Distributed Systems 3: Distributed Transactions

Sam Kumar

CS 162: Operating Systems and System Programming

Lecture 25

https://inst.eecs.berkeley.edu/~cs162/su20

Read: Distributed Systems for Fun and Profit, Ch 4

Recall: Key-Value Stores (KV Store)

- Simple Interface:
  - `put(key, value); // Insert/write value associated with key`
  - `get(key); // Retrieve/read value associated with key`

- Behaves just like a dictionary in Python

- What about a collaborative key-value store rather than message passing or file sharing?
- Can we make it scalable and reliable?

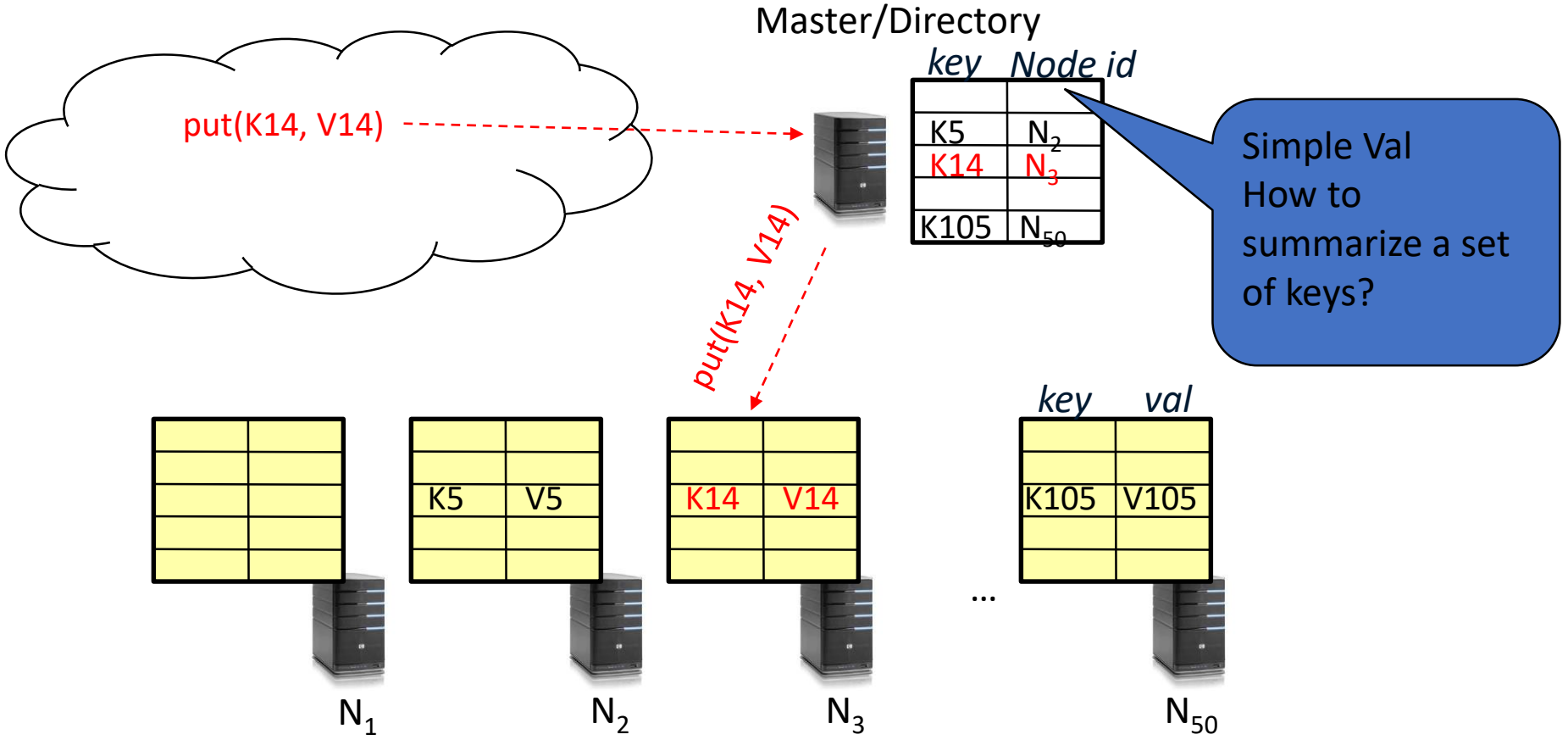
Recall: Distributed Key-Value Store vs. Distributed File System

- Distributed File System
  - One server
  - Challenge: keep clients' caches consistent

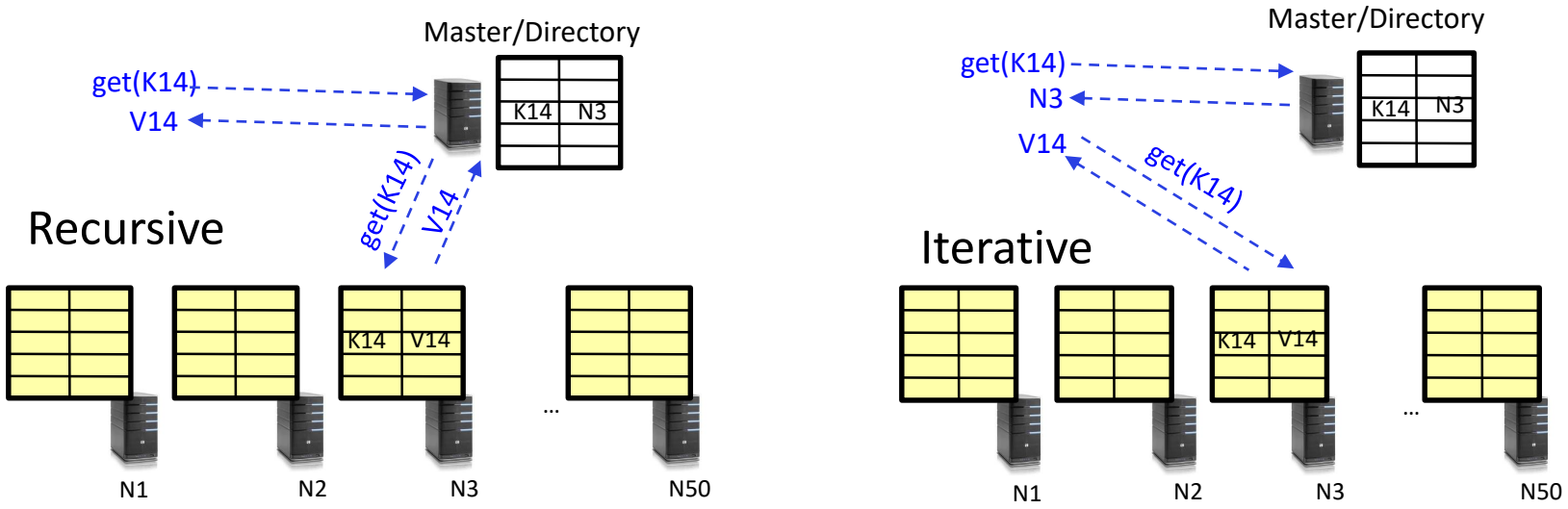
- Distributed Key-Value Store
  - Many servers
  - Challenge: keep server state consistent
  - Challenge: take advantage of multiple servers to scale the system
  - (Often, safe to assume no cache at the client --- or if it's there, it isn't part of the consistency problem)

Recall: Directory-Based Architecture

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



Recall: Iterative vs. Recursive Query



- Recursive Query: Directory server queries the appropriate node
- Iterative Query: Directory server tells client the node; then client queries that node

Reliability, Performance, and Consistency

- What is the key to **reliability** and **performance**?

Replication

- What is the source of **inconsistency**?

Replication

How to ensure that PUTs are applied to the replicas in order?

- **Wait** for explicit acknowledgment
  - Just like synchronization to order threads in a process…

- But doesn't this make it run slower???
  - Intuitively, the system should run *faster*, since we have more replicas to work with (remember RAID?)

Recall: Quorum Consensus

- Improve `put` and `get` operation performance **in the presence of replication**
  - Reduce the number of replicas to hear back from

- Define a replica set of size N
  - `put` waits for acknowledgements from at least ***W*** replicas (a "write quorum")
  - `get` waits for responses from at least ***R*** replicas (a "read quorum")
  - Make sure that ***W + R > N***

- Why does it work?
  - If ***W + R > N***, then every read quorum has at least one replica in common with the write quorum used for the latest write

- Why might you use W+R > N+1?

Recall: Scalability

- **How easy is it to make the system bigger?**

- Scaling Amount of Storage: Use more nodes
- Scaling 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
- Scaling the Master/Directory Node
  - Replicate It (multiple identical copies)
    - Maintain consistency across them
  - Partition it, so different keys are served by different directories

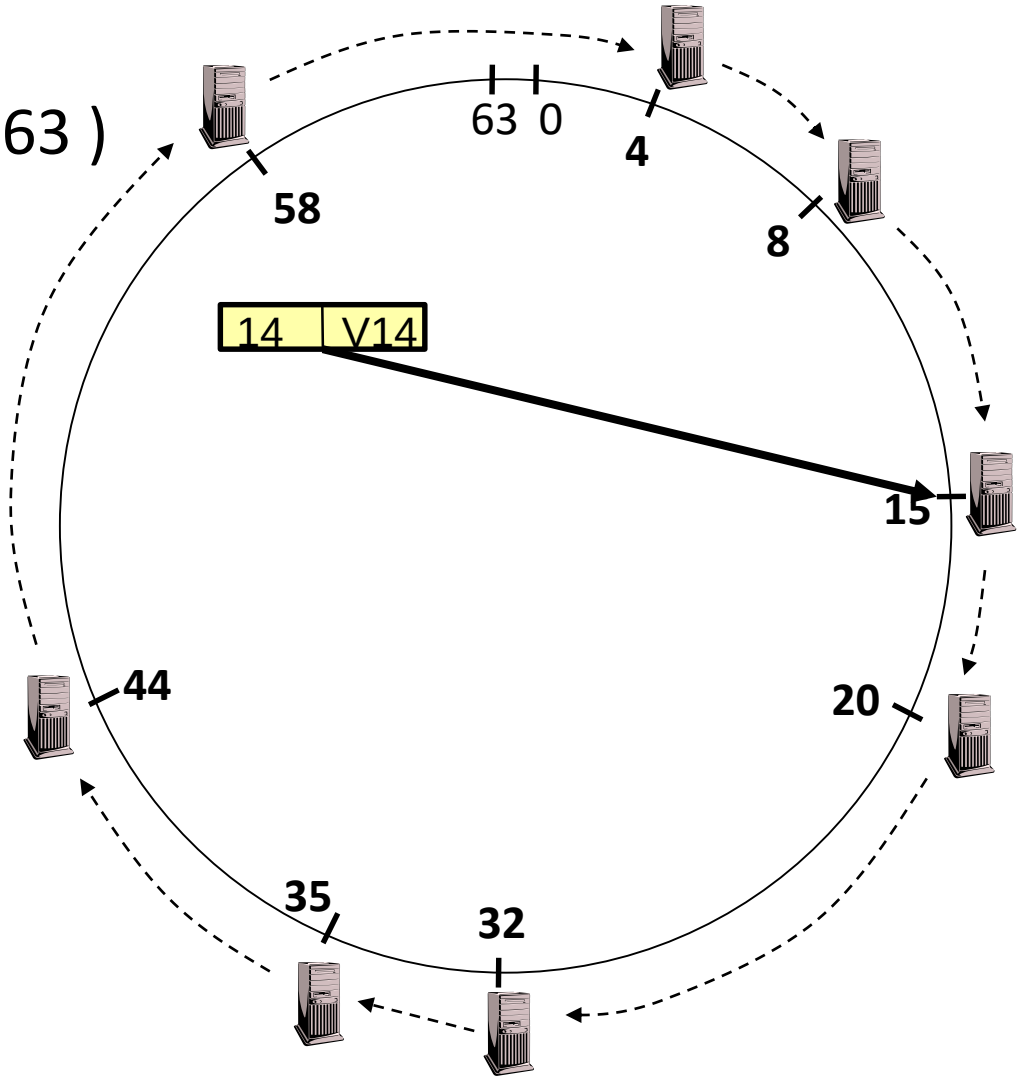
Scaling the Directory

- 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)



Scaling the Directory even further…

- For the directory to hash a key to a node, it must know the IDs of ***all*** nodes in the system
  - Still a central point of failure!

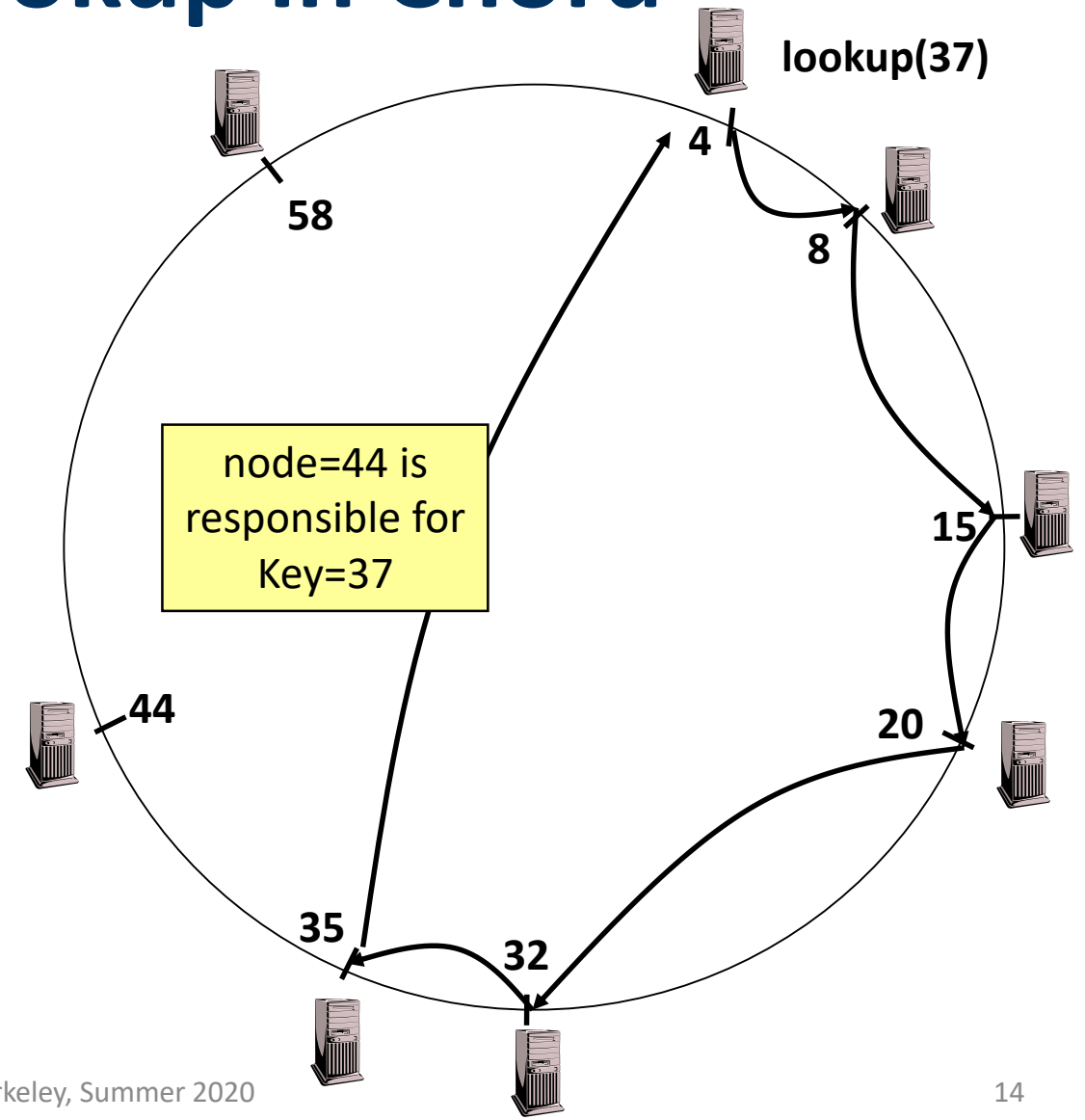
- Can we design a distributed KV store where no node has to know about all the other nodes? Or even most of them?
  - Yes! The Chord DHT does this!
  - For N nodes, each node keeps track of log(N) others
  - Can look up a key in log(N) network round-trips
  - Directory is distributed among all nodes

Chord: Distributed Directory Service

- Import aspect of the design space:
  - Decouple correctness from efficiency
  - Combined *Directory* and *Storage*
- Properties
  - Correctness:
    - Each node needs to know about neighbors on ring (one predecessor and one successor)
    - Connected rings will perform their task correctly
  - Performance:
    - Each node needs to know about O(log(*M*)), where *M* is the total number of nodes
    - Guarantees that a tuple is found in O(log(*M*)) steps
- Many other *Structured, Peer-to-Peer* lookup services:
  - CAN, Tapestry, Pastry, Bamboo, Kademlia, …
  - Several designed here at Berkeley!

Basic (Unoptimized) Lookup in Chord

- Each node maintains pointer to its successor
- Route packet (Key, Value) to the node responsible for ID using successor pointers
  - E.g., node=4 lookups for node responsible for Key=37
- Worst-case (correct) lookup is O(n)
  - But with optimizations (next slide), we can achieve an O(log n) lookups in the normal case

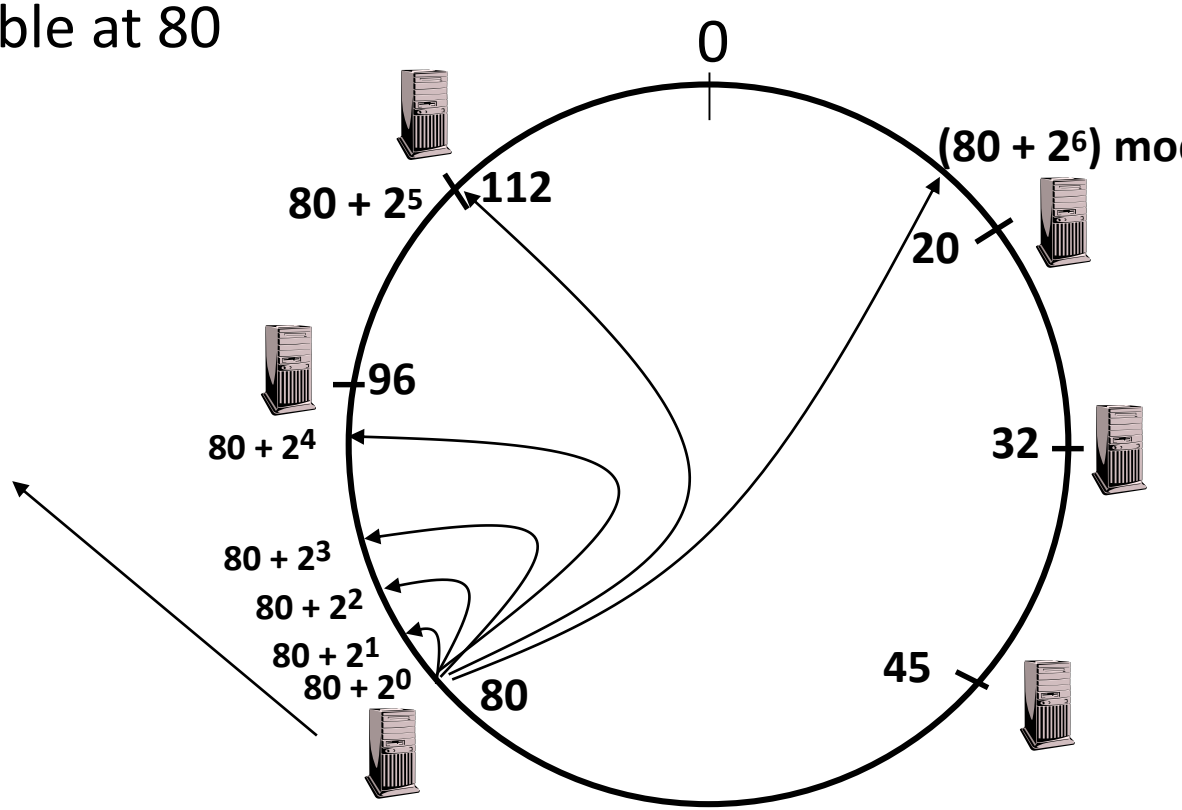


Efficient Lookup in Chord: Finger Tables

Say *m=7*

Finger Table at 80

i	*ft[i]*
0	96
1	96
2	96
3	96
4	96
5	112
6	20



"Short-circuit" the lookup by following the fingers (chords) in the circle where possible

ith entry at peer with id n is first peer with id >= $n + 2^i \pmod{2^m}$

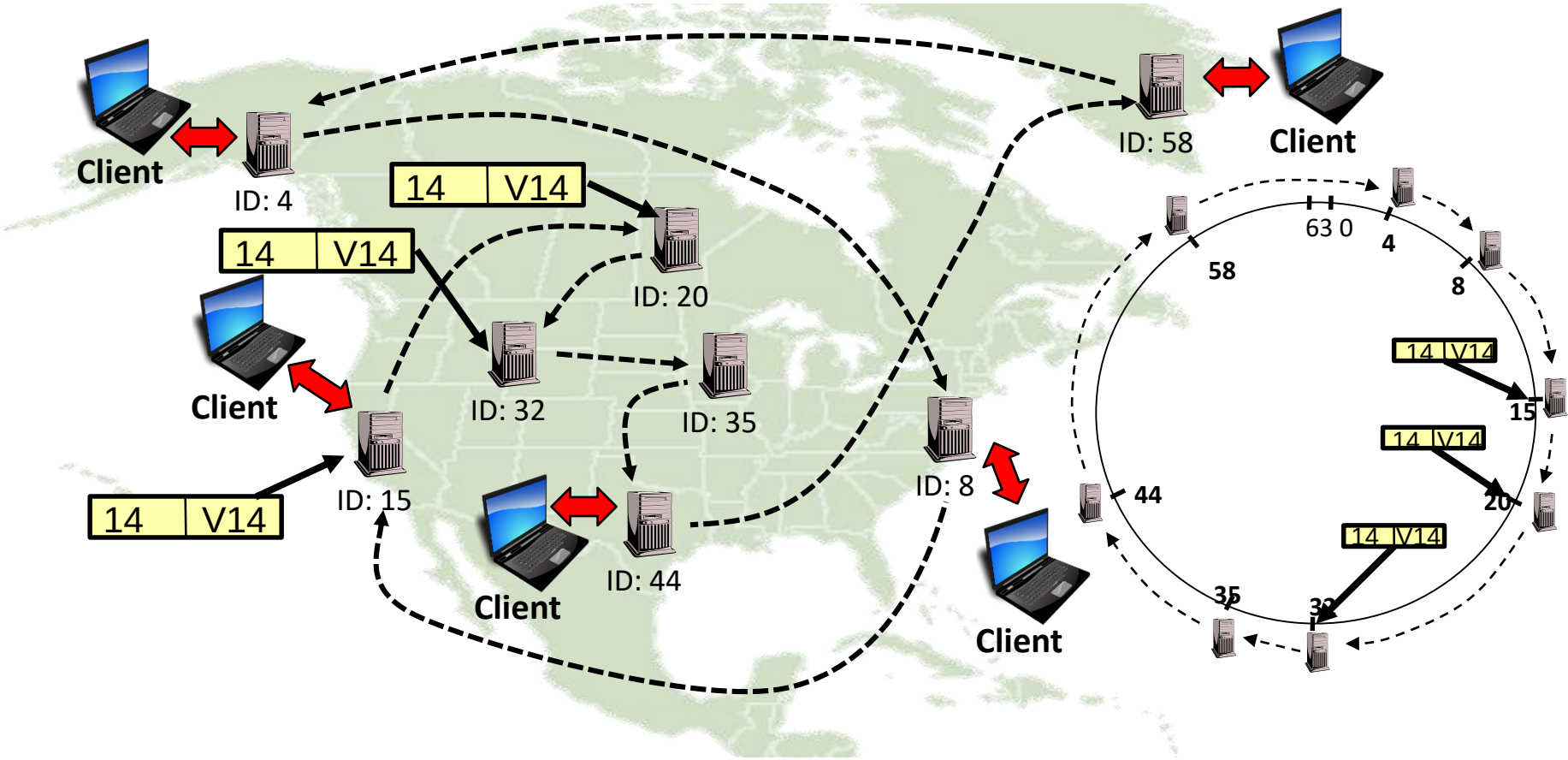
Various Other Measures in Chord

- Robustness in the face of fault-tolerance
  - Remember not just the successor, but the next k successors
  - Store key:value not just on the node that results from lookup, but also the next few successors…

- What if a new node joins?
  - Use directory mechanism to “look up” its place on the ring and find its predecessor and successor
  - Coordinate with its new successor and predecessor to notify them of its presence
  - Obtain a copy of the data it’s supposed to hold

Replication in Physical Space

- Replicating in adjacent nodes of virtual space → geographic separation in physical space
  - Avoids single points of failure through randomness



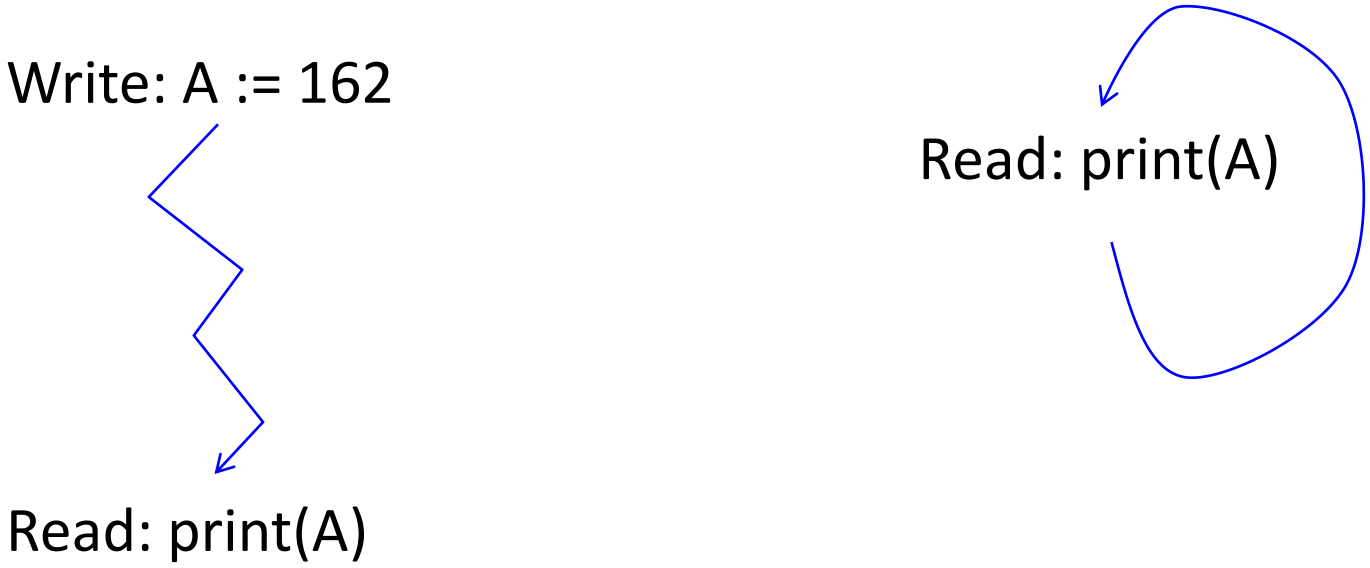
Consistency in Shared Storage

The Shared Storage Abstraction

- Information (and therefore control) is communicated from one point of computation to another by
  - The former storing/writing/sending to a location in a shared address space
  - And the second later loading/reading/receiving the contents of that location

- Memory (address) space of a process
- File systems
- Dropbox, …
- Google Docs, …
- Facebook, …

Example (think of Threads on a Single Node)

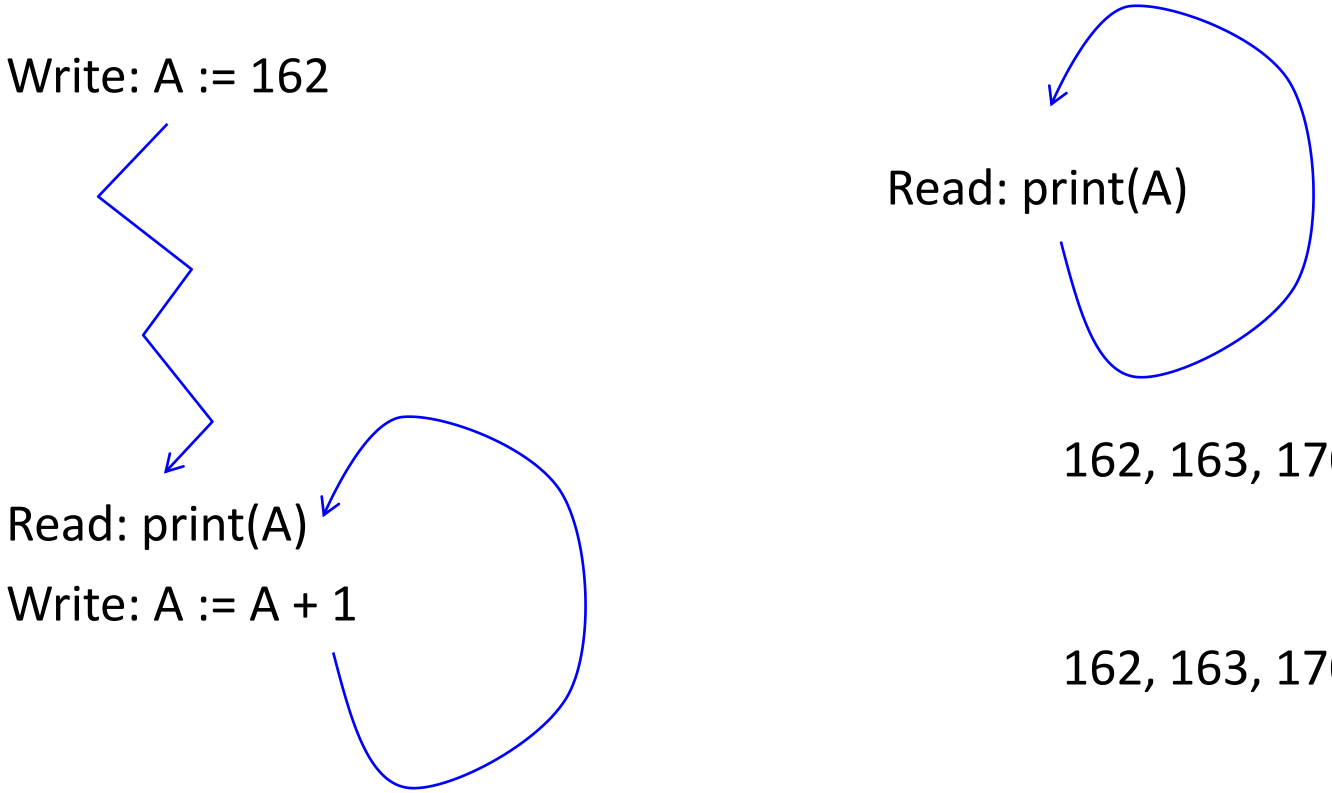


Based on Single-Node Experience, what do we Expect?

- Within a sequential thread, a read following a write returns the value written by that write

- Each write will eventually become visible to other readers

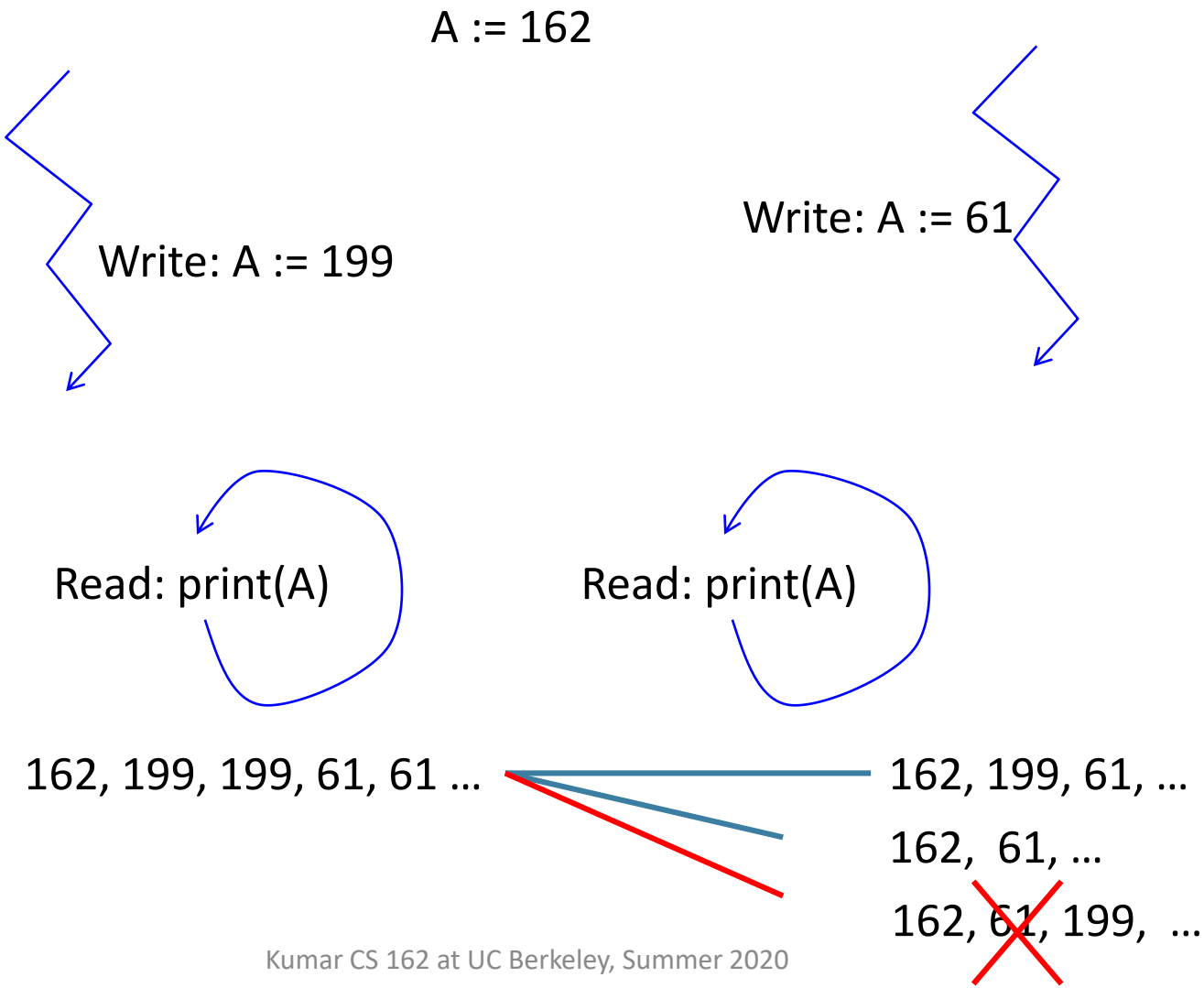
Example (think of Threads on a Single Node)



Based on Single-Node Experience, what do we Expect?

- Within a sequential thread, a read following a write returns the value written by that write
- Each write will eventually become visible to other readers
- A sequence of writes will be visible in order

Example (think of Threads on a Single Node)



Based on Single-Node Experience, what do we Expect?

- Within a sequential thread, a read following a write returns the value written by that write
- Each write will eventually become visible to other readers
- A sequence of writes will be visible in order
- All readers see a consistent order
  - It is as if there exists a canonical order of the operations, and each party takes samples

“Strong” Consistency

- **It is as if there exists a canonical order of the operations, and each party takes samples**

- When each operation is a read or write to a single object, this is called **Linearizability**
  - Linearizability also requires the canonical order to be consistent with real time order, up to concurrent operations

- When each operation is a transaction over multiple objects, this is called **Serializability**
  - “Strict Serializability” if the canonical order is consistent with real time

Consistency: Discussion

- Does the POSIX file I/O abstraction provide consistency (linearizability)?
  - Effectively, yes: "POSIX requires that a ***read****(2)* which can be proved to occur after a **write**() has returned returns the new data." (see `man 2 write`)
- Does NFS provide consistency (linearizability)?
  - No; reads could (temporarily) return stale data
- What about quorum consensus?

Is Quorum Consensus *really* Consistent?

- What if a GET and PUT happen concurrently?
  - GET may see either old value or new value
  - Still linearizable (!)
- What if you fail in the middle of a PUT?
  - For recursive queries, directory node can "finish" the PUT when it comes back up (write to log…)
- What about iterative queries?
  - Make each PUT a transaction over the nodes --- hold locks so that GETs can't proceed until PUT has committed (expensive!)
  - Hold up GETs until all R quorum members agree (not a perfect solution)
  - **Trade off consistency for availability???**

Announcements

- Work on Project 3

- Homework 5B deadline extended to Monday
- Homework 5A due tomorrow

- Guidance on calculating grades posted on Piazza

Course Evaluations

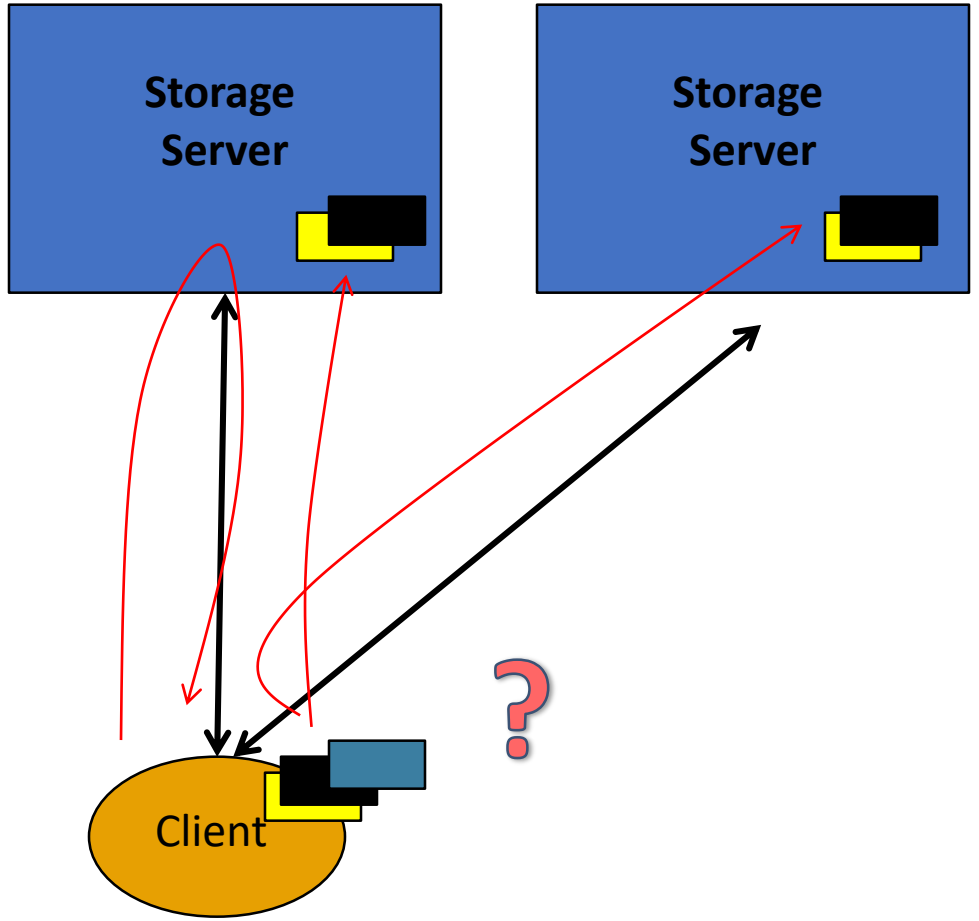
- https://course-evaluations.berkeley.edu/
- Fill out the course evaluation now!
- Class will resume in 15 minutes

- Incentive: If at least **70%** of the class submits the course evaluation form, then all students in the class will receive **1 point** of extra credit
  - That's 46 students (out of 65 enrolled)

Unfinished Business: Multiple Servers

- What happens if the client cannot update all the replicas?
- Availability → Inconsistency

- 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
  - Durability: transaction results persist in the face of failures



Distributed Consensus

- 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?

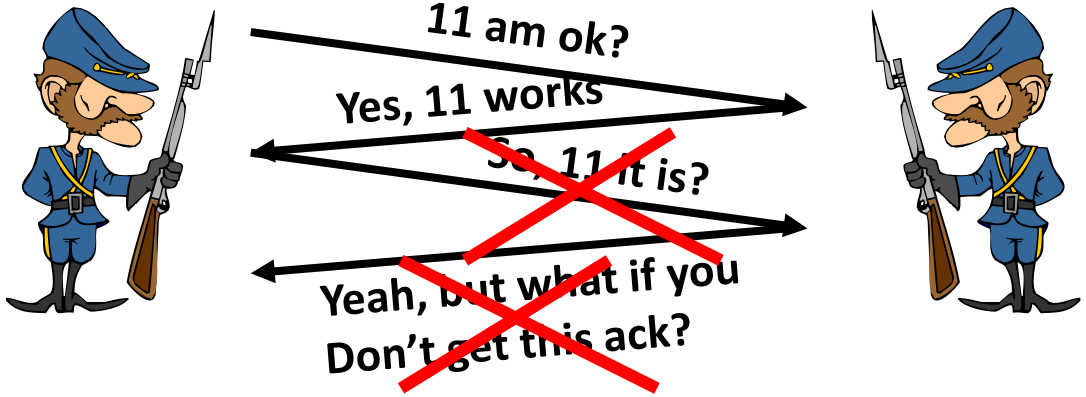
Two Generals' Problem (Generals' Paradox)

- Constraints of problem:
  - Two generals, on separate mountains
  - Can only communicate via messengers
  - Messengers can be captured
- Problem: need to coordinate attack
  - If they attack at different times, they all die
  - If they attack at same time, they win

- Difficulty: how do you every know your message got through?
  - Requires a message … that might not get through

Two Generals' Problem (Generals' Paradox)

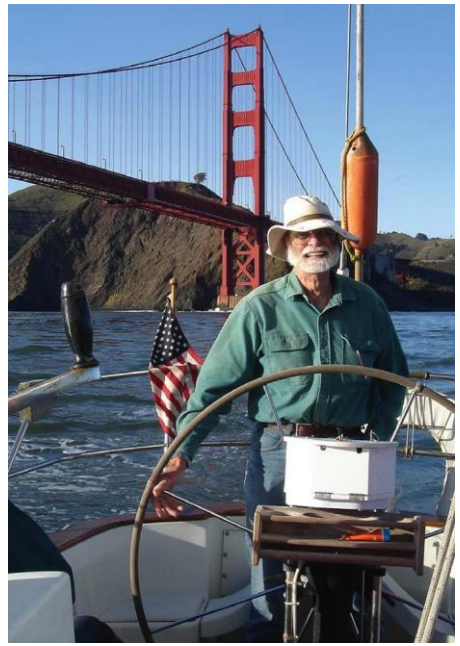
- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, "no", even if all messages get through



  - No way to be sure last message gets through!
- So, clearly, we need something other than simultaneity!

Two-Phase Commit (2PC)

- Distributed transaction: Two or more machines agree to do something, or not do it, atomically
  - No constraints on time, just that it will eventually happen!
- Two-Phase Commit protocol: Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)
- High-level problem statement
  - If no node fails and all nodes are ready to commit, then all nodes **COMMIT**
  - Otherwise **ABORT** at all nodes



Jim Gray

2PC Algorithm

- One coordinator
- N workers (replicas)
- Each machine has a durable log

- 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

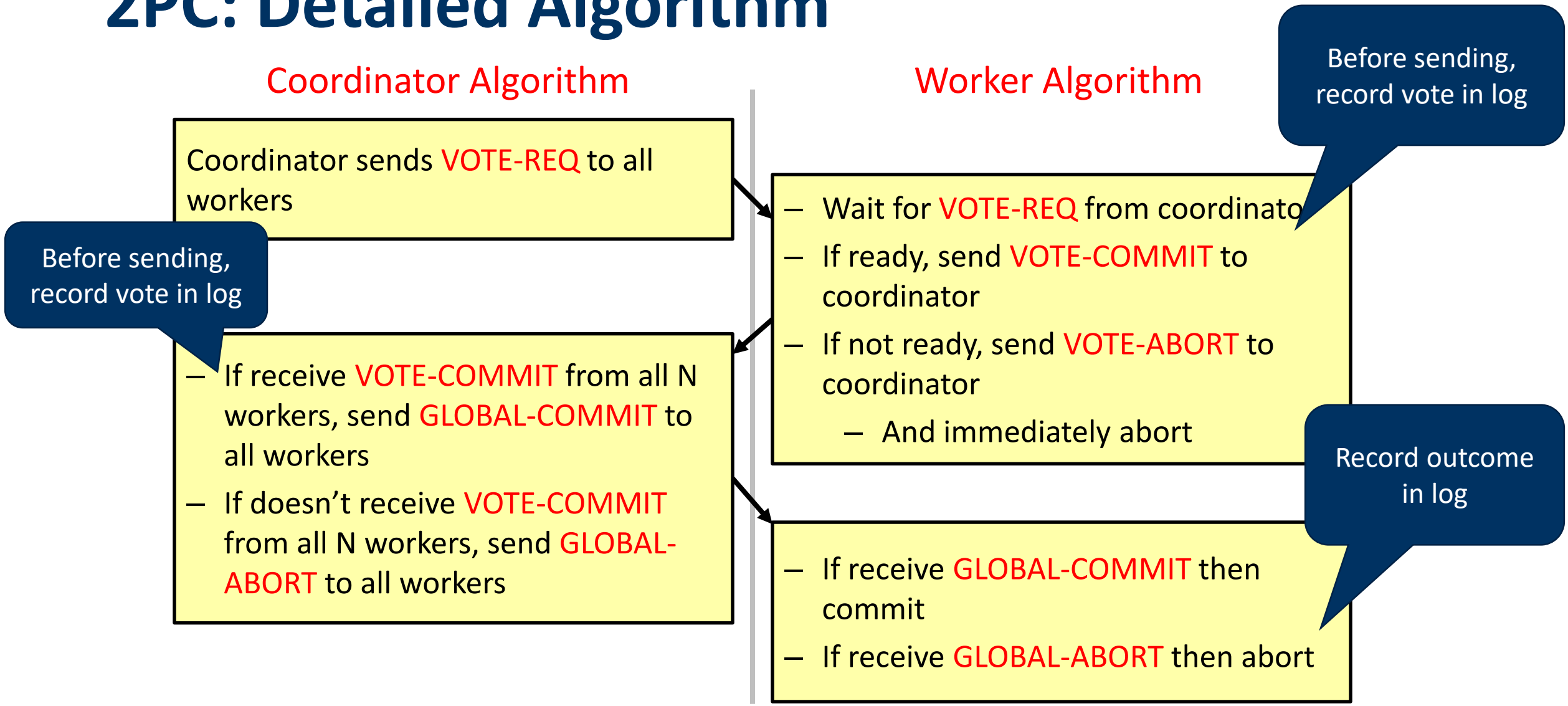
2PC: Messages

- Coordinator → Worker
  - VOTE-REQ
- Worker → Coordinator
  - VOTE-COMMIT
  - VOTE-ABORT
- Coordinator → Worker
  - GLOBAL-COMMIT
  - GLOBAL-ABORT

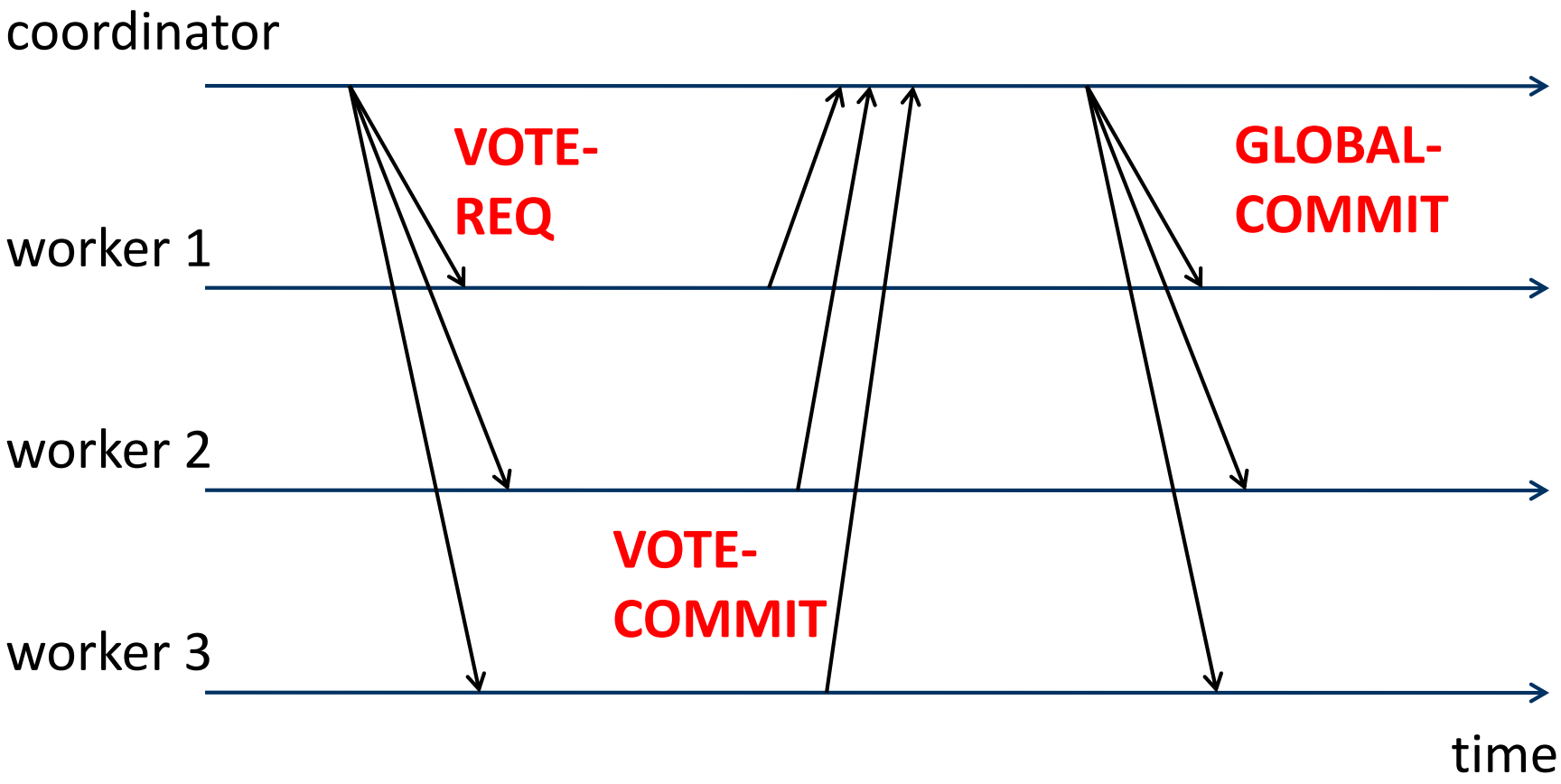
No taking back: always logged before sending

Also log transaction results (COMMIT or ABORT)

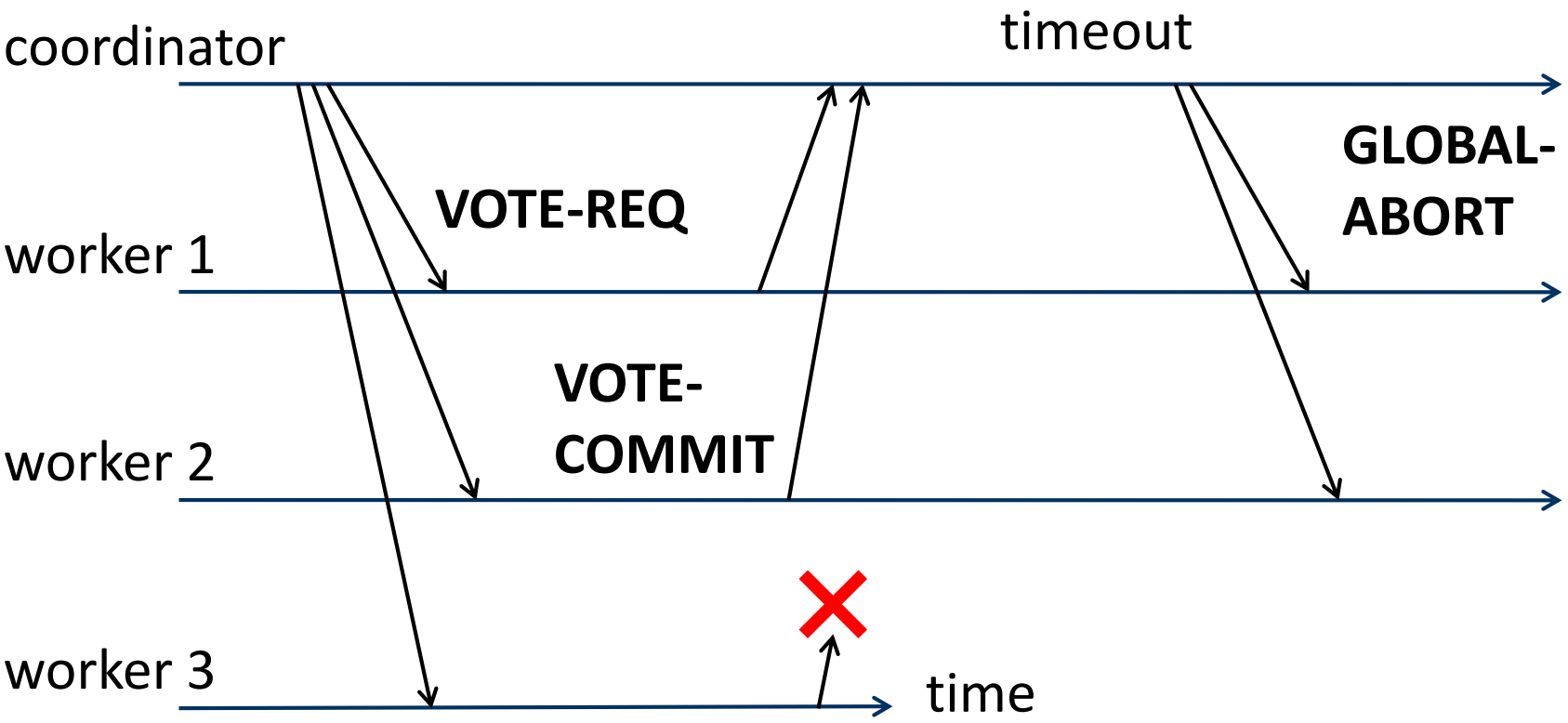
2PC: Detailed Algorithm



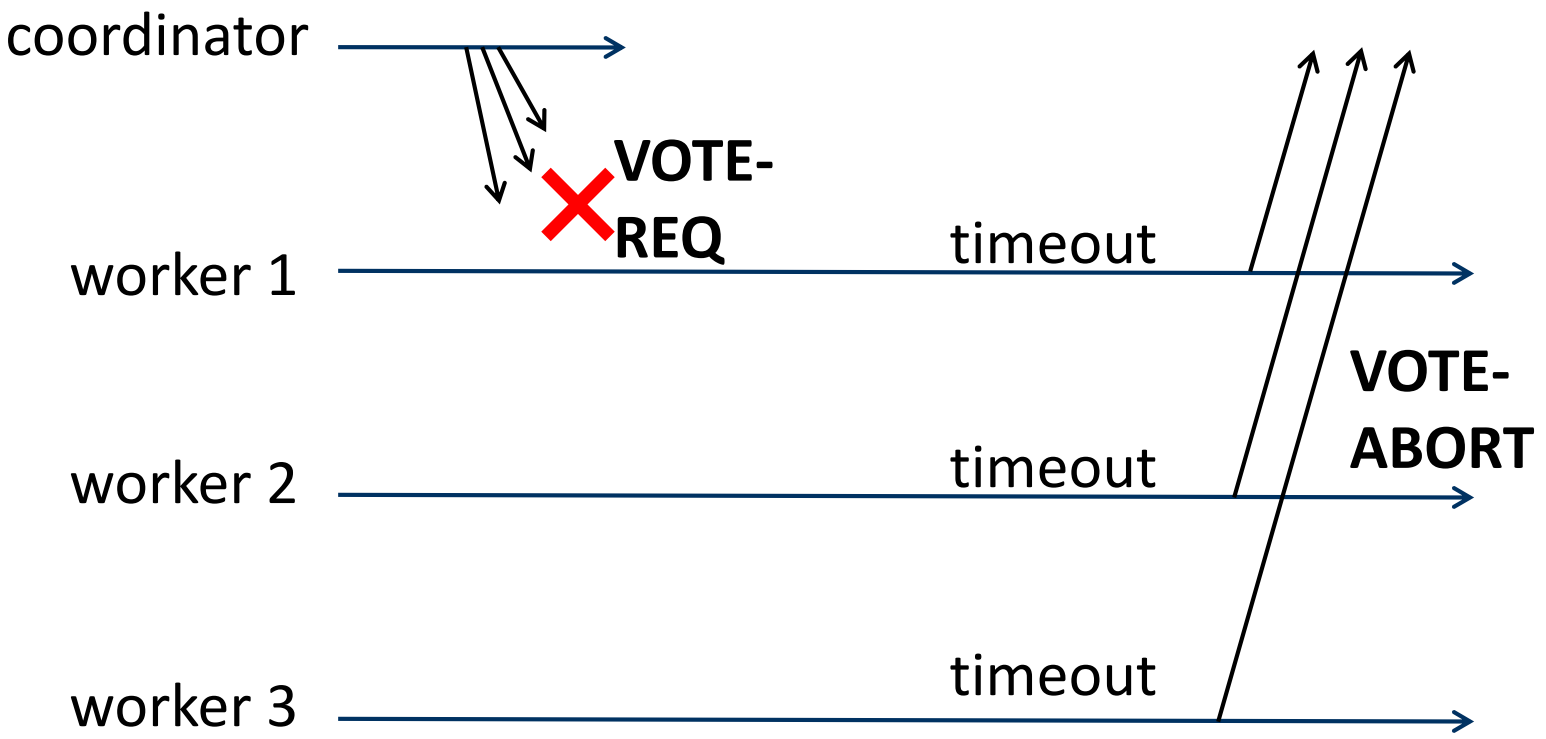
Example of Failure-Free 2PC



Example of Worker Failure in 2PC



Example of Coordinator Failure in 2PC



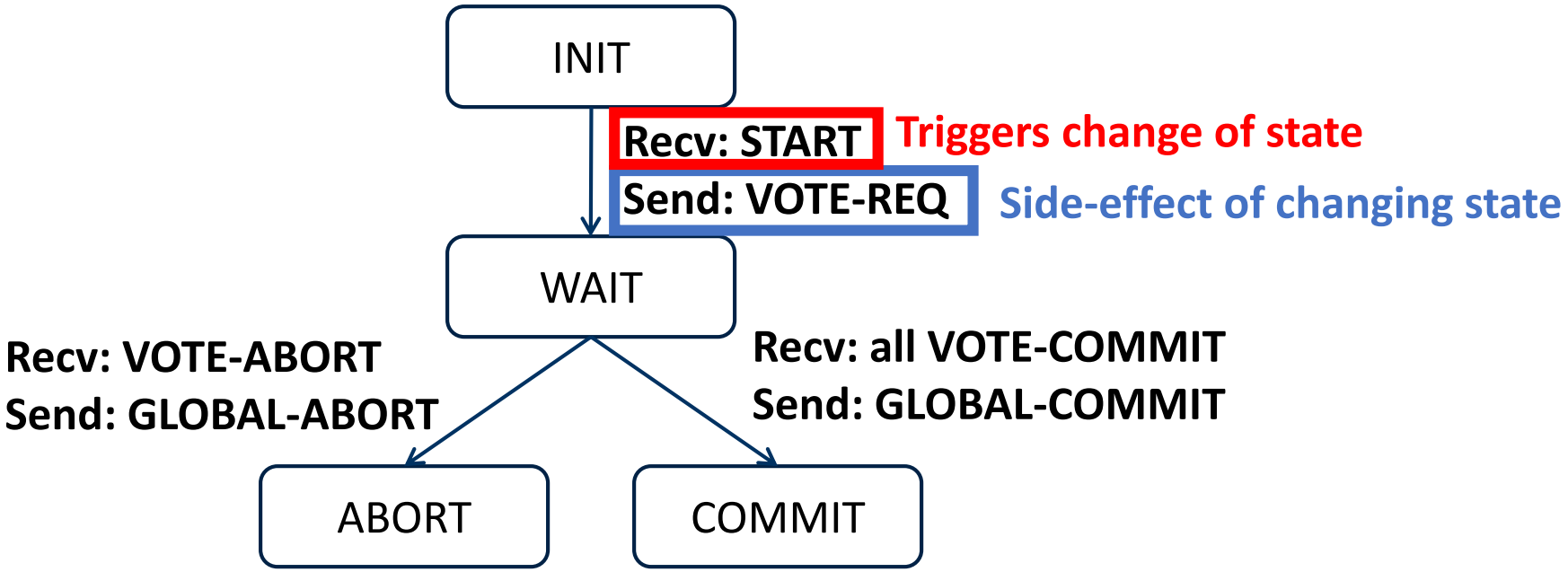
State Machines

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

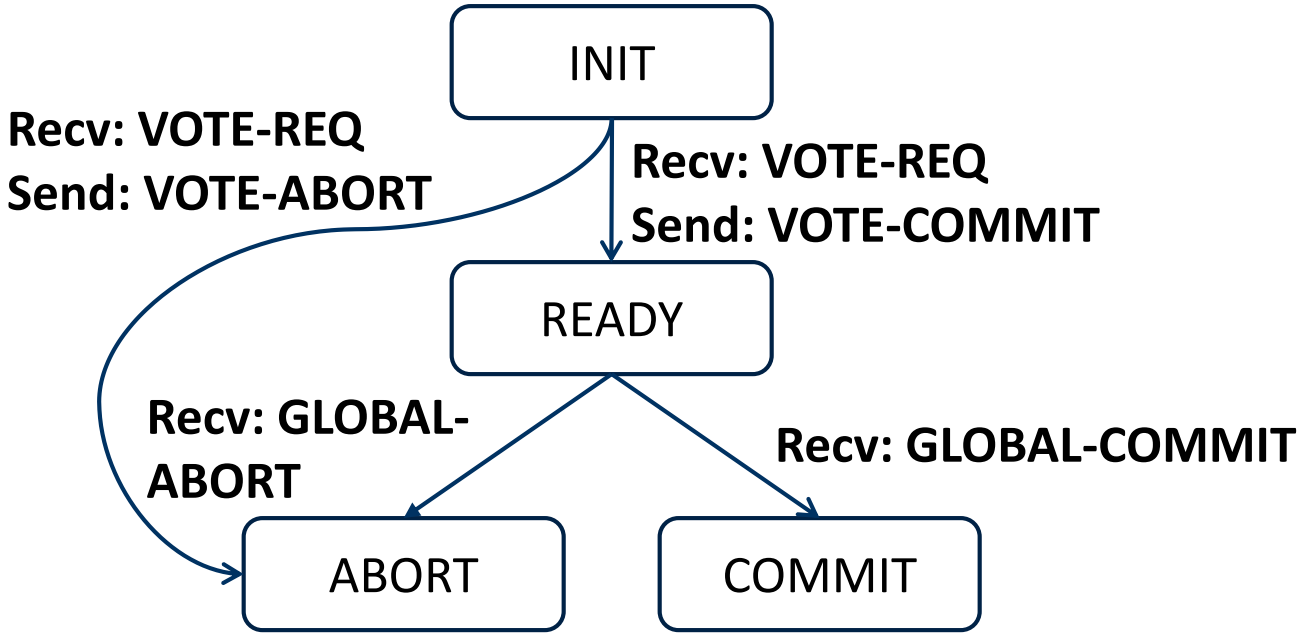
- One approach: **State Machine**
  - Every node is in a *state*
  - When the node receives a message (or timeout),
  - it *transitions* to another state and
  - Sends zero or more messages

- In hardware, state transition is atomic by design
- In software, we use mechanisms like a log to provide atomicity

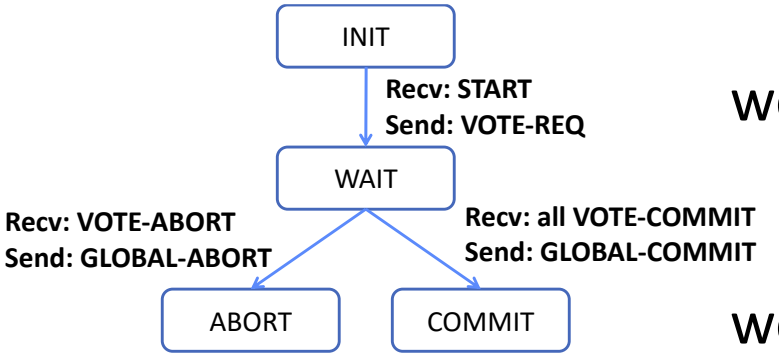
Coordinator's State Machine

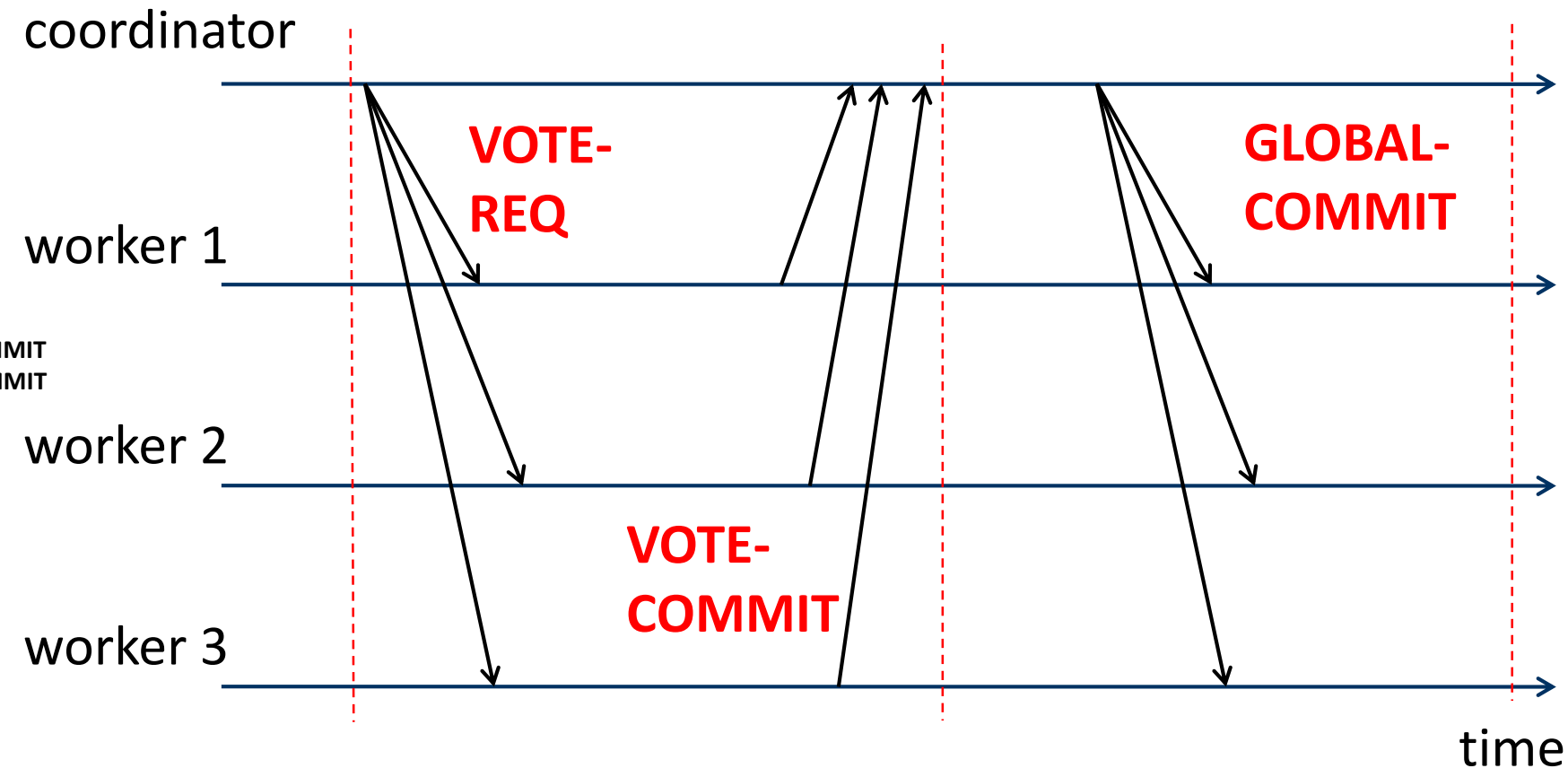


Worker's State Machine



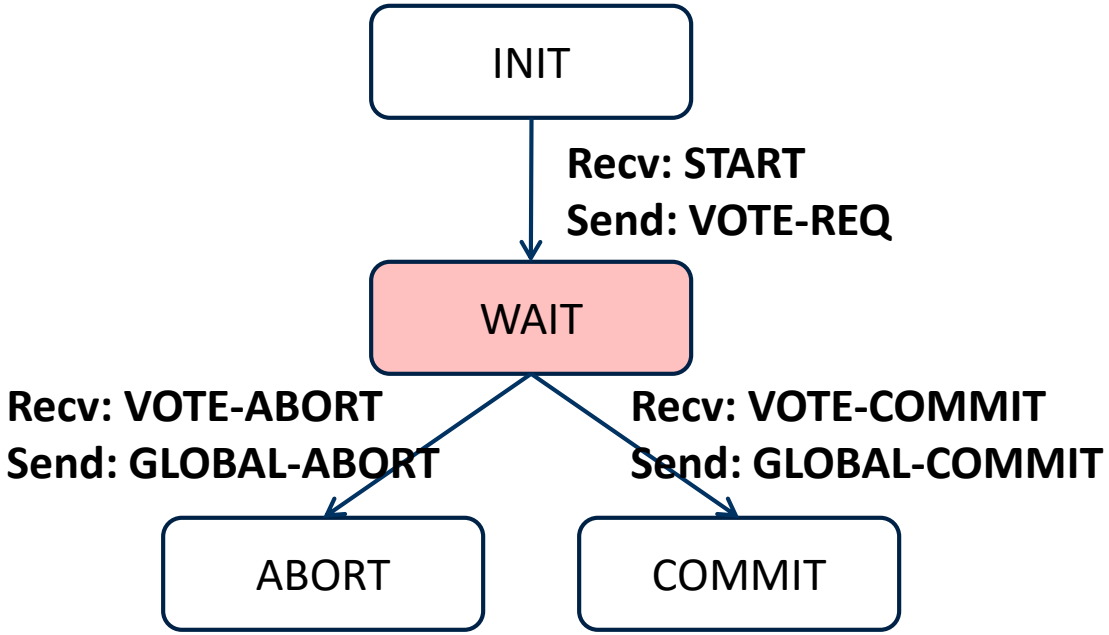
Example of Failure-Free 2PC with State





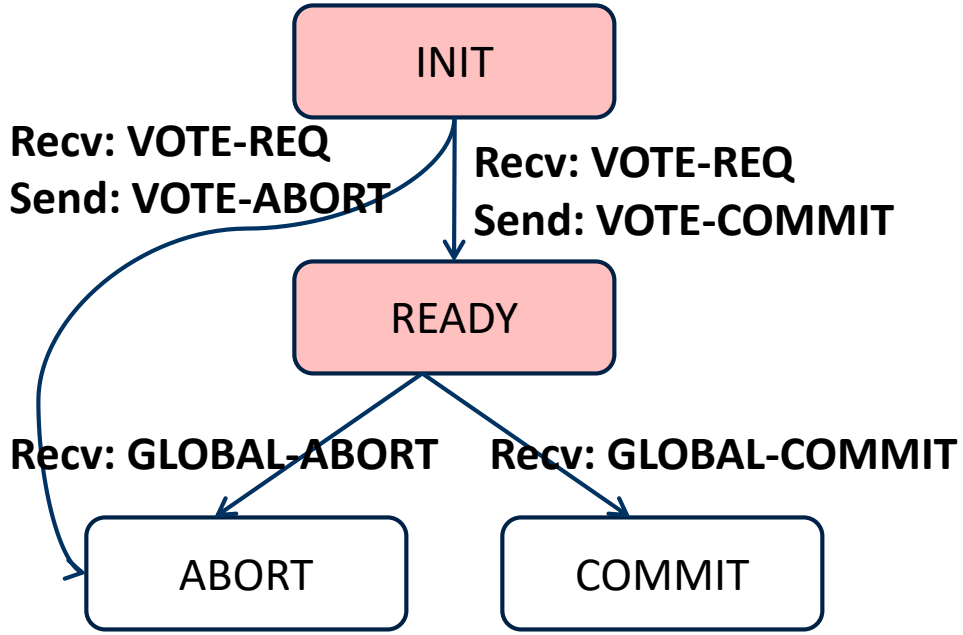
Dealing with Worker Failures

- 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



Dealing with Coordinator Failure

- Worker waits for VOTE-REQ in INIT
  - Worker can time out and abort
- Worker waits for GLOBAL message in READY
  - Workers must BLOCK waiting for coordinator to recover
  - Workers could try to consult each other
    - If one of them in COMMIT or ABORT state, then they all know the result
    - In another worker is still in INIT, it is safe to ABORT
    - If all of them are in the READY state, then they must still BLOCK
  - What if coordinator and worker both fail?
    - Workers can consult each other, but can't come to any decision
    - This motivates **Three-Phase Commit (3PC)**



Distributed Consensus

- 2PC (and 3PC) make a decentralized decision
  - E.g., changing the value of a key among all replicas for the key

- But they are hardly the only solutions to this problem!

Summary: Consistency

- Strong Consistency (Linearizability and Serializability)
  - Equivalent to sampling a canonical total order
  - Canonical total order should be consistent with wall-clock time, in the case of linearizability
- Consensus: 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: 2PC

- 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 lockstep (for an item)
  - Don’t perform new transactions until old one is resolved
  - Stall until transaction is resolved