CS 162: Operating Systems and Systems Programming

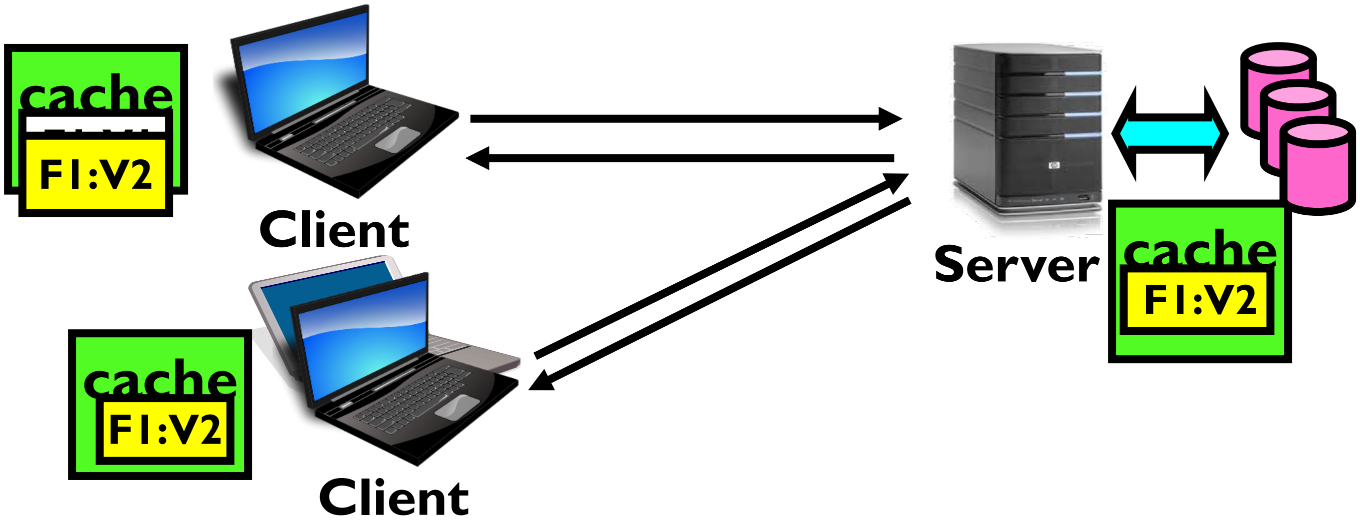
Lecture 21: Distributed Key-Value Store & 2-Phase Commit

November 12, 2019

Instructor: David E. Culler

https://cs162.eecs.berkeley.edu

Distributed File Systems



- VFS (Virtual File System) interface allows client "front end" to carry out protocol with server "back end"
  - E.g., NFS RPC, stateless, open, read/write close
- Unit of access: NFS blocks, AFS whole file
- Client caching and consistency management
  - NFS eventual consistency, AFS last writer wins, notification on close
  - Enforce single-write, multiple reader discipline
- Handling failures

Recall: Distributed

- **Transparent access to files located on remote disks**
  - Caching for performance
  - Blocks or whole files
  - Introduces consistency issues
    - » File save vs write
    - » Remote vs actively shared
  - NFS: Check periodically for changes to server copy
  - AFS: Server notifies client of changes

The Promise of Dist. Systems

- **Availability: One machine goes down, overall system stays up**
- **Durability: One machine loses data, but *system* does not lose anything**
- **Security: Easier to secure each component of the system individually?**

Dist. System – the darker side

- **Availability: Failure in one machine causes others to hang waiting for it**
  - **Two sides of Fate sharing**
- **Durability: Lots of machines that might lose your data**
- **Security: More components means more points of attack**

- **Engineering of distributed systems – both cloud and end hosts – are fundamentally more reliable than in the 80's and 90's when the approach emerged**

Sharing Data, rather than Files ?

- Key:Value stores are used everywhere
- Native in many programming languages
  - Associative Arrays in Perl
  - Dictionaries in Python
  - Maps in Go
  - …
- What about a collaborative key-value store rather than message passing or file sharing?
  - An alternative basis for building distributed systems
  - Especially distributed within the cloud
- Can we make it scalable and reliable?

Key Value Storage

Simple interface

- `put(key, value); // Insert/write "value" associated with key`

- `get(key); // Retrieve/read value associated with key`

- Remember wordcount? Word:Count
  - AddCount(char *word, int count) over KV?

Why Key Value Storage?

- Easy to Scale
  - Handle huge volumes of data (e.g., petabytes)
  - Uniform items: distribute easily and roughly equally across many machines
- Relatively simple consistency properties
  - vs general database transactions or file system ops over multiple blocks
- Used as a simpler but more scalable "database"
  - Or as a building block for a more capable DB

Key Values: Examples

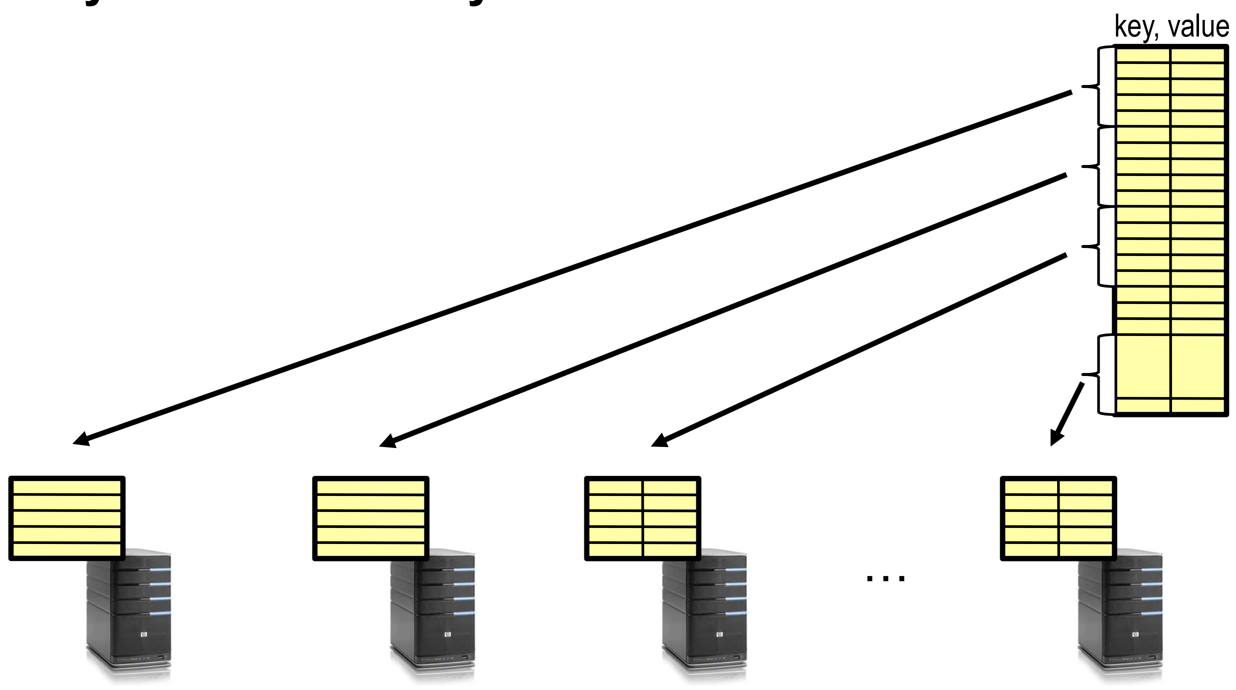
- **Amazon:**
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, ..)
- **Facebook, Twitter:**
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)
- **iCloud/iTunes:**
  - Key: Movie/song name
  - Value: Movie, Song

KV Storage Systems in the Wild

- Amazon
  - DynamoDB: internal key value store used to power Amazon.com (shopping cart)
  - Simple Storage System (S3)
- BigTable/HBase/Hypertable: distributed, scalable data storage
  - All the different services share distributed systems infrastructure
- Cassandra: "distributed data management system" (developed by Facebook)
- Memcached: in-memory key-value store for small chunks of arbitrary data (strings, objects)

Key Value Store

- **Also called Distributed Hash Tables (DHT)**
- **Main idea: partition set of key-value pairs across many machines**
  - **Easy to be a local key:value store**



Important Questions

- **put(key, value):**
  - **where do you store a new (key, value) tuple?**
- **get(key):**
  - **where is the value associated with a given "key" stored?**

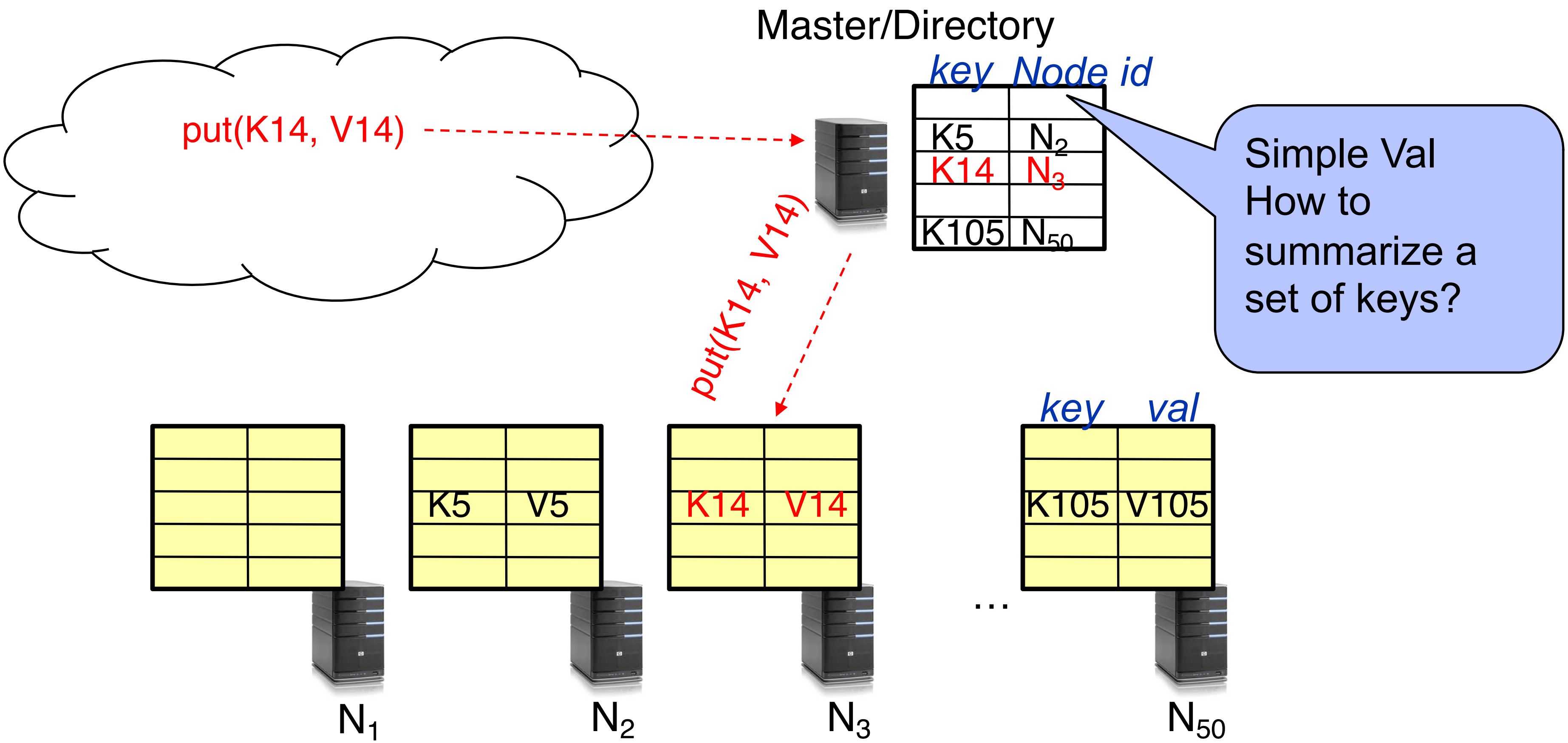
- **And, do the above while providing**
  - **Fault Tolerance**
  - **Scalability**
  - **Consistency**

How to solve the “where?”

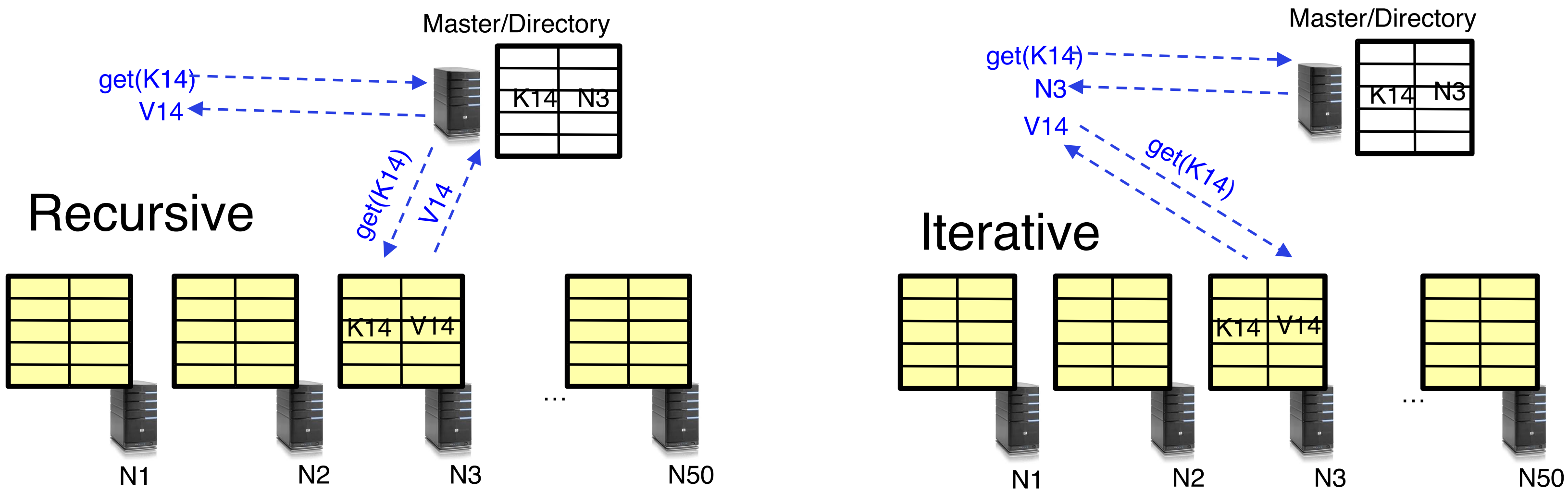
- Hashing
  - Given M nodes, all share a hash function: Key → {0, … , M-1}
  - Send Put/Get to node id == Hash(key)
  - Process it with local KV store
- Challenges
  - But what if you don’t know “who” all the nodes are that are participating?
  - Perhaps they come and go …
  - What if some keys are *really* popular? (load balance)
  - Replicate K:V in multiple storage servers?
- Have something keep track - directory
- Lookup
  - Hmm, won’t this be a bottleneck and single point of failure?

Directory-Based Architecture

Have a node maintain the mapping between keys and the *machines* (nodes) that store the values associated with the keys



Iterative vs. Recursive Query



- **Recursive Query: Directory Server Delegates**
- **Iterative Query: Client Delegates**

Iterative vs Recursive Query

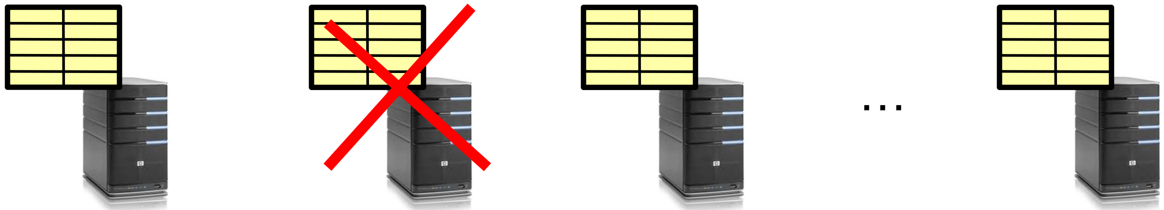
Recursive

+ Faster, as directory server is typically close to storage nodes
+ Easier for consistency: directory can enforce *an order* for all puts and gets
- Directory is a performance bottleneck

Iterative

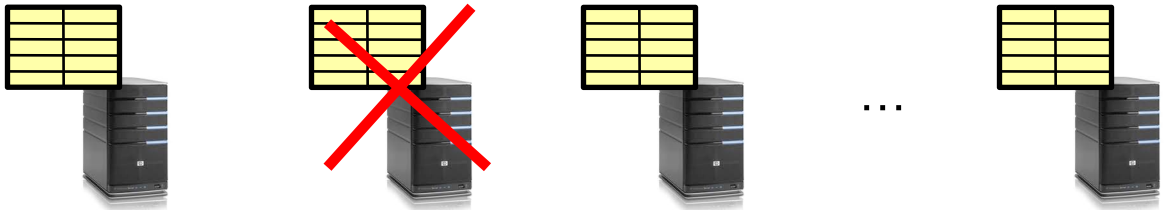
+ More scalable, clients do more work
- Harder to enforce consistency

Challenges



- **Fault Tolerance: handle machine failures without losing data and without degradation in performance**
- **Scalability:**
  - **Need to scale to thousands of machines**
  - **Need to allow easy addition of new machines**

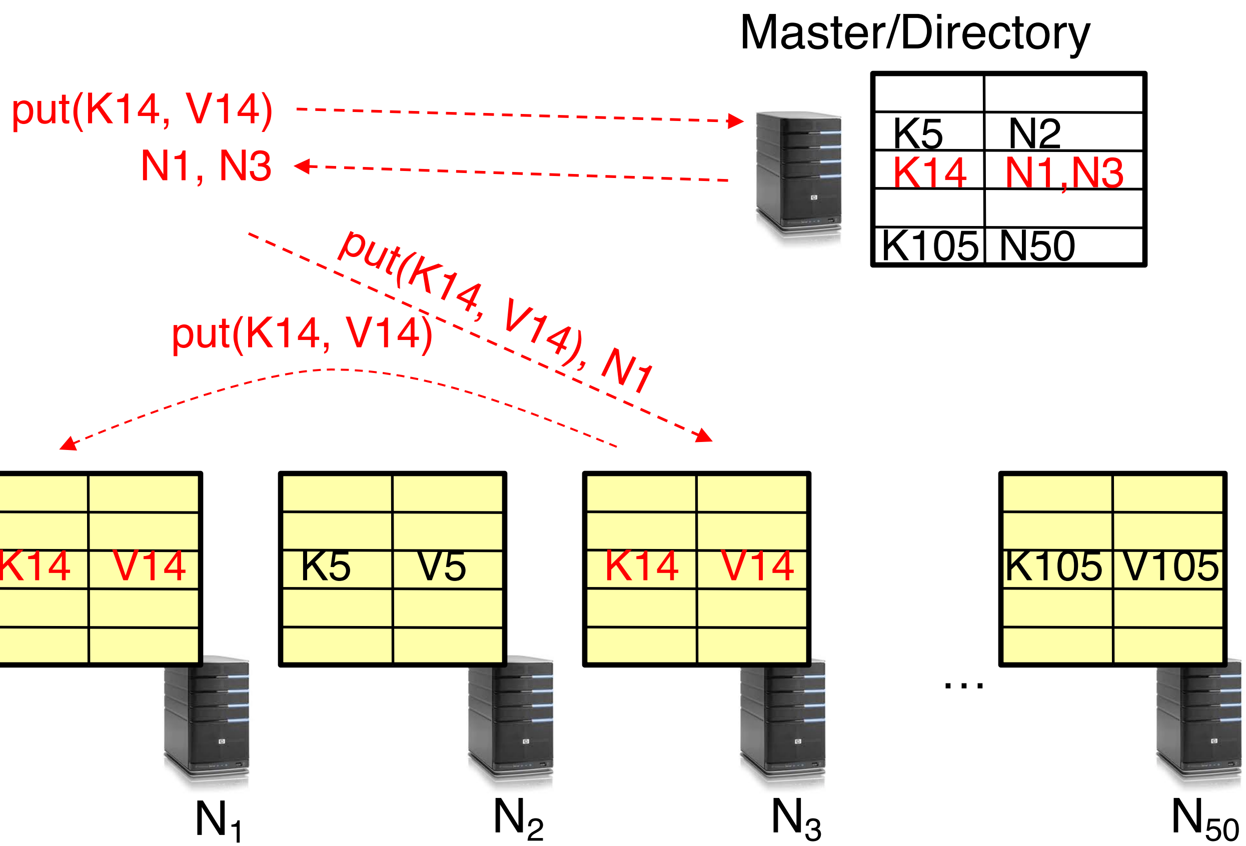
Challenges



- **Consistency: maintain data consistency in face of node failures and message losses**
- **Heterogeneity (esp. if deployed as peer-to-peer systems):**
  - **Latency: 1ms to 1000ms**
  - **Bandwidth: 32 Kb/s to 1 Gb/s**

Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures



Consistency

- Replication is essential for fault tolerance (and performance)
  - But introduces inherent challenges
- Need to make sure a value is replicated correctly

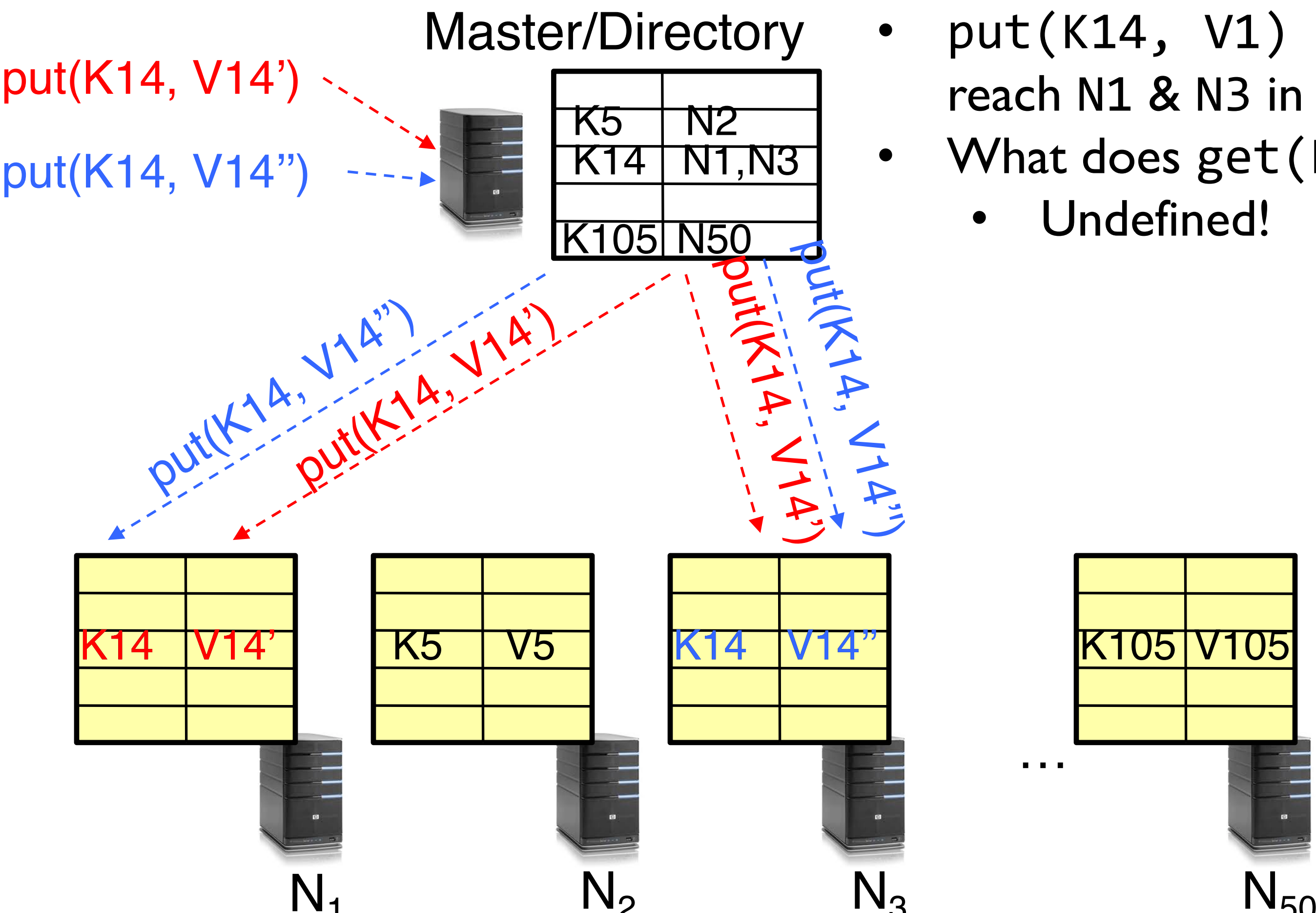
- How do you know a value is replicated on every expected node?

- *Wait* for acknowledgements from all expected nodes ???

Consistency

- **If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the “same order”**

- `put(K14, V1)` and `put(K14, V2)` reach N1 & N3 in reverse order
- What does `get(K14)` return?
  - Undefined!



How to ensure order?

- **Wait – for explicit acknowledgement**

- **Hmmm..**

Consistency & Fault Tolerance

- **What happens if a node fails during replication?**
  - **Pick another node and try again**

- **What happens if a node is slow?**
  - **Slow down entire `put`? Pick another node?**

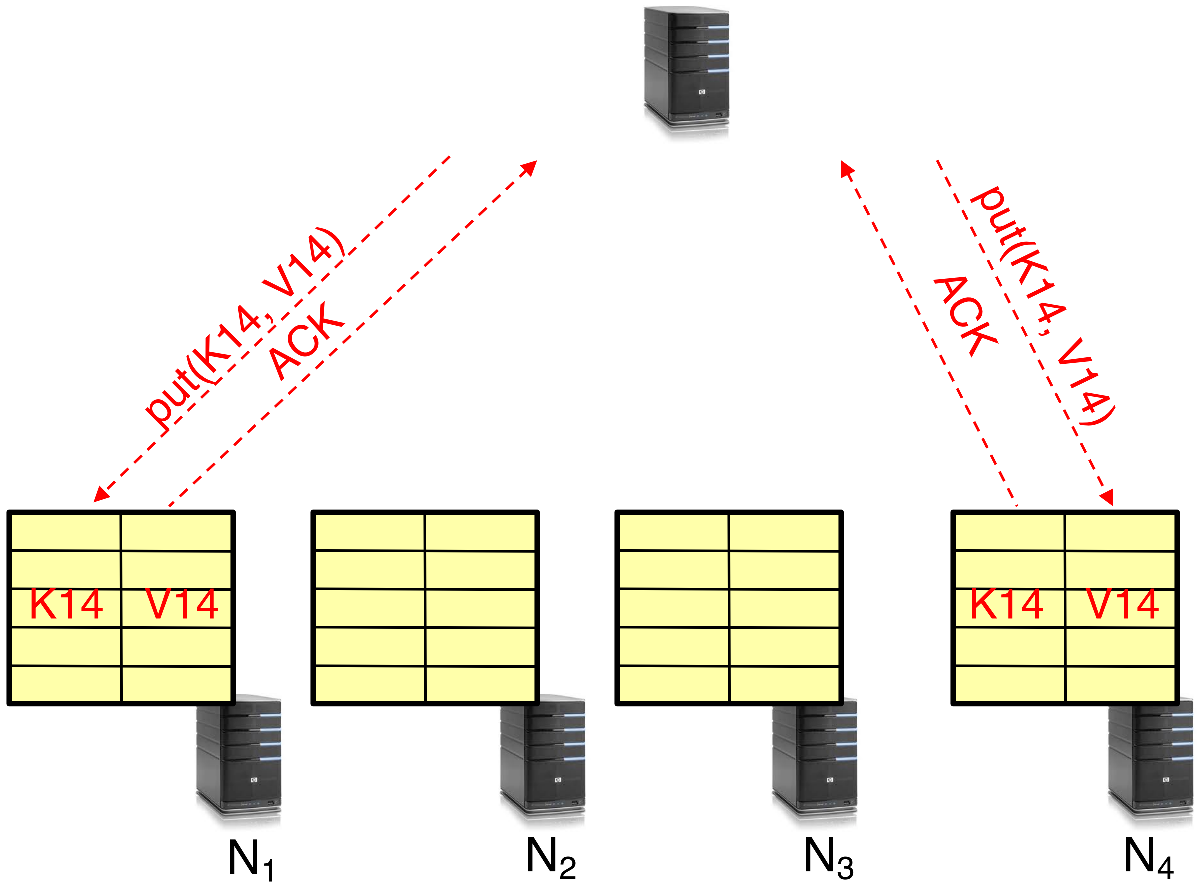
- **In general with multiple replicas: slow `put` and fast `get` operations**

Quorum Consensus

- Improve `put` and `get` operation performance by reducing the # of replicas to hear back from
- Define a replica set of size N
  - `put` waits for acknowledgements from at least W replicas
  - `get` waits for responses from at least R replicas
  - $W + R > N$
- Why does it work?
  - There is at least one node that contains the update
- Why might you use W+R > N+1?

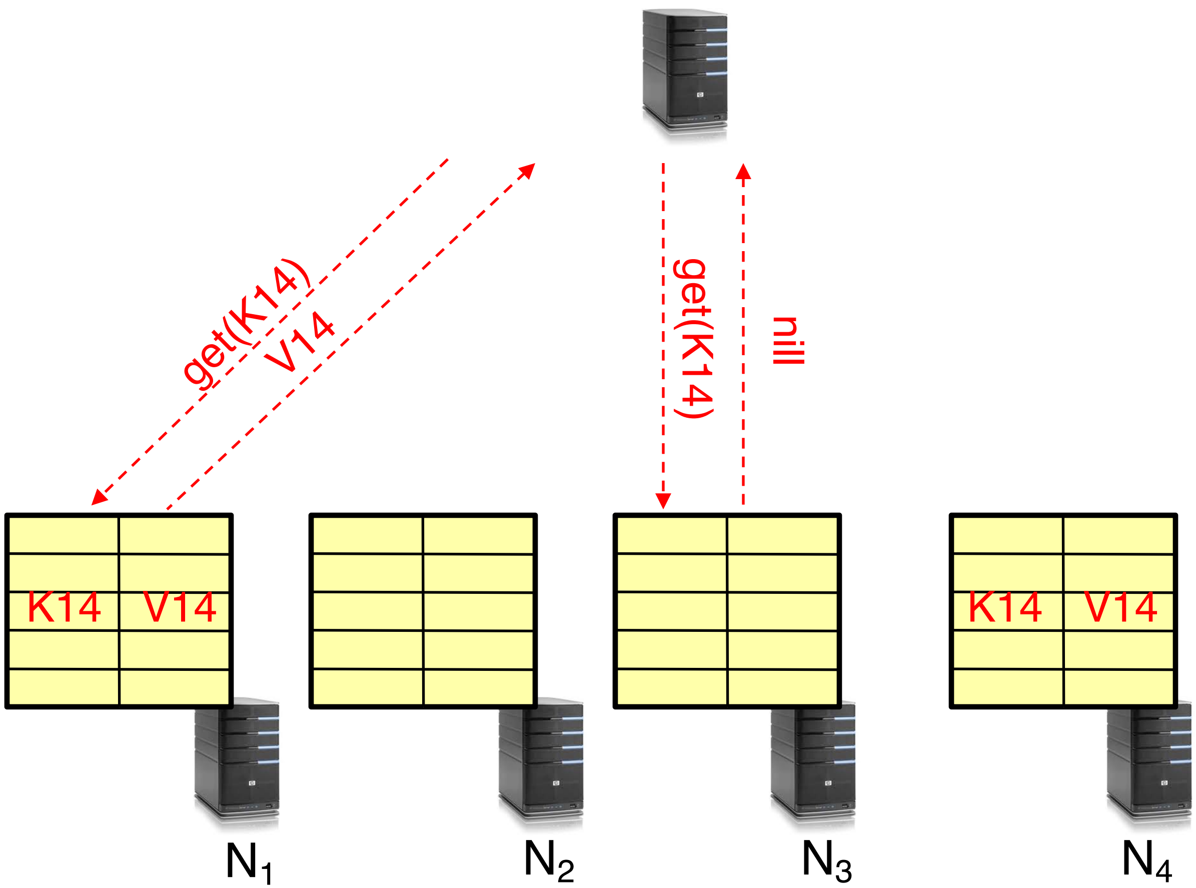
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N1, N2, N4}



Quorum Consensus Example

- Now, issuing `get` to any two nodes out of three will return the answer



Scalability

How easy is it to make the system bigger?

- **Storage: Use more nodes**
- **Number of Requests**
  - **Can serve requests from all nodes on which a value is stored in parallel**
  - **Master can replicate a popular item on more nodes**
- **Master/Directory Scalability**
  - **Replicate It (multiple identical copies)**
    - » **Maintain consistency across them**
  - **Partition it, so different keys are served by different directories**

Scalability: Load Balancing

- Directory tracks available storage at each node
  - Prefer to insert at nodes with more storage available
- What happens when a new node is added?
  - Cannot insert only new values at new node
  - Move values from heavily loaded nodes to new node
- What happens when a node fails?
  - Replicate values from failed node to other nodes

Scaling Up Directory

- Directory contains number of entries equal to number of key/value pairs in entire system
  - Could be tens or hundreds of billions of pairs
- Solution: Consistent Hashing
  - The set of storage nodes may change dynamically
    - » fail, enter, leave
  - Assign each node a unique ID in large namespace $[0..2^m-1]$
    - » m bit namespace, s.r., $M << 2^m$
    - » Each node can pick its ID at random !
  - hash keys in a manner that everyone assigns same range of IDs to a node
  - Each (key,value) stored at node with *smallest ID larger than hash(key)*
- Important property: Adding a new bucket doesn't require moving lots of existing values to new buckets

Key to Node Mapping Example

Partitioning example with
m = 6 → ID space: 0..63

0: Node 4 maps keys [59, 4]

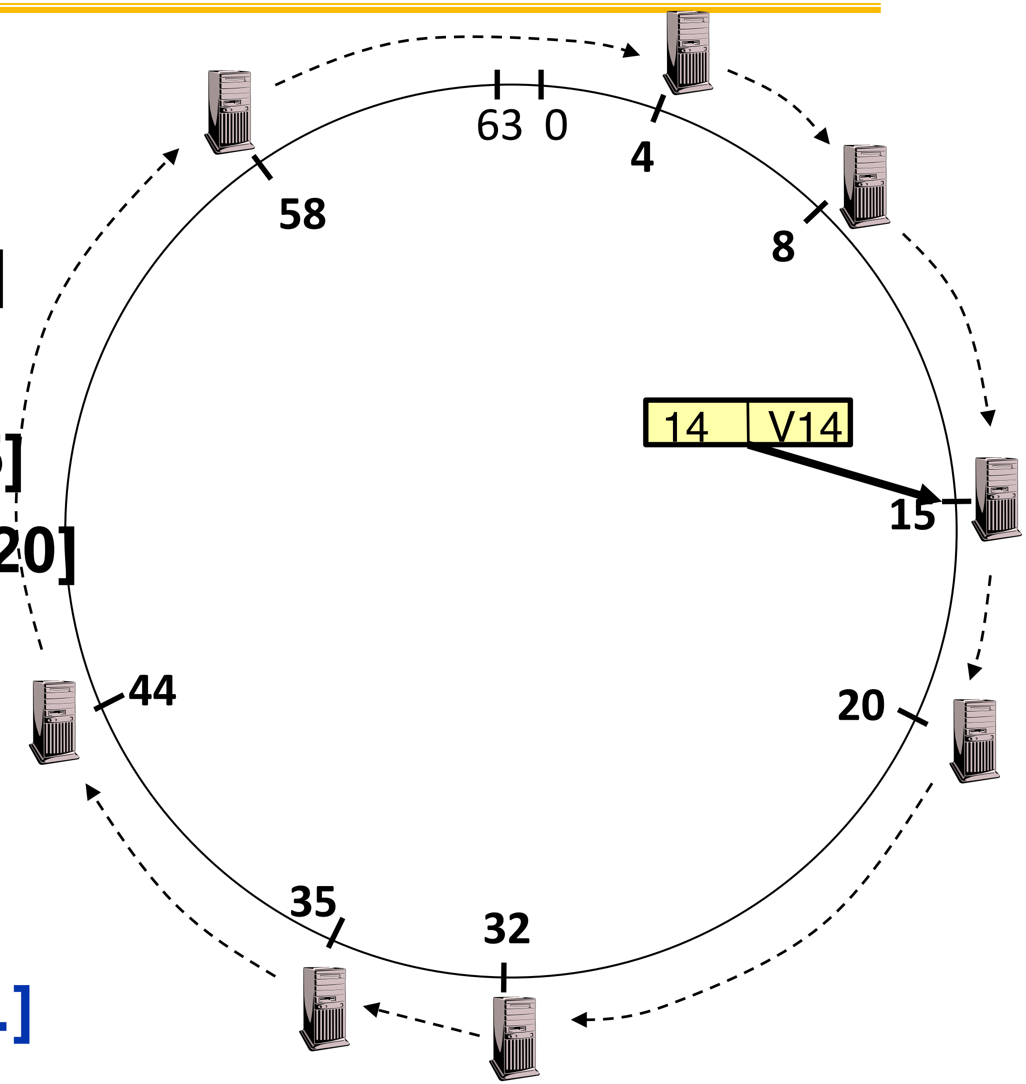
1: Node 8 maps keys [5,8]

2: Node 15 maps keys [9,15]

3: Node 20 maps keys [16, 20]

…

M-1: Node 58 maps [45, 58]



n := Hash(key)

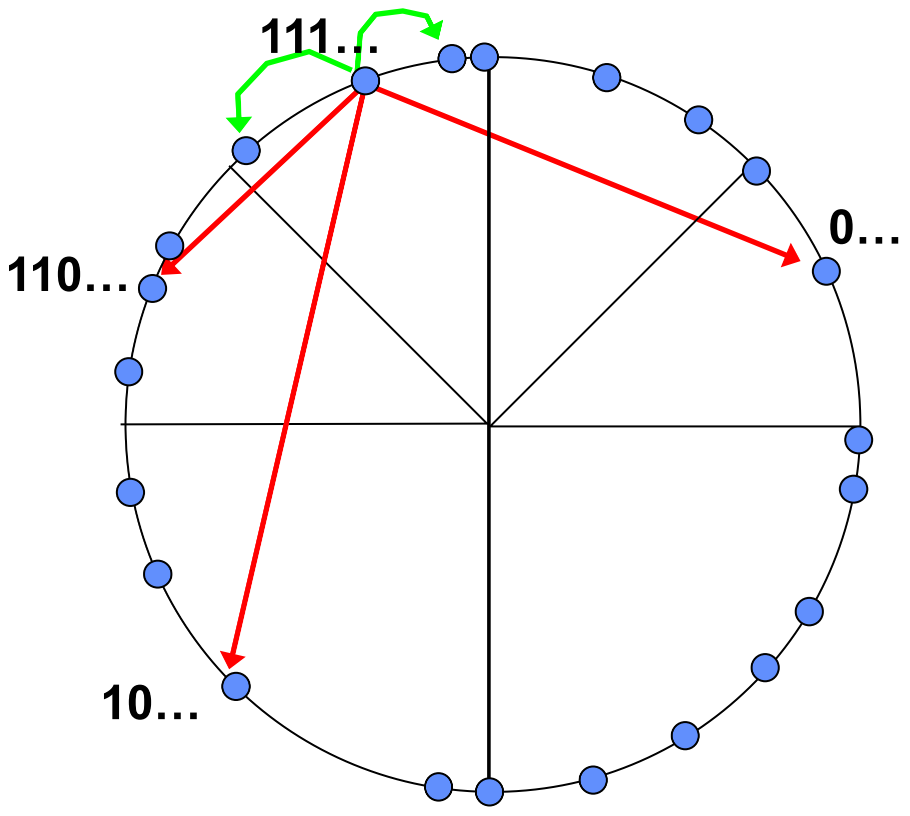
Find first *i* in [4, 8, 15, 20, …]

s.t., $N_i > n$ (mod M)

Performing a Lookup

- Any one with a list of Node IDs can hash a key and pick the node responsible for that range
  - But the mapping may have changed due to nodes join/leave
- Each node knows about its successor and predecessor in the "circle"
  - If it is no longer responsible for the hash(key), it can find node that is
  - All that is strictly needed for correctness
- Fully decentralized
  - Any node can act as a directory for clients
  - Still works if a node leaves the network
- Faster lookups: Each node maintains a routing table, allows client to get closer to destination in one hop

Example: Chord



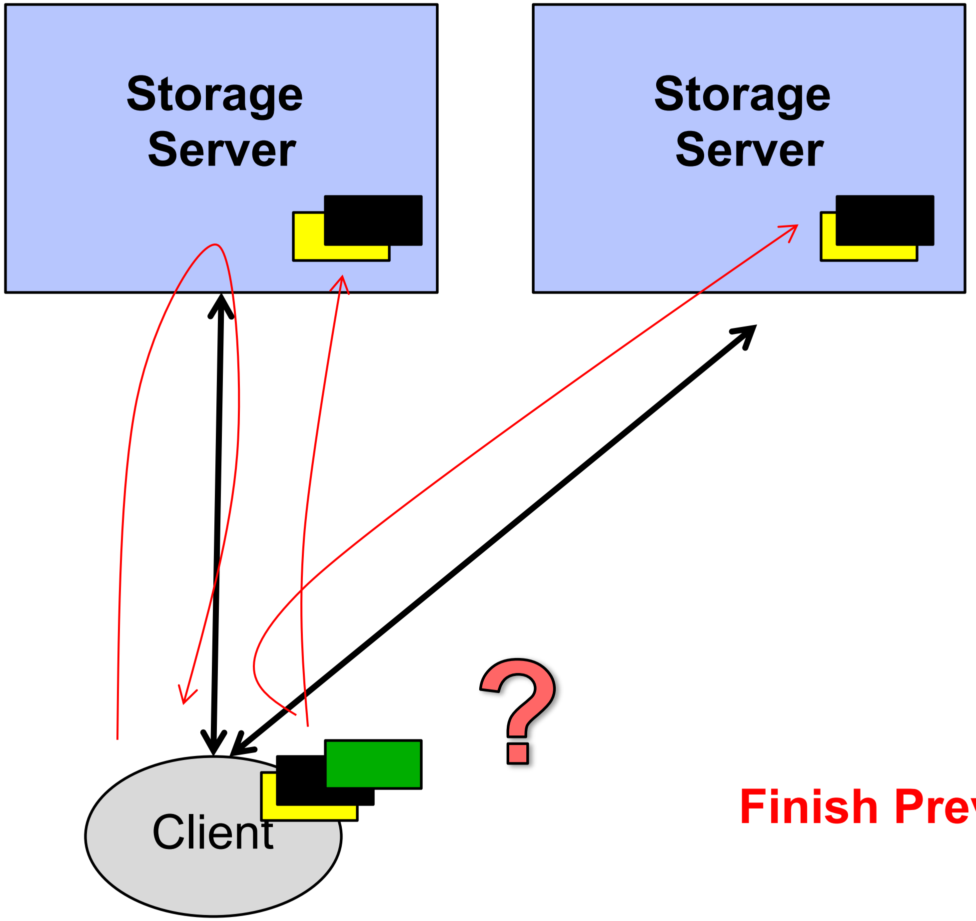
Logistics Break

- Key:Value, DHT questions
- 12/14 Eric Brewer guest lecture – Google VP Infra
- Work for remainder of term released
  - Provide as much flexibility as possible, given other demands
  - Do both HW5a and HW6a and one of HW{5 or 6}b
  - Extra credit applied to midterm for the other part b
    - » HW6b still in beta
- Project 3
  - Simplified concurrency aspects
  - Buffer Cache, inode extension, dirent
- … on to 2-phase commit

Consistency Review

- **Problem: shared state replicated across multiple clients, do they see a consistent view?**
  - **Propagation: Writes become visible to reads**
  - **Serializability: The order of writes seen by each client's series of reads and writes is *consistent* with a total order**
    - » **As if all writes and reads had been serviced at a single point**
    - » **The total order is not actually generated, but it could be**
- **Many distributed systems provide weaker semantics**
  - **Eventual consistency**

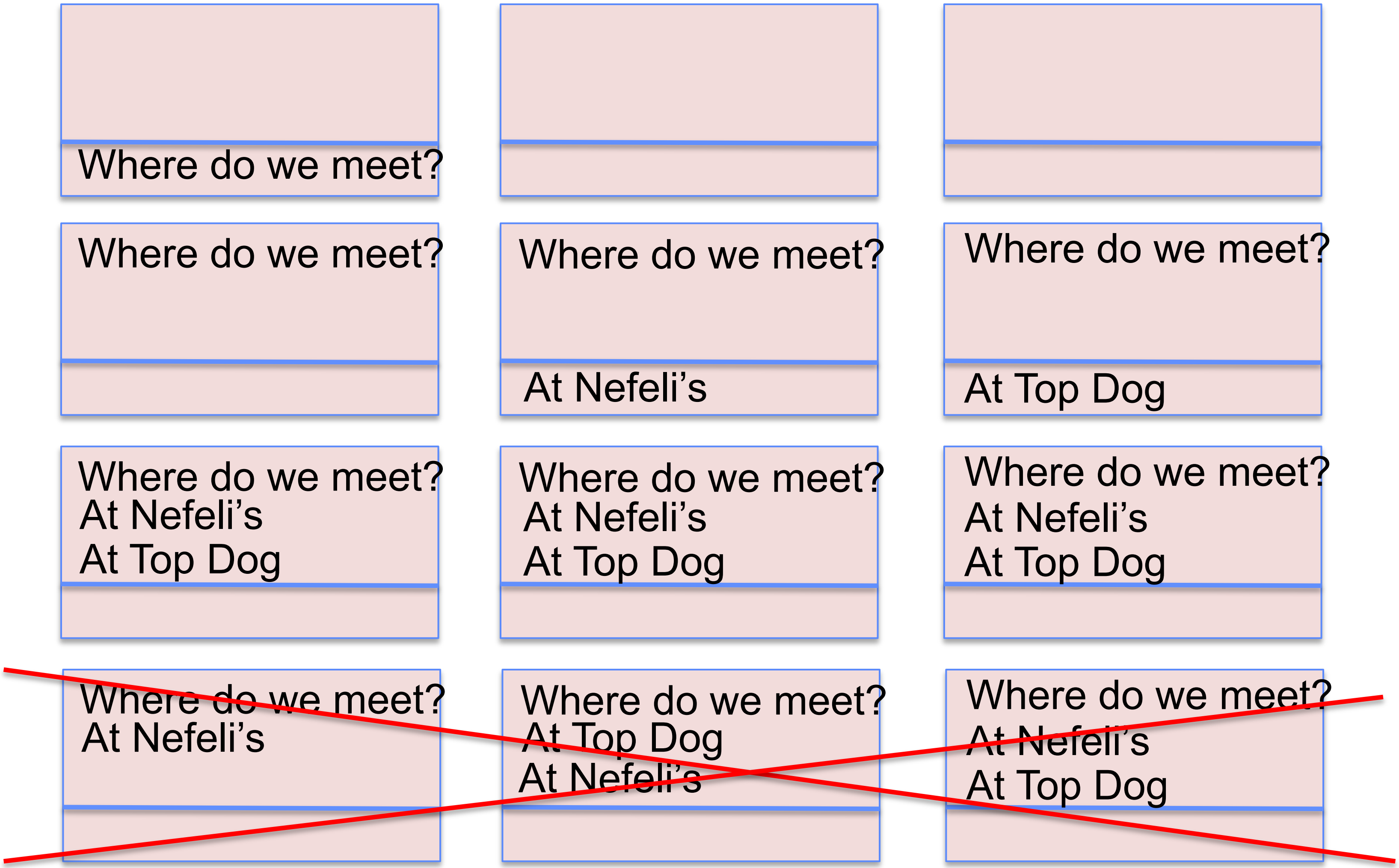
Unfinished Business: Multiple Servers



Finish Previous Lecture

- What happens if cannot update all the replicas?
- Availability => Inconsistency

In Everyday Life



Durability and Atomicity

- How do you make sure transaction results persist in the face of failures (e.g., server node failures)?

- Replicate store / database
  – Commit transaction to each replica

- What happens if you have failures during a transaction commit?
  – Need to ensure atomicity: either transaction is committed on all replicas or none at all

Distributed Consensus Making

- **Consensus problem**
  - All nodes propose a value
  - Some nodes might crash and stop responding
  - Eventually, all remaining nodes decide on the same value from set of proposed values
- **Distributed Decision Making**
  - Choose between "true" and "false"
  - Or Choose between "commit" and "abort"
- **Equally important (but often forgotten!): make it durable!**
  - How do we make sure that decisions cannot be forgotten?
    - » This is the "D" of "ACID" in a regular database
  - In a global-scale system?
    - » What about erasure coding or massive replication?
    - » Like BlockChain applications!

"Byzantine General's Paradox"

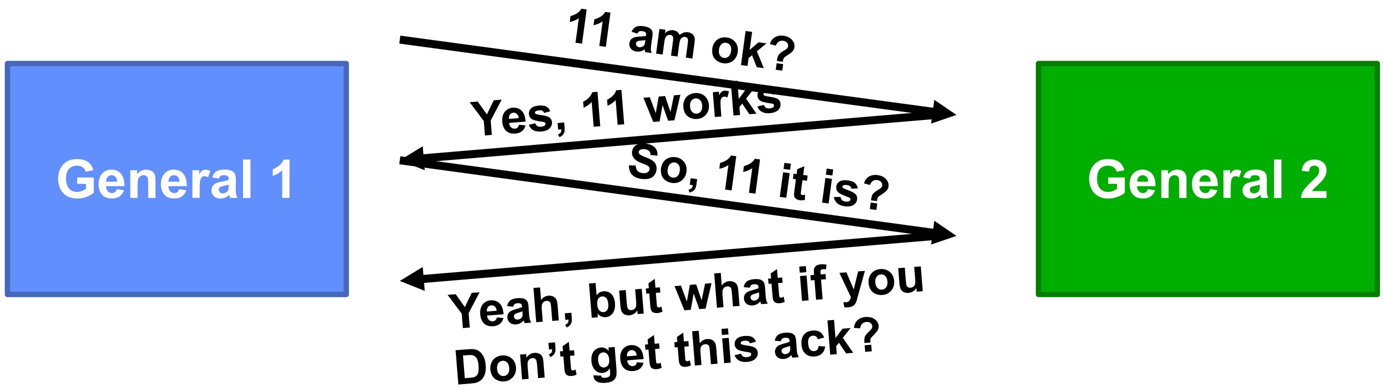
- Two generals located on opposite sides of their enemy's position
- Can only communicate via messengers
- Messengers go through enemy territory: might be captured

- Problem: Need to coordinate time of attack
  - Both generals lose unless they attack at same time
  - If they attack at same time, they win

- How do you ever know your message got through?
  - Requires a message… that might not get through

General's Paradox

- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
- No, even if all messages go through



Two-Phase Commit

- We can't solve the General's Paradox
  - No simultaneous action
  - But we can solve a related problem
- Distributed Transaction: Two (or more) machines agree to do something *or not do it* atomically
- Extra tool: Persistent Log
  - If machine fails, it will remember what happened
  - Assume log itself can't be corrupted

Two Phase (2PC) Commit

- 2PC is a distributed protocol
- High-level problem statement
  - If no node fails and all nodes are ready to commit, then all nodes COMMIT
  - Otherwise ABORT at all nodes
- Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)

2PC Algorithm

- One coordinator
- N workers (replicas)

- High level algorithm description
  - Coordinator asks all workers if they can commit
  - If all workers reply "VOTE-COMMIT", then coordinator broadcasts "GLOBAL-COMMIT",

    Otherwise coordinator broadcasts "GLOBAL-ABORT"
  - Workers obey the GLOBAL messages

Two-Phase Commit: Setup

- **One machine *(coordinator)* initiates the protocol**
- **It asks *every* machine to vote on transaction**

- **Two possible votes:**
  - **Commit**
  - **Abort**

- **Commit transaction only if unanimous approval**

Two-Phase Commit: Preparing

Agree to Commit

- Machine has guaranteed that it will accept transaction
- Must be recorded in log so machine will remember this decision if it fails and restarts

Agree to Abort

- Machine has guaranteed that it will never accept this transaction
- Must be recorded in log so machine will remember this decision if it fails and restarts

Two-Phase Commit: Finishing

Commit Transaction

- **Coordinator learns *all machines have agreed to commit***
- **Apply transaction, inform voters**
- **Record decision in local log**

Abort Transaction

- **Coordinator learns *at least on machine has voted to abort***
- **Do not apply transaction, inform voters**
- **Record decision in local log**

Two-Phase Commit: Finishing

Commit Transaction

- Coordinator learns *all machines have agreed to commit*
- Apply transaction, inform voters
- Record decision in local log

Abort Transaction

- Coordinator learns *at least on machine has voted to abort*
- Do not apply transaction, inform voters
- Record decision in local log

Because no machine can take back its decision, exactly one of these will happen

Detailed Algorithm

Coordinator Algorithm

Coordinator sends VOTE-REQ to all workers

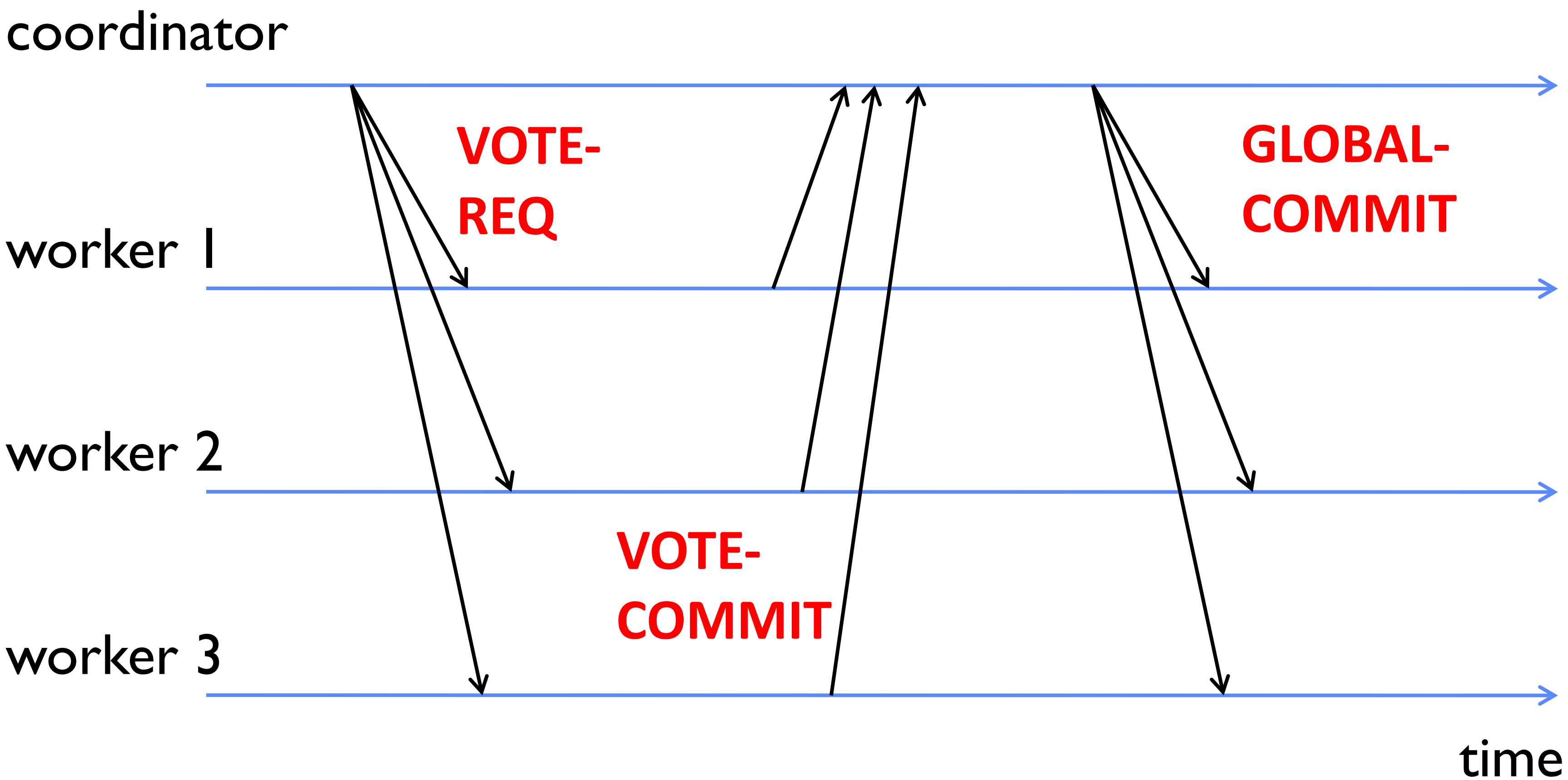
- If receive VOTE-COMMIT from all N workers, send GLOBAL-COMMIT to all workers
- If doesn't receive VOTE-COMMIT from all N workers, send GLOBAL-ABORT to all workers

Worker Algorithm

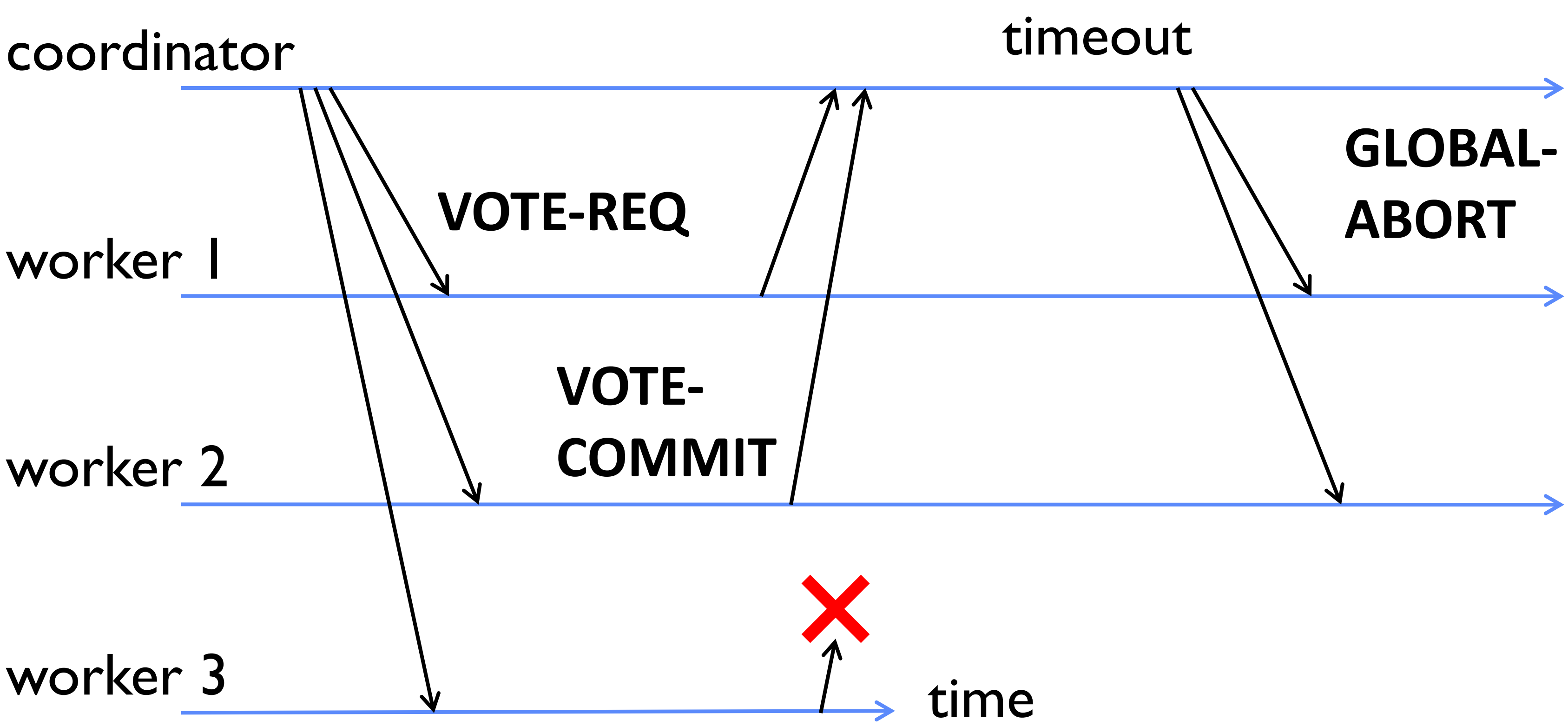
- Wait for VOTE-REQ from coordinator
- If ready, send VOTE-COMMIT to coordinator
- If not ready, send VOTE-ABORT to coordinator
  - And immediately abort

- If receive GLOBAL-COMMIT then commit
- If receive GLOBAL-ABORT then abort

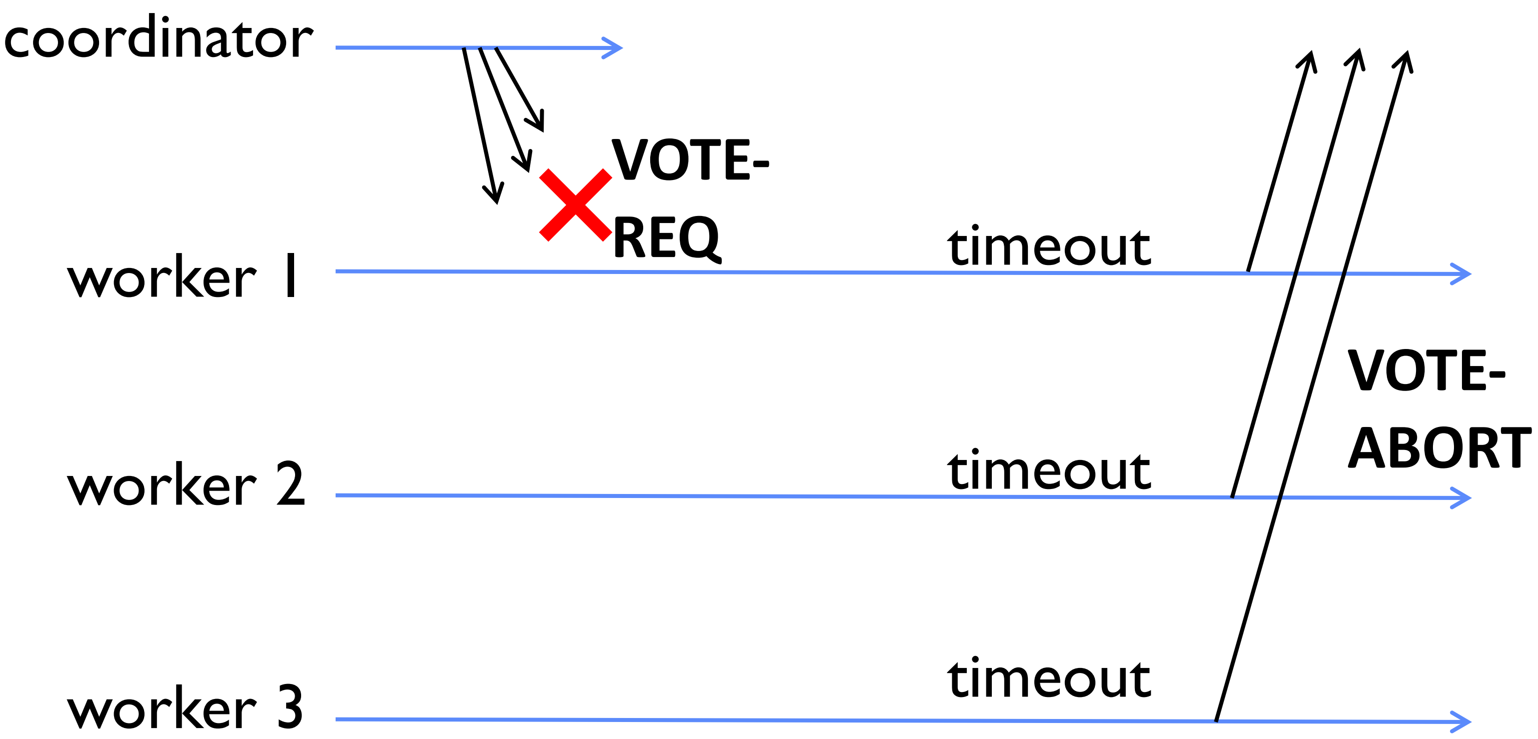
Example: Failure-Free 2PC



Example of Worker Failure



Example of Coordinator Failure



Formalizing Two-Phase Commit

- ***N*** **workers (replicas): actually perform *transactions***
- **One coordinator (may also serve a worker)**
  - **Asks each worker to vote on transaction**
  - **Tells every machine result of the vote (workers don't need to ask each other)**

Messages in Two-Phase Commit

Coordinator → Worker

- **VOTE-REQ**

Worker → Coordinator

- **VOTE-COMMIT**
- **VOTE-ABORT**

Coordinator → Worker

- **GLOBAL-COMMIT**
- **GLOBAL-ABORT**

No taking back: always logged before sending

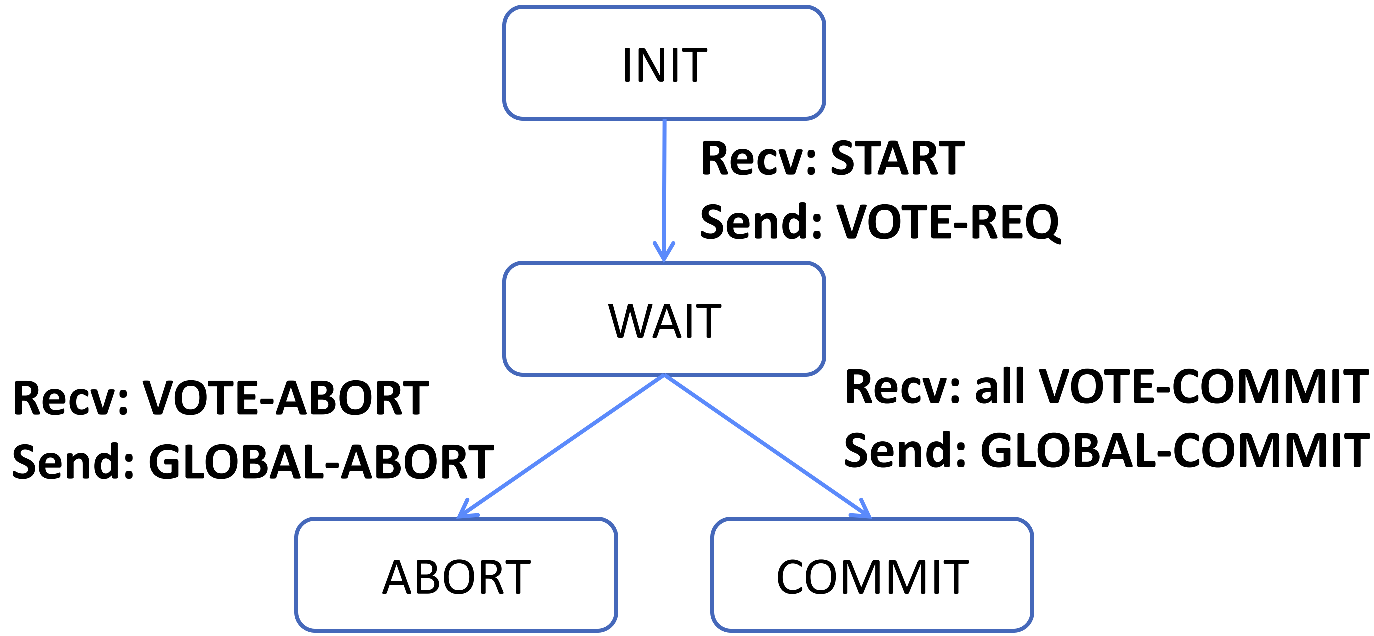
Actual result of transaction attempt

State Machines

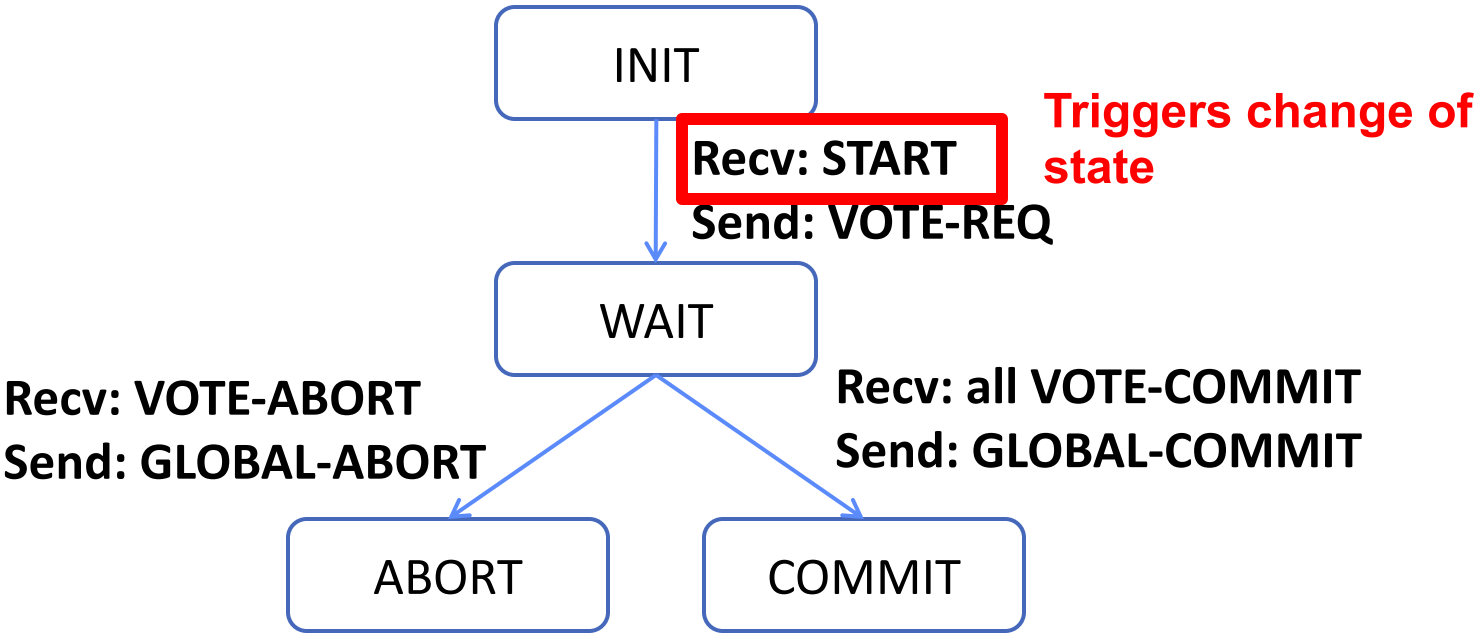
- **Distributed systems are hard to reason about**
- **Want a *precise* way to express each node's behavior that is also *easy to reason about***

- **One approach: State Machine**
  - **Every node is in a *state***
  - **When the node receives a message (or timeout),**
  - **it *transitions* to another state and**
  - **Sends zero or more messages**

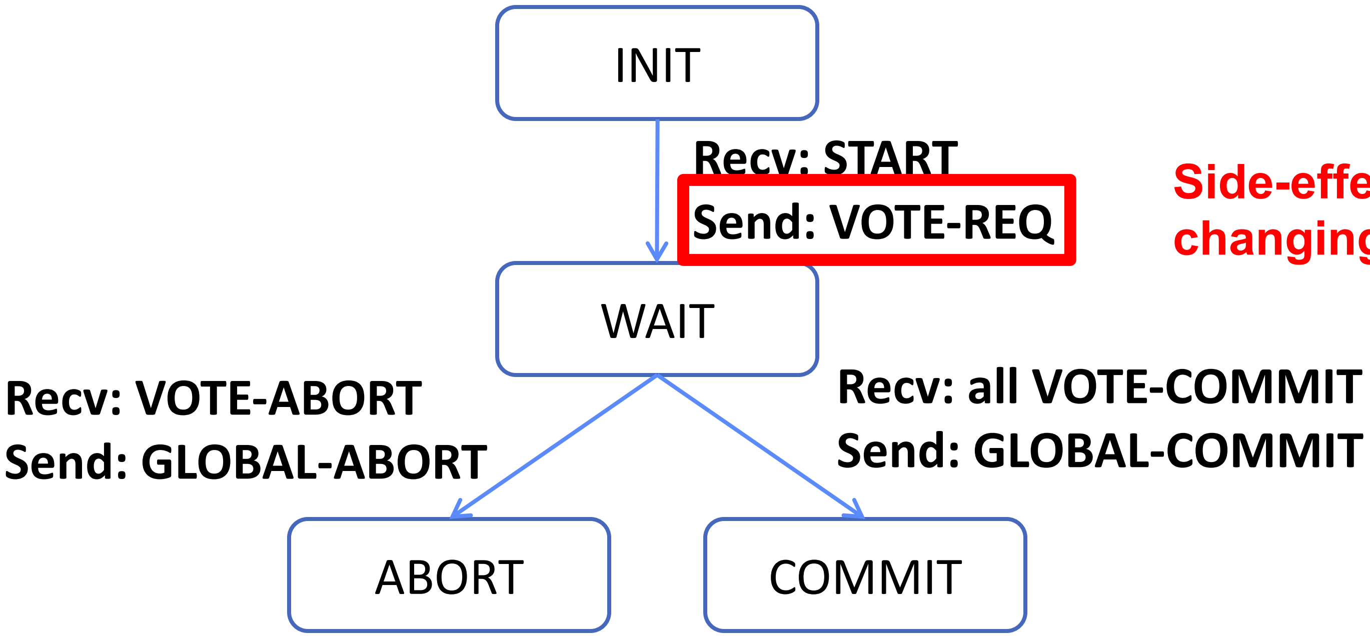
Coordinator's State Machine



Coordinator's State Machine

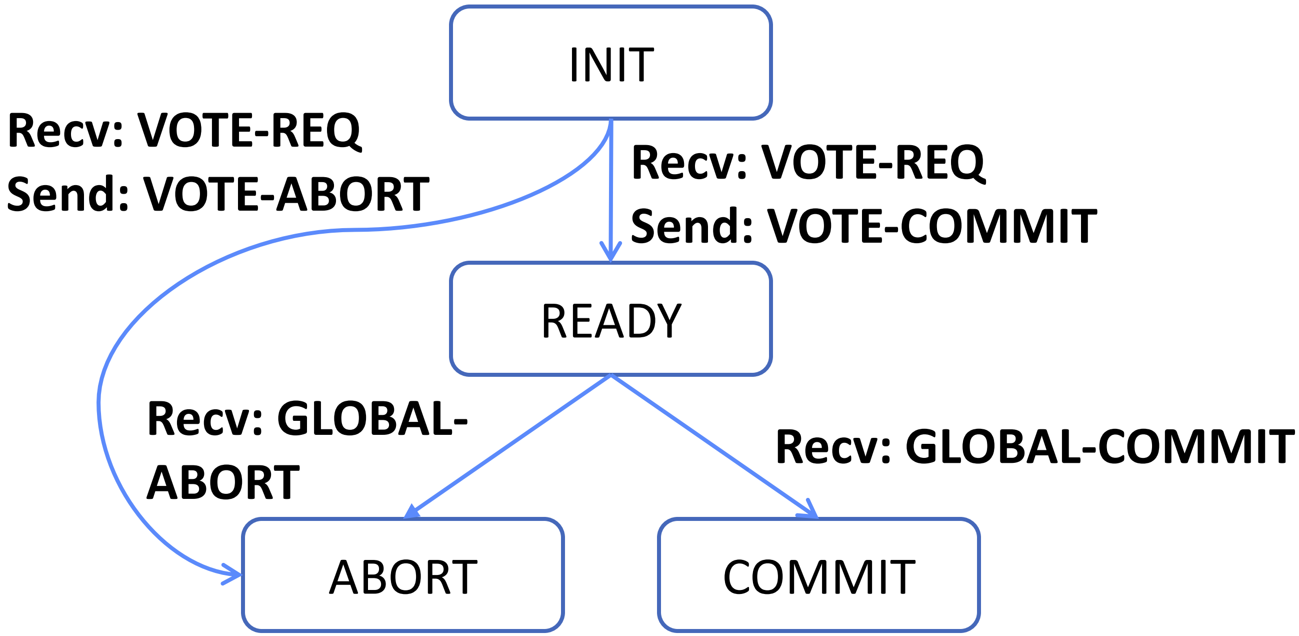


Coordinator's State Machine

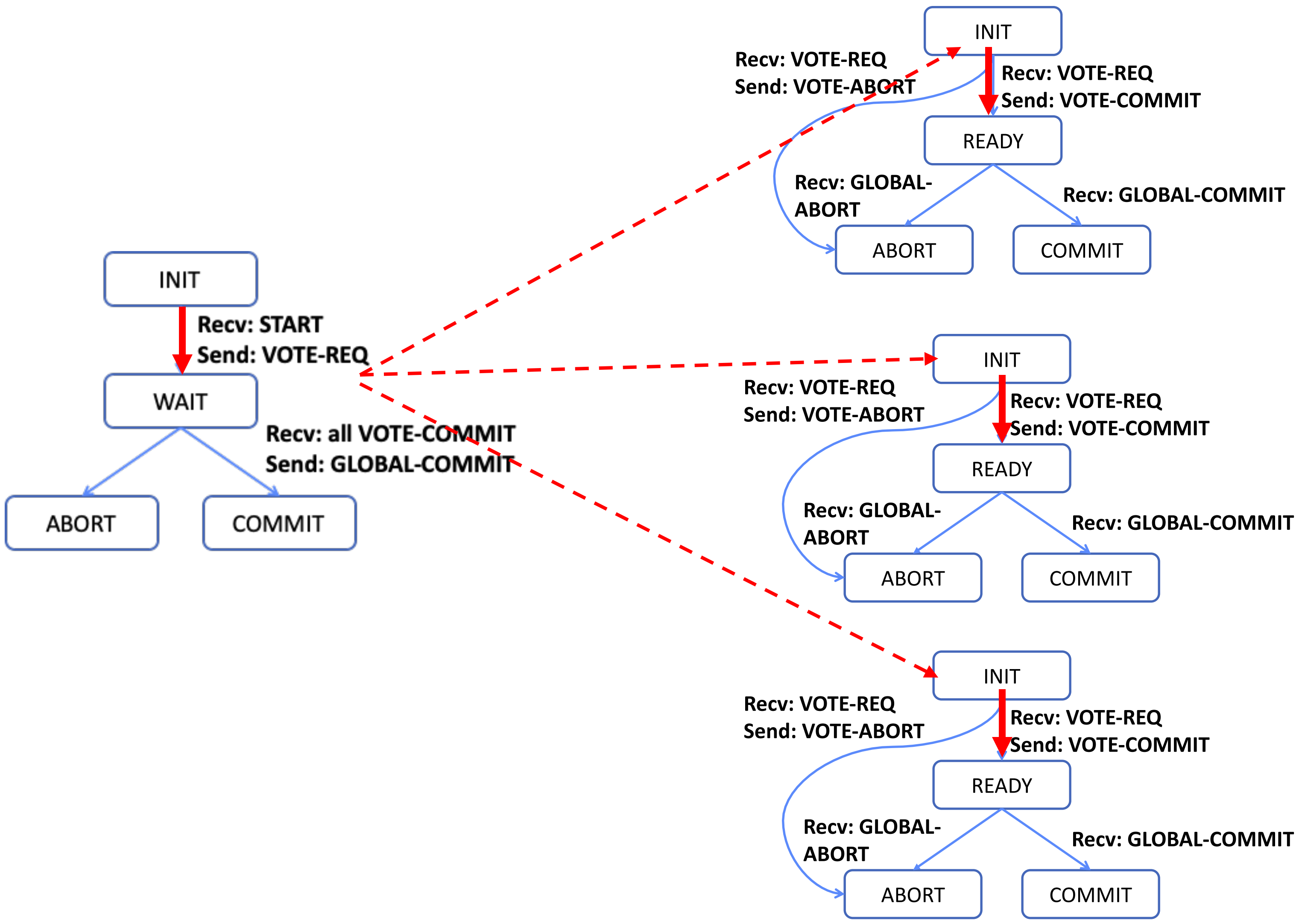


Side-effect of changing state

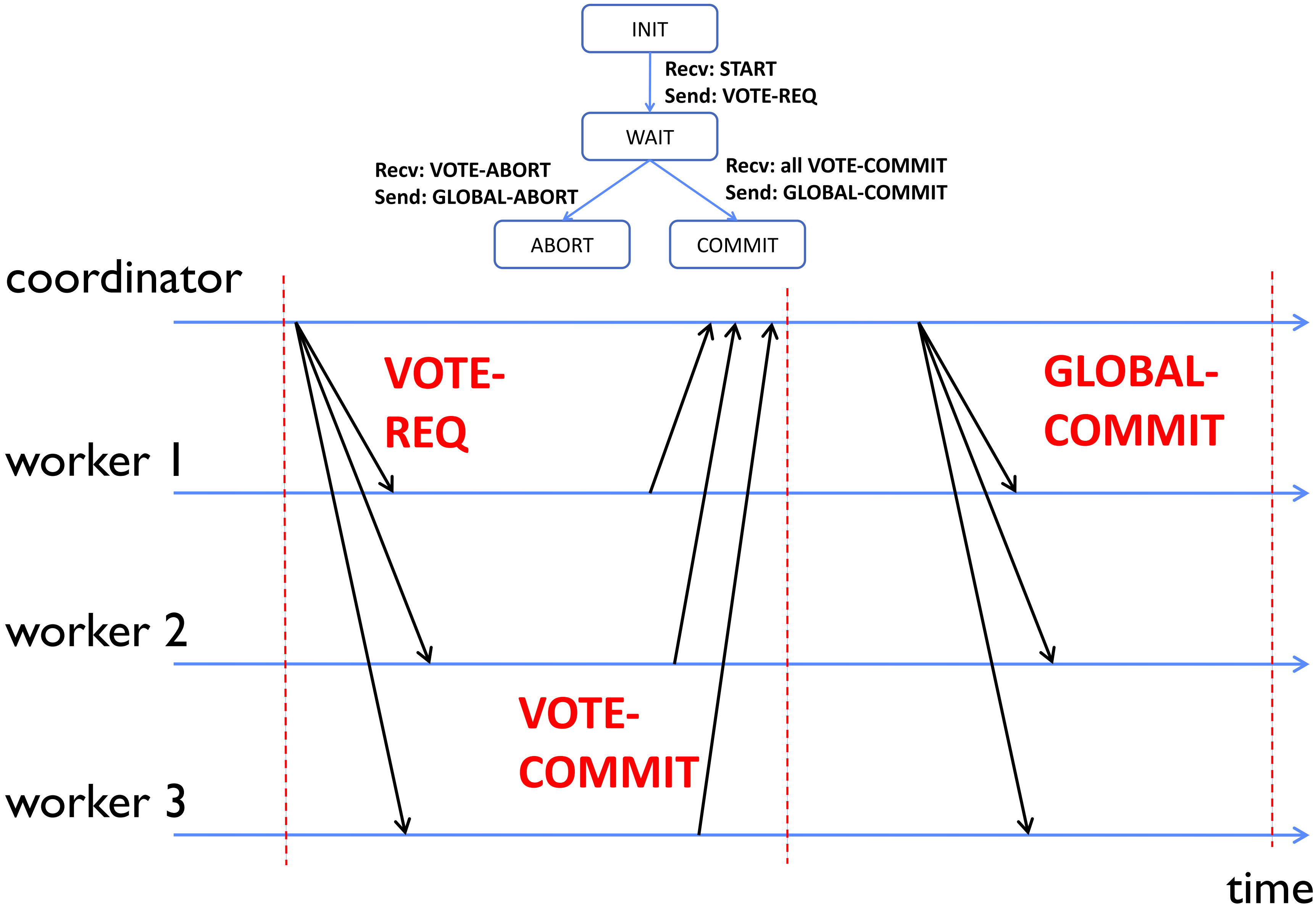
Worker's State Machine



Protocol: cooperating state machines

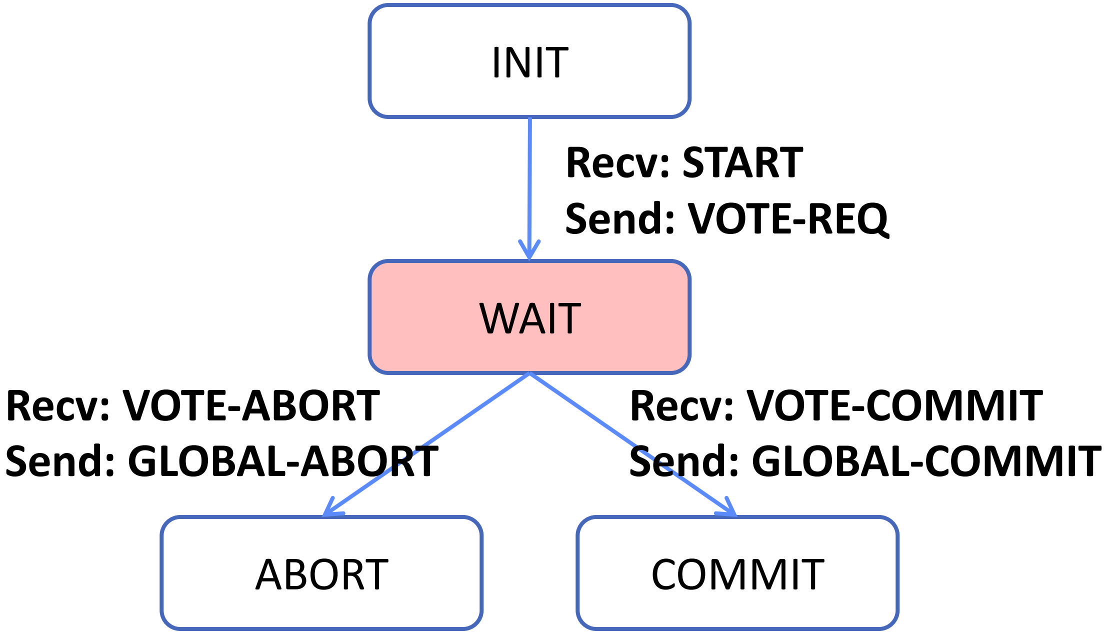


Example: Failure-Free 2PC (w/ state)

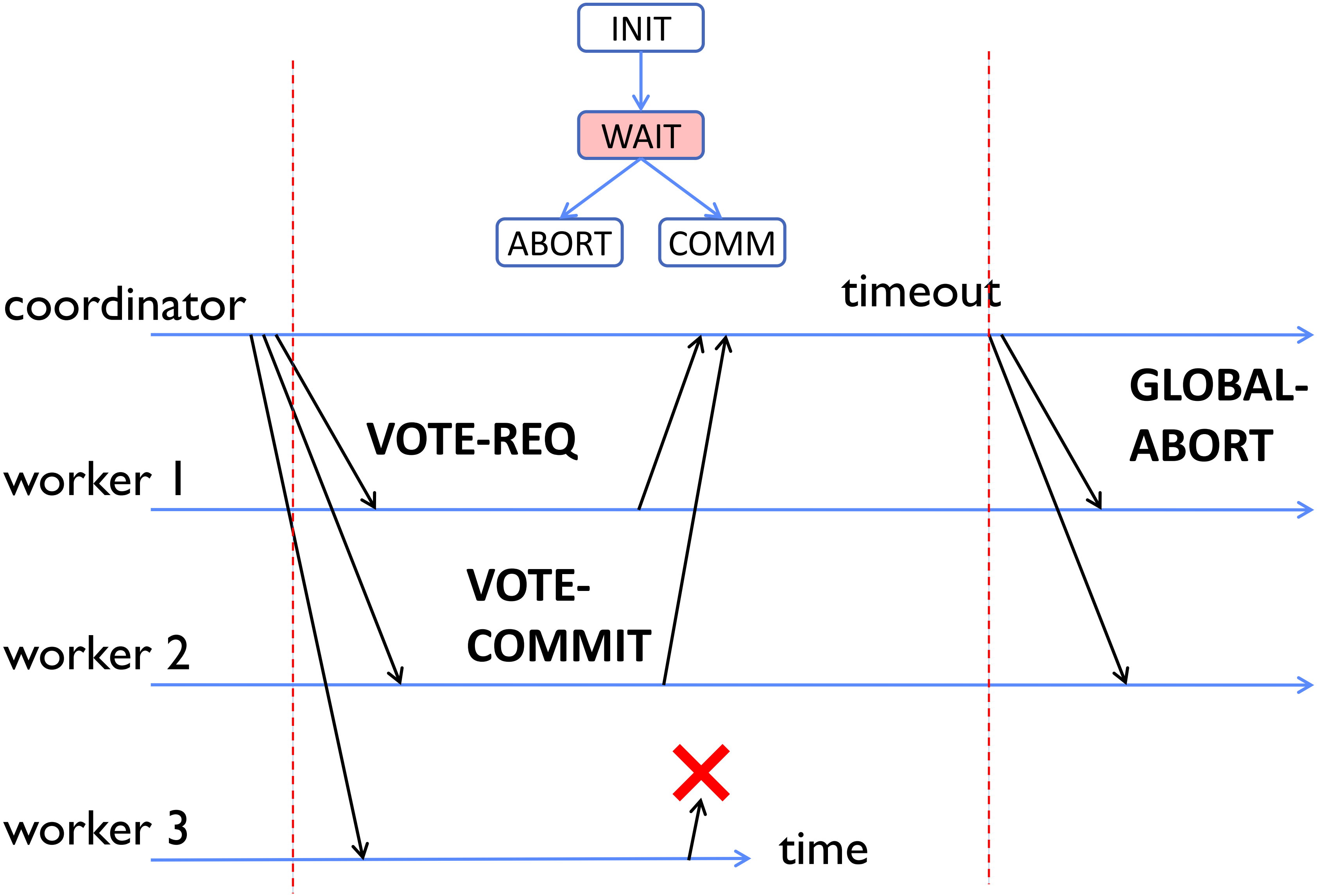


Dealing with Worker Failures

- **Failure only affects states in which the coordinator is waiting for messages**
- **In WAIT, if coordinator doesn't receive *N* votes, it times out and sends GLOBAL-ABORT**

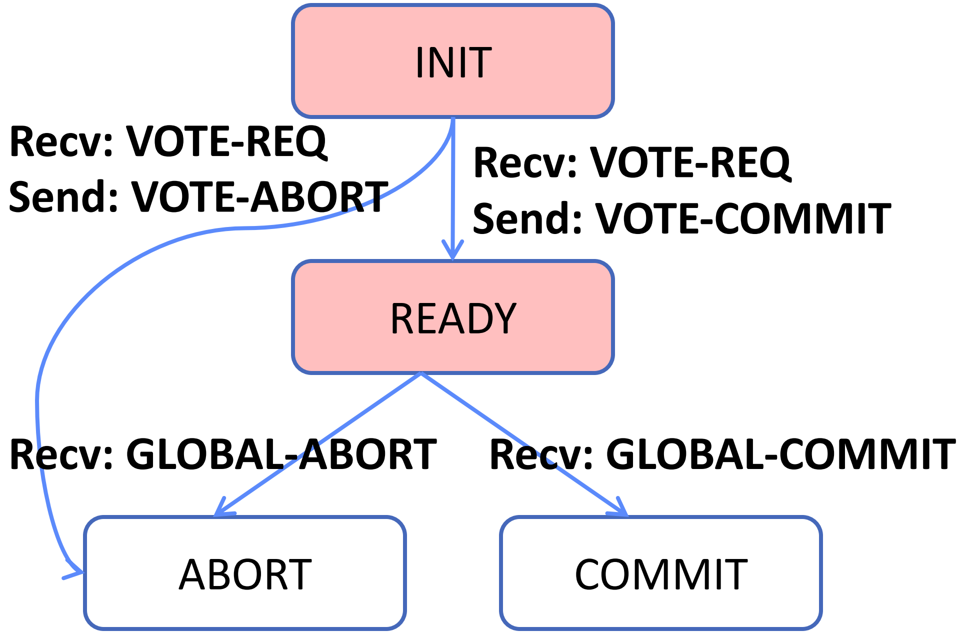


Example of Worker Failure (w/ state)

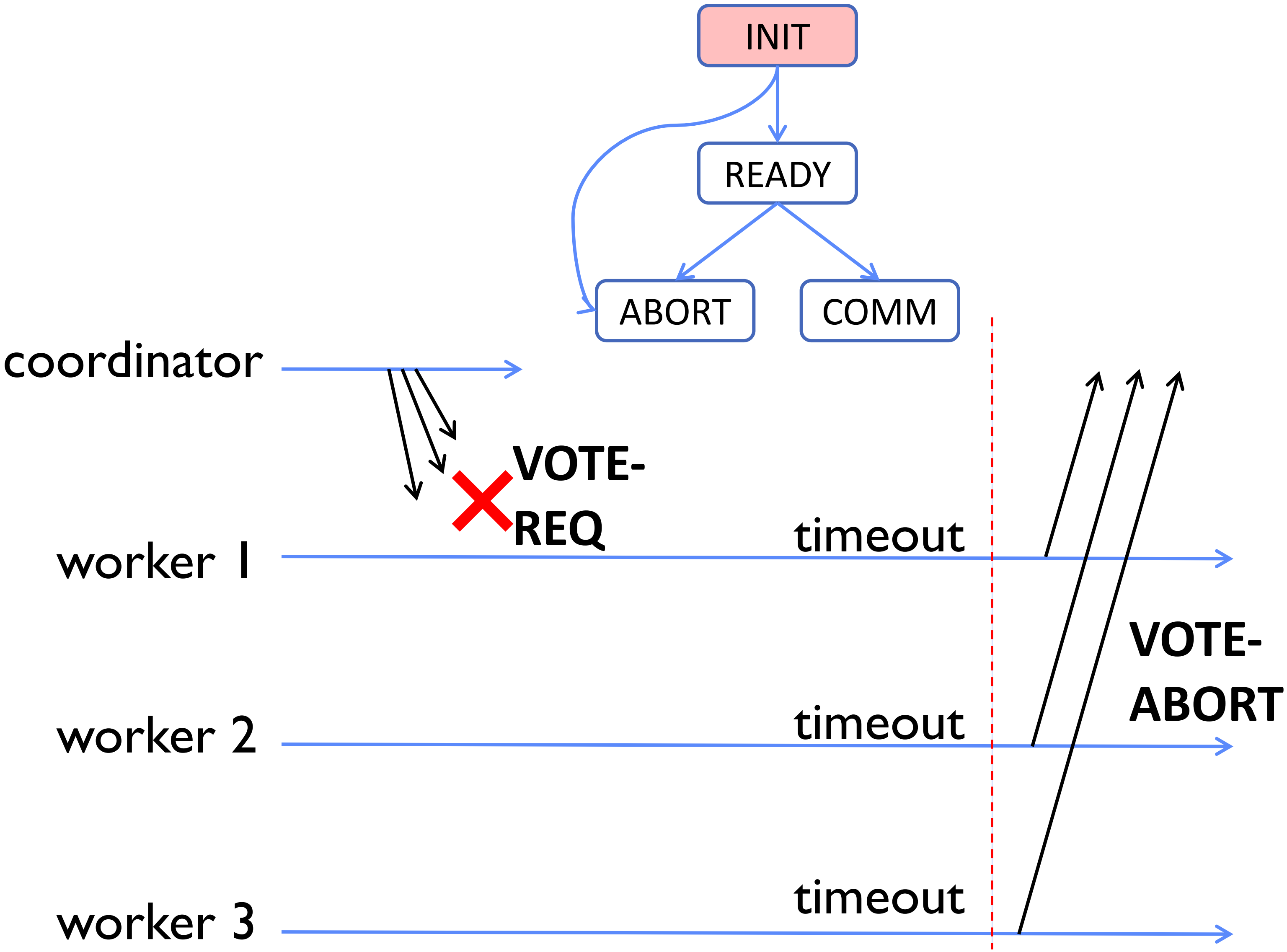


Dealing with Coordinator Failure

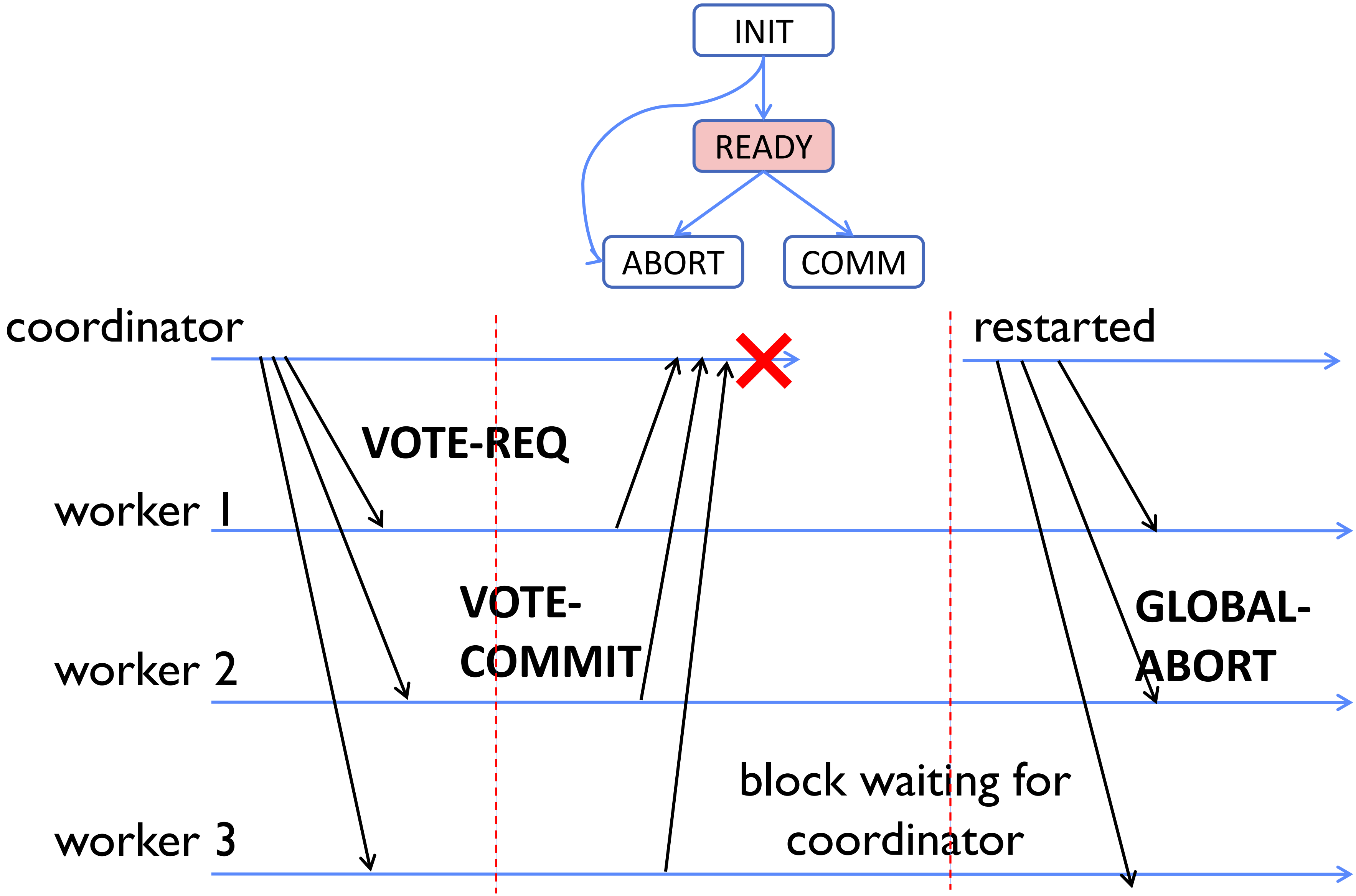
- **Worker waits for VOTE-REQ in INIT**
  - **Worker can time out and abort (coordinator handles it)**
- **Worker waits for GLOBAL-* message in READY**
  - **If coordinator fails, workers must BLOCK waiting for coordinator to recover and send GLOBAL_* message**



Example of Coordinator Failure (1)



Example of Coordinator Failure (2)

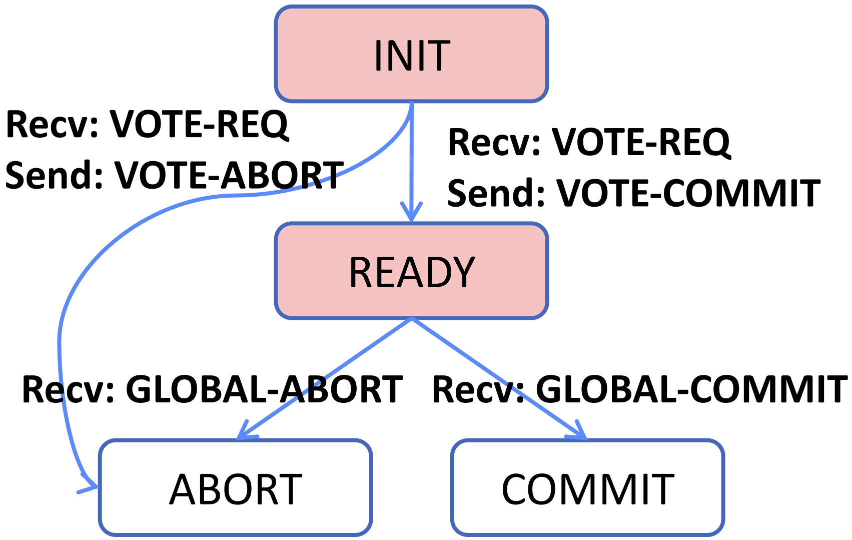


Failure Recovery

- **Nodes need to know what state they are in when they come back from a failure**
- **How? Log events on local disk, SSD, NVRAM**
- **Then we have the following recovery rules:**
  - **Coordinator *aborts* transaction if it was in the INIT, WAIT, or ABORT states**
  - **Coordinator *commits* transaction if it was in COMMIT**
  - **Worker *aborts* if in INIT or ABORT states**
  - **Worker *commit*s if it was in COMMIT state**
  - **Worker "*asks*" coordinator what to do if in READY state**

Blocking for Coordinator to Recover

- A worker waiting for global decision can ask fellow workers about their state
  - If another worker is in ABORT or COMMIT state then coordinator must have sent GLOBAL-*
    - Thus, worker can safely abort or commit, respectively
  - If another worker is still in INIT state then both workers can decide to abort
  - If all workers are in ready, need to **BLOCK** (don't know if coordinator wanted to abort or commit)



Blocking

- What if *both* coordinator and a worker fail?
- The remaining workers can still consult each other
- But they can't reach a conclusion on what to do!

Why?

- If all workers in INIT, we still don't know state of failed worker *w*
- *w* may have been first to be notified of a commit, and then coordinator and *w* crashed

Blocking for Coordinator

- What if *both* coordinator and a worker fail?
- The remaining workers can still consult each other
- But they can't reach a conclusion on what to do!

This problem motivated *Three Phase Commit*

Distributed Consensus

- Two- and Three-Phase commit make a decentralized decision
- Example: Changing the value of a key among all replicas for the key

- But they are hardly the only solutions to this problem

Parallel vs Distributed

- **Distributed: different machines responsible for different parts of task**
  - Usually no centralized state
  - Usually about different responsibilities or redundancy

- **Parallel: different parts of same task performed on different machines**
  - Usually about performance

Summary

- **Key Value Store: Simple put and get operations**
  - **Fault tolerance: replication**
  - **Scalability: Add nodes, balance load, no central directory**
  - **Consistency: Quorum consensus for better performance**
- **Consensus Goal: Everyone agrees on the state of the distributed system**
  - **Doesn’t depend who you ask**
  - **Doesn’t matter if nodes go down**
- **Distributed Transactions**
  - **Atomic, can’t revert once agreement is reached**

Summary: Two-Phase Commit

- Voting protocol requires unanimity
- Transaction committed if and only if: all workers and coordinator vote to commit
- Nodes never take back their vote
  - Logged for durability
- Nodes work in lock step (for an item)
  - Don't perform new transactions until old one is resolved
  - Stall until transaction is resolved