5
committed timestamp = 0
RTS:
TW:

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS:
TW:

ABC789: state
committed value = 5
committed timestamp = 0
RTS:
TW:

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW:

ABC789: state
committed value = 5
committed timestamp = 0
RTS:
TW:

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW:

ABC789: state
committed value = 5
committed timestamp = 0
RTS:
TW:

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW:

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW:

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW:

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW:

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW: (5, 1)

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW:

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW: (5, 1)

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW:

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123); wait
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW: (5, 1)

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW:

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123); wait
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW: (5, 1)

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW:

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123); wait
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW: (5, 1)

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW: (10, 1)

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123); wait
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW: (5, 1)

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW: (10, 1)

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123); wait
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = ~~10~~ 5
committed timestamp = ~~0~~ 1
RTS: 1
TW: ~~(5, 1)~~

ABC789: state
committed value = ~~5~~ 10
committed timestamp = ~~0~~ 1
RTS: 1
TW: ~~(10, 1)~~

Next Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123); wait
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = ~~10~~ 5
committed timestamp = ~~0~~ 1
RTS: 1
TW: ~~(5, 1)~~

ABC789: state
committed value = ~~5~~ 10
committed timestamp = ~~0~~ 1
RTS: 1
TW: ~~(10, 1)~~

T2 then proceeds after T1 commits

Timestamped ordering

- Assign each transaction an id
- Transaction id determines its position in serialization order.
- Ensure that for a transaction T, both are true:
  1. T's write to object O allowed only if transactions that have read or written O had lower ids than T.
  2. T's read to object O is allowed only if O was last written by a transaction with a lower id than T.
- Implemented by maintaining read and write timestamps for the object
- If rule violated, abort!
- *Never results in a deadlock! Older transaction never waits on newer ones.*

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 too late.
    - timestamped ordering.

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)

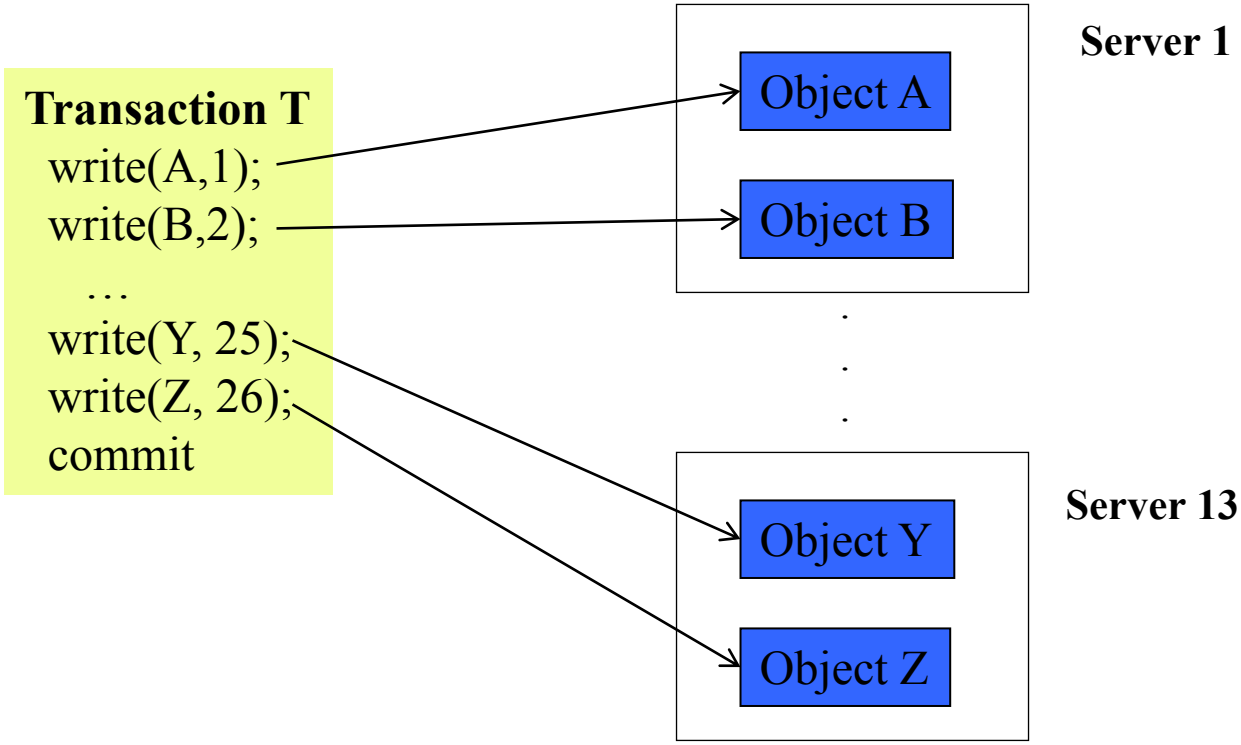
- Distributed Transactions

Distributed Transactions

- Transaction processing can be *distributed* across multiple servers.
  - Different objects can be stored on different servers.
    - Our 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

- **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

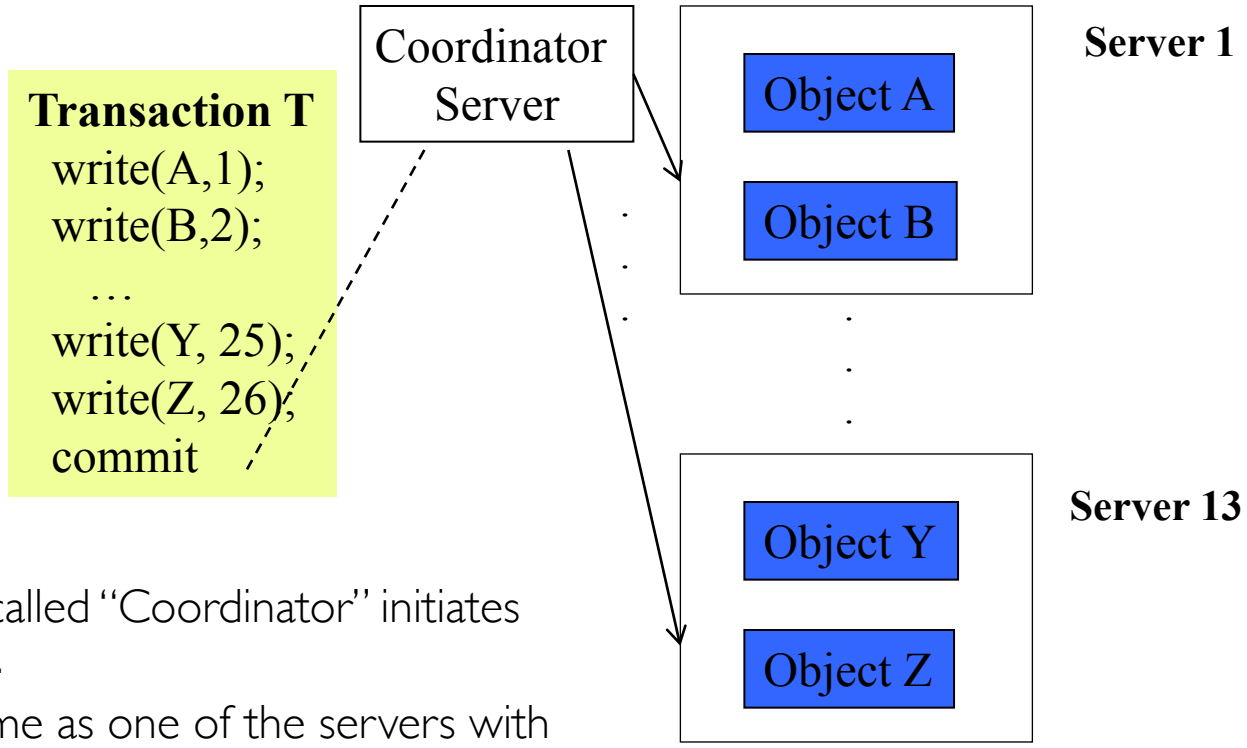
- **Atomic: all-or-nothing**
  - ***Must ensure atomicity across servers.***
- **C**onsistent: rules maintained
  - *Generally done locally, but may need to check non-local invariants at commit time.*
- **Isolation: multiple transactions do not interfere with each other**
  - ***Locks at each server. 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



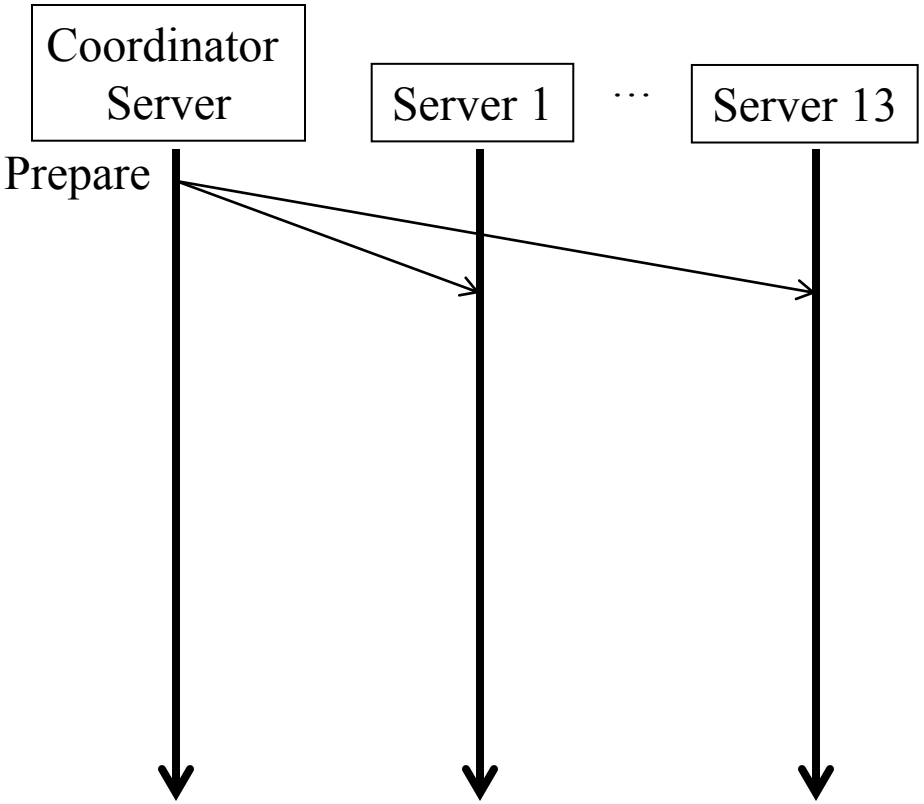
- Special server called "Coordinator" initiates atomic commit.
  - can be same as one of the servers with objects.
- Different transactions may have different coordinators.

One-phase commit

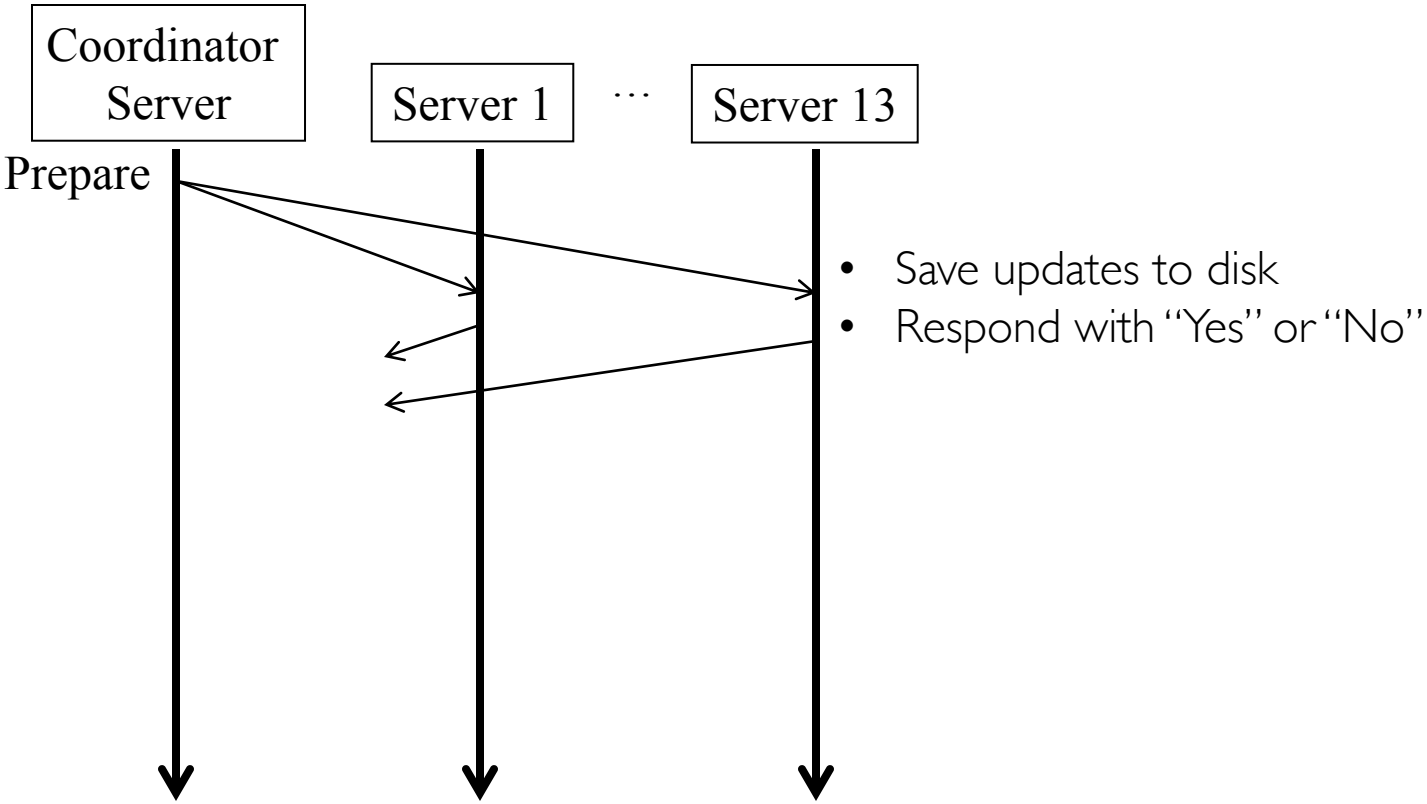
- Client relays the "commit" or "abort" command to the coordinator.
  - Coordinator tells other servers to commit / abort.

- *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.

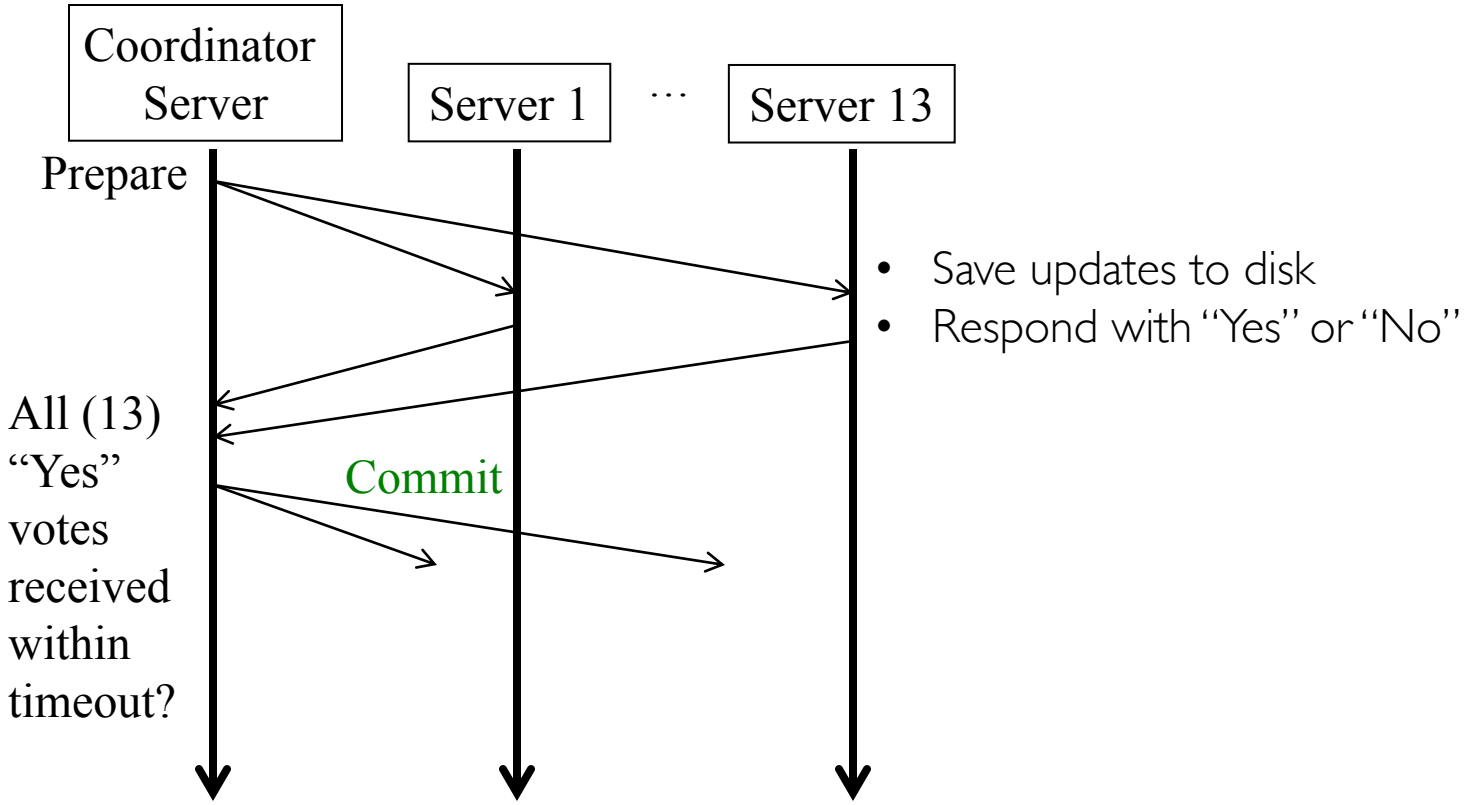
Two-phase commit



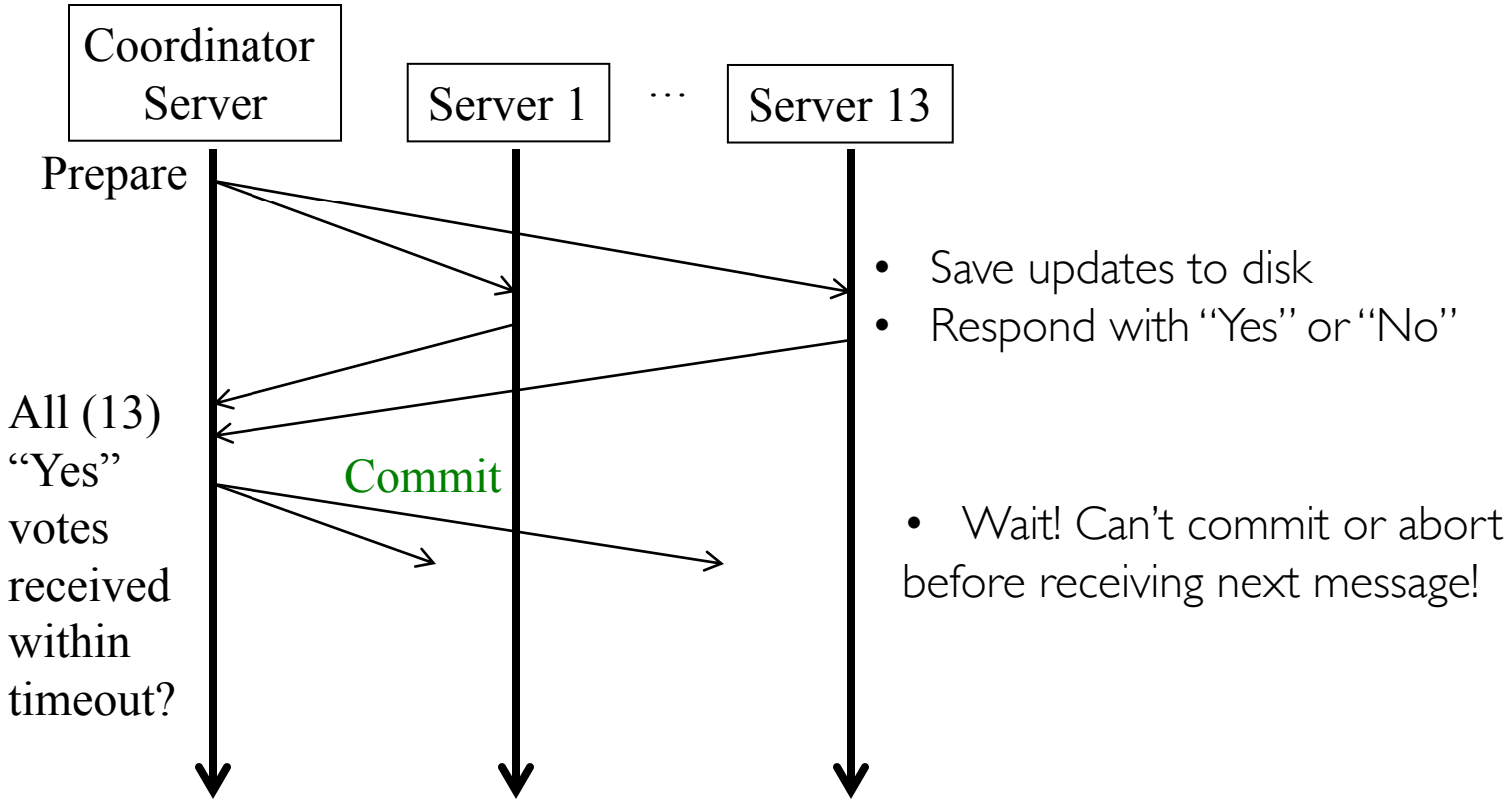
Two-phase commit



Two-phase commit



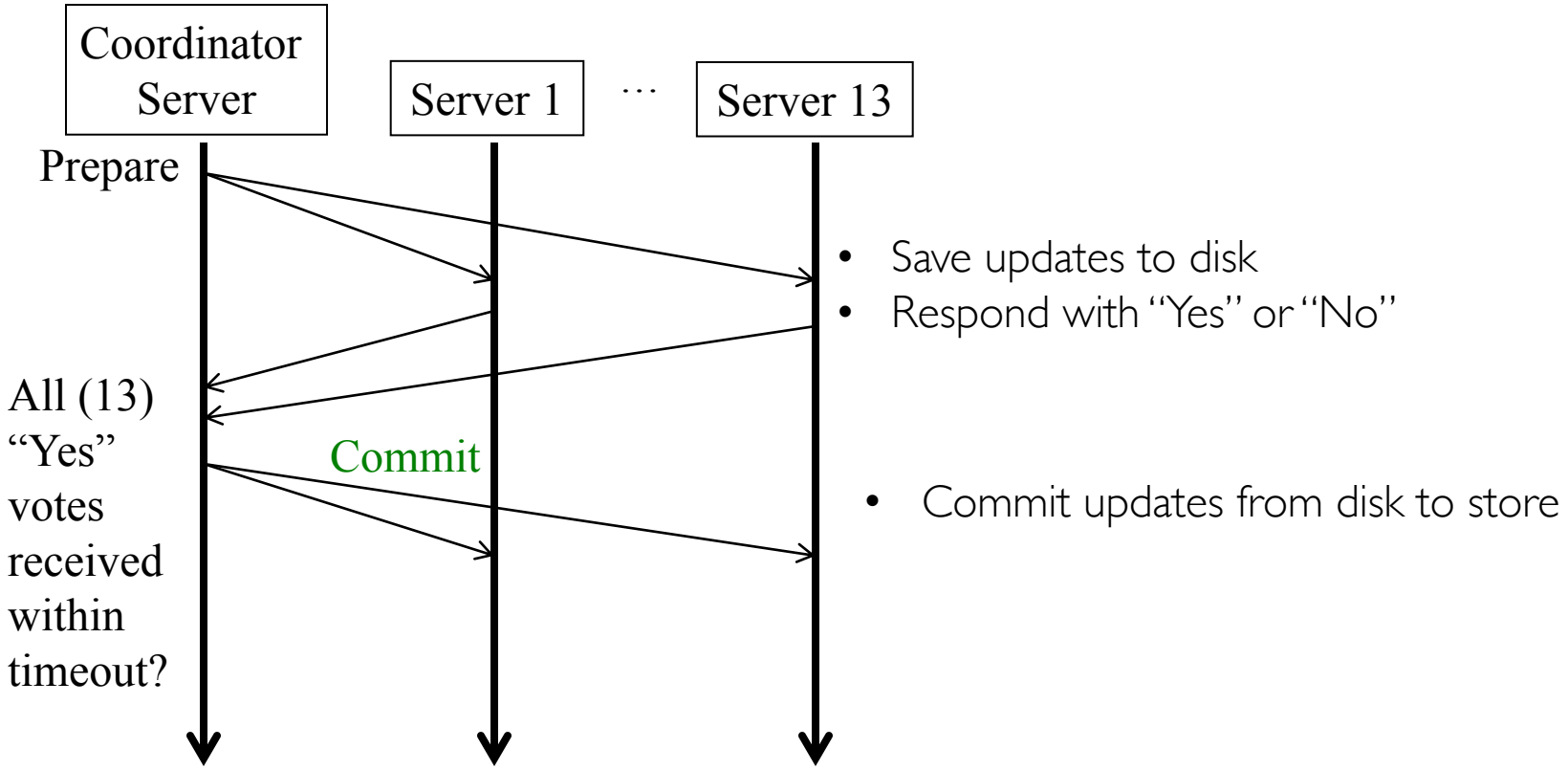
Two-phase commit



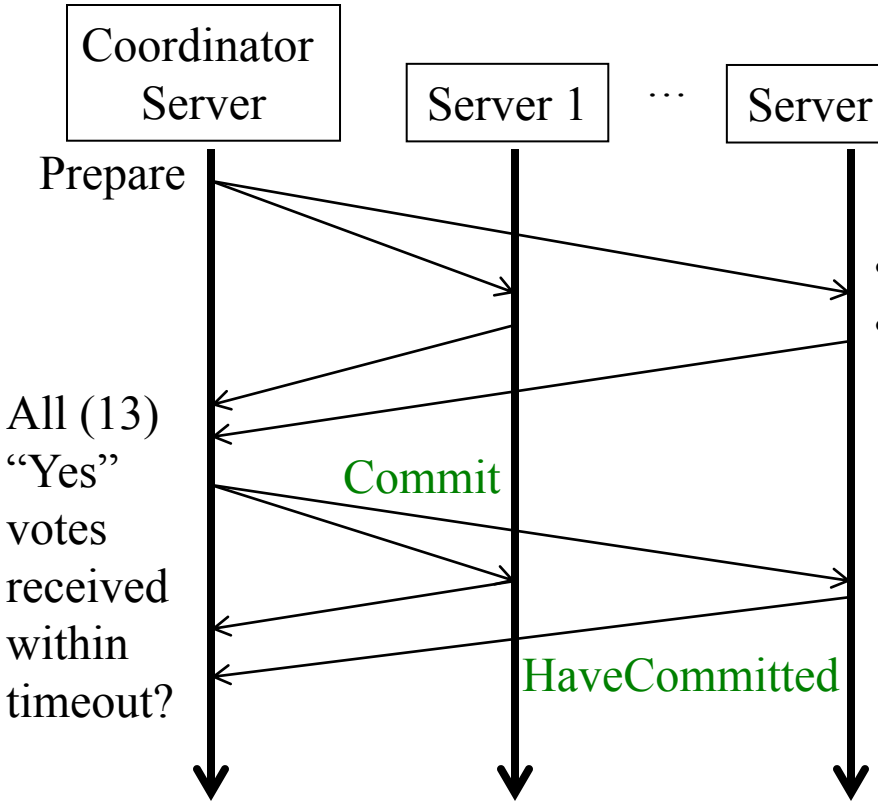
- Save updates to disk
- Respond with "Yes" or "No"

- Wait! Can't commit or abort before receiving next message!

Two-phase commit



Two-phase commit

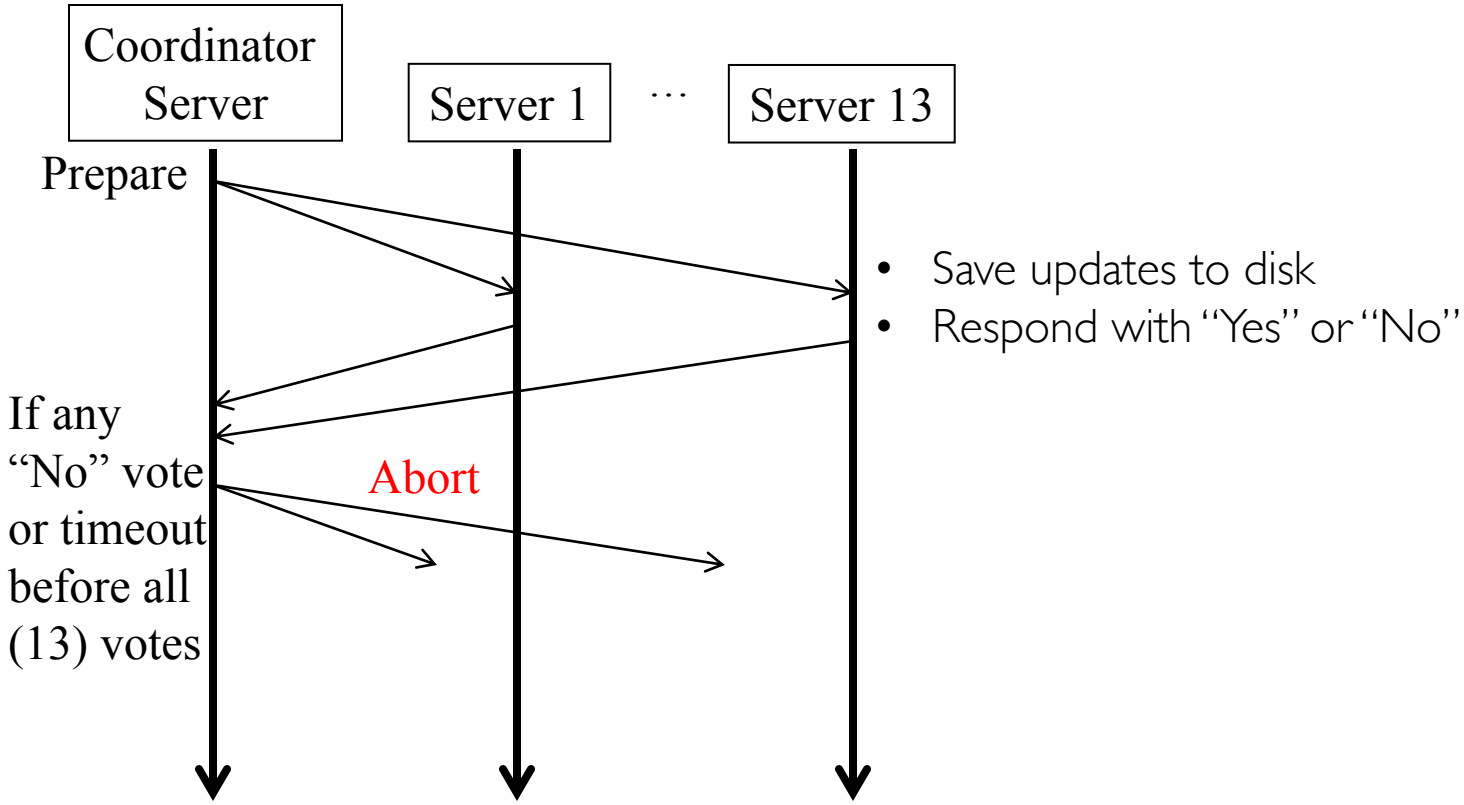


- Save updates to disk
- Respond with "Yes" or "No"

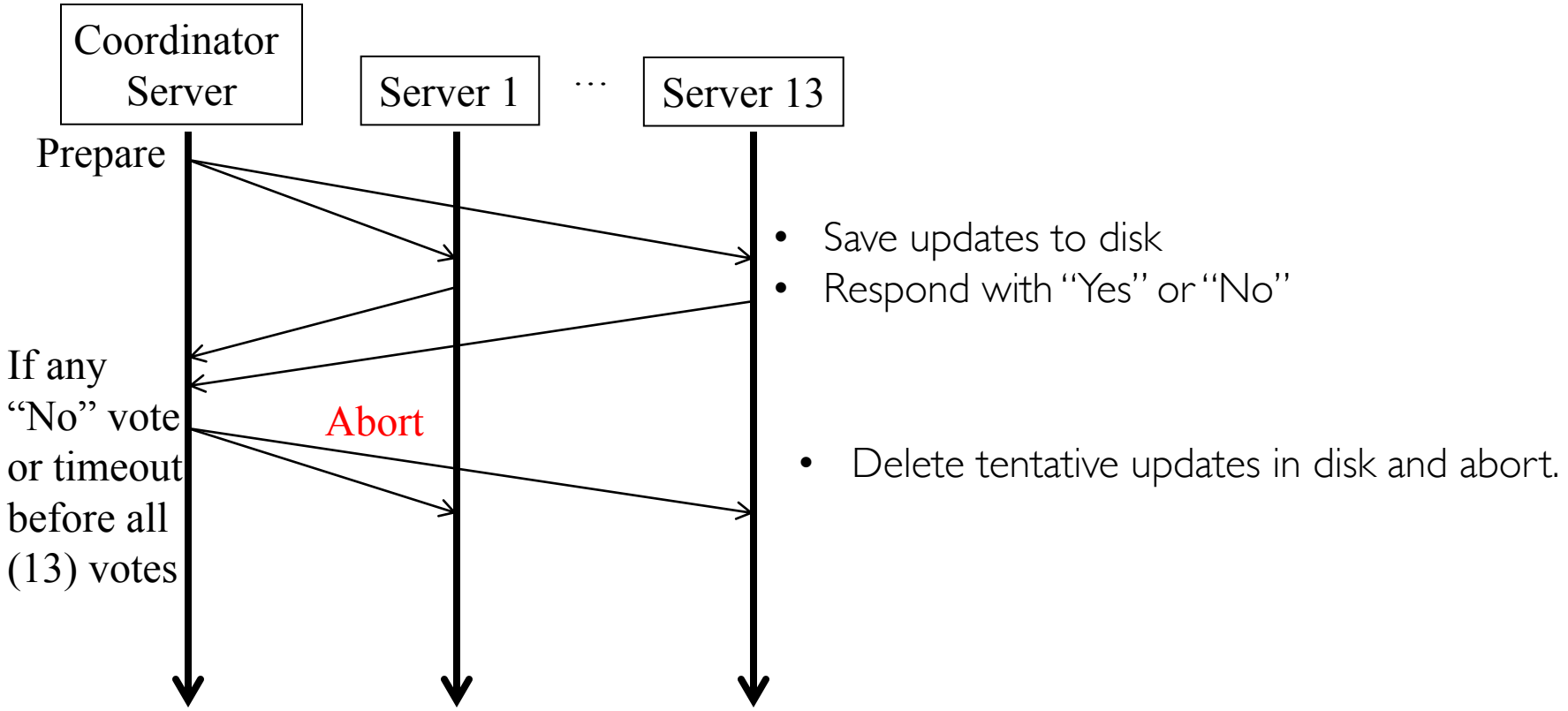
- Commit updates from disk to store

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

Two-phase commit



Two-phase commit



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.
  - Or: the server may decide to abort unilaterally after a timeout for first phase (and will send abort to coordinator, causing the transaction to 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, 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

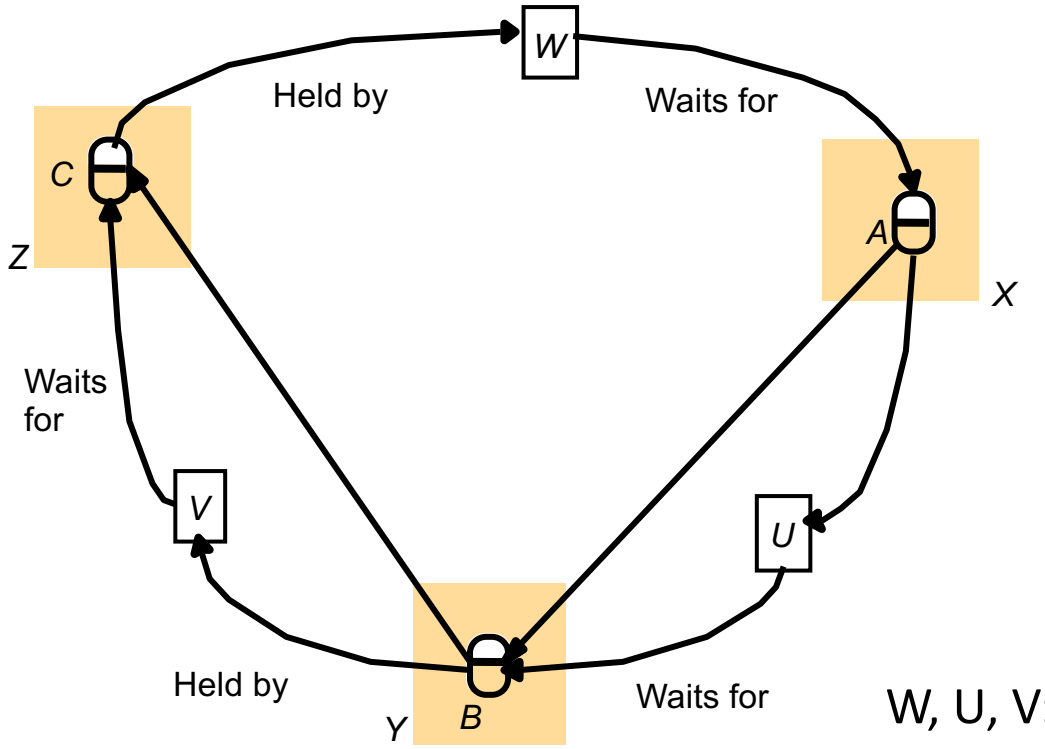
- The wait-for graph in a distributed set of transactions is distributed.

- 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

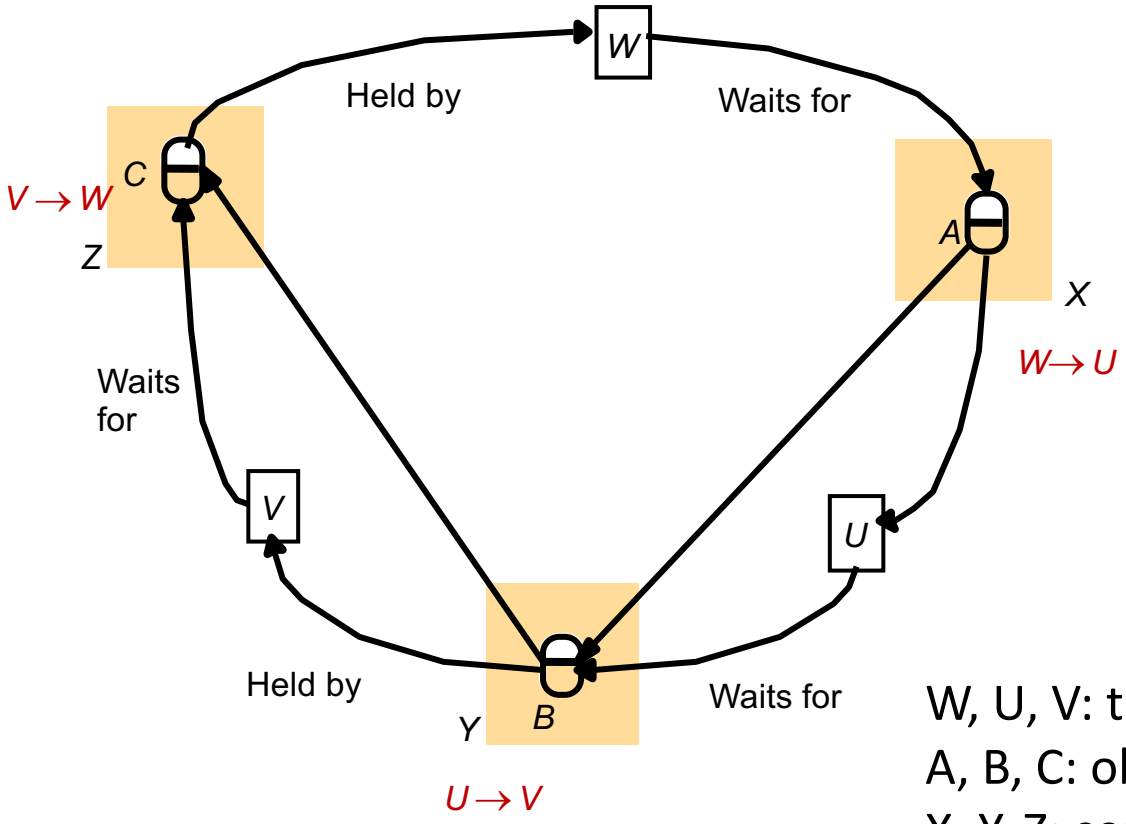
- **Edge chasing:** Forward "probe" messages to servers in the edges of wait-for graph, pushing the graph forward, until cycle is found.



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.



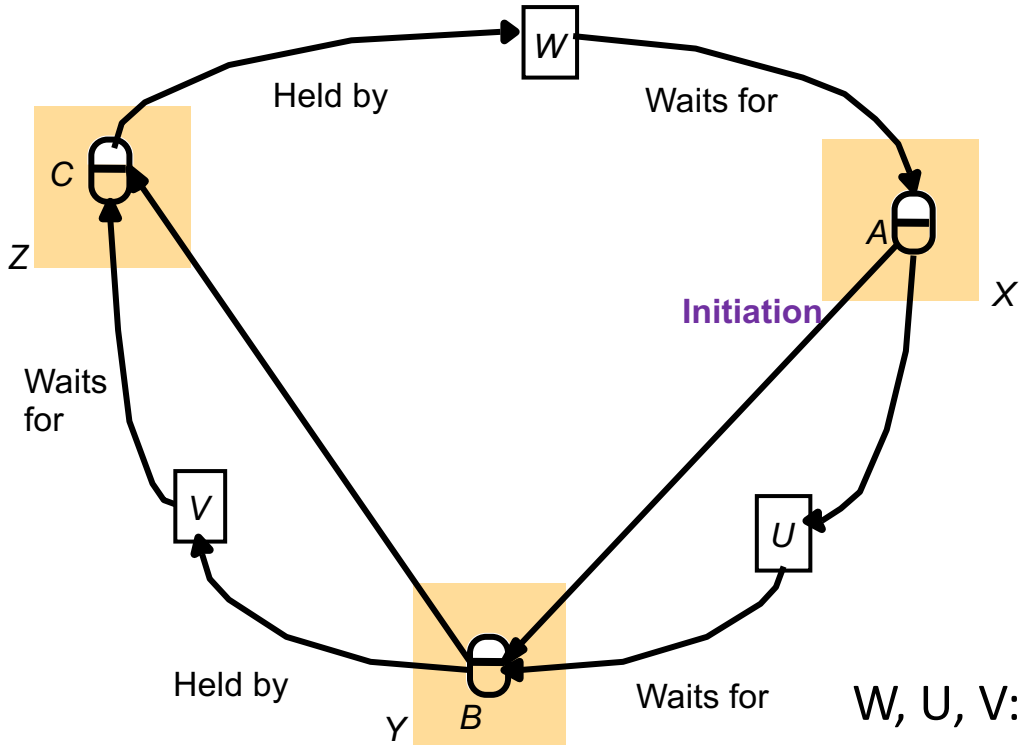
All servers know local wait-for relationships.

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.

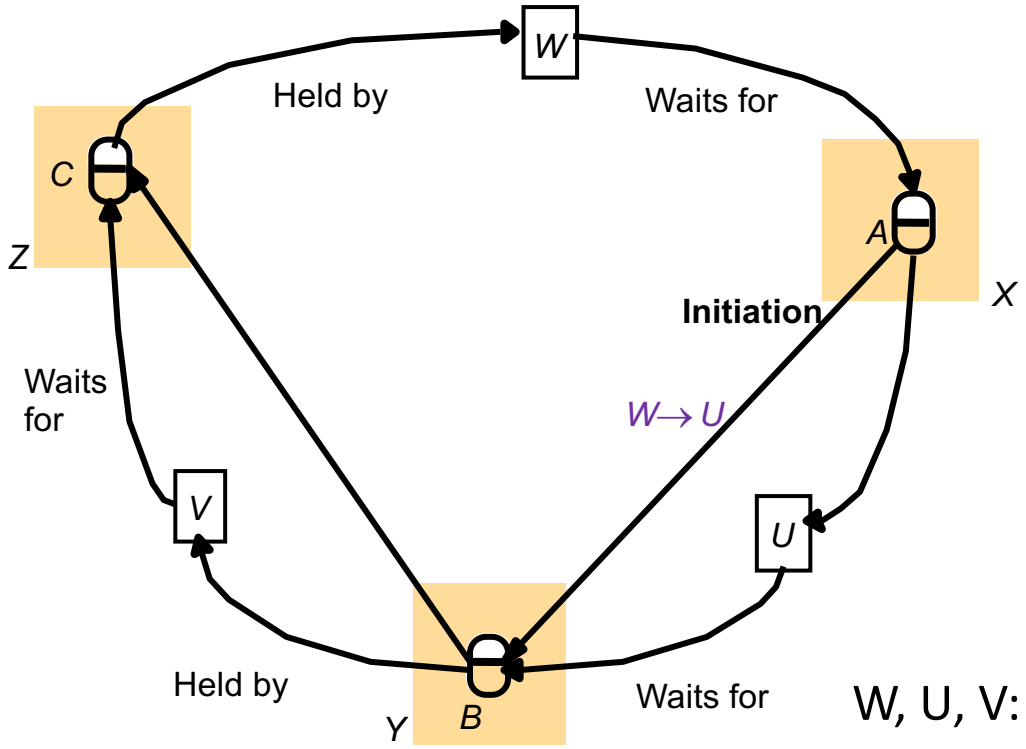


- Server X realizes W is waiting on U (a potential edge in the wait-for graph).
- Ask U's coordinator whether U is waiting on anything, and at which server.

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.

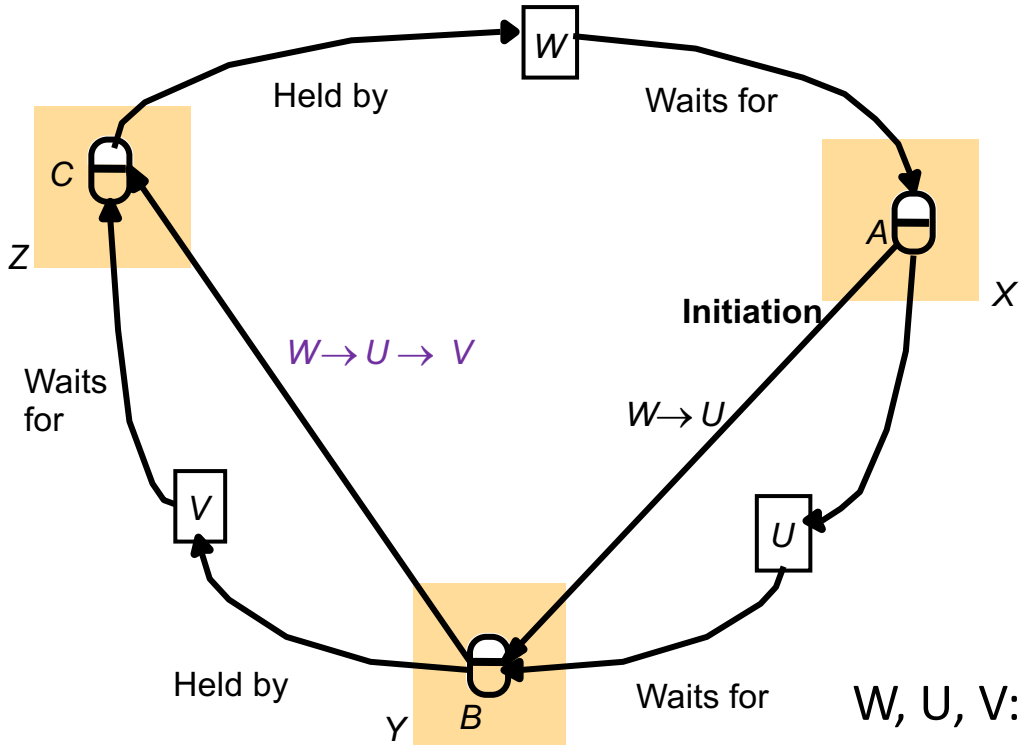


- Server X realizes W is waiting on U (a potential edge in the wait-for graph).
- Ask U's coordinator whether U is waiting on anything, and at which server.
- *Send a probe to the next server.*

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.

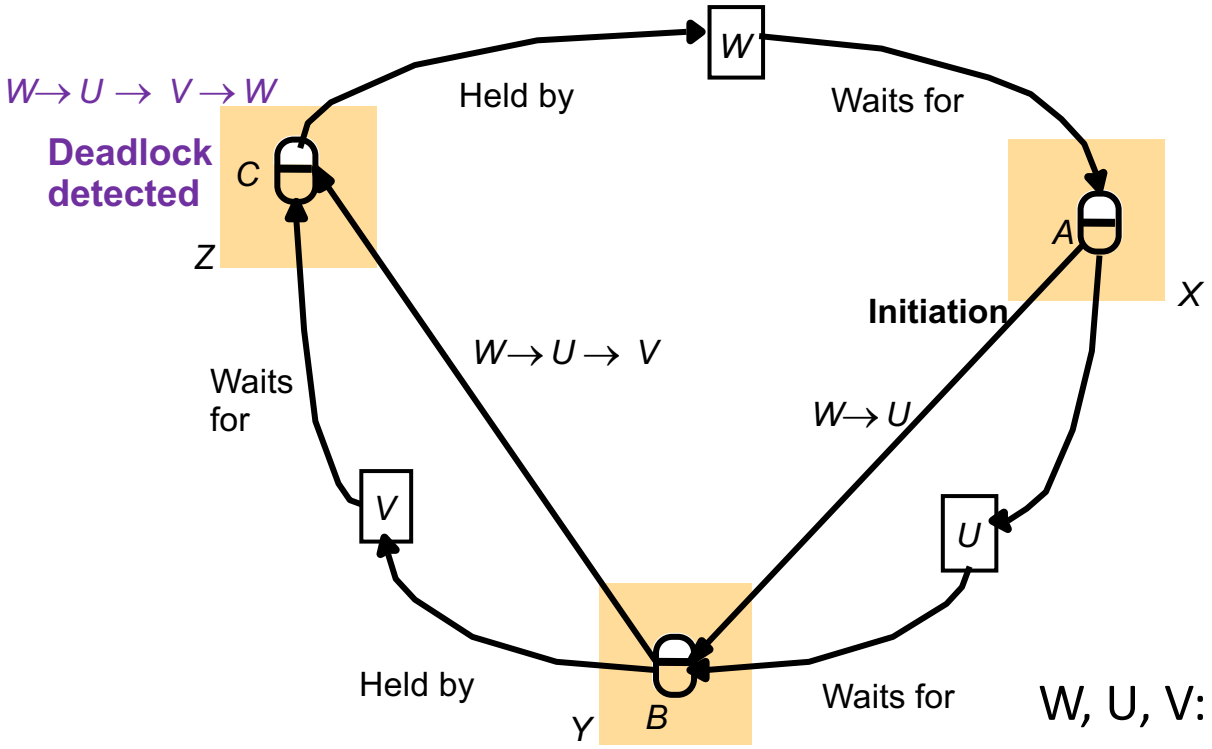


- Y adds another edge, and forwards the probe to the next server.

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.

Distributed Transactions

- *Sharding:* objects can be distributed across multiple (1000's of) servers
  - what we have been discussing so far.
  - Primary reason: load balancing and scalability.

- *Replication:* the same object may be replicated among a handful of nodes.
  - Primary reason: fault-tolerance, availability, durability.

Replication: Natural way to handle failures

- Node failures are common.

In each cluster's first year, it's typical that 1,000 individual machine failures will occur; thousands of hard drive failures will occur; one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours; 20 racks will fail, each time causing 40 to 80 machines to vanish from the network; 5 racks will "go wonky," with half their network packets missing in action; and the cluster will have to be rewired once, affecting 5 percent of the machines at any given moment over a 2-day span. And there's about a 50 percent chance that the cluster will overheat, taking down most of the servers in less than 5 minutes and taking 1 to 2 days to recover.

-- Jeff Dean (Google), source: cnet.com

Replication: Natural way to handle failures

- Node failures are common.
- What could happen if a node fails?
  - Objects unavailable until recovery.
  - 2PC "stuck" after coordinator failure
- Even worse: what happens if the drive failures.
  - no recovery!
- Replication provides greater availability and robustness to failures.
  - Geo-replication (spanning datacenters across the world) for greater robustness.

Replication

- Replication = An object has identical copies, each maintained by a separate server.
  - Copies are called "replicas"

- With *k* replicas of each object, can tolerate failure of any (*k*-*1*) servers in the system

Replication: Availability

- If each server is down a fraction f of the time
  - Server's failure probability

- With no replication, availability of object
  = Probability that single copy is up
  = $(1 - f)$

- With k replicas, availability of object
  = Probability that at least one replicas is up
  = 1 – Probability that all replicas are down
  = $(1 - f^k)$

Replication: Availability

- With no replication, availability of object =
  = Probability that single copy is up
  = $(1 - f)$
- With k replicas, availability of object =
  Probability that at least one replicas is up
  = 1 – Probability that all replicas are down
  = $(1 - f^k)$

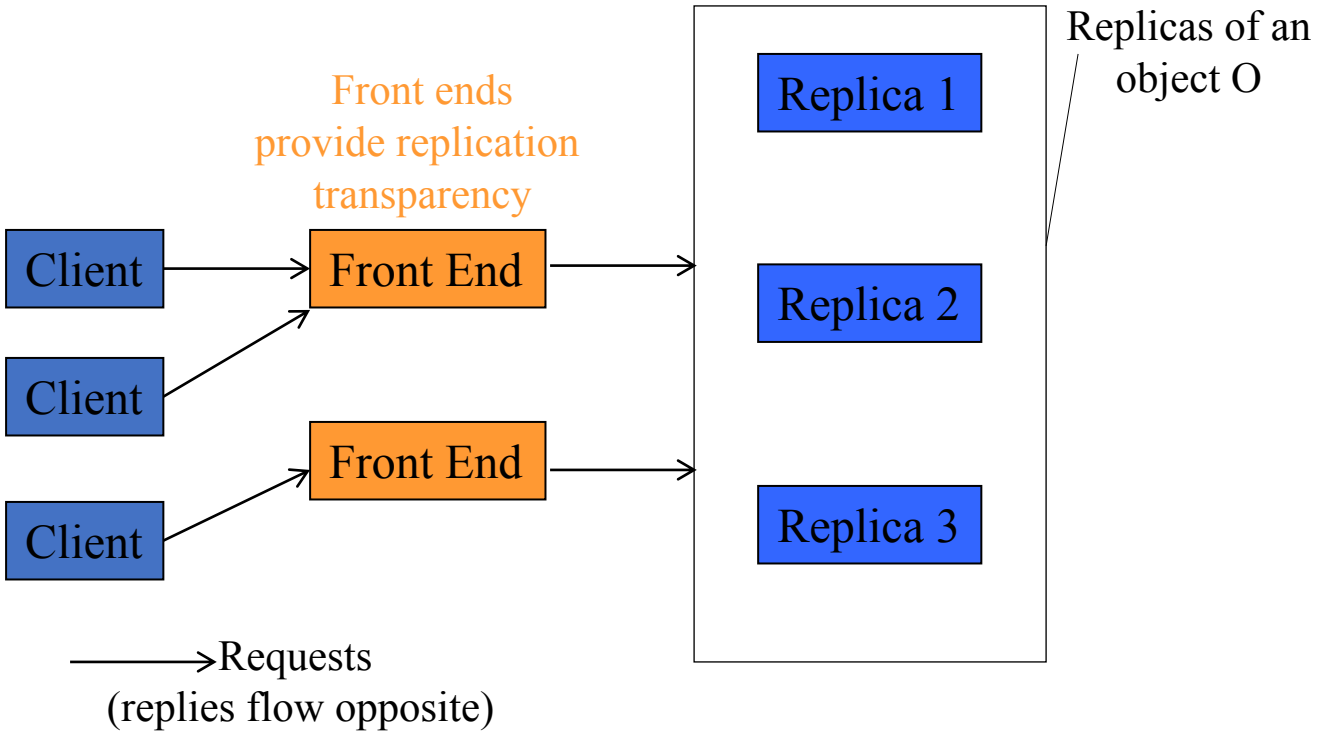
f=failure probability	No replication	*k*=3 replicas	*k*=5 replicas
0.1	90%	99.9%	99.999%
0.05	95%	99.9875%	6 Nines
0.01	99%	99.9999%	10 Nines

Replication: Challenges

1. Replication Transparency
   - A client ought not to be aware of multiple copies of objects existing on the server side

2. Replication Consistency
   - All clients see single consistent copy of data, in spite of replication
   - For transactions, guarantee ACID

Replication Transparency

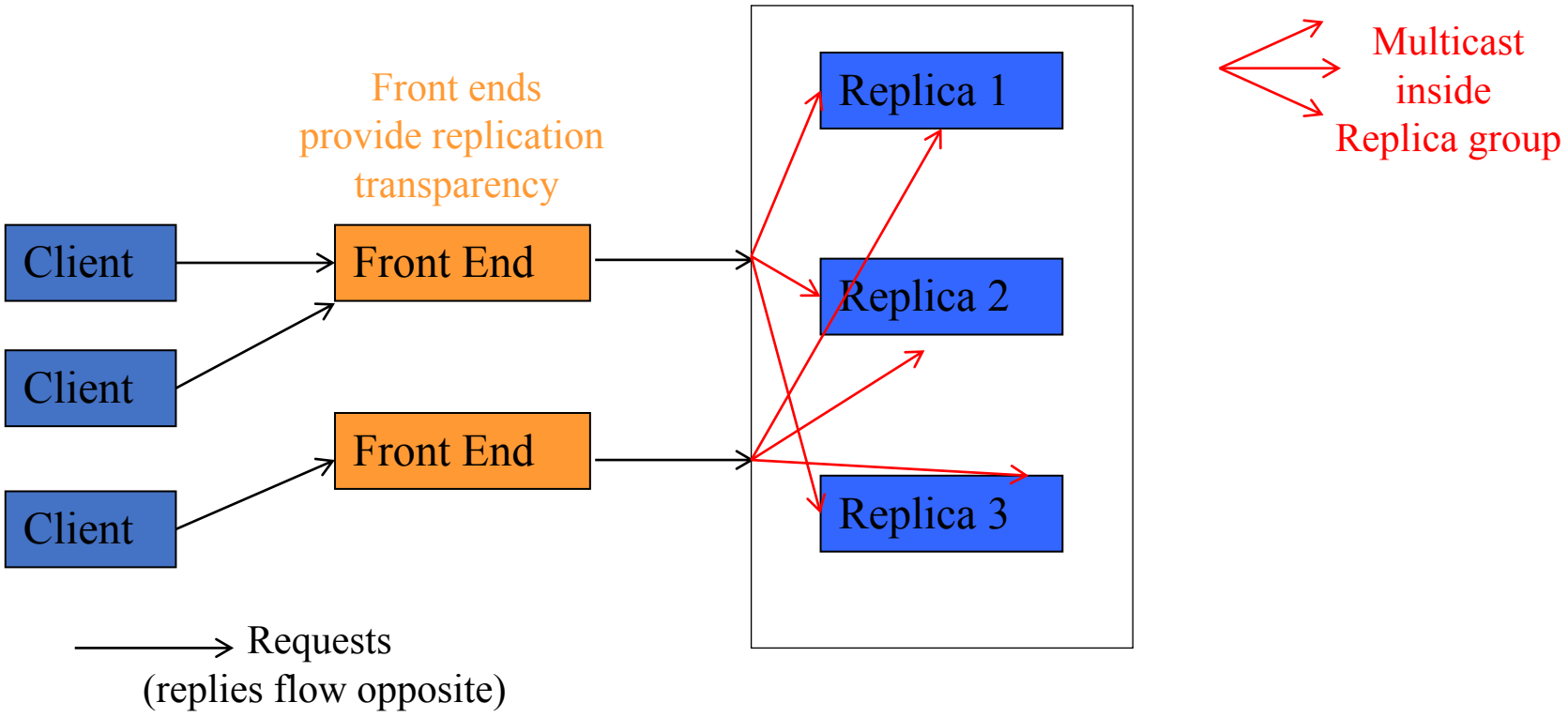


Replication Consistency

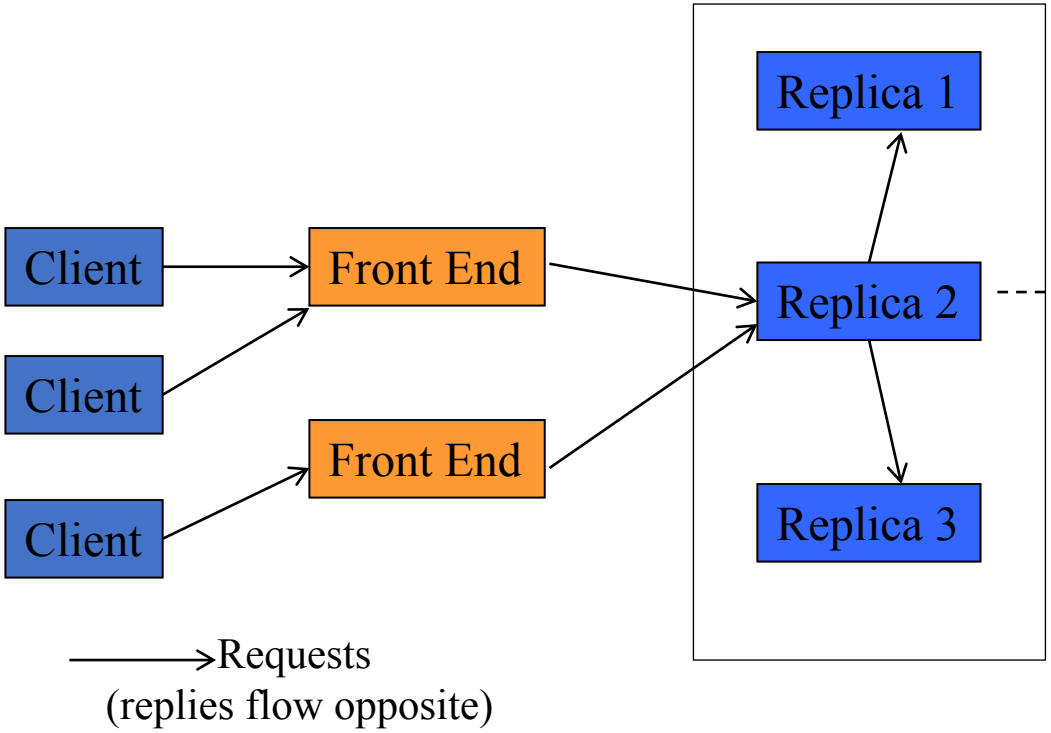
- Two ways to forward updates from front-ends (FEs) to replica group
  - Active Replication: treats all replicas identically
  - Passive Replication: uses a primary replica (leader)

- Both approaches use the concept of "Replicated State Machines"
  - Each replica's code runs the same state machine
  - *Multiple copies of the same State Machine begun in the Start state, and receiving the same Inputs in the same order will arrive at the same State having generated the same Outputs.* [Schneider 1990]

Active Replication



Passive Replication



- Leader => total reliable ordering of all updates
- On leader failure, run election

Transactions and Replication

- One-copy serializability
  - *A concurrent execution of transactions in a replicated database is one-copy-serializable if it is equivalent to a serial execution of these transactions over a single logical copy of the database.*
  - (Or) The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at a time on a single set of objects (i.e., 1 replica per object).
- In a non-replicated system, transactions appear to be performed one at a time in some order.
  - Correctness means serial equivalence of transactions
- When objects are replicated, transaction systems for correctness need one-copy serializability.

Transactions and Replication

- Objects distributed among 1000's cluster nodes for load-balancing (sharding)
- Objects replicated among a handful of nodes for availability / durability.
  - Replication across data centers, too
- Two-level operation:
  - Use transactions, coordinators, 2PC per object
  - Use Paxos / Raft among object replicas
- Consensus needed across object replicas, e.g.
  - When acquiring locks and executing operations
  - When committing transactions

2PC and Paxos

- E.g. workflow:
  - Coordinator leader sends Prepare message to leaders of each replica group
  - Each replica leader uses Paxos to commit the Prepare to the group logs
  - Once commit prepare succeeds, reply to coordinator leader
  - Coordinator leader uses Paxos to commit decision to its group log.
  - Coordinator leader sends Commit message to leaders of each replica group.
  - Each replica leader uses Paxos to process the final commit.
  - Replica leader send the "commit ok / have committed" message back to coordinator.