Distributed Systems

CS425/ECE428

April 5 2022

Instructor: Radhika Mittal

Acknowledgements for the materials: Indy Gupta, Nikita Borisov, Spanner authors

Logistics

- MP2 is due today.
- MP3 will be released on Thursday

Distributed Transactions

- Transaction processing can be *distributed* across multiple servers.
    - Different objects can be stored on different servers.
        - How to achieve atomicity and isolation in this setting?
        - We discussed two-phase commit for atomicity
        - First on our agenda for today: achieving isolation and handling deadlocks.
    - An object may be replicated across multiple servers.
        - Next on our agenda for today
- Case study: Google's Spanner System

Distributed Transaction Challenges

System Model: Different objects can be stored on different servers.

- Atomic: all-or-nothing
  - *Must ensure atomicity across servers.*
  - Achieved via two-phase commit
- 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 protocol.

- How to handle deadlocks?

Deadlock Detection in Distributed Transactions

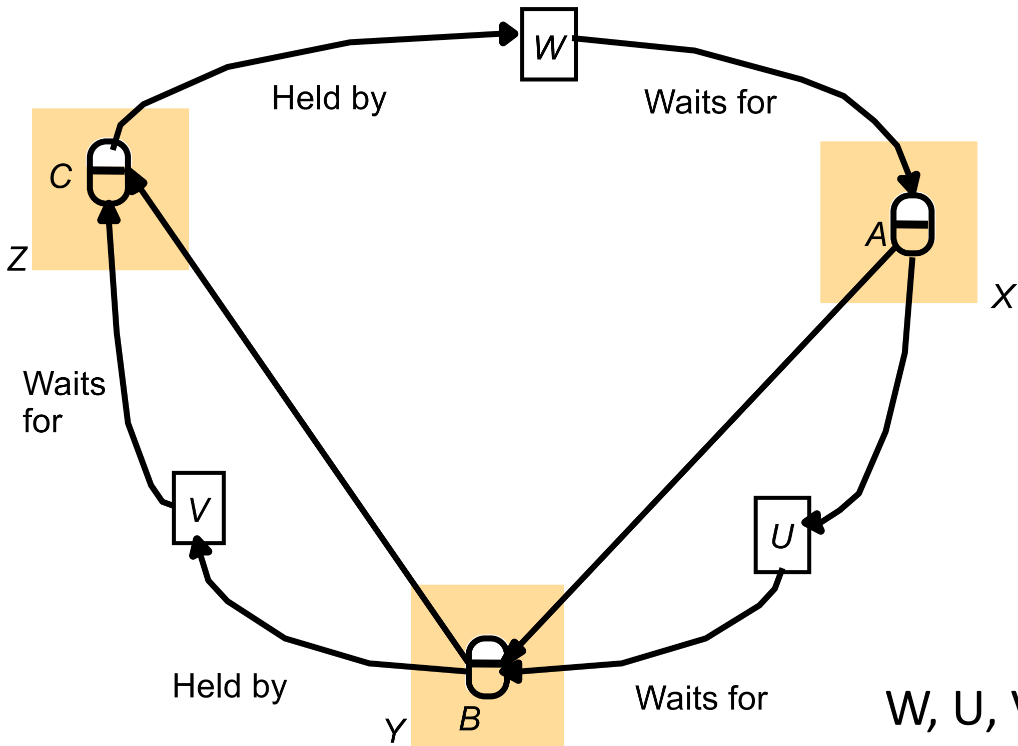
- The wait-for graph in a distributed set of transactions is distributed.

- 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 with the central server fails).
  - Scalability.

Decentralized Deadlock Detection

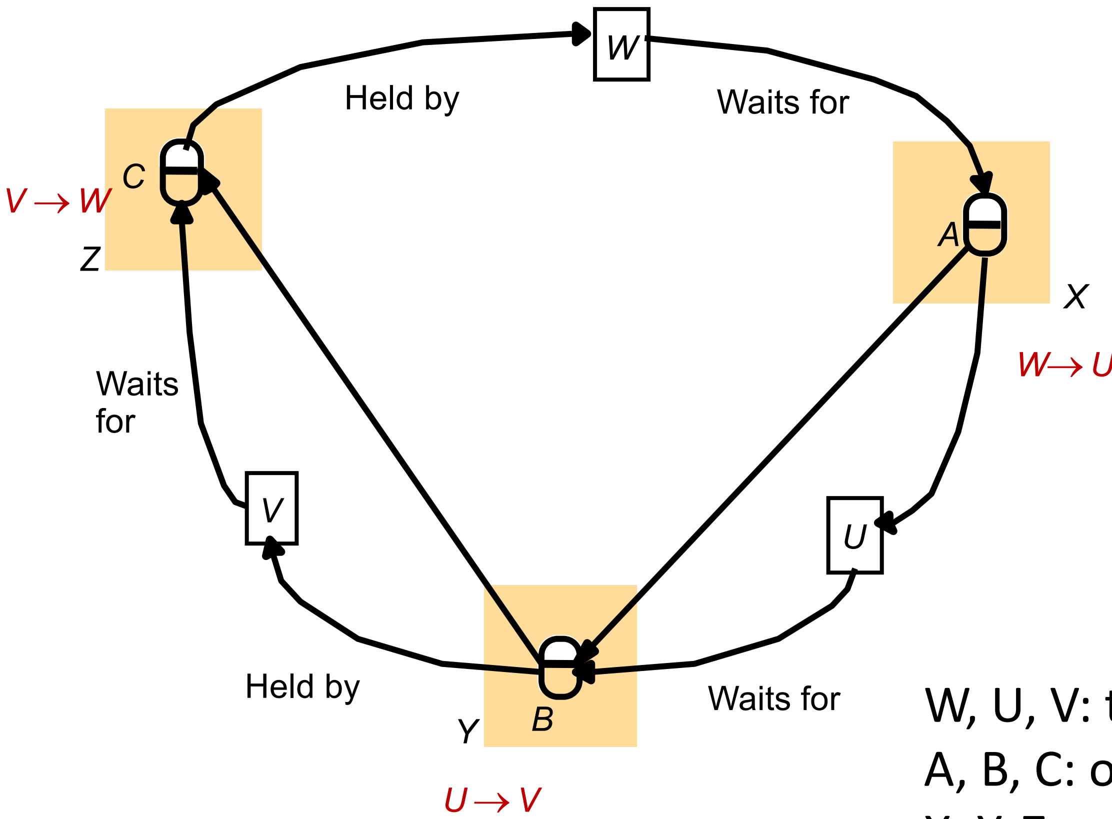
- **Edge chasing:** Forward "probe" messages to servers in the edges of wait-for graph, pushing the graph forward, until cycle is found.



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.



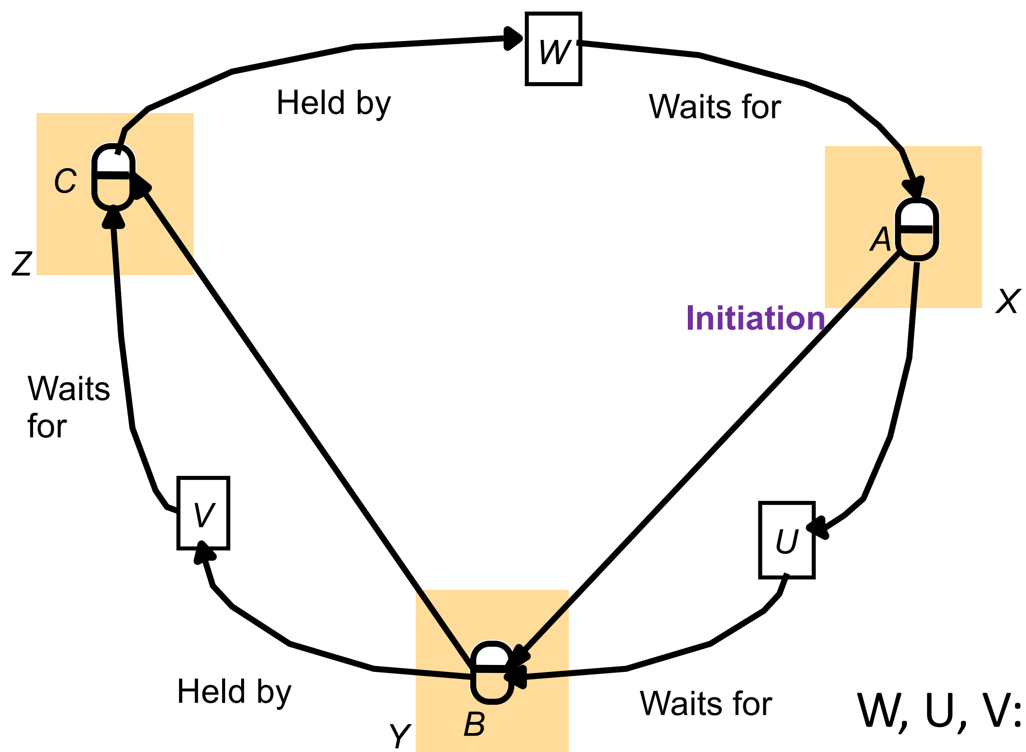
All servers know local wait-for relationships.

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.

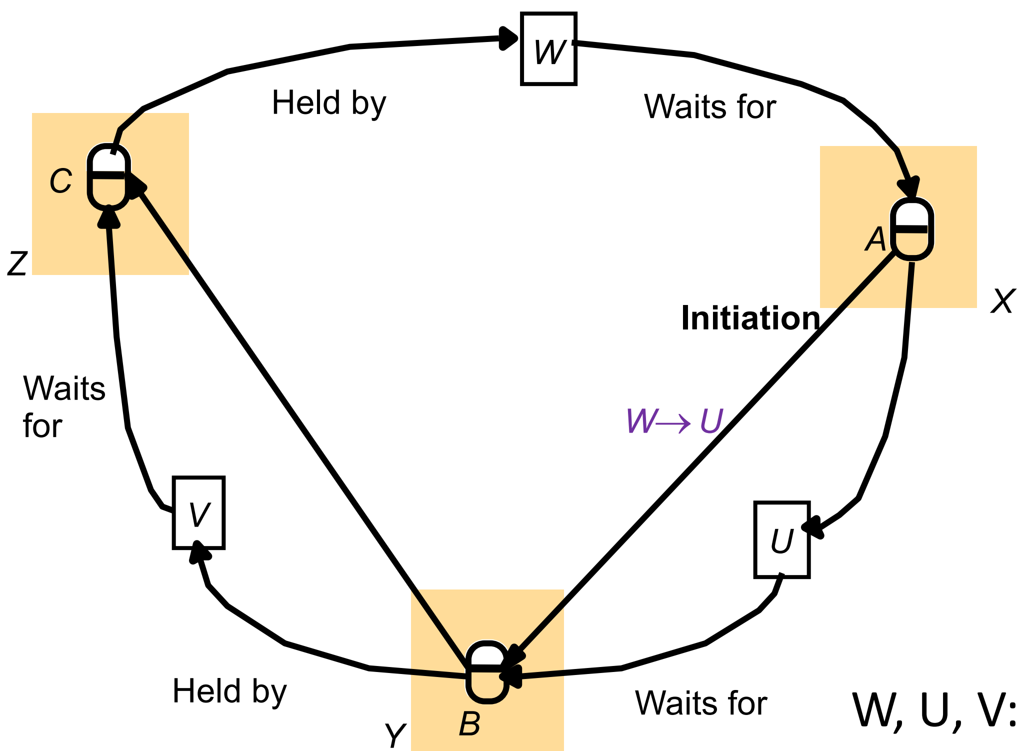


- Server X realizes W is waiting on U (a potential edge in the wait-for graph).
- Ask U's coordinator whether U is waiting on anything, and at which server.

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.

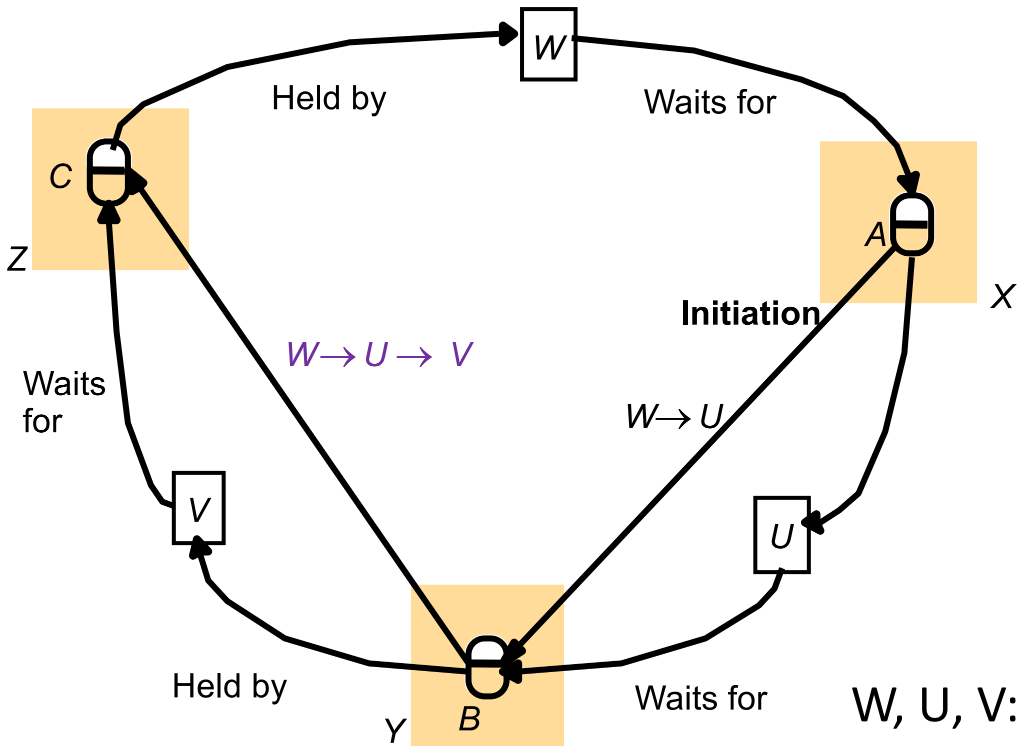


- Server X realizes W is waiting on U (a potential edge in the wait-for graph).
- Ask U's coordinator whether U is waiting on anything, and at which server.
- *Send a probe to the next server.*

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.

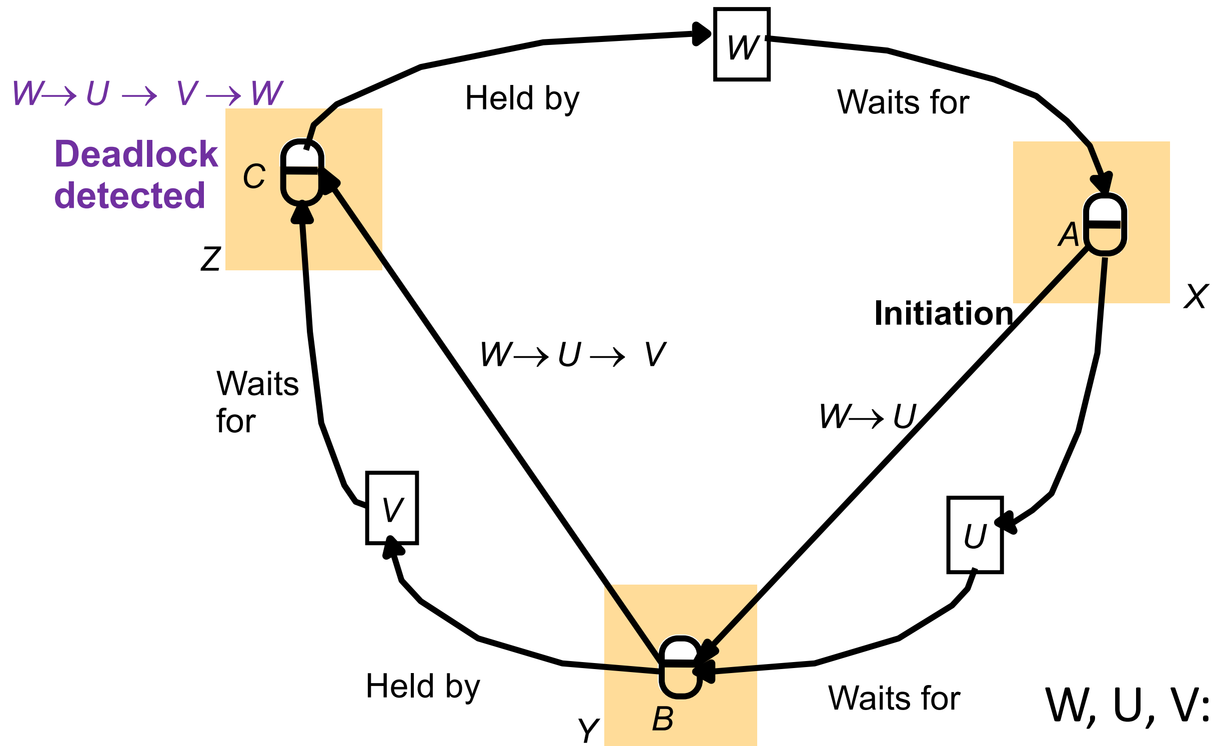


- Y adds another edge, and forwards the probe to the next server.

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.



- C 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

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

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
  = $(1 - f^k)$

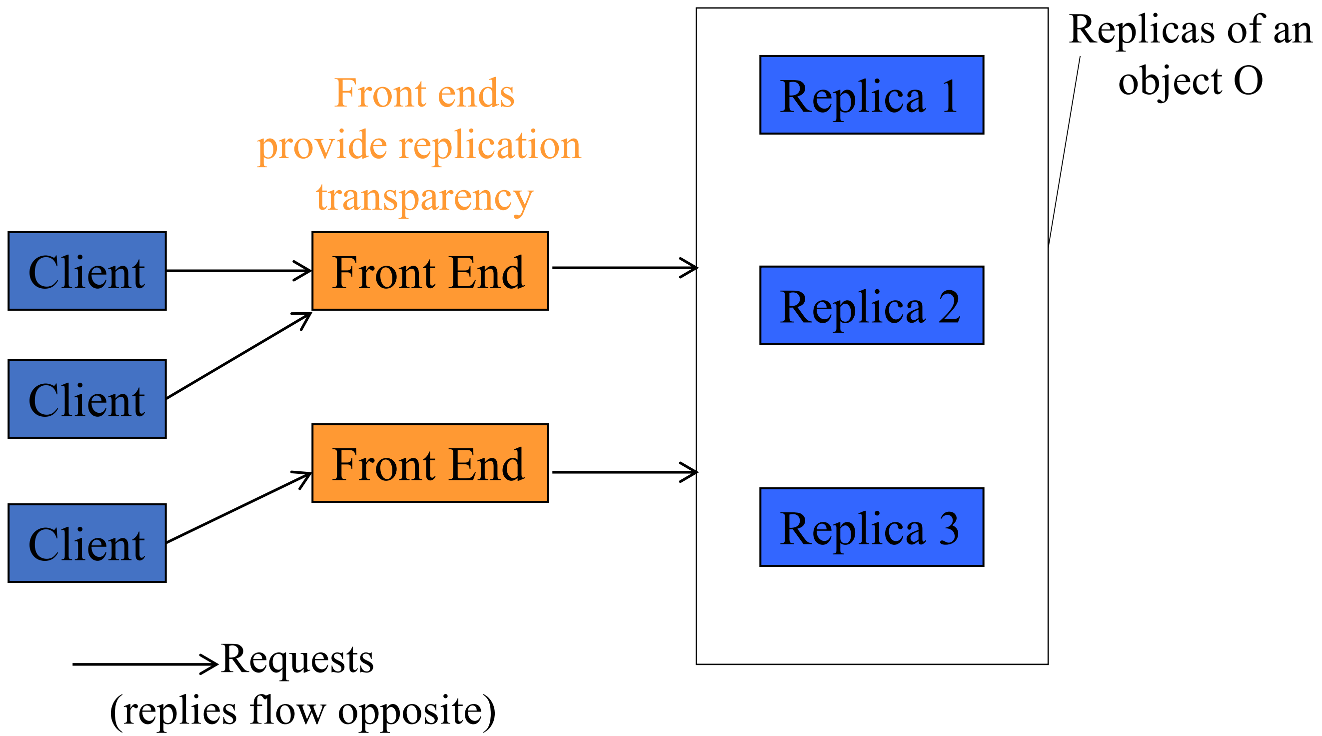
***f*=failure probability**	**No replication**	***k*=3 replicas**	***k*=5 replicas**
0.1	90%	99.9%	99.999%
0.05	95%	99.9875%	6 Nines
0.01	99%	99.9999%	10 Nines

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

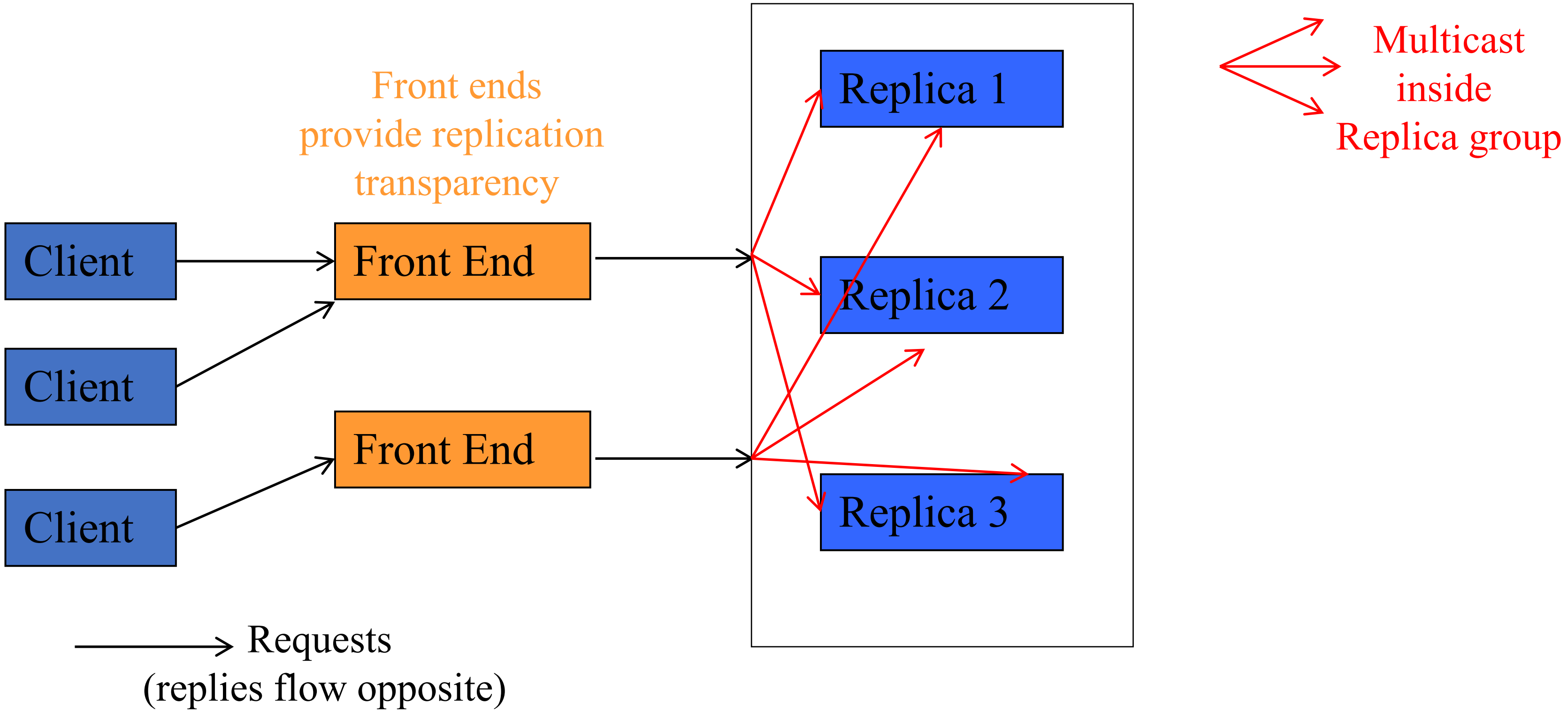


Replication Consistency

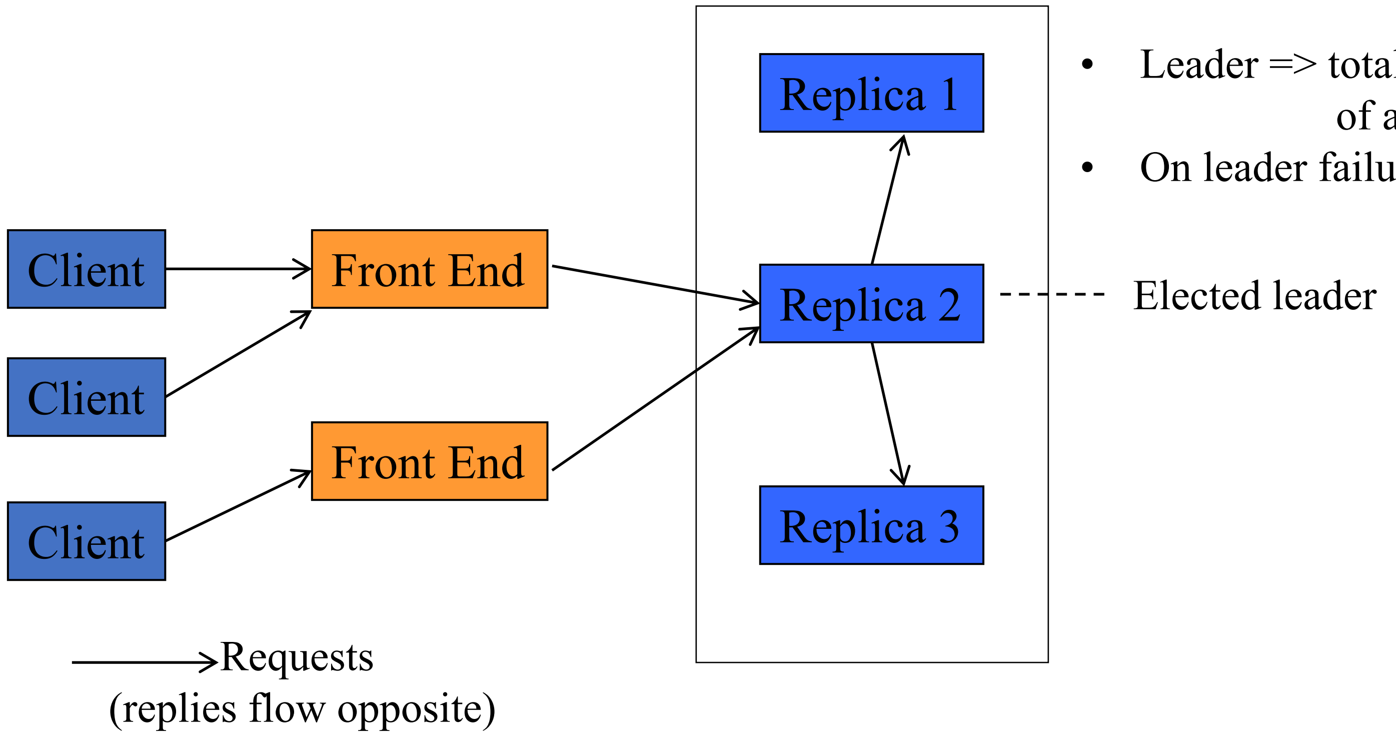
- Two ways to forward updates from front-ends (FEs) to replica group
  - Active Replication: treats all replicas identically
  - Passive Replication: uses a primary replica (leader)

- 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.