From Bits to Clouds

A Gentle Introduction to Distributed Systems

Rushil Ambati
27th March 2024

Taco Tales

Chapter 1

How do we currently deal with a large number of people?

Queueing

Queueing

... but what if there are a *lot* of people?

Scaling up vertically

Scaling up vertically

... but the single queue becomes a bottleneck!

Scaling out horizontally

Scaling out horizontally

Scaling out horizontally

"A **distributed system**
is a collection of independent computers
that appear to its users as one computer"

Andrew Tanenbaum, ***Distributed Systems: Principles and Paradigms***

Back to Reality

Chapter 2

Centralised vs Distributed Systems

Centralised systems store state on a single computer.

- Simpler
- Easier to reason about
- Can be faster for a single user

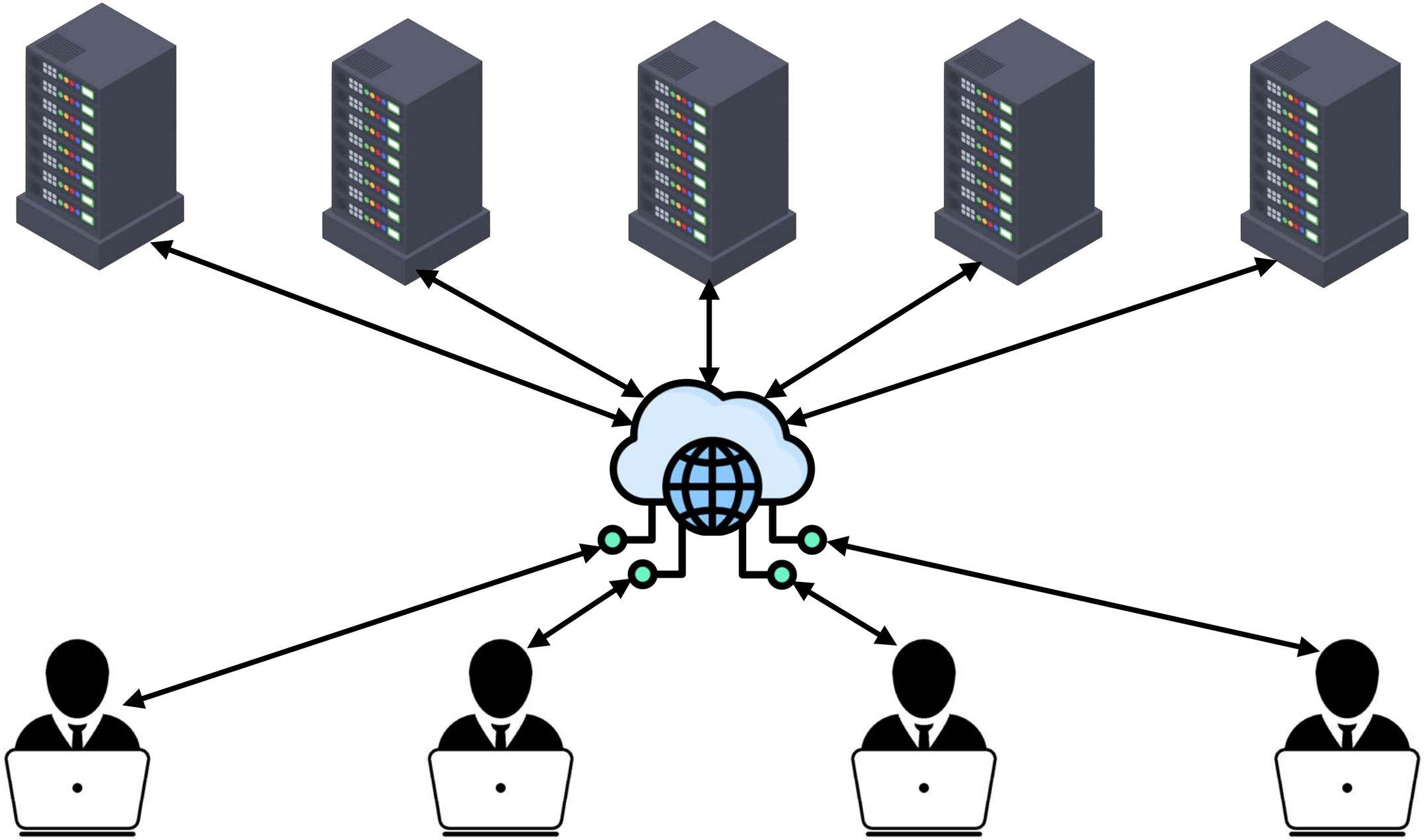
Centralised vs Distributed Systems

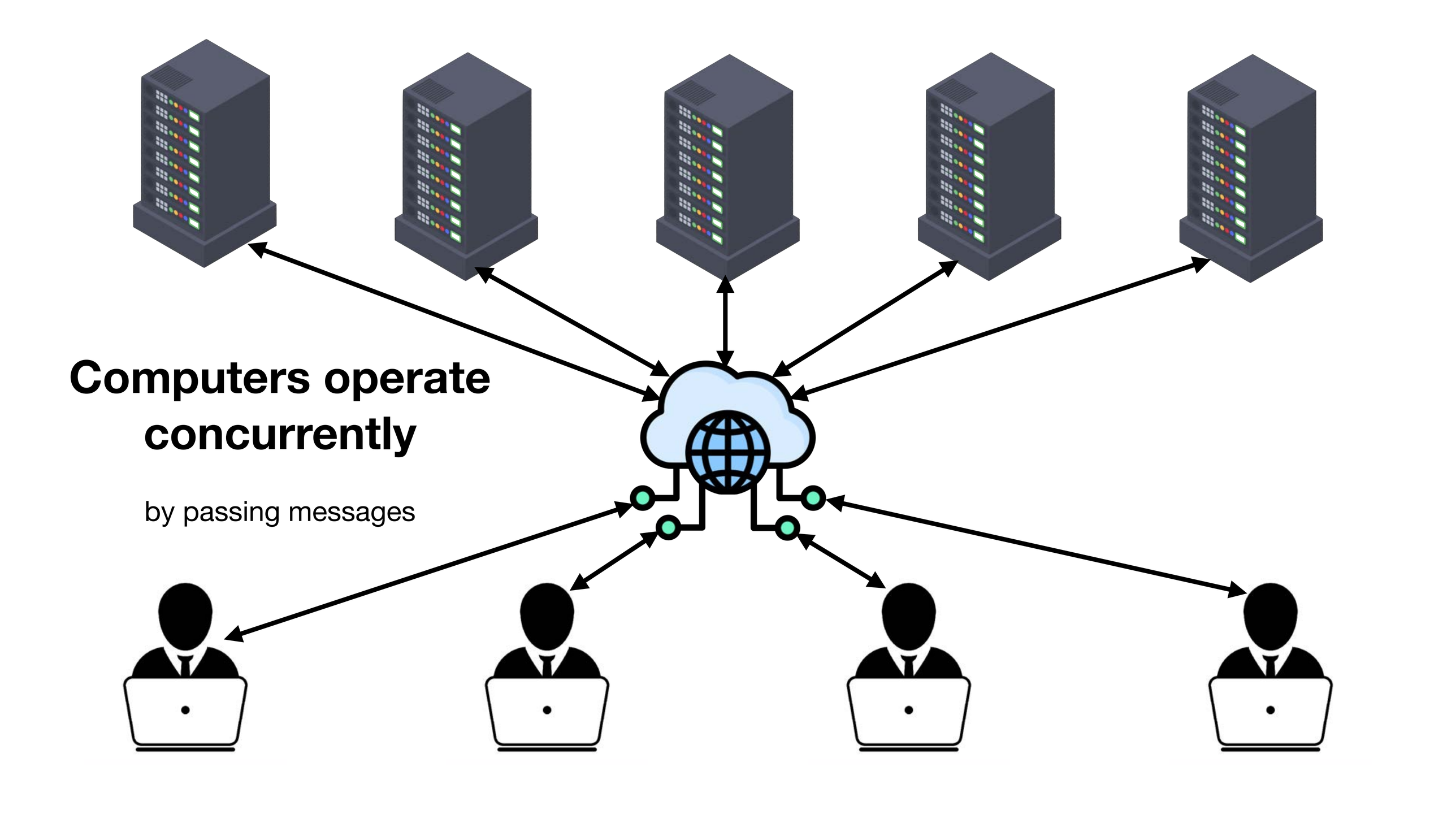
Centralised systems store state on a single computer.

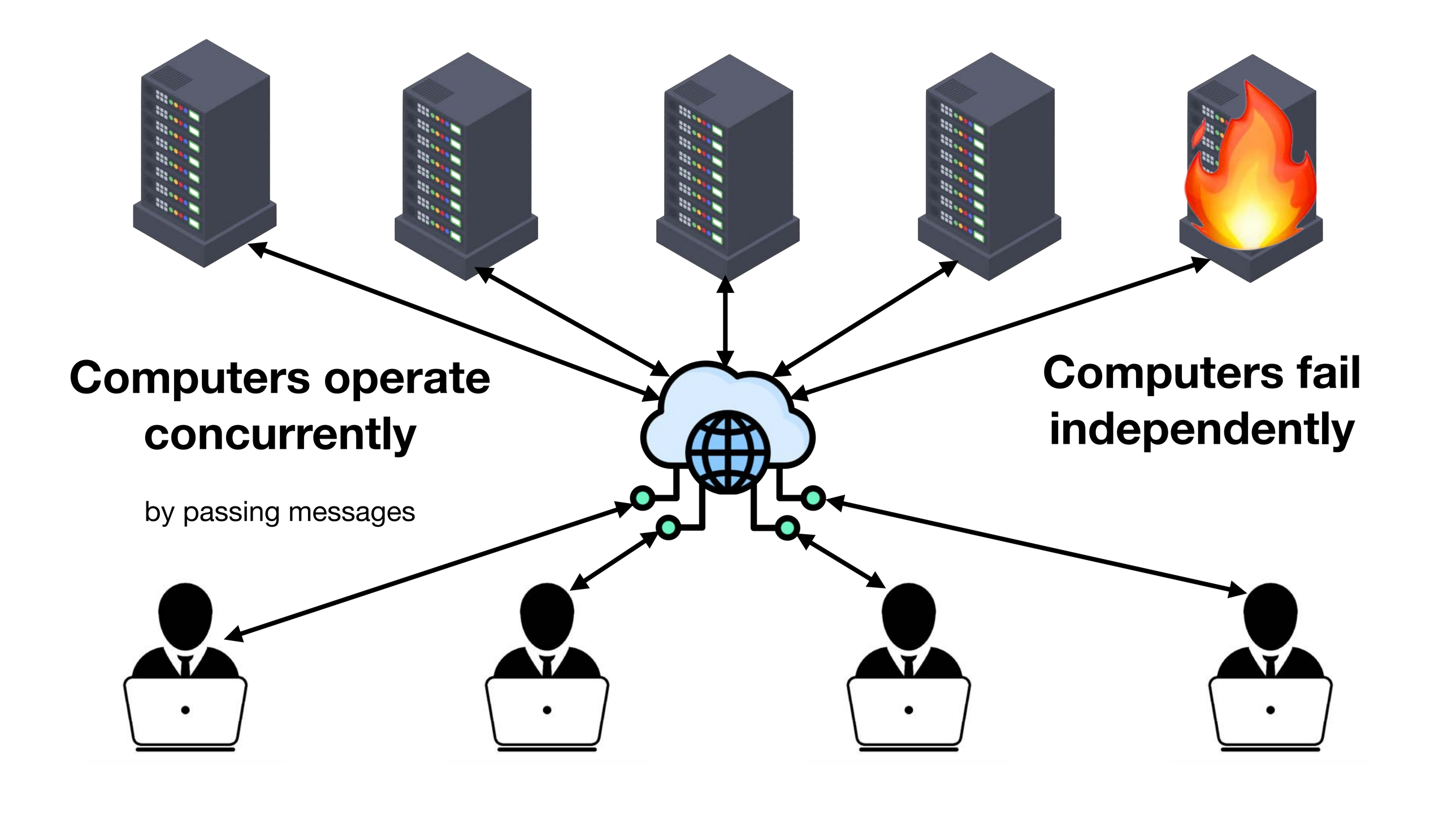
- Simpler
- Easier to reason about
- Can be faster for a single user

Distributed systems store state over multiple computers.

- Scalable
- Fault-tolerant
- Complex - hard to get right!




Computers operate concurrently
by passing messages


Computers operate concurrently
by passing messages
Computers fail independently

Examples of Distributed Systems

- **DNS** provides a lookup table of hostnames to IP addresses 

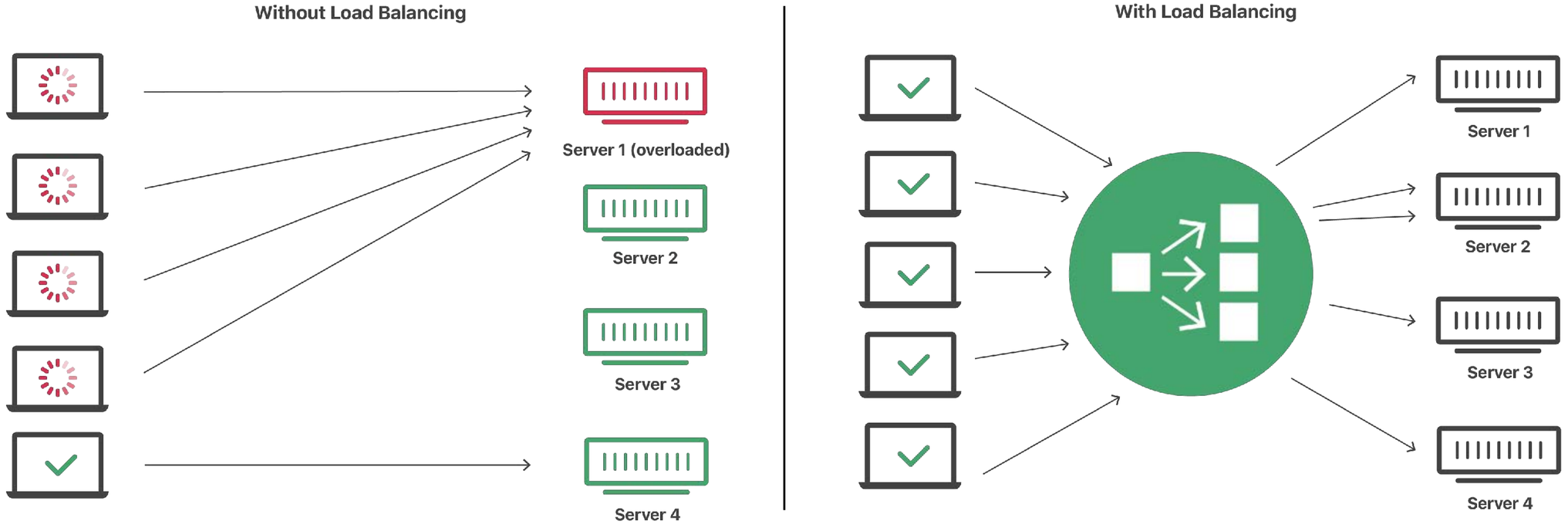
- **BitTorrent** is a peer-to-peer protocol for direct file sharing 
- **Bitcoin** uses blockchains or ledgers to maintain transactions 
- **Kubernetes** orchestrates app deployments across many machines 
- **Telecom Networking** was where the field originally developed 

Perfectly Balanced

Chapter 3

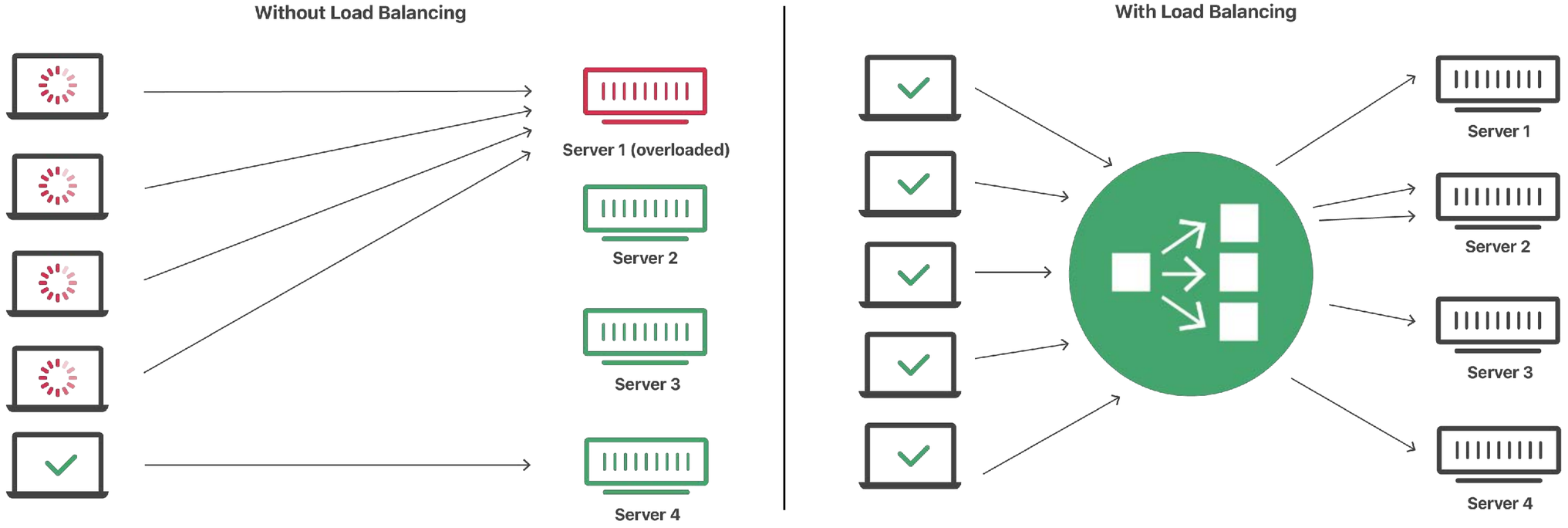
Load Balancers

- Divide network traffic among several servers
- Reduces strain on each server, increasing performance and reducing latency



Load Balancers

- Divide network traffic among several servers
- Reduces strain on each server, increasing performance and reducing latency



- *How to decide where to send each request?*

Static Load Balancing

Stateless Algorithms

Round Robin (RR)

Balances by operating on a rotation

e.g. Cluster of servers `A`, `B` and `C`
Forwards requests in the queue to servers `A`, `B`, `C`, `A`, `B`, `C`, `A`, `B`, `C`, `A`, ...

Static Load Balancing

Stateless Algorithms

Round Robin (RR)

Balances by operating on a rotation

e.g. Cluster of servers `A`,`B` and `C`
Forwards requests in the queue to servers `A`,`B`,`C`,`A`,`B`,`C`,`A`,`B`,`C`,`A`,...

- Simple to implement
- Works well if all servers are able to handle similar volumes of requests

Static Load Balancing

Stateless Algorithms

Round Robin (RR)

Balances by operating on a rotation

e.g. Cluster of servers `A`, `B` and `C`
Forwards requests in the queue to servers `A`, `B`, `C`, `A`, `B`, `C`, `A`, `B`, `C`, `A`, ...

Weighted Round Robin (WRR)

Proportional to predefined weights

e.g. Servers `A` (weight `3`) and `B` (weight `1`)
Forwards requests in the queue to servers `A`, `A`, `A`, `B`, `A`, `A`, `A`, `B`, `A`, `A`, `A`, ...

- Simple to implement
- Works well if all servers are able to handle similar volumes of requests

Static Load Balancing

Stateless Algorithms

Round Robin (RR)

Balances by operating on a rotation

e.g. Cluster of servers `A`, `B` and `C`
Forwards requests in the queue to servers `A`, `B`, `C`, `A`, `B`, `C`, `A`, `B`, `C`, `A`, ...

Weighted Round Robin (WRR)

Proportional to predefined weights

e.g. Servers `A` (weight `3`) and `B` (weight `1`)
Forwards requests in the queue to servers `A`, `A`, `A`, `B`, `A`, `A`, `A`, `B`, `A`, `A`, `A`, ...

- Simple to implement
- Works well if all servers are able to handle similar volumes of requests

These algorithms are also commonly used for **process and network scheduling**

Dynamic Load Balancing

Stateful Algorithms

Incorporating Real-time Data

Track each server's *current request load*, *idle capacity* or *response time*

Forward request to the 'best' server (min/max) at that point in time

Adaptive Weights

Assign weights based on current capacity and performance

Higher weight = receive more tasks

Dynamic Load Balancing

Stateful Algorithms

Incorporating Real-time Data

Track each server's *current request load*, *idle capacity* or *response time*

Forward request to the 'best' server (min/max) at that point in time

Adaptive Weights

Assign weights based on current capacity and performance

Higher weight = receive more tasks

Typically, "agent" runs on each server, measuring CPU/RAM utilisation - dynamic LB queries agents to get data

- More difficult to configure
- Continuously adapts to keep distribution even and efficient

Dynamic Load Balancing

Stateful Algorithms

Incorporating Real-time Data

Track each server's *current request load*, *idle capacity* or *response time*

Forward request to the 'best' server (min/max) at that point in time

Adaptive Weights

Assign weights based on current capacity and performance

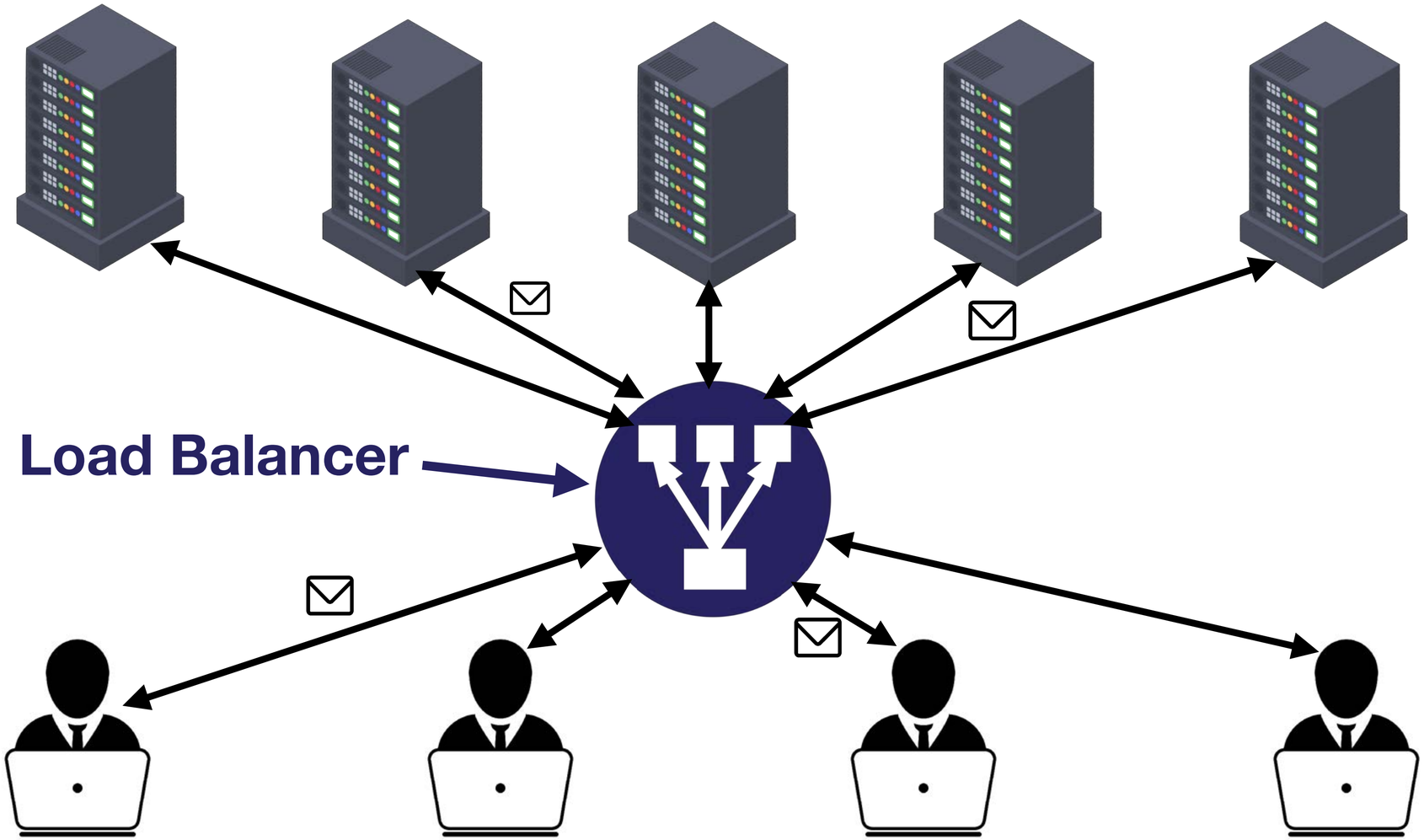
Higher weight = receive more tasks

Typically, "agent" runs on each server, measuring CPU/RAM utilisation - dynamic LB queries agents to get data

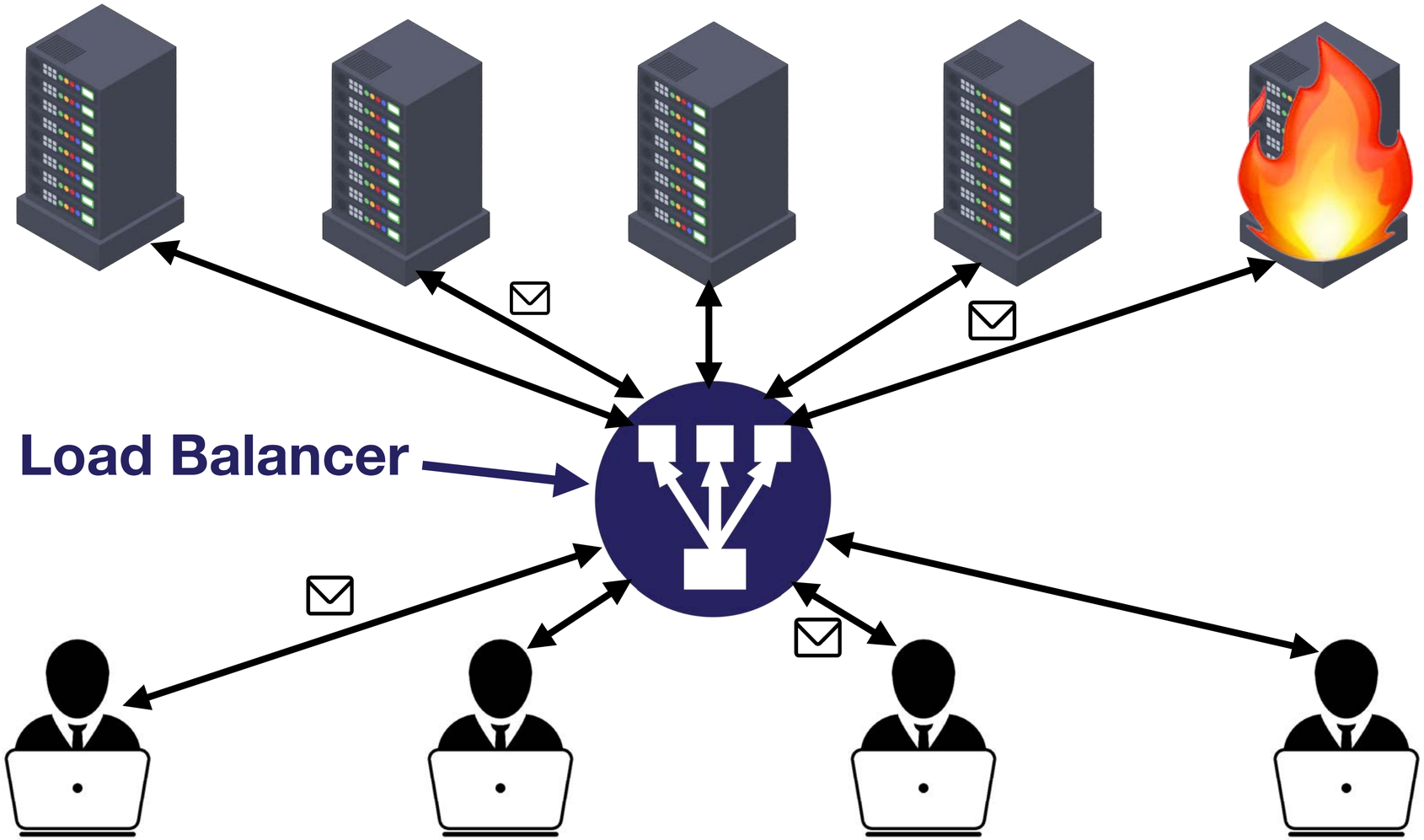
- More difficult to configure
- Continuously adapts to keep distribution even and efficient

Cutting-edge dynamic LB algorithms use **predictive analytics and machine learning**

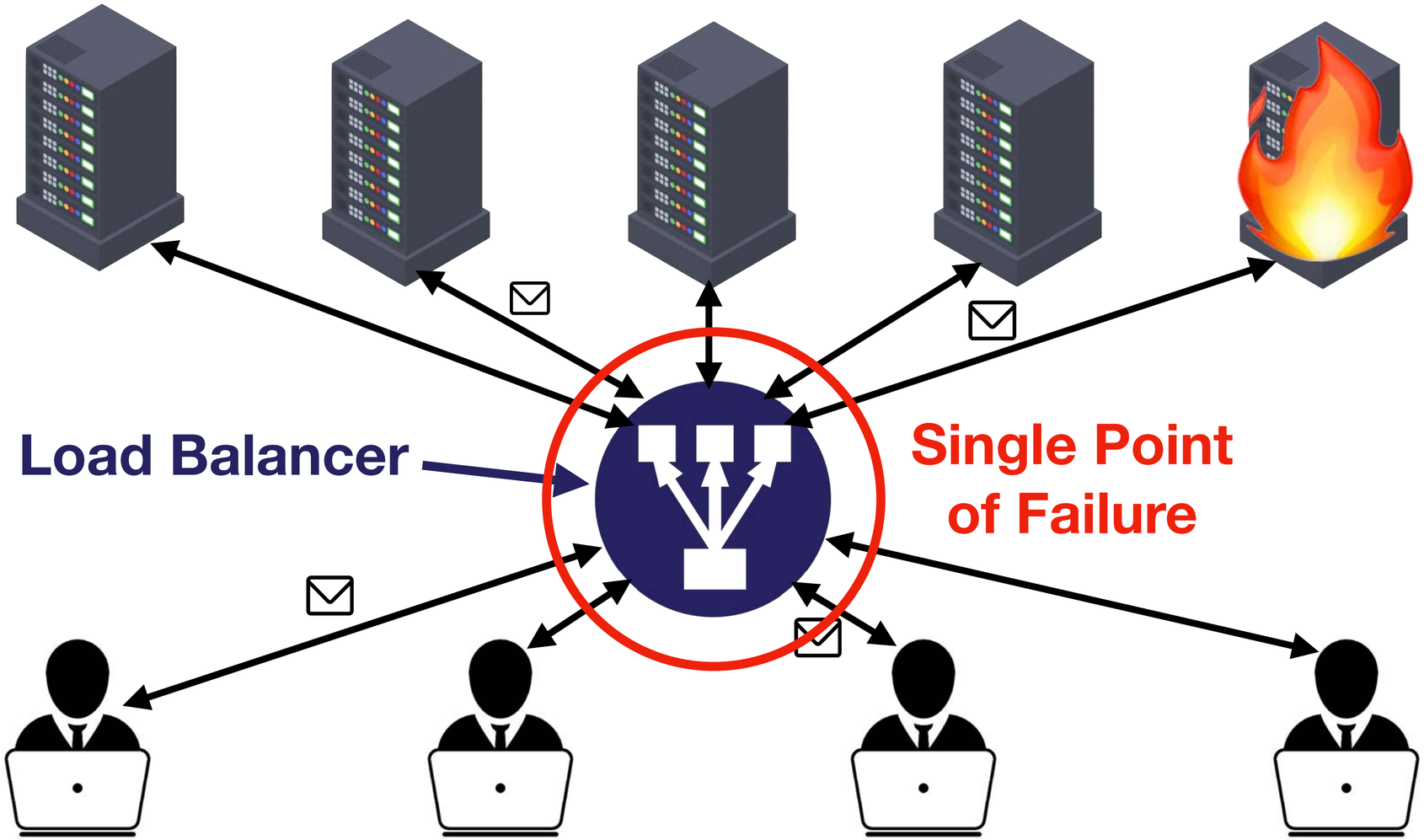
Single Load Balancer



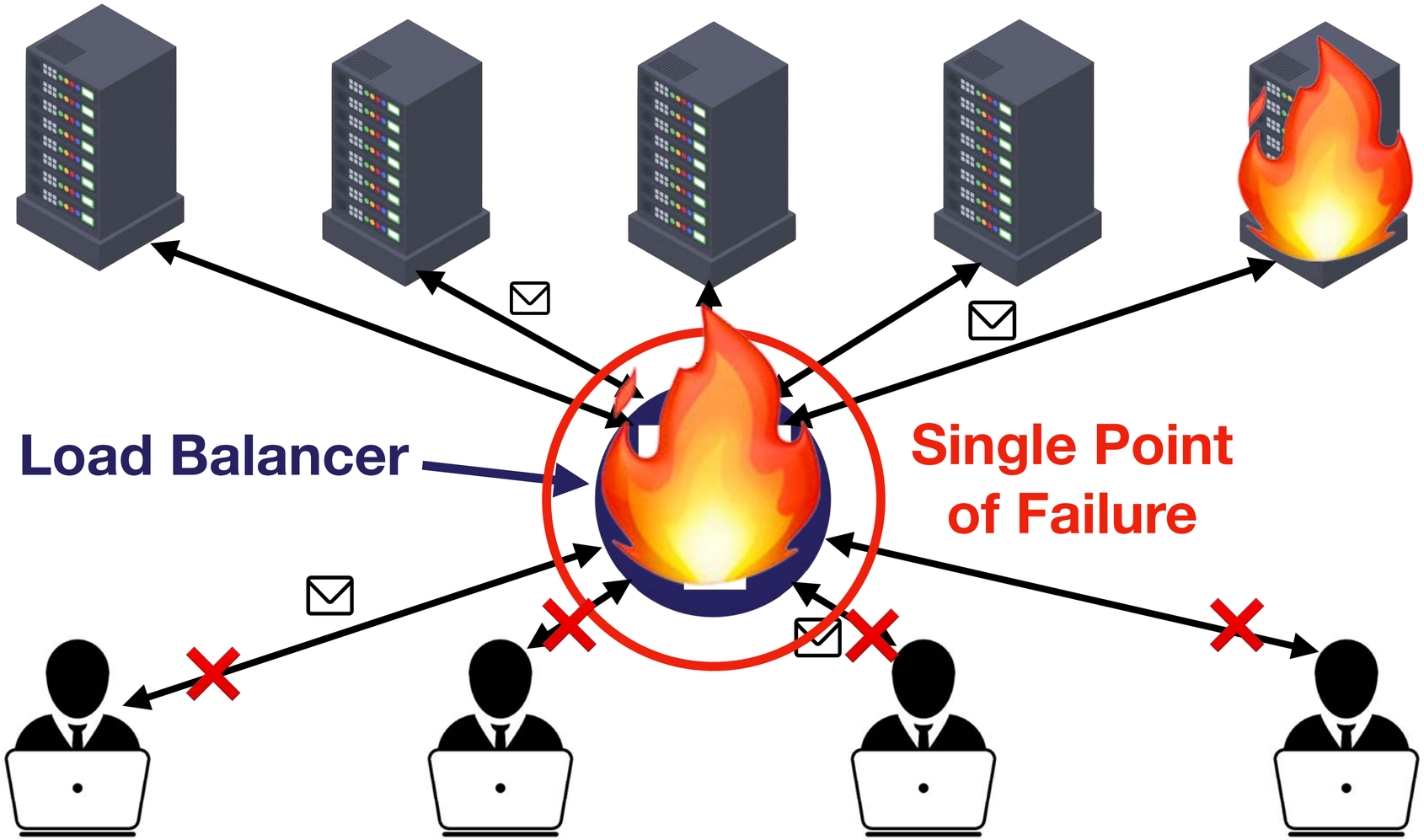
Single Load Balancer



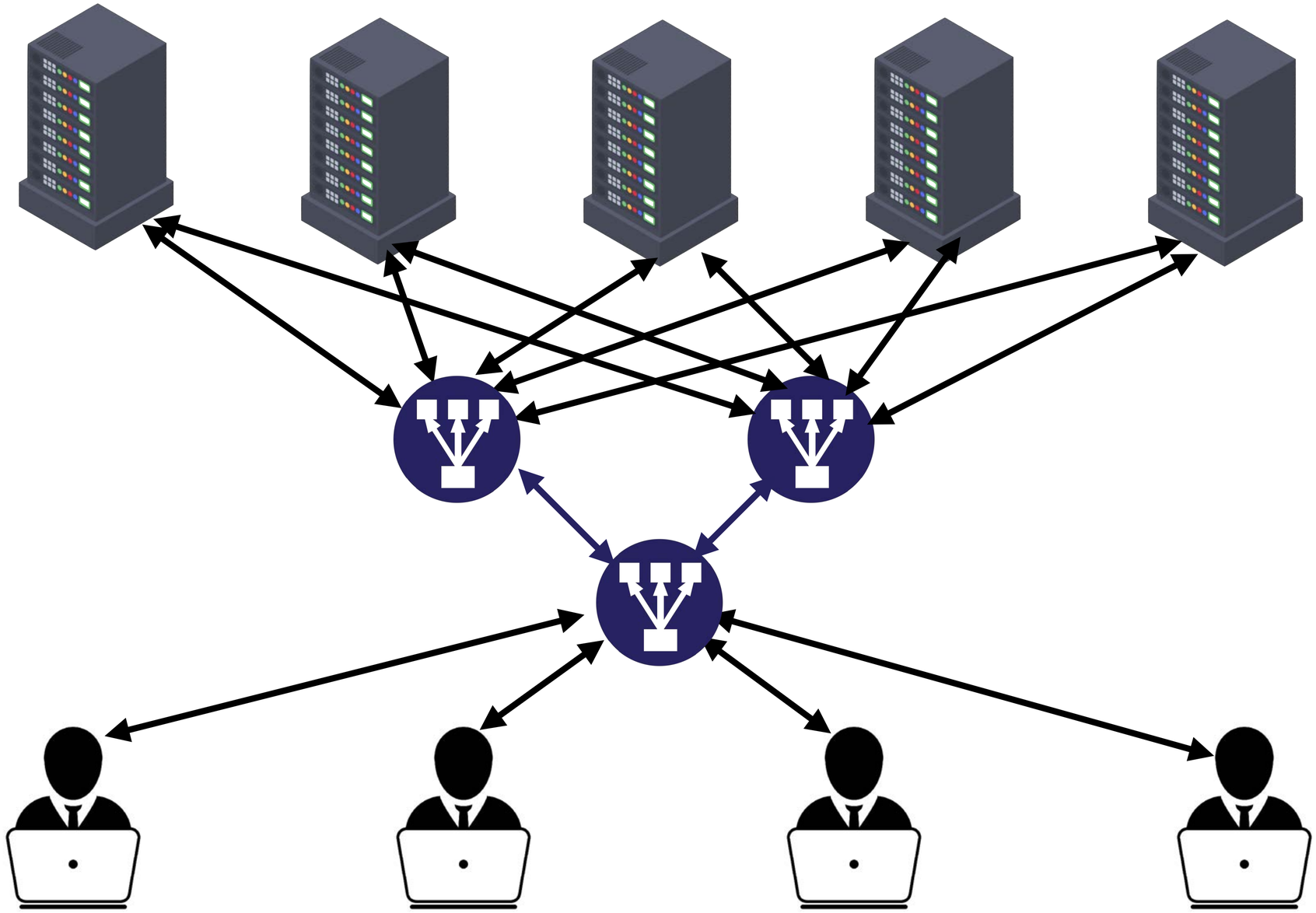
Single Load Balancer



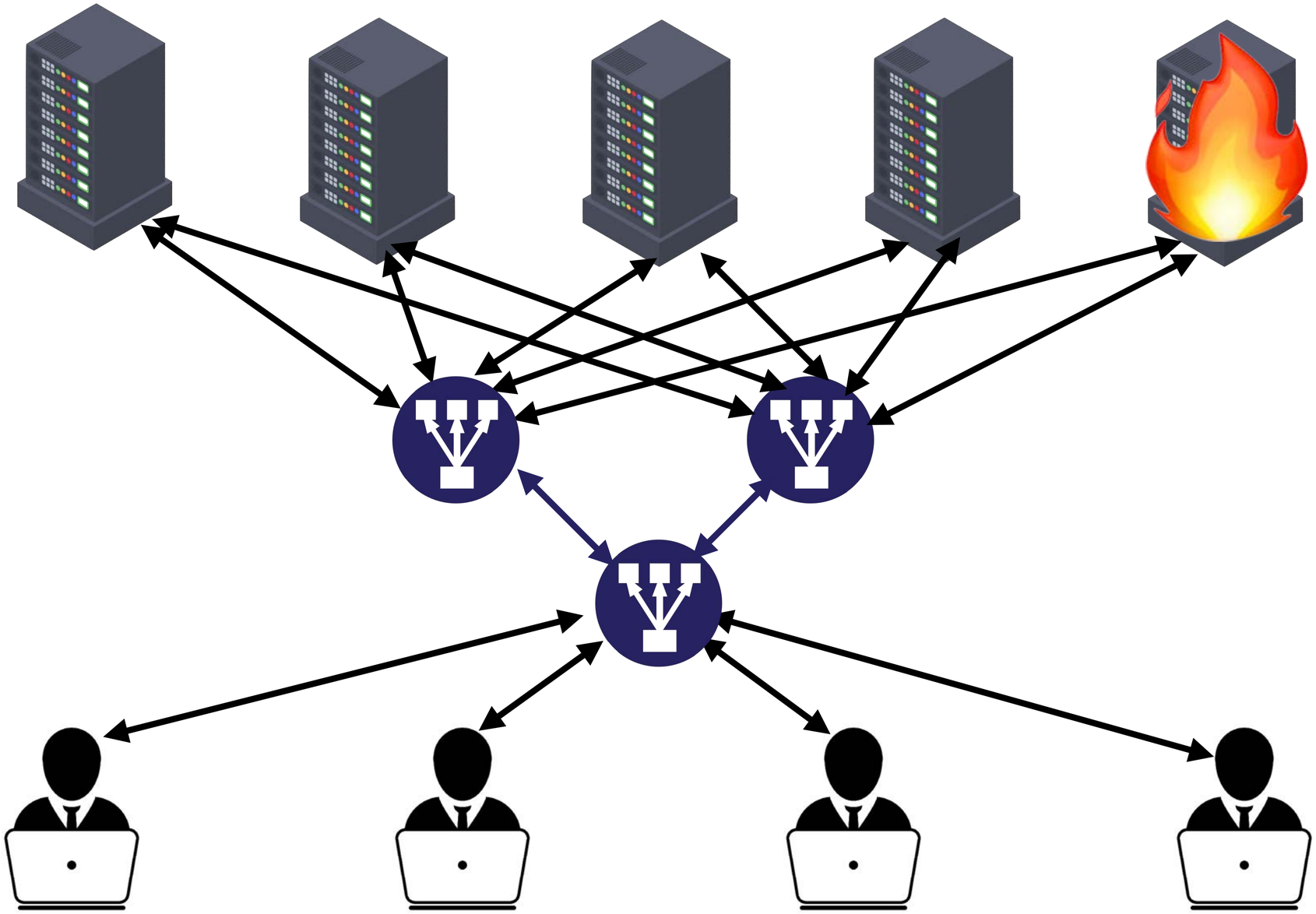
Single Load Balancer



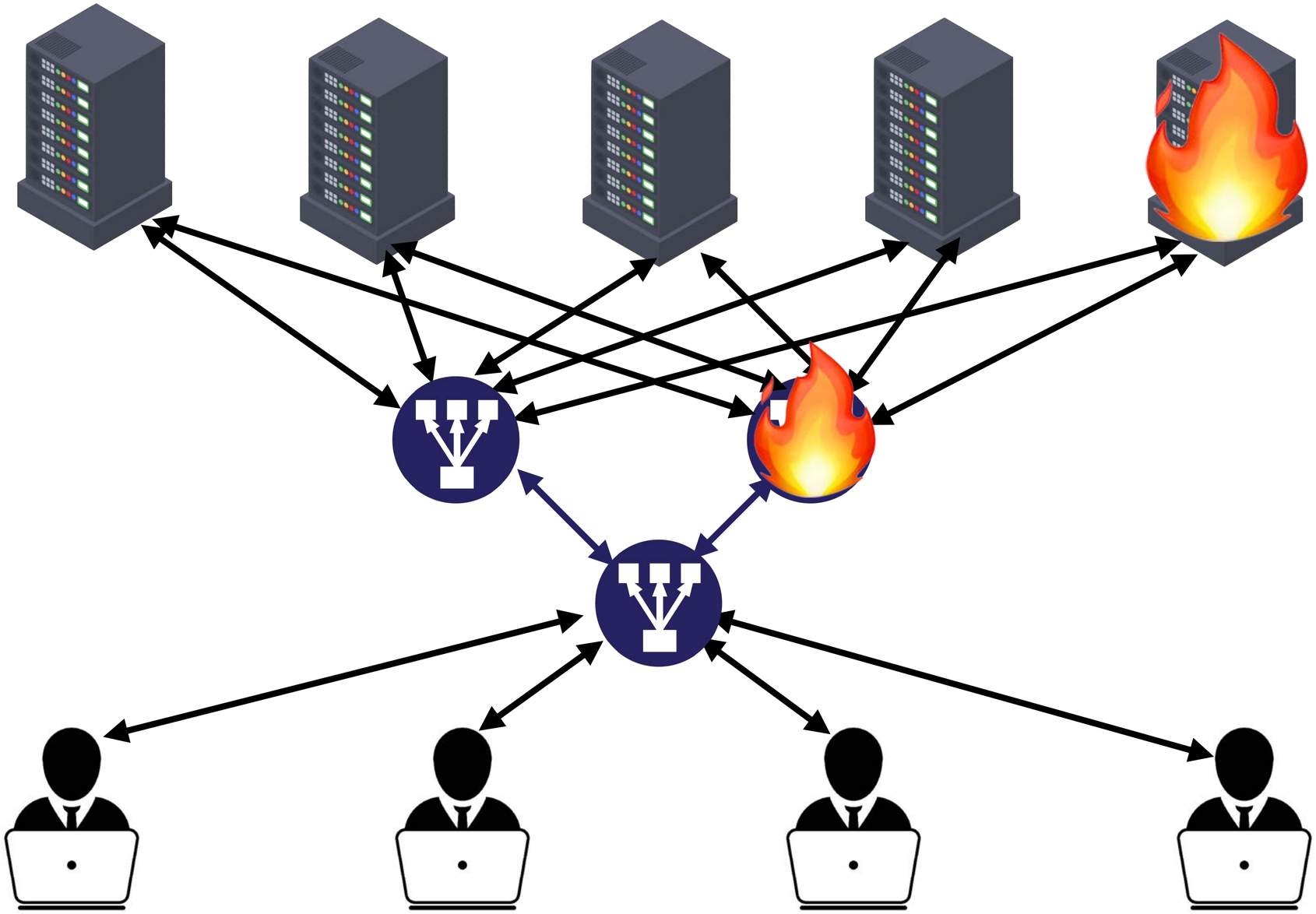
Scaling out Load Balancers



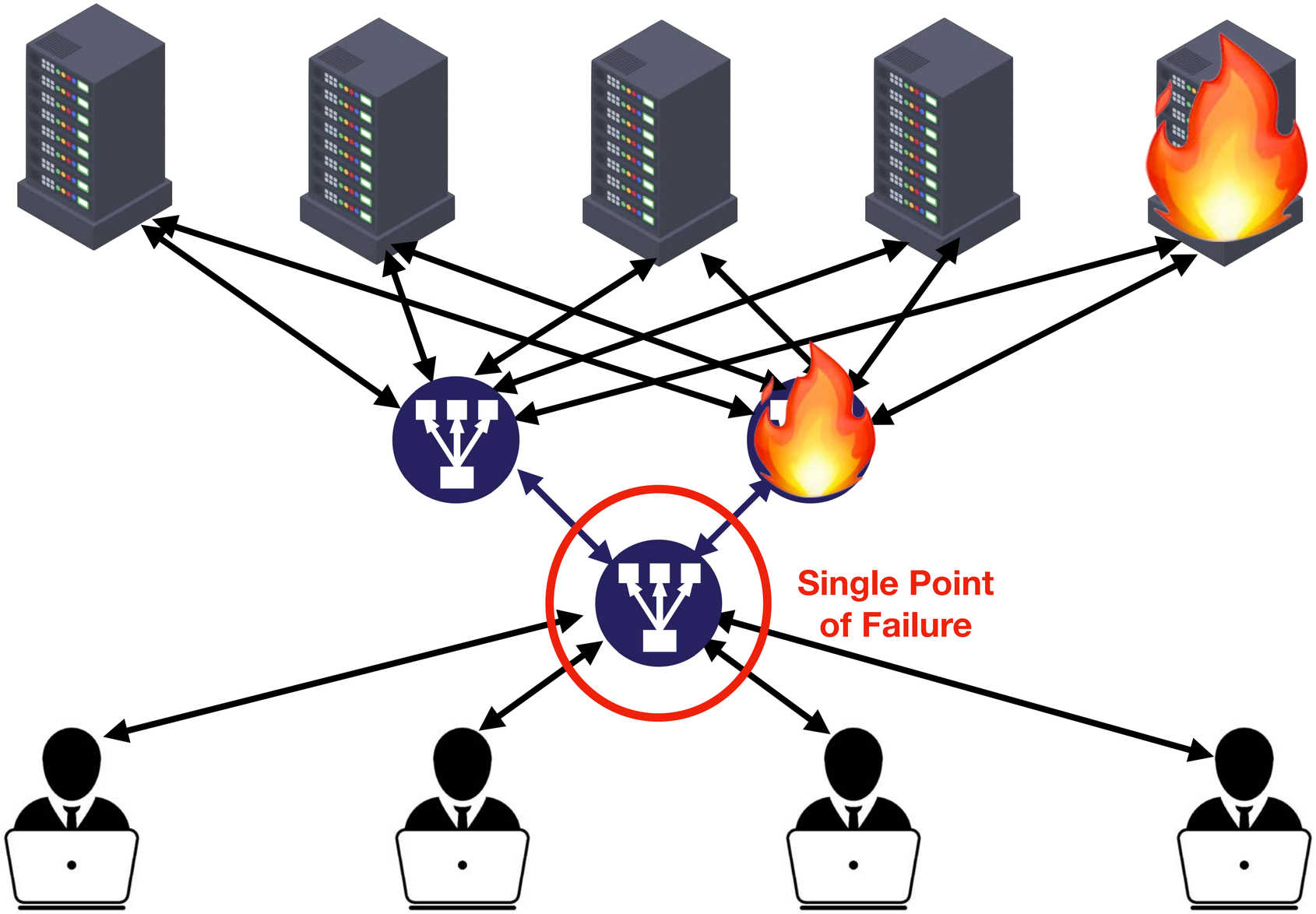
Scaling out Load Balancers



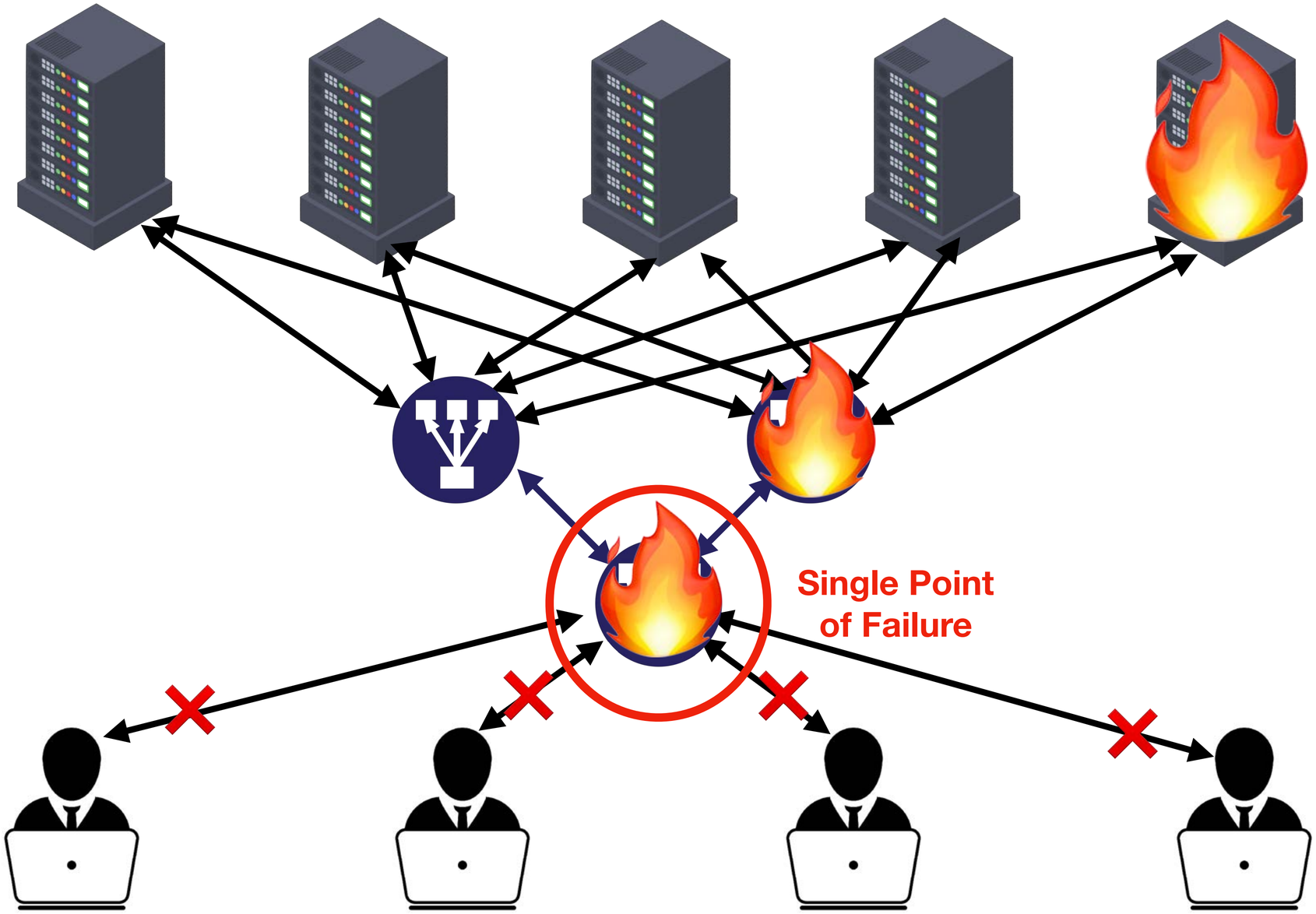
Scaling out Load Balancers



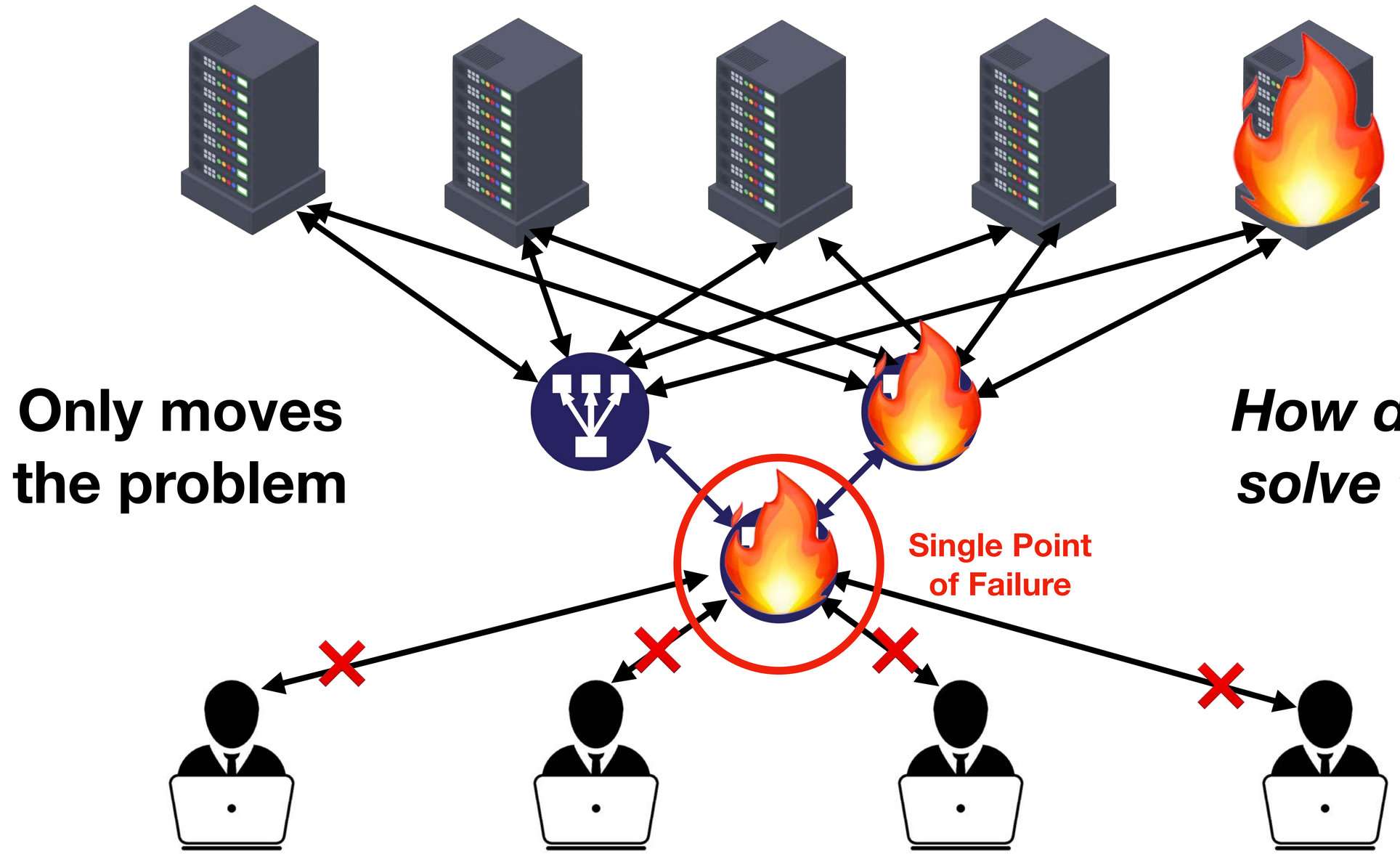
Scaling out Load Balancers



Scaling out Load Balancers



Scaling out Load Balancers



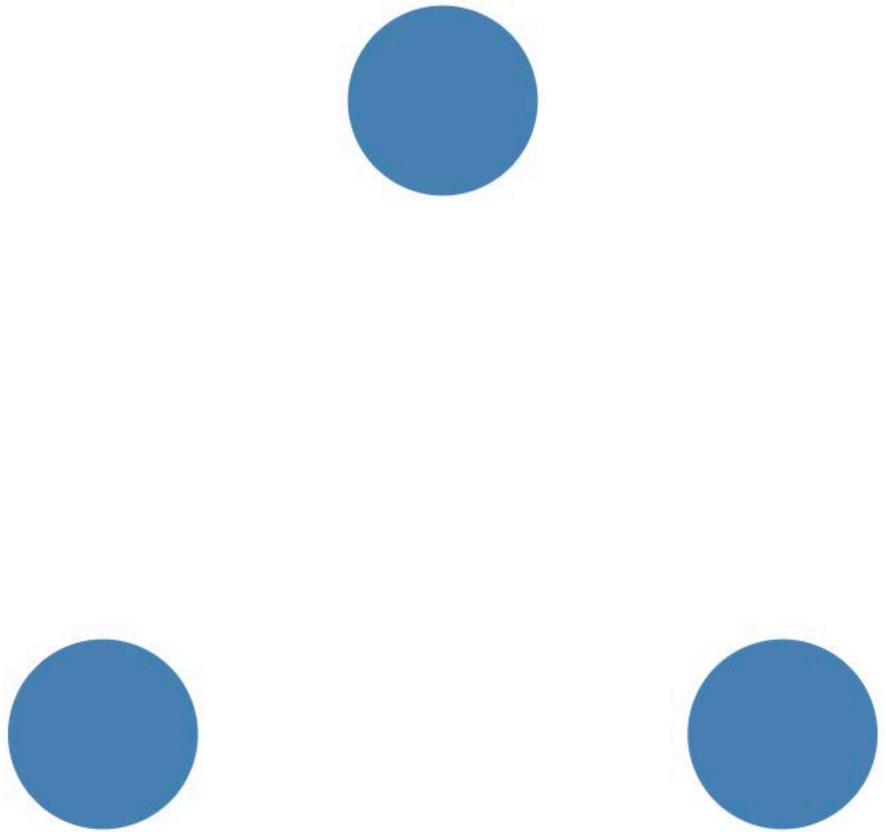
Democracy

Chapter 4

Leader Election

Initial Configuration

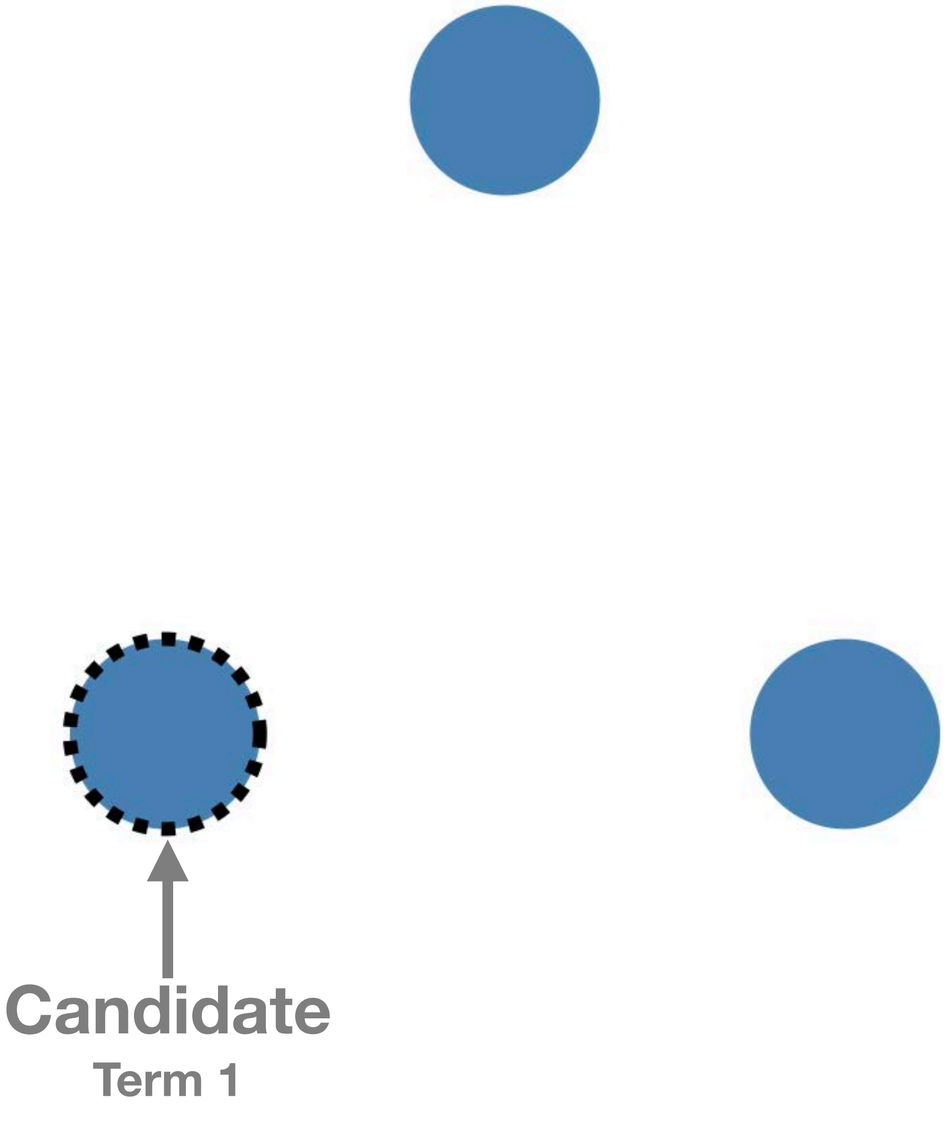
- Nodes (servers) can be in 3 states *Follower*, *Candidate* or *Leader*
- All nodes start in the follower state



Leader Election

Candidacy

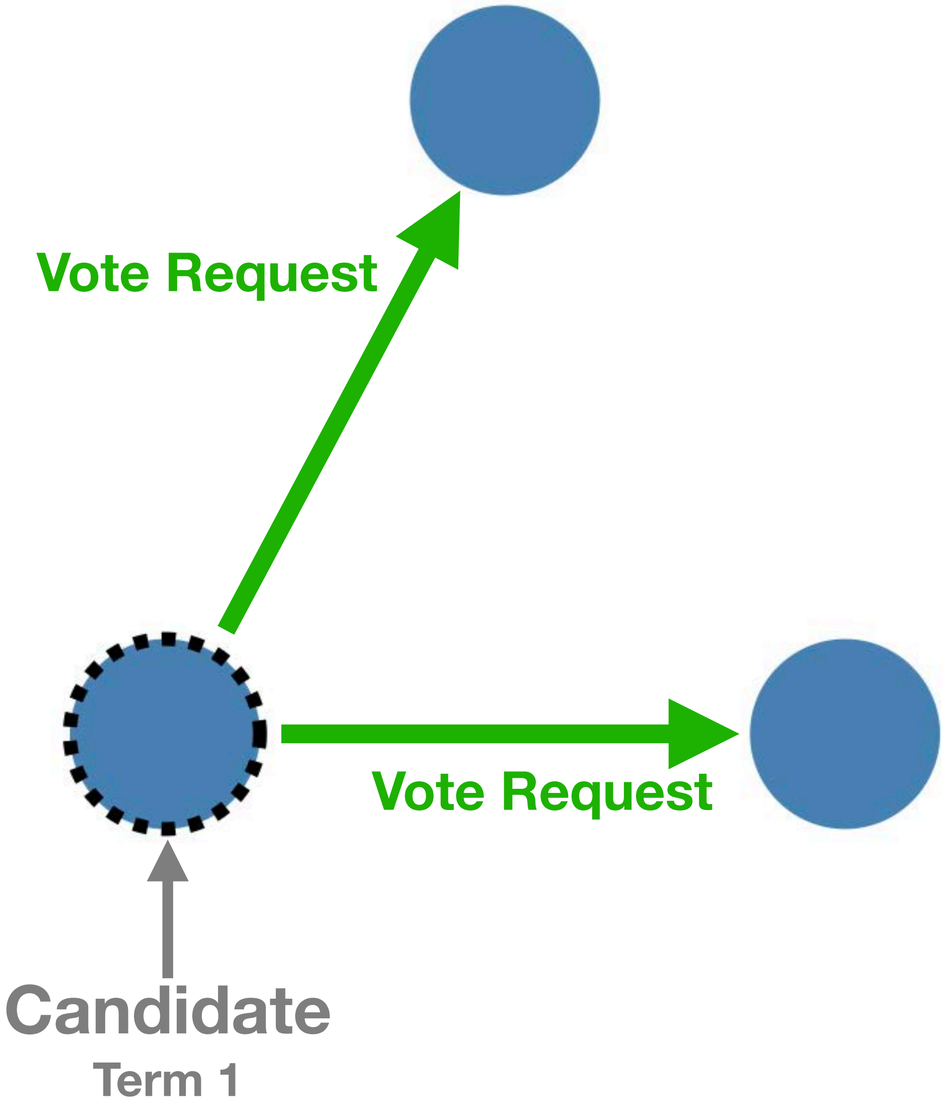
- Nodes (servers) can be in 3 states *Follower*, *Candidate* or *Leader*
- All nodes start in the follower state
- If followers don't hear from a leader then they can become a candidate (voting for itself)



Leader Election

Candidacy

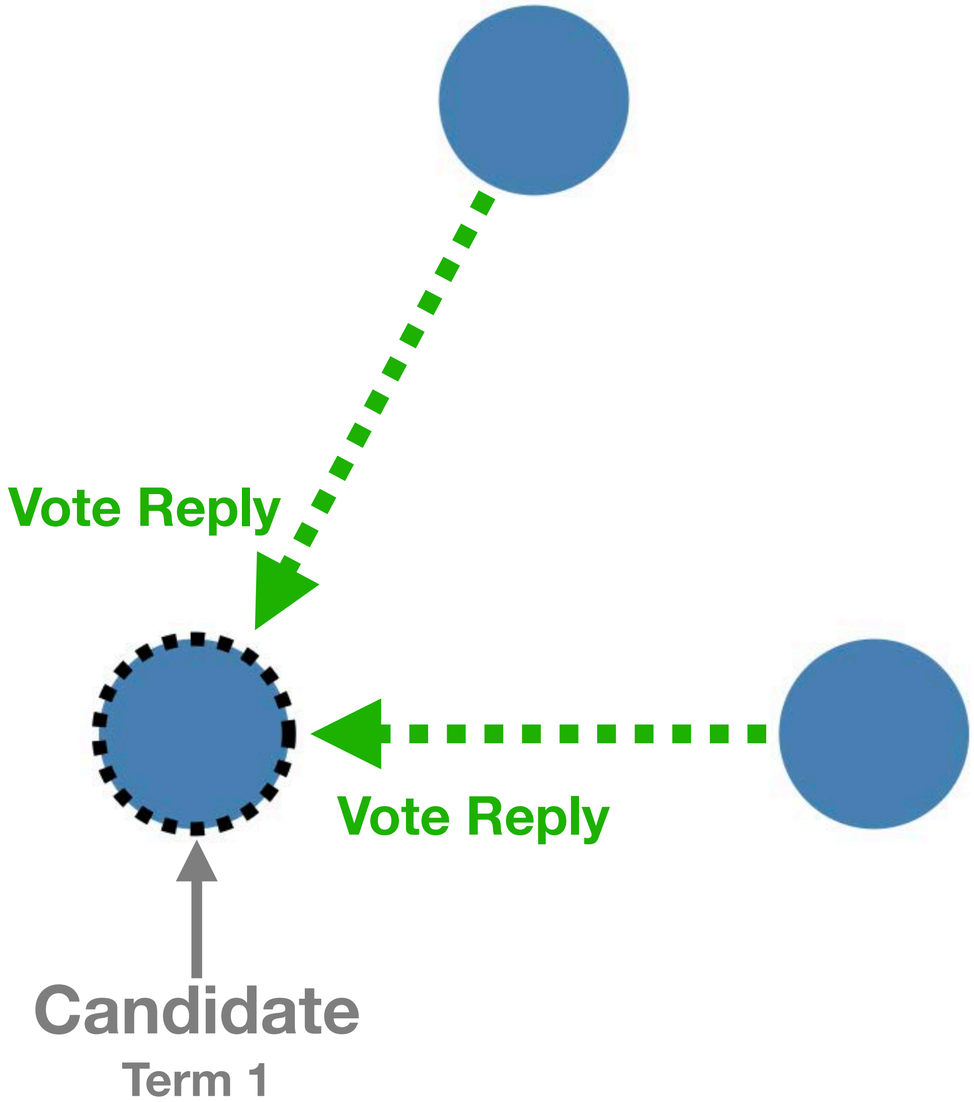
- Nodes (servers) can be in 3 states *Follower*, *Candidate* or *Leader*
- All nodes start in the follower state
- If followers don't hear from a leader then they can become a candidate (voting for itself)
- The candidate then requests votes from other nodes



Leader Election

Candidacy

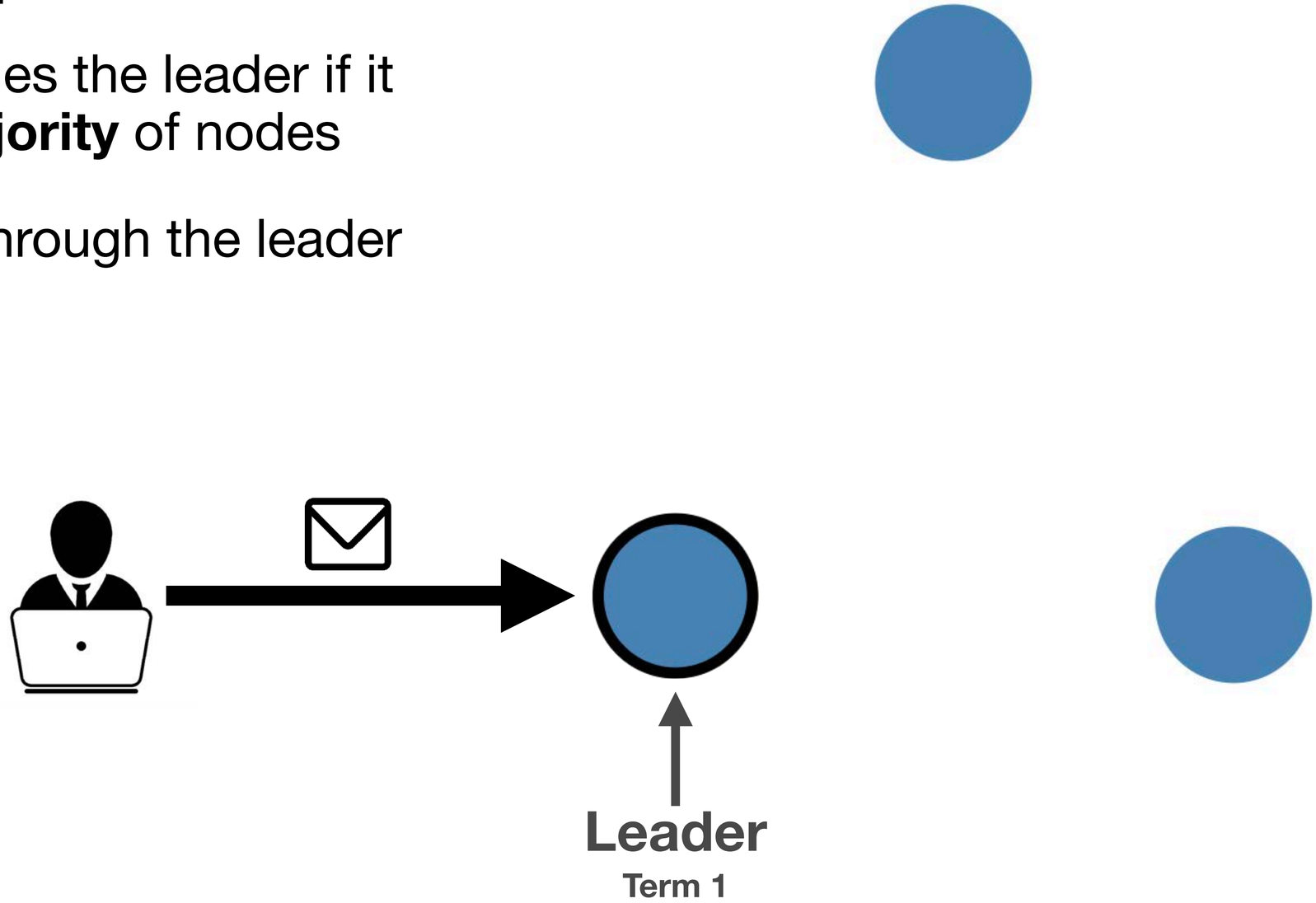
- Nodes (servers) can be in 3 states *Follower*, *Candidate* or *Leader*
- All nodes start in the follower state
- If followers don't hear from a leader then they can become a candidate (voting for itself)
- The candidate then requests votes from other nodes
- Nodes will reply with their vote



Leader Election

Successful Election

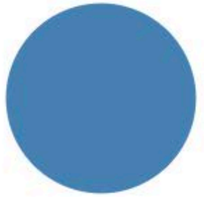
- The candidate becomes the leader if it gets votes from a **majority** of nodes
- All requests now go through the leader



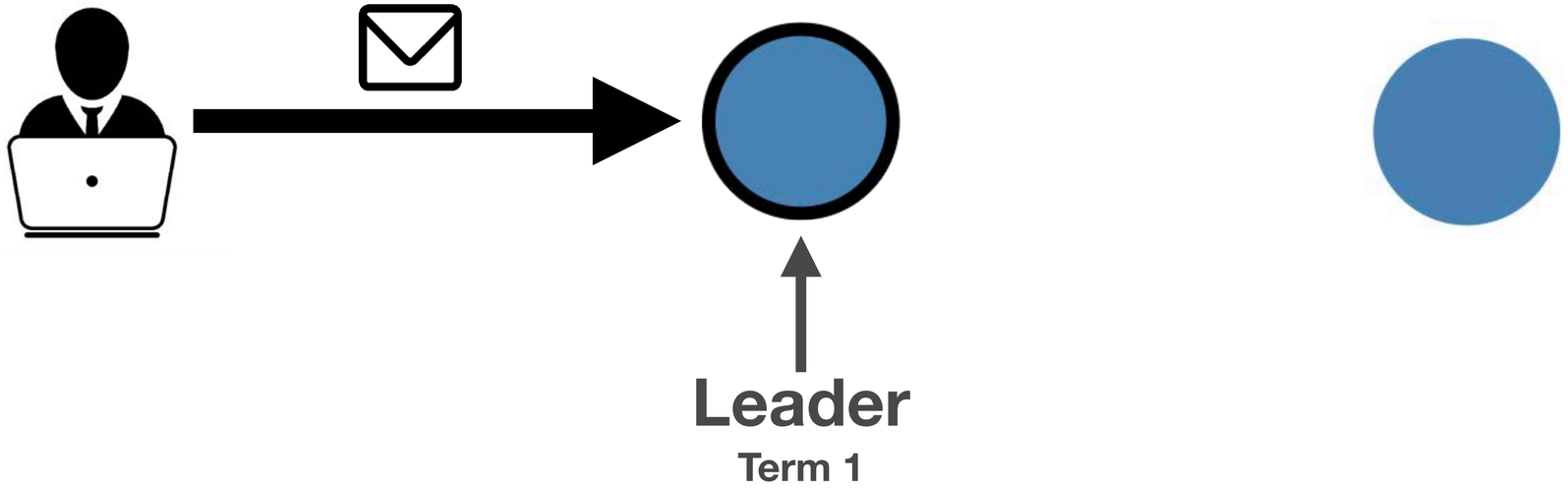
Leader Election

Successful Election

- The candidate becomes the leader if it gets votes from a **majority** of nodes
- All requests now go through the leader



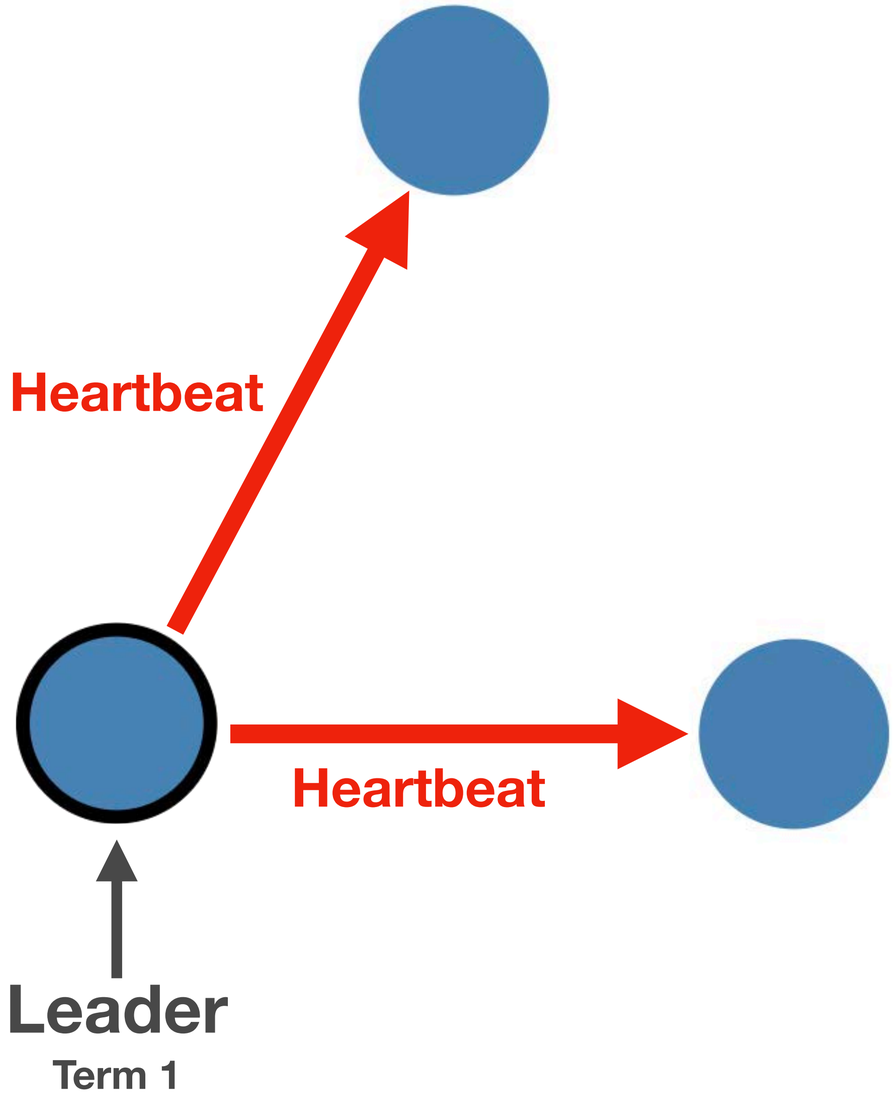
Protocol guarantees at most one leader can be elected
Unsuccessful election term if majority offline or split vote



Leader Election

Heartbeats

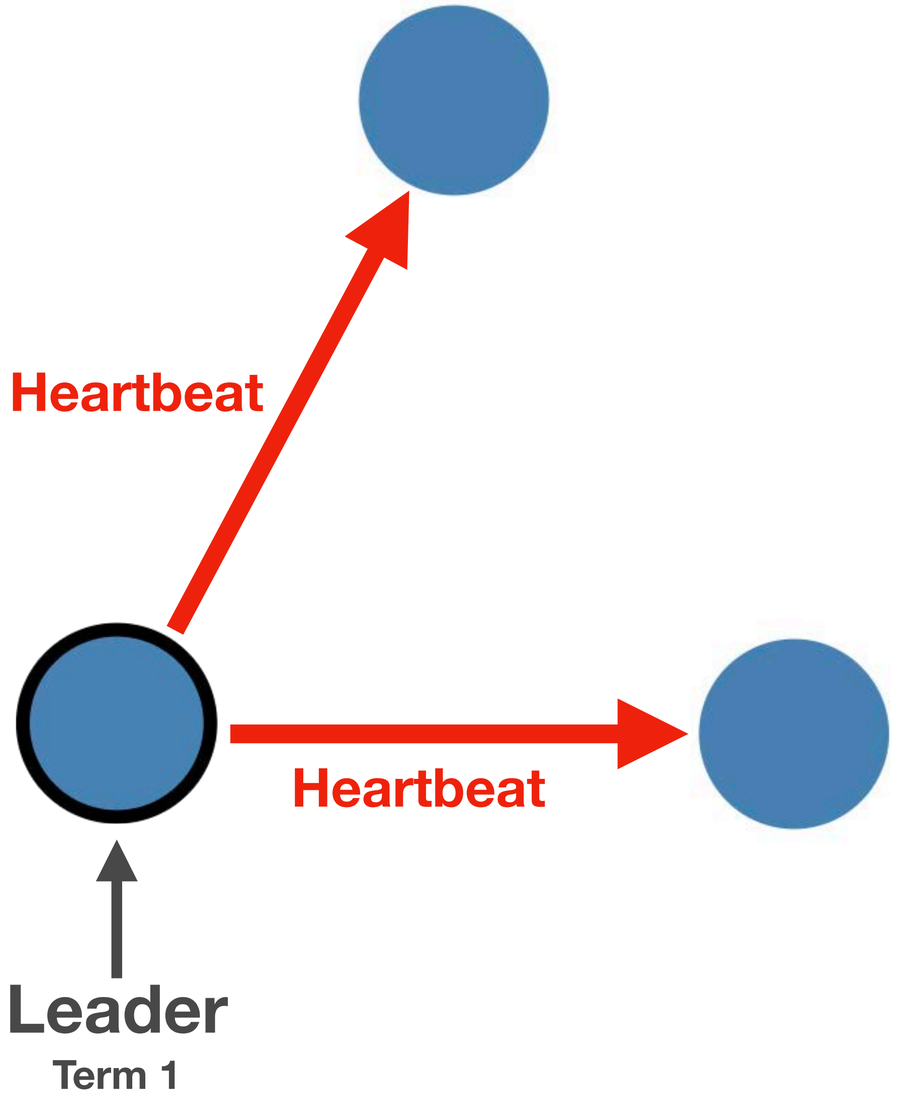
- The candidate becomes the leader if it gets votes from a **majority** of nodes
- All requests now go through the leader
- The leader sends heartbeat messages in intervals to all followers while it is alive



Leader Election

Heartbeats

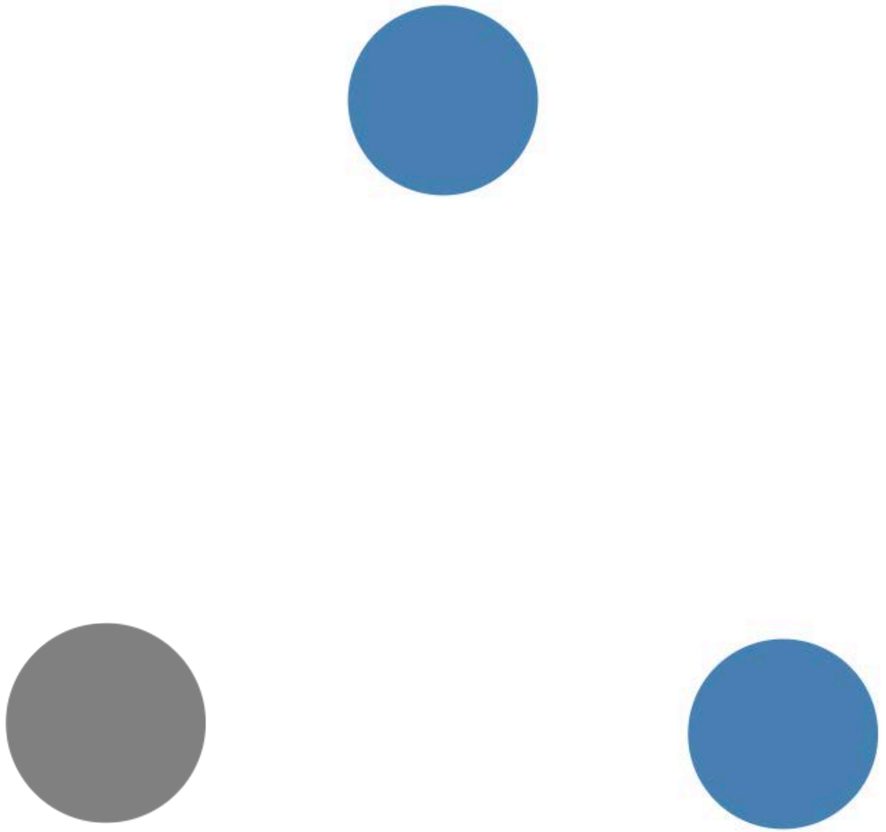
- The candidate becomes the leader if it gets votes from a **majority** of nodes
- All requests now go through the leader
- The leader sends heartbeat messages in intervals to all followers while it is alive
- This election *term* continues until a follower stops receiving heartbeats and becomes a candidate



Leader Election

Fault Tolerance

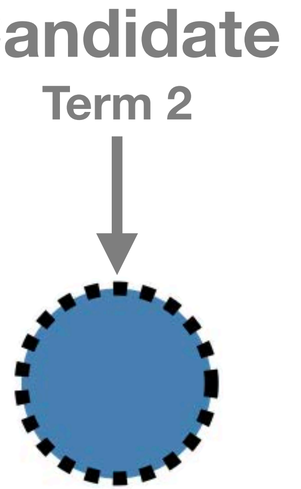
- The candidate becomes the leader if it gets votes from a **majority** of nodes
- All requests now go through the leader
- The leader sends heartbeat messages in intervals to all followers while it is alive
- This election *term* continues until a follower stops receiving heartbeats and becomes a candidate
- *Let's stop the leader and see what happens...*



Leader Election

Fault Tolerance

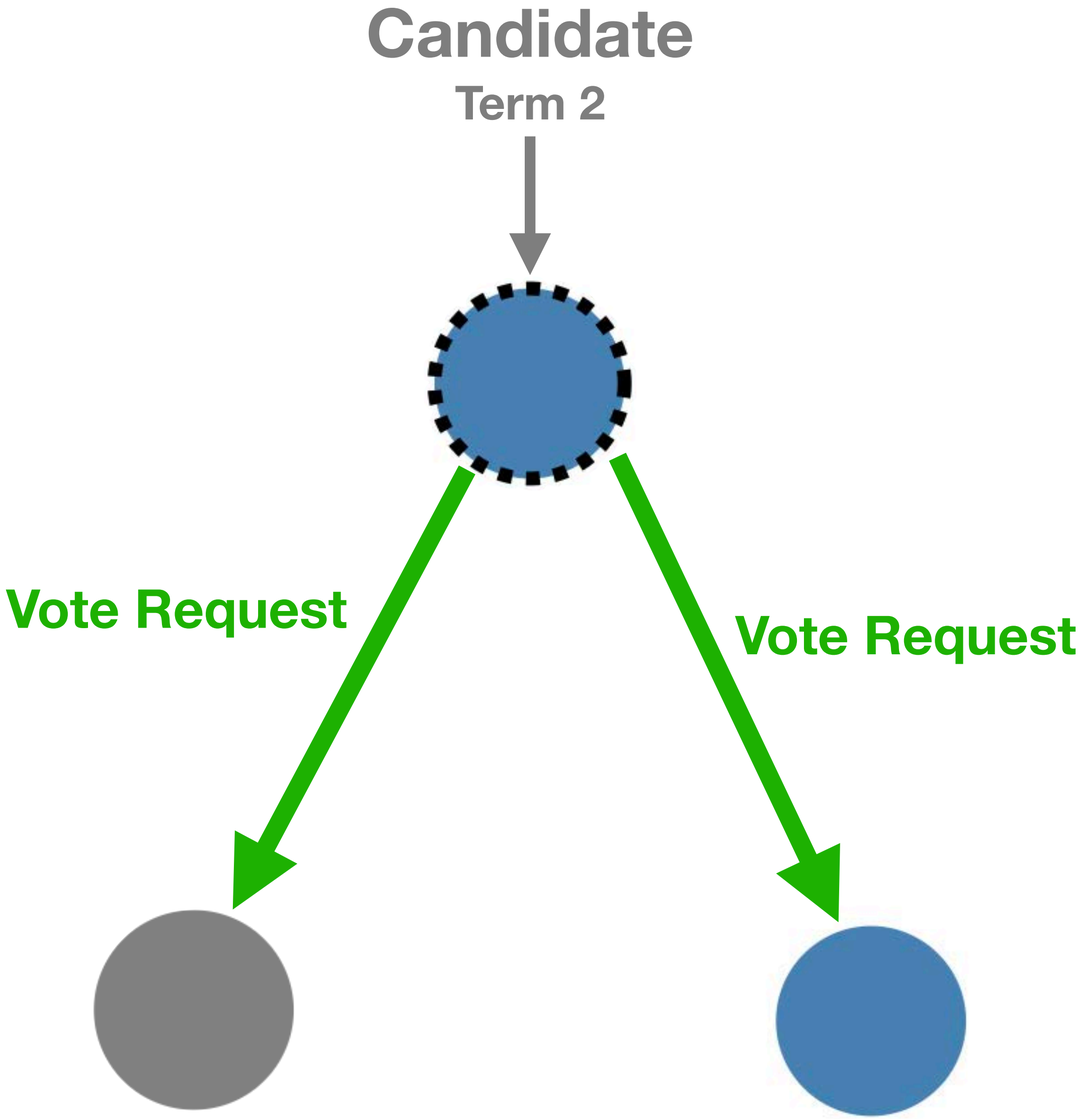
- Another node doesn't hear from the leader within the timeout and becomes a candidate



Leader Election

Fault Tolerance

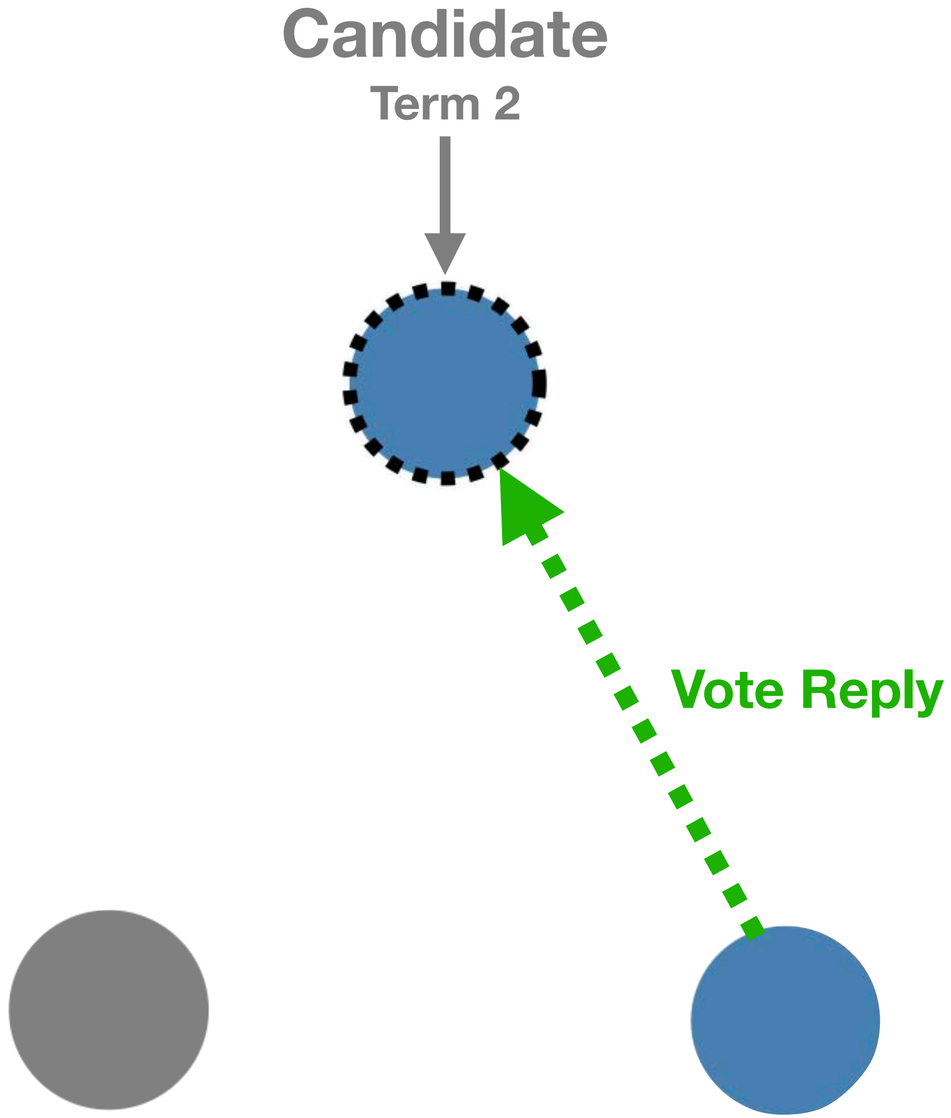
- Another node doesn't hear from the leader within the timeout and becomes a candidate
- It votes for itself and requests votes



Leader Election

Fault Tolerance

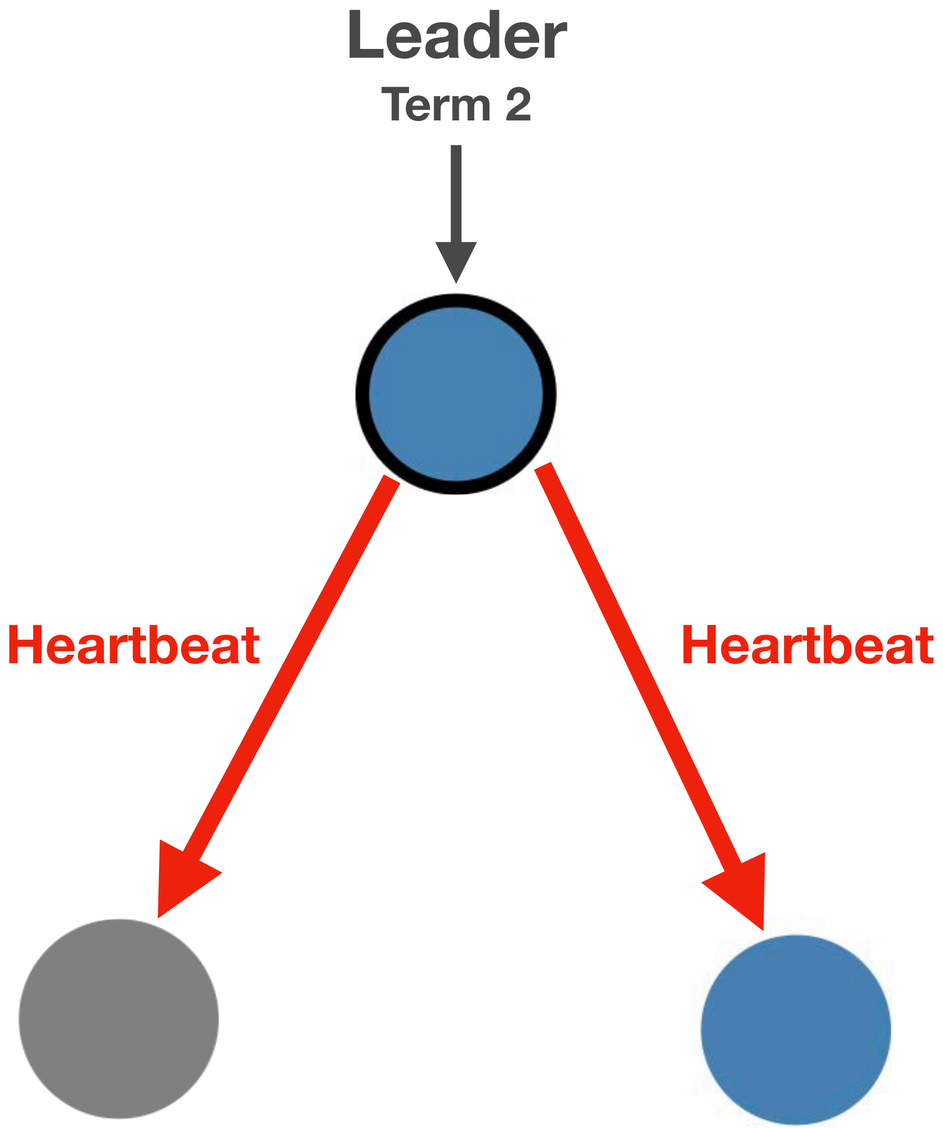
- Another node doesn't hear from the leader within the timeout and becomes a candidate
- It votes for itself and requests votes
- It only receives one reply, but this is enough for a **majority** (2 out of 3)



Leader Election

Fault Tolerance

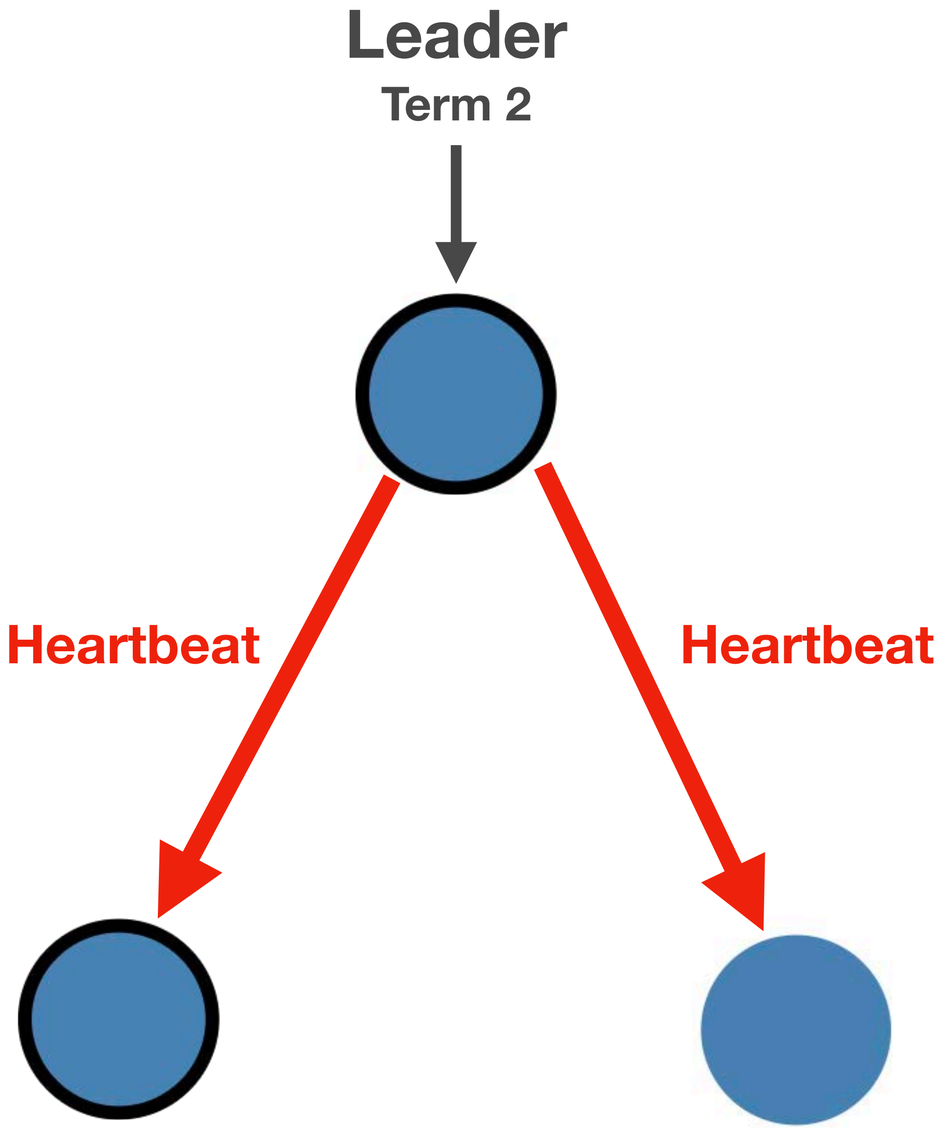
- Another node doesn't hear from the leader within the timeout and becomes a candidate
- It votes for itself and requests votes
- It only receives one reply, but this is enough for a **majority** (2 out of 3)
- A leader is elected for the next *term*



Leader Election

Neutralising Old Leaders

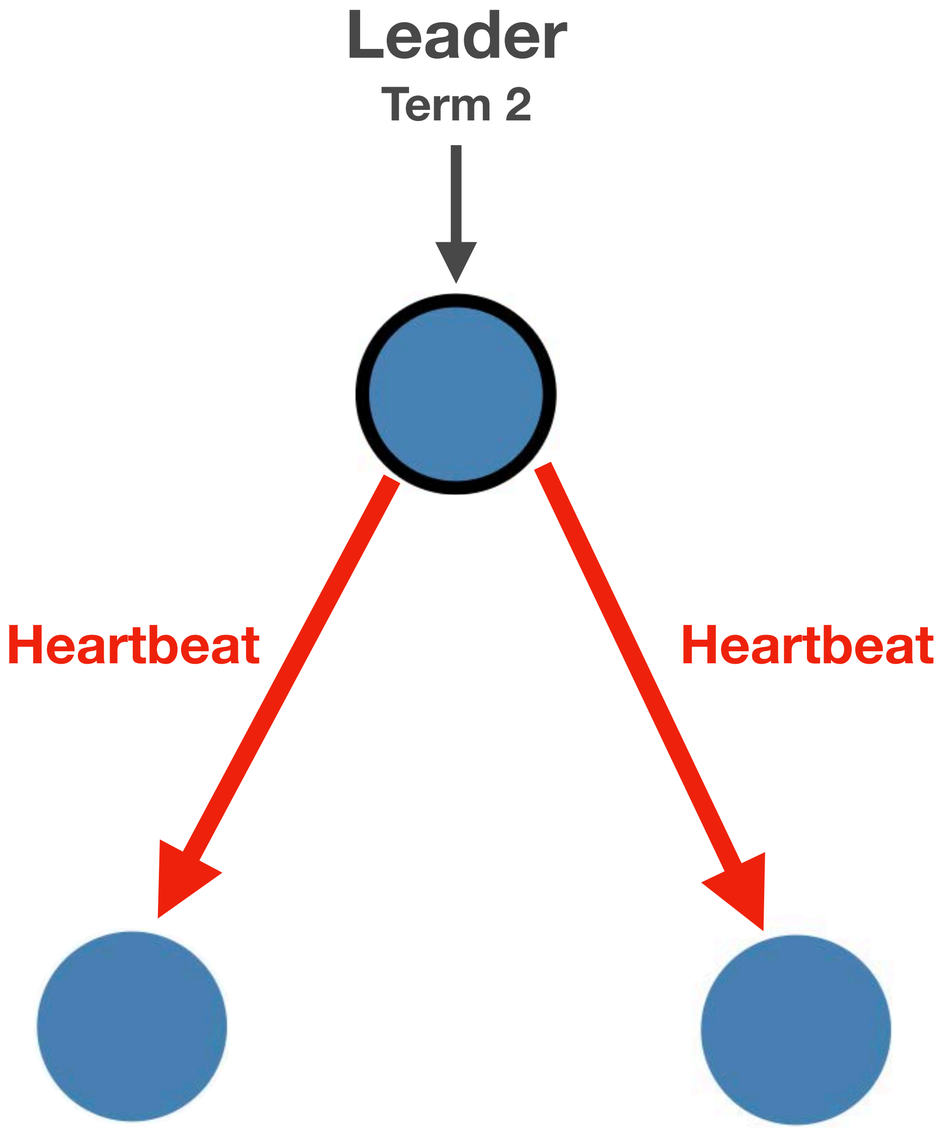
- *What if the old leader is restarted?*



Leader Election

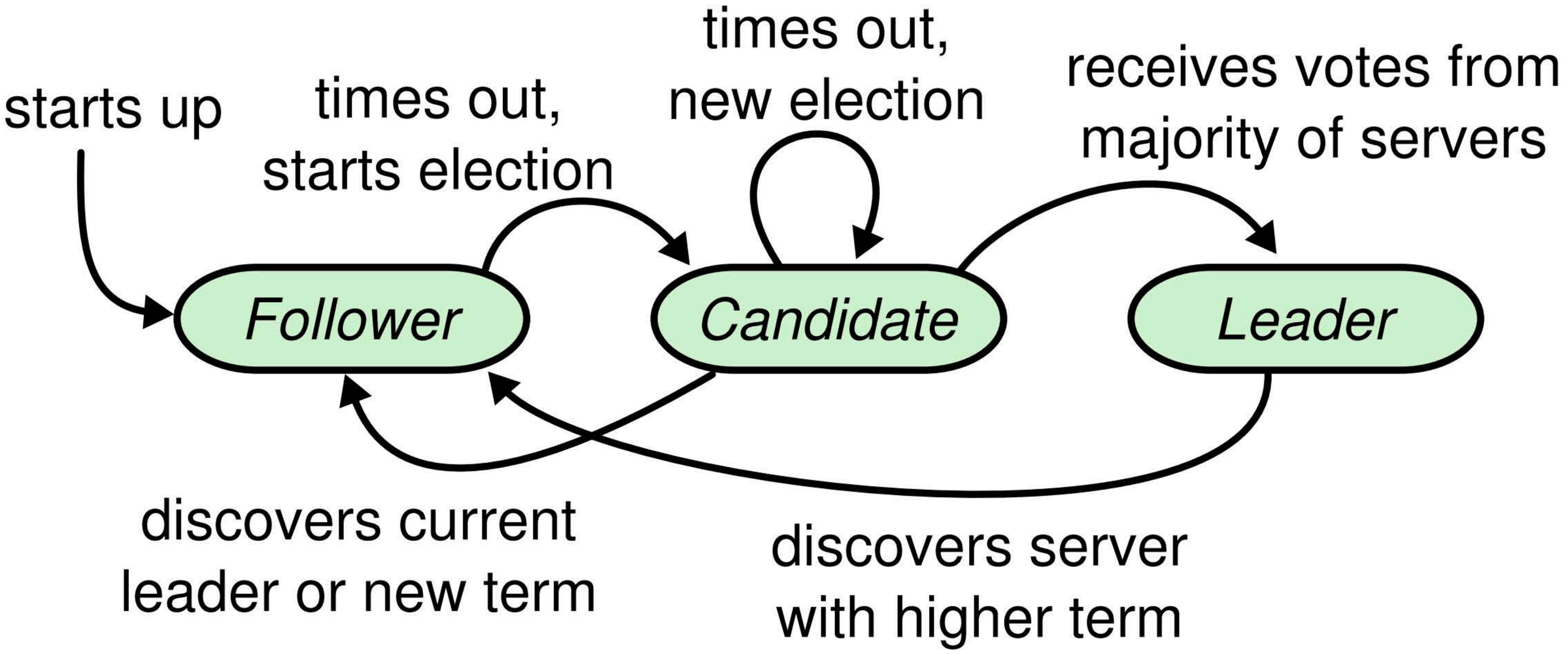
Neutralising Old Leaders

- *What if the old leader is restarted?*
- Upon receiving a heartbeat message from a leader on a greater term, it returns to the follower state

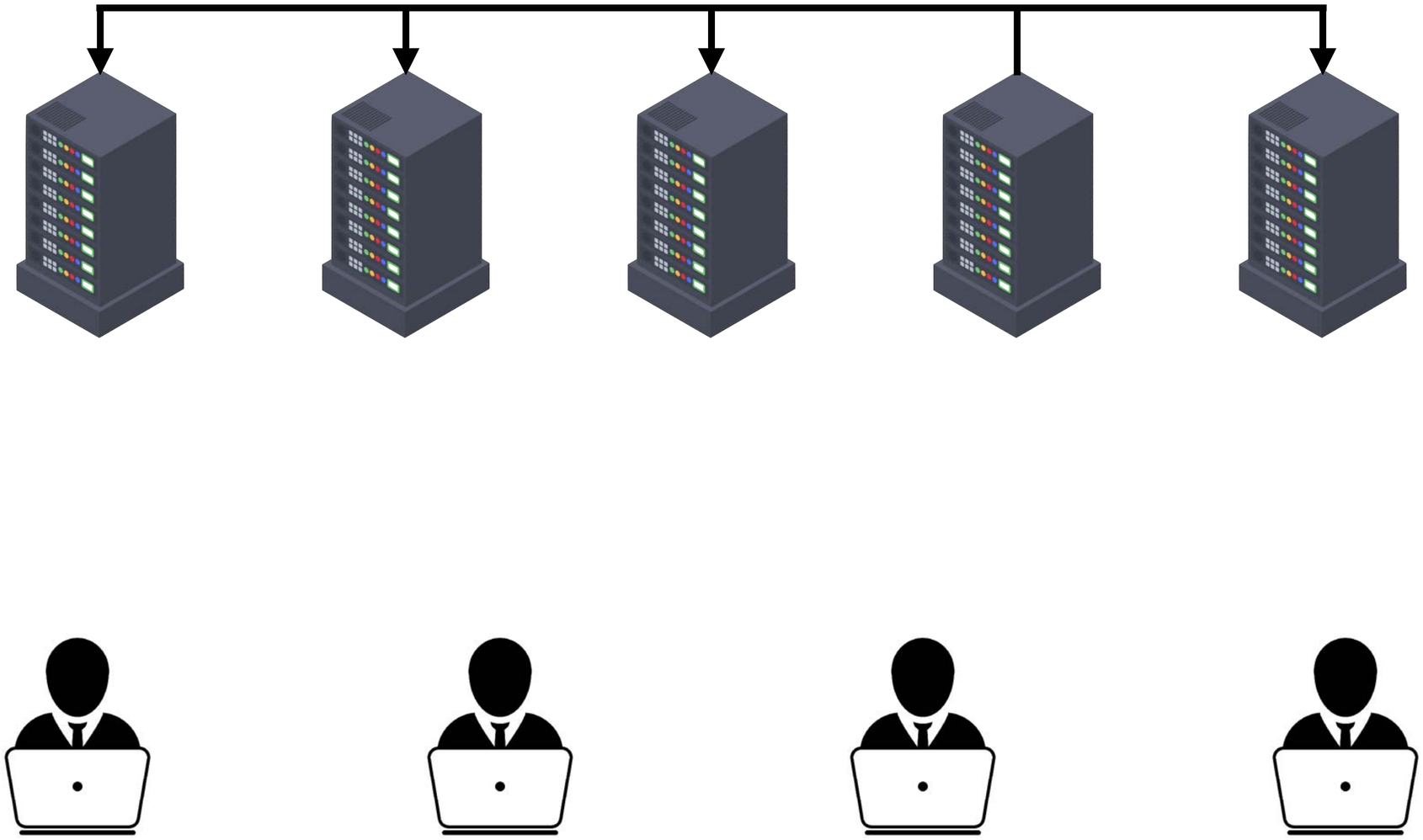


Leader Election

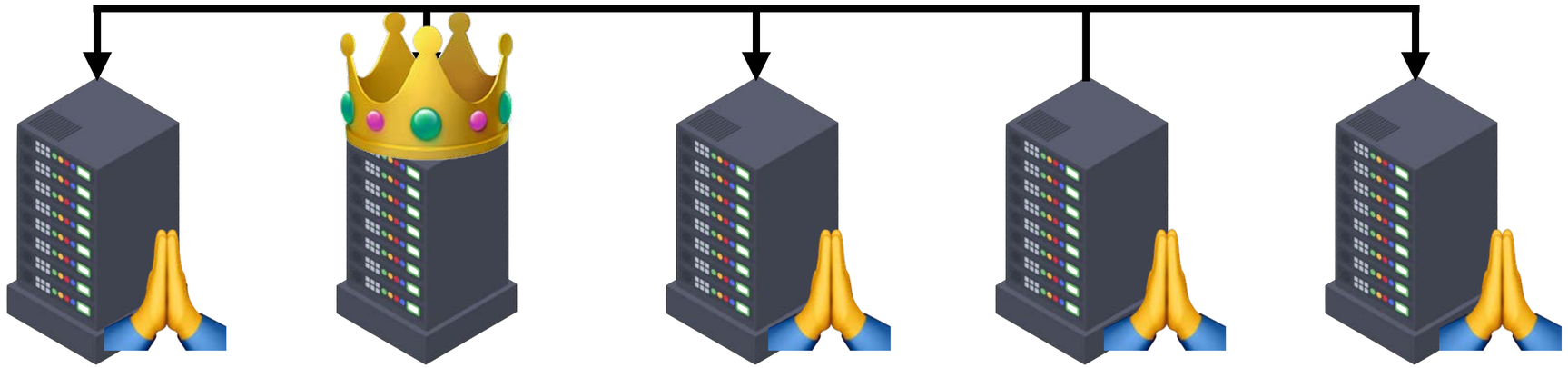
Server State Diagram



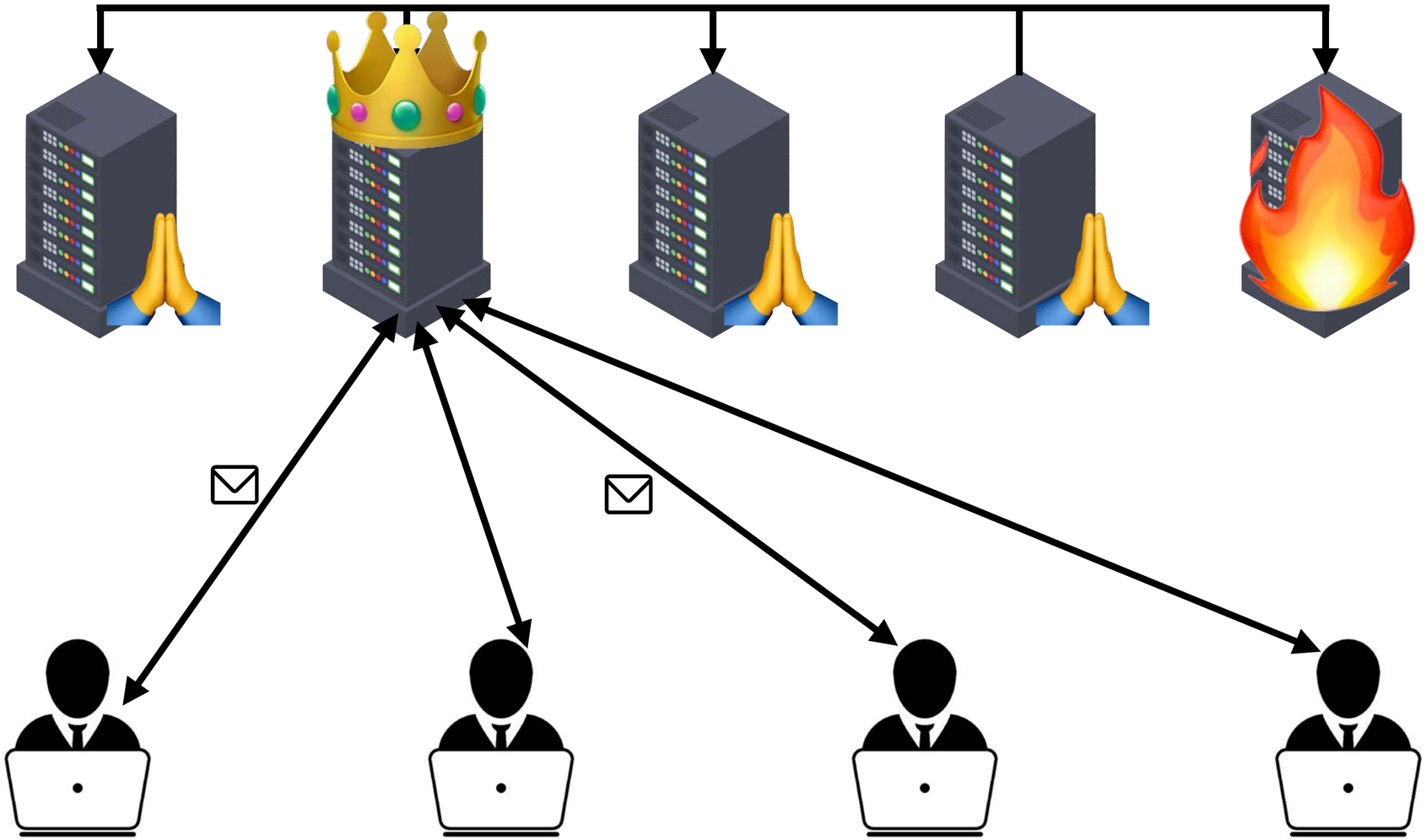
Active-Passive Architecture



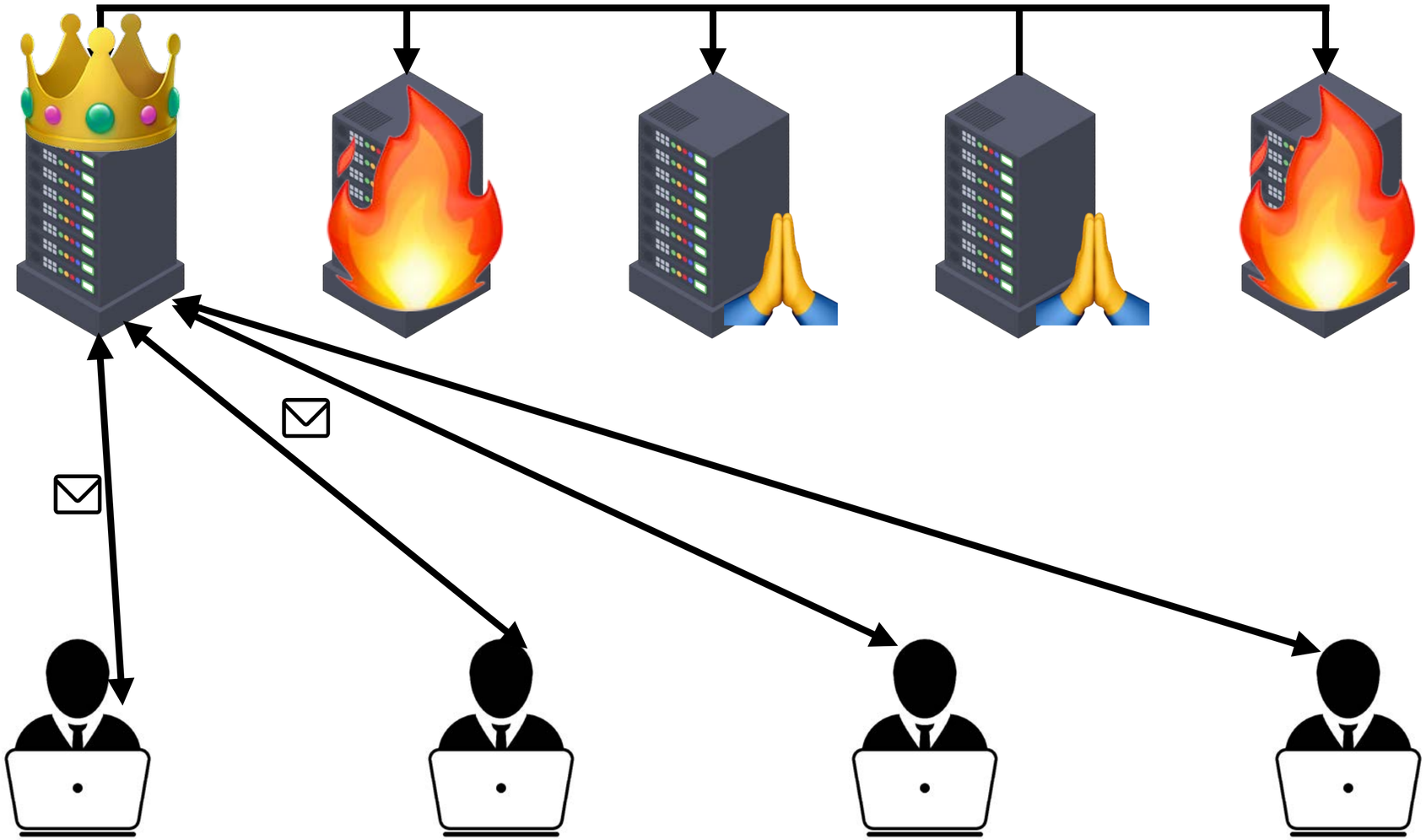
Active-Passive Architecture



Fault Tolerant



Fault Tolerant



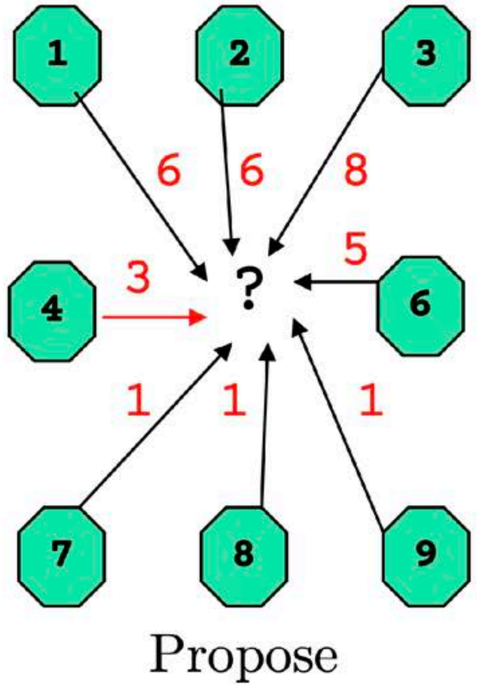
Agreement

Chapter 5

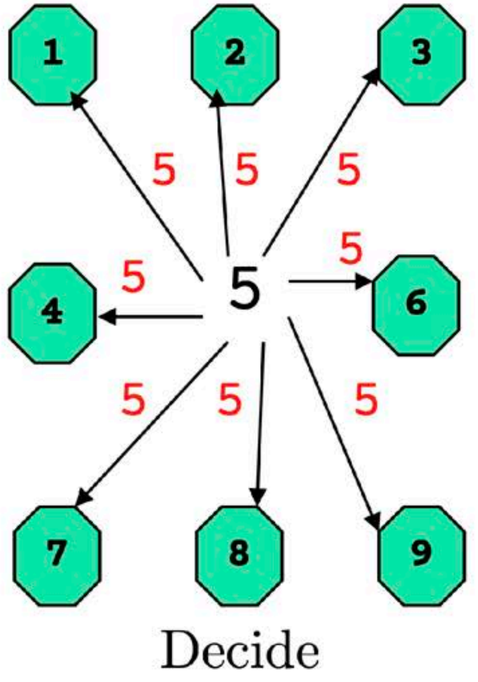
Consensus

How can servers in a distributed system agree on a value?

- Each server *proposes* a value & all servers *decide* one of the proposed values
- Values are often commands to be carried out by a set of **replicated servers**



Propose



Decide

Consensus

Replicated Banking Example

- Consider a bank that decides to run 2 replicas to manage bank accounts. The aim is improved reliability and availability. It must appear as one service.

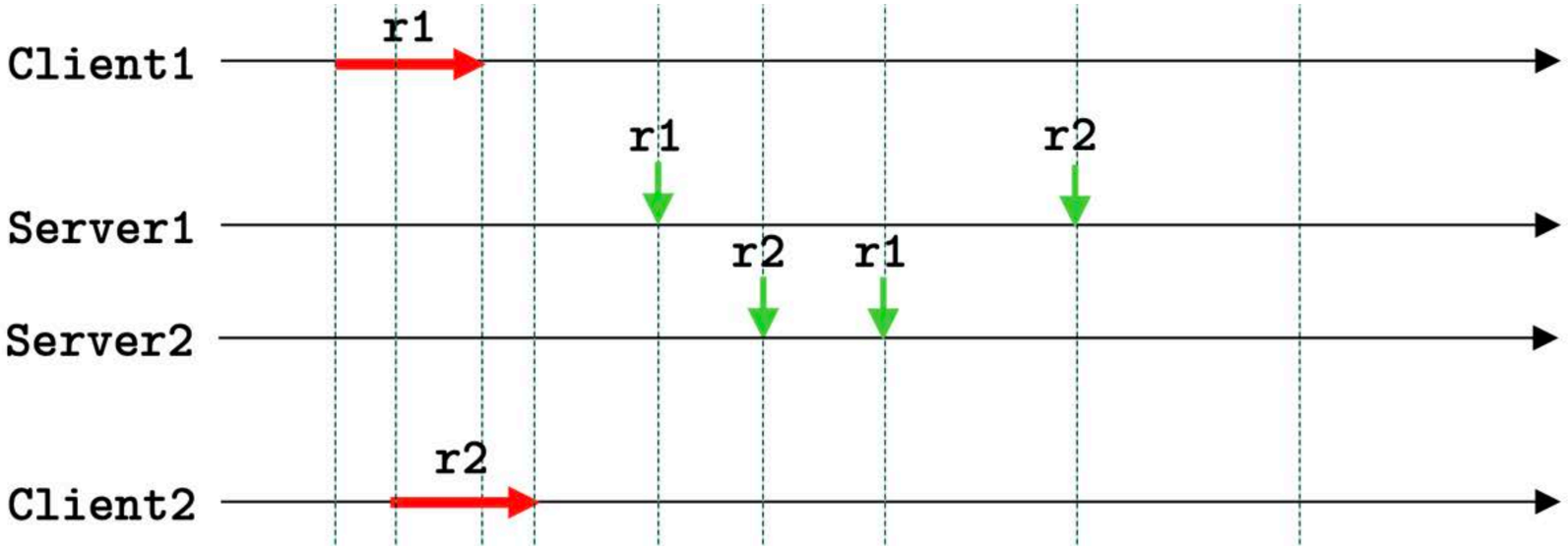
- Client1 (r1): `update set balance=balance+100 where acct='Alice'`
  Client2 (r2): `update set balance=balance*1.1 where acct='Alice'`

Consensus

Replicated Banking Example

- Consider a bank that decides to run 2 replicas to manage bank accounts. The aim is improved reliability and availability. It must appear as one service.

- Client1 (r1): `update set balance=balance+100 where acct='Alice'`
  Client2 (r2): `update set balance=balance*1.1 where acct='Alice'`

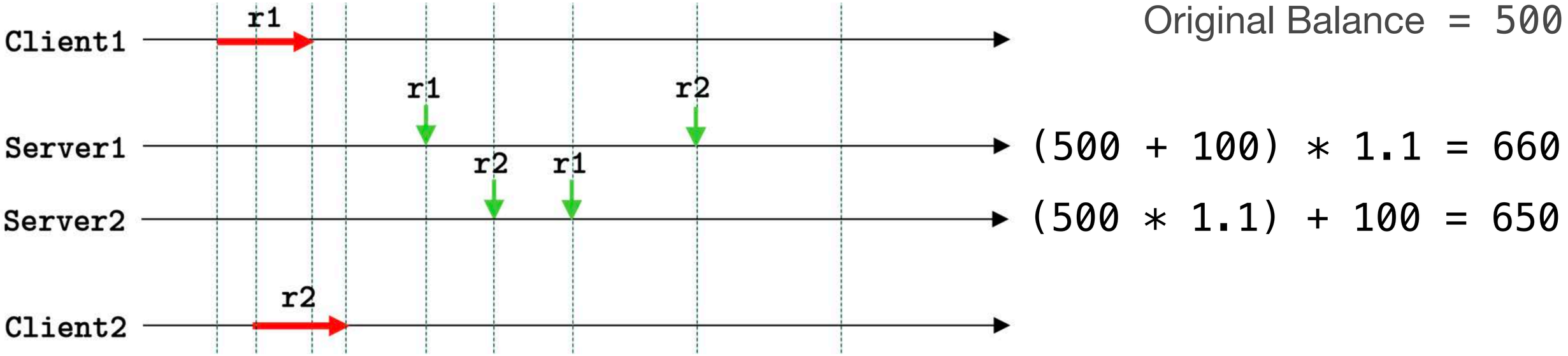


Consensus

Replicated Banking Example

- Consider a bank that decides to run 2 replicas to manage bank accounts. The aim is improved reliability and availability. It must appear as one service.

- Client1 (r1): `update set balance=balance+100 where acct='Alice'`
  Client2 (r2): `update set balance=balance*1.1 where acct='Alice'`

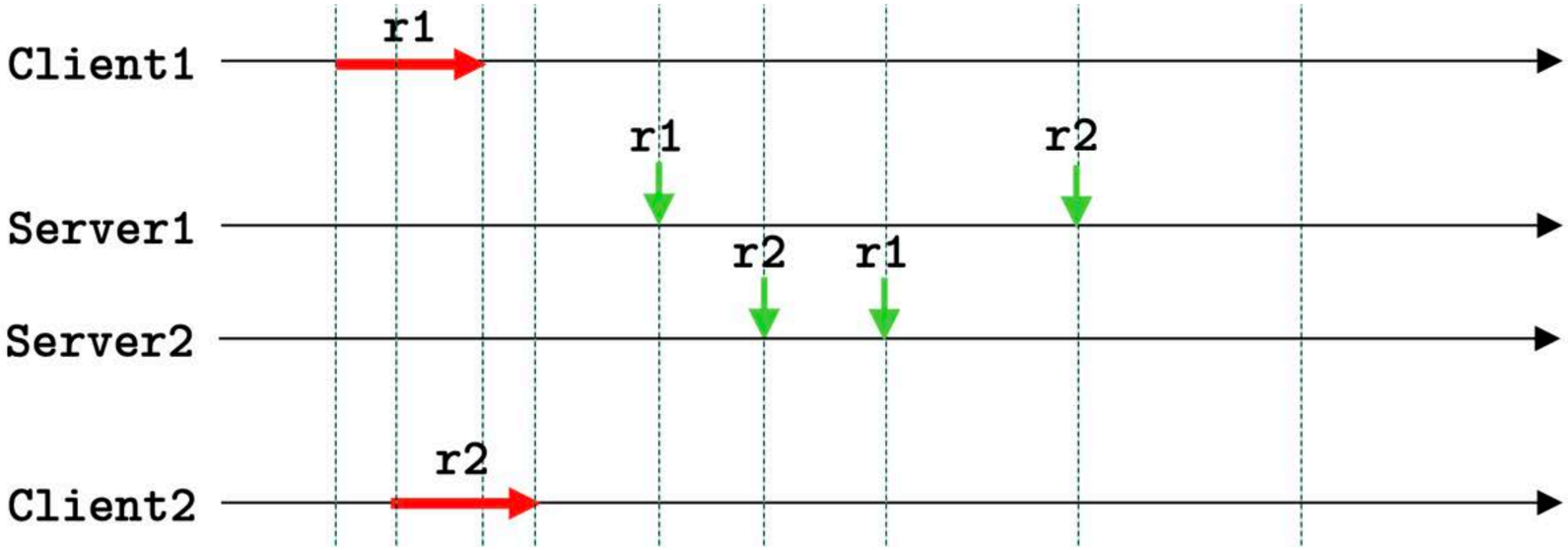


Consensus

Replicated Banking Example

- Consider a bank that decides to run 2 replicas to manage bank accounts. The aim is improved reliability and availability. It must appear as one service.

- Client1 (r1): `update set balance=balance+100 where acct='Alice'`
  Client2 (r2): `update set balance=balance*1.1 where acct='Alice'`



Original Balance = 500

(500 + 100) * 1.1 = 660

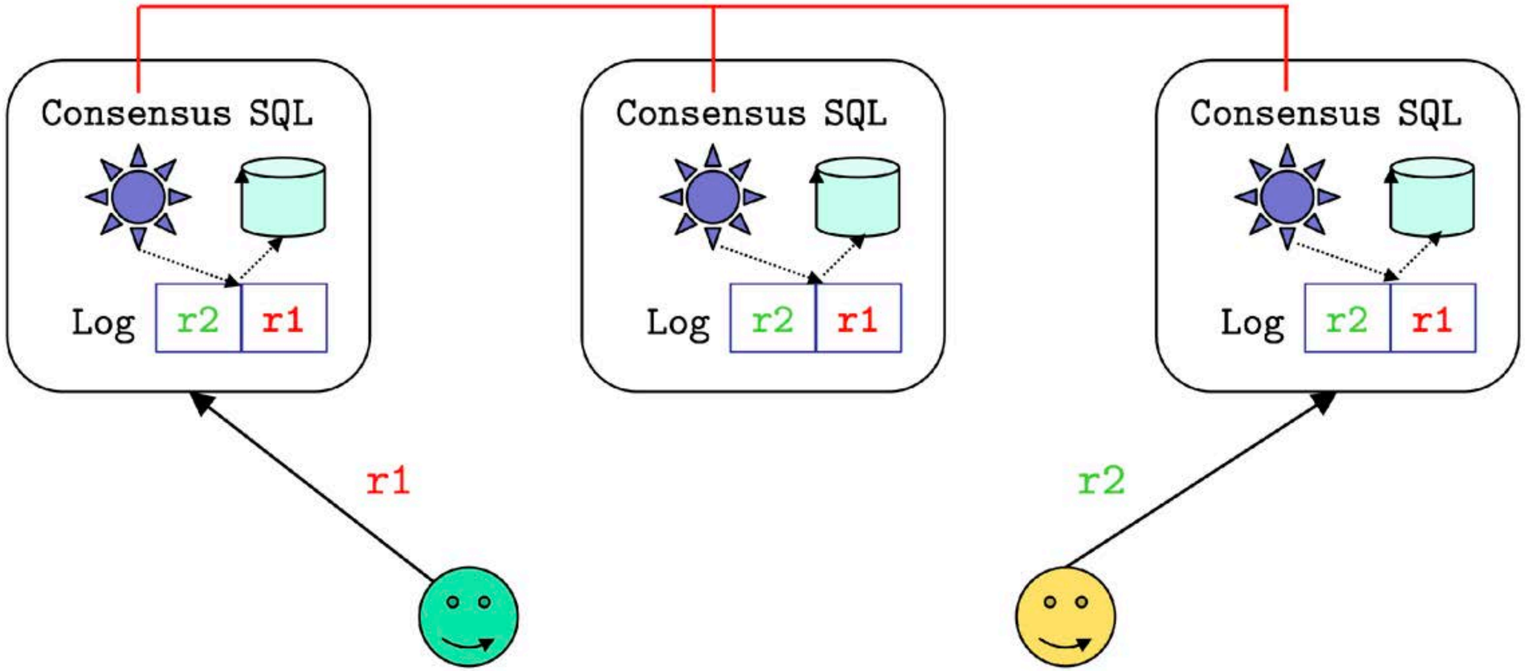
(500 * 1.1) + 100 = 650

Inconsistent!

Consensus

Replicated Banking Example

- The replicas need to **agree** on the ordering of the requests
- The sequence of commands executed will be the same for all **replicated state machines**, preserving consistency



Consensus

Properties

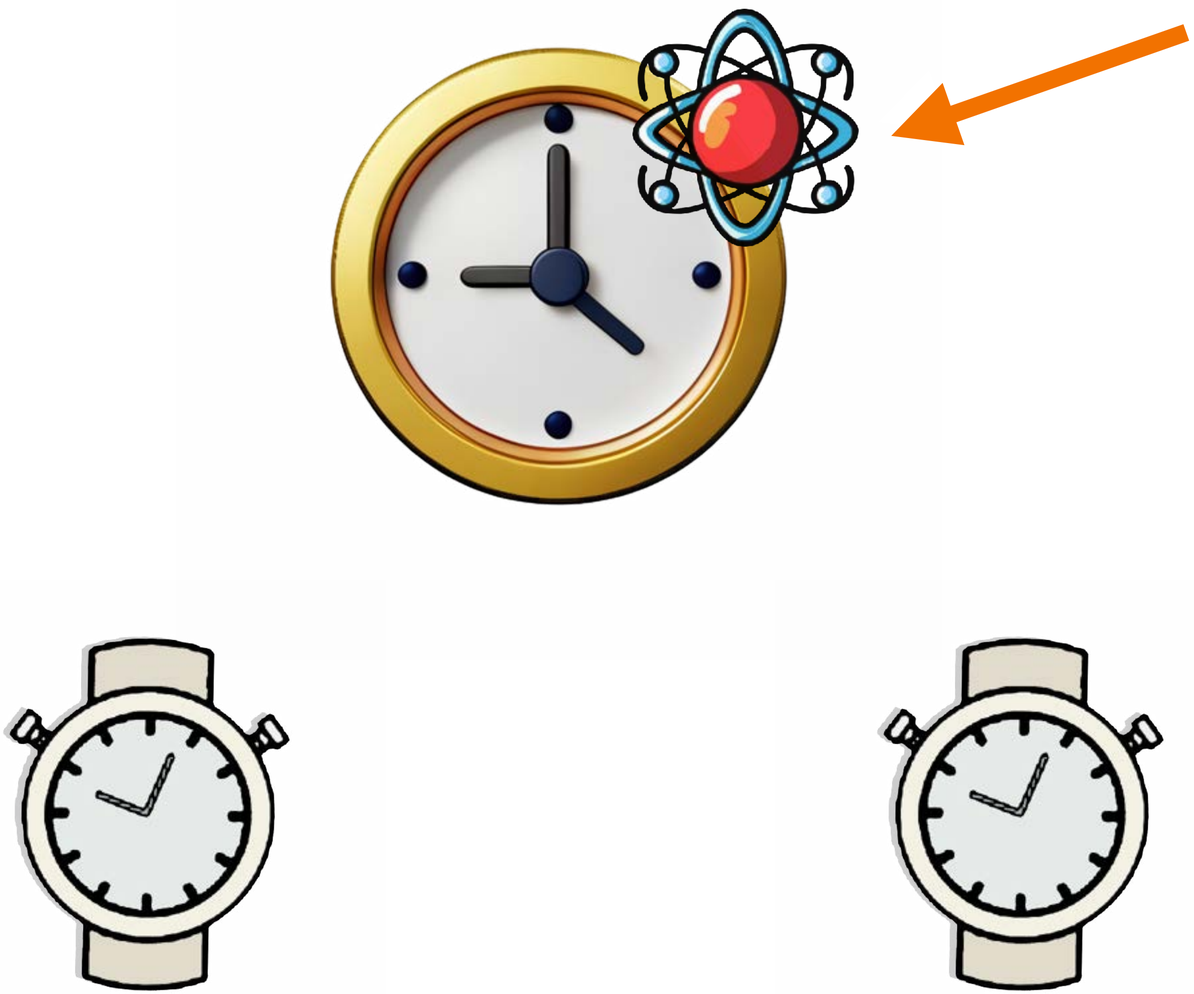
In concurrent/distributed system execution, **safety (S)** properties state what is permitted and **liveness (L)** properties describe what must *eventually* happen

Regular consensus has the following properties:

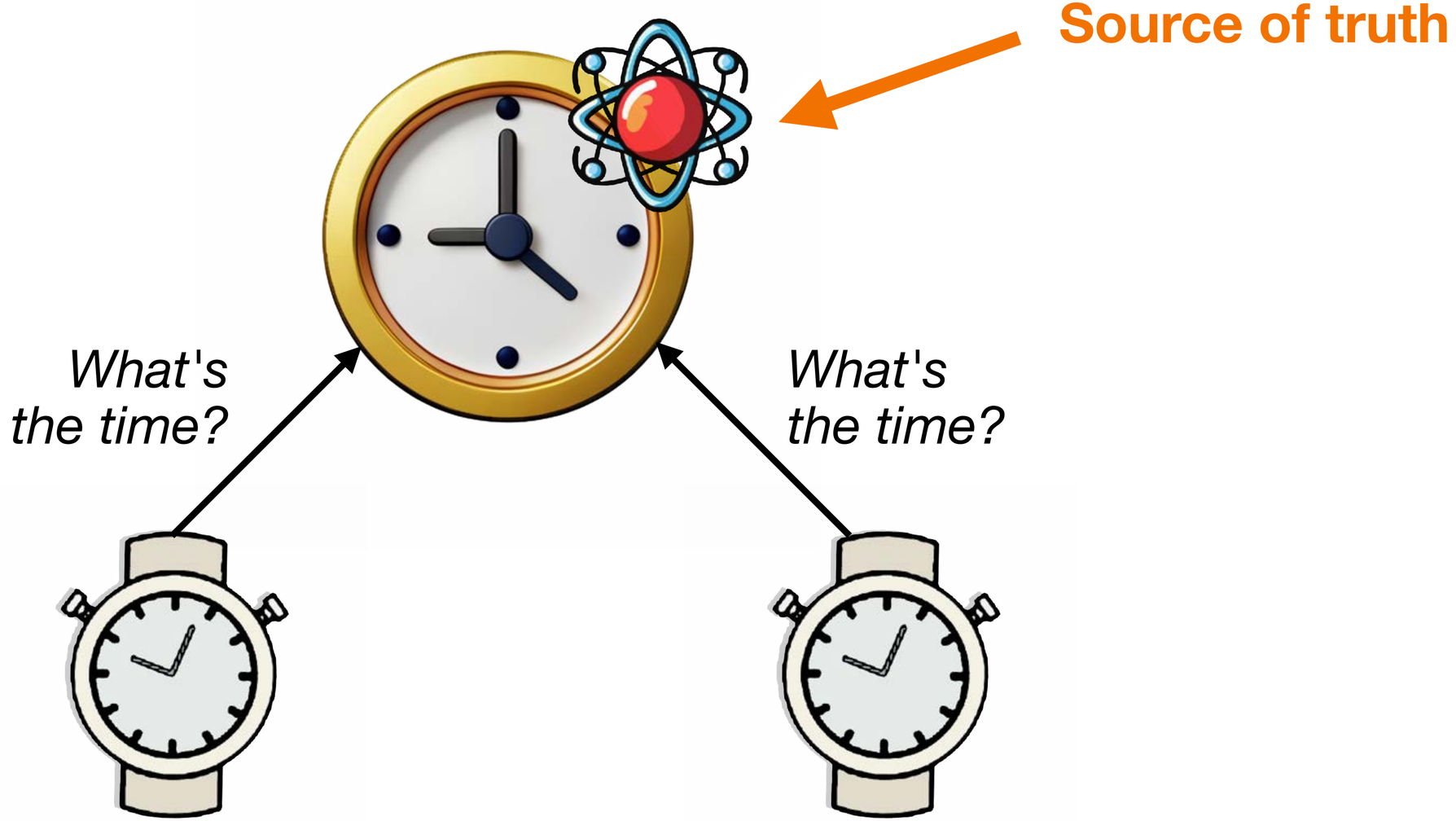
- **Validity (S)** - if a server decides a value, it was proposed by some server
- **Integrity (S)** - a server decides at most one value
- **Termination (L)** - each *correct* (alive) server eventually decides
- **Agreement (S)** - no two servers decide different values

Timing
Chapter 6

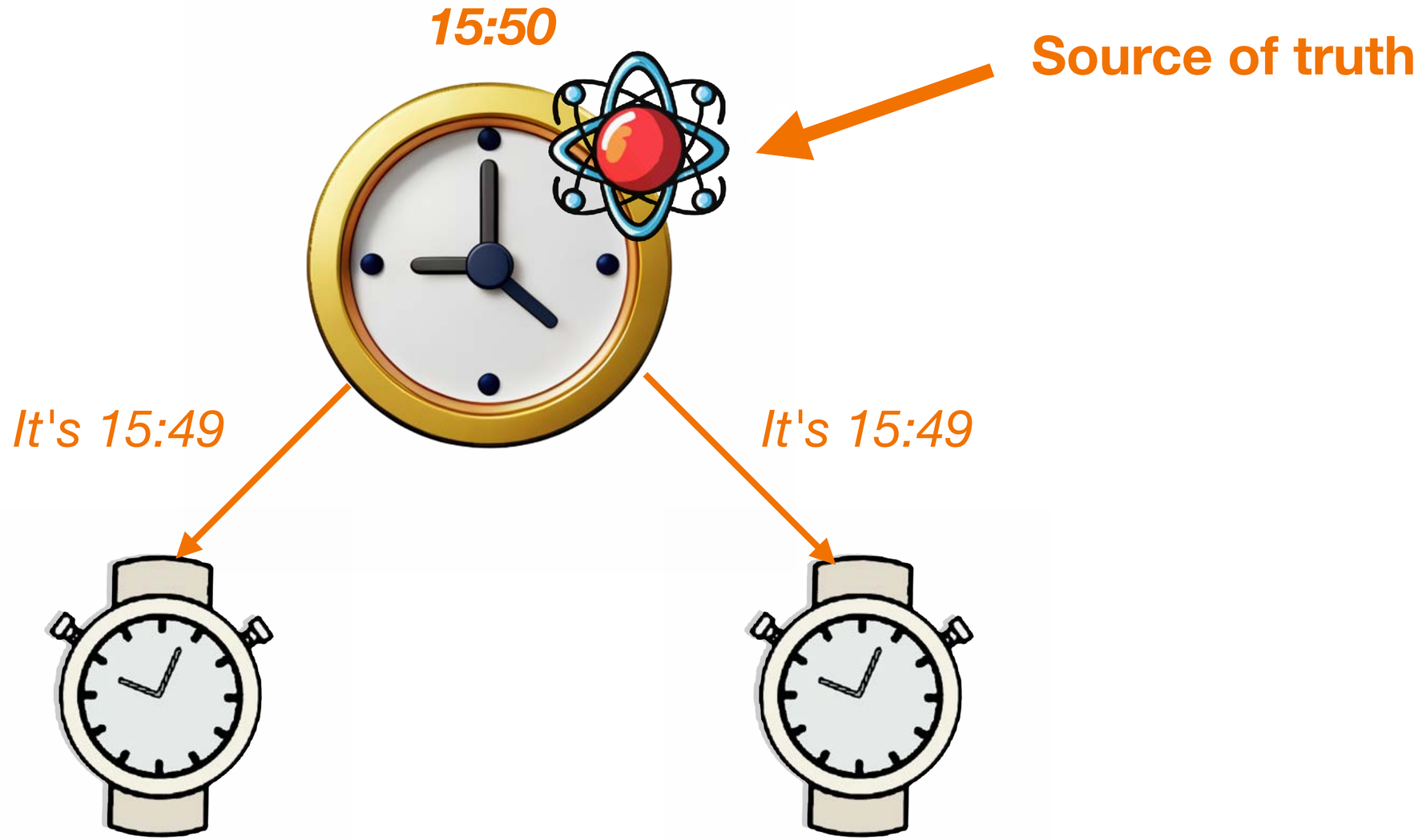
Time Synchronisation



Time Synchronisation



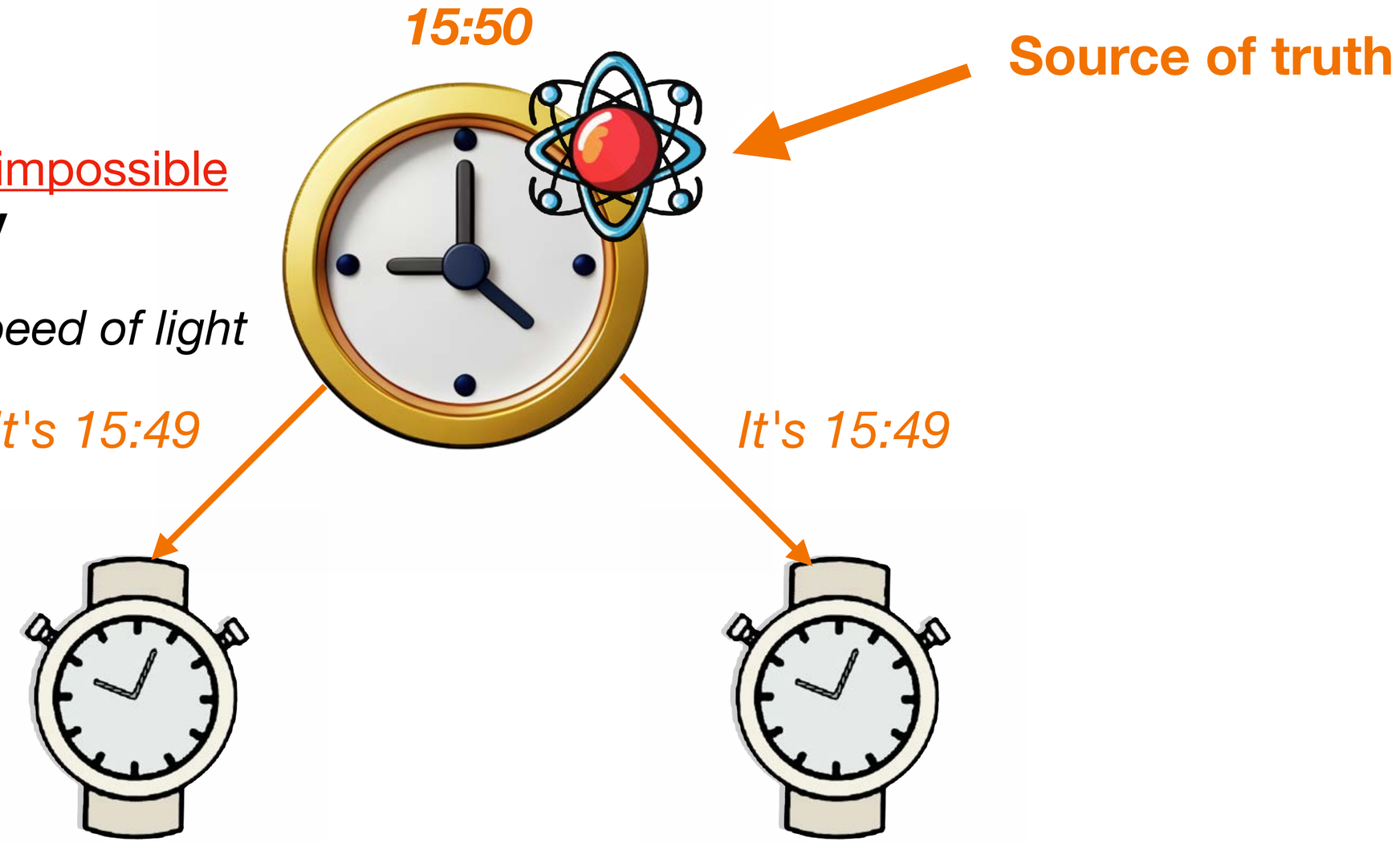
Time Synchronisation



Time Synchronisation

True synchronisation is impossible due to **network latency**

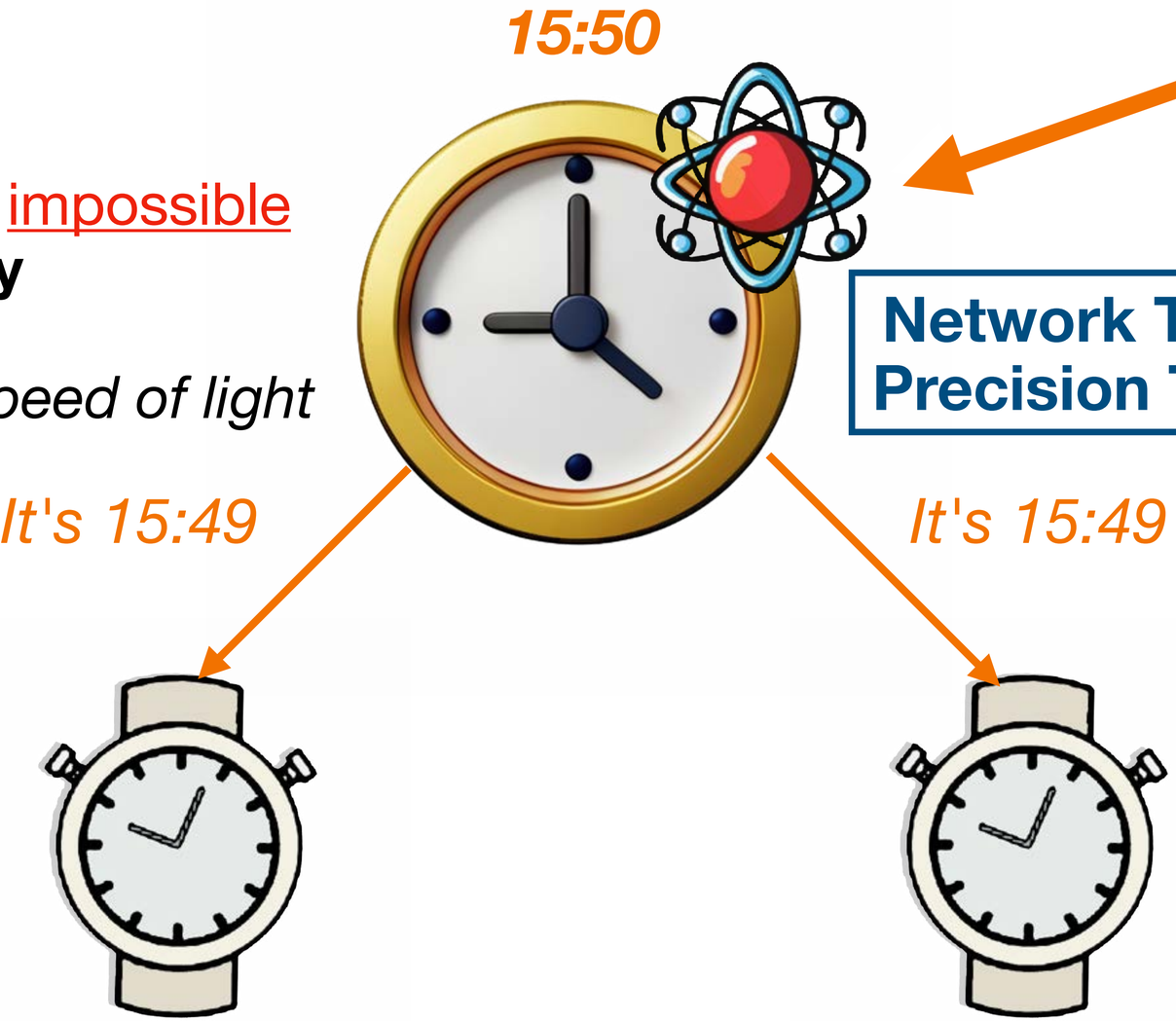
We are limited by the *speed of light*



Time Synchronisation

True synchronisation is impossible due to **network latency**

We are limited by the *speed of light*



Asynchronous systems
have *no global clock*

Asynchronous systems have *no global clock*

Servers coordinate using an event-driven architecture (e.g. network packet arrival)

0%

of consensus algorithms work
in pure async if servers can crash

FLP Impossibility Result

Consensus cannot be solved in a purely asynchronous system where servers can crash. *Proof: Fischer, Lynch and Paterson, 1985*

This result applies even when agreeing on a single bit value, or when message passing is 100% reliable, or when at most 1 server crashes.

- Intuitively, in a purely asynchronous system (without any concept of time), we can never tell if a server has crashed or just very slow to respond - no bounds
- Similarly, leader election and mutual exclusion can't be solved in pure async

FLP Impossibility Result

Consensus cannot be solved in a purely asynchronous system where servers can crash. *Proof: Fischer, Lynch and Paterson, 1985*

This result applies even when agreeing on a single bit value, or when message passing is 100% reliable, or when at most 1 server crashes.

- Intuitively, in a purely asynchronous system (without any concept of time), we can never tell if a server has crashed or just very slow to respond - no bounds
- Similarly, leader election and mutual exclusion can't be solved in pure async

Eventually synchronous system - messages take up to time *mT* most of the time, but sometimes they take longer. Consensus is possible in this case.

Quantifying Fault Tolerance

How many server crashes can any consensus algorithm tolerate?

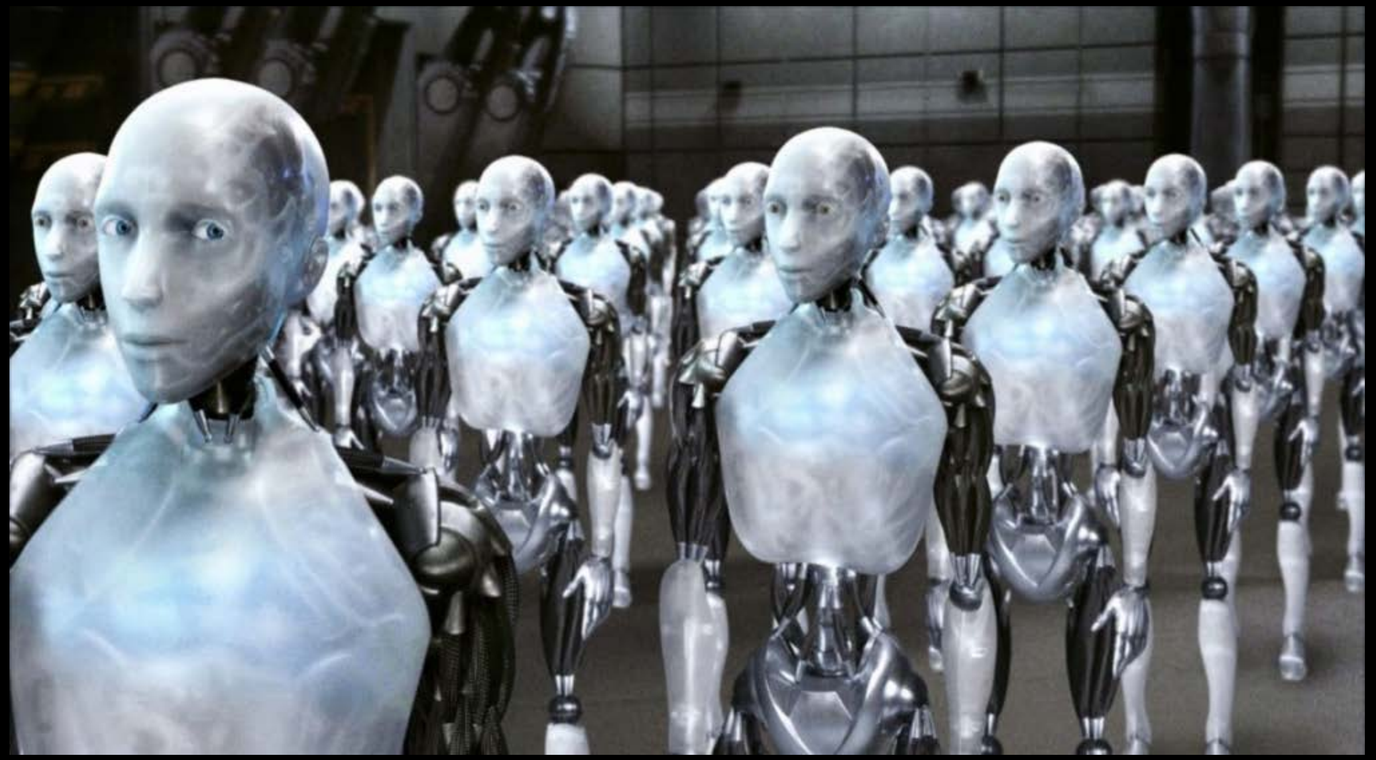
- *With **2** servers, consensus is not possible (peer may respond too slow/crash)*

- With **3** servers, consensus algorithms can tolerate up to 1 crash
- With **9** servers, consensus algorithms can tolerate up to 4 crashes
- With ***2f+1*** servers, consensus algorithms can tolerate up to *f* crashes

Known as a *correct* '*majority*' or '*quorum*' of servers

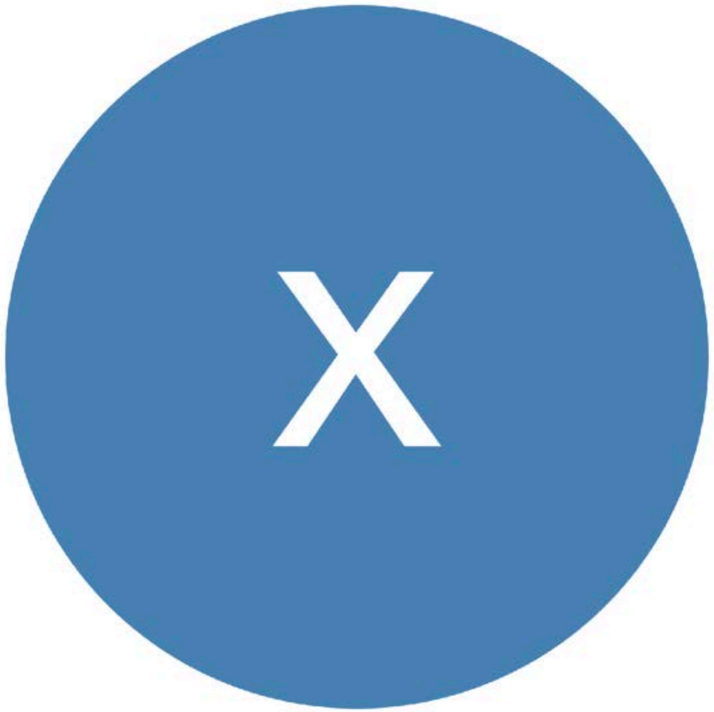
Replication

Chapter 7



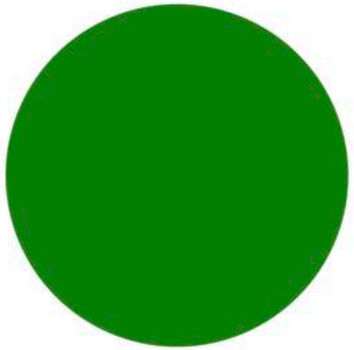
Database Server

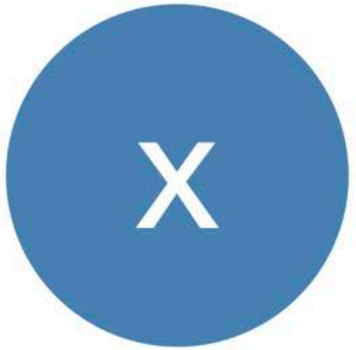
Consider a *single-node* system, where our node is a database, storing just a single value



Database Server

We also have a client that can send a value to the server





Database Server

We also have a client that can send a value to the server

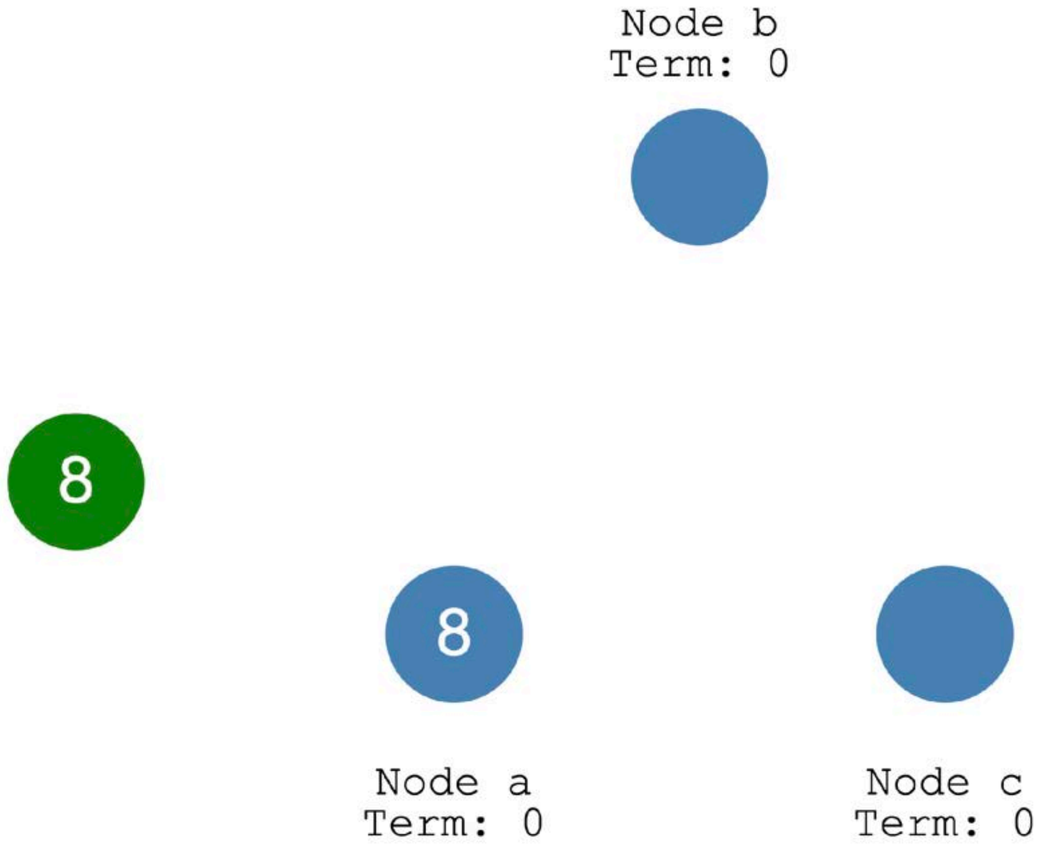
Database Server

We also have a client that can send a value to the server

Agreeing on that value is easy with one node

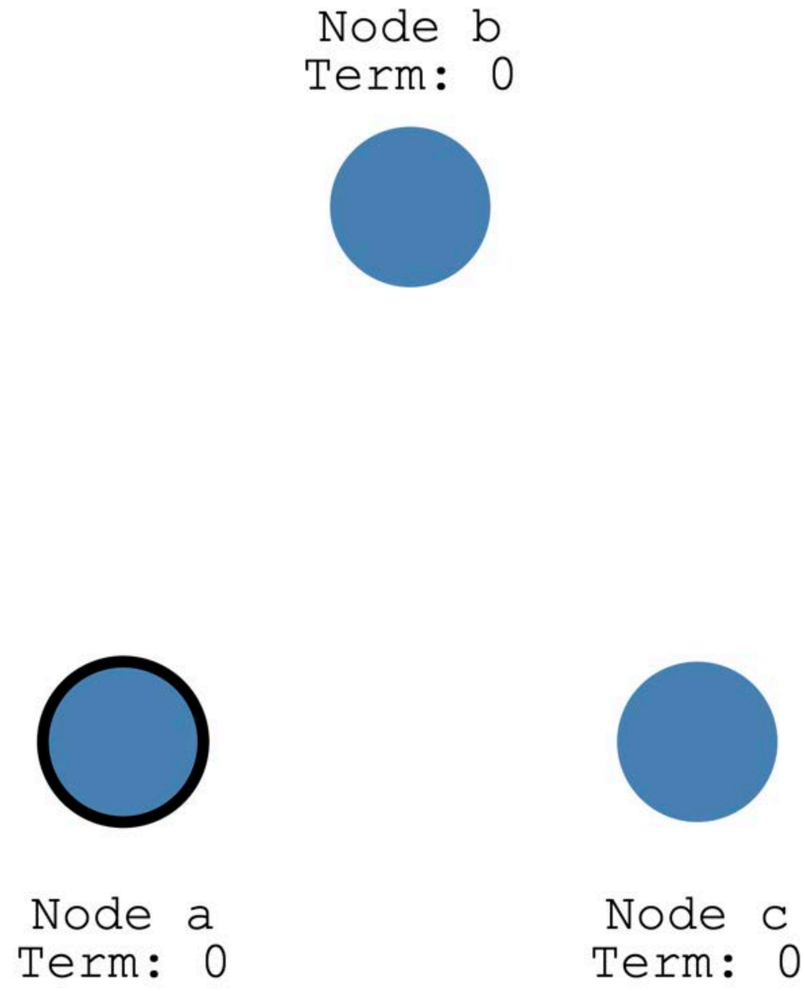
Distributed Database Server

Now consider consensus with multiple nodes (*distributed consensus*)



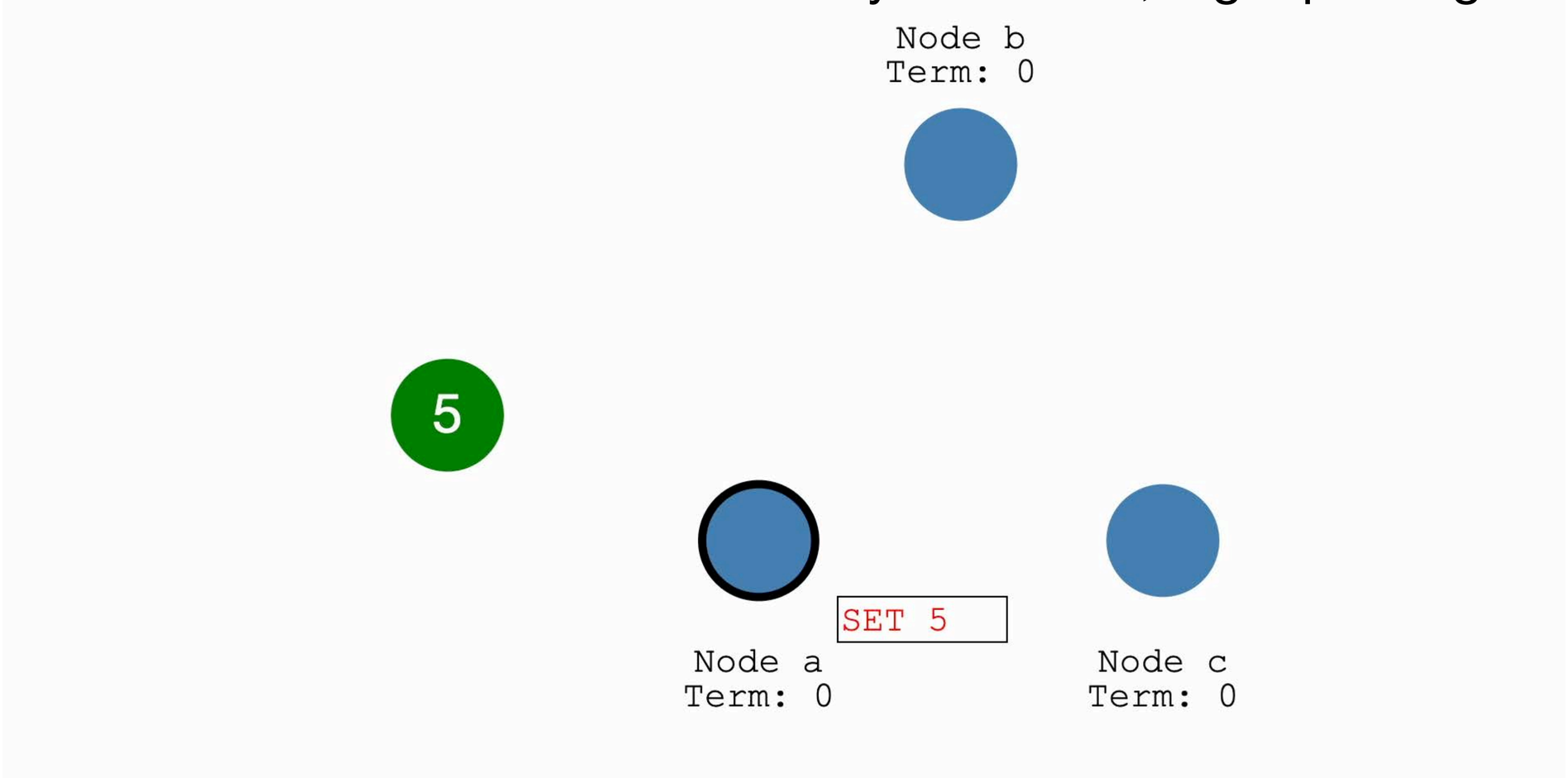
Distributed Database Server

First, a **leader is elected** using the protocol we looked at earlier



Distributed Database Server

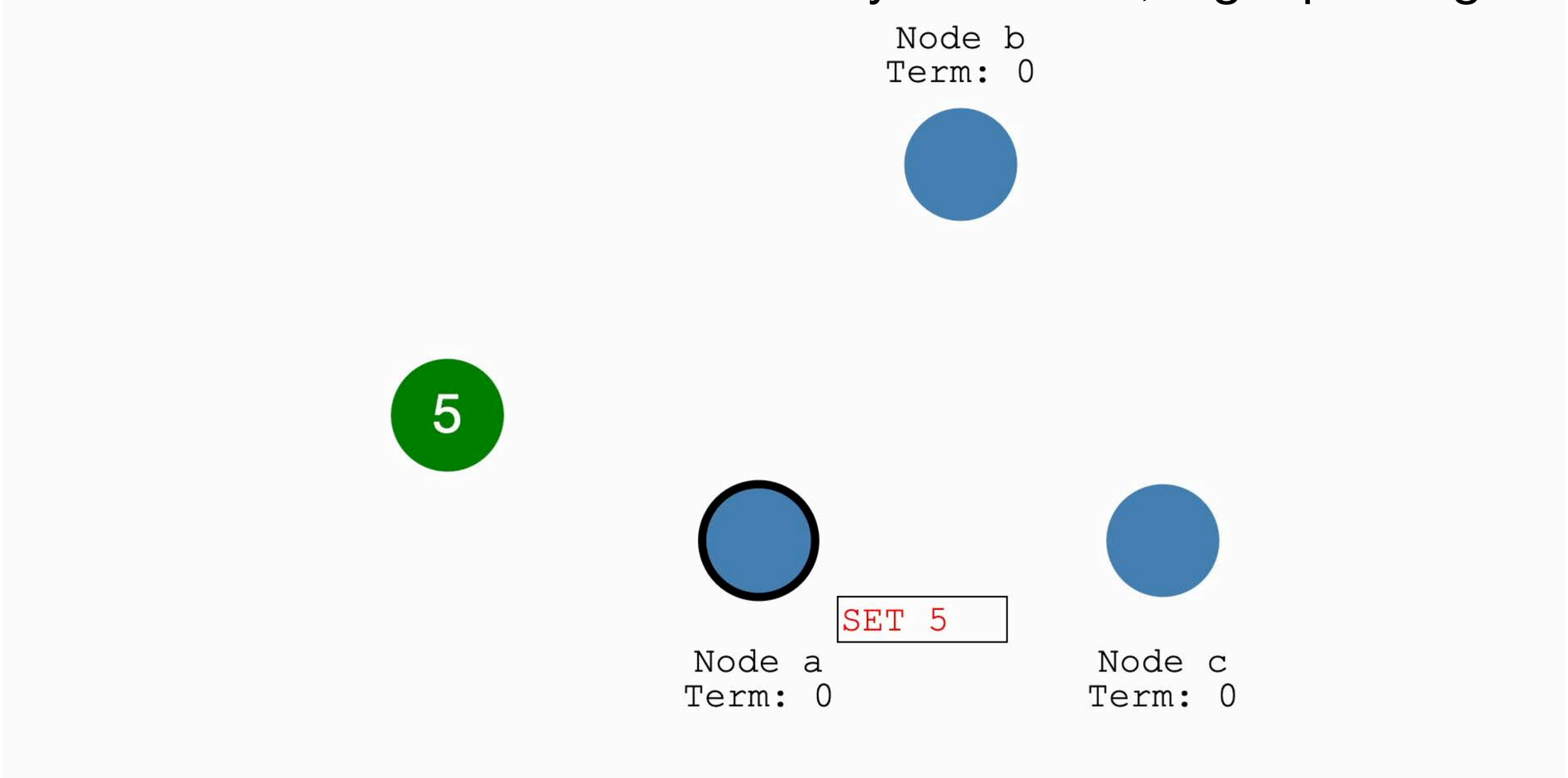
Clients send **commands** to be executed by the server, e.g. updating the value



Each change is added as an *entry* in the node's **log**

Distributed Database Server

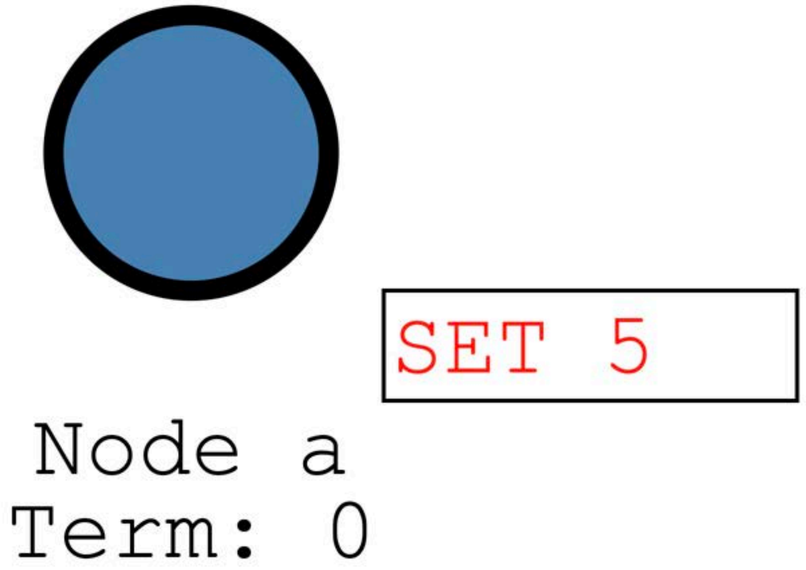
Clients send **commands** to be executed by the server, e.g. updating the value



Each change is added as an *entry* in the node's **log**

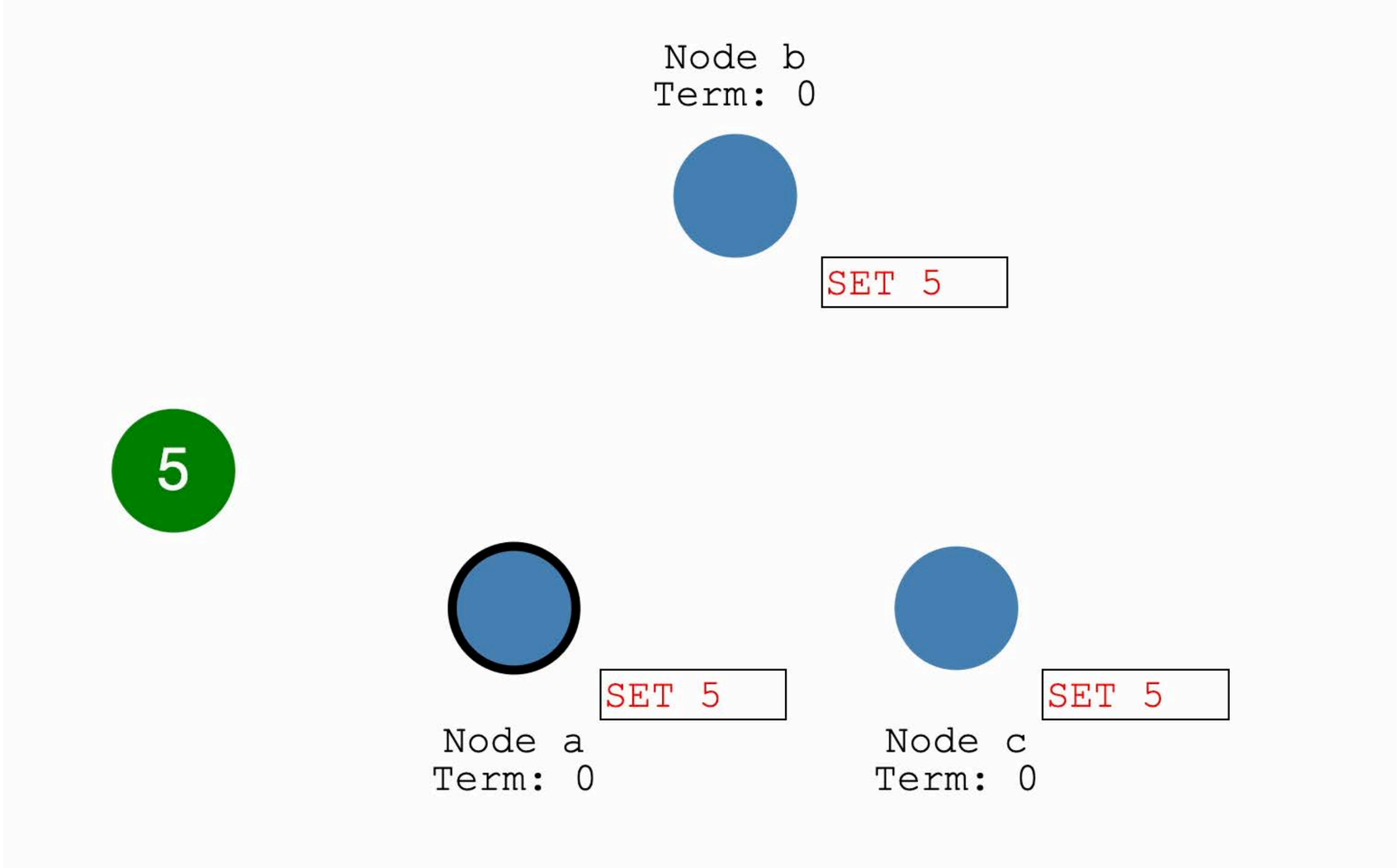
Distributed Database Server

This log entry is currently **uncommitted**,
so it won't update the node's value



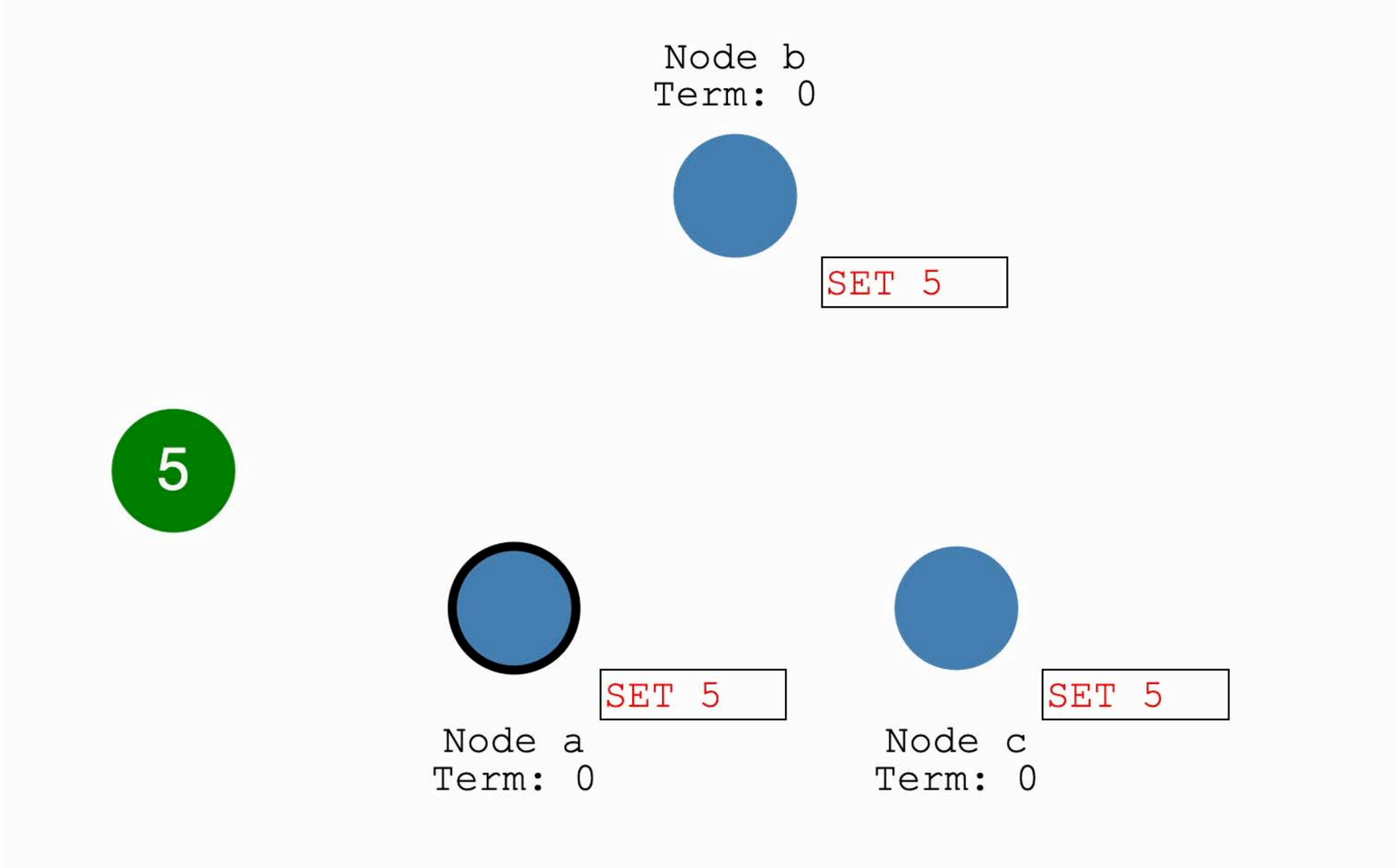
Distributed Database Server

To **commit** the entry, the node first *replicates* it to the follower nodes...



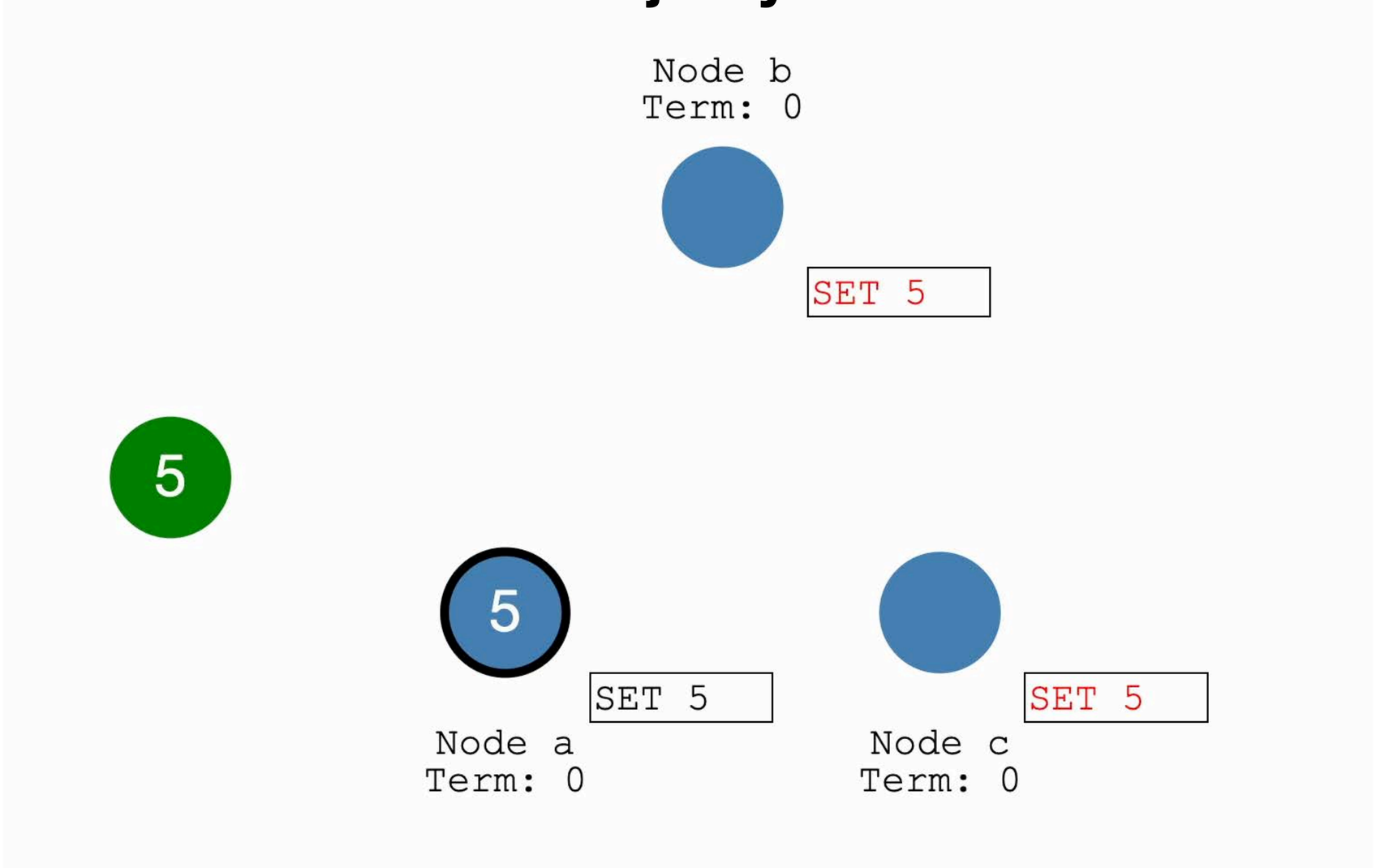
Distributed Database Server

To **commit** the entry, the node first *replicates* it to the follower nodes...



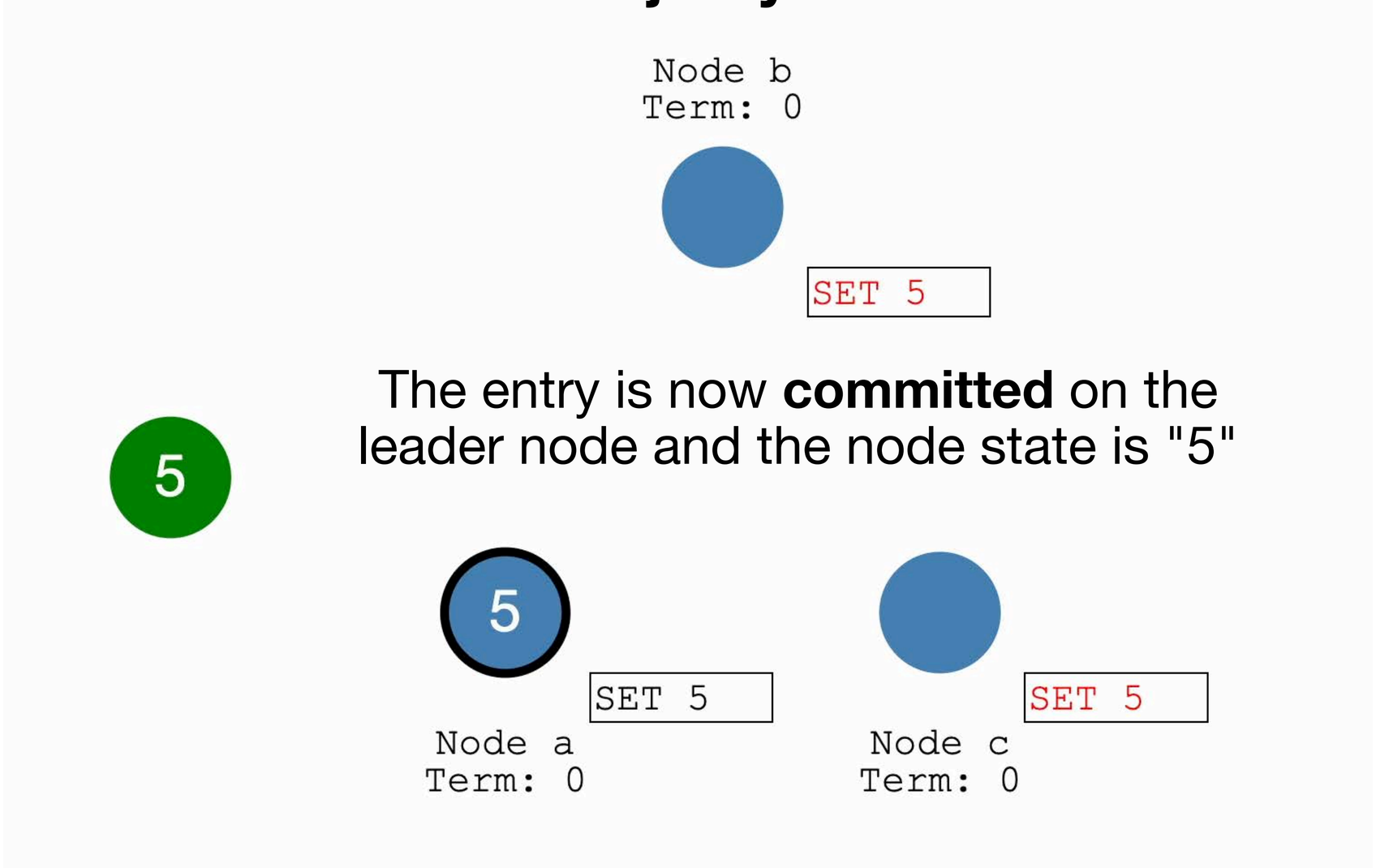
Distributed Database Server

... then the leader waits until a **majority** of nodes have written the entry



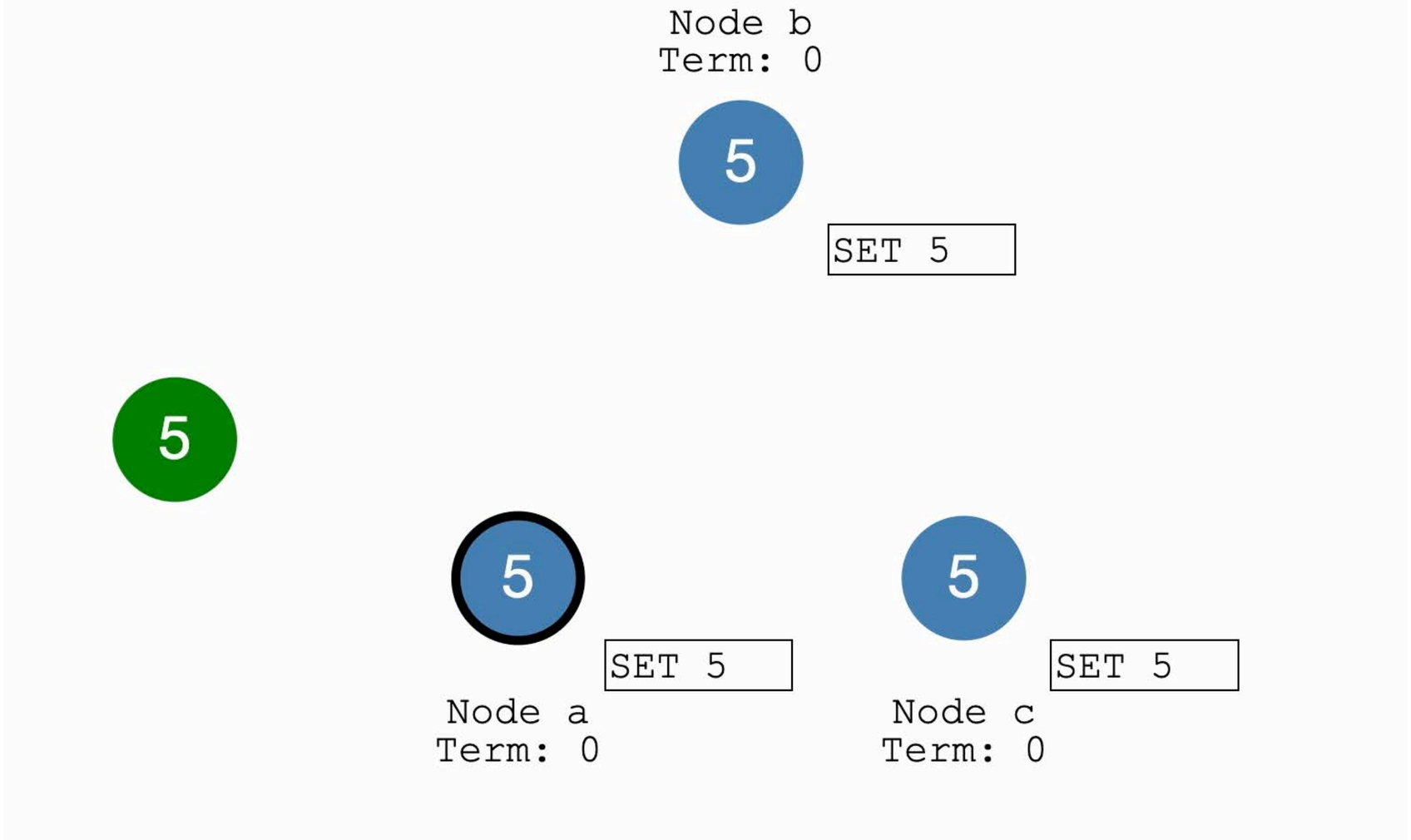
Distributed Database Server

... then the leader waits until a **majority** of nodes have written the entry



Distributed Database Server

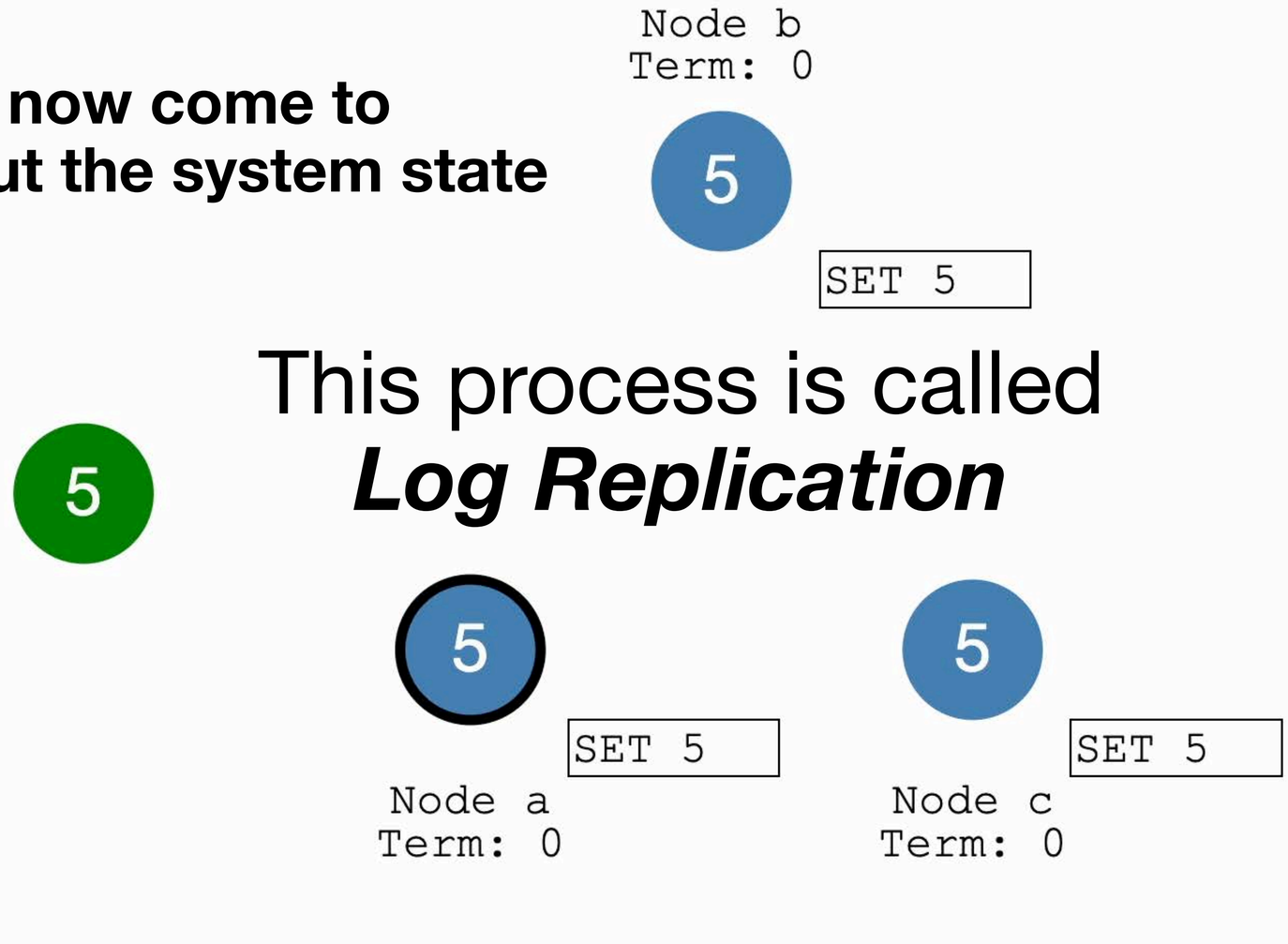
The leader then notifies the followers that the entry is committed



Distributed Database Server

The leader then notifies the followers that the entry is committed

The cluster has now come to consensus about the system state



Raft

by Diego Ongaro

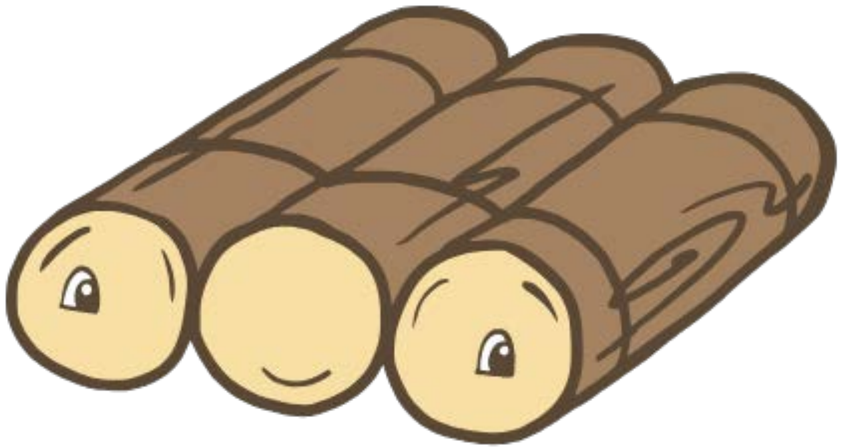
***R**eliable, **R**eplicated, **R**edundant **A**nd **F**ault-**T**olerant*

Leader Election & *Log Replication,* as discussed,
form the basis of the Raft consensus algorithm

More understandable alternative to *Paxos* family

`raft.github.io` to learn more
(the paper is an excellent read!)

Widely used: *CockroachDB, Etcd, Kubernetes,*
Docker Swarm, MongoDB, Neo4j, RabbitMQ



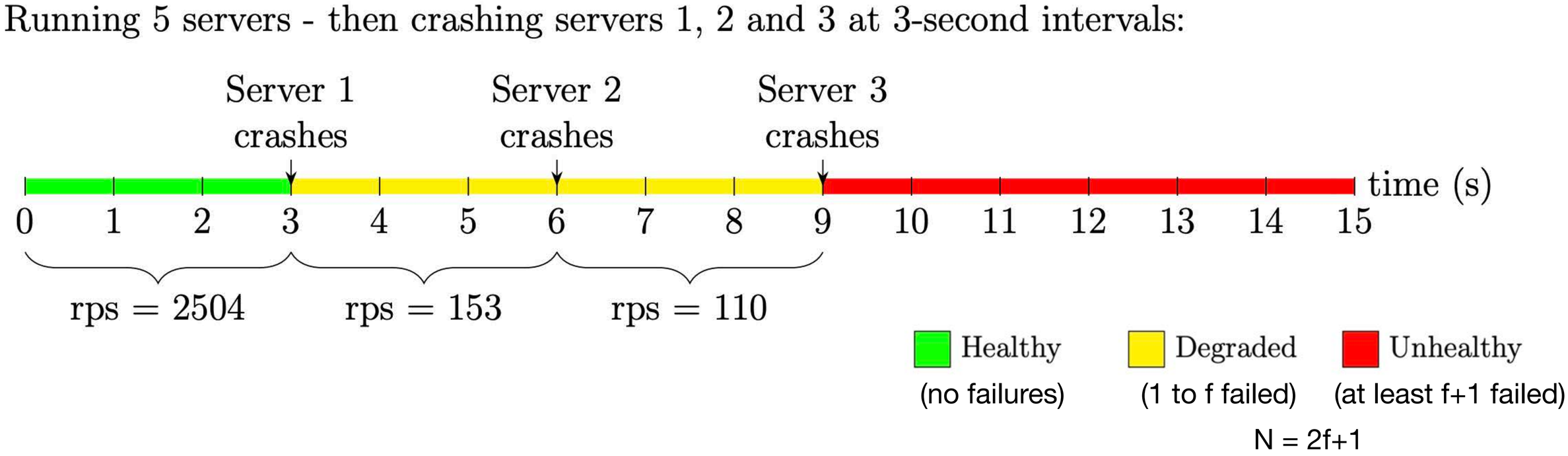
Failure

Chapter 8

Permanent Failures

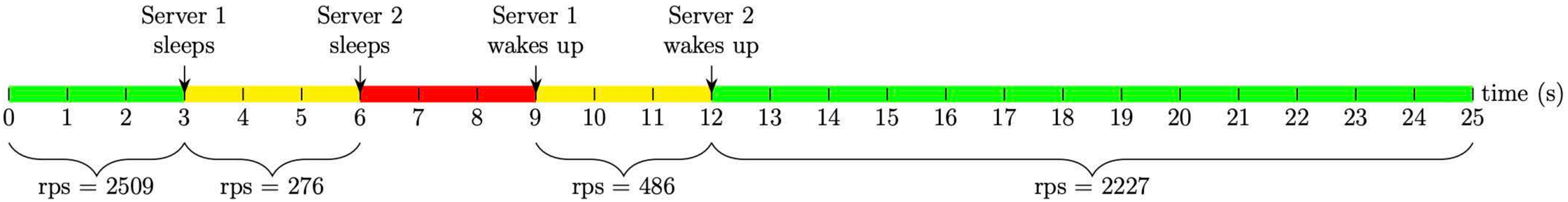
Recall a majority is needed to *commit* an entry

For **5**-server consensus, at most **2** can fail before correct majority is lost.
If 3+ servers crash, servers still replicate logs but cannot *commit* any entries

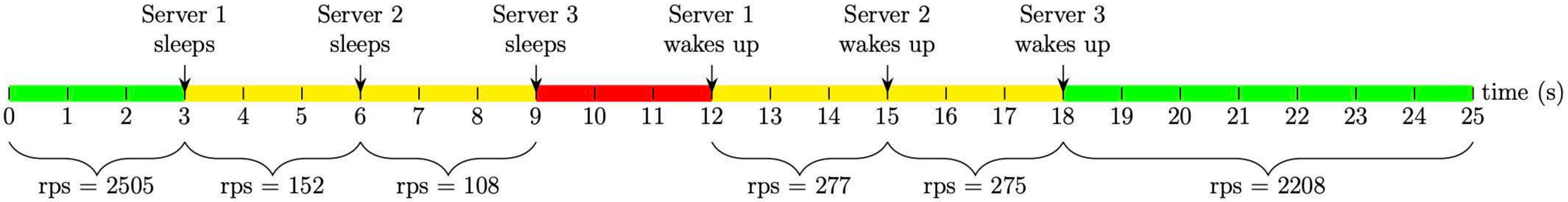


Temporary Failures

Running 3 servers - then sleeping servers 1 and 2 at 3-second intervals for 6 seconds:



Running 5 servers - then sleeping servers 1, 2 and 3 at 3-second intervals for 9 seconds:



Healthy (no failures)

Degraded (1 to f failed)

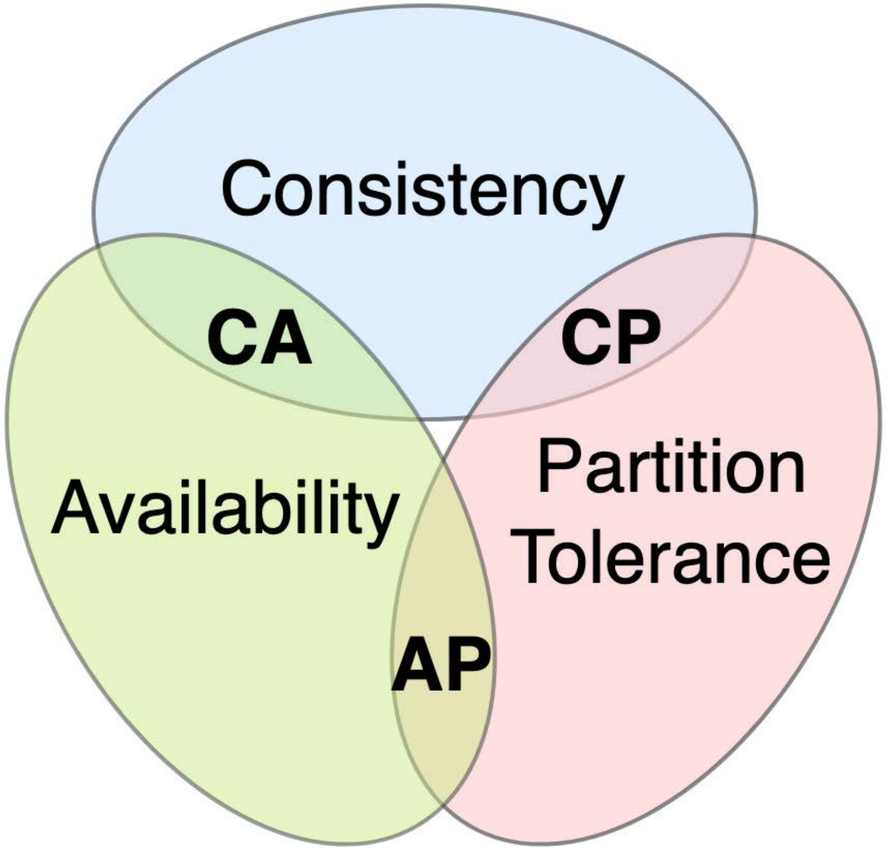
Unhealthy (at least f+1 failed)

N = 2f+1

CAP Theorem 🧢

Eric Brewer

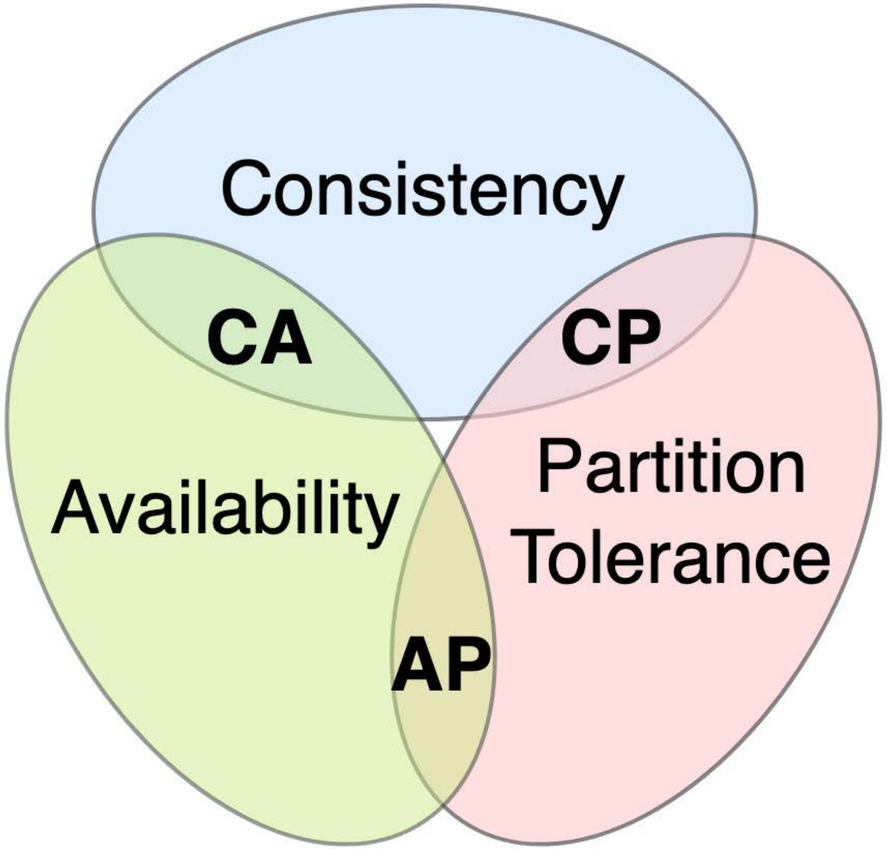
- Any *distributed data store* can only provide two of these three guarantees:
  - **Consistency**: Every read receives the most recent write or an error
  - **Availability**: Every request receives a (non-error) response, without the guarantee that it contains the most recent write
  - **Partition Tolerance**: The system continues to operate despite being an arbitrary number of messages being dropped (or delayed) by the network



CAP Theorem 🧢

Eric Brewer

- Any *distributed data store* can only provide two of these three guarantees:
  - **Consistency**: Every read receives the most recent write or an error
  - **Availability**: Every request receives a (non-error) response, without the guarantee that it contains the most recent write
  - **Partition Tolerance**: The system continues to operate despite being an arbitrary number of messages being dropped (or delayed) by the network

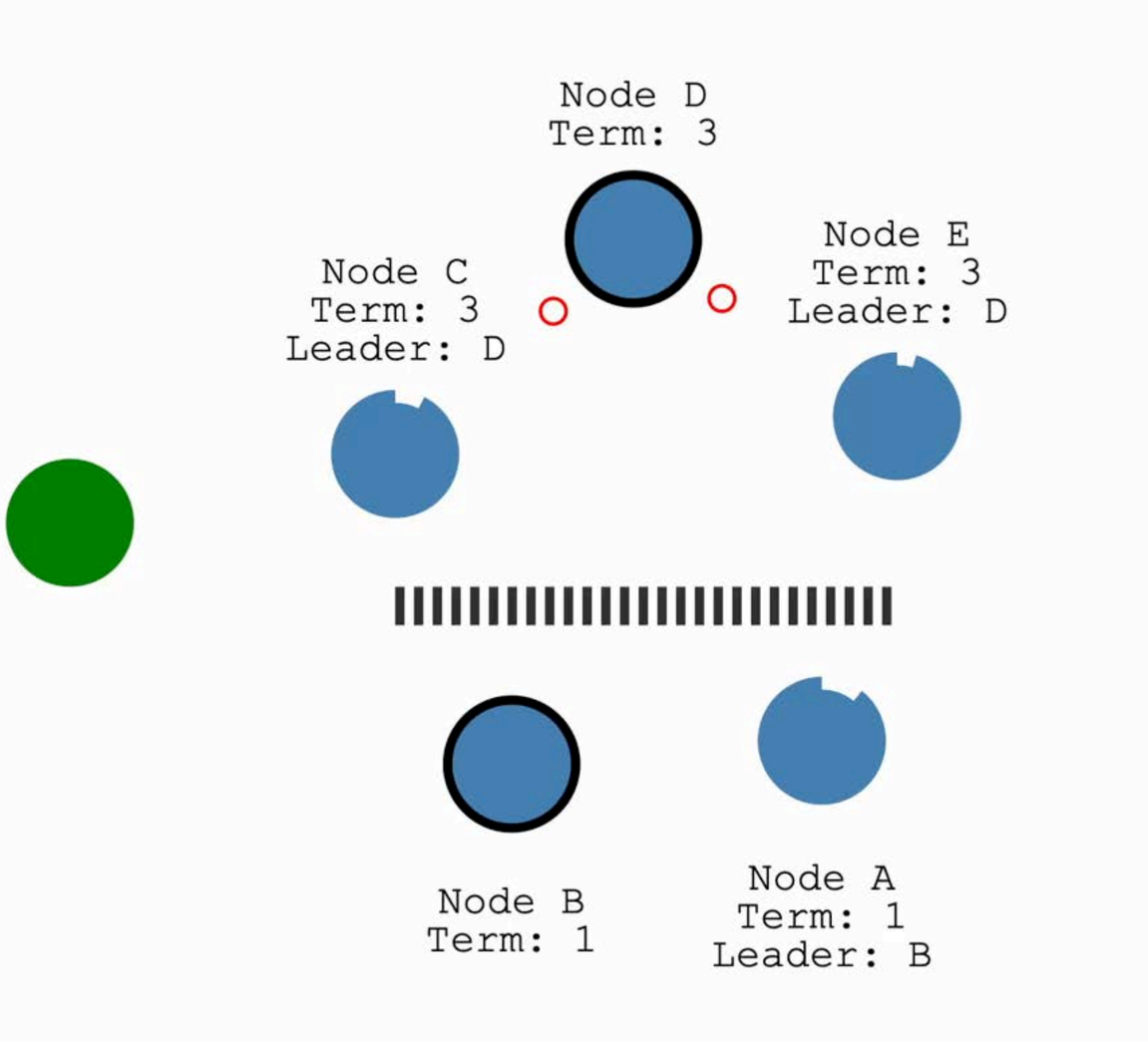


***Raft* is a CP system**

in the event of a network partition it times out replies, sacrificing availability for consistency

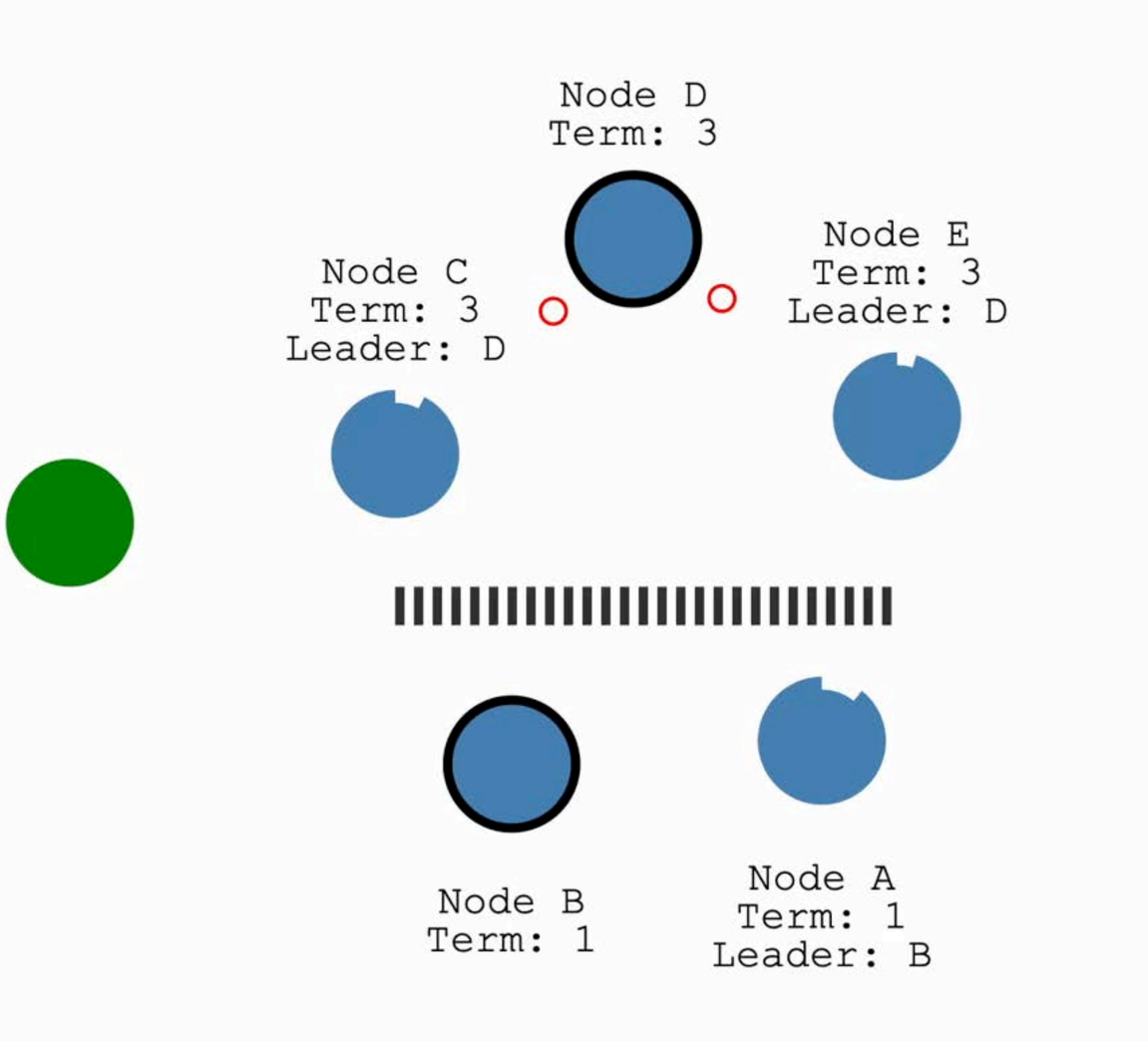
Network Partitions

- Raft can even stay consistent in the face of *network partitions*
- Partitions can result in two leaders
- Multiple clients may try to update both
- Any new log entries of the minority partition's leader remain *uncommitted*
- When the partition is healed, the minority leader will step down and match the majority leader's log, ensuring consistency.



Network Partitions

- Raft can even stay consistent in the face of *network partitions*
- Partitions can result in two leaders
- Multiple clients may try to update both
- Any new log entries of the minority partition's leader remain *uncommitted*
- When the partition is healed, the minority leader will step down and match the majority leader's log, ensuring consistency.



Sharding

Chapter 9

Replication vs Fragmentation

Replication involves duplicating data across multiple nodes for redundancy

- Allows for *fault tolerance*, *load balancing*, and *read performance*
- Main challenge: *consistency*

Replication vs Fragmentation

Replication involves duplicating data across multiple nodes for redundancy

- Allows for *fault tolerance*, *load balancing*, and *read performance*
- Main challenge: *consistency*

Fragmentation involves partitioning data, distributing subsets over multiple nodes

- Allows for *horizontal scaling*, *flexibility,* and *operation localisation*
- Main challenge: *joins/transactions*

Replication vs Fragmentation

Replication involves duplicating data across multiple nodes for redundancy

- Allows for *fault tolerance*, *load balancing*, and *read performance*
- Main challenge: *consistency*

Fragmentation involves partitioning data, distributing subsets over multiple nodes

- Allows for *horizontal scaling*, *flexibility,* and *operation localisation*
- Main challenge: *joins/transactions*

Large-scale production systems likely use a combination of both

Horizontal vs Vertical Partitioning

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Horizontal Partitions

HP1

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN

HP2

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Greatest **scalability** potential

Also improves **performance**
by *parallelising processing*

Vertical Partitions

VP1

CUSTOMER ID	FIRST NAME	LAST NAME
1	TAEKO	OHNUKI
2	O.V.	WRIGHT
3	SELDA	BAĞCAN
4	JIM	PEPPER

VP2

CUSTOMER ID	FAVORITE COLOR
1	BLUE
2	GREEN
3	PURPLE
4	AUBERGINE

Improves **query performance**
by *specialising shards*

... but limits horizontal scalability
and *schema changes*

Range Based Sharding

PRODUCT	PRICE
WIDGET	$118
GIZMO	$88
TRINKET	$37
THINGAMAJIG	$18
DOODAD	$60
TCHOTCHKE	$999

($0-$49.99)

PRODUCT	PRICE
TRINKET	$37
THINGAMAJIG	$18

($50-$99.99)

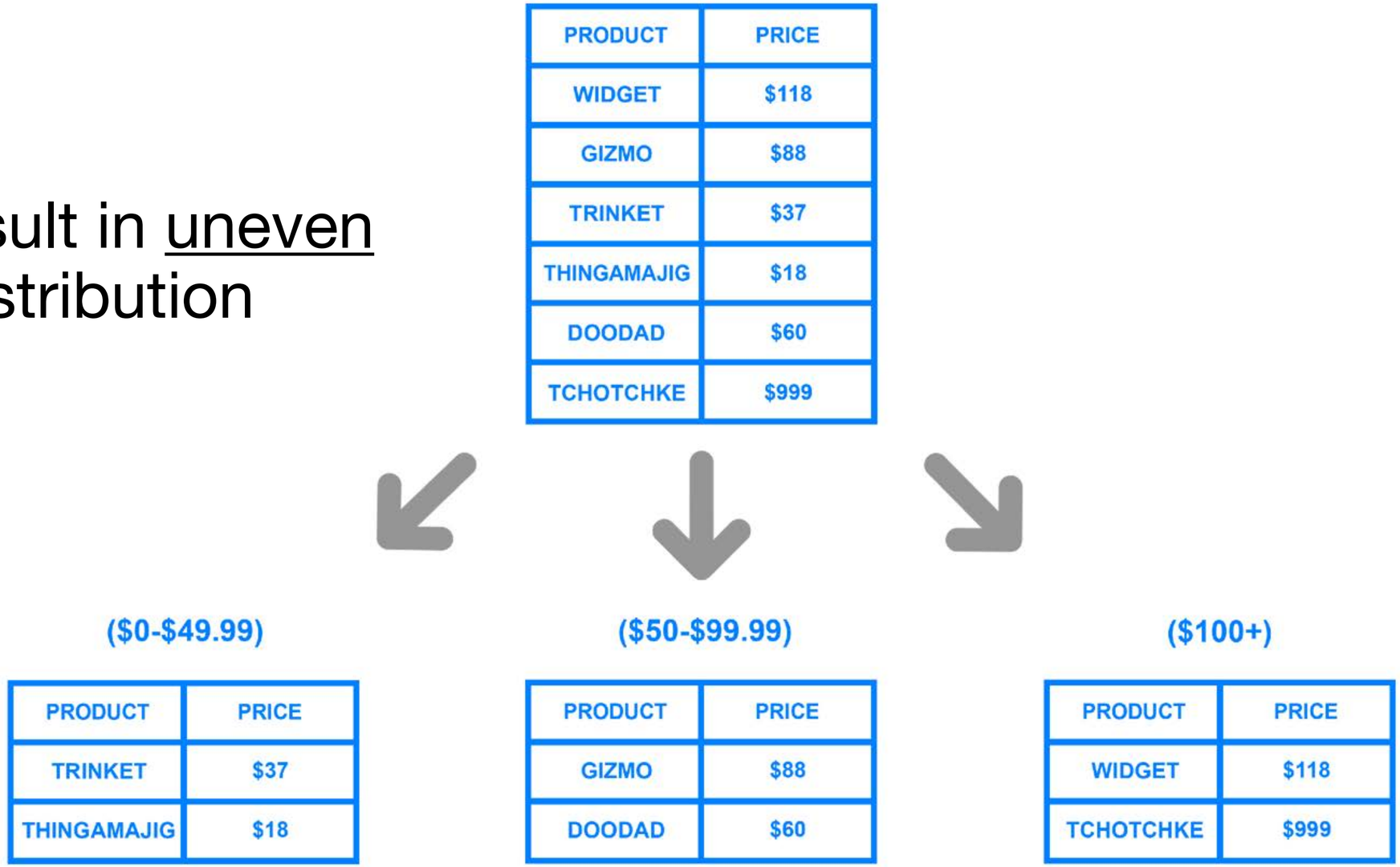
PRODUCT	PRICE
GIZMO	$88
DOODAD	$60

($100+)

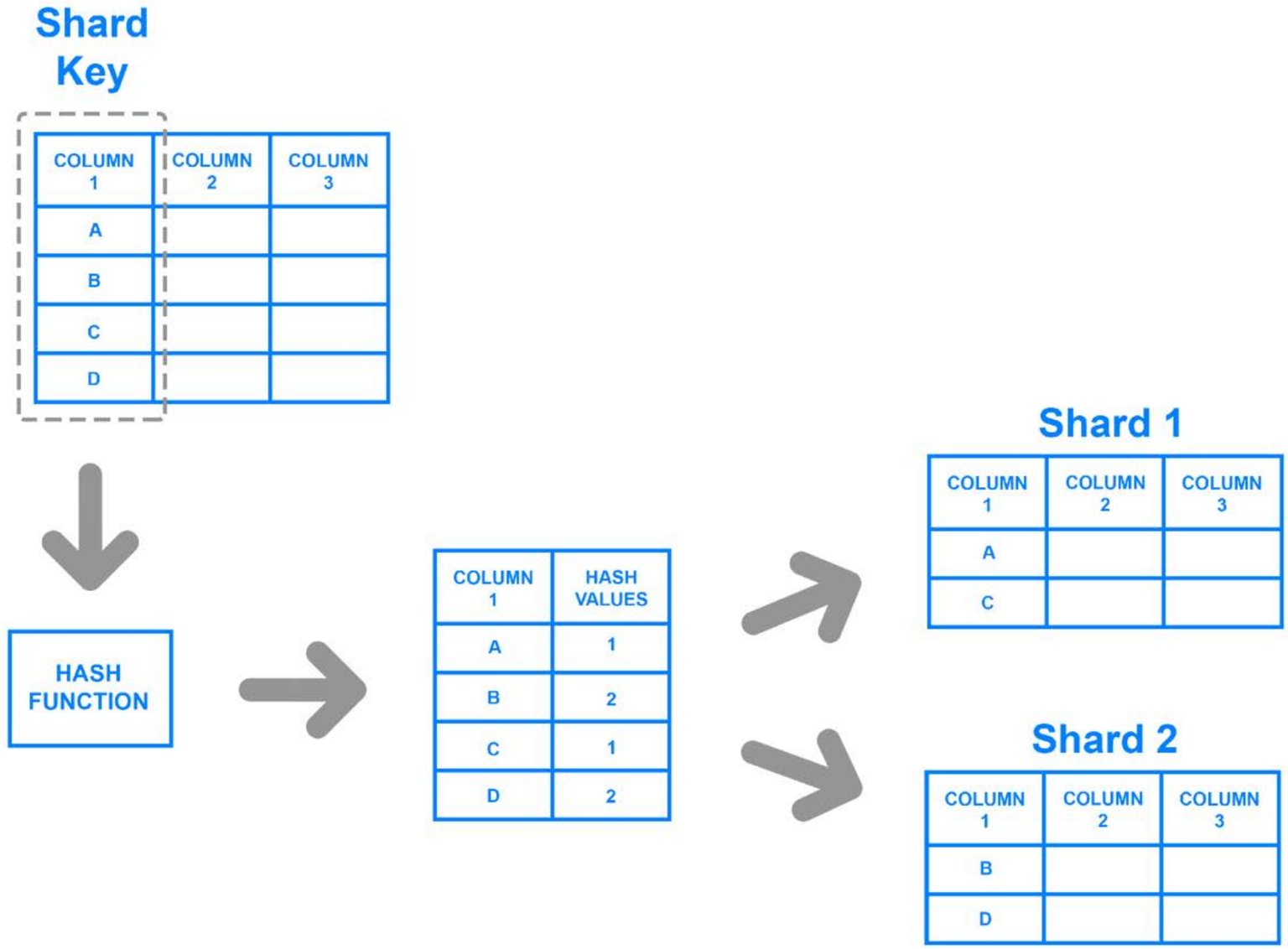
PRODUCT	PRICE
WIDGET	$118
TCHOTCHKE	$999

Range Based Sharding

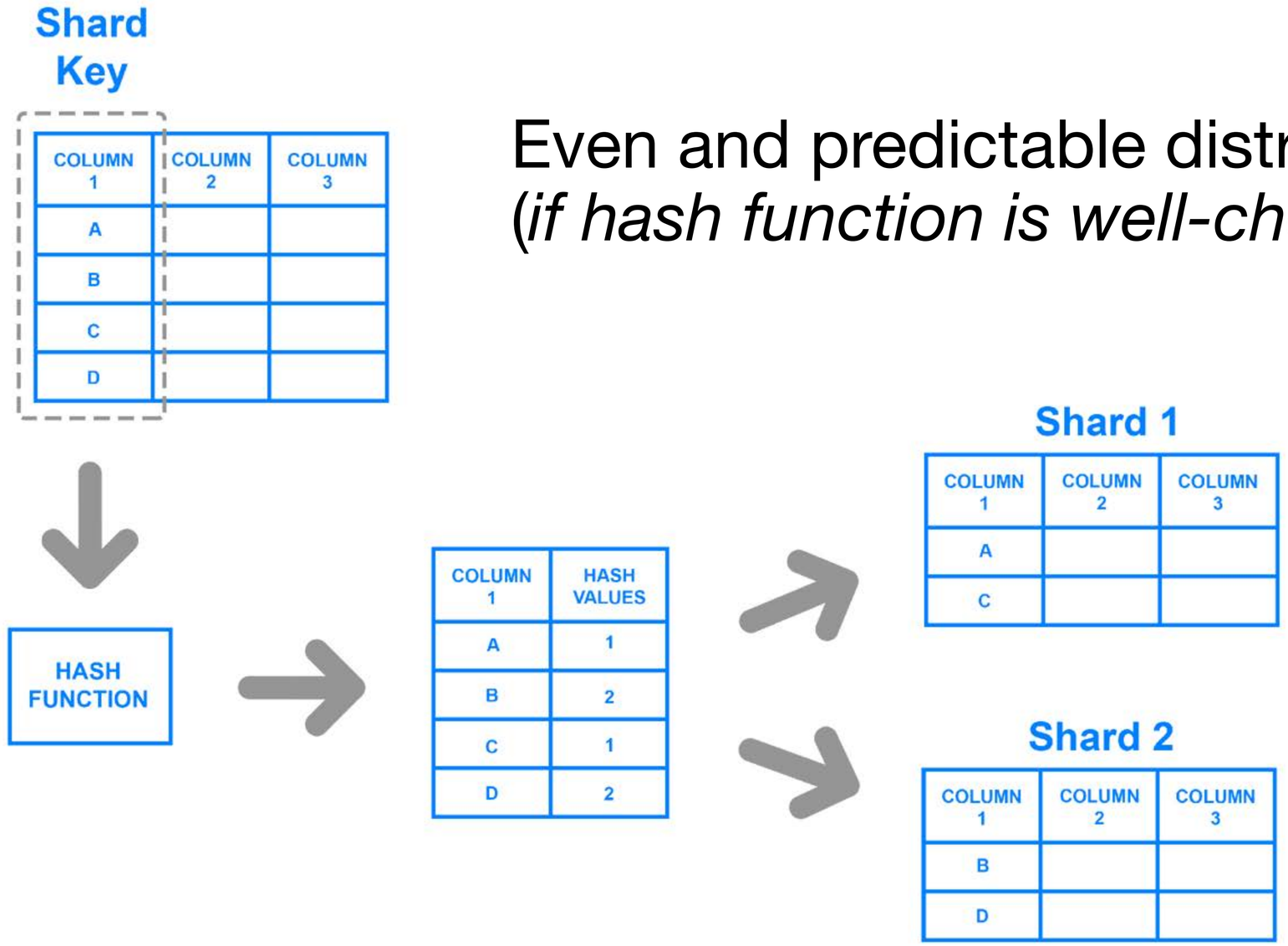
Can result in uneven data distribution



Key Based Sharding

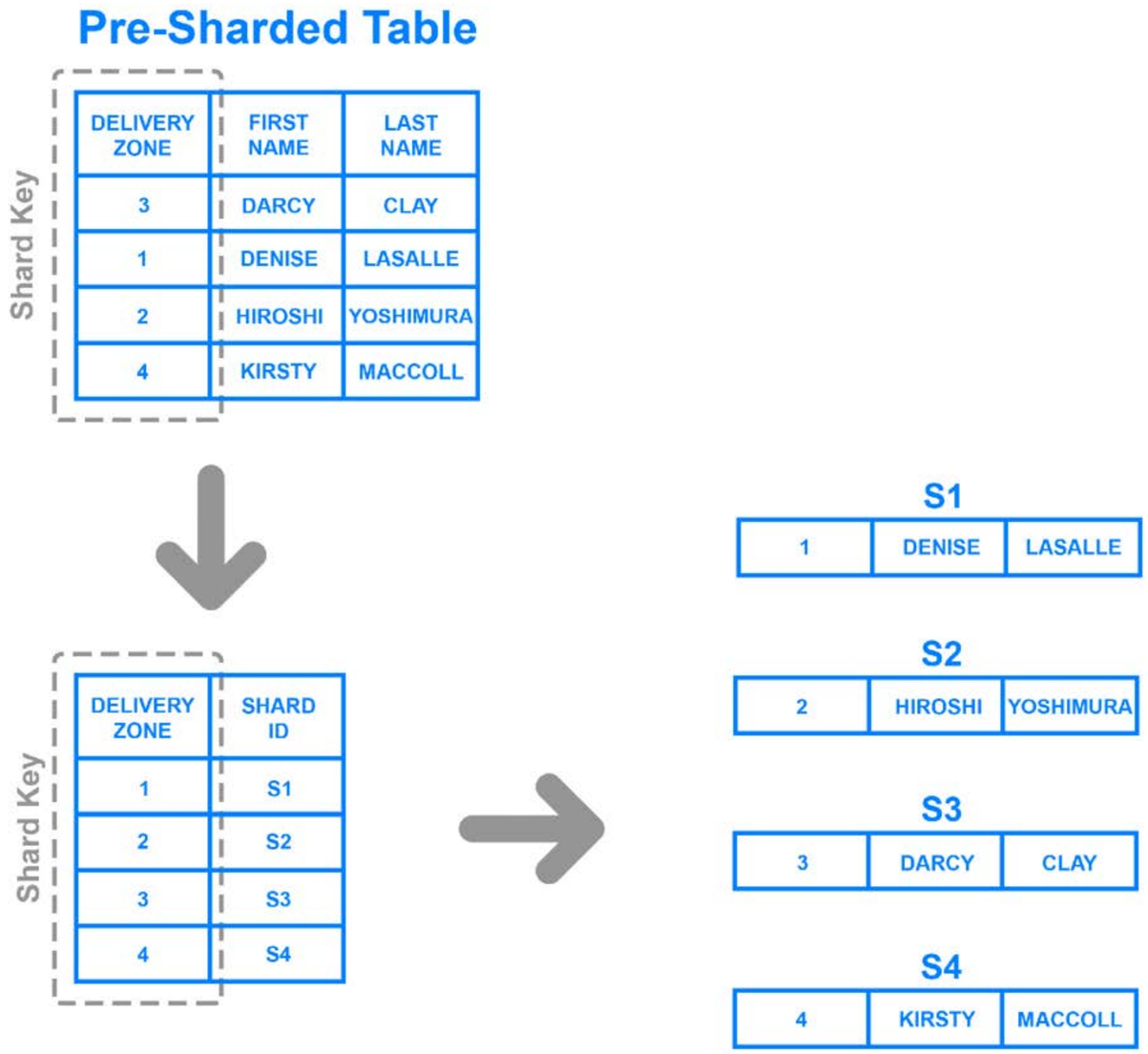


Key Based Sharding

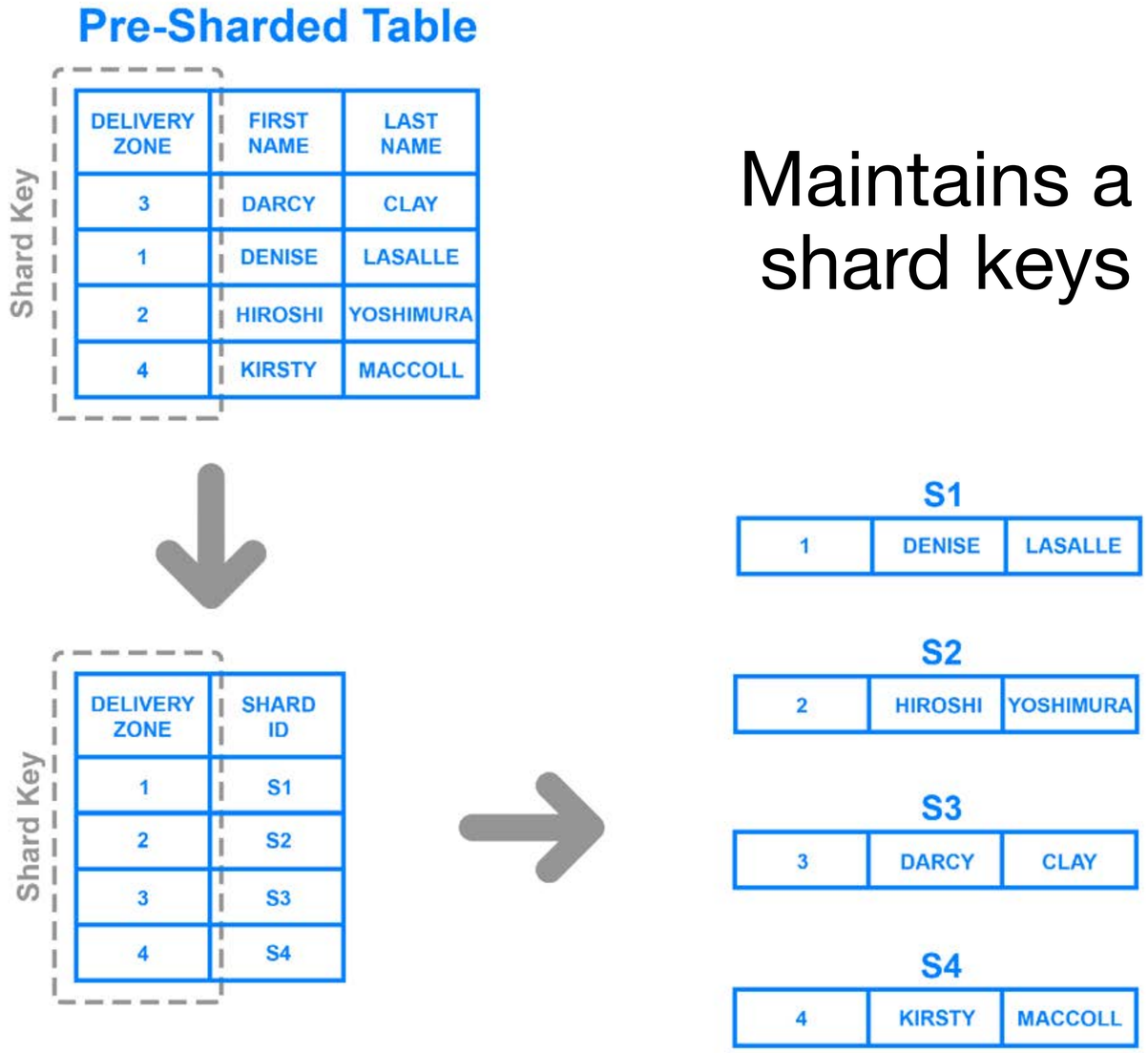


Even and predictable distribution
(*if hash function is well-chosen*)

Directory Based Sharding

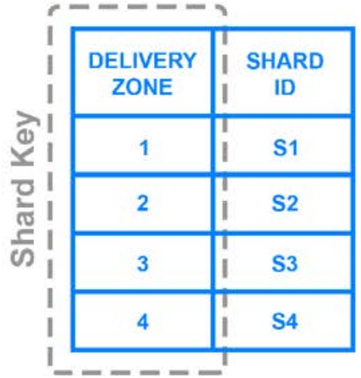


Directory Based Sharding



Maintains a *lookup table*, mapping shard keys to shard IDs

Directory Based Sharding



Maintains a *lookup table*, mapping shard keys to shard IDs

Most **flexible**

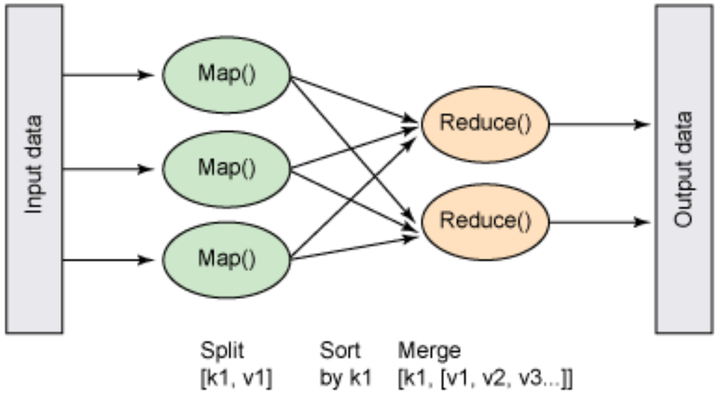
...but LUT forms **SPoF/bottleneck**

🅱ig Data

Chapter 10

MapReduce

Parallelised Operations on Big Data



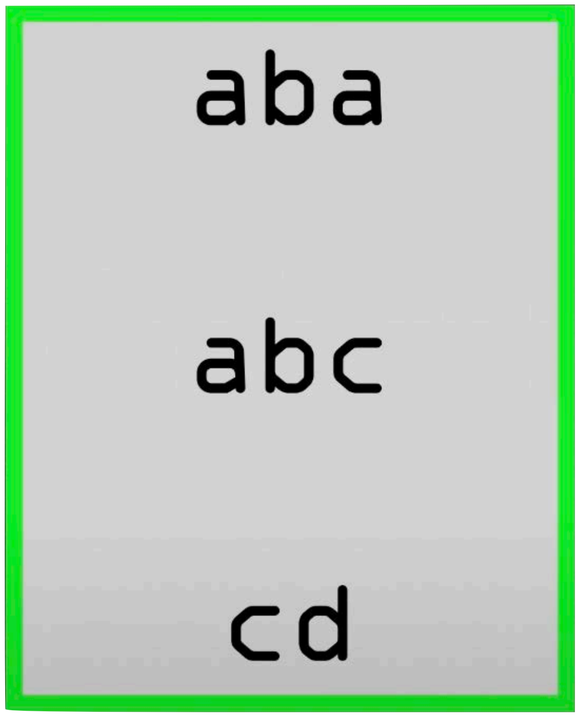
- Programming model and implementation for processing/generating big datasets using a parallel, distributed algorithm on a cluster
- Originated at *Google*, popularised via *Apache Hadoop* (open-source)
- Task must be composable into a *Map* phase and *Reduce* phase
  Map procedure performs the same computation across all elements
  Reduce method performs 'summary' operation to yield final result
- MapReduce framework handles orchestration of servers and parallelisation

MapReduce

Word Counting

Task: count number of occurrences (frequency) of each character in a file

Map procedure: create key-value pairs for each letter, initialised to 1

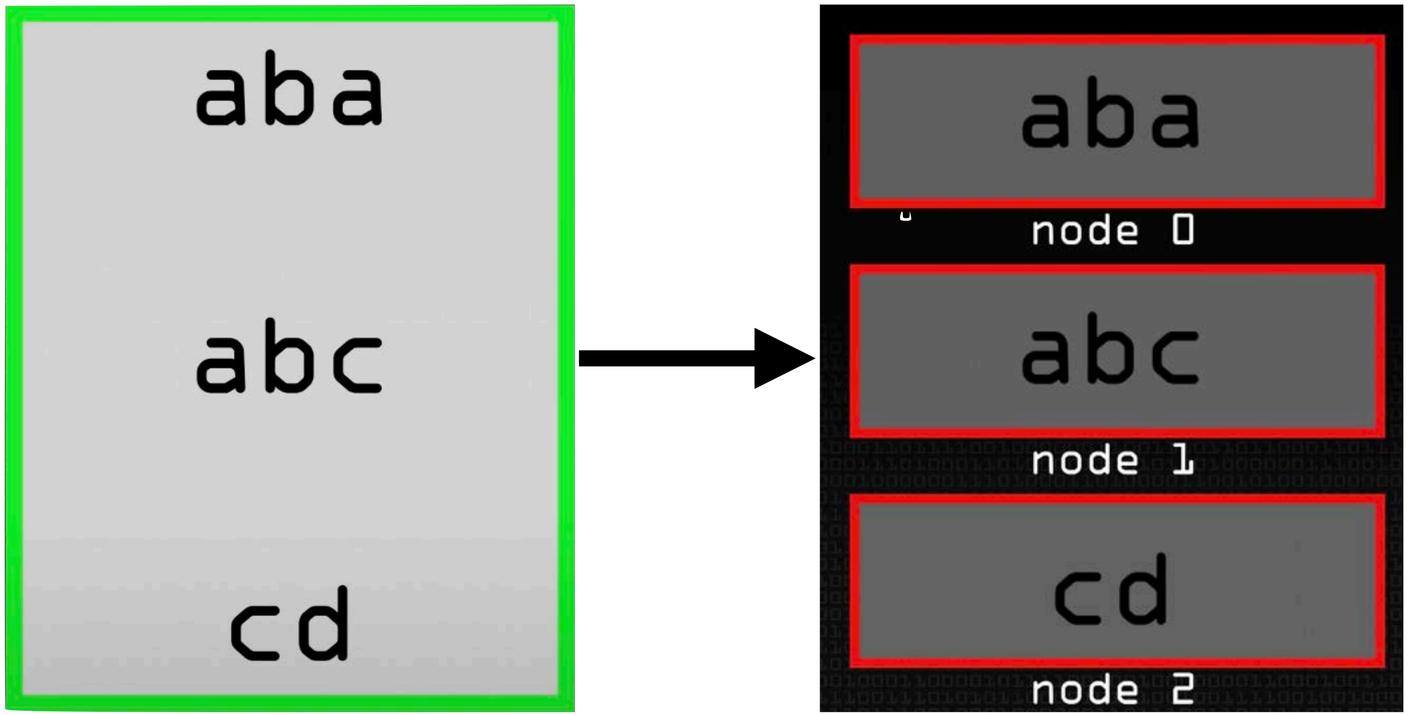


MapReduce

Word Counting

Task: count number of occurrences (frequency) of each character in a file

Map procedure: create key-value pairs for each letter, initialised to 1

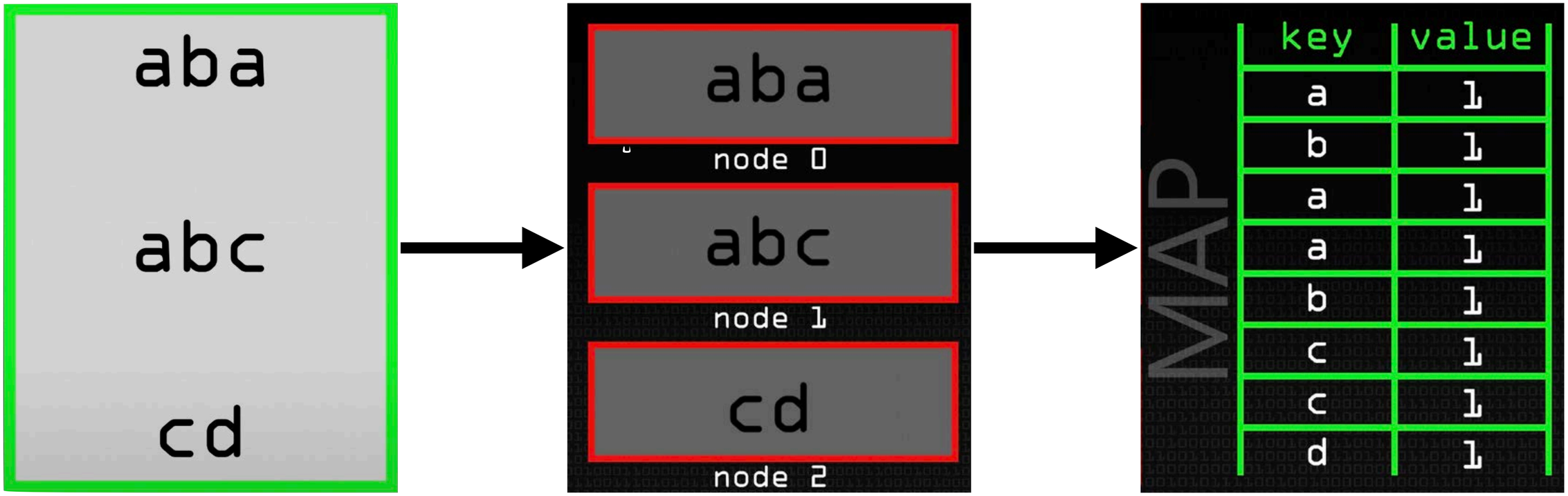


MapReduce

Word Counting

Task: count number of occurrences (frequency) of each character in a file

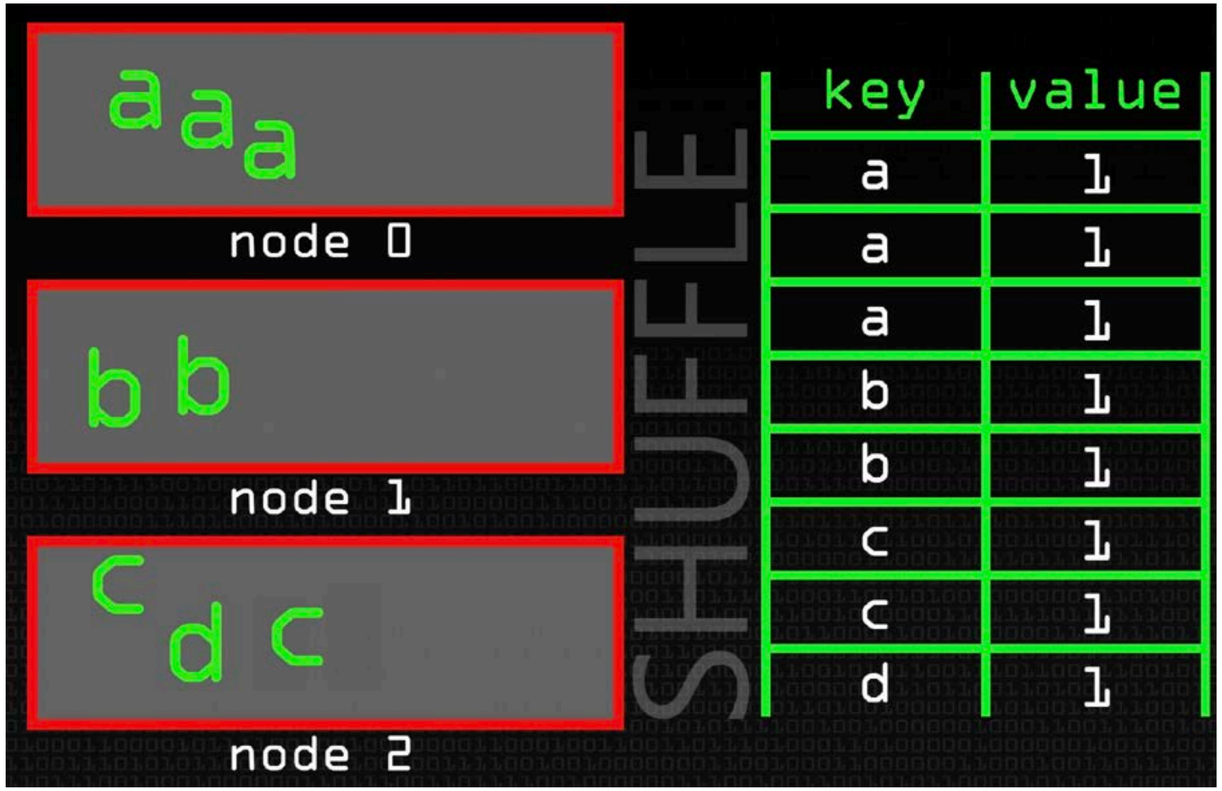
Map procedure: create key-value pairs for each letter, initialised to 1



MapReduce

Word Counting

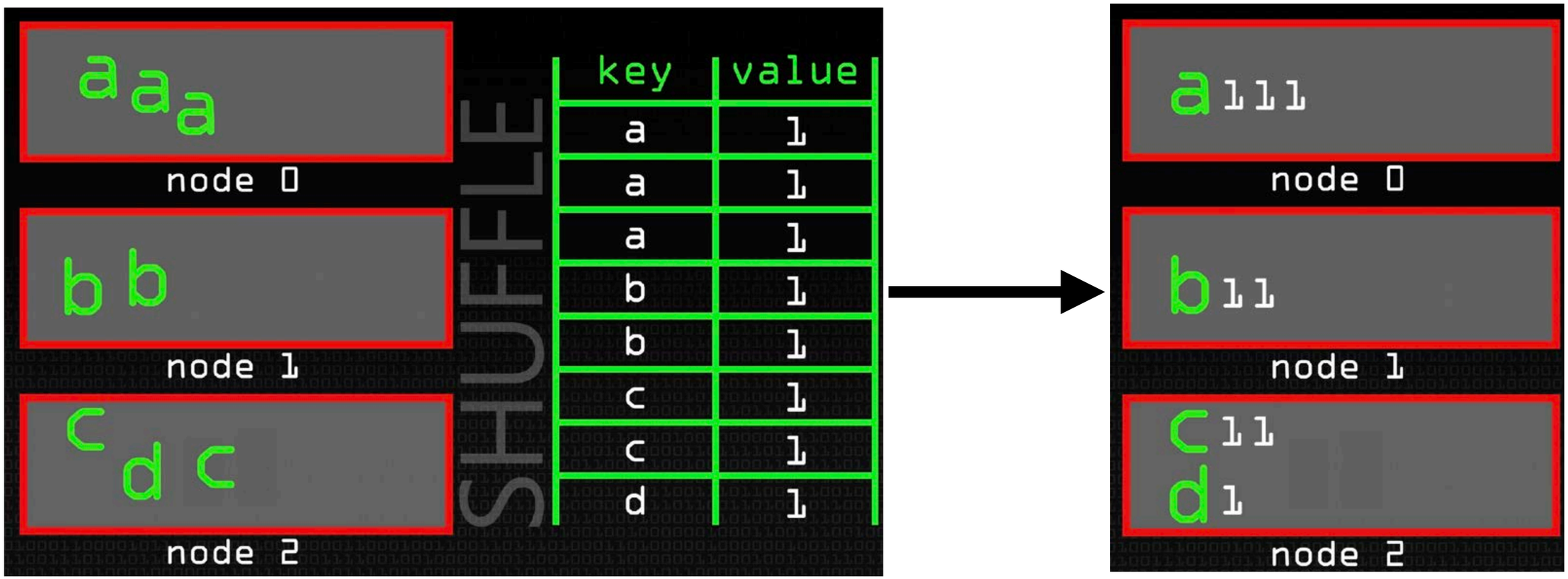
Shuffle (automatic) - nodes redistribute data based on *Map* output to group all data belonging to one key onto the same node



MapReduce

Word Counting

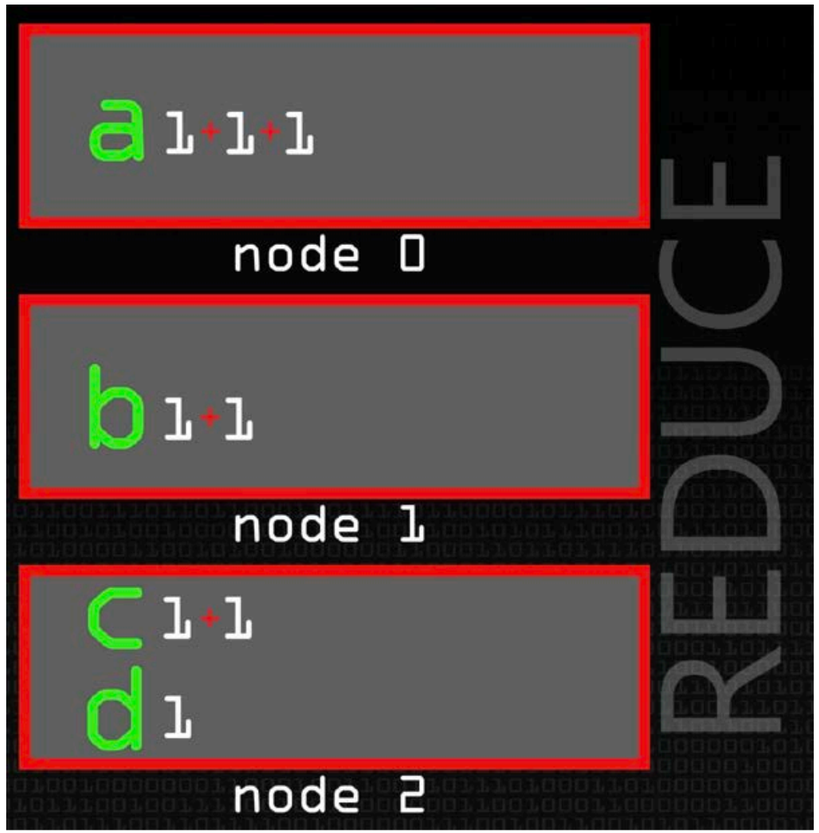
Shuffle (automatic) - nodes redistribute data based on *Map* output to group all data belonging to one key onto the same node



MapReduce

Word Counting

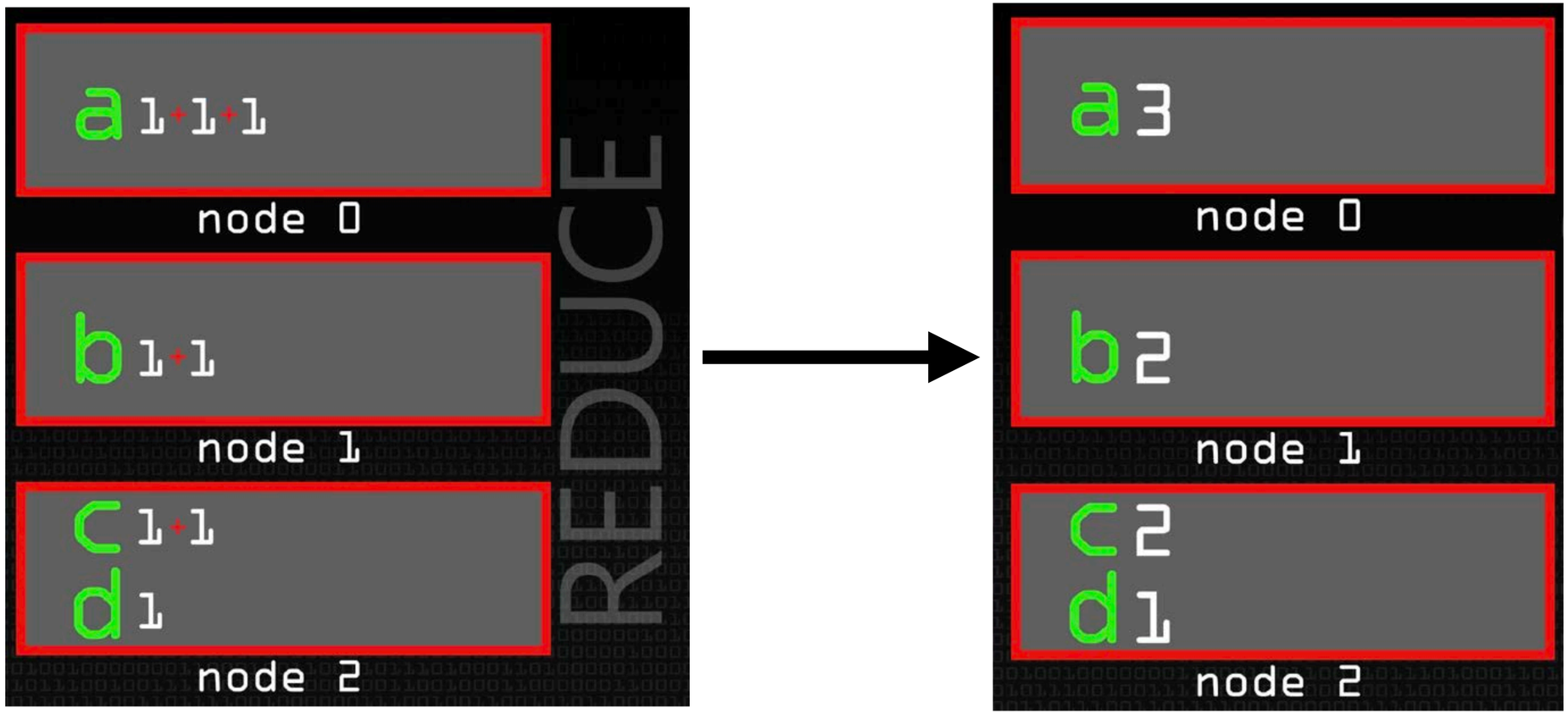
Reduce method: addition (+) operator



MapReduce

Word Counting

Reduce method: addition (+) operator



Summary

Topics Covered

Horizontal & Vertical **Scaling**

Static & Dynamic **Load balancing**

Consensus Algorithms

Asynchrony & FLP Impossibility Result

Raft Leader Election & Log Replication

Failure Modes in Distributed Systems

Network Partitions

Replication & Fragmentation

Range, Key & Directory Based **Sharding**

MapReduce for Big Data Processing

Q&A

Rushil Ambati

About Me

- Left Reading School in 2021
- 3rd Year of MEng Computing at Imperial College London
- Previously interned at various software companies
- Starting a work placement at a high-frequency trading firm
- Kabaddi wrestling athlete

Reach Out

- ambati.rushil@gmail.com
- LinkedIn