CS 425 / ECE 428 Distributed Systems Fall 2015

Indranil Gupta (Indy)

Oct 22, 2015

Lecture 18: Replication Control

Server-side Focus

- Concurrency Control = how to coordinate multiple concurrent clients executing operations (or transactions) with a server

Next:

- Replication Control = how to handle operations (or transactions) when there are objects are stored at multiple servers, with or without replication

Replication: What and Why

- Replication = An object has identical copies, each maintained by a separate server
  - Copies are called "replicas"
- Why replication?
  - Fault-tolerance: With *k* replicas of each object, can tolerate failure of any (*k-1*) servers in the system
  - Load balancing: Spread read/write operations out over the *k* replicas => load lowered by a factor of *k* compared to a single replica
  - Replication => Higher Availability

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

  = $(1 - f^k)$

Nines Availability

- With no replication, availability of object =
  $= (1 - f)$
- With k replicas, availability of object =
  $= (1 - f^k)$

Availability Table

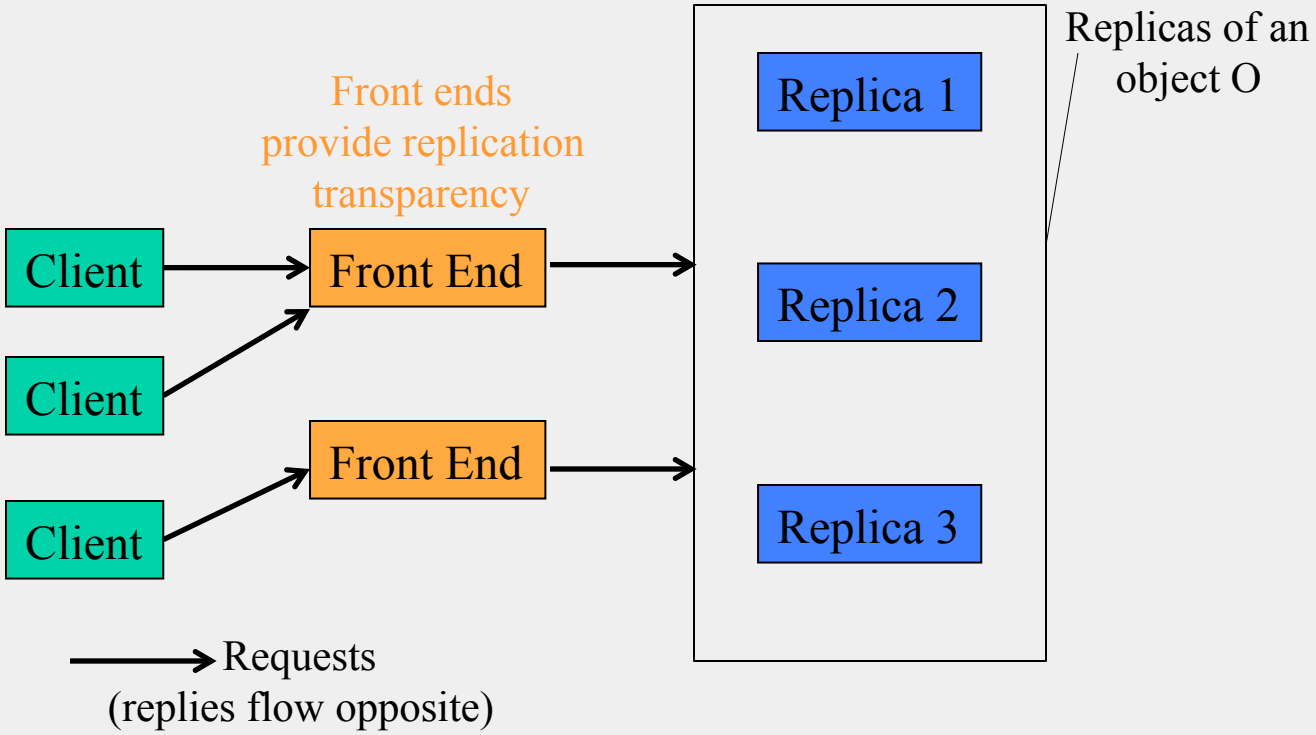
f=failure probability	No replication	k=3 replicas	k=5 replicas
0.1	90%	99.9%	99.999%
0.05	95%	99.9875%	6 Nines
0.01	99%	99.9999%	10 Nines

What's the Catch?

- Challenge is to maintain two properties

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

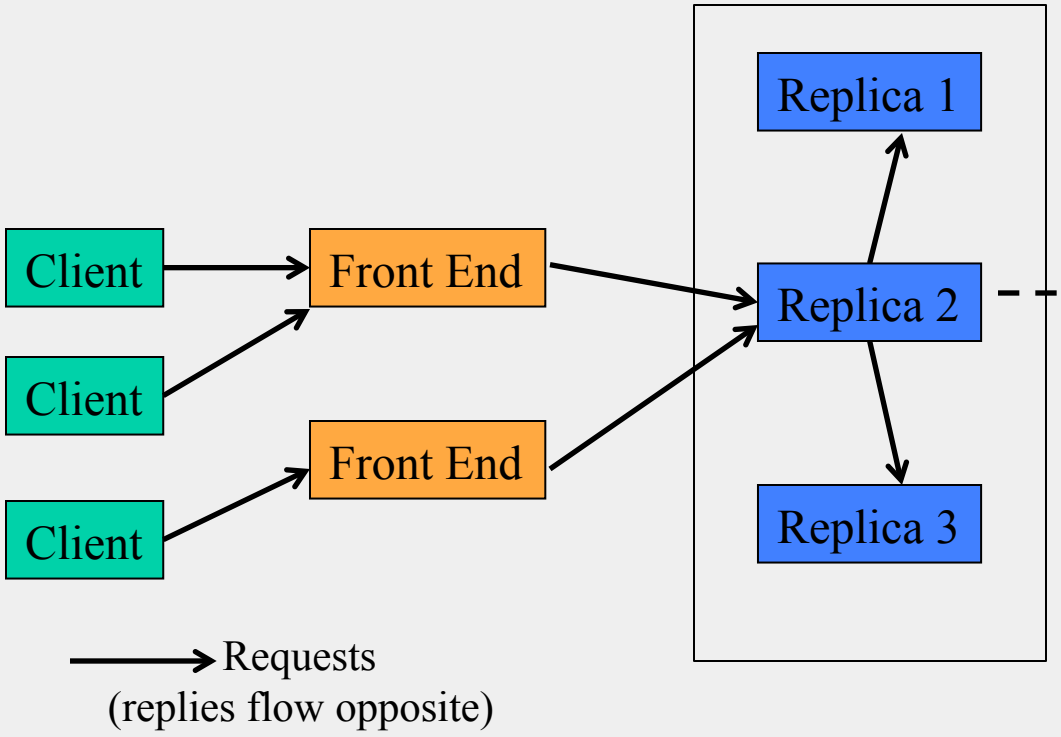
Replication Transparency



Replication Consistency

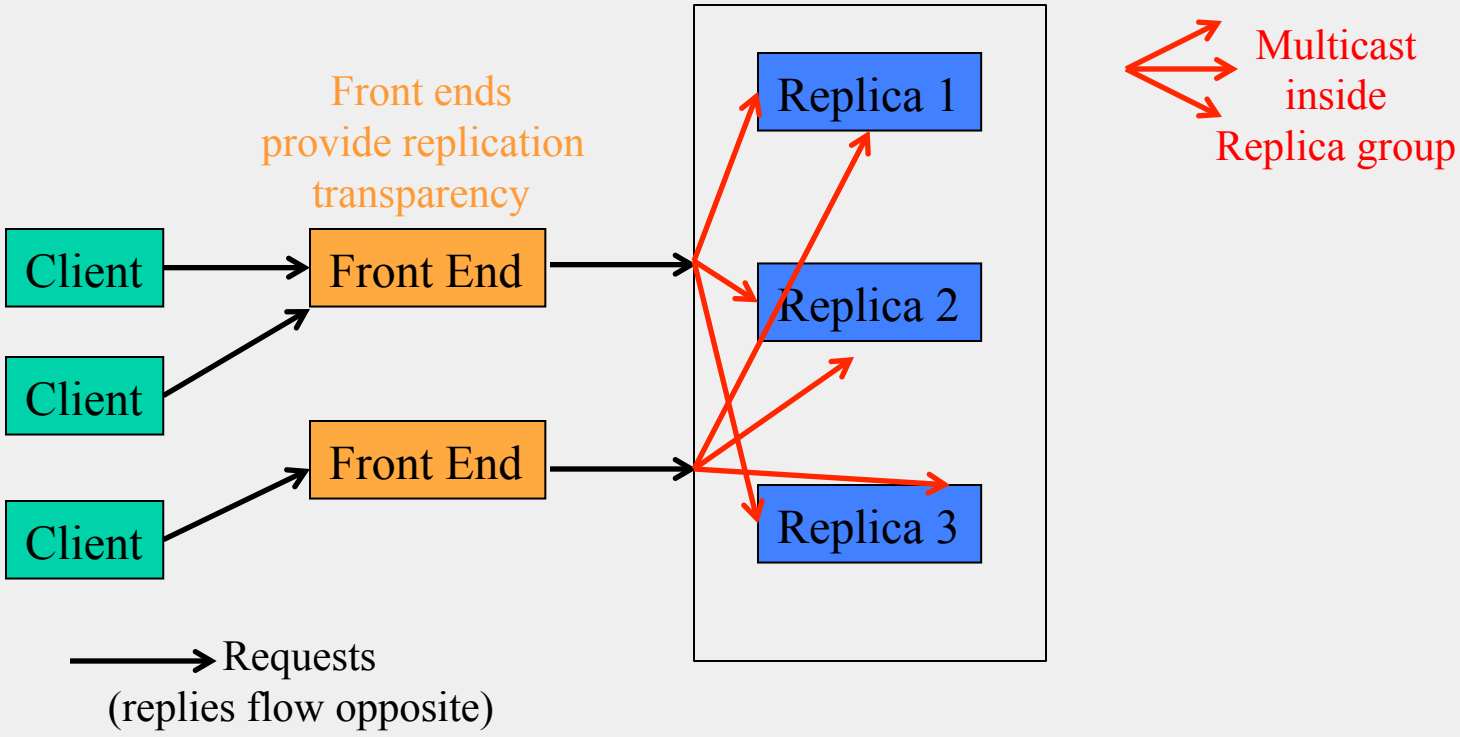
- Two ways to forward updates from front-ends (FEs) to replica group
  - Passive Replication: uses a primary replica (master)
  - Active Replication: treats all replicas identically
- Both approaches use the concept of "Replicated State Machines"
  - Each replica's code runs the same state machine
  - *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]

Passive Replication



- Master => total ordering of all updates
- On master failure, run election

Active Replication



Active Replication Using Concepts You've Learnt earlier

- Can use any flavor of multicast ordering, depending on application
  - FIFO ordering
  - Causal ordering
  - Total ordering
  - Hybrid ordering
- Total or Hybrid (*-Total) ordering + Replicated State machines approach
  - => all replicas reflect the same sequence of updates to the object

Active Replication Using Concepts You've Learnt earlier (2)

- What about failures?
  - Use virtual synchrony (i.e., view synchrony)
- Virtual synchrony with total ordering for multicasts =>
  - All replicas see all failures/joins/leaves and all multicasts in the same order
  - Could also use causal (or even FIFO) ordering if application can tolerate it

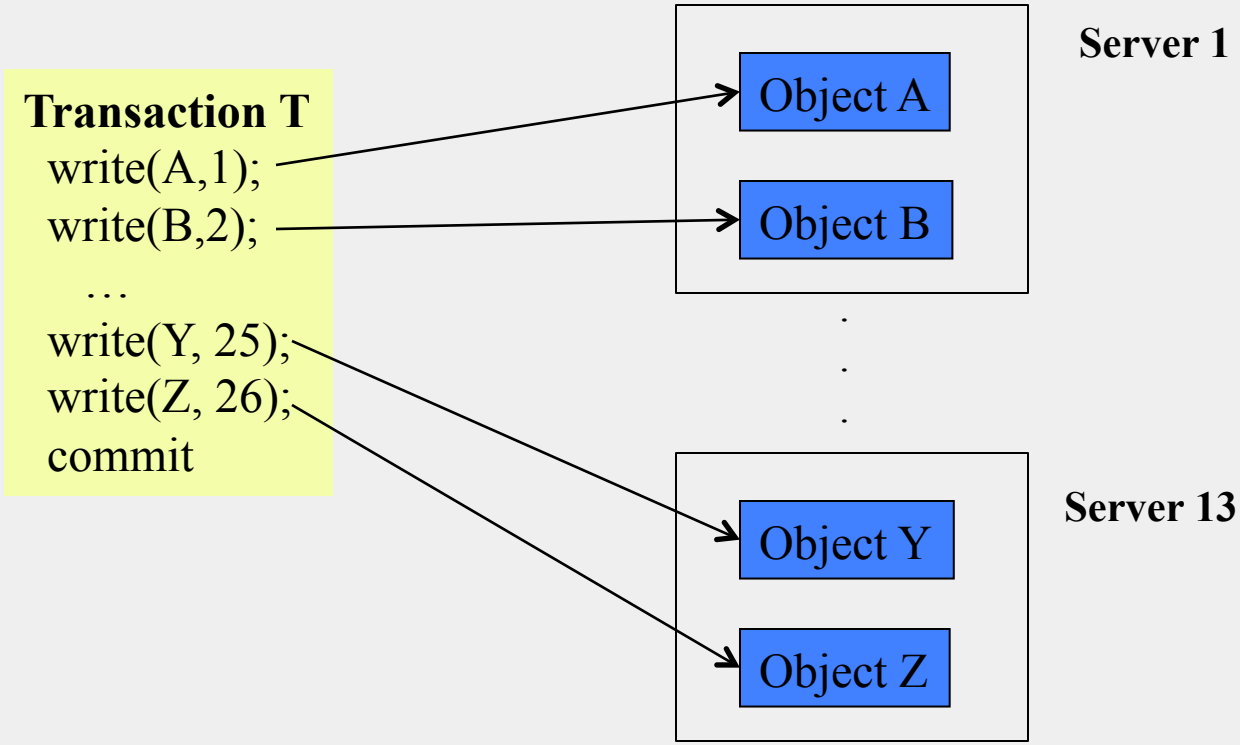
Transactions and Replication

- 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

Next

- Committing transactions with distributed servers

Transactions with Distributed Servers



Transactions With Distributed Servers

- Transaction T may touch objects that reside on different servers
- 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?

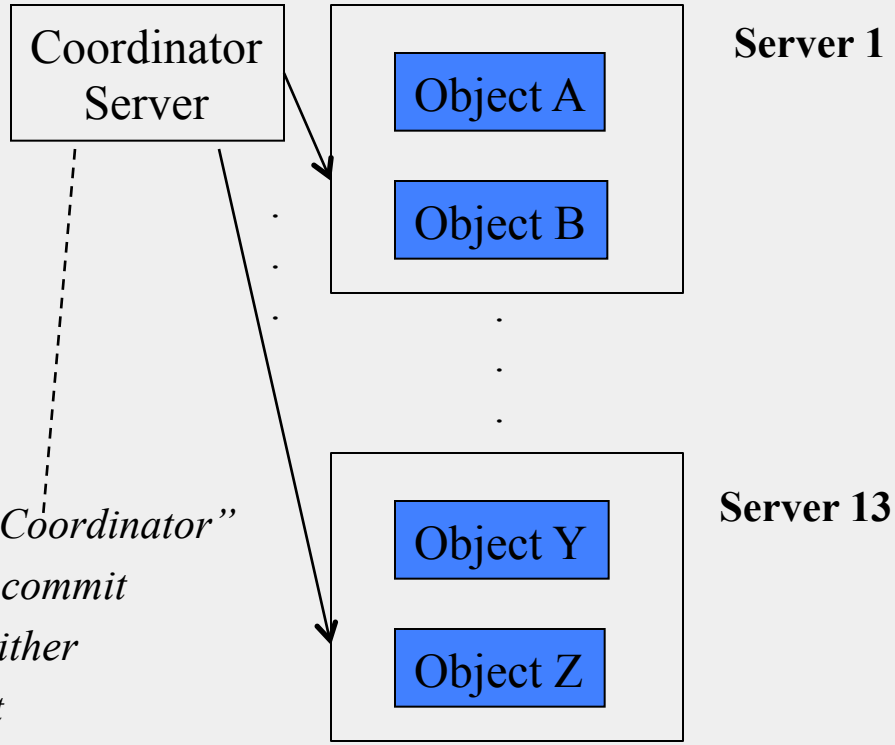
Transactions With Distributed Servers

- Transaction T may touch objects that reside on different servers
- 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")

One-phase Commit

Transaction T

write(A,1);
write(B,2);
…
write(Y, 25);
write(Z, 26);
commit

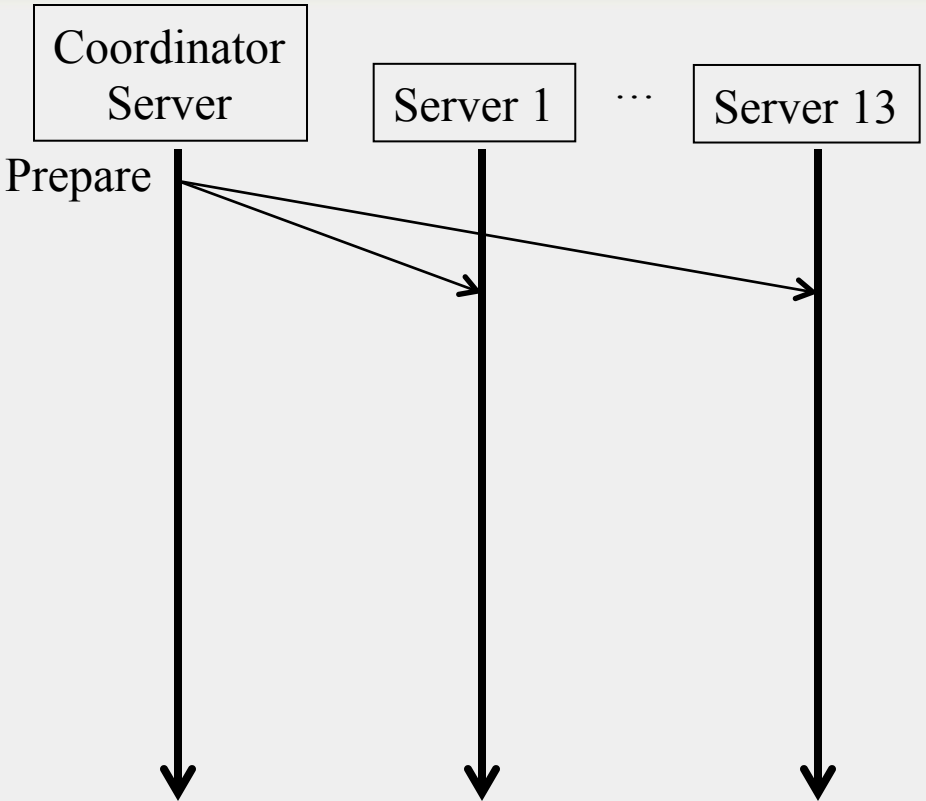


- *Special server called "Coordinator" initiates atomic commit*
- *Tells other servers to either commit or abort*

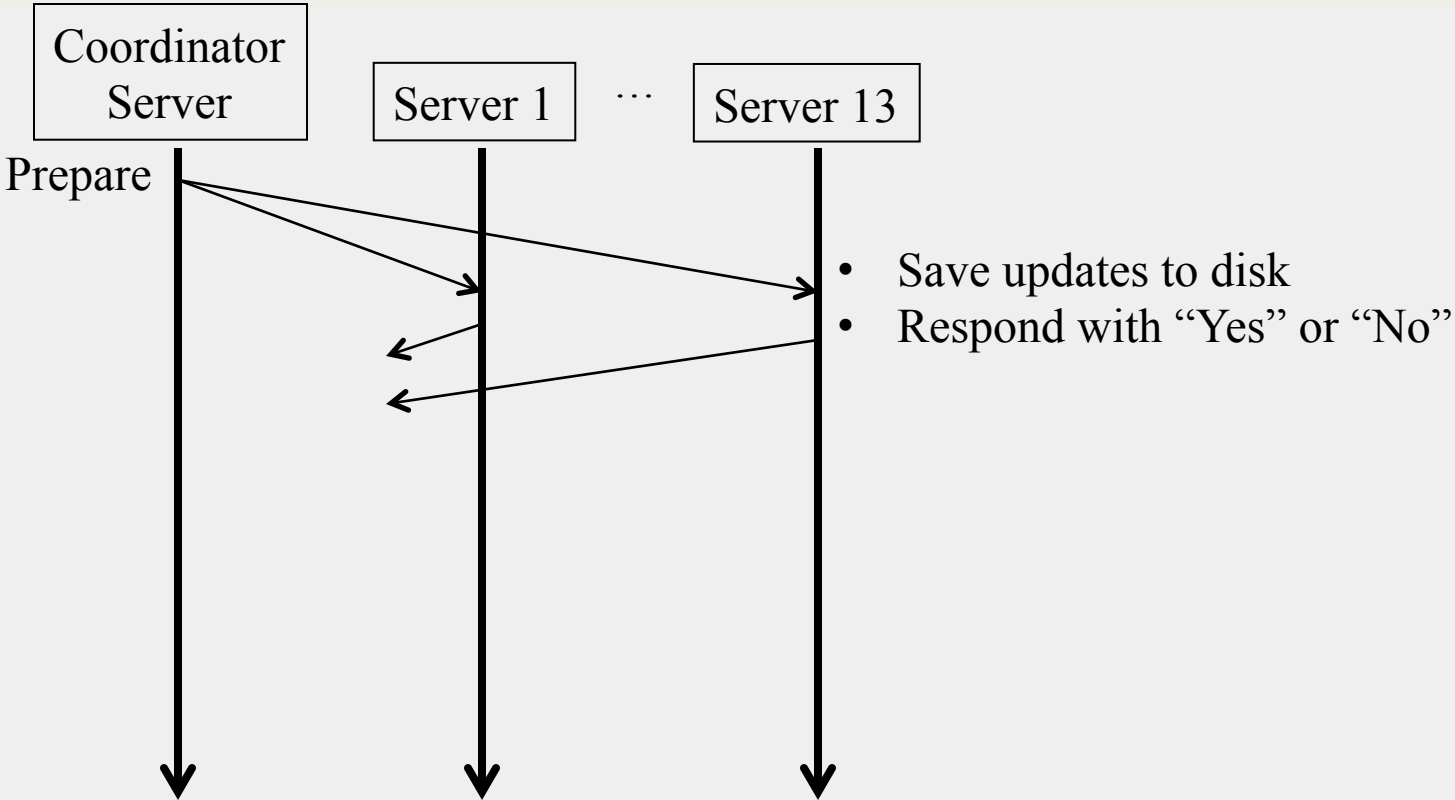
One-phase Commit: Issues

- Server with object has no say in whether transaction commits or aborts
  - If object corrupted, it just cannot commit (while other servers have committed)
- Server may crash before receiving commit message, with some updates still in memory

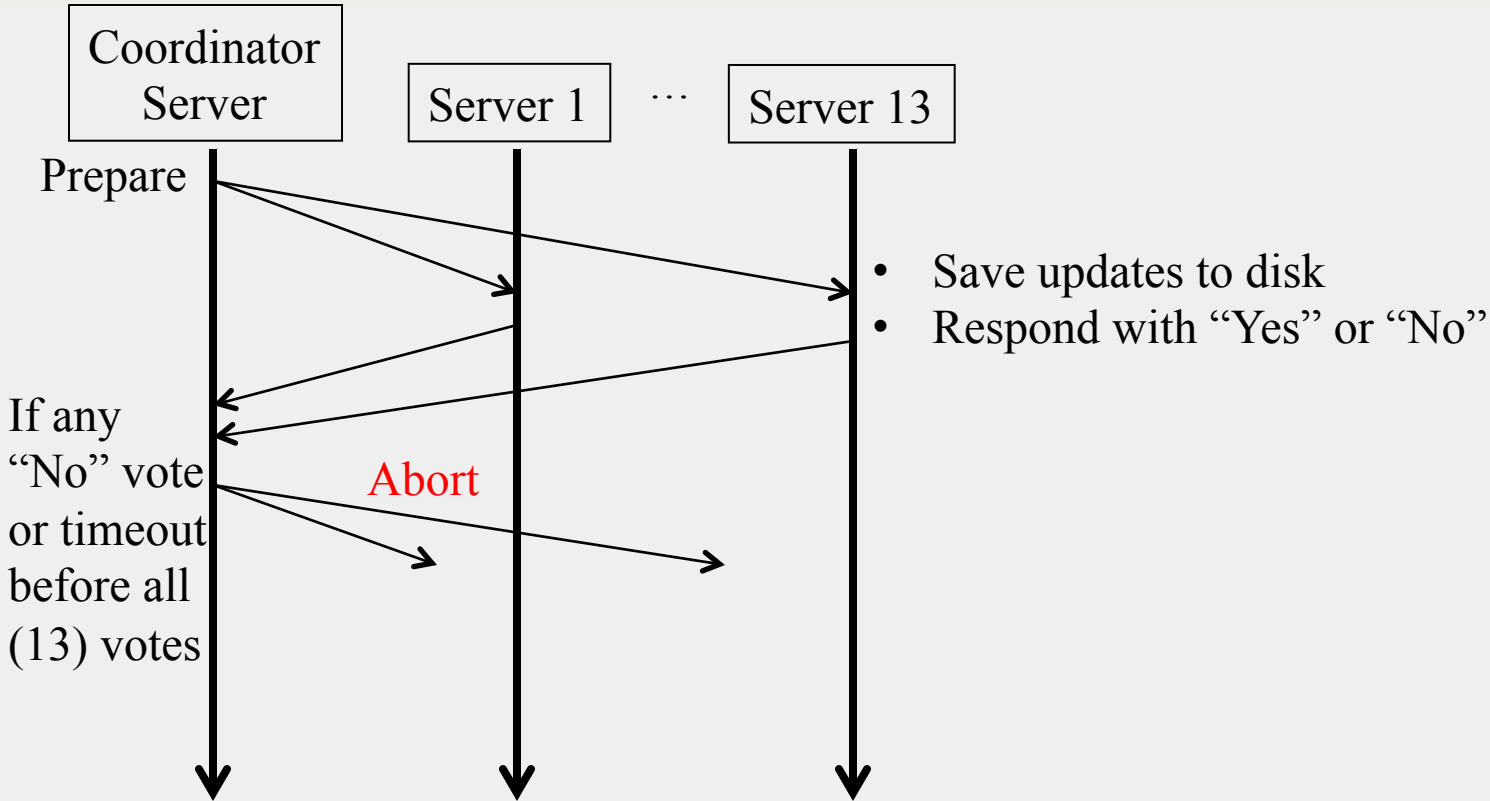
Two-phase Commit



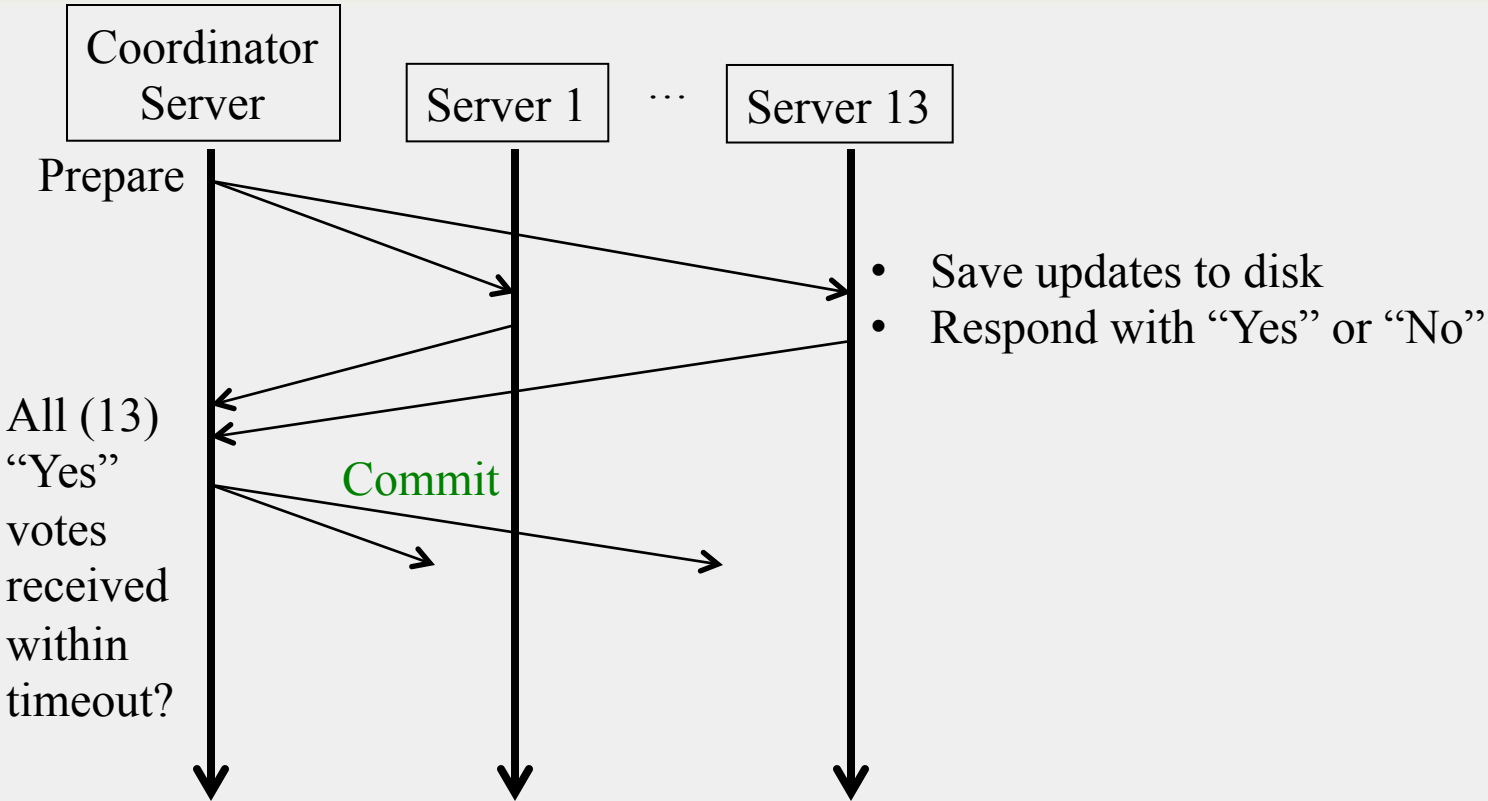
Two-phase Commit



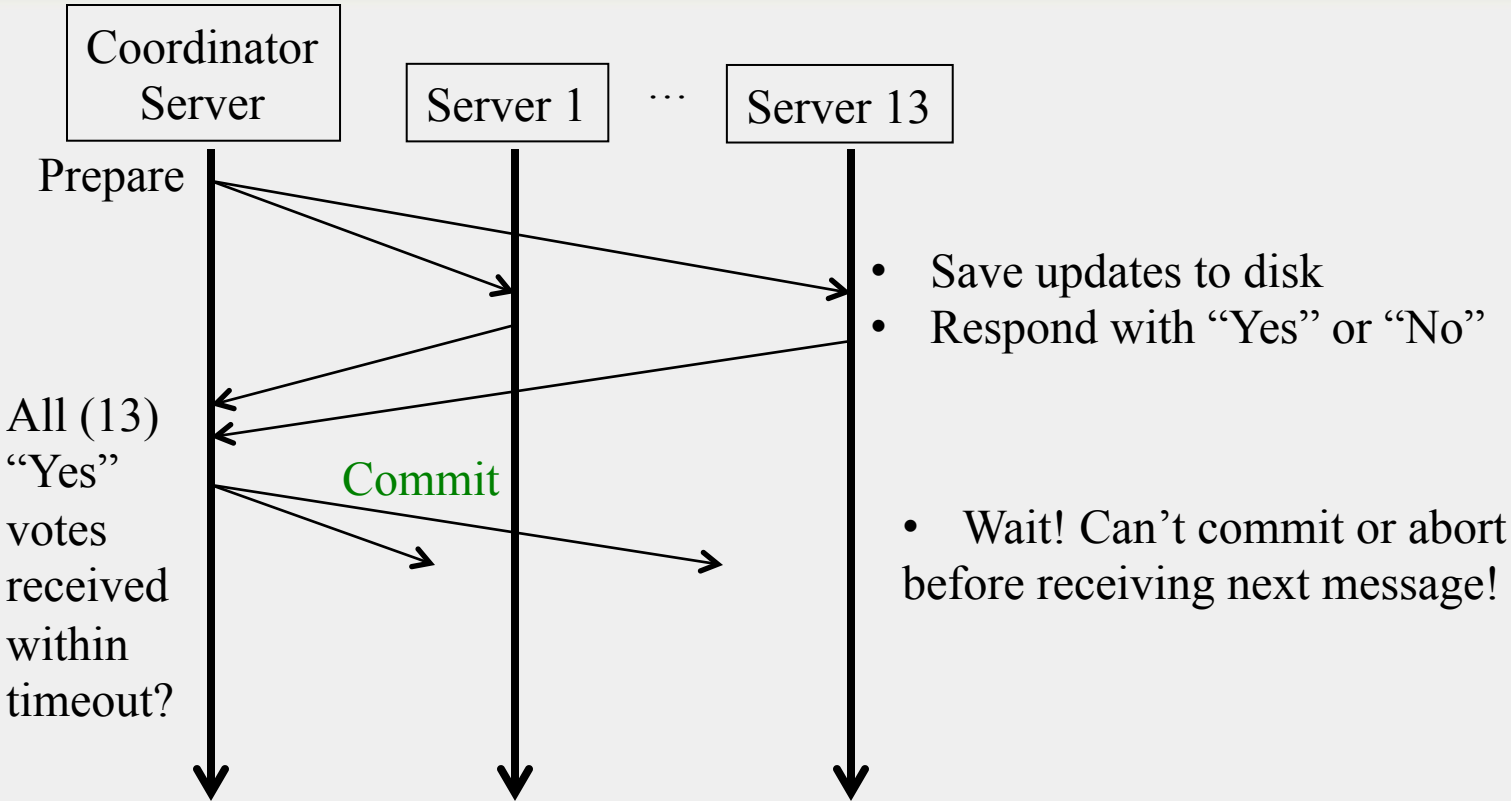
Two-phase Commit



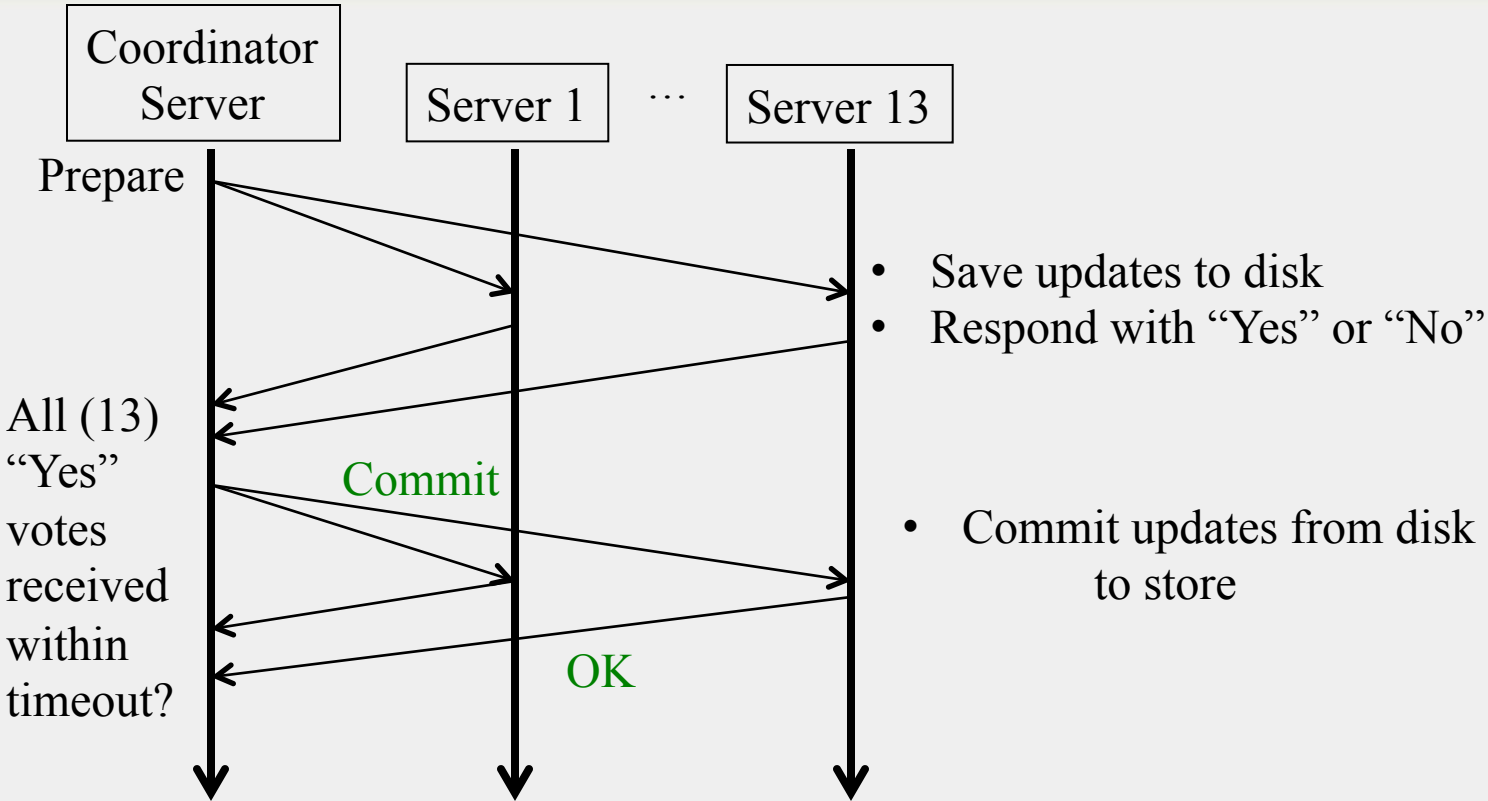
Two-phase Commit



Two-phase Commit



Two-phase Commit



Failures in Two-phase Commit

- If server voted Yes, it cannot commit unilaterally before receiving Commit message
- If server voted No, can abort right away (why?)
- 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 or new election => new coordinator takes over

Failures in Two-phase Commit (2)

- To deal with Prepare message loss
  - The server may decide to abort unilaterally after a timeout for 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)

Using Paxos in Distributed Servers

Atomic Commit

- Can instead use Paxos to decide whether to commit a transaction or not
- But need to ensure that if any server votes No, everyone aborts

Ordering updates

- Paxos can also be used by replica group (for an object) to order all updates – iteratively do:
  - Server proposes message for next sequence number
  - Group reaches consensus (or not)

Summary

- Multiple servers in cloud
  - Replication for Fault-tolerance
  - Load balancing across objects
- Replication Flavors using concepts we learnt earlier
  - Active replication
  - Passive replication
- Transactions and distributed servers
  - Two phase commit