CS162: Operating Systems and Systems Programming

Lecture 17: Transactions for Consistency

20 July 2015

Charles Reiss

https://cs162.eecs.berkeley.edu/

Recall: Distributed: Promise

Availability

- One machine goes down, *system* up

Durability

- One machine loses data, *system* does not

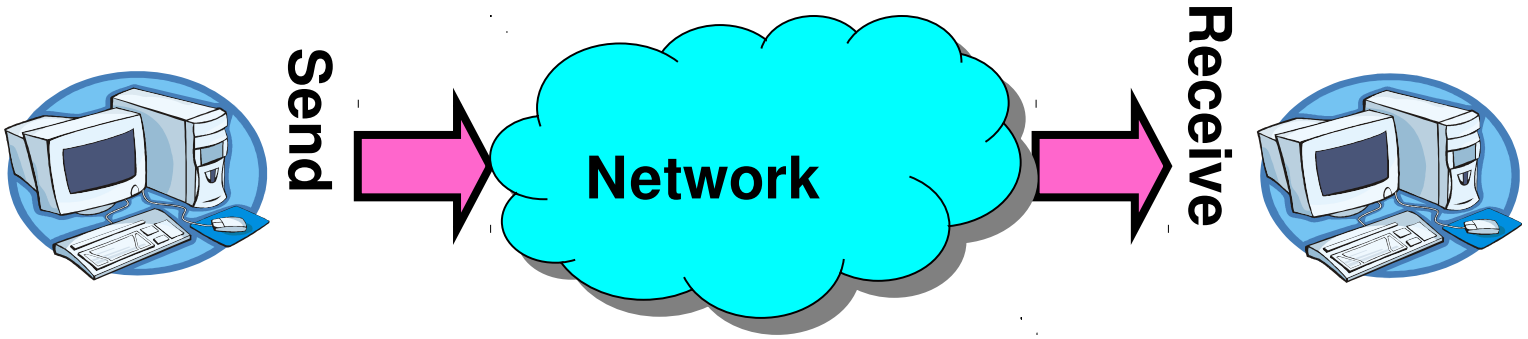
Security

- Divide security problem into simpler pieces?

Recall: The Mailbox Abstraction (2)

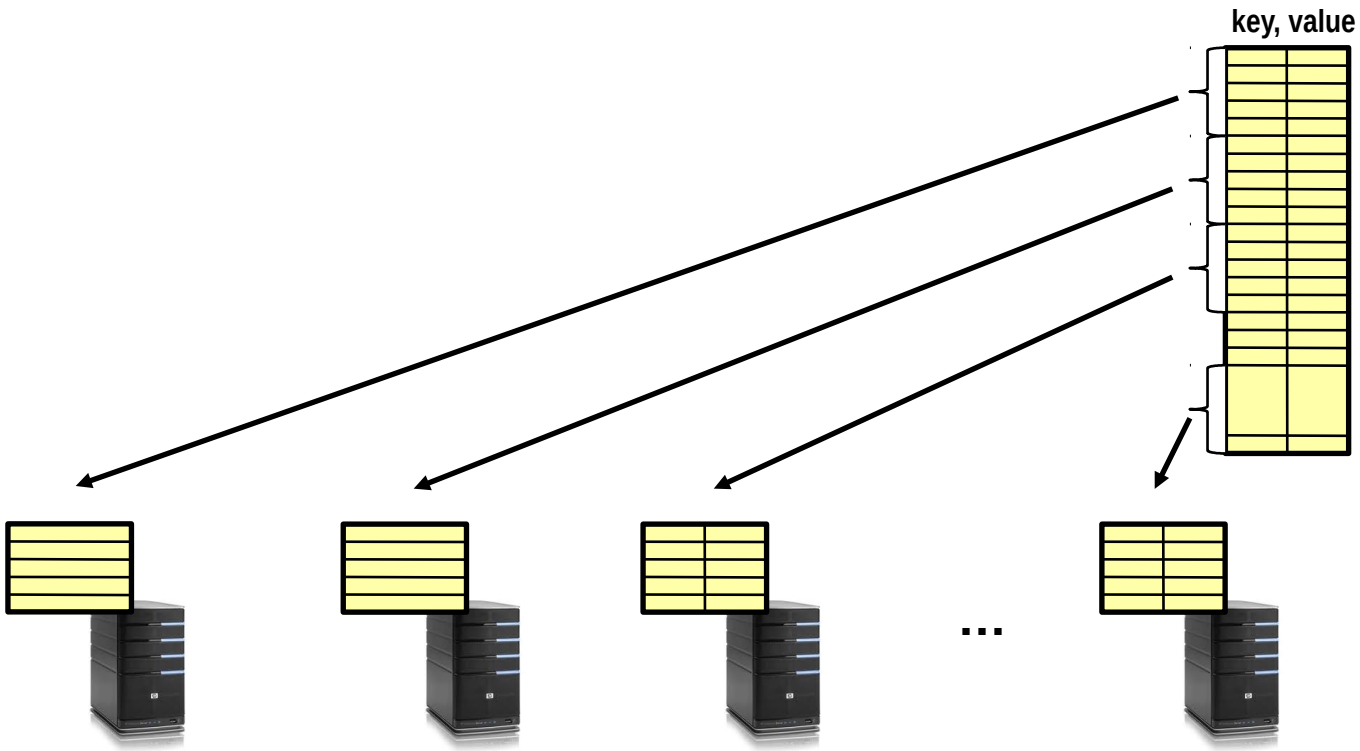
Interface:

- Mailbox (mbox): temporary holding area for messages
  - Includes both destination location and queue
- Send(message,mbox)
  - Send message to remote mailbox identified by mbox
- Receive(buffer,mbox)
  - Wait until mbox has message, copy into buffer, and return
  - If threads sleeping on this mbox, wake up one of them

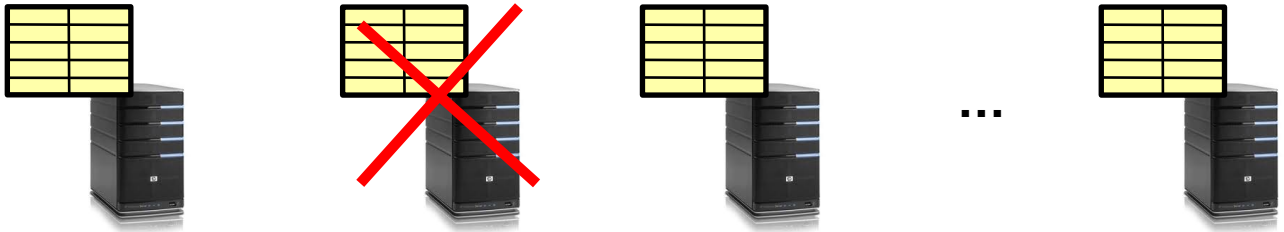


Recall: Key-Value Store

Main idea: partition set of key-values across many machines



Recall: 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

Consistency: maintain data consistency in face of node failures and message losses

Heterogeneity (if deployed as peer-to-peer systems):

- Latency: 1ms to 1000ms
- Bandwidth: 32Kb/s to 100Mb/s

Key Questions

put(key, value): where do you store a new (key, value) tuple?

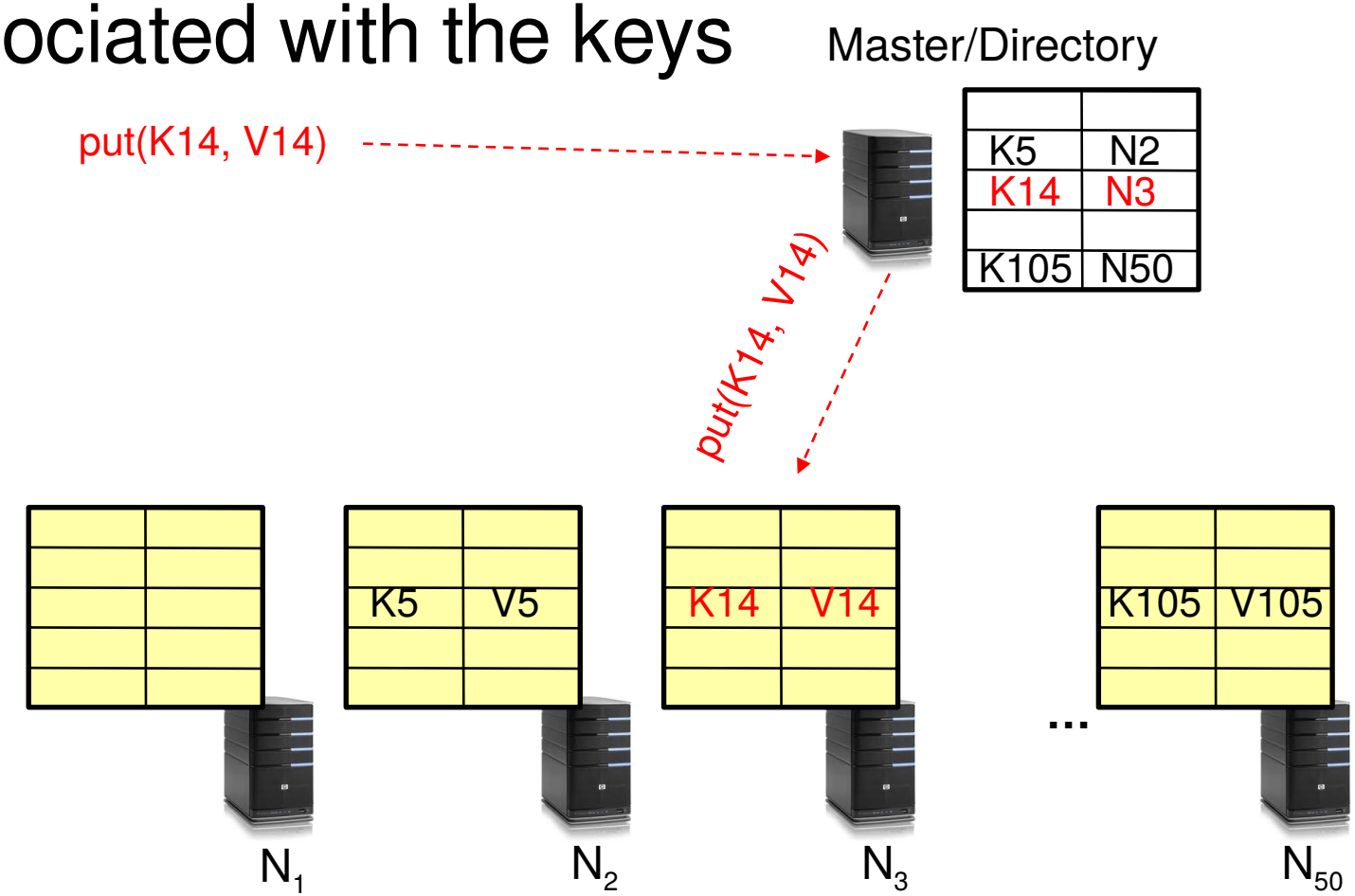
get(key): where is the value associated with a given “key” stored?

And, do the above while providing

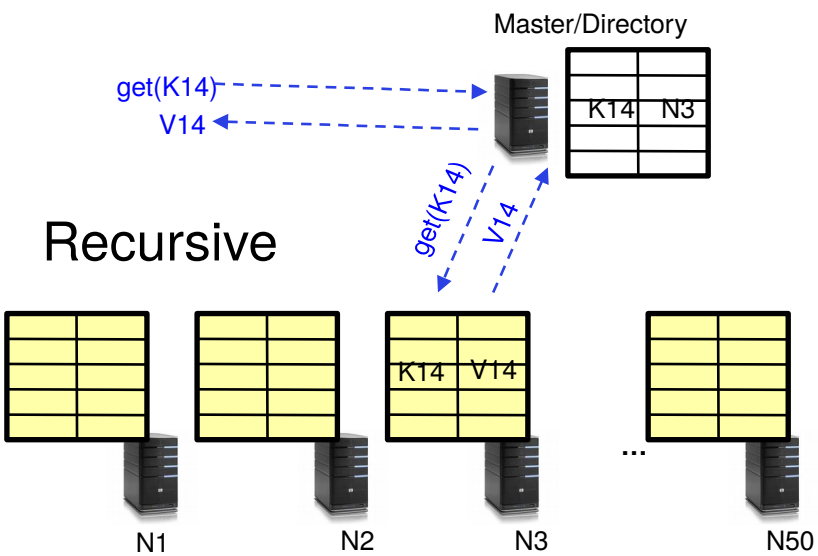
- Fault Tolerance
- Scalability
- Consistency

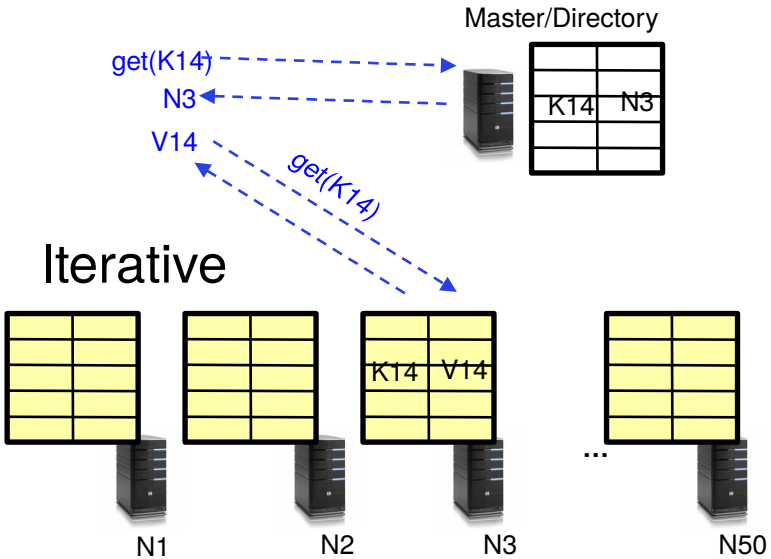
Directory-Based Architecture

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



Discussion: Iterative vs. Recursive Query





Recursive Query: Directory delegates

Iterative Query: Client delegates

Discussion: Iterative vs. Recursive Query

Recursive Query:

- Advantages:
  - Faster, as typically master/directory closer to nodes
  - Easier to maintain consistency, as master/directory can serialize put()s/get()s
- Disadvantages: scalability bottleneck, as all values go through master/directory

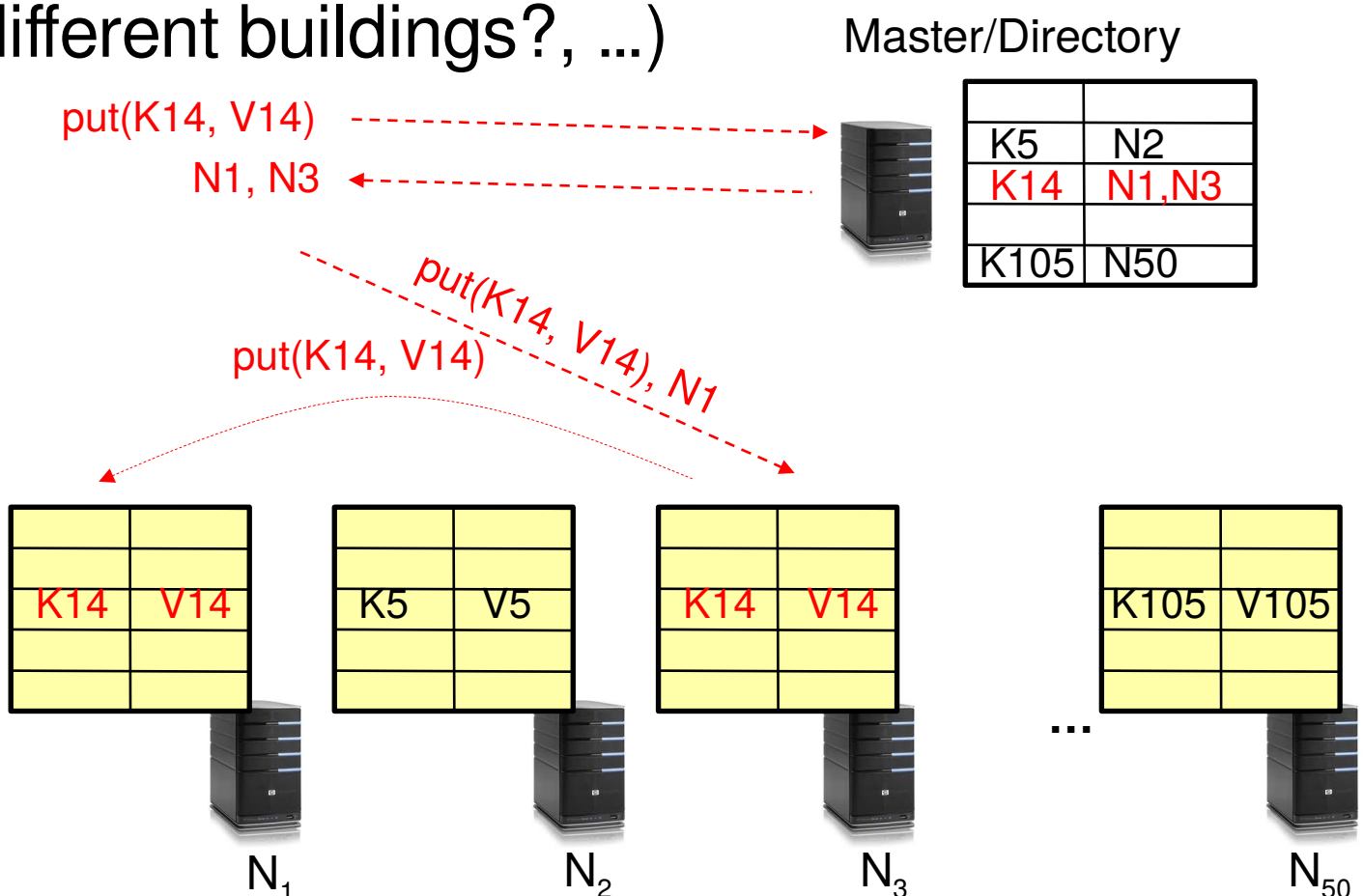
Iterative Query

- Advantages: more scalable
- Disadvantages: slower, harder to enforce data consistency

Fault Tolerance

Replicate value on several nodes

- Ideally r eplicas *physically separated* (different networking equipment, different power supplies, different buildings?, …)



Scalability

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 value on more nodes

Master/directory scalability:

- Replicate it
- Partition it, so different keys are served by different masters/directories
  - How do you partition?

Scalability: Load Balancing

Directory keeps track of the storage availability at each node

- Preferentially insert new values on nodes with more storage available

What happens when a new node is added?

- Cannot insert only new values on new node. Why?
- Move values from the heavy loaded nodes to the new node

What happens when a node fails?

- Need to replicate values from fail node to other nodes

Scaling Up Directory

Challenge:

- Directory contains a number of entries equal to number of (key, value) tuples in the system
- Can be tens or hundreds of billions of entries in the system!

Solution: **consistent hashing**

Associate to each node a unique id in an uni-dimensional space 0..2^m-1

- Partition this space across m machines
- Assume keys are in same uni-dimensional space
- Each (Key, Value) is stored at the node with the smallest ID larger than Key

Key to Node Mapping Example

m = 6 / ID space: 0..63

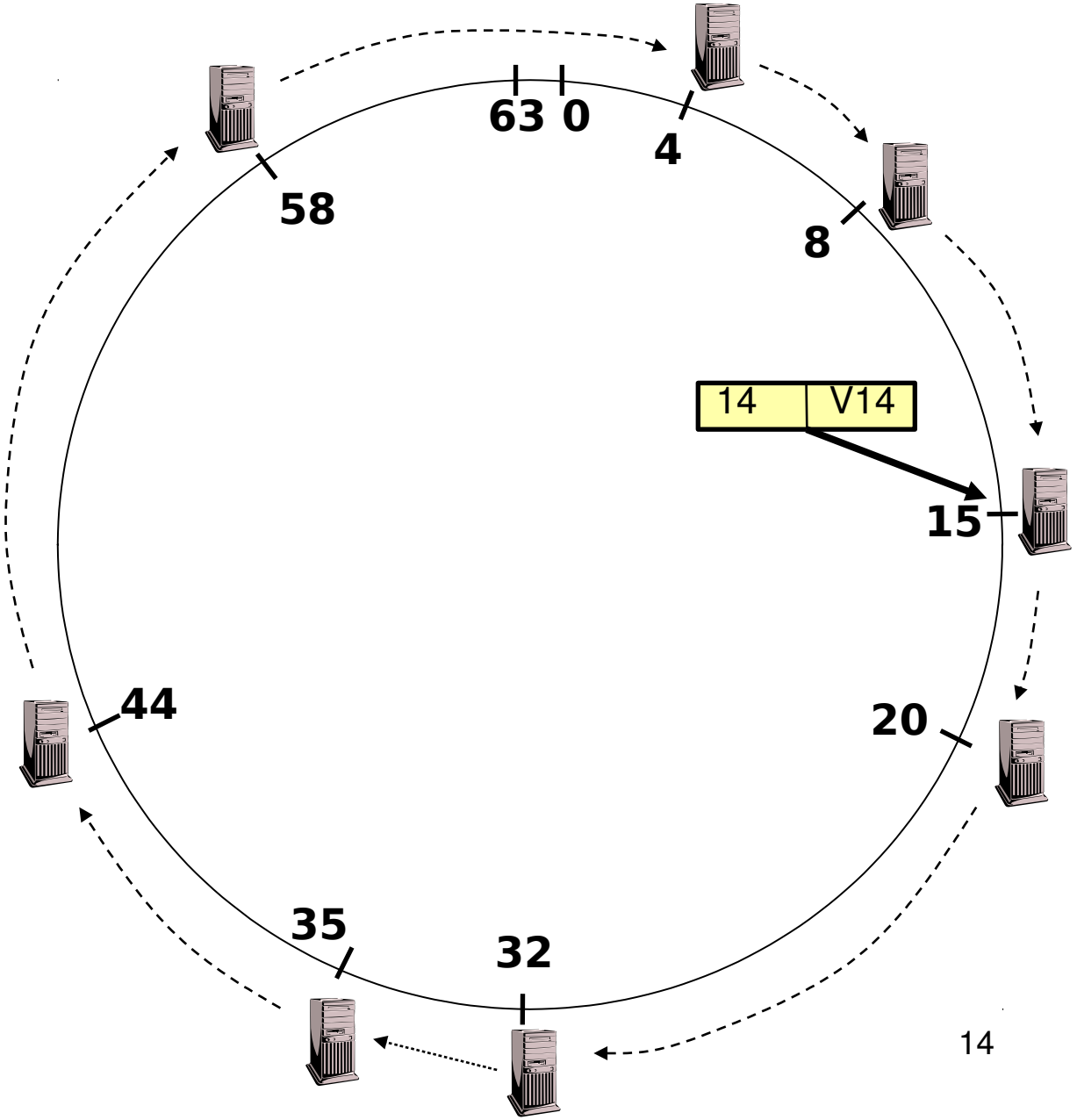
Node 8 maps keys [5,8]

Node 15 maps keys [9,15]

Node 20 maps keys [16, 20]

...

Node 4 maps keys [59, 4]



Lookup in Chord-like system (with Leaf Set)

Assign IDs to nodes

- Map hash values to node with closest ID

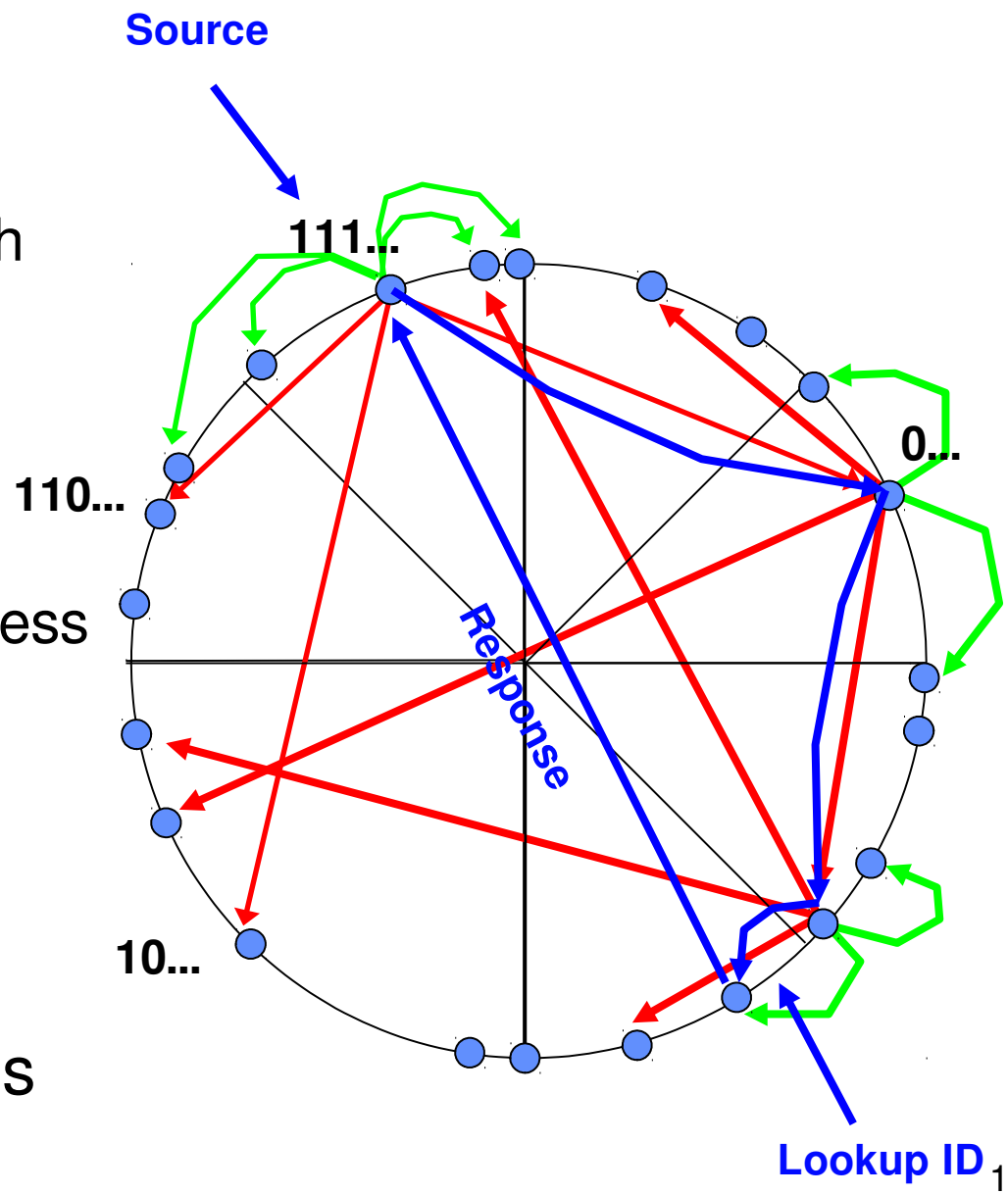
Leaf set is successors and predecessors

- All that's needed for correctness

Routing table matches successively longer prefixes

- Allows efficient lookups

Replicate to "adjacent" nodes



Chord network maintainence

Adding a node:

- New node chooses ID
- New node queries network for "neighbors"
  - This is the new node's **leaf set**
- Node announces to its adjacent nodes
  - They update their **leaf sets**

Fully decentralized

- Can start anywhere
- No decision-maker
- Still works if any node leaves network

"Distributed Hash Tables"

Popular in the early 2000s networking research

Designed **huge scale** + **completely decentralized**

Several variants – slightly different kinds of consistent hashing/"leaf sets"

- Chord, CAN, Tapestry, Pastry, Kademlia
- All published 2001-2

Largest (?) in existence: BitTorrent DHT

- Based on Kademlia design

DynamoDB

Amazon's key-value store

Uses consistent hashing (ring)

Central directory (replicated everywhere)

- Not targeting Internet-level scalability like Chord

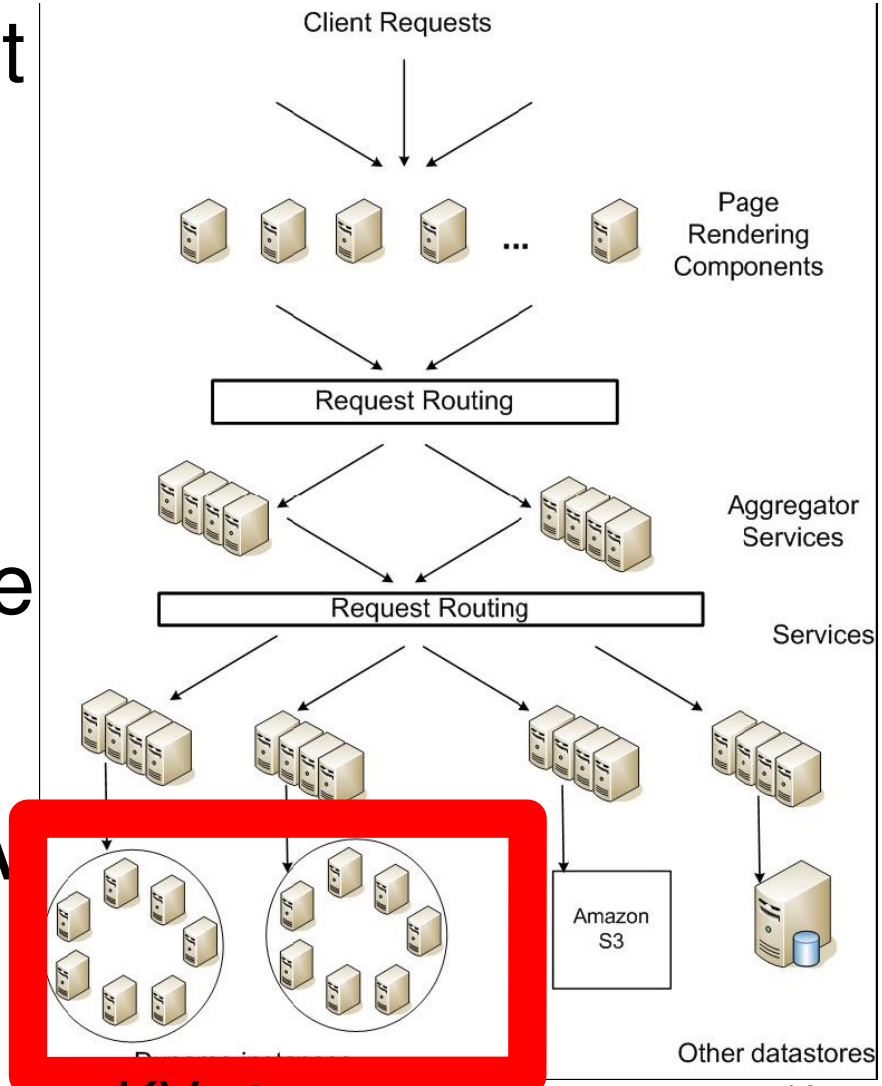
Composing services

We've scaled a hashtable, but that's not enough to do something useful alone...

Example: Amazon

Trees of services with multiple distinct KV stores

Each service has its own SLA (service level agreement)



KV stores instances

SLAs

How fast should you be should you be?

Example goal:

- **99.9th percentile** latency <300 ms

KV store reads:

- Speed of slowest replica with an answer

KV store writes:

- Speed of slowest replica that needs to ACK write

SLA composition

Why a high percentile? [From DynamoDB paper:]

Factor of 10

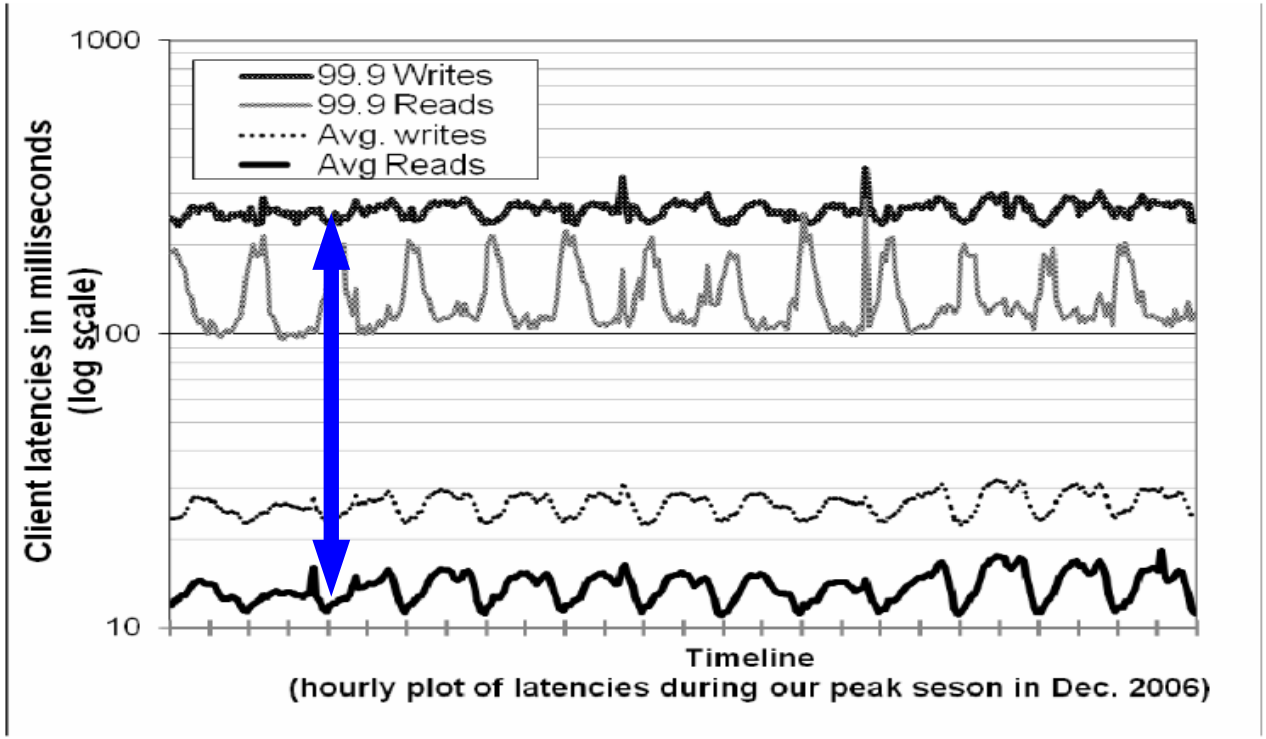


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

Consistency (1)

Need to make sure that a value is replicated correctly

How do you know a value has been replicated on every node?

Wait for acknowledgements from every node

Consistency (2)

What happens if a node fails during replication?

- Pick another node and try again

What happens if a node is slow?

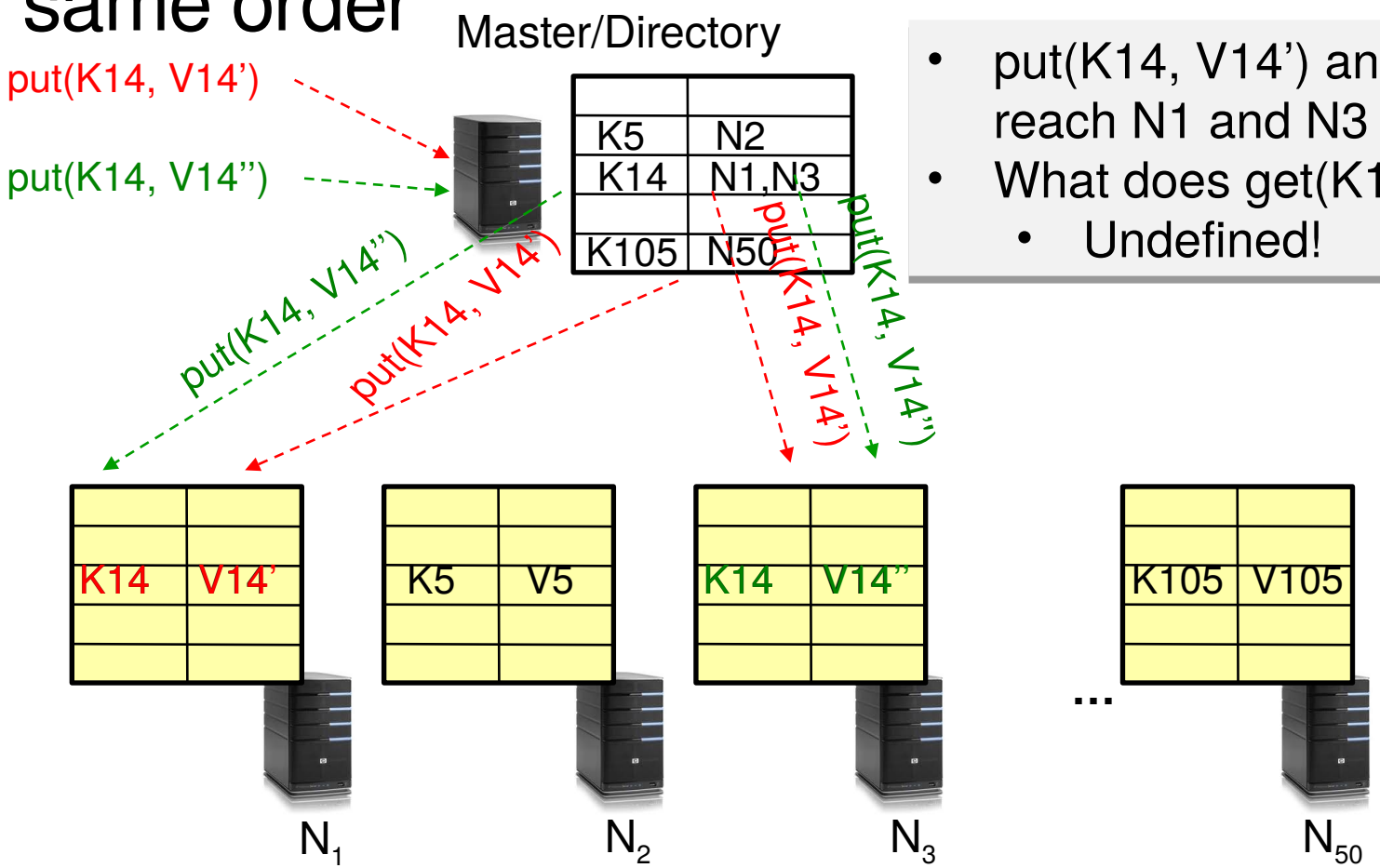
- Slow down the entire put()? Pick another node?

In general, with multiple replicas

- Slow puts and fast gets

Consistency (3)

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



- put(K14, V14') and put(K14, V14'') reach N1 and N3 in reverse order
- What does get(K14) return?
  - Undefined!

Consistency (More Generally)

Consistency also includes preserving other properties

Example:

- no money lost in the bank accounts
- if the second line of output of a program is in a file, then the first line is also
- ...

Application dependent:

- For key-value stores, we'll start with just caring about consistency within a particular key *only*

Quorum Consensus

Improve put() and get() operation performance

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

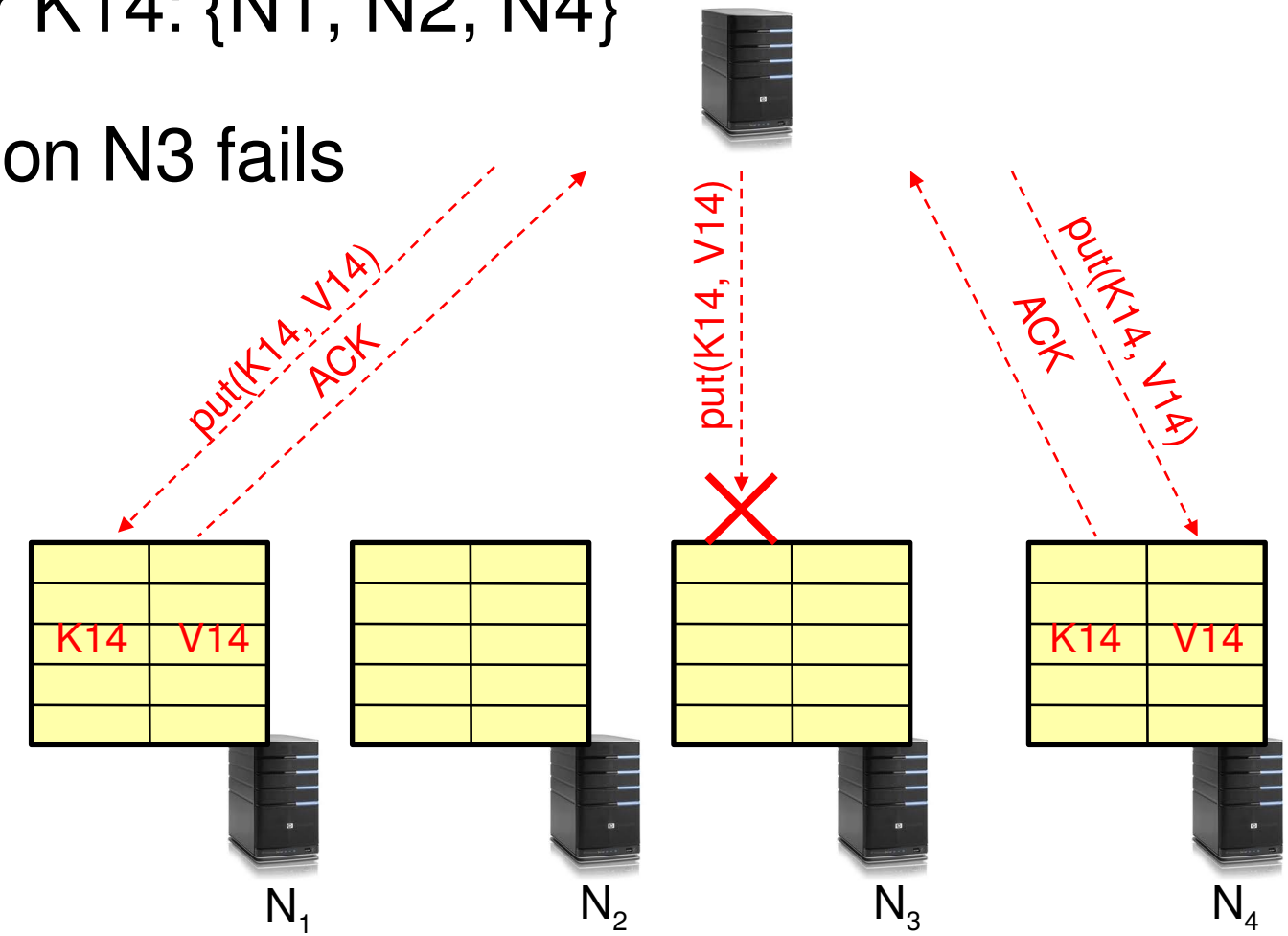
Why might you use $W+R > N+1$?

Quorum Consensus Example

N=3, W=2, R=2

Replica set for K14: {N1, N2, N4}

Assume put() on N3 fails



Transactions

Closely related to critical sections in manipulating shared data structures

Extend concept of atomic update from memory to atomic update of distributed system's state

Key concept: Transaction

An atomic sequence of actions (reads/writes) on a storage system (or database)

That takes it from one consistent state to another

Typical Structure

Begin a transaction – get transaction id

Do a bunch of updates

- If any fail along the way, roll-back

Commit the transaction

“Classic” Example: Transaction

```
BEGIN;    --BEGIN TRANSACTION
UPDATE accounts SET balance = balance - 100.00 WHERE
  name = 'Alice';
UPDATE branches SET balance = balance - 100.00 WHERE
  name = (SELECT branch_name FROM accounts WHERE name =
  'Alice');
UPDATE accounts SET balance = balance + 100.00 WHERE
  name = 'Bob';
UPDATE branches SET balance = balance + 100.00 WHERE
  name = (SELECT branch_name FROM accounts WHERE name =
  'Bob');


COMMIT;    --COMMIT WORK
```

Transfer $100 from Alice’s account to Bob’s account

The ACID properties of Transactions

Atomicity: all actions in the transaction happen, or none happen

Consistency: transactions maintain data integrity, e.g.,

- Balance cannot be negative
- Cannot reschedule meeting on February 30

Isolation: execution of one transaction is isolated from that of all others; no problems from concurrency

Durability: if a transaction commits, its effects persist despite crashes

The ACID properties of Transactions

Atomicity: all actions in the transaction happen, or none happen

Consistency: transactions maintain data integrity, e.g.,

- **Balance cannot be negative**
- **Cannot schedule two meetings at the same time**

Isolation: execution of one transaction is isolated from that of all others; no problems from concurrency

Durability: if a transaction commits, its effects persist despite crashes

Inconsistency with Quorum Consensus

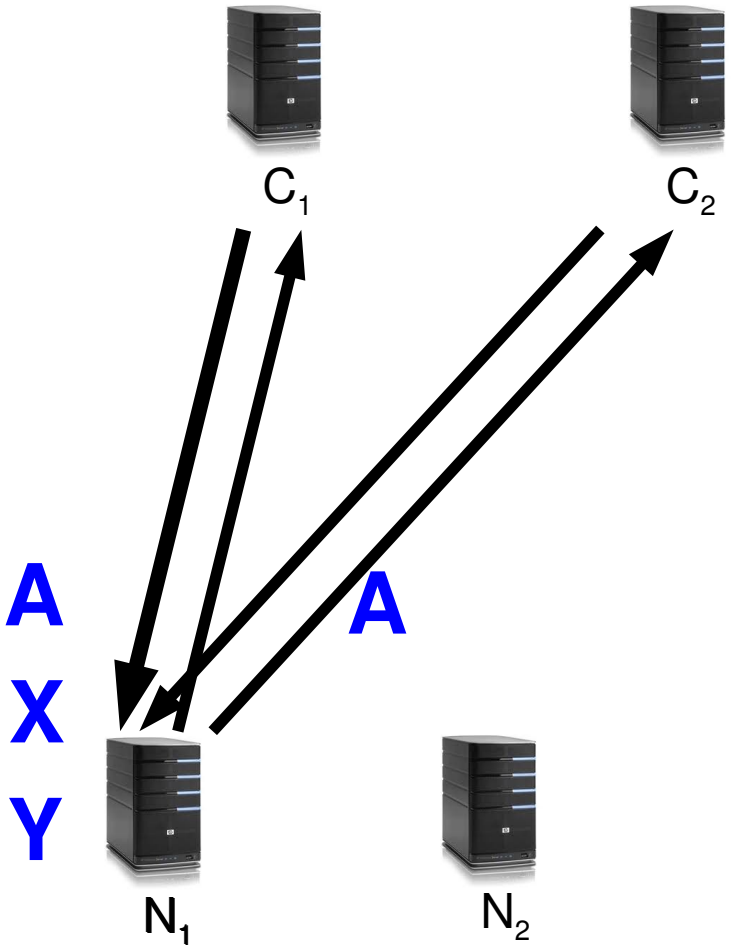
Simplify: One key, initially A

N***=2,*** W=2, R=2

Two clients

Timeline:

- C1 & N1: put(X) + ACK
- C2 & N1: put(Y) + ACK
- C2 & N2: put(Y) + ACK
- C1 & N2: put(X) + ACK



Inconsistency with Quorum Consensus

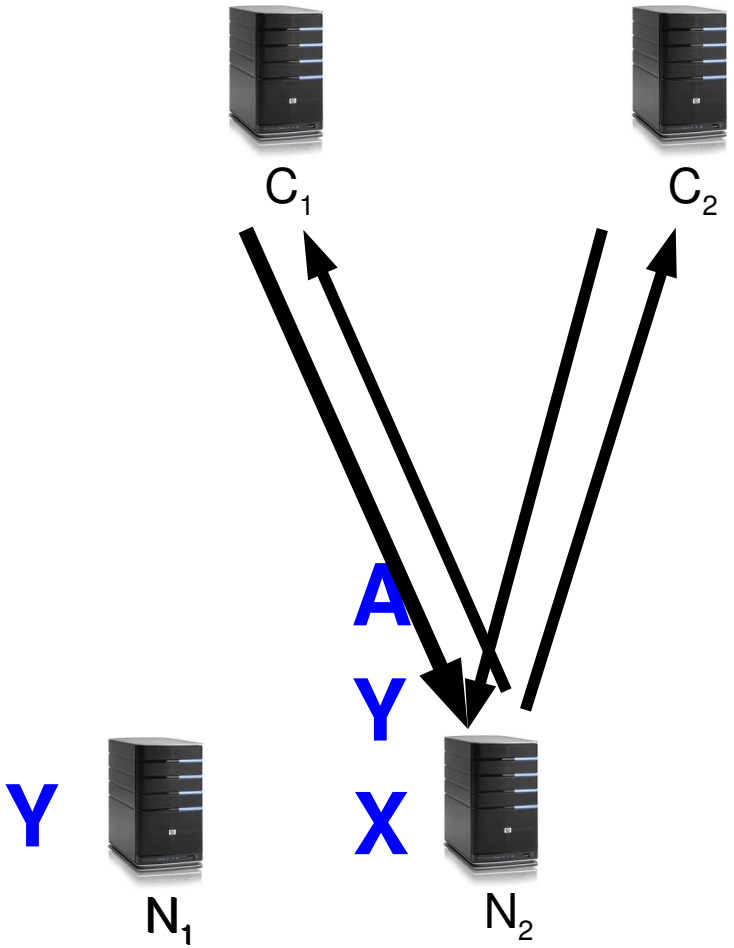
Simplify: One key, initially A

N=2, W=2, R=2

Two clients

Timeline:

- C1 & N1: put(X) + ACK
- C2 & N1: put(Y) + ACK
- C2 & N2: put(Y) + ACK
- C1 & N2: put(X) + ACK



Inconsistency with Quorum Consensus

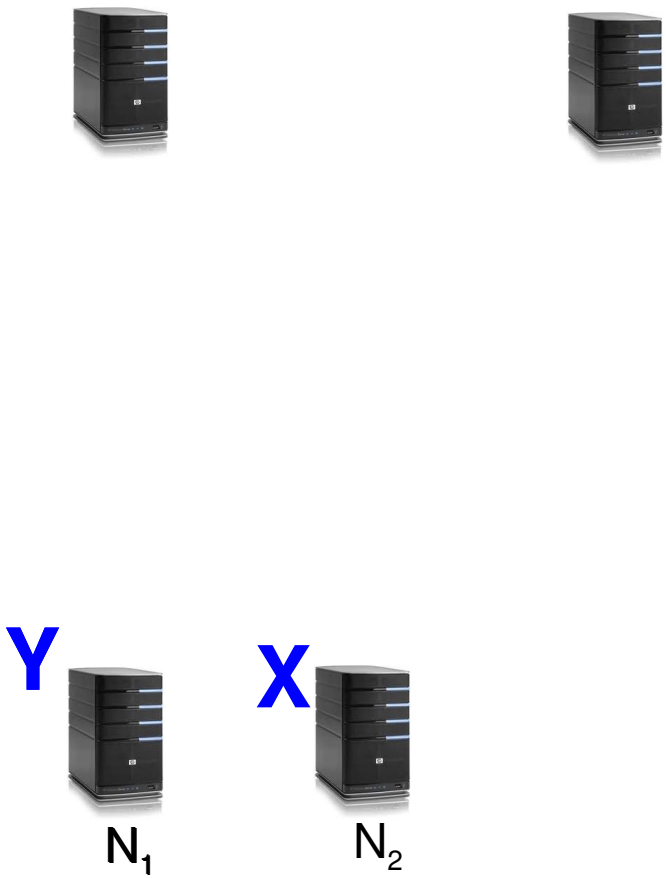
Simplify: One key, initially A

N=2, W=2, R=2

Two clients

Timeline:

- C1 & N1: put(X) + ACK
- C2 & N1: put(Y) + ACK
- C2 & N2: put(Y) + ACK
- C1 & N2: put(X) + ACK



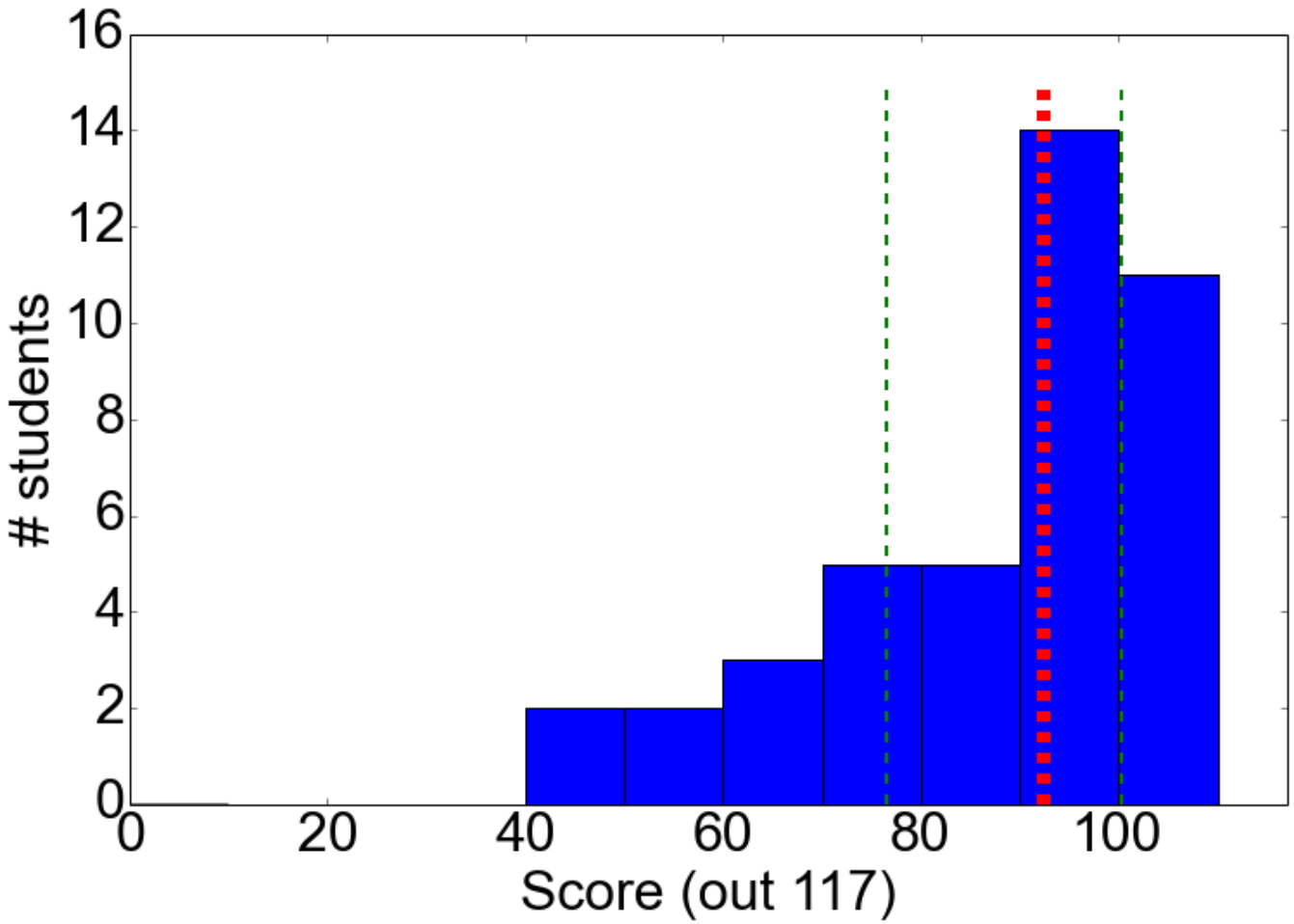
Logistics

Project 2 Checkpoint 1 **due tonight**

HW2 due **Thursday**

Exams are graded, will be returned soon

Midterm Distribution



Break

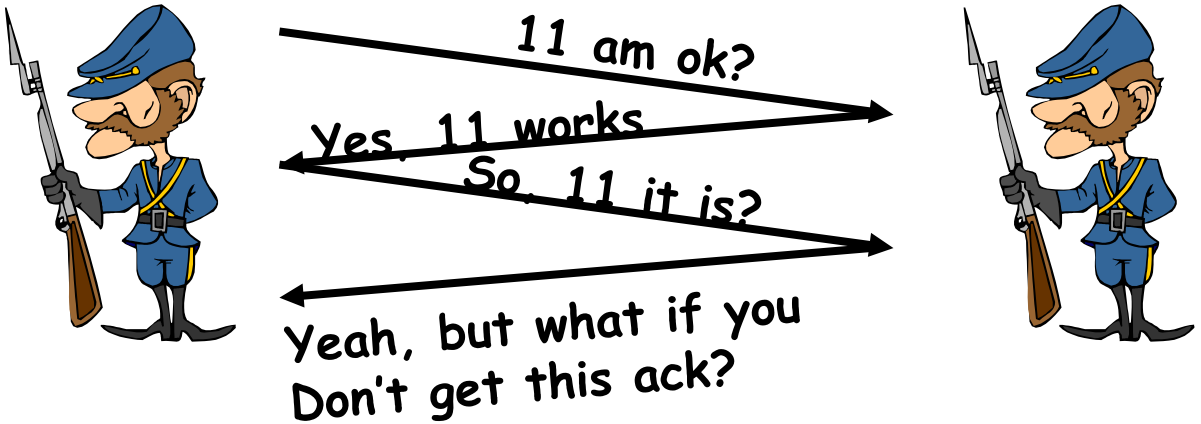
Recall: General's Paradox

General's 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
- Named after Custer, who died at Little Big Horn because he arrived a couple of days too early

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!

Two-Phase Commit

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 **no other types of failures**

Two-Phase Commit: Setup

Ask *every machine* to **"vote"** on transaction

Two possible votes:

- **Commit**
- **Abort**

Rule: unanimous commit only

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 comes back up

Agree to Abort

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

Two-Phase Commit: Finishing

Commit Transaction:

- Machine learns *all machines have agreed to commit*
- Apply transaction
- Record decision in local log

Abort Transaction:

- Machine learns *at least one machine has agreed to abort*
- Do not apply transaction
- Record decision in local log

Two-Phase Commit: Finishing

Commit Transaction:

- Machine learns *all machines have agreed to commit*
- Ap
- Re

Because machines can't "take back" decision, only one of these can happen!

Abort

- Machine learns *at least one machine has agreed to abort*
- Do not apply transaction
- Record decision in local log

Two-Phase Commit: Example (1)

Example: A: Bank of America, W: Wells Fargo

Prepare phase:

- A->W: Transfer $100 to me?

Suppose W *agrees to commit*:

- W writes "transfer $100 to A; <I'll commit>" to its log
- W->A: I'll commit

Two-phase Commit: Example (2)

Example: A: Bank of America, W: Wells Fargo

Prepare phase (con't, commit case):

A receives guarantee of commit:

- A writes "transfer $100 from W; <everyone will commit>" to log
- A applies transaction
- A->W: Everyone's agreed to commit this.

W finalizes transaction:

- W applies transaction

Two-phase Commit: Example (2)

Example: A: Bank of America, W: Wells Fargo

Prepare phase (con't, commit case):

A rec es gua

- A writes "tra log
- A applies tra
- A->W: Every

W finalizes tra

- W applies tra

A doesn't get W's "I'll commit"? A asks W again.

Recall: W won't change its mind. (How does it know it's the same? Transaction ID?)

Two-phase Commit: Example (2)

Example: A: Bank of America, W: Wells Fargo

Prepare phase (con't, commit case):

A receives guarantee of commit:

- writes "
- A applies
- A->W: I'll

W finalizes

- W applies

A crashes before applying transaction?
Reads log when it is restarted; reapplies transaction.
Tells W again.
Called ***redo logging***

Two-phase Commit: Example (2)

Example: A: Bank of America, W: Wells Fargo

Prepare phase (con't, commit case):

A receives guarantee of commit:

- A writes "transfer $100 from W: <I'll commit>" to log
- A applies
- A → W: I'

W finalizes

- W applies transaction

Message from to W lost?
W can ask for it again – A can always check its log

Two-phase Commit: Example (2)

Example: A: Bank of America, W: Wells Fargo

Prepare phase (con't, commit case):

A receives guarantee of commit:

- A writes "transfer $100 to W; <everyone will commit>" to log
- A applies transaction
- A->W: Everyone's agreed to commit this

W finalizes transaction:

- W applies transaction

Two-phase Commit: Example (3)

Example: A: Bank of America, W: Wells Fargo

Prepare phase:

- A->W: Transfer $100 to me?

Suppose W *does not agree to commit*:

- W writes "transfer $100 to A; <I'll abort>" to log
- W->A: I'll abort

Two-phase Commit: Example (4)

Example: A: Bank of America, W: Wells Fargo

Prepare phase (con't, commit case):

A receives guarantee to abort:

- A *does not apply transaction*
- *(*A can reattempt the transfer in *a new transaction*)

Two-phase Commit: Example (4)

Example: A: Bank of America, W: Wells Fargo

Prepare phase (con't, commit case):

A rece[illegible]es gu[illegible]

- A writes "<a[illegible]
- A *does not apply transaction*
- *(*A can reattempt the transfer in *a new transaction*)

A doesn't get answer.
A asks W again.

Formalizing Two-Phase Commit

Roles in Two-Phase Commit

N workers

- Actually perform transaction

One coordinator

- Asks each worker to vote on transaction
- Tells every machine the result of the vote
  - Workers don't need to ask *every* other worker

Previous example:

- A was coordinator *and* worker
- W was worker (only)

Messages in Two-Phase Commit

Coordinator $\rightarrow$ Worker:

- PREPARE

Worker $\rightarrow$ Coordinator:

- VOTE-COMMIT
- VOTE-ABORT

Coordinator $\rightarrow$ Worker:

- GLOBAL-COMMIT
- GLOBAL-ABORT

Messages in Two-Phase Commit

Coordinator → Worker:

- PREPARE

Worker → Coordinator:

- VOTE-COMMIT
- VOTE-ABORT

No taking back!
Always logged
before sending

Coordinator → Worker:

- GLOBAL-COMMIT
- GLOBAL-ABORT

Actual result of
transaction

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

State Machines

Distributed systems are hard to reason about

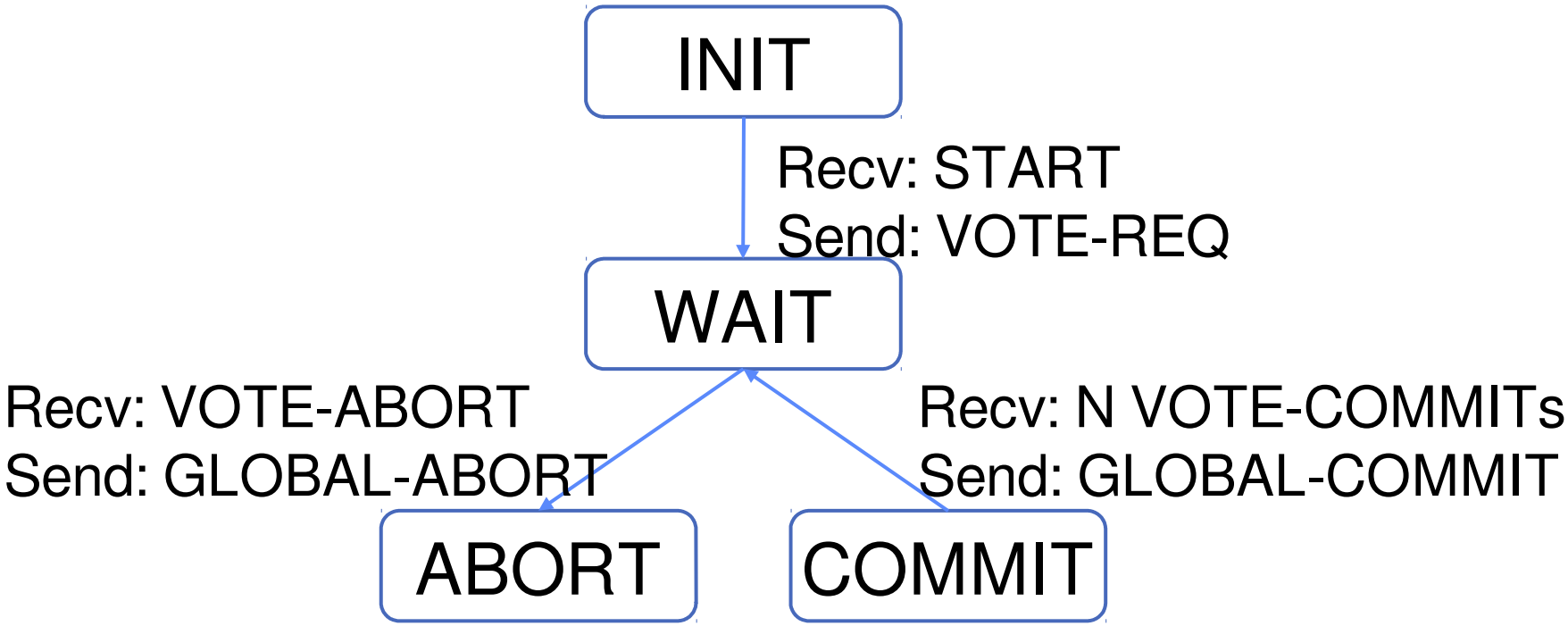
Want a *precise*, *easy-to-reason* about way to express each nodes behavior

One form: **state machine**

- Every node is in a *state*
- When it sends/receives a message, it *transitions* to another state

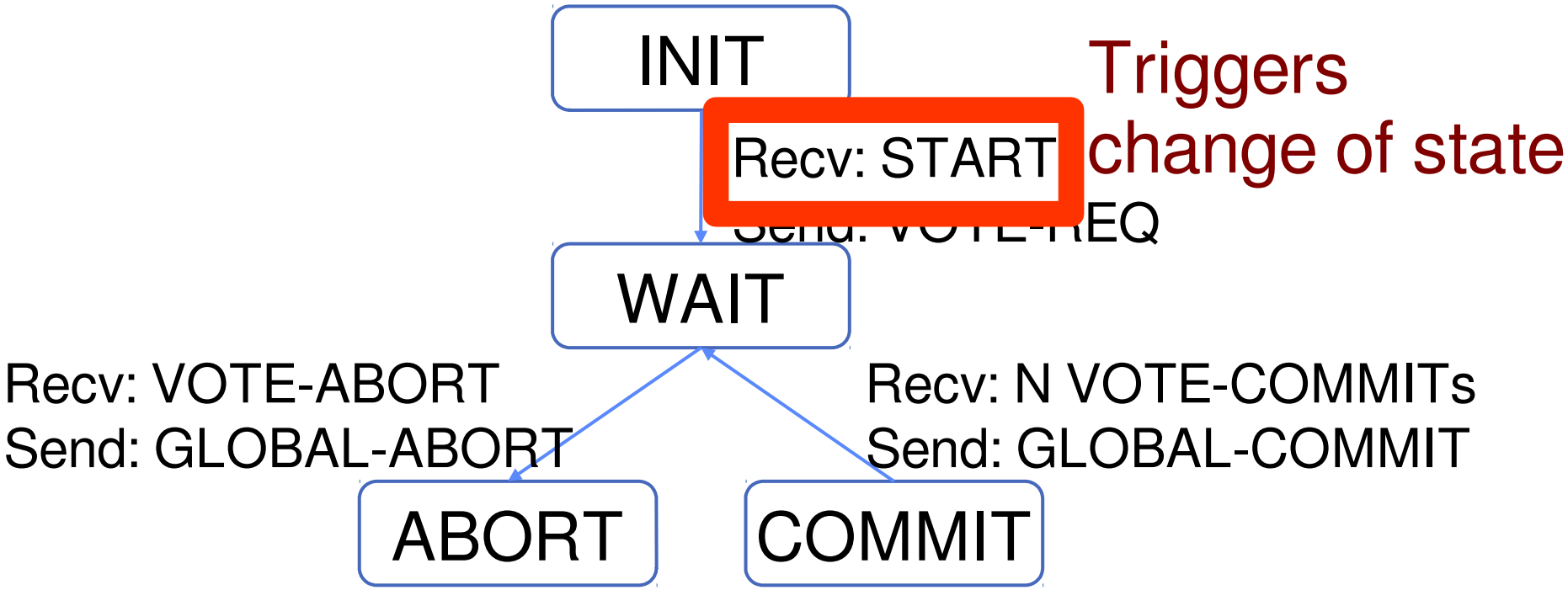
State Machine of Coordinator

Coordinator implements simple state machine:



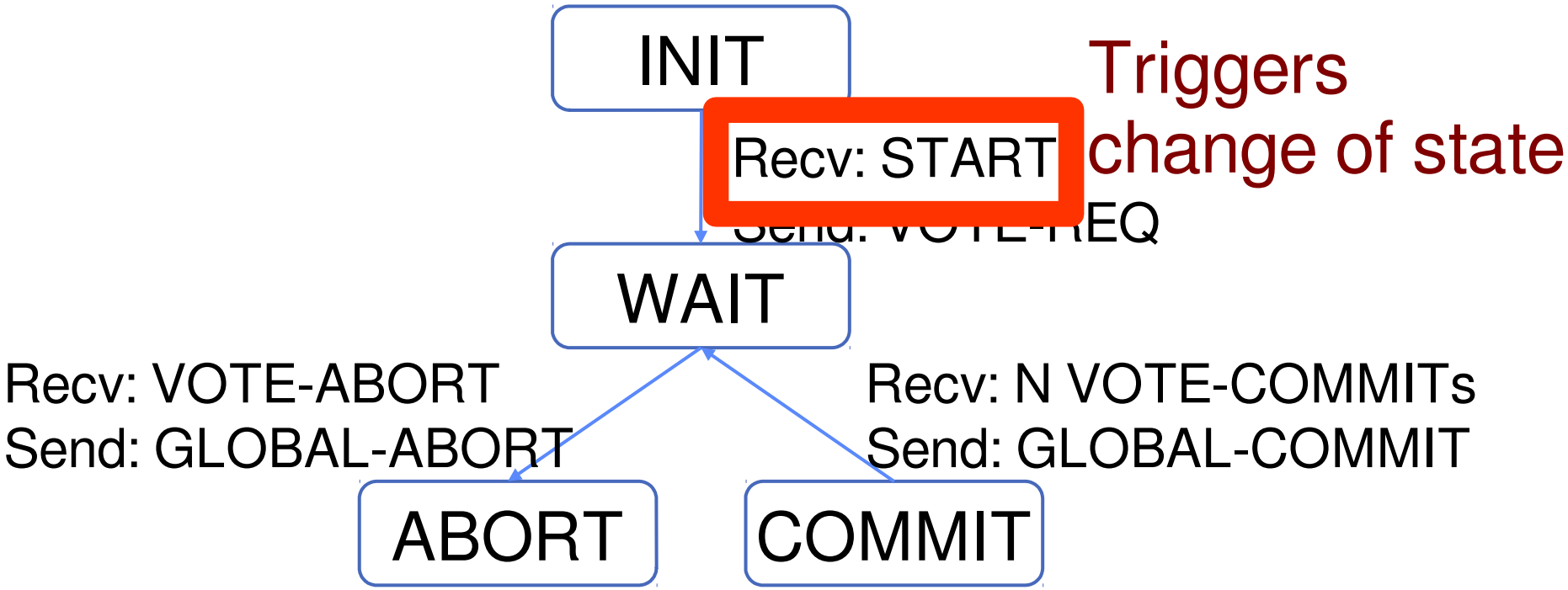
State Machine of Coordinator

Coordinator implements simple state machine:



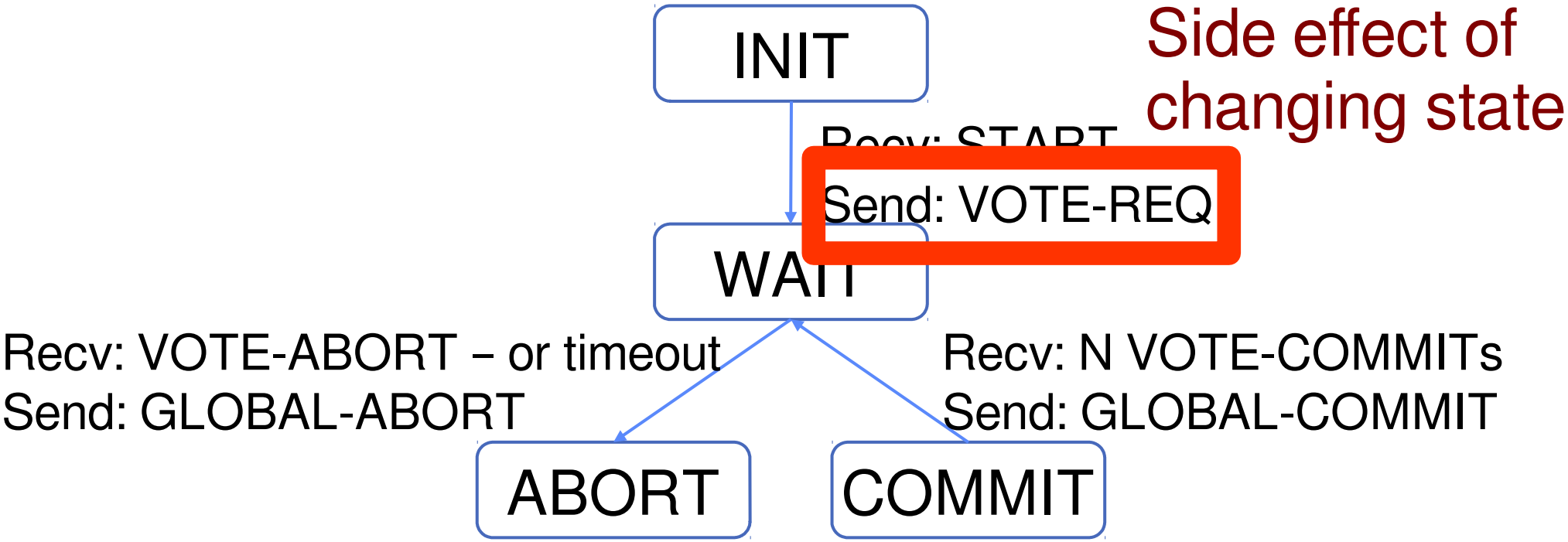
State Machine of Coordinator

Coordinator implements simple state machine:



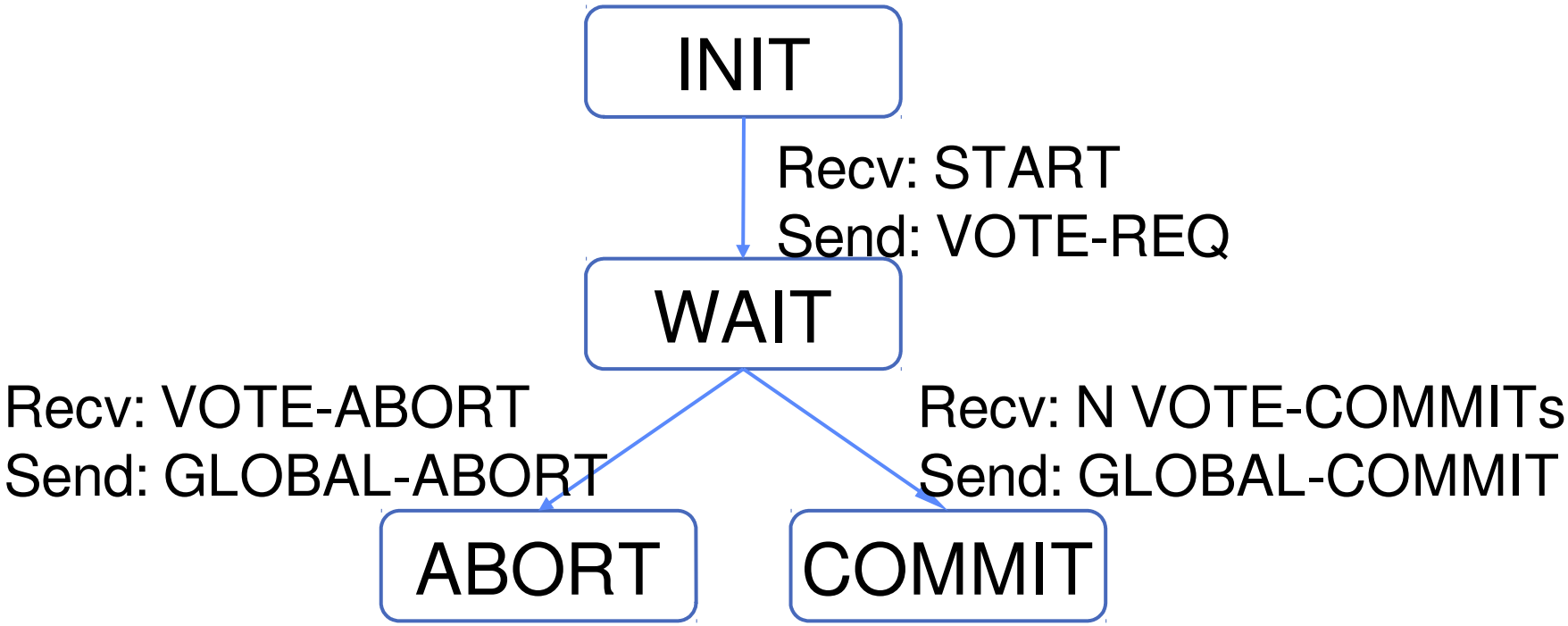
State Machine of Coordinator

Coordinator implements simple state machine:

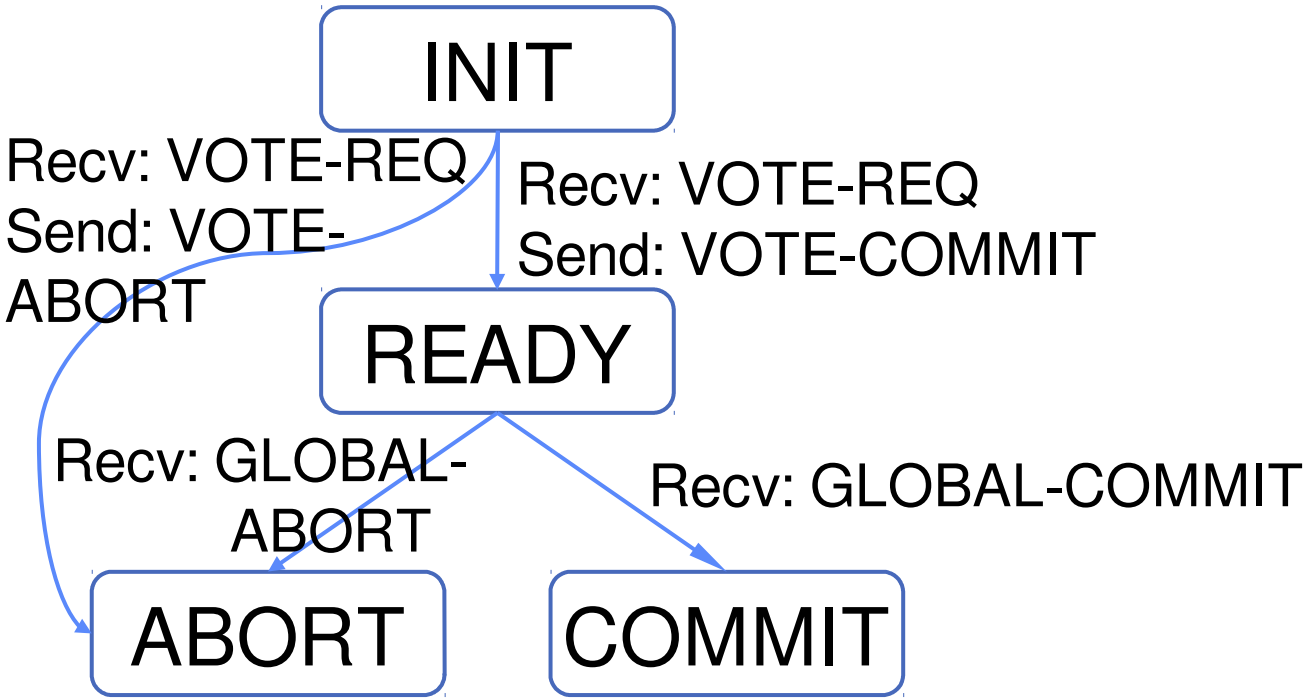


State Machine of Coordinator

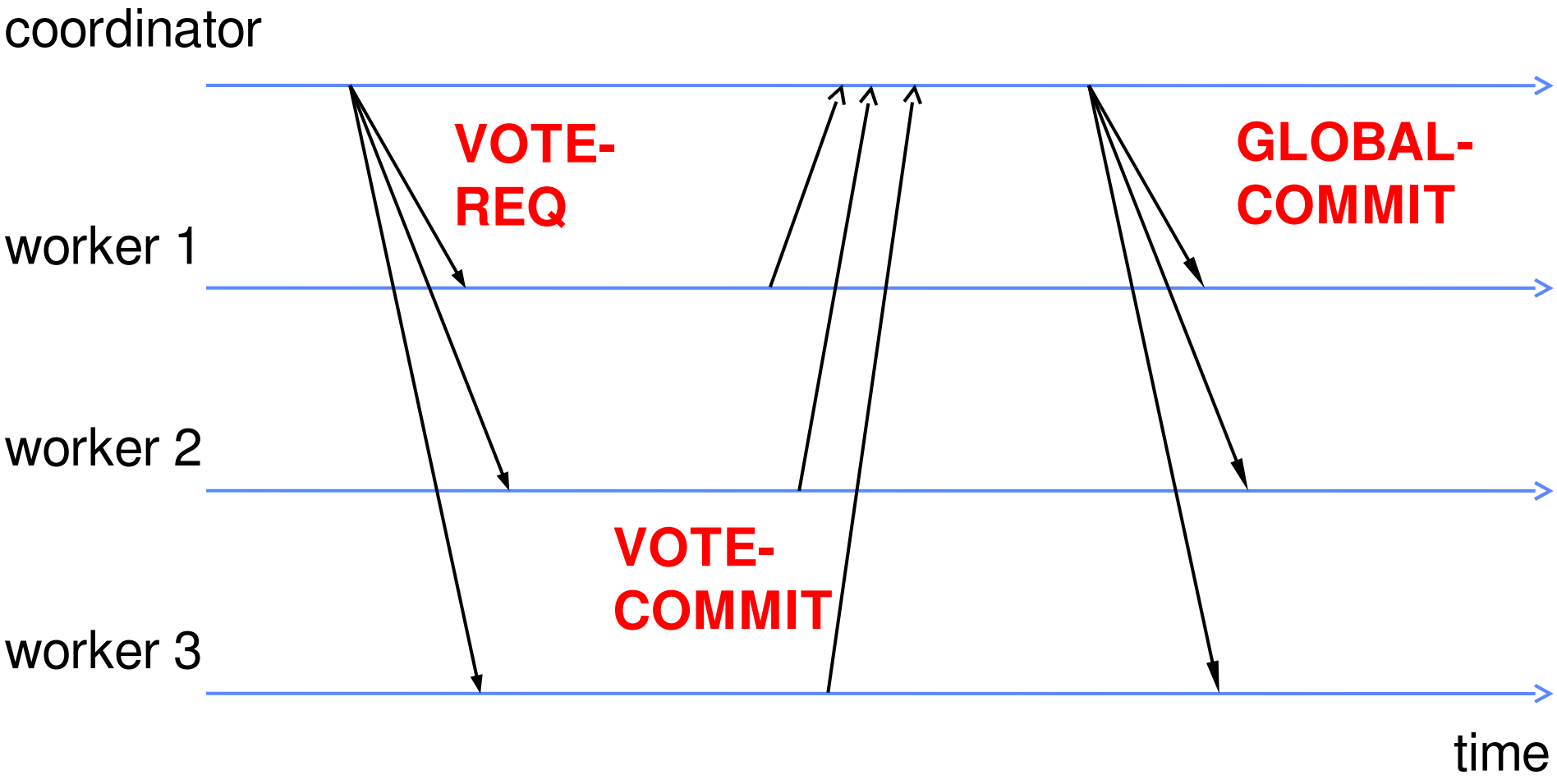
Coordinator implements simple state machine:



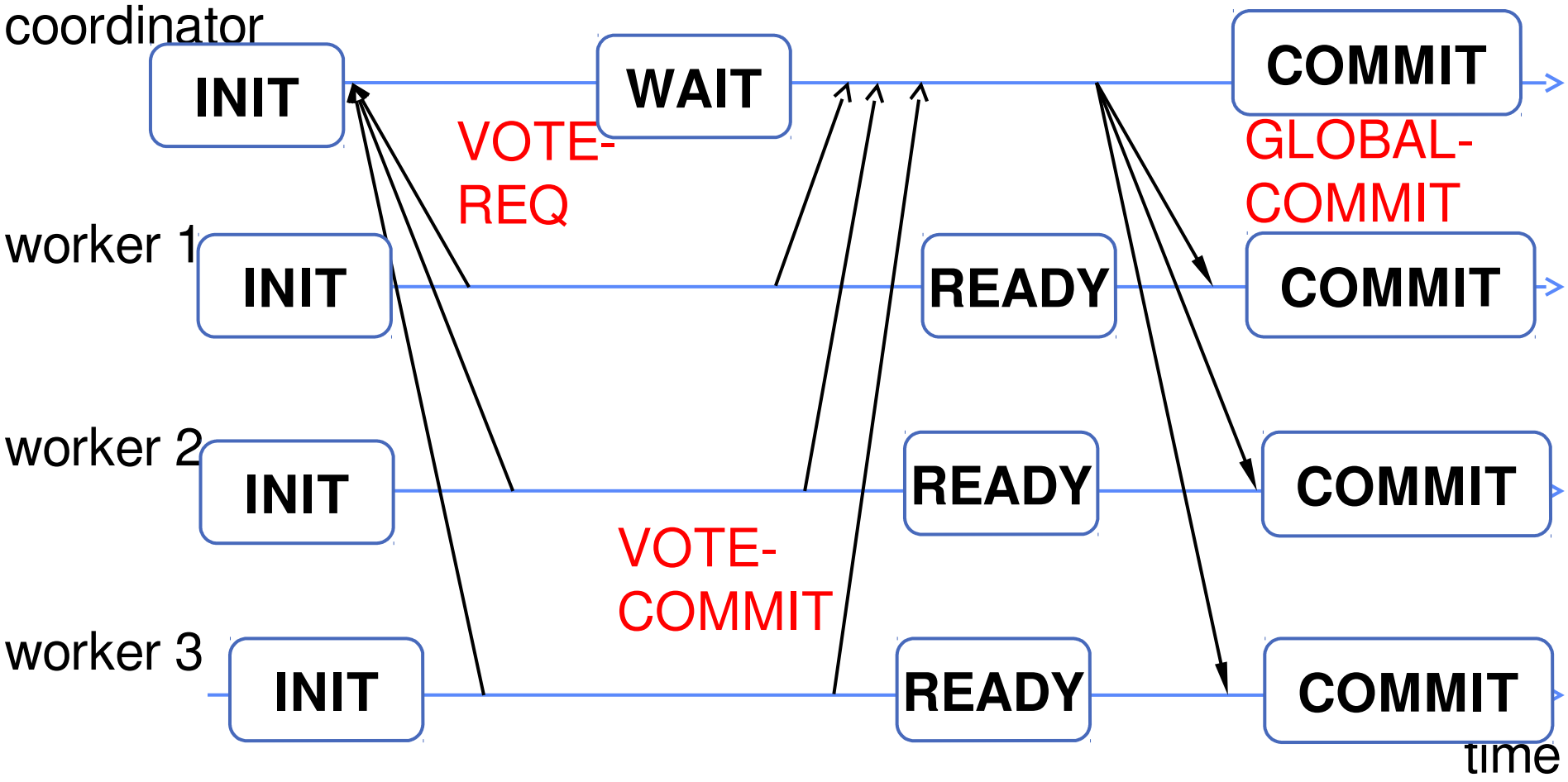
State Machine of Workers



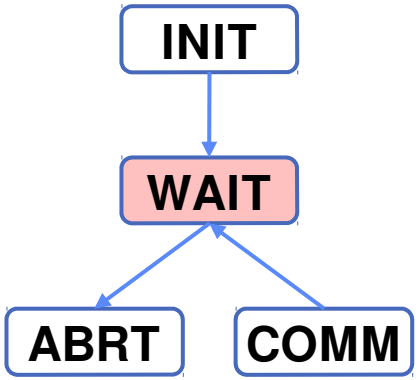
Failure Free Example Execution

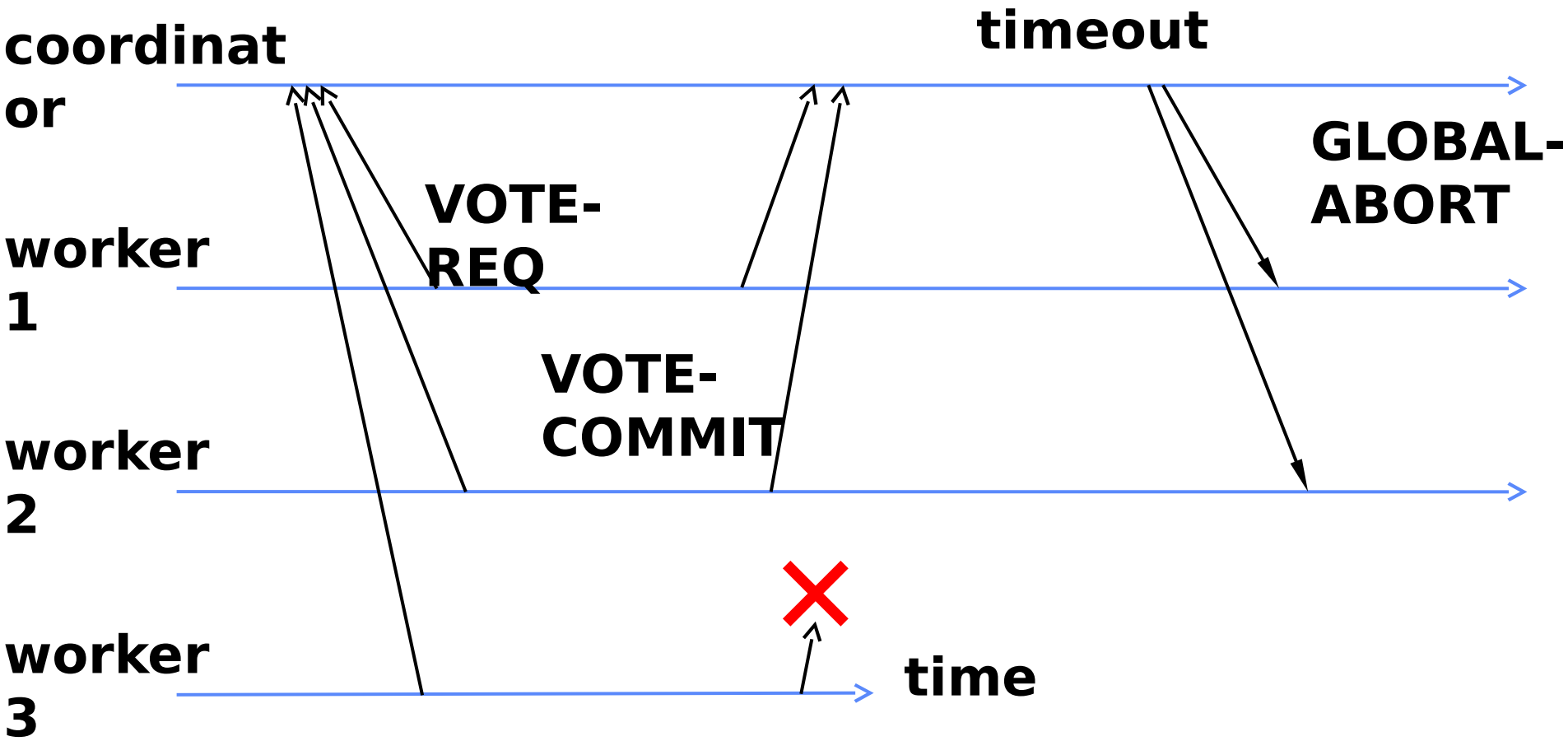


Failure Free Example Execution

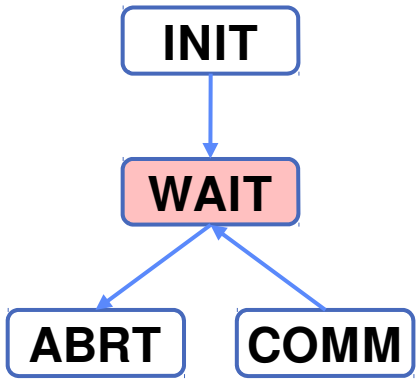


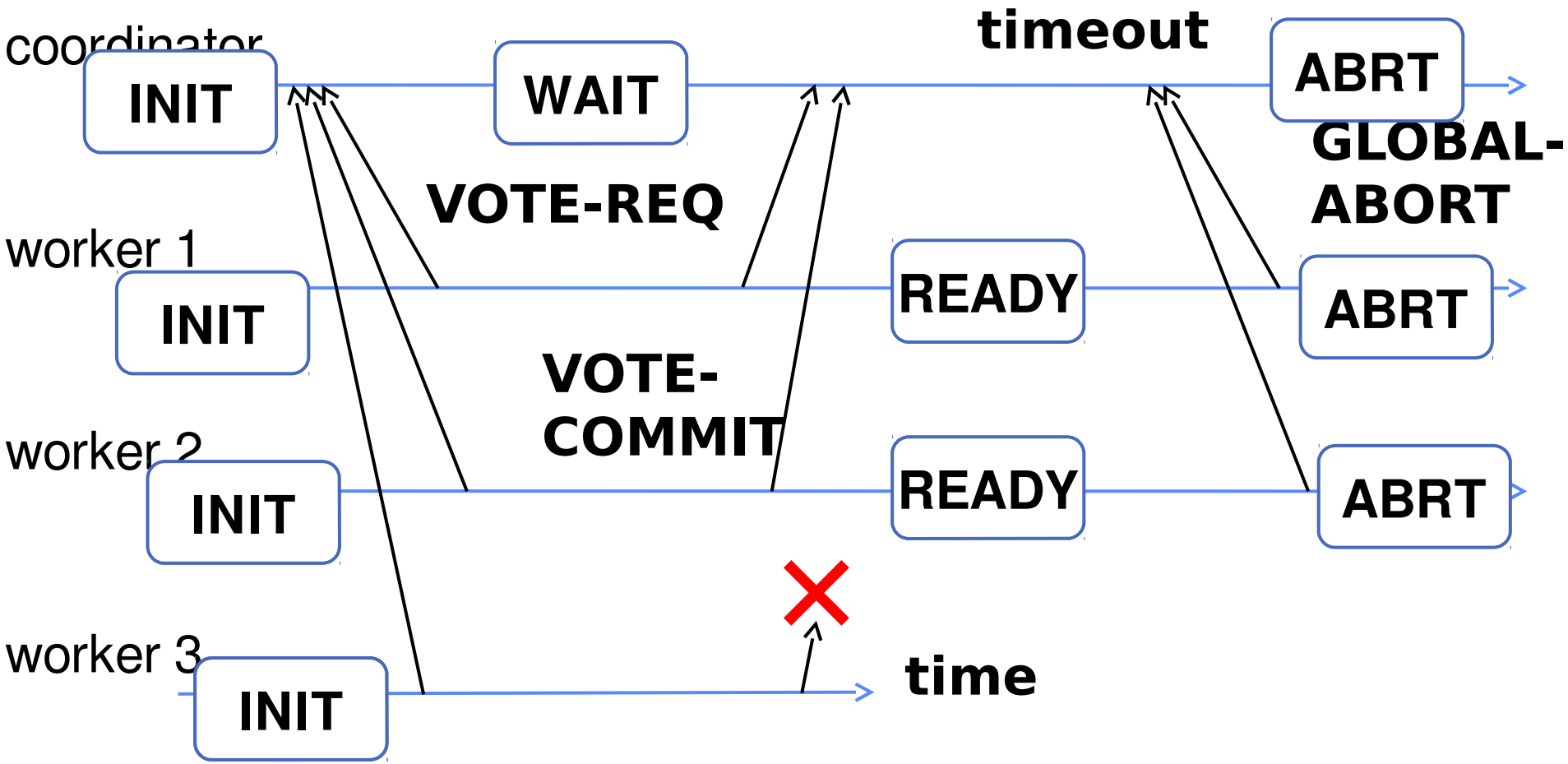
Worker Failure



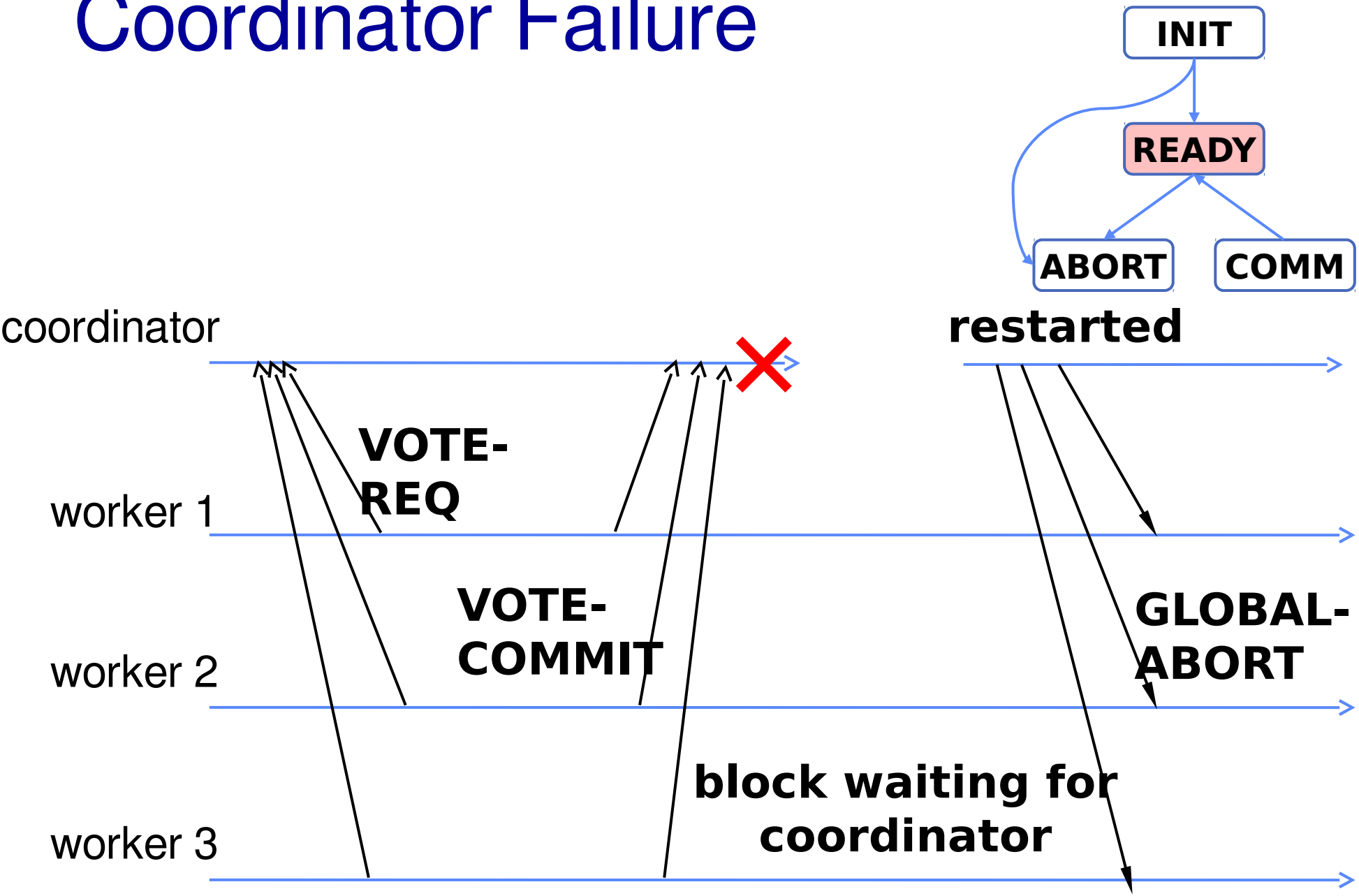


Worker Failure

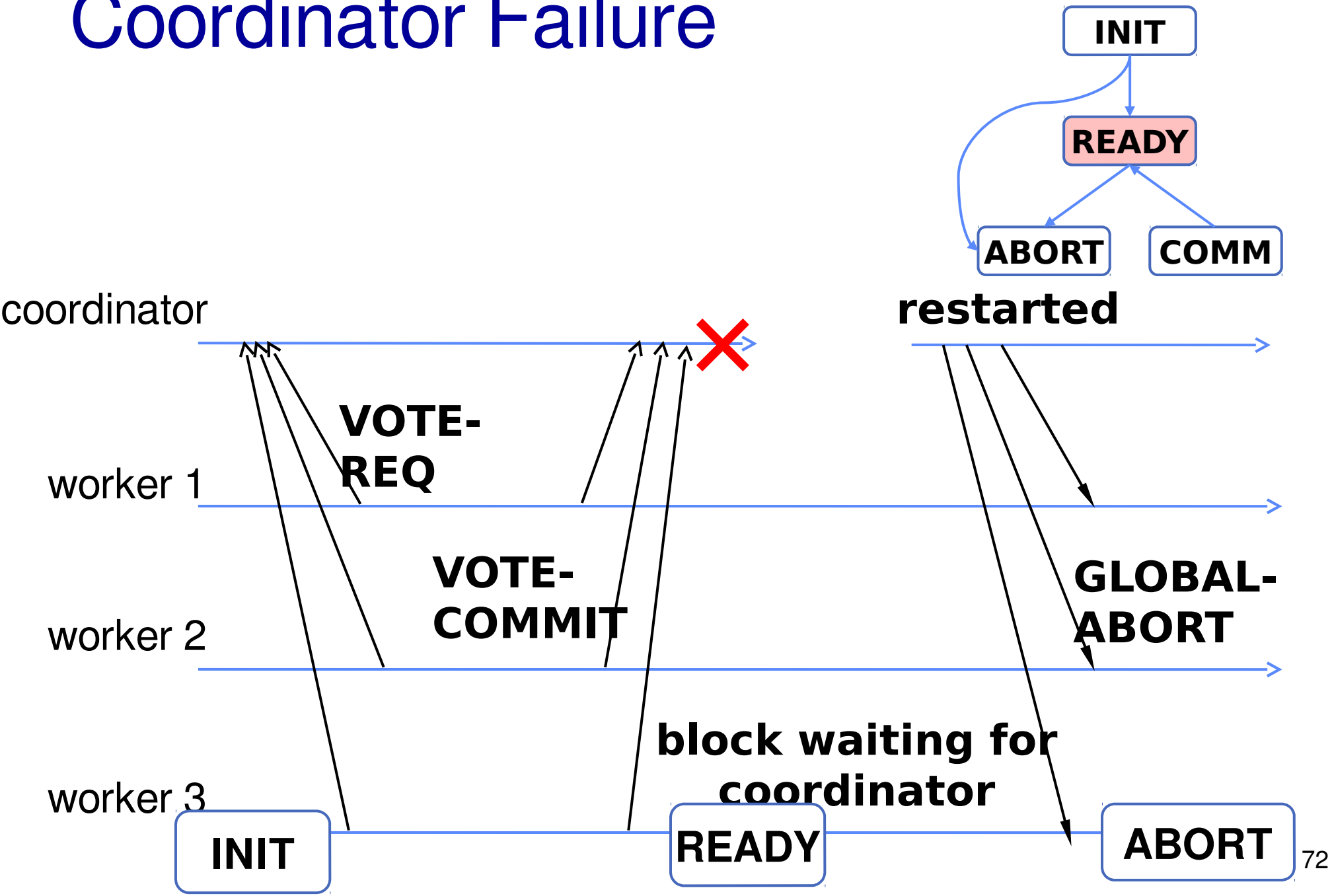




Coordinator Failure



Coordinator Failure



Failure Recovery (1)

Nodes need to **know what state they are in** when they come back from a failure

How? One way: Logs on hard disk. More when we talk about filesystems.

Then recovery rules:

- Coordinator: If was in INIT, WAIT, or ABORT, goto ABORT (resend GLOBAL-
- Coordinate: If in COMMIT, goto COMMIT

Failure Recovery (2)

Recovery rules when coming back from failure:

Coordinator:

- Was in INIT, WAIT, or ABORT?
  Goto ABORT, resend GLOBAL-ABORT
- Was in COMMIT?
  Goto COMMIT, resend GLOBAL-COMMIT

Worker:

- Was in INIT, ABORT?
  Goto ABORT, resend VOTE-ABORT
- Was in COMMIT?
  Goto COMMIT, resend VOTE-COMMIT
- Was in READY?
  Goto READY, ask coordinator for another copy of the VOTE-REQ message

Summary: (Distributed) Consistency

Everyone agrees on the state of the system:

- Won't depend on who you ask
- Won't depend on if nodes go down

Transcation idea:

- **Atomic** change of state everywhere
- Once it happens, never taken back

CAP theorem: perfect consistency and perfect availability impossible

- Two-phase commit: **stall** system instead of letting disconnected node get out of sync

Summary: Two-Phase Commit

Voting protocol – requires unanimity

Transaction logically committed if and only if

- All workers and coordinator decide to vote to commit

Key idea: nodes **never** take back their vote

- On failure: need to recover same votes

Nodes work in lock step (for an item):

- Don't perform new transactions until old one resolved
- **Stall** until transaction is resolved

Consistency with TPC

Simplify: One key, initially A

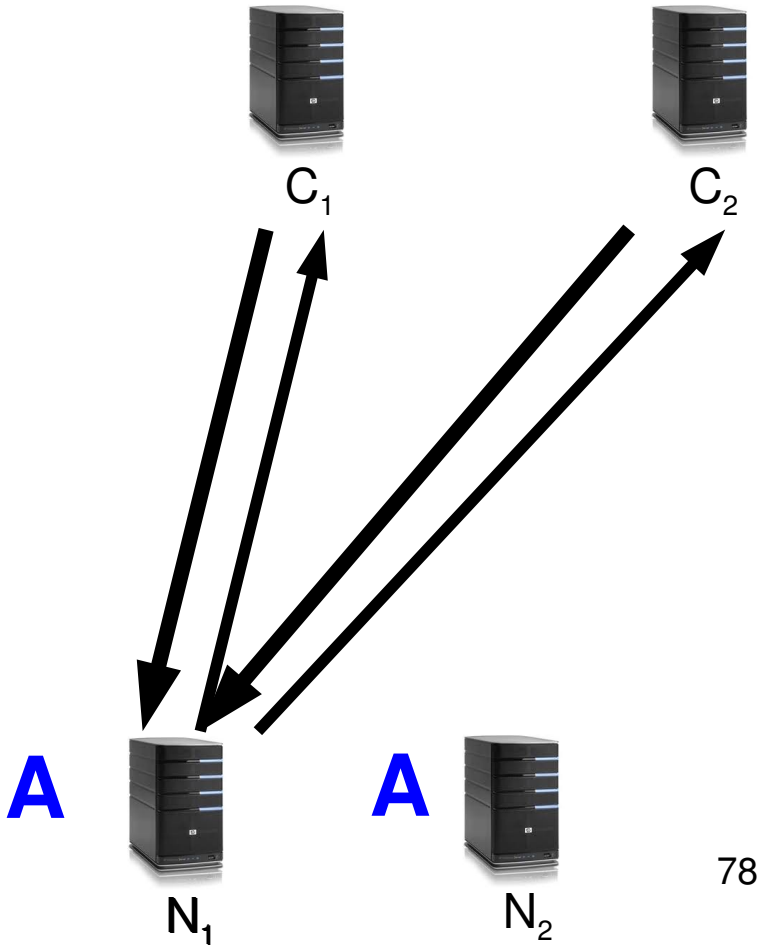
N***=2,*** W=2, R=2

Two clients

Timeline:

- C1 → N1: PREPARE C1:put(X)
- N1 → C1: VOTE-COMMIT C1:put(X)
- C2 → N1: PREPARE C2:put(Y)
- N1 → C2: VOTE-ABORT C2:put(Y)

So **C2 GLOBAL-ABORTs**



Consistency with TPC

Simplify: One ke

N***=2,*** W=2, R=2

Two clients

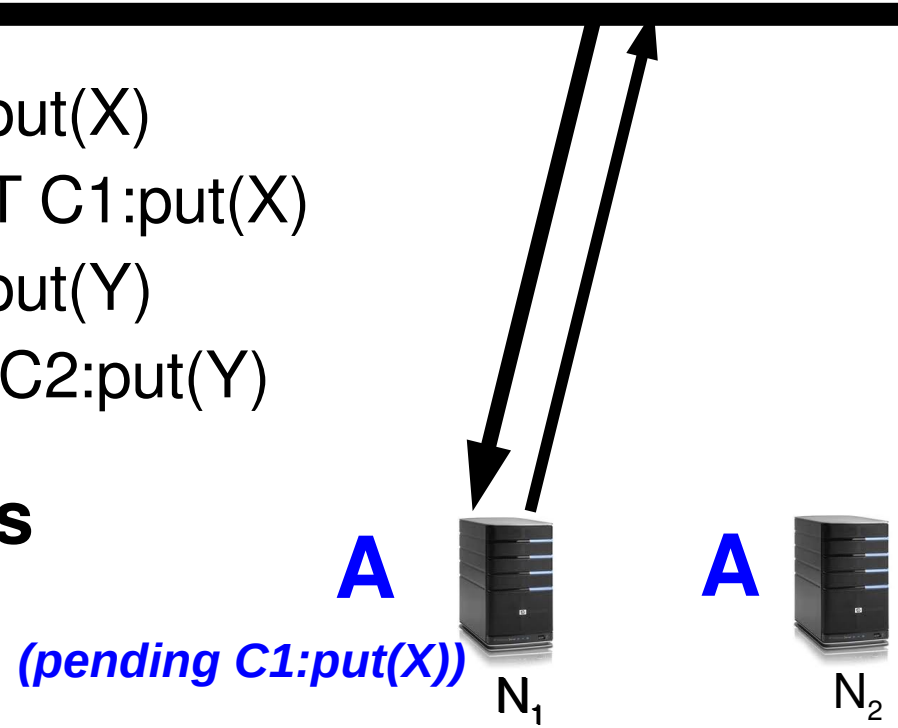
Timeline:

- C1 → N1: PREPARE C1:put(X)
- N1 → C1: VOTE-COMMIT C1:put(X)
- C2 → N1: PREPARE C2:put(Y)
- N1 → C2: VOTE-ABORT C2:put(Y)

So **C2 GLOBAL-ABORTs**

Why does N1 vote to abort?

Needs to track what it's agreed to do. (*Locking*.)

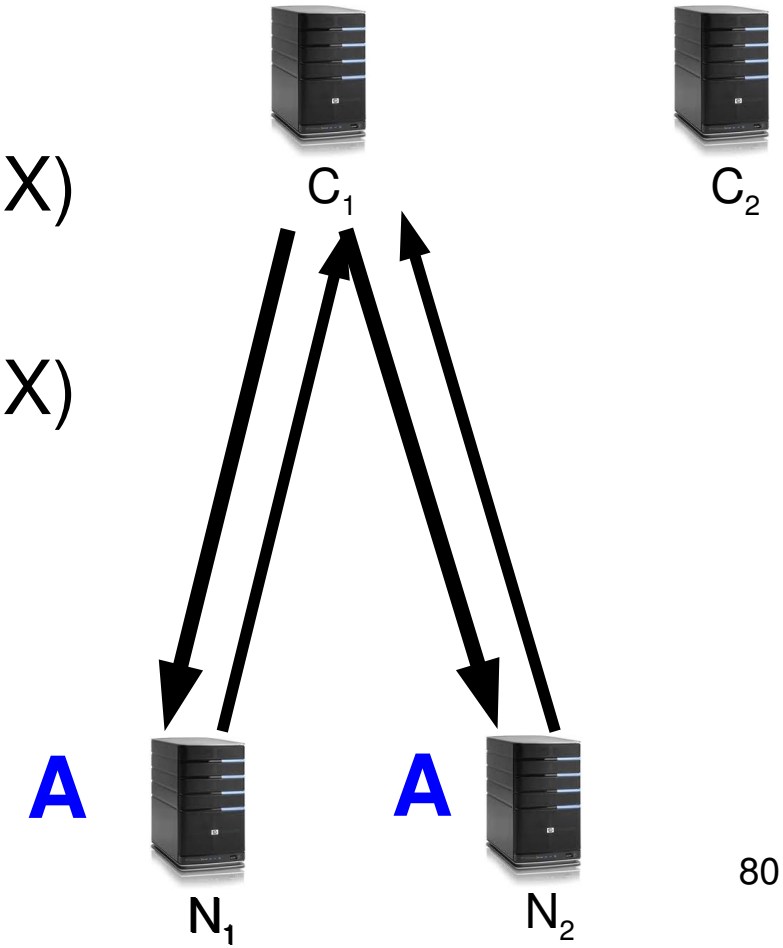


What about reads?

One key, initially A

Timeline:

- C1 → N1: PREPARE put(X)
- N1 → C1: VOTE-COMMIT put(X)
- C1 → N2: PREPARE put(X)
- N2 → C1: VOTE-COMMIT put(X)
- C1 → N1: GLOBAL-COMMIT
- ***C2 → N1: What's the value?***
- C1 → N2: GLOBAL-COMMIT



W

On

Tim

–

–

–

–

– C1 → N1: GLOBAL-COMMIT

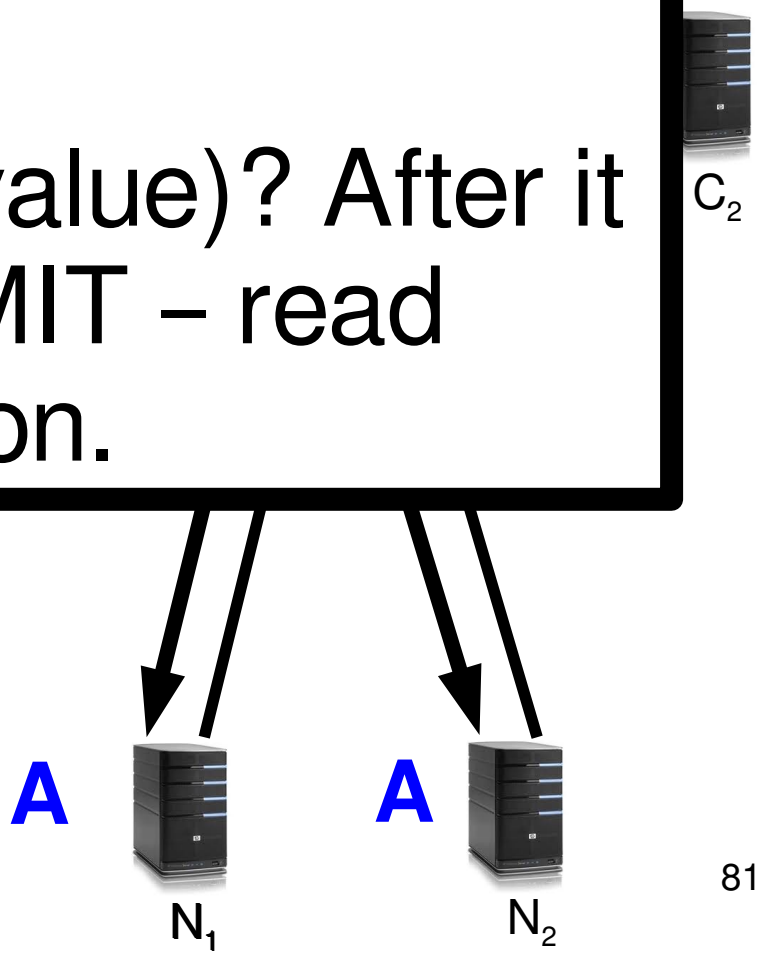
– ***C2 → N2: What's the value?***

– C1 → N2: GLOBAL-COMMIT

N2 hasn't applied transaction yet.

Can it return A (old value)? – Yes, read occurs *before* transaction.

Can it return X (new value)? After it gets GLOBAL-COMMIT – read occurs *after* transaction.

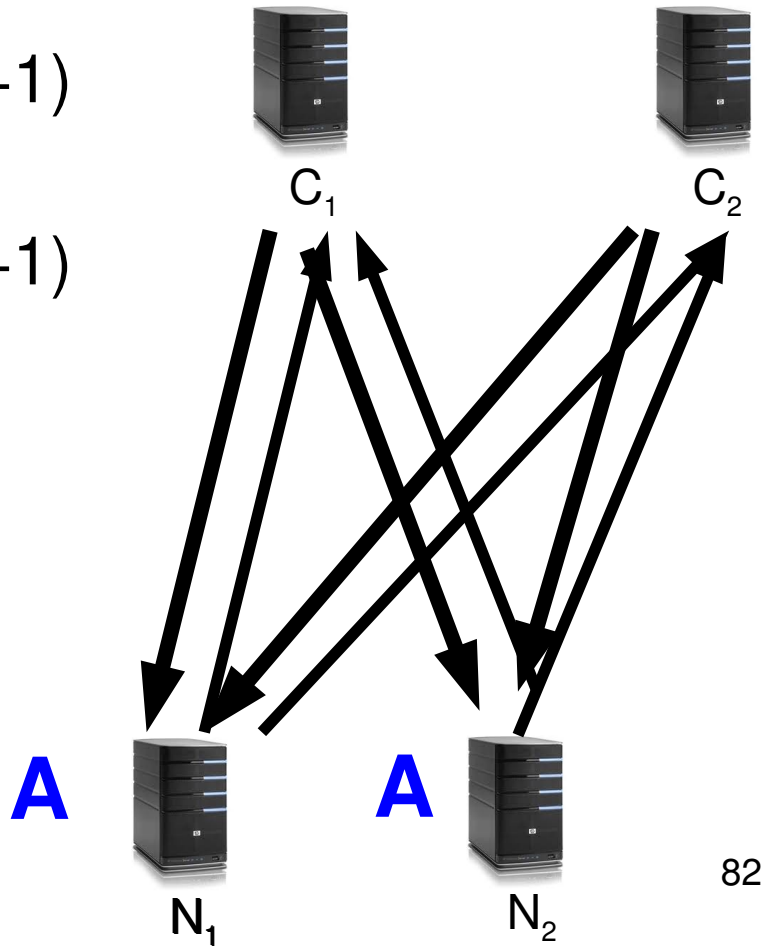


Consistency with TPC Take Two

Timeline:

- C1 → N1: get() → A
- C2 → N2: get() → A
- C1 → N1, N2: PREPARE C1:put(A+1)
- N1, N2 → C1: VOTE-COMMIT
- C2 → N1, N2: PREPARE C2:put(A+1)
- N1, N2 → C2: VOTE-COMMIT

C1 and C2 didn't see each other's updates?



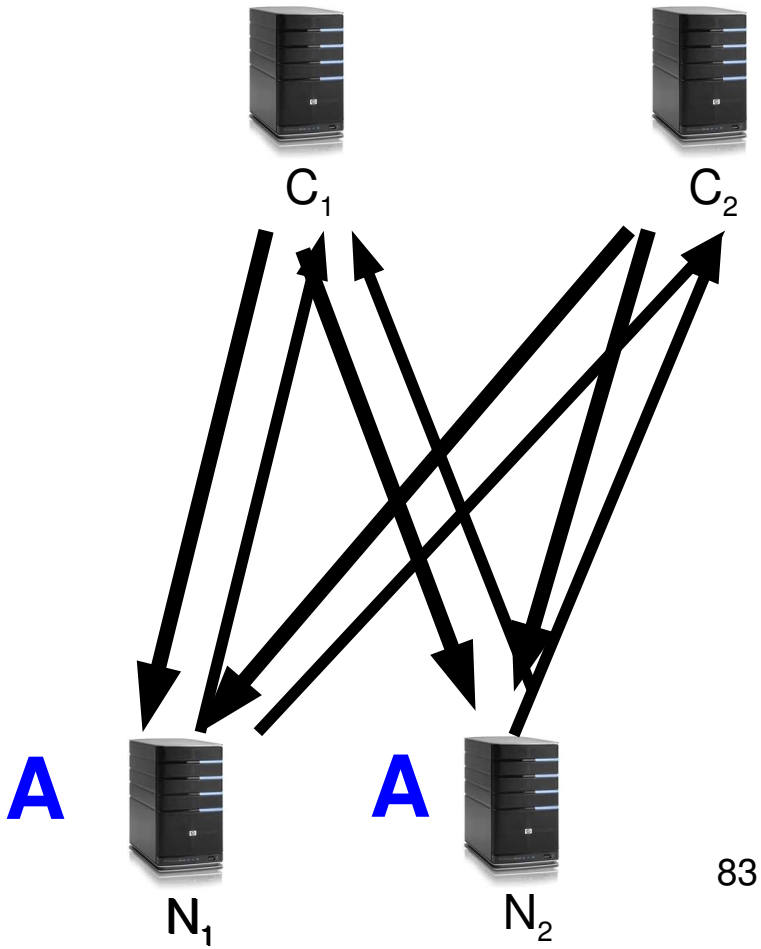
Consistency with TPC Take Three

Timeline:

- C1 → N1, N2: get() → PREPARE: C1:get()=A
- N1, N2 → C1: VOTE-COMMIT
- C2 → N1, N2: get() → PREPARE: C2:get()=A
- N1, N2 → C2: VOTE-COMMIT
- C1 → N1, N2: PREPARE C1:put(A+1)
- N1, N2 → C1: VOTE-COMMIT
- C2 → N1, N2: PREPARE C2:put(A+1)
- N1, N2 → C2: VOTE-COMMIT

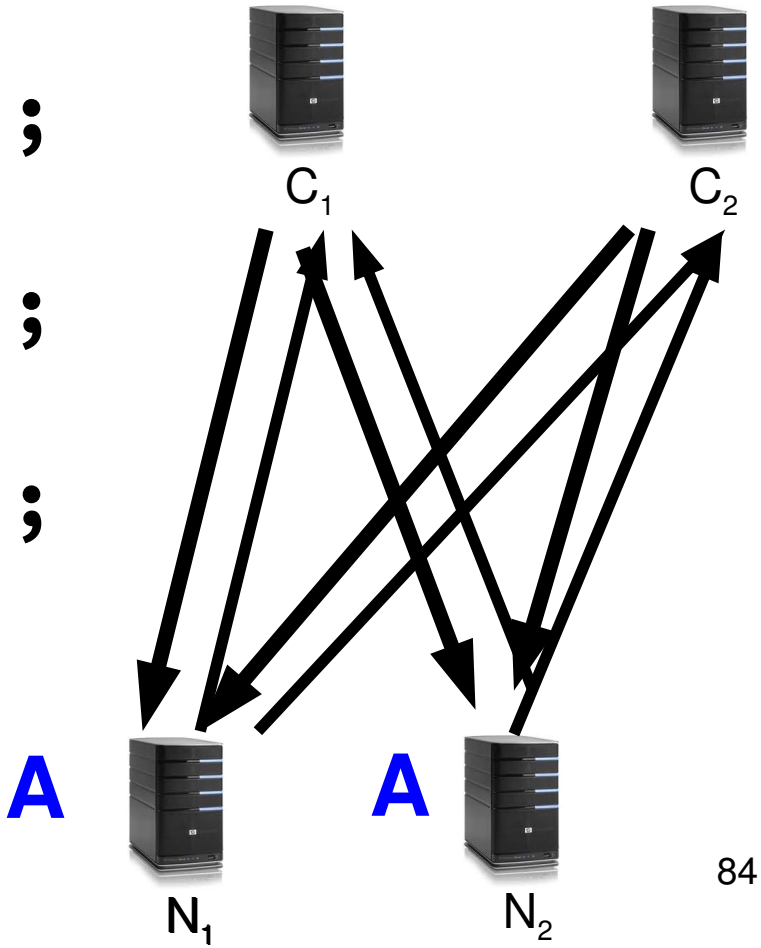
We did the read as a transaction – consistent and atomic!

C1 and C2 *still* didn't see each other's updates?



Recall: Atomic Load/Stores

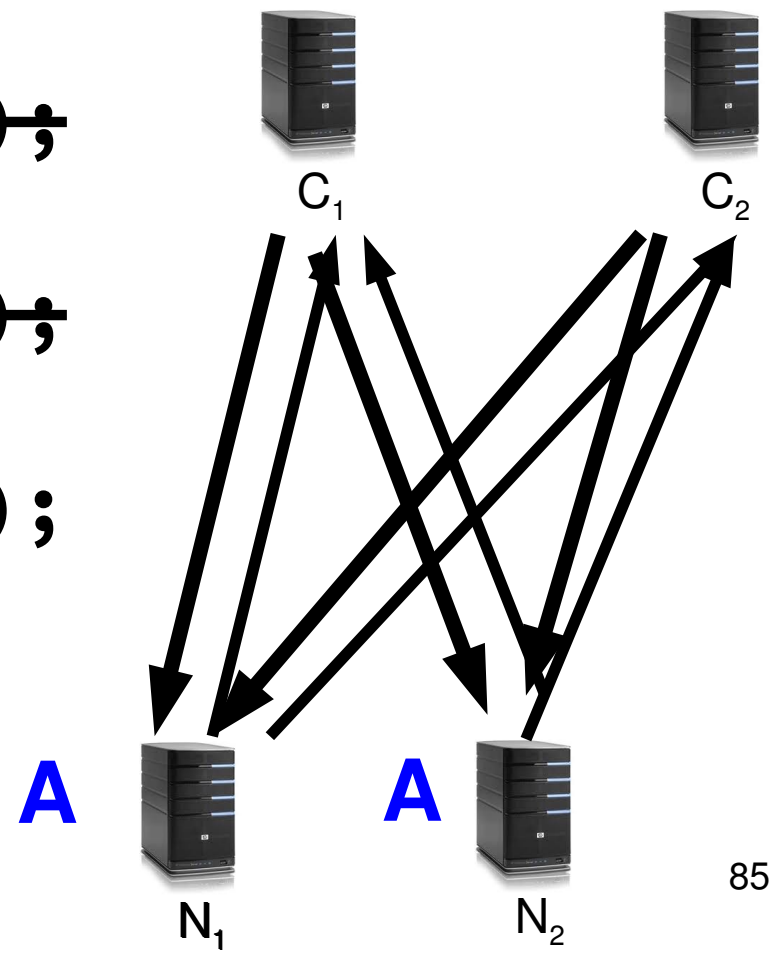
```
lock_acquire(&value_lock);
temp = value;
lock_release(&value_lock);

lock_acquire(&value_lock);
value = temp + 1;
lock_release(&value_lock);
```



Recall: Atomic Load/Stores

```
lock_acquire(&value_lock);
temp = value;
```
~~`lock_release(&value_lock);`~~

~~`lock_acquire(&value_lock);`~~
```
value = temp + 1;
lock_release(&value_lock);
```

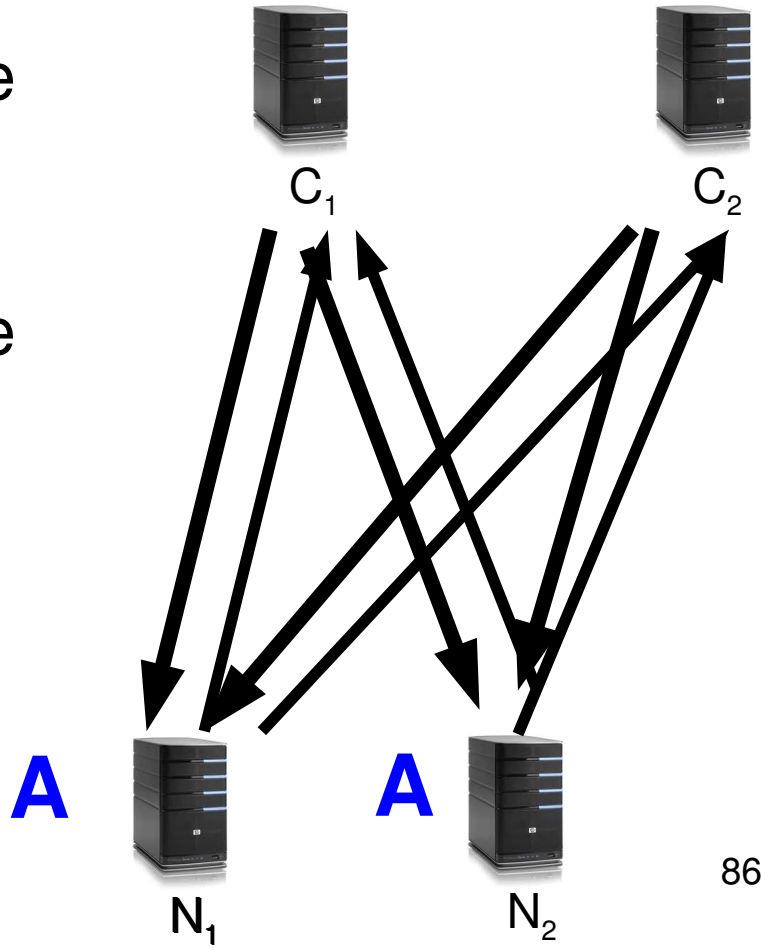


Consistency with TPC Take Four

Timeline:

- C1 → N1: get() → A
- C2 → N2: get() → A
- C1 → N1, N2: PREPARE C1:change from A to A+1
- N1, N2 → C1: VOTE-COMMIT
- C2 → N1, N2: PREPARE C2:change from A to A+1
- N1, N2 → C2: **VOTE-ABORT**

Transacation *includes entire update*



The ACID properties of Transactions

Atomicity: all actions in the transaction happen, or none happen

Consistency: transactions maintain data integrity, e.g.,

- Balance cannot be negative
- Cannot reschedule meeting on February 30

Isolation: execution of one transaction is isolated from that of all others; no problems from concurrency

Durability: if a transaction commits, its effects persist despite crashes

Isolation and Locking

Full answer in a databases class

Simplest solution: Always lock everything

- One transaction at a time

Generally: lock on on prepare, unlock on global-commit/abort

The CAP Theorem

Consistency – as if single serial order

Availability – system will accept writes/reads

Partition Tolerance – system will handle nodes failing

Choose (at most) **two**

The CAP Theorem Choics

Two-phase commit with a **partition** (some nodes separated from rest of system or down):

- **Consistent** (reads never return wrong values)
- **Not available** (can't perform new transactions)

Distributed Agreement

Two-phase commit makes a **decentralized** decision

Example decision: Change value of a key.

Problem: What happens if a machine fails?

- Two-phase commit: ***blocking***

Two-Phase Commit: Blocking

A->B: Prepare to set value to 42

B: writes "transaction: set value to 42; agree-to-commit" to its log

B->A: Commit

A crashes, loses message

B ***cannot*** allow access to value

Agreement in Face of Failure

Idea: if a majority of nodes agree, do it

If a minority don't participate, ignore them.

Fail-stop → non-agreeing nodes don't participate

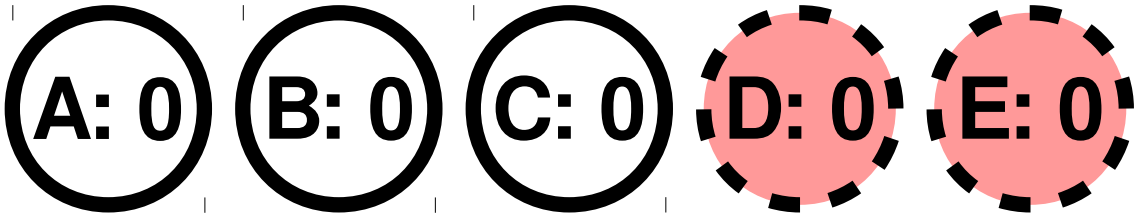
Algorithms that do this: Paxos, Raft

- ***very very very tricky***
- similar idea to two-phase commit

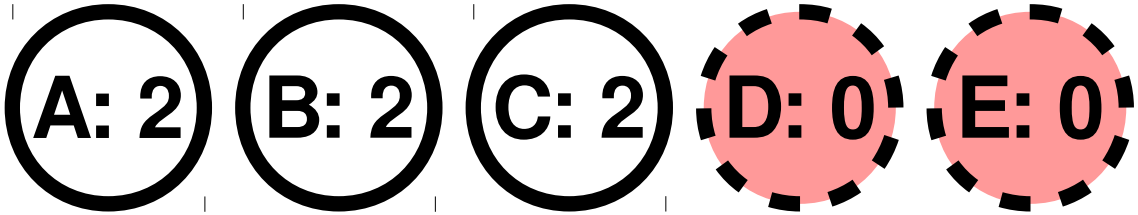
Why a majority? (1)

Key property: **Overlap**

Suppose we use transactions to track a value, initially 0

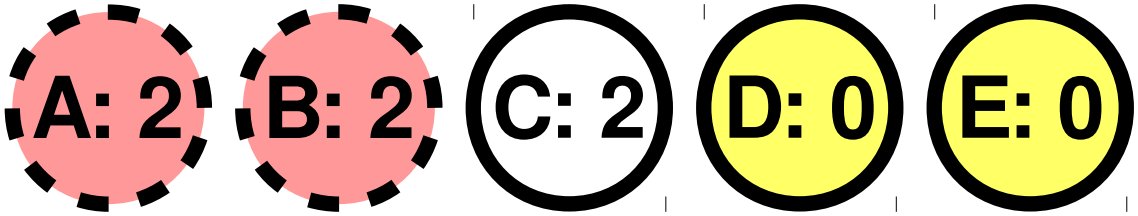


We run the transaction "+ 2" while D, E are down:



Why a majority? (2)

Now, D, E come back up and A, B go down:



Need **overlap** (C in in this case)

- Guaranteed by choice of majority

Overlap prevents us from ***losing*** transactions

- Means *every node* is responsible for resending missed updates

Beyond Fail-Stop

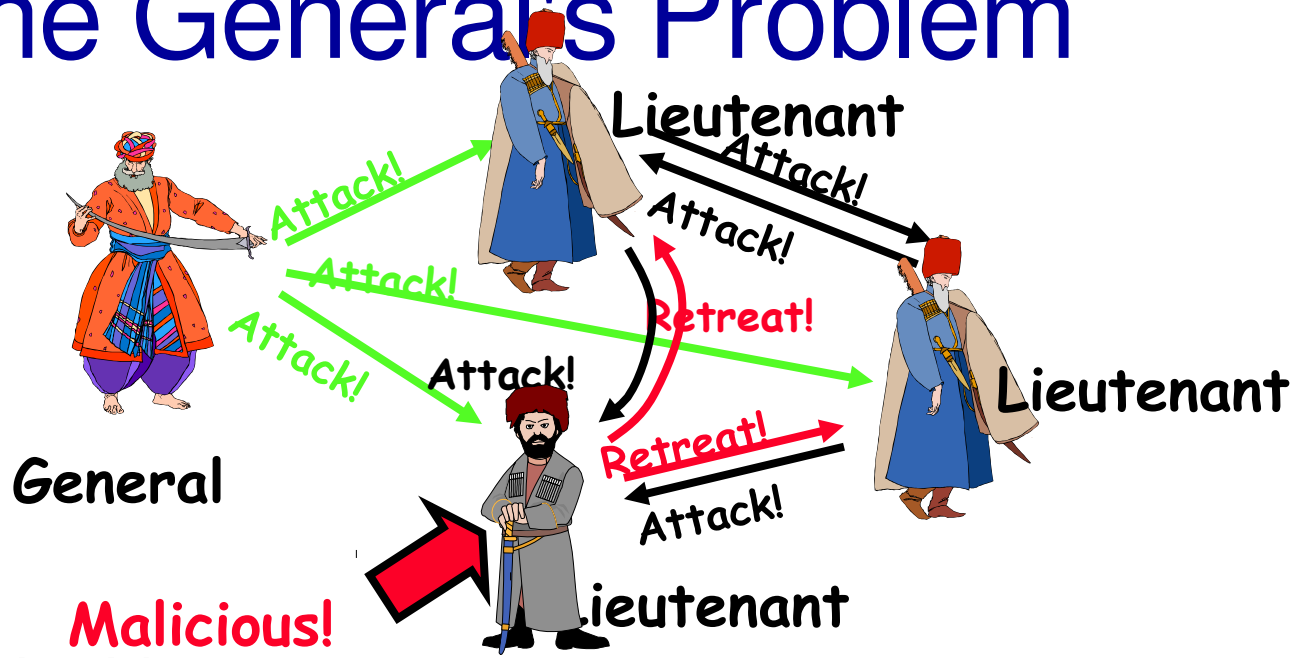
What if a minority of nodes send bad data?
Selectively? Selectively drop messages?

What if a minority of nodes is malicious?

Can Paxos/Raft still work? **No.**

This is called ***Byzantine failures***

Byzantine General's Problem



One general, N-1 Lieutenants

Some number (F) want chaos

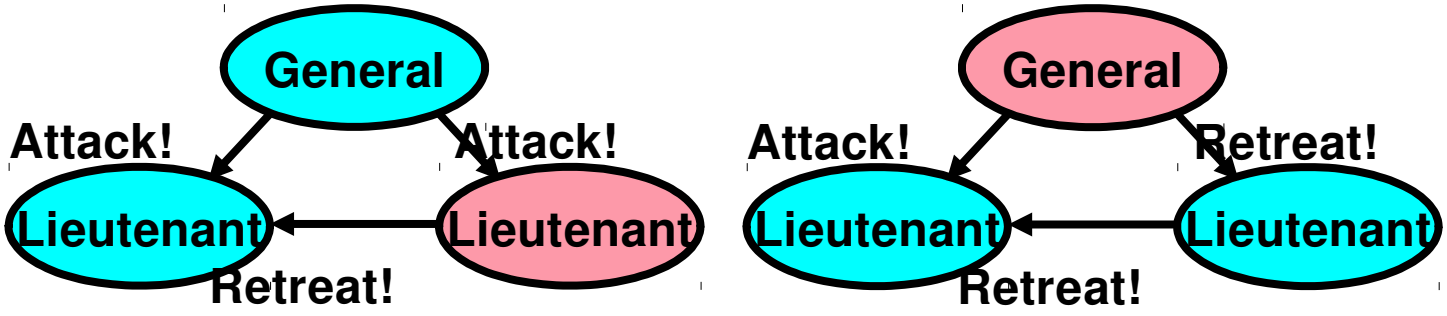
- Can say contradictory things!

Goal: General sends order and

- 1: All non-insane lieutenants obey the *same order*
- 2: If the general is not insane, they obey the General's order

Byzantine Generals: Impossibility

N=3: → **NO SOLUTION**



General theorem: **need** N >= 3F + 1

Byzantine Generals: Solutions

There are protocols that solve byzantine generals for N >= 3F + 1 (the lower bound).

Original algorithm: O(2^N) messages!

Castro and Liskov, "Practical Byzantine Fault Tolerance", O(**N^2**) messages

- Note: A lot worse than Paxos/Raft (failstop)
- Also a lot more complicated