

# Distributed Computing

A-01. Introduction to the Course

# Welcome!

- This course will be shared between all 1<sup>st</sup> year students
- Different organization per curriculum
  - SSE: 6 credits (October~November)
  - DSE: 9 credits (October-December)

# What Is a Distributed System?

- A collection of **autonomous computing elements (nodes)** that appear to its users as a **single coherent system** (**van Steen & Tanenbaum**)
  - Colored text are links... Please follow them to know more!
- Distributed systems are **everywhere**: pretty much anything you see on the Internet is one...
- Also, a single multi-core computer can be seen as a distributed system; distributed system techniques can be and are applied even there (**Bauman et al.**)

# Distributed Computing is Hard

The **Eight fallacies of distributed computing** (whitepaper, tech talk):

- 1) The network is reliable
- 2) Latency is zero
- 3) Bandwidth is infinite
- 4) The network is secure
- 5) Topology doesn't change
- 6) There is one administrator
- 7) Transport cost is zero
- 8) The network is homogeneous

# Do I Need A Distributed System?

- **Availability:** if one computer (or 10) break down, my website/DB/fancy Ethereum app will still work
- **Performance:** a single-machine implementation won't be able to handle the load/won't be fast enough
- **Decentralization:** I don't want my system to be controlled by a single entity
- And all these things are related in non-trivial ways... **We'll see!**

# Making a System Coherent

- We've seen that distributed systems are about making disparate machines **coherent**
- To be coherent, nodes need to **collaborate**
  - We need **synchronization** (there is no **global clock**)
  - We need to **manage group membership & authorizations**
  - We need to deal with **node failures**

# Distributed Systems: a Huge Topic

- Many courses go in **depth**: you have a (small enough) topic and they teach you everything about it
- Here, we'll go in **breadth**: we'll introduce several topics at a high level, and go deeper on a few
  - Idea: giving you starting points for learning on your own, including after university

# How We See the Course

- Security Software and Engineering
  - You'll be among the ones **designing** and **securing** these systems
  - You'll be the “mechanics” who will be tinkering with these systems
- Data Science and Engineering
  - You'll be **using** these systems
  - You'll be the “pilots” who need to know their systems to use them well

# Organization of the Course

- Part A (6 CFU, **everybody**):
  - Distributed systems in general
  - Lessons held by Matteo Dell'Amico
- Part B (3 CFU, **DSE**):
  - Big Data Engines
  - Lessons held by Giorgio Delzanno

# Exams: SSE

- You'll have **assignments** based on a simulator you have to complete
  - Questions like “in this scenario, which are the best design choices”?
  - Alone or in groups of two people
- You'll **tinker** with the simulation and write reports answering the question
- There will be a peer review process: you'll be evaluated based on it
- Oral exams, where you'll present the assignments and will be asked questions about all the program

# Exams: DSE

- You'll have **exercises** on Apache Spark to do during the year
- A written final examination

# Part A: Some Topics We'll Touch

- Making systems **consistent**
  - **Consensus** mechanisms
- What **queueing theory** tells us
  - Effects of sharing load between servers
- Handling data efficiently
  - Modeling systems with data **replication**
  - **Erasure coding**: the gifts of coding theory
- Introduction to decentralized systems
- Introduction to big data engines

# Distributed Computing

A-02. Transactions, ACID and CAP

# Transactions

- A transaction for us is an independent modification in a system that stores data
  - Database, file system, ...
- While they may change **several** parts of the system at once, we think about them as a single modification
- When money is transferred, it is “simultaneously” removed from one account and put in another one
- A directory is removed
- A new version of a file is saved

# ACID Properties (1)

- **Atomicity, Consistency, Isolation, Durability**
- A classic set of properties to implement transactions
- Makes it **easier** to think about how the system behaves
- Implemented in 1973 (Grey and Reuter 1993, page 42), even though the acronym was coined 10 years later (Härder and Reuter 1983)

# ACID properties (2)

- **Atomicity**: each transaction is treated as a single unit, that is it either succeeds or fails completely
  - E.g., If money is taken from my account, it gets to the destination
  - E.g., If I save a new version of a file, nobody will see a “half-written” version of it
- **Consistency (Correctness)**: the system remains in a valid state
  - E.g., All accounts have non-negative balance
  - E.g., A non-deleted directory is reachable from the root

# ACID Properties (3)

- **Isolation:** even if transactions may be run concurrently, the system behaves as if they've been running sequentially
  - Transactions are seen as “ordered”
- **Durability:** Even in case of a system failure, the result of the transaction is not lost

# The CAP Theorem

- Proposed as a conjecture by **Fox and Brewer** in 1999
- Proven as a theorem by **Gilbert and Lynch** in 2002
- In a system (that allows transactions), you cannot have all of **consistency, availability** and **partition tolerance**

# C, A and P

- **Consistency:** every read receives the most recent write or an error
- **Availability:** every request receives a non-error response
- **Partition Tolerance:** the system keeps working even if an arbitrary number of messages between the nodes of our distributed system is dropped

# The Easy Proof

- Suppose the system is partitioned in two parts,  $G_1$  and  $G_2$ : no communication happens between them
- A write happens in  $G_1$
- A read happens in  $G_2$
- The result of the write is not accessible from  $G_2$ , so one of these happens:
  - The system returns an error (we lose **availability**)
  - The system returns old data (we lose **consistency**)
  - The system doesn't reply (we lose **partition tolerance**)

# Questions!

- [app.wooclap.com/DC23UNIGE](https://app.wooclap.com/DC23UNIGE)



# The Not-So-Obvious Consequences

- In any distributed system, you have a trade-off:
  - Either (part of) your system will be **offline** until the network partition is resolved
  - Or you will have to live with inconsistent and stale data
- Later in the course, we'll dive in work that explores this tradeoff. Distributed systems are very often about tradeoffs!
  - A piece about how this impacted system design by Brewer in 2012

# Examples of Non-ACID Systems

- Can you think of systems that can work with inconsistent functionality?
  - GIT (conflicts)
  - DNS
  - Social networks
  - NoSQL databases

# Distributed Computing

A-03. Consensus

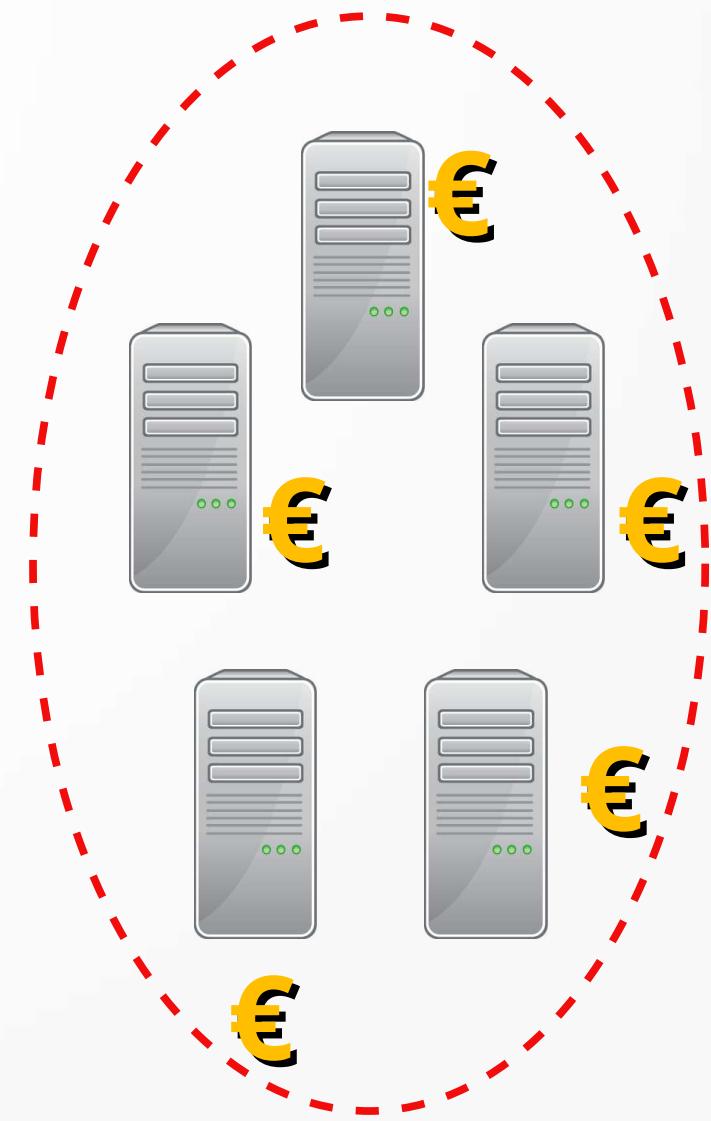
# Scaling Vertically

- Your (videogame, finance service, etc.) start-up needs a server
  - You run one (or use the cloud). All is fine.
- Congratulations, you're successful! You need a bigger server!
  - To do twice as many operations, you need to spend more than twice, but it's ok
- At some point the cost is not manageable, or a single-server solution still doesn't work
- Also, now that you're a huge business, you can't afford to lose money because there's a black-out or a flood



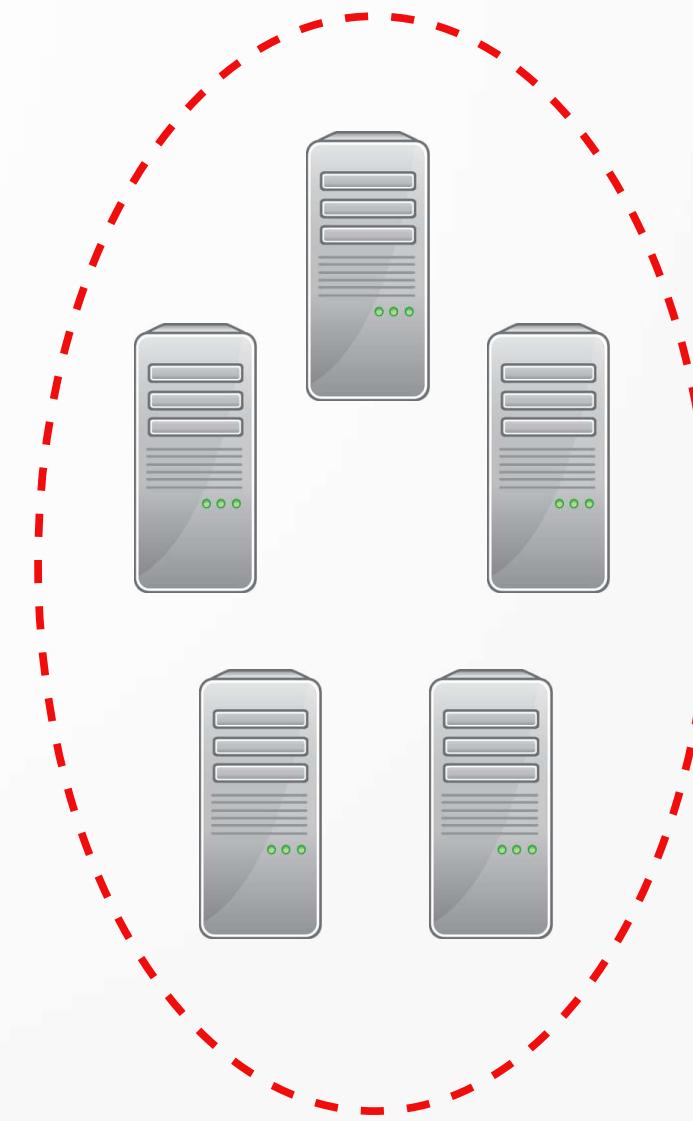
# Scaling Horizontally

- A desireable alternative would be to **divide work between multiple servers**
  - We can then buy the servers with the best performance/price ratio
  - We can distribute them geographically for better redundancy (e.g., catastrophes)
- Two problems:
  - Coordination: making them work together **as if they were a single computer**
  - Scalability: make it so that **costs of coordination** are not huge



# Remember ACID?

- Transaction properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Taken together, our system behaves “as if it is a single always-on machine”, processing all transactions one after the other
- How to pull this off?



# **State Machine Replication**

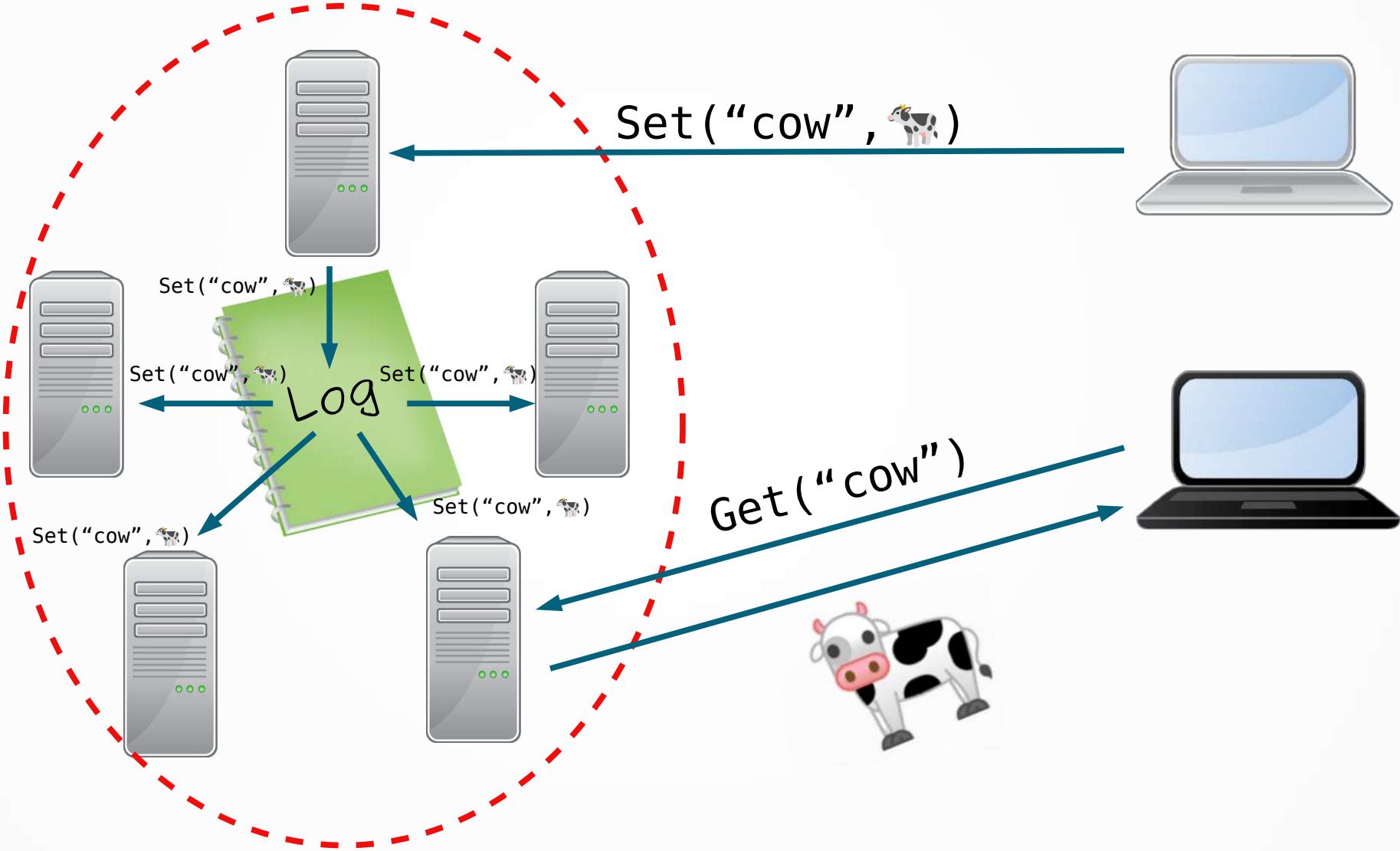
# Deterministic State Machines

- Each machine has a **state** that determines its future behavior
  - Represents e.g. memory, registers, configuration
- Each machine has the same **starting state**
- Receiving an **input** changes the state (and may produce output) **deterministically**
  - I.e., there is a transition function ( $\text{Input} \times \text{State} \rightarrow \text{State} \times \text{Output}$ )
- Hence, **two machines** are in the same state **if they have received the same input in the same order**

# Read-Intensive Workloads

- Input as a **list (log) of commands**
- In many architectures, **most commands don't change the state**—e.g.:
  - Read a file from a distributed filesystem
  - Perform a database query
  - Get a webpage
- Our distributed system can be based on servers that
  - Work like deterministic **state machines**
  - Put “write” commands that change the state in a **shared log**, and execute them all on all machines
  - Run “read” commands **locally**
- If most of the workload is made of “read”, our system will scale well

# An Example Key-Value Store



# How To Implement a Shared Log?

- The **fail-stop** scenario:
  - One-to-one messages only
  - They can take arbitrary time
  - They can **be lost**
  - Computers can **stop for an arbitrary time**
  - But they **don't lose data**
- Kind of like having to communicate only via SMS when you have bad coverage



# Note: Other Consensus Problems

- There is a worse failure model than fail-stop
- Broken nodes may send **any** message, maybe even acting **maliciously**
- This is the kind of scenario you need in **cryptocurrencies**
- This is called **byzantine** consensus
  - Solved by classical algorithms and blockchains

# Paxos

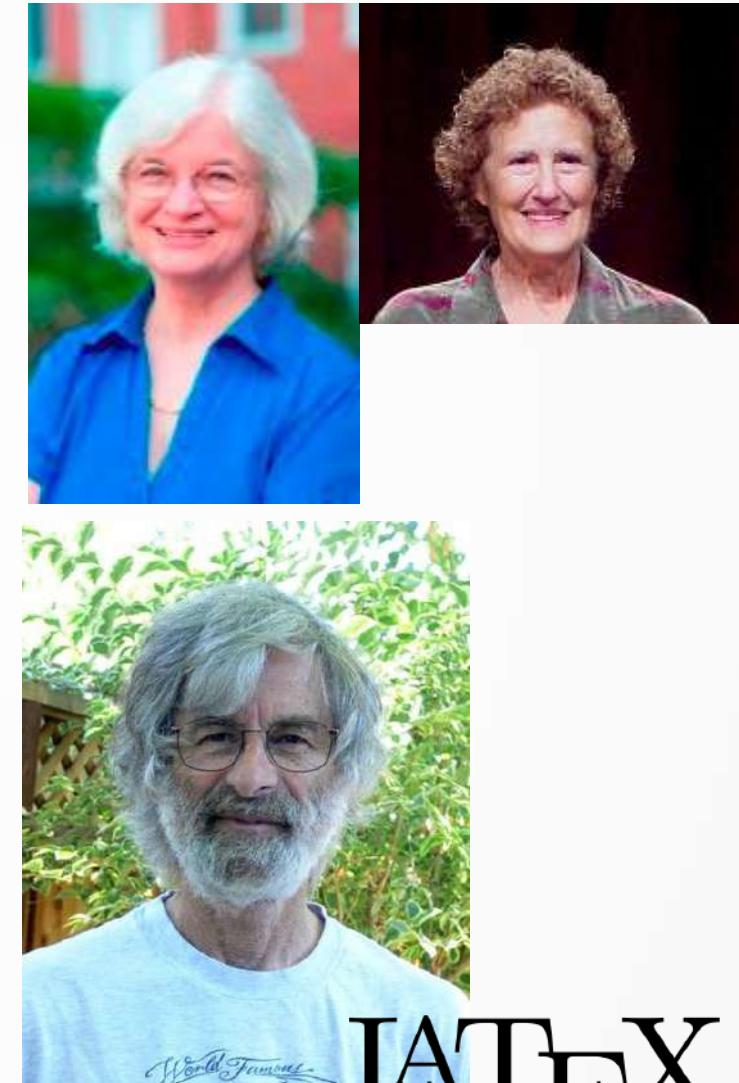
(or *How to confuse people for a quarter of a century*)

# See Also

- Lesson by Chris Colohan
- Talk by Luis Quesada Torres

# The Context

- In the '80s, Nancy Lynch and Barbara Liskov built a reliable redundant distributed system
  - No proof their system was correct for every single corner case
- Leslie Lamport wanted to show that they made mistakes, and their goal was impossible
  - He failed in that, because he found an algorithm that **could** do it: Paxos



LA<sup>T</sup>E<sub>X</sub>

# The Confusing Story of Paxos

## The Part-Time Parliament

LESLIE LAMPORT

Digital Equipment Corporation

- Paper submitted in **1989**
- Reviewers didn't understand it
- Only published in **1998!**
- Still was considered difficult

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.



For example, legislator *Αἰνύχος*'s ledger had the entry

155: *The olive tax is 3 drachmas per ton*

132: *Lamps must use only olive oil*

# Paxos Made Simple (?)

## Paxos Made Simple

Leslie Lamport

01 Nov 2001

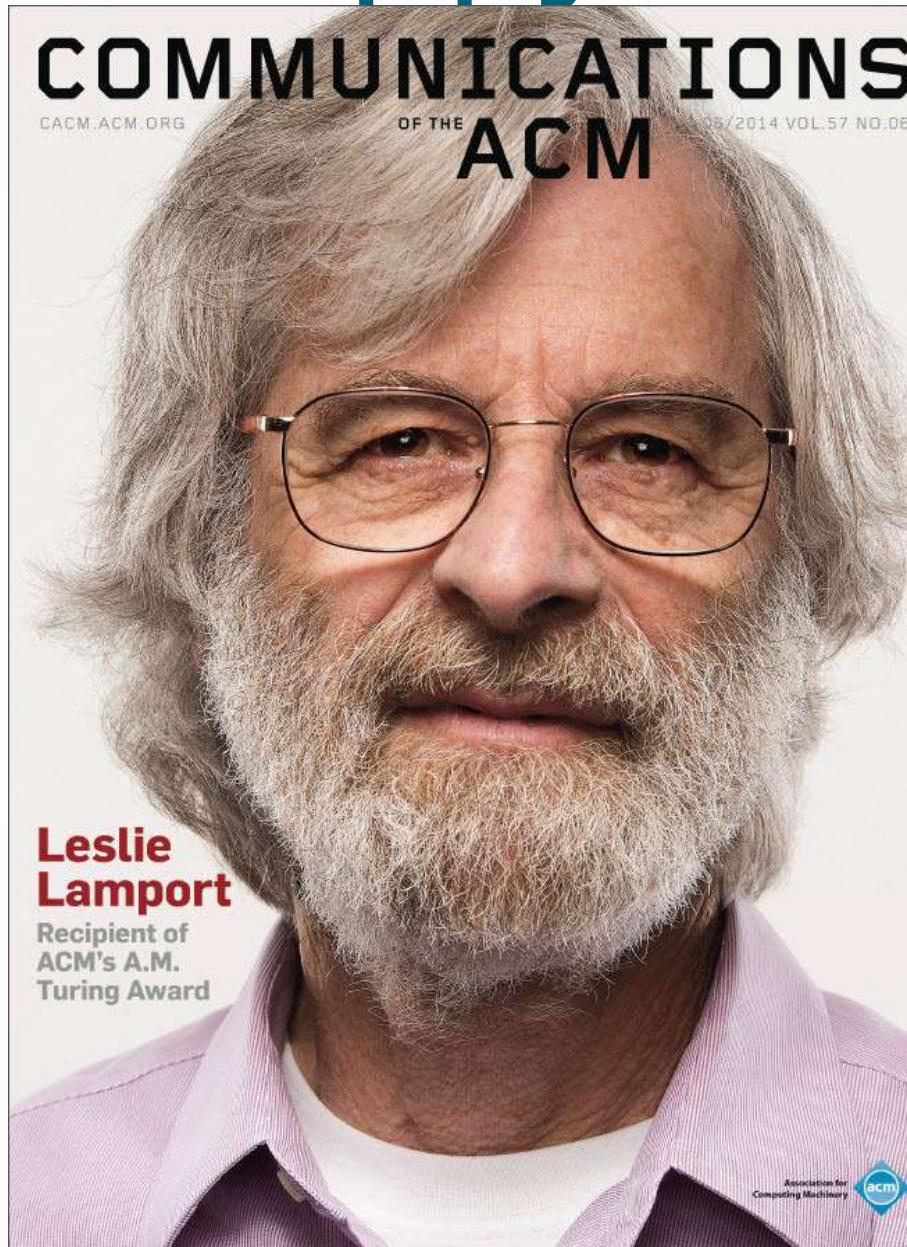
### Abstract

The Paxos algorithm, when presented in plain English, is very simple.

However... ([Ongaro and Ousterhout 2014](#))

Unfortunately, Paxos has two significant drawbacks. The first drawback is that Paxos is exceptionally difficult to understand. The full explanation [15] is notoriously opaque; few people succeed in understanding it, and only with great effort. As a result, there have been several attempts to explain Paxos in simpler terms [16, 20, 21]. These explanations focus on the single-decree subset, yet they are still challenging. In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers. We struggled with Paxos ourselves; we were not able to understand the complete protocol until after reading several simplified explanations and designing our own alternative protocol, a process that took almost a year.

# Lamport's Happy Ending



# So, What Is Paxos About?

- **Consensus**: creating a **shared log** in an asynchronous distributed system
- More precisely, this problem is solved by *Multi-Paxos*
- The basic Paxos algorithm finds consensus on **one** log entry
- Basic Paxos is *not* so difficult...

# How Does Paxos Work?

- Say you have the old mobile phone we've seen before, and want to decide with your friends whether to go for pizza or sushi
  - You want to all agree and go to the same place
- Works like this:
  - 1) Send to all “*Hey, what are we doing tonight?*”
  - 2) If most of your friends answered and nobody has plans, propose (say, “*pizza*”) and send to all
  - 3) If most of your friends agree, then it’s decided.
- Doesn’t sound very complicated, of course the devil is in the details...



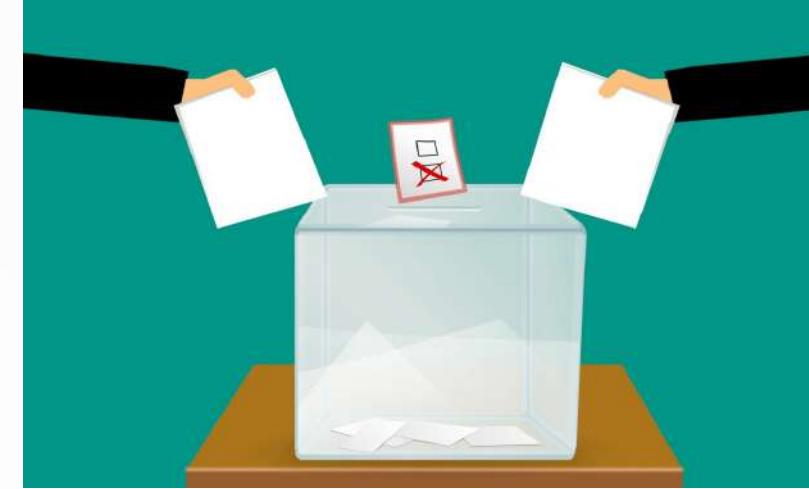
# Fail-Stop Failure Model

- *Failed* nodes stop working (e.g., your mobile has no connectivity)
  - They may resume (e.g., you eventually get signal)
  - Any message may get lost or arrive even after days... (you know, SMS...)
- **But**
  - Nobody loses their state (Nokia 3310 lasts forever)
  - Everybody is honest and makes no mistakes
  - To deal with malicious behavior, you'll need **byzantine** failure models



# Majority Wins

- We want to **tolerate  $n$  failures**
- We'll need  **$2n+1$  servers**
- A decision is taken when a majority of  **$n+1$  nodes** agree on it
  - Key reason: **two majorities must intersect**
  - Hence, **at least one participant will see a conflict** and avoid it
- Note: unlike regular elections, there are **no conflicting interests here**
  - Participants just want to **agree on something**

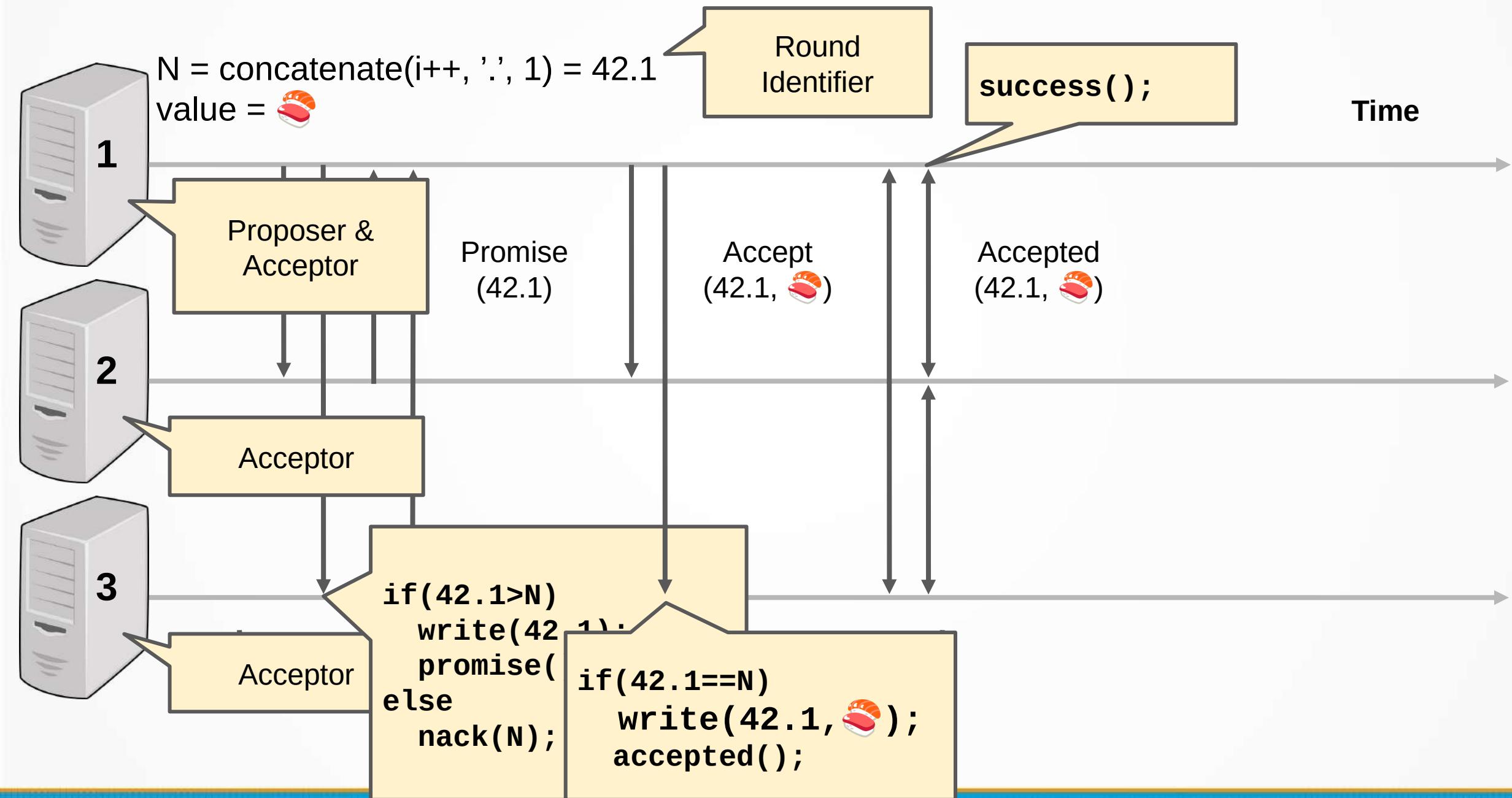


# Thanks!

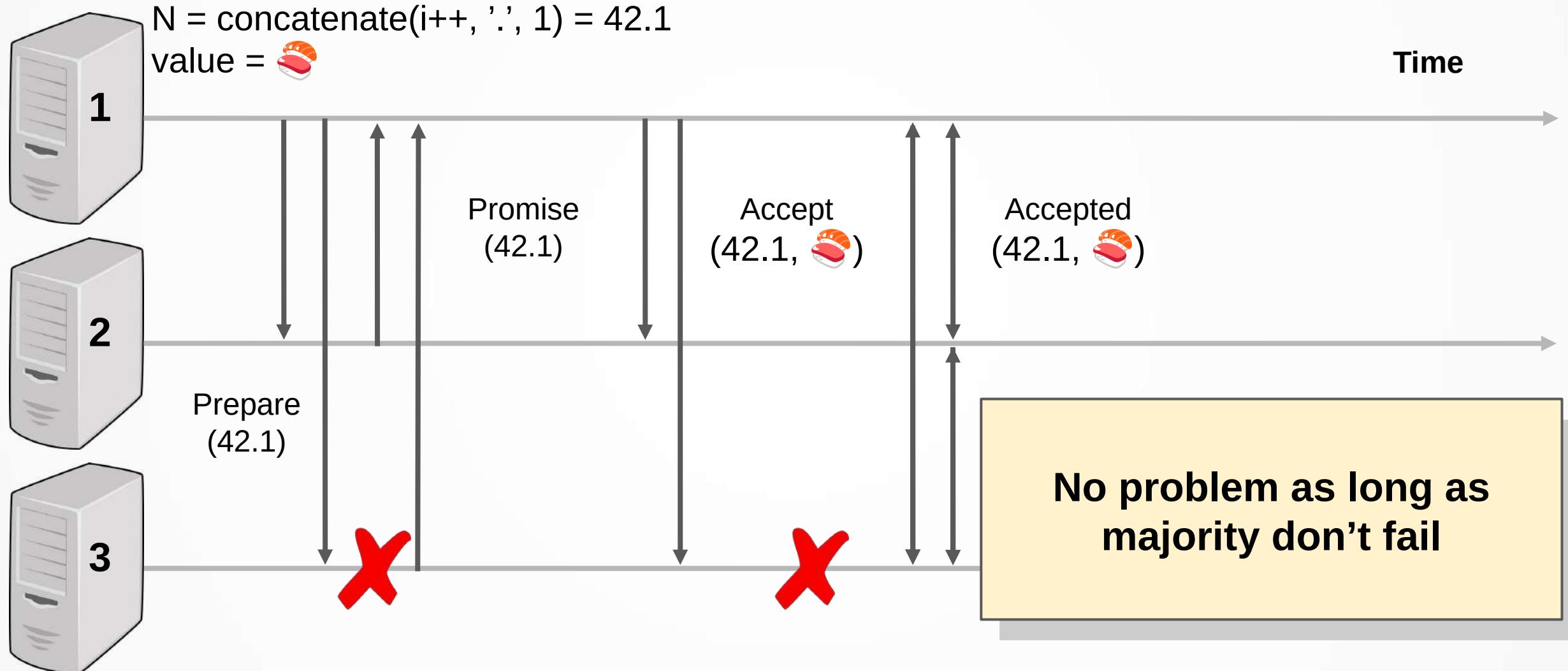


- The next slides are coming, with minor modifications and with permission, from the course by Chris Colohan at <http://www.distributedsystemscourse.com/>. Check it out!

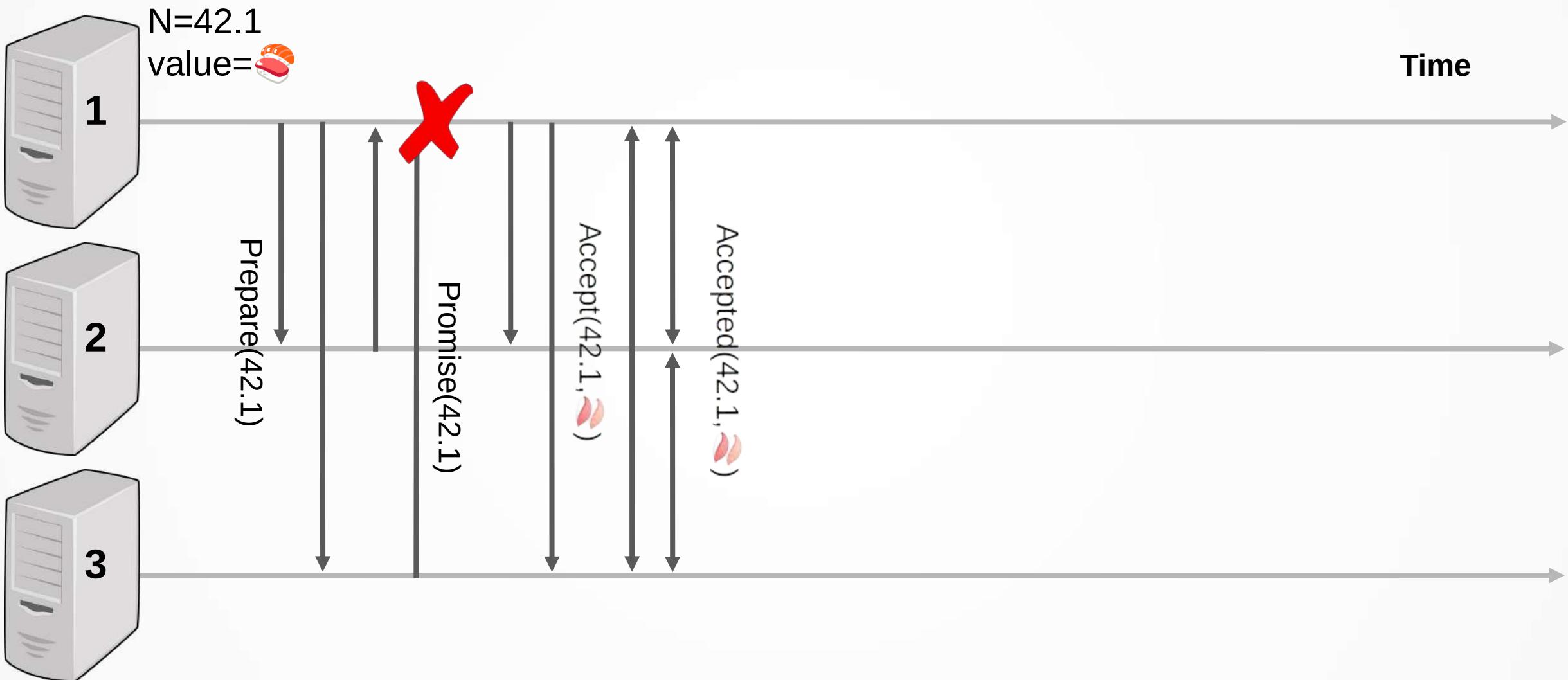
# Basic Paxos



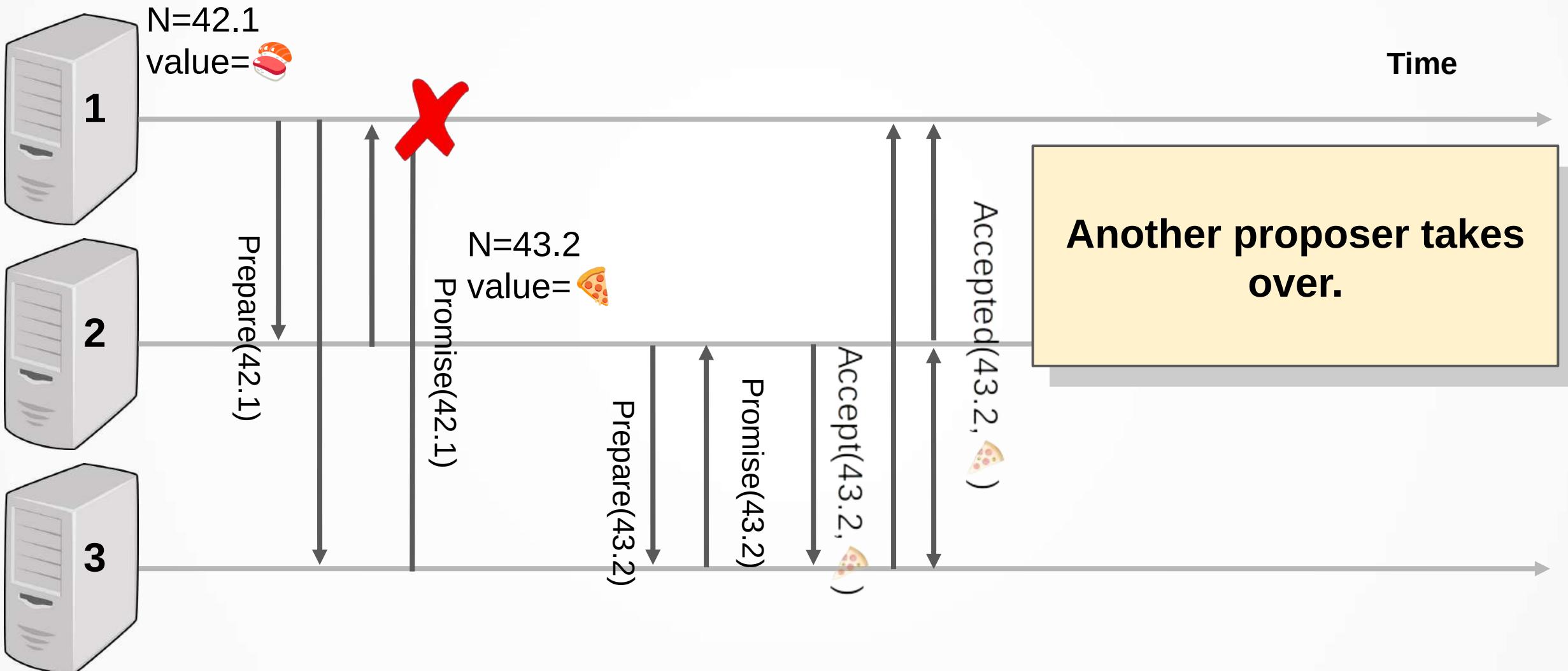
# Failures: Acceptor



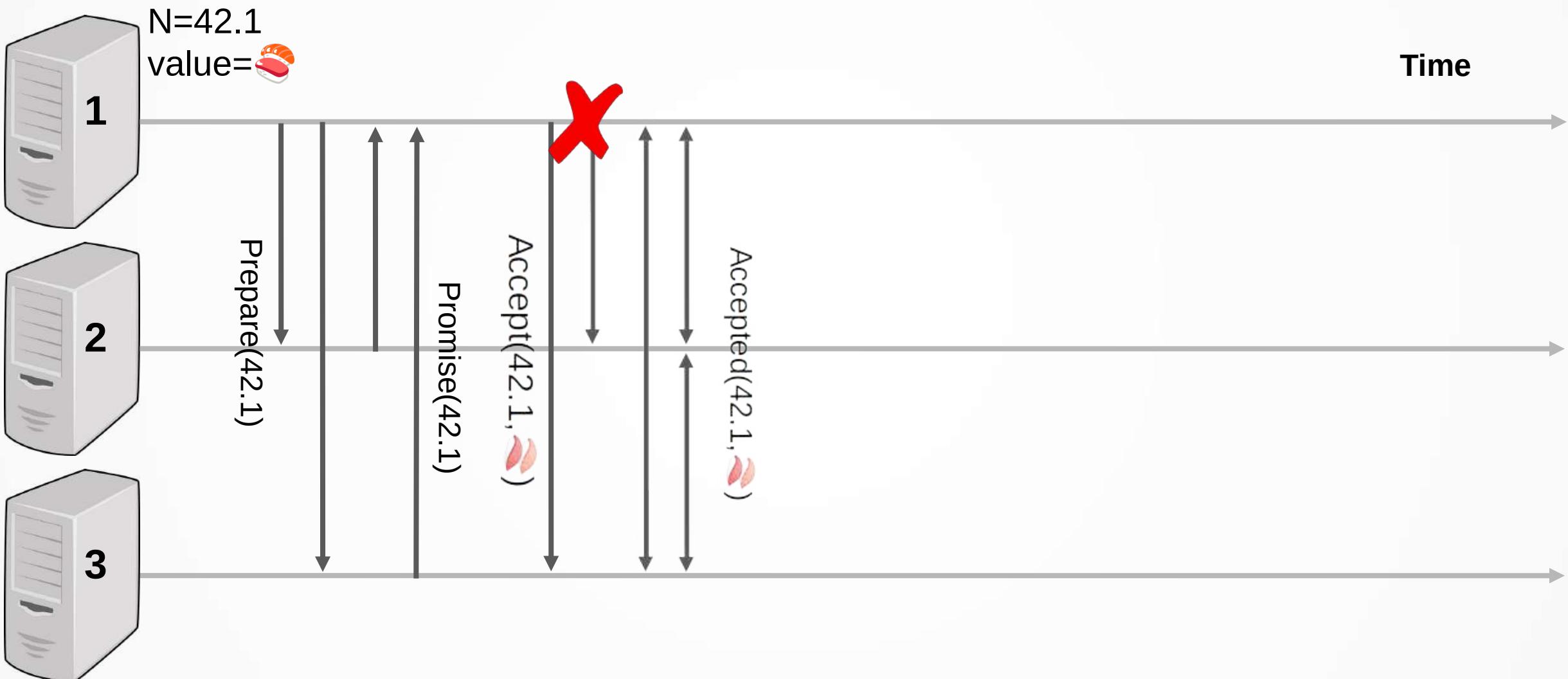
# Failures: Proposer in Prepare Phase



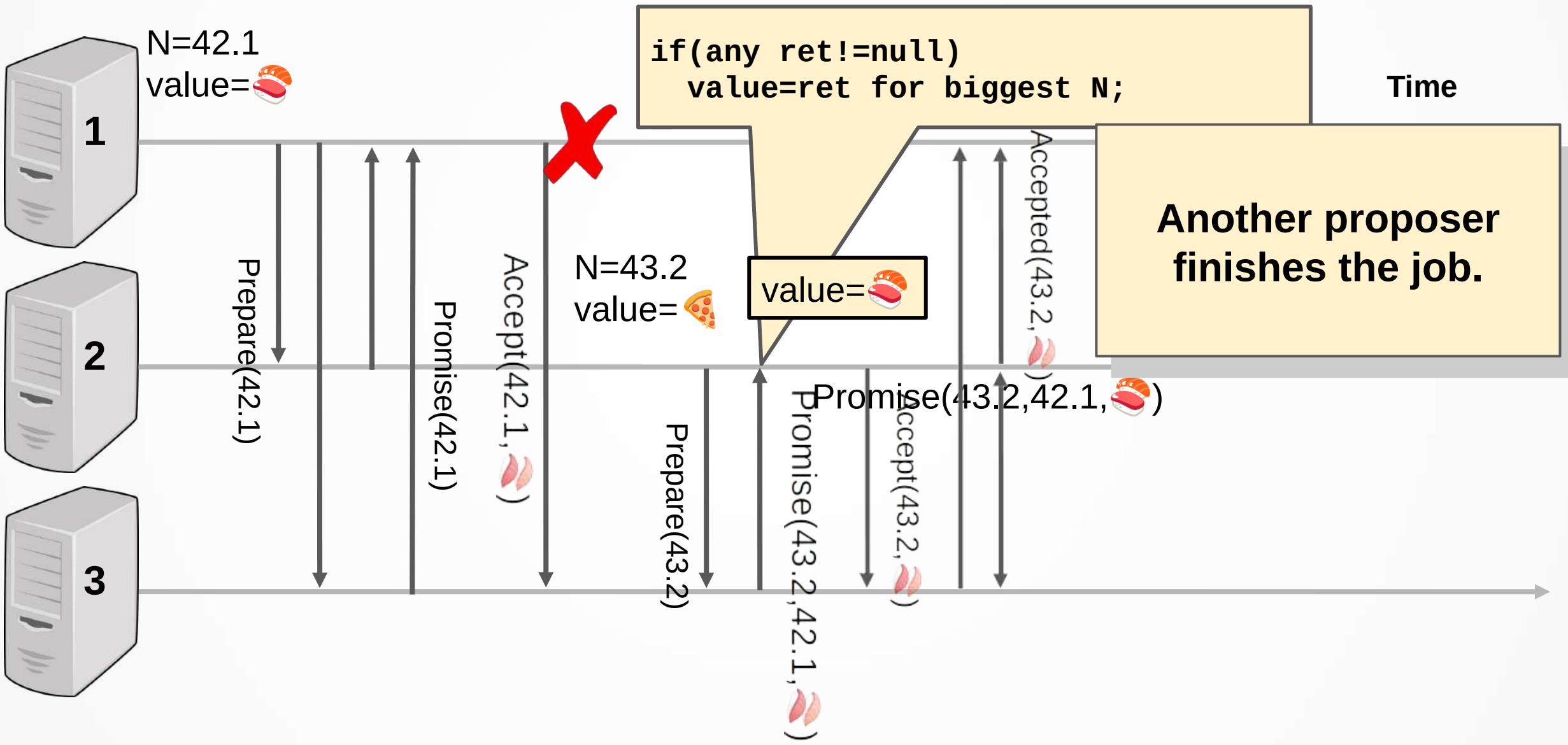
# Failures: Proposer in Prepare Phase



# Failures: Proposer in Accept Phase



# Failures: Proposer in Accept Phase



# What could go wrong?

- If you have just one proposer
  - One or more acceptors fail
    - Still works as long as the majority is up
  - A proposer fails in the “prepare” phase
    - Nothing happens, another proposer should eventually show up and start the algorithm
  - A proposer fails in the “accept” phase
    - Either another proposer overwrites the decision
    - Or it gets notified of what’s been done until now, and finishes the job
- Two or more proposers
  - The algorithm will never result in a wrong output
  - But (in rare cases) it can result in a *livelock*--an infinite loop of messages

# In the Real World

- Leader election
  - Only have a proposer at a time
  - You can do it with Paxos itself!
- Multi-Paxos
  - Build a full log of decisions
  - Additional complexity
  - Faster: one round-trip per transaction
- Cluster management
- Developing, testing and debugging is hard!

# Raft

# Credits Again

- Slides by Ben B. Johnson, CC BY 4.0 license
  - With very minor edits
- See also the original paper by Ongaro and Ousterhout

# Conclusions

# The Cost of Consensus

- Very difficult to implement and use correctly
  - Luckily, there are libraries!
- All machines responsible for consensus risk being a bottleneck
  - Risk of going at the speed of the slowest
- In common cases, a transaction takes a network round-trip to complete
  - If machines are all close, it's quick **but** what about correlated failures?
  - Trade-off between reliability and latency!

# Raft

*The Understandable  
Distributed Consensus Protocol*

@benbjohnson

*What is  
Distributed Consensus?*

**Distributed = Many nodes**

**Consensus = Agreement**

Distributed = Many nodes

**Consensus = Agreement**



# Data Replication



# Leader Election



# Distributed Locks

# A Really Short History Of Distributed Consensus Protocols

# A Really Short History Of Distributed Consensus Protocols

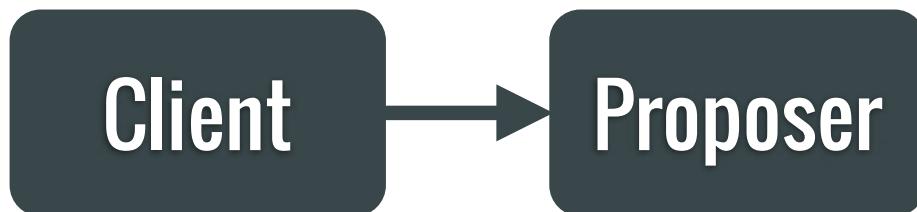
Paxos (1989)

# Paxos In A Nutshell

# Paxos In A Nutshell

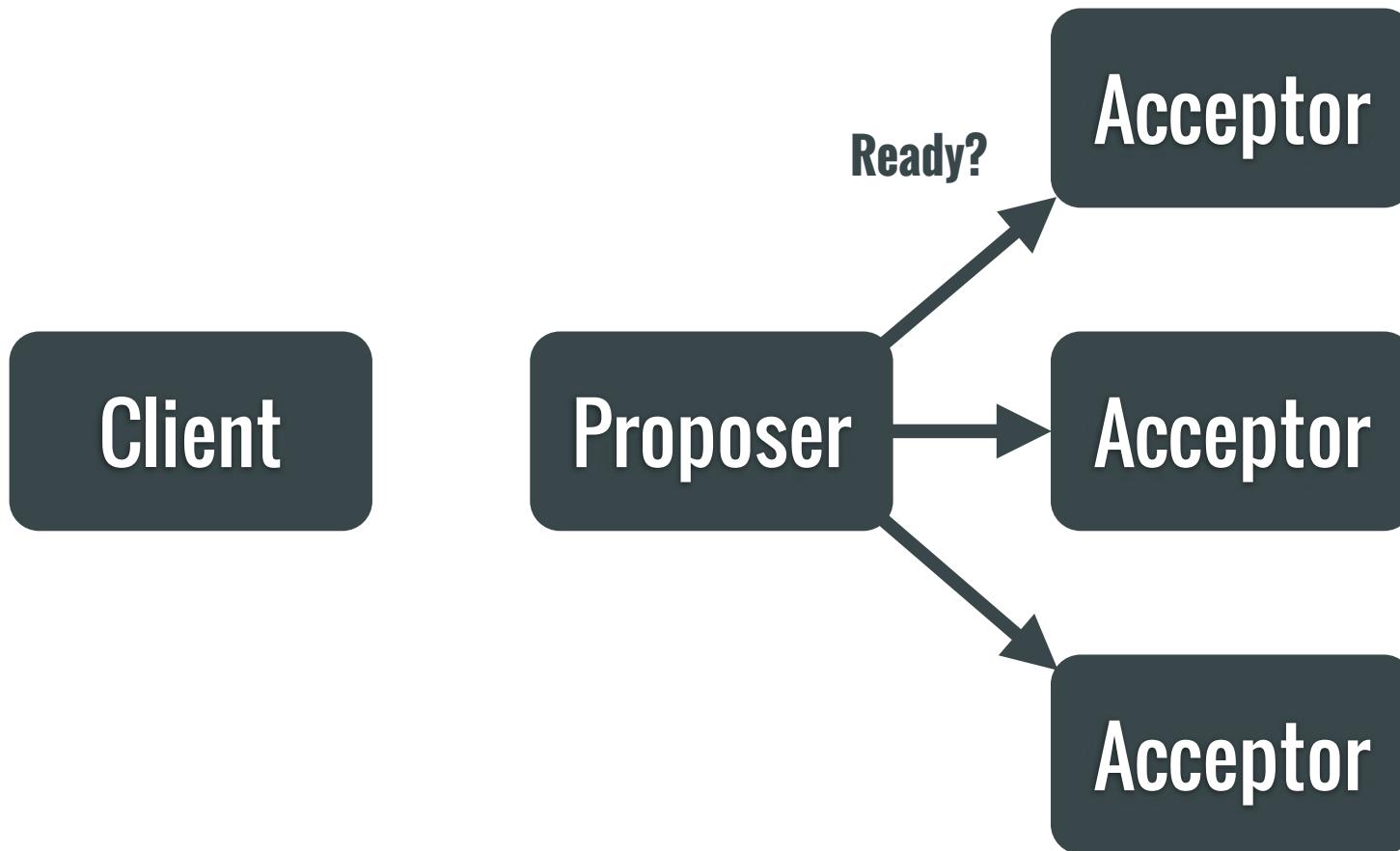
Client

# Paxos In A Nutshell



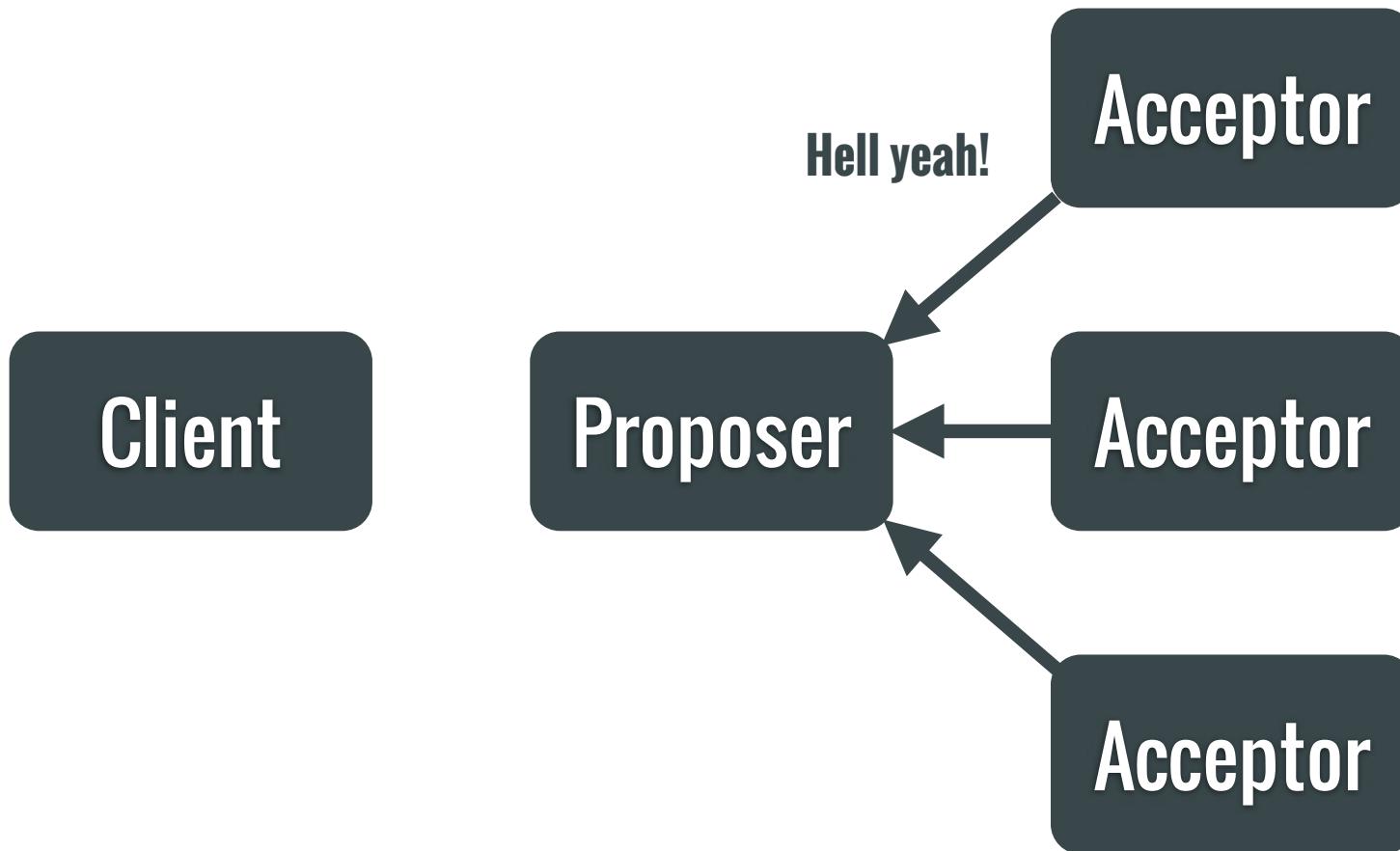
Client requests change to system

# Paxos In A Nutshell



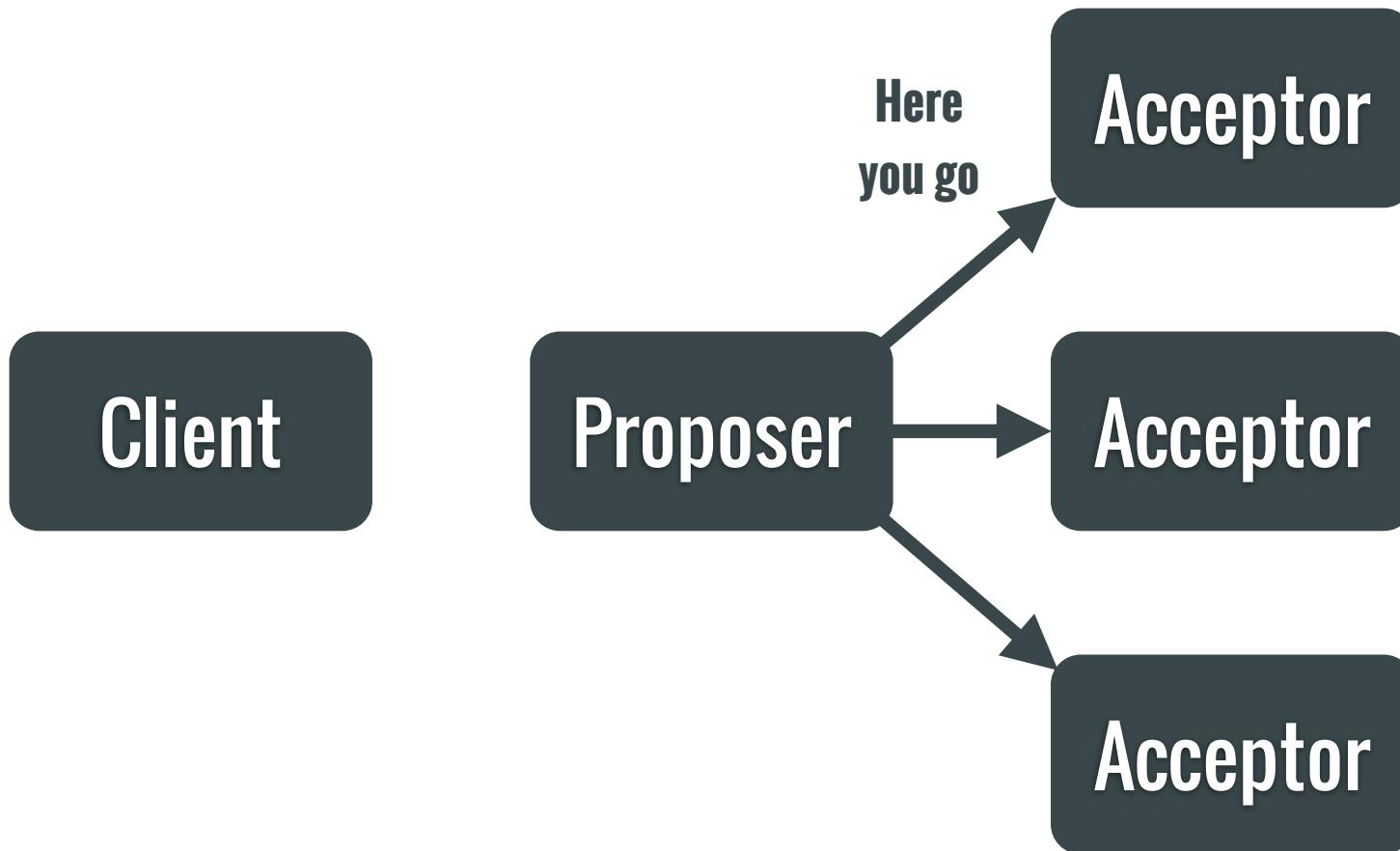
Proposer tells Acceptors to get ready for a change

# Paxos In A Nutshell



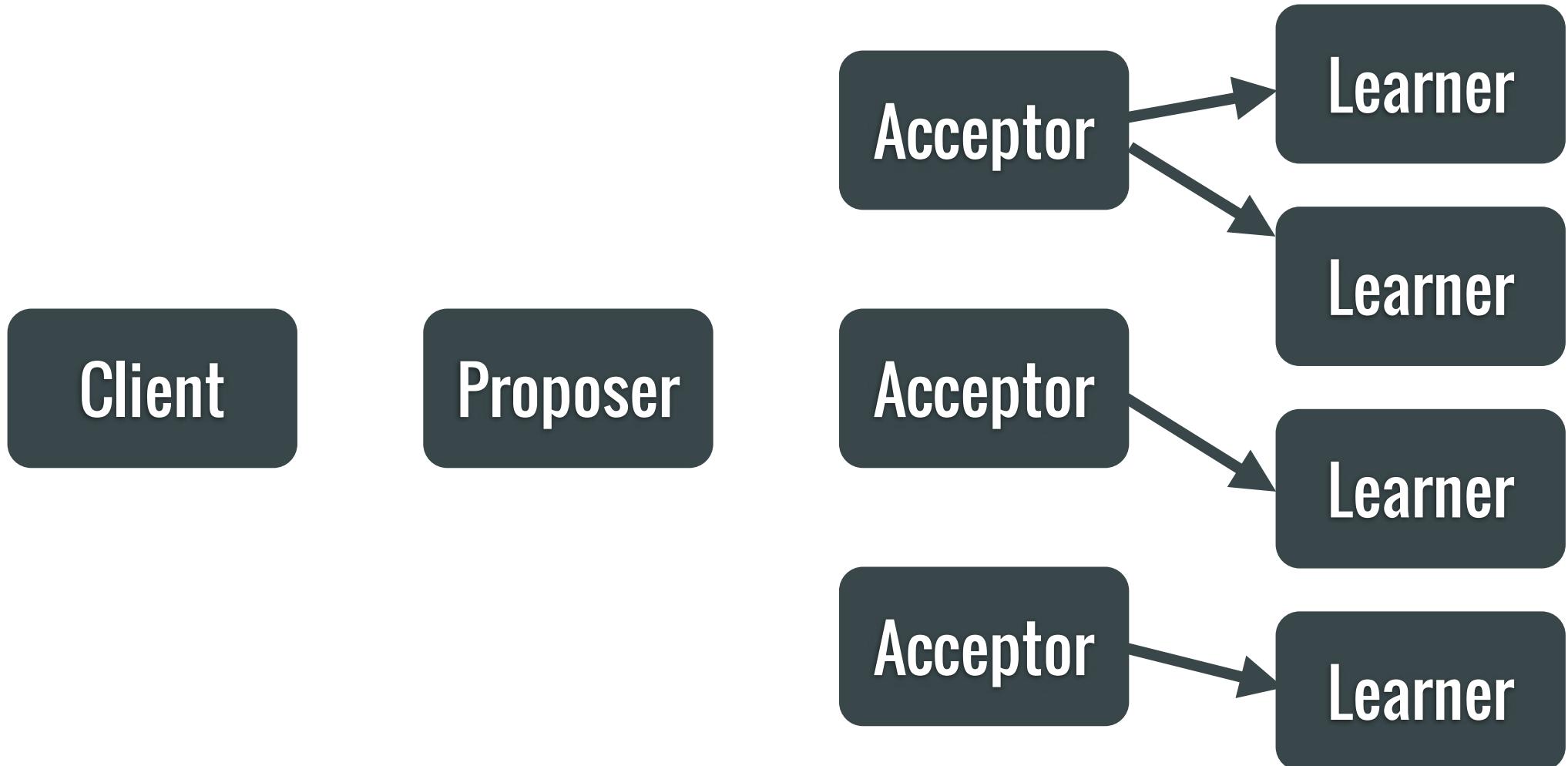
Acceptors confirm to Proposer that they're ready

# Paxos In A Nutshell



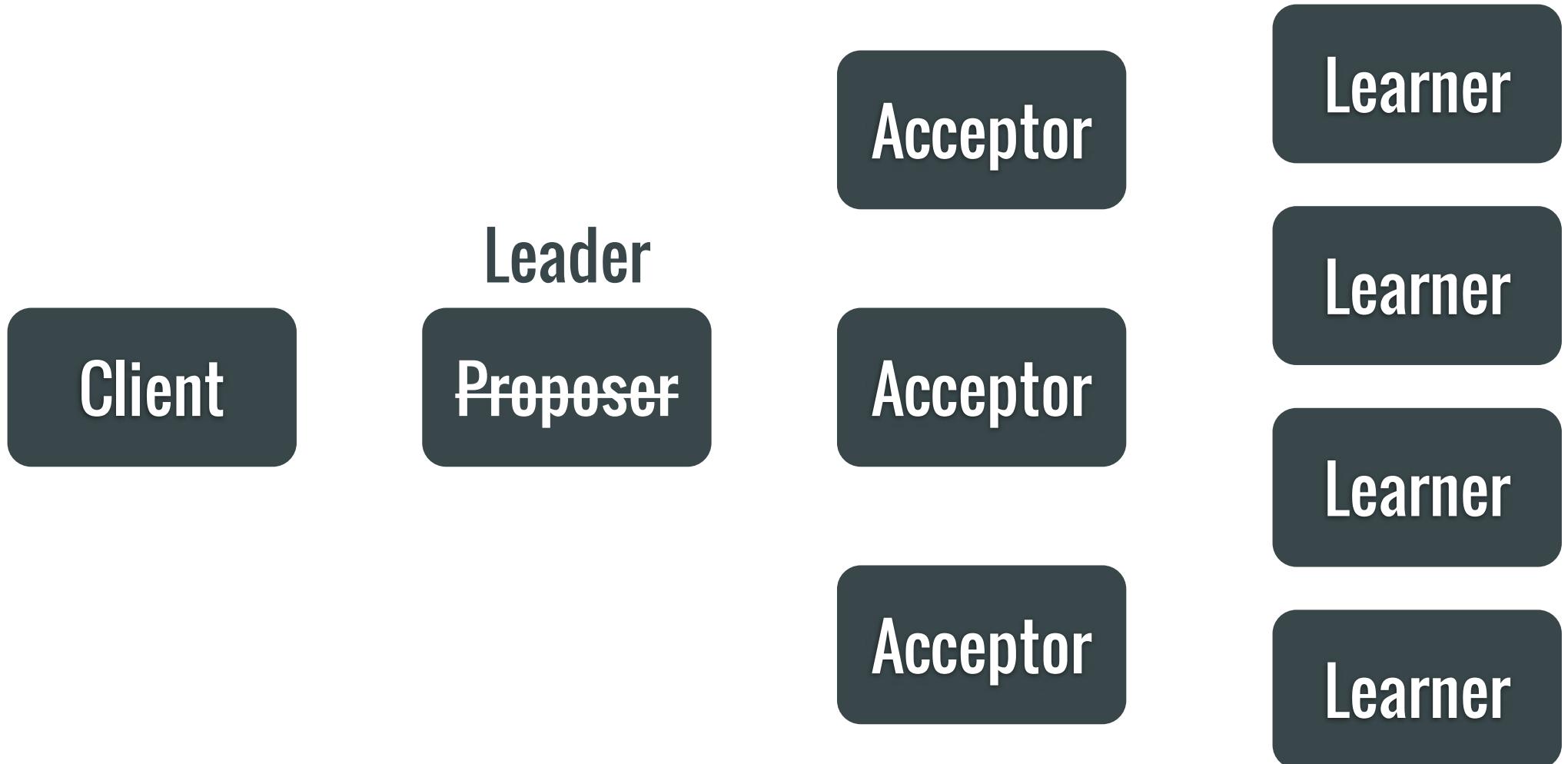
Proposer sends change to Acceptors

# Paxos In A Nutshell



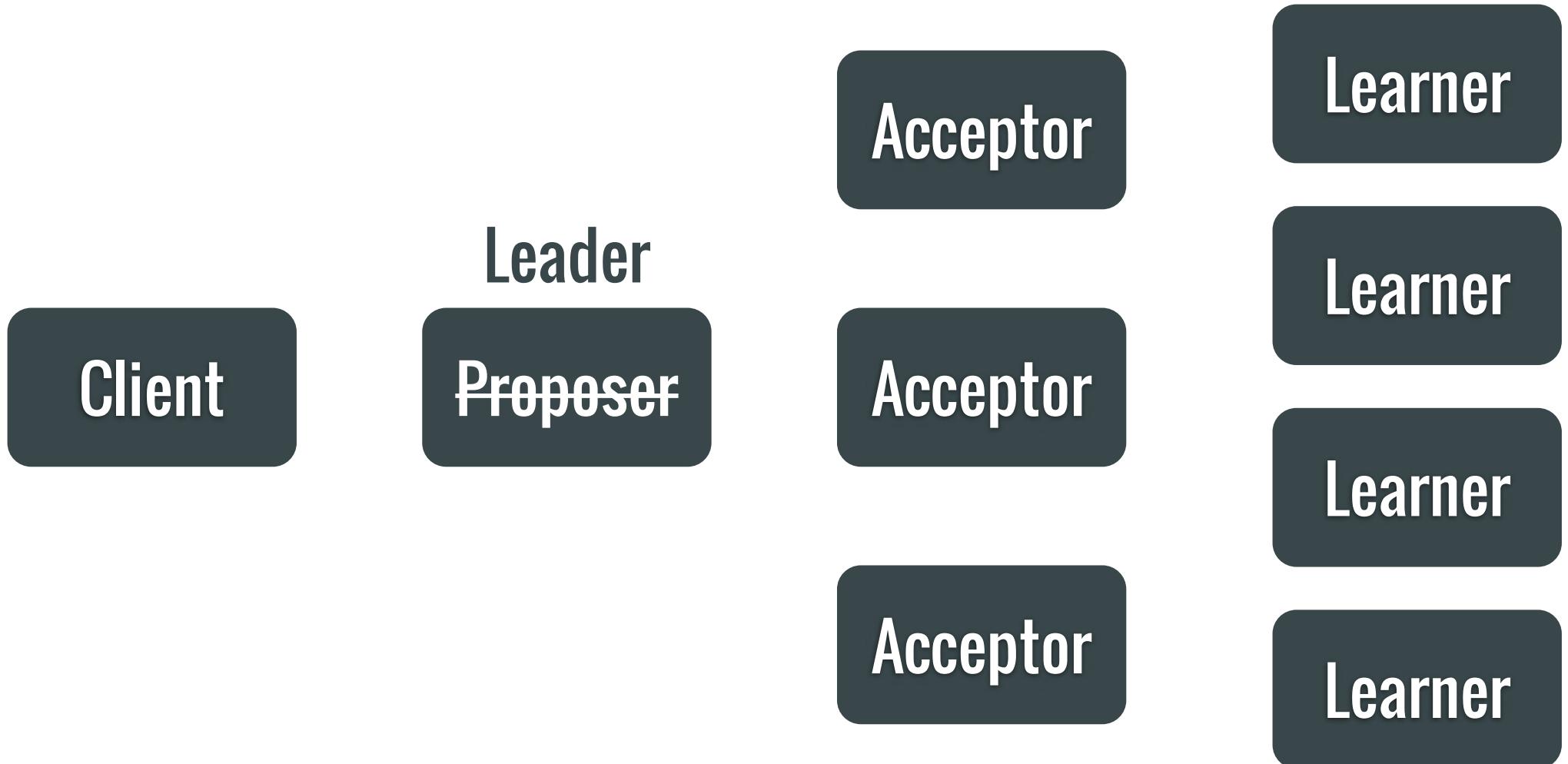
Acceptors propagate change to Learners

# Paxos In A Nutshell



Proposer is now recognized as leader

# Paxos In A Nutshell



Repeat for every new change to the system

# Fun Raft Facts

**Created By:**



# Diego Ongaro

Ph.D. Student  
Stanford University



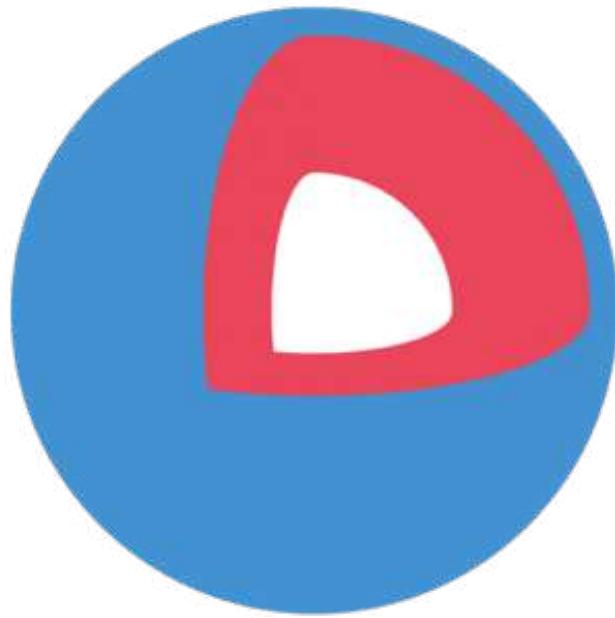
Diego Ongaro  
Ph.D. Student  
Stanford University

John Ousterhout  
Professor of Computer Science  
Stanford University



**28 Implementations  
across various languages**

# In Commercial Use



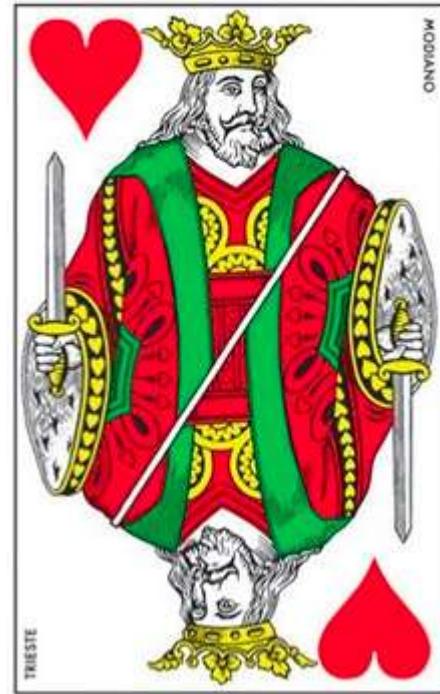
CoreOS  
(etcd)



go-raft

# Raft Basics

# Three Roles:



# The Leader



# The Follower



# The Candidate

# High-Level Example:

F

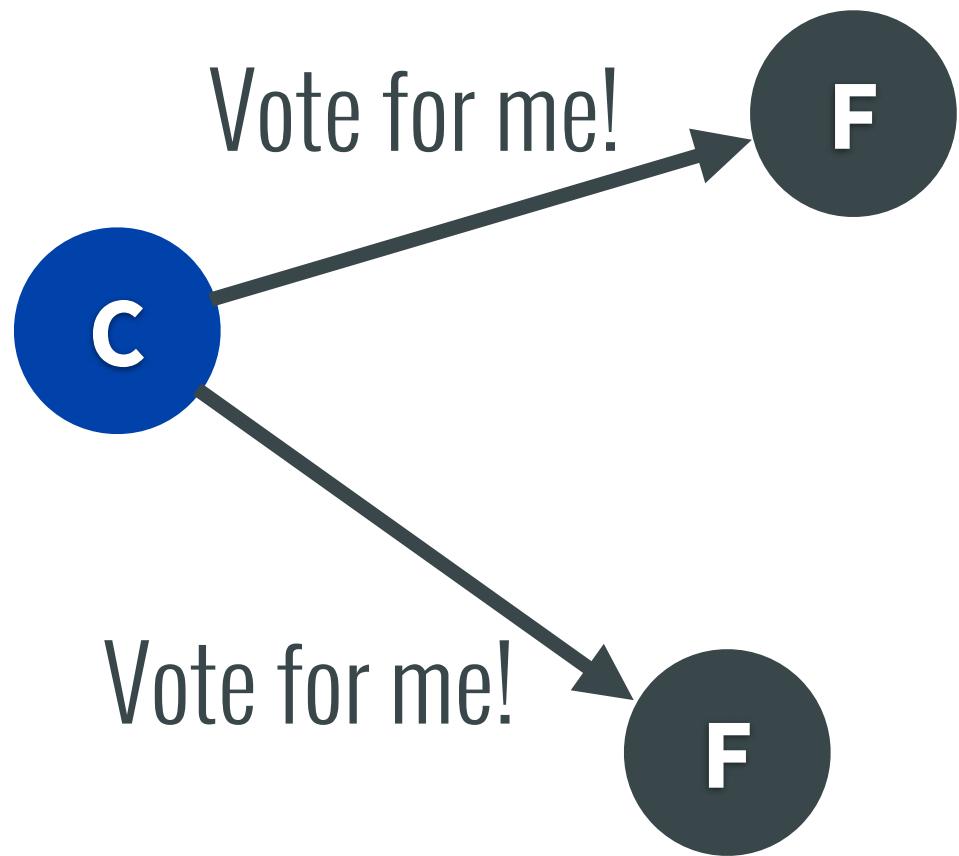
F

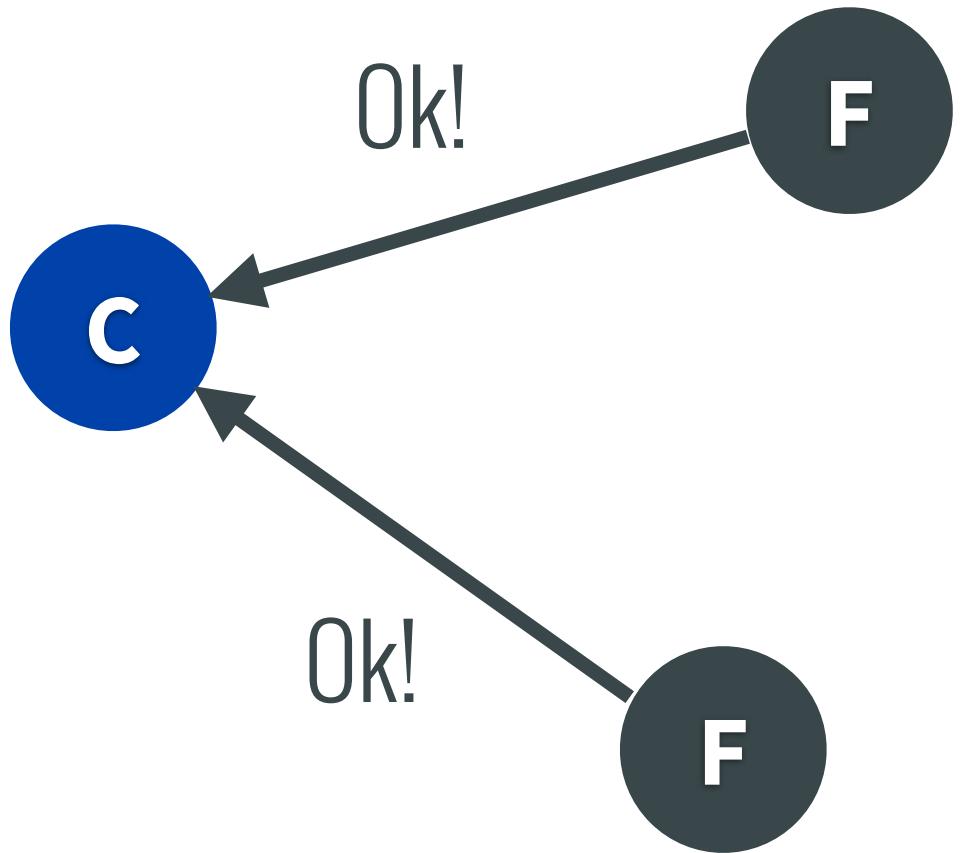
F

C

F

F

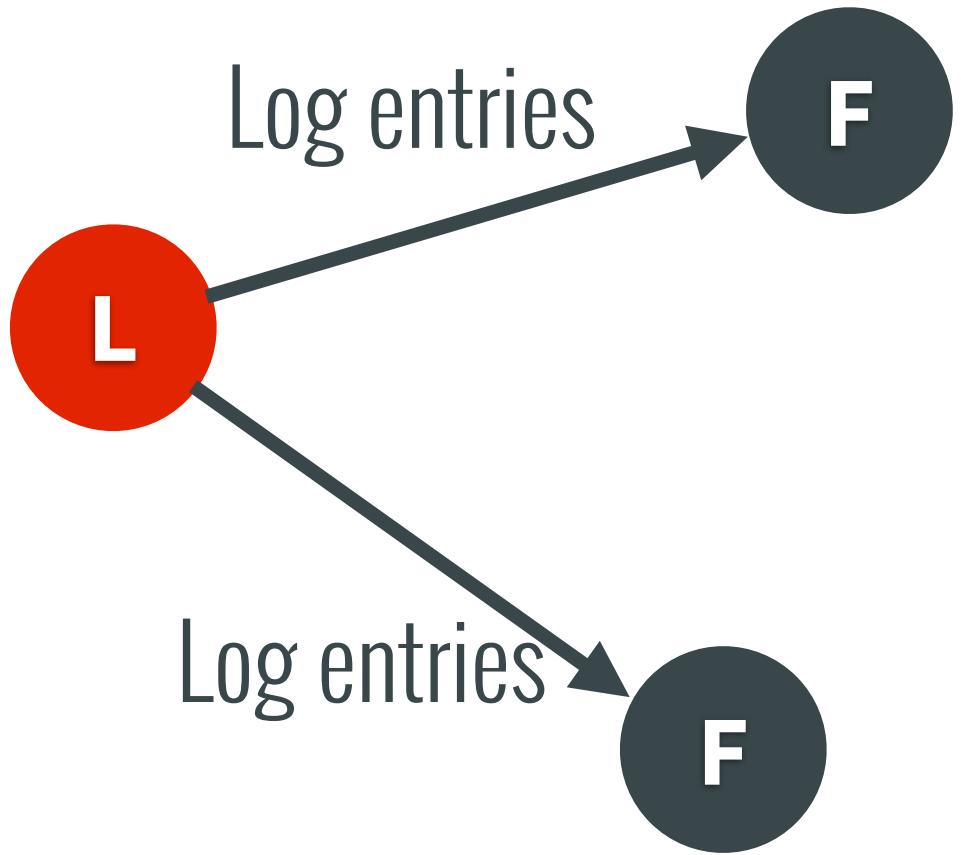


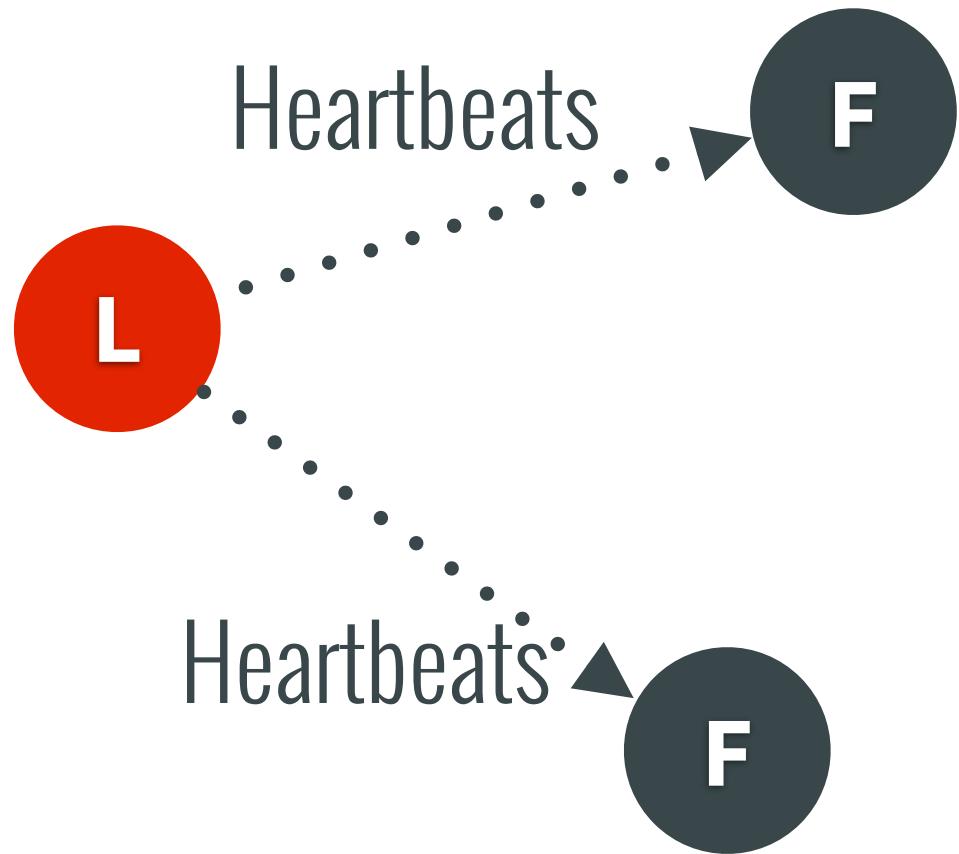


L

F

F





**X**

**F**

**F**



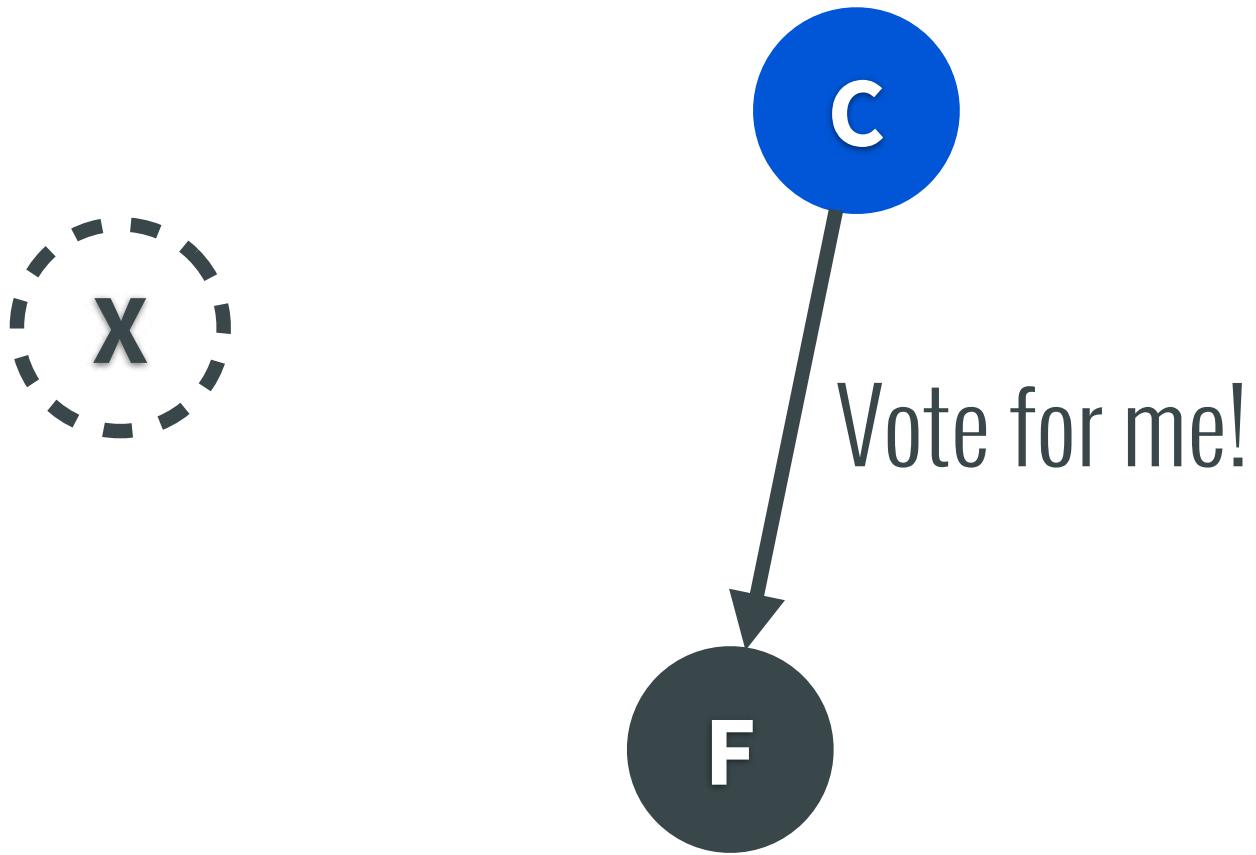
X

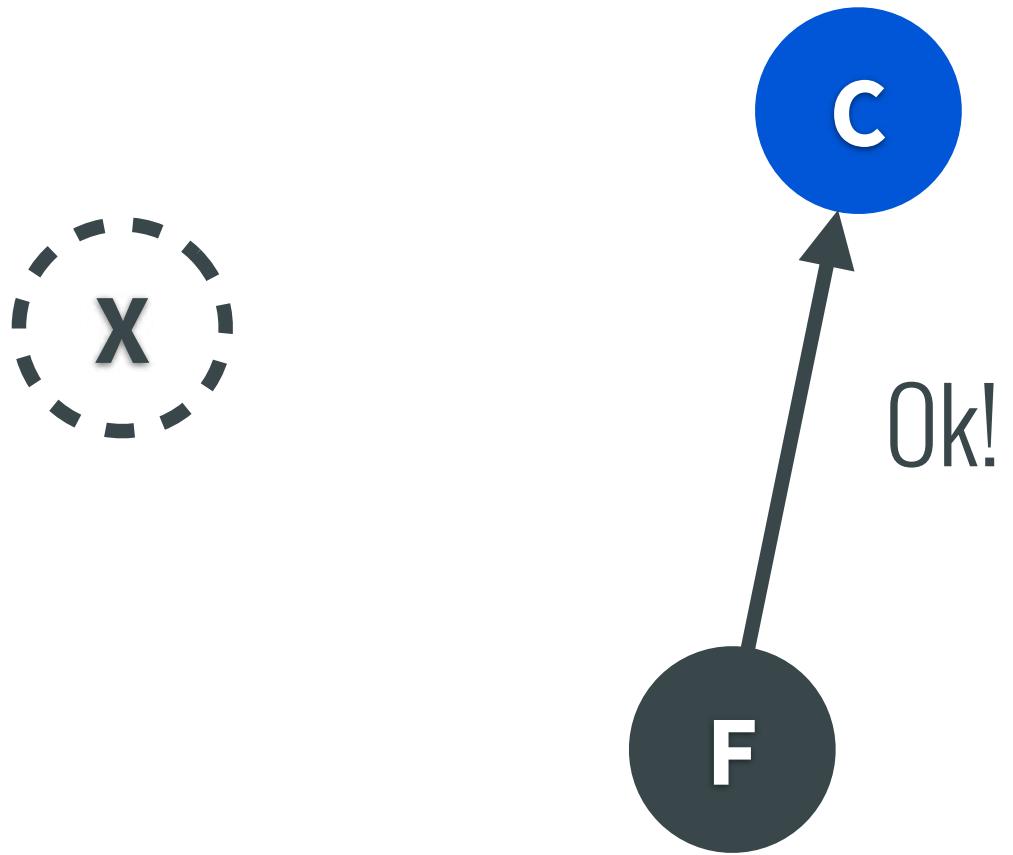


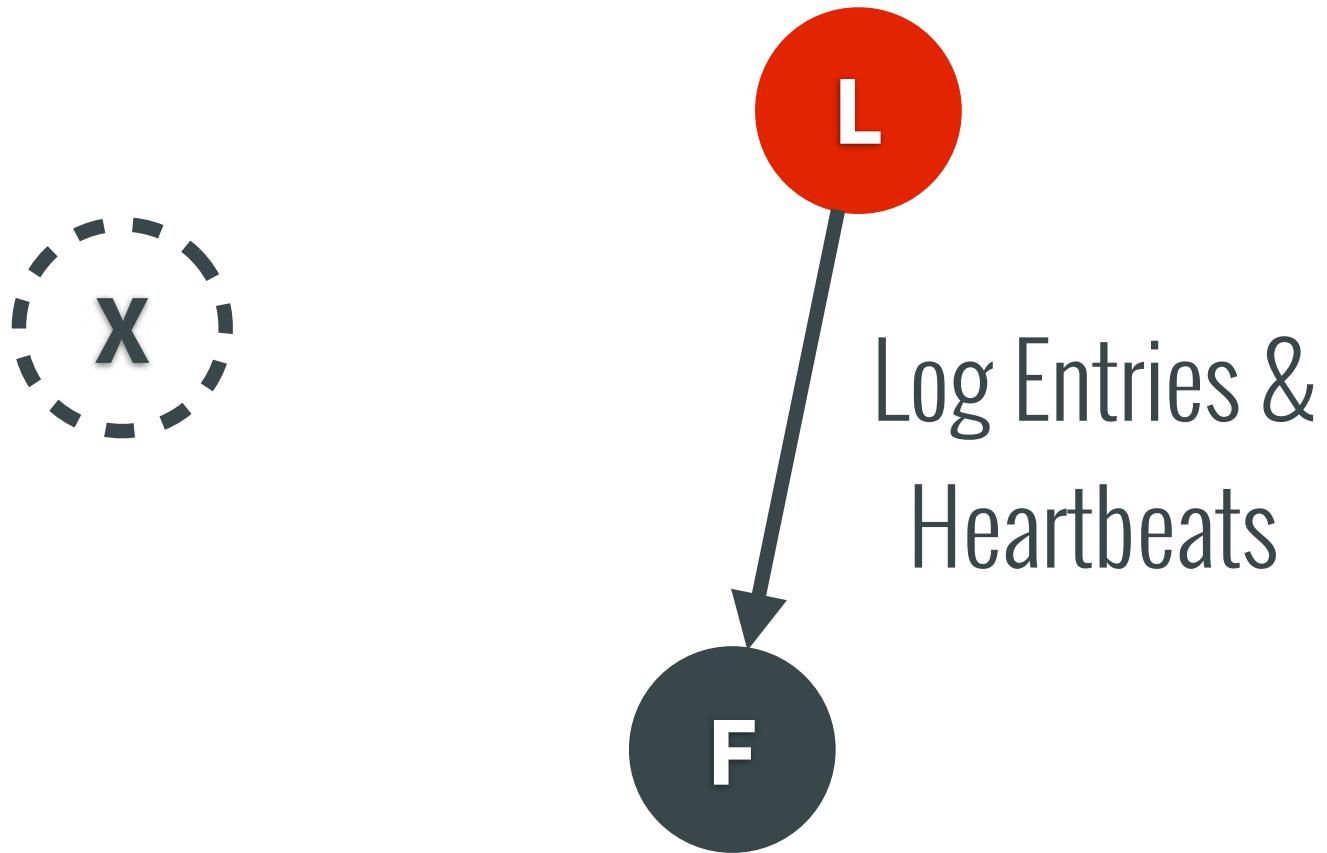
C



F



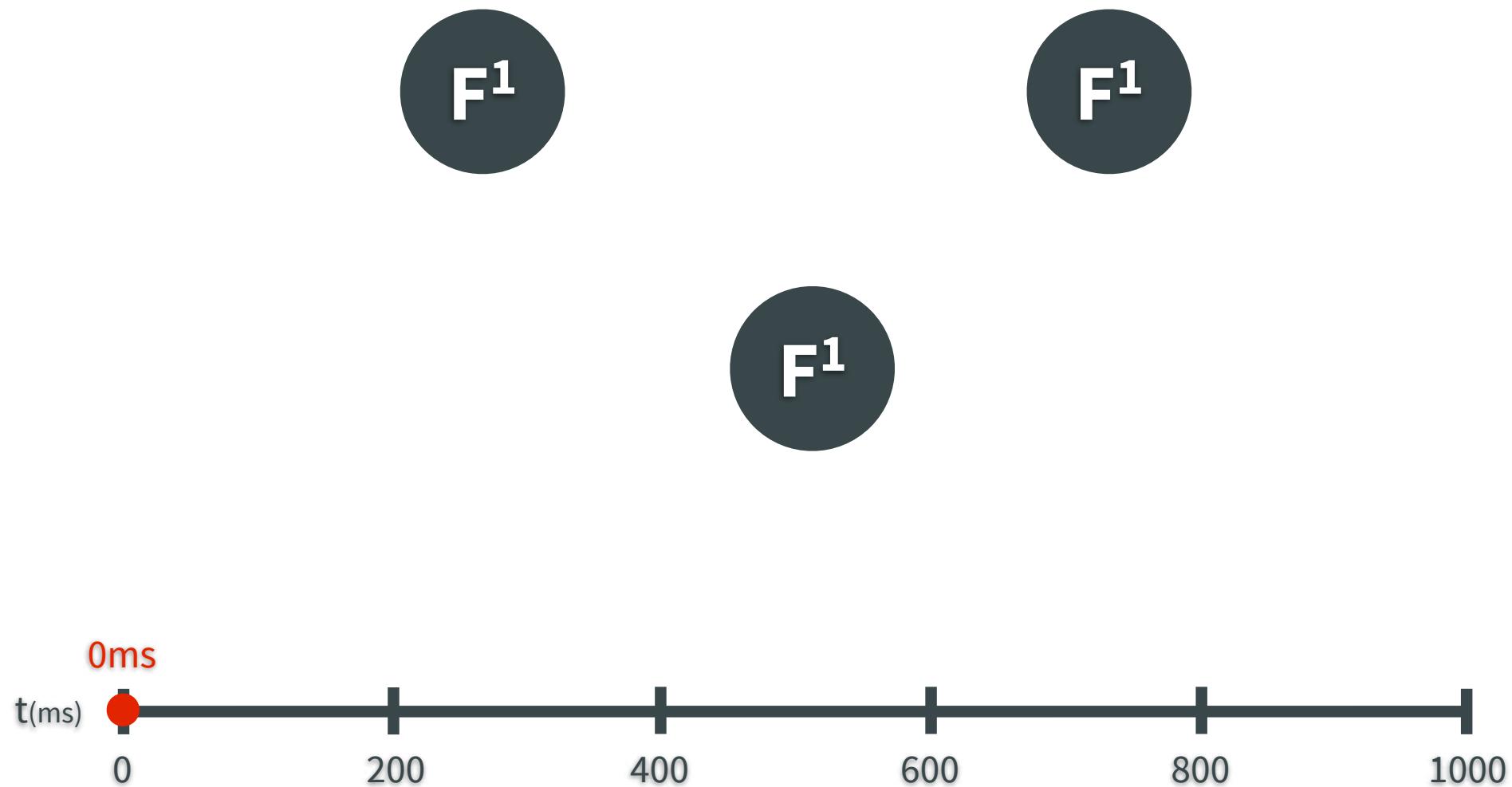




# Leader Election

# Leader Election

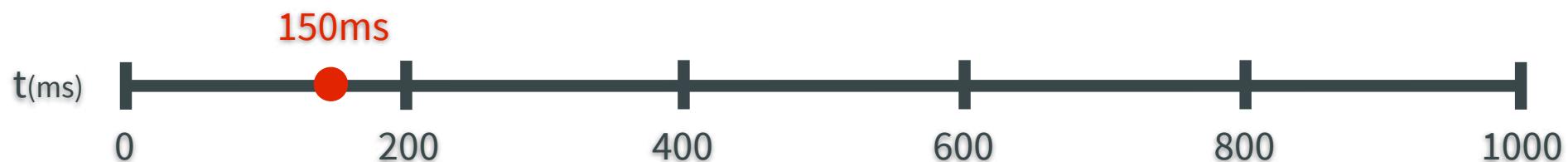
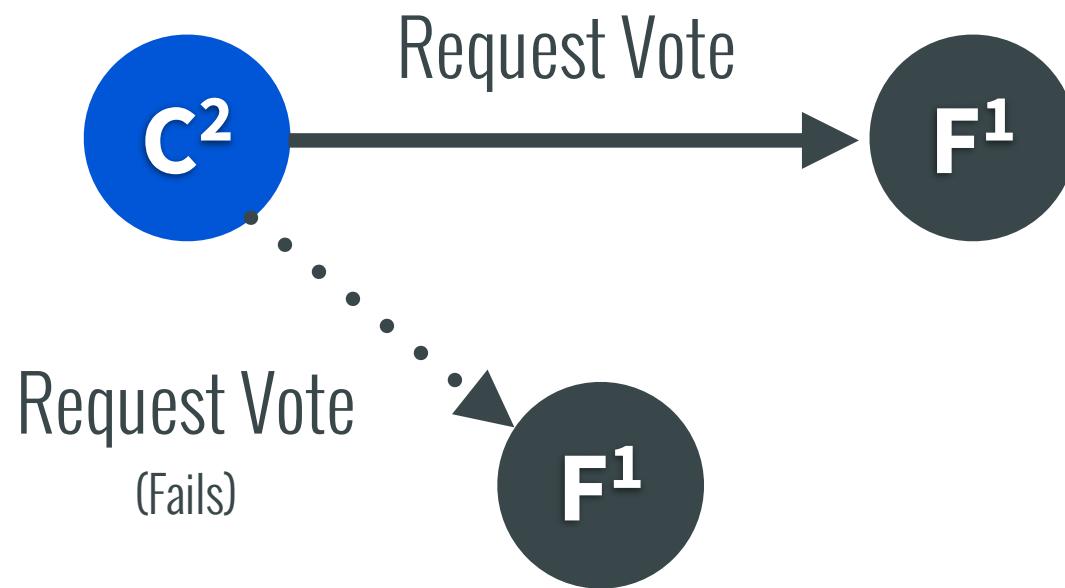
---



# Leader Election

---

One follower becomes a candidate after an election timeout and requests votes



# Leader Election

---

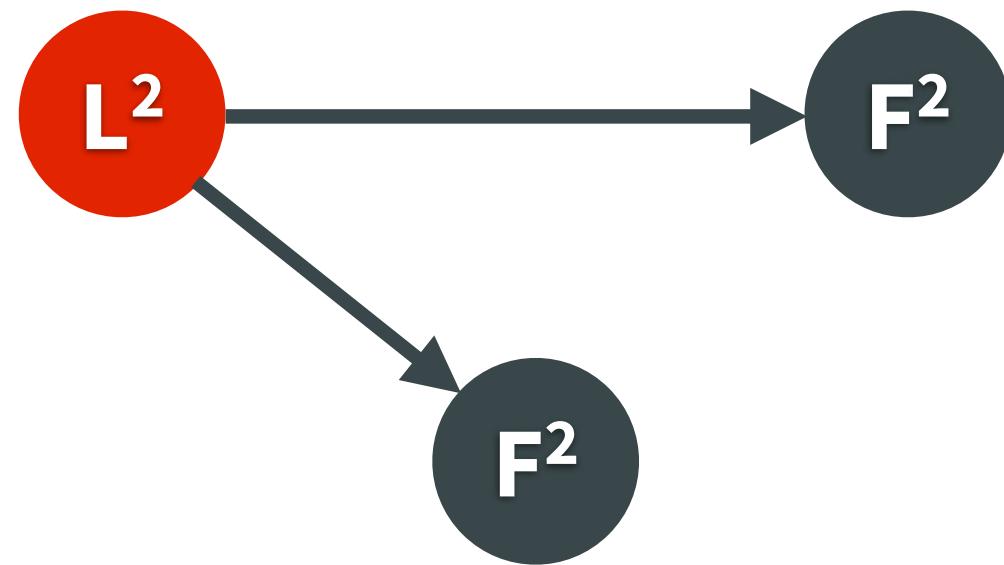
Candidate receives one vote from a peer and one vote from self



# Leader Election

---

Two votes is a majority so candidate becomes leader

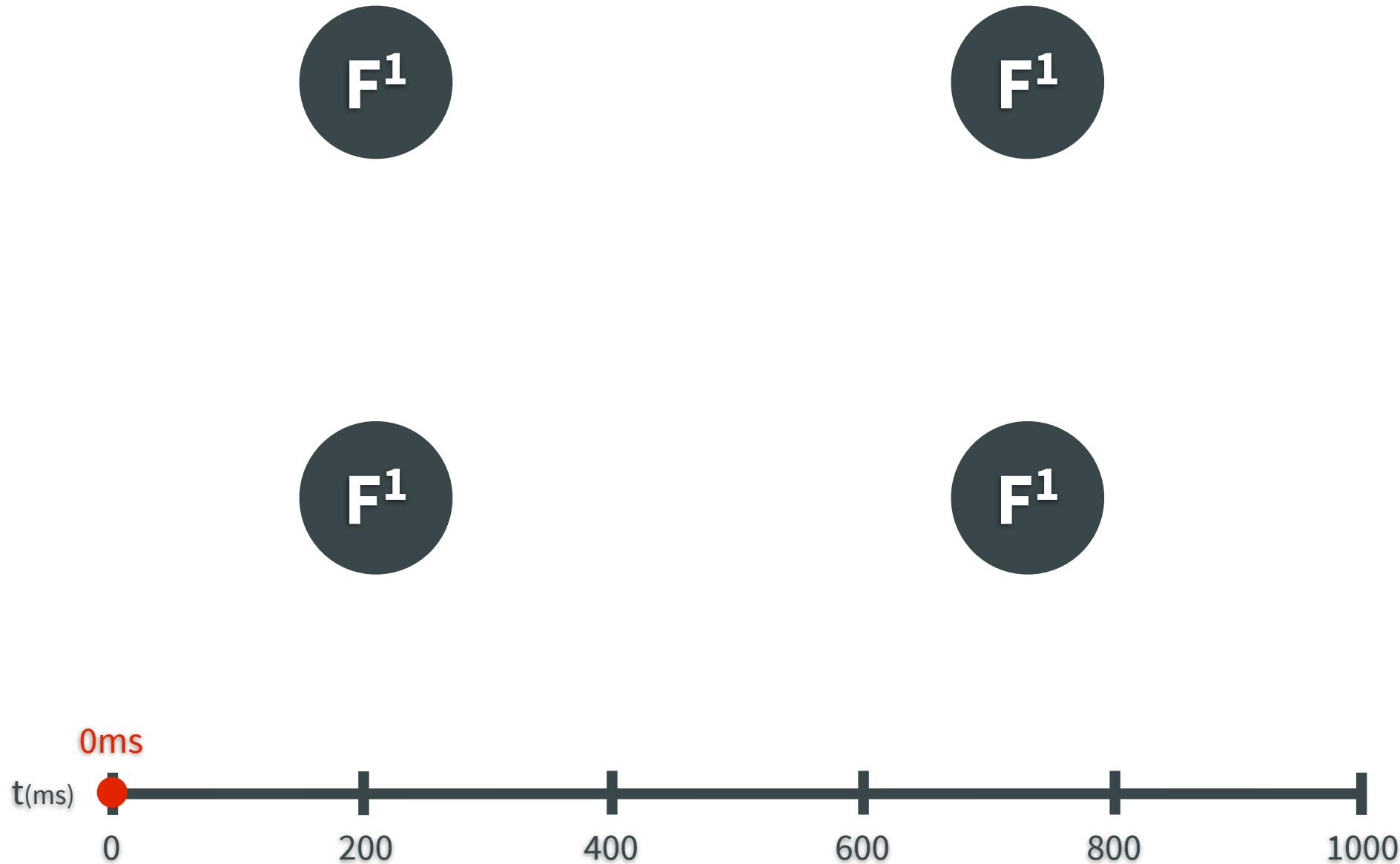


# **Leader Election**

## **(Split Vote)**

# Leader Election

---



# Leader Election

---

Two followers become candidates simultaneously and begin requesting votes



# Leader Election

---

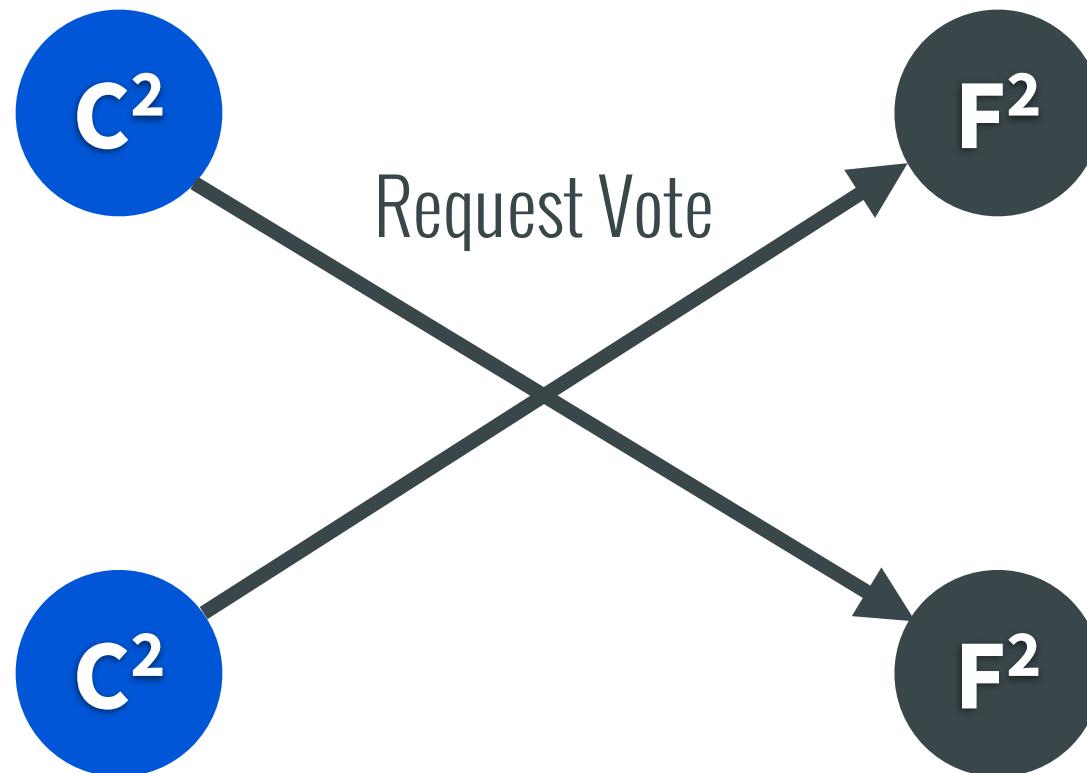
Each candidate receives a vote from themselves and from one peer



# Leader Election

---

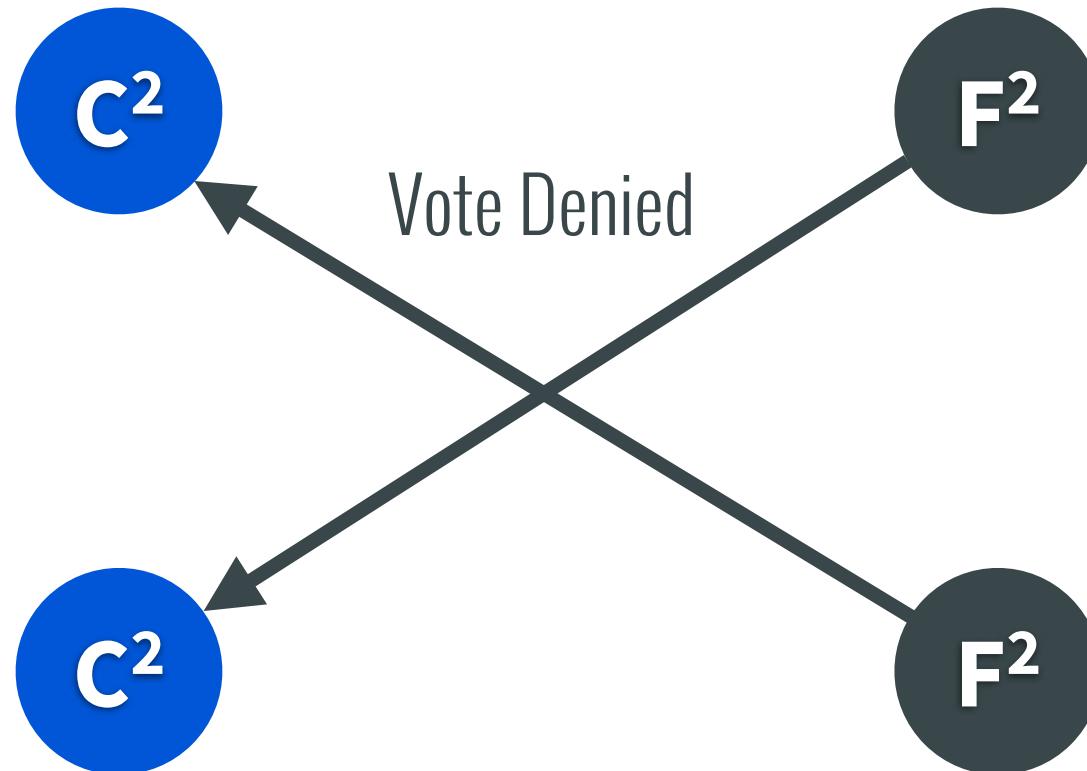
Each candidate requests a vote from a peer who has already voted



# Leader Election

---

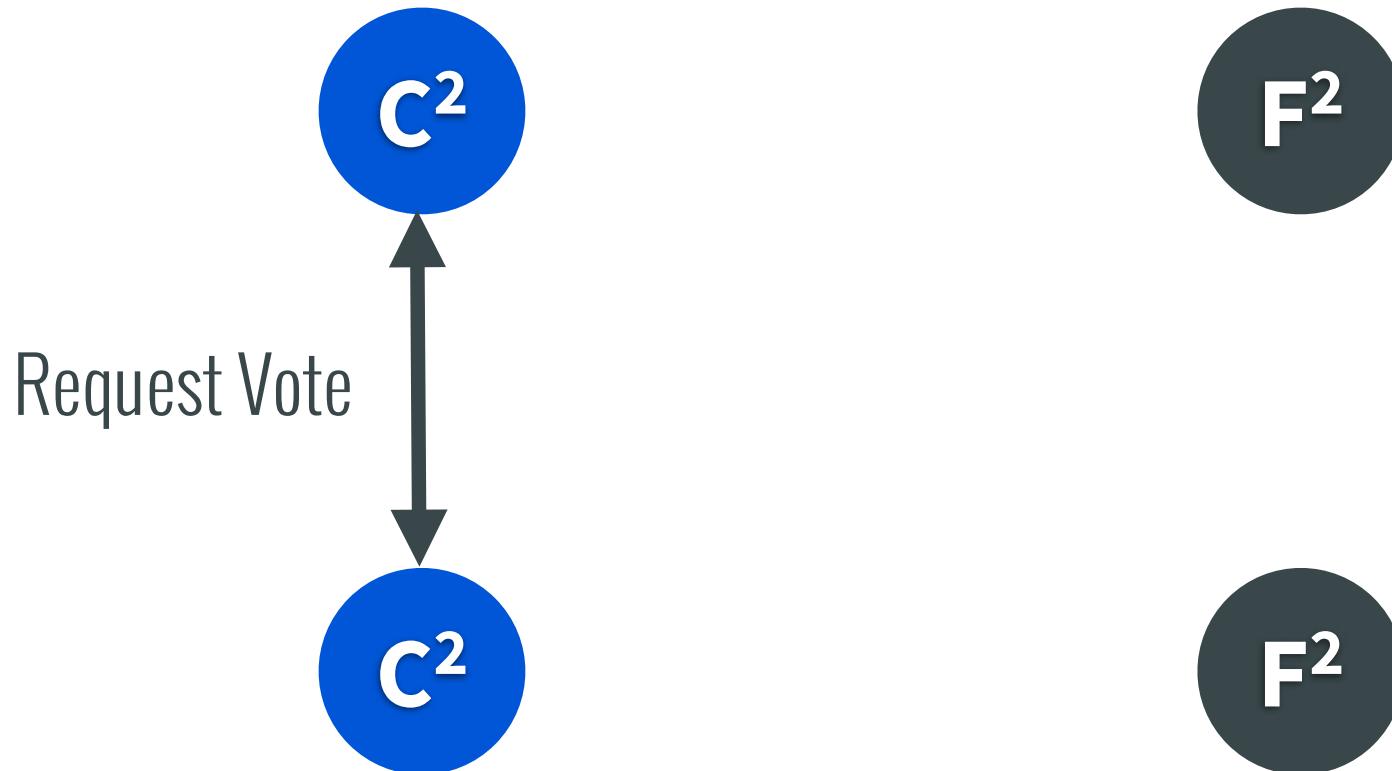
Vote requests are denied because the follower has already voted



# Leader Election

---

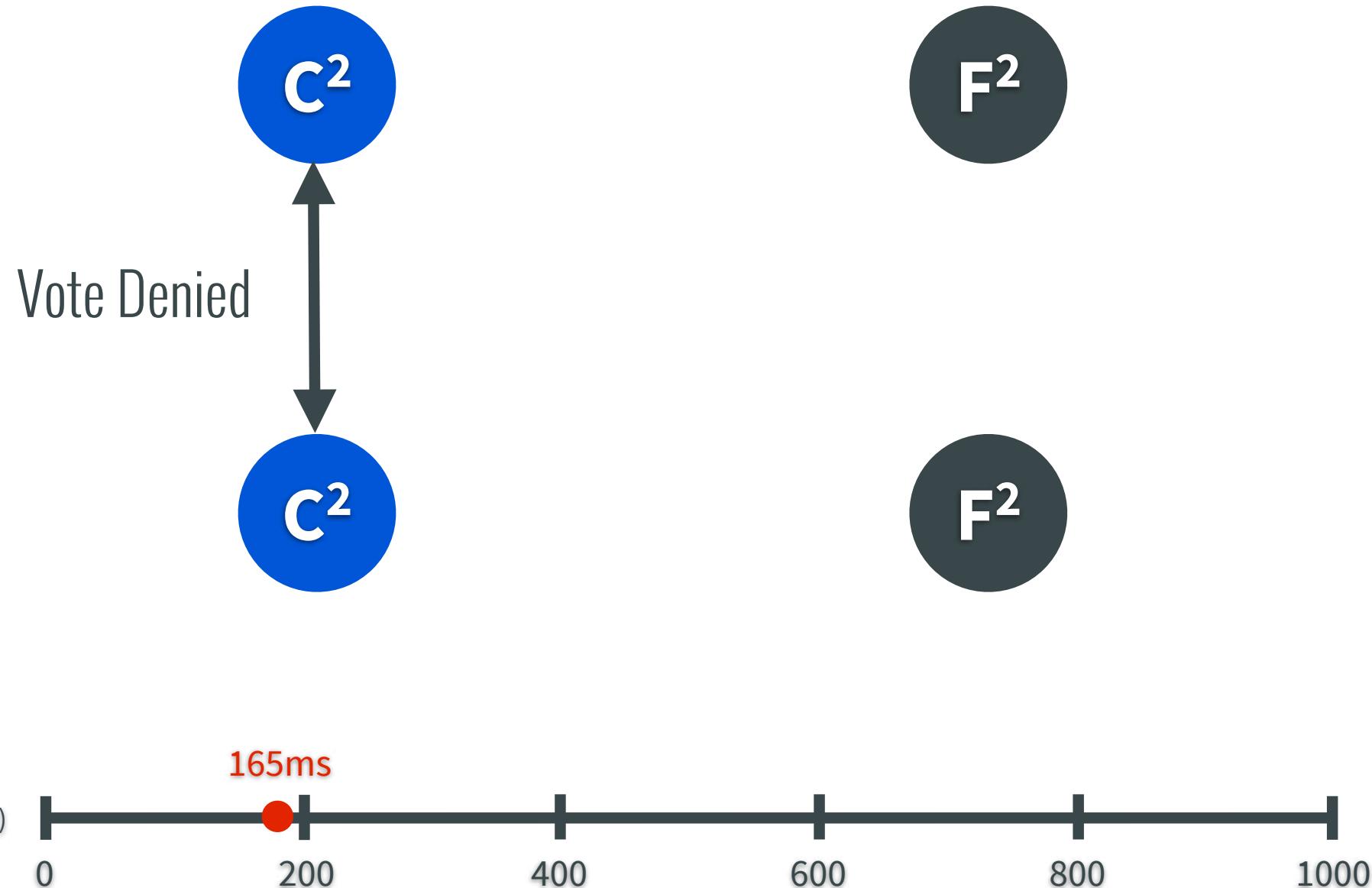
Candidates try to request votes from each other



# Leader Election

---

Vote requests are denied because candidates voted for themselves



# Leader Election

---

Candidates wait for a randomized election timeout to occur (150ms - 300ms)



# Leader Election

---

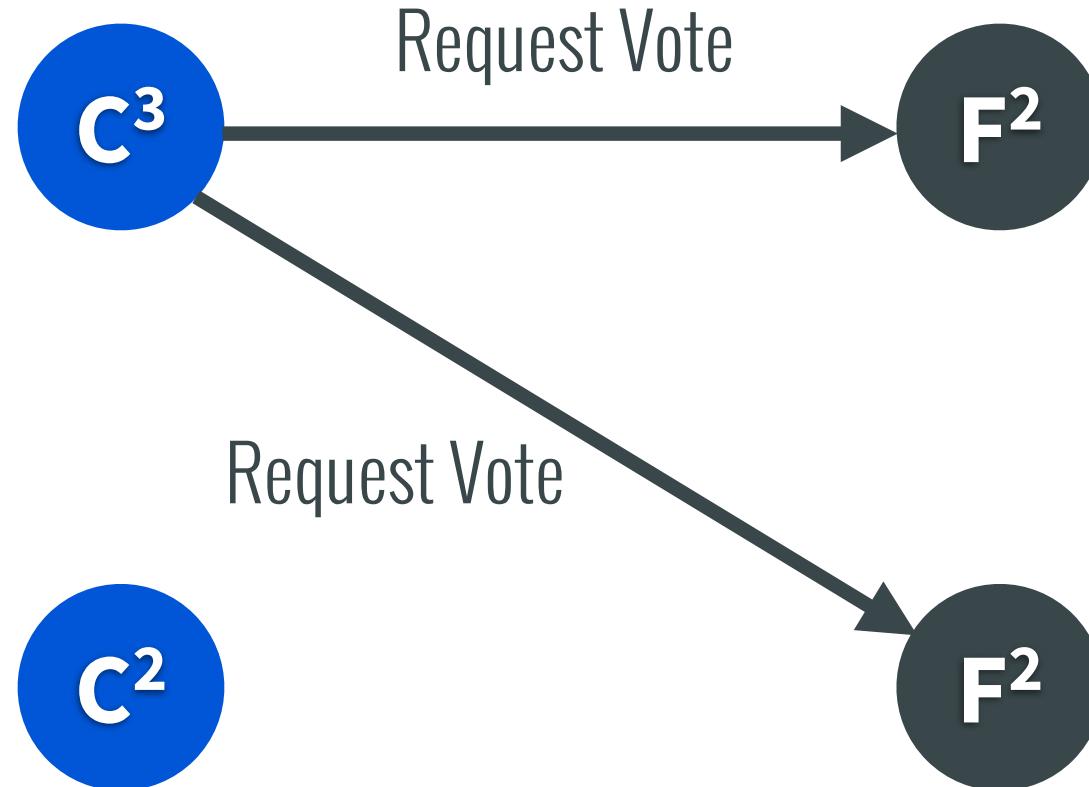
Still waiting...



# Leader Election

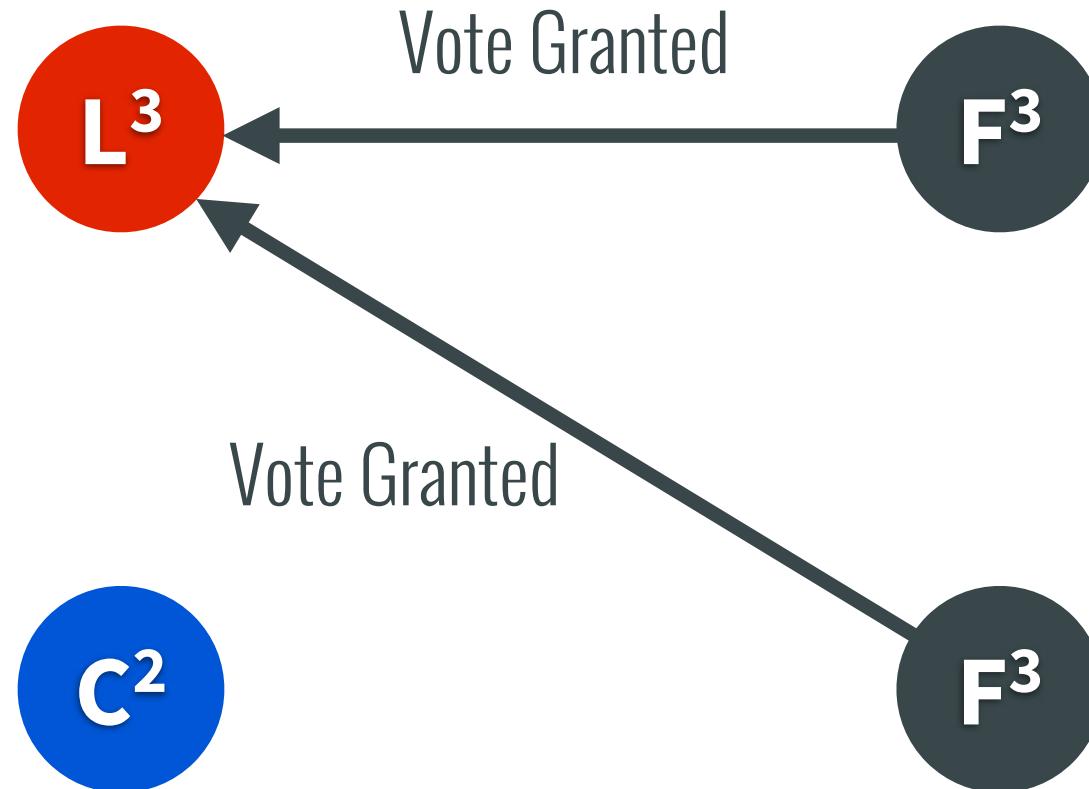
---

One candidate begins election term #3



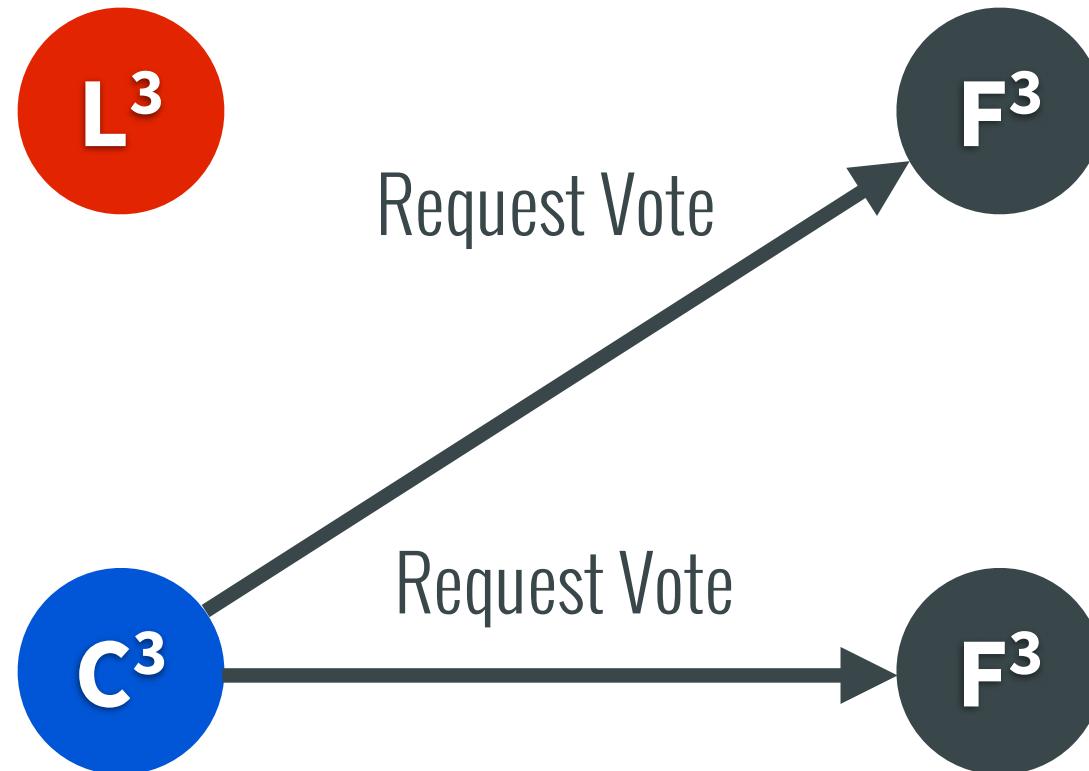
# Leader Election

Candidate receives vote from itself and two peer votes so it becomes leader for election term #3



# Leader Election

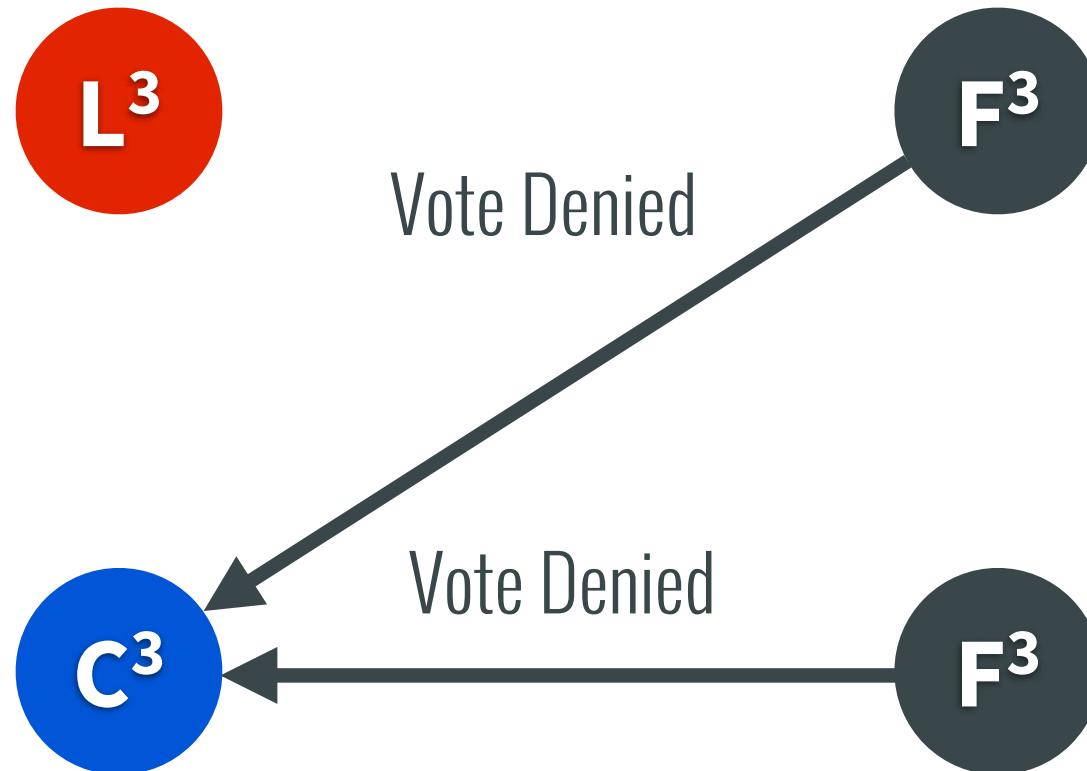
Second candidate doesn't know first candidate won the term and begins requesting votes



# Leader Election

---

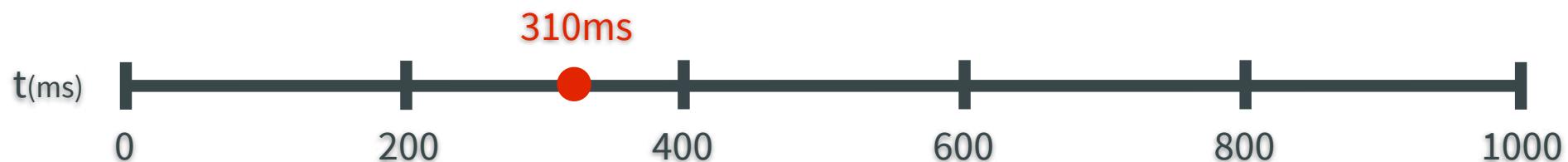
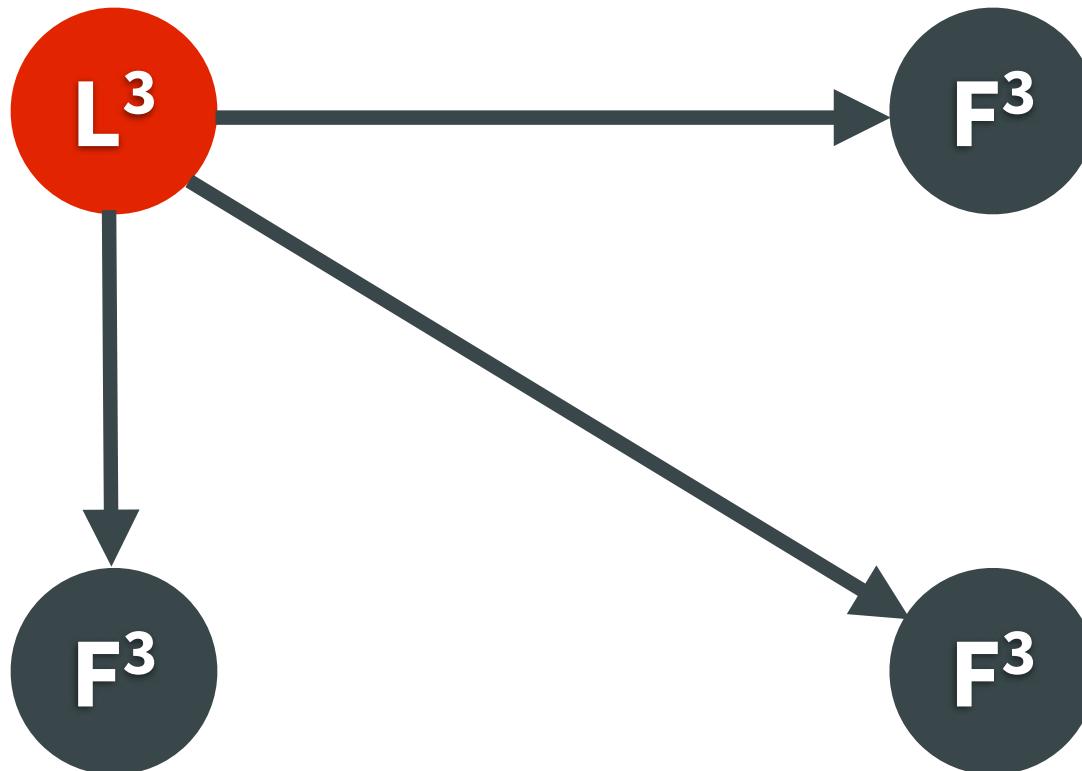
Peers already voted so votes are denied



# Leader Election

---

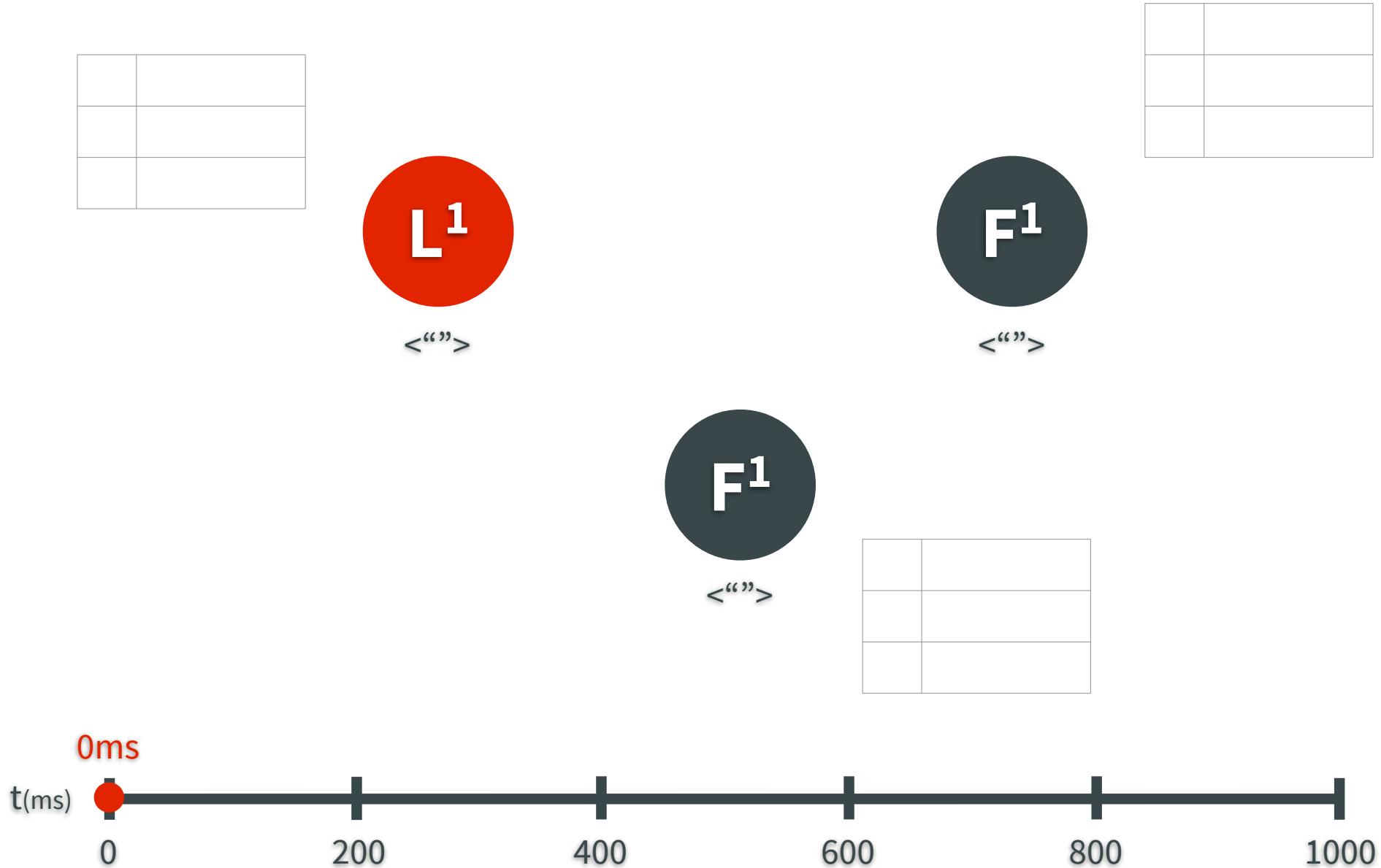
Leader notifies peers of election and other candidate steps down



# Log Replication

# Log Replication

---



# Log Replication

---

A new uncommitted log entry is added to the leader

I	“sally”



<“”>



<“”>

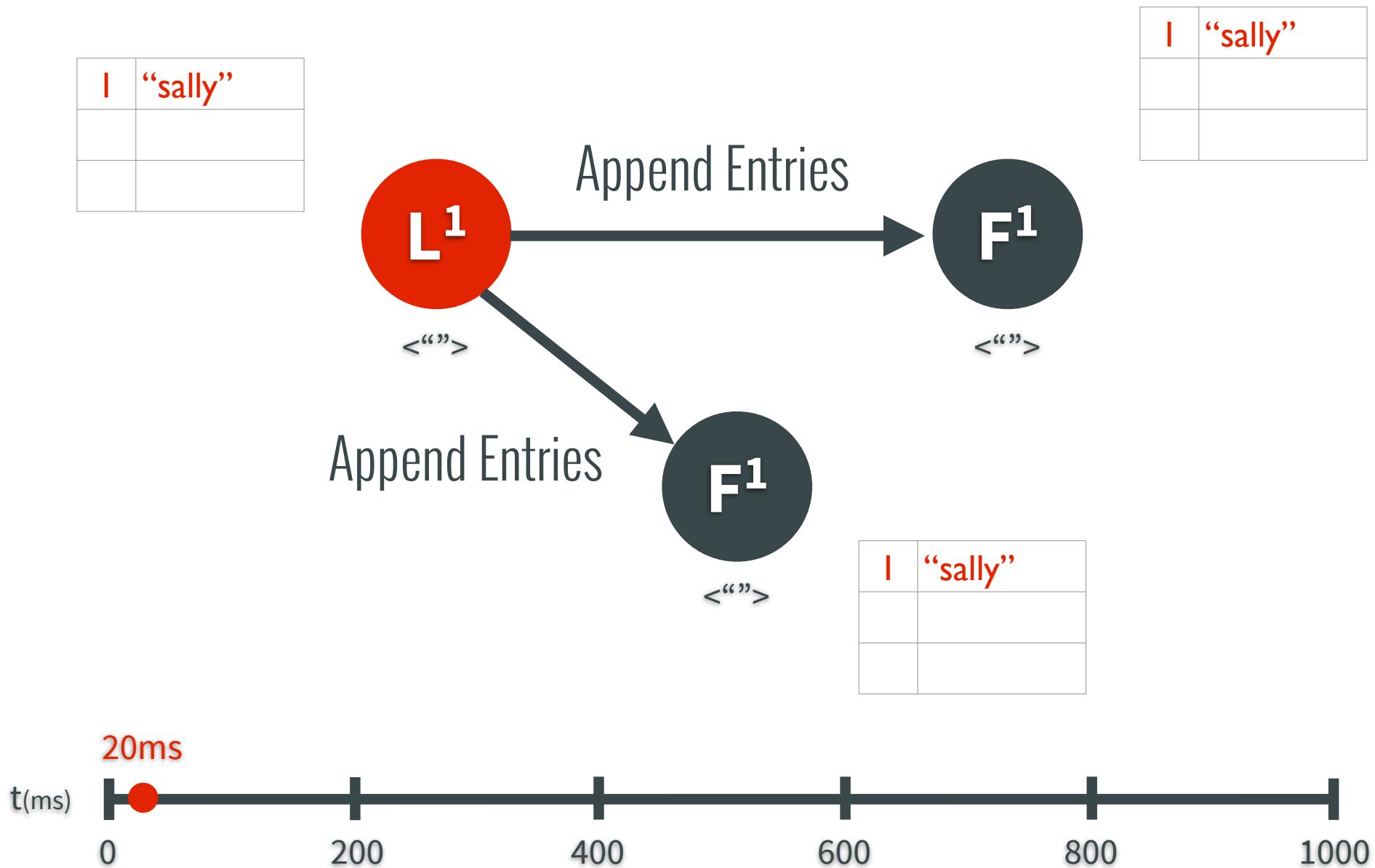


<“”>



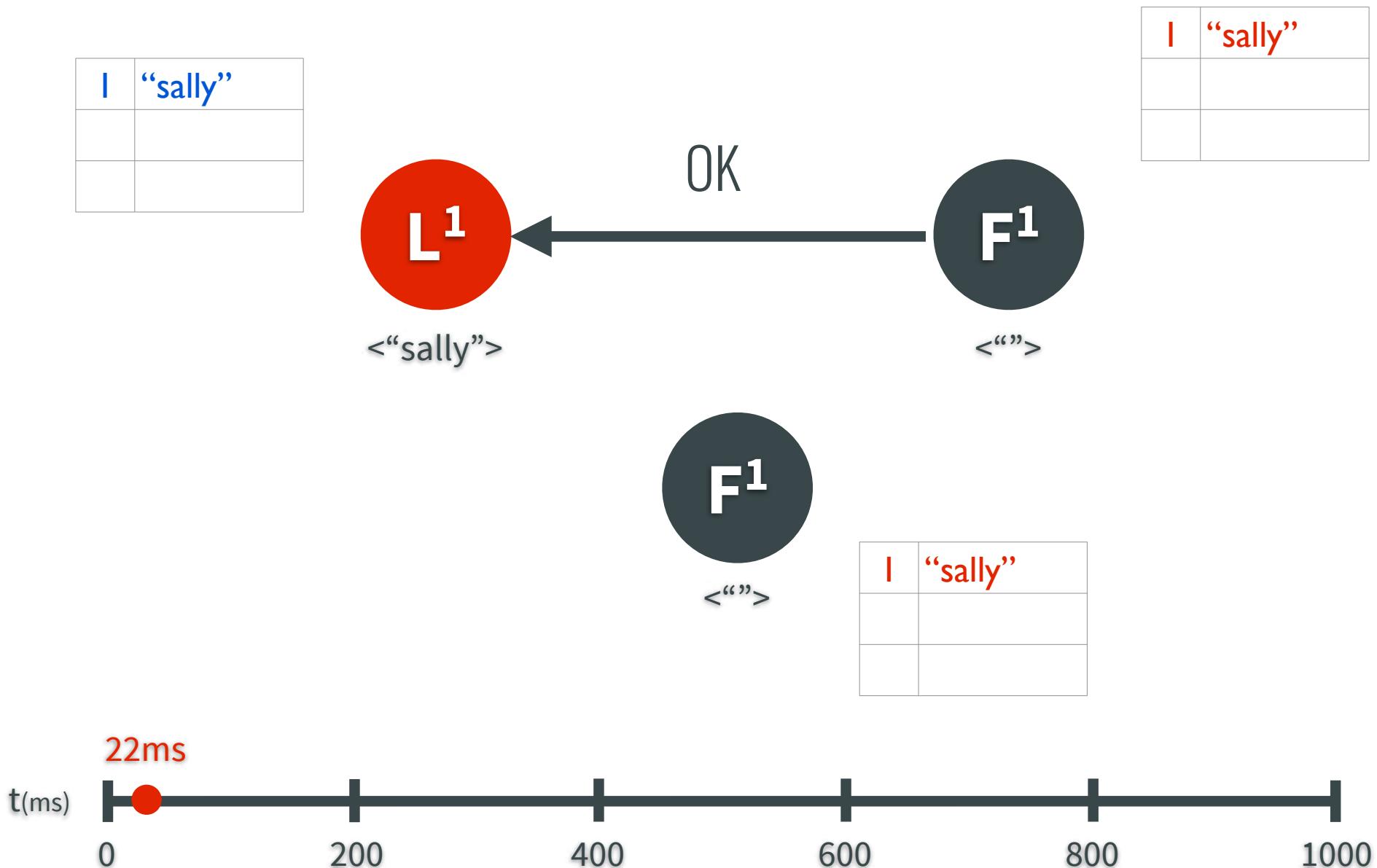

# Log Replication

At the next heartbeat, the log entry is replicated to followers



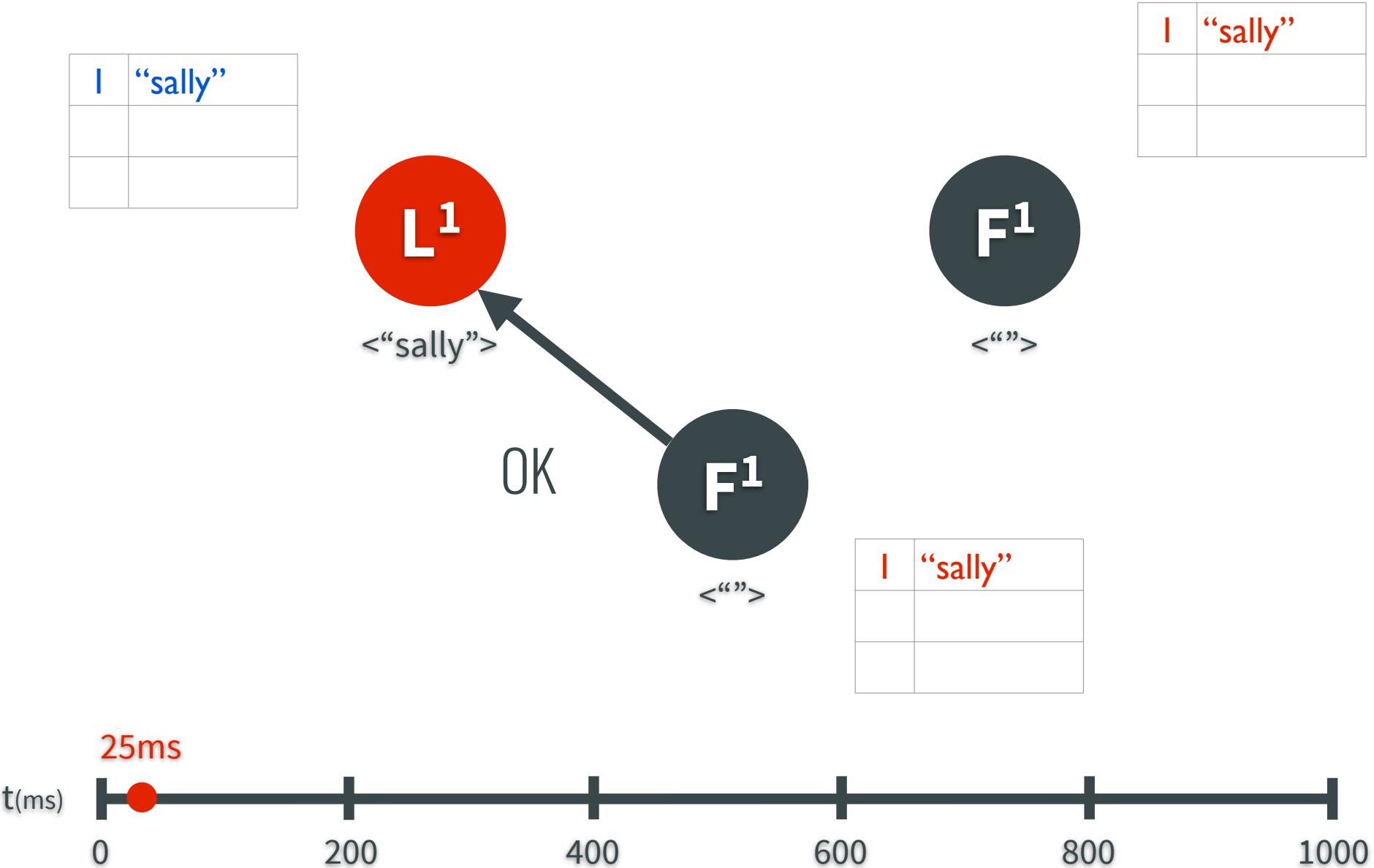
# Log Replication

A majority of nodes have written the log entry written to disk so it becomes committed



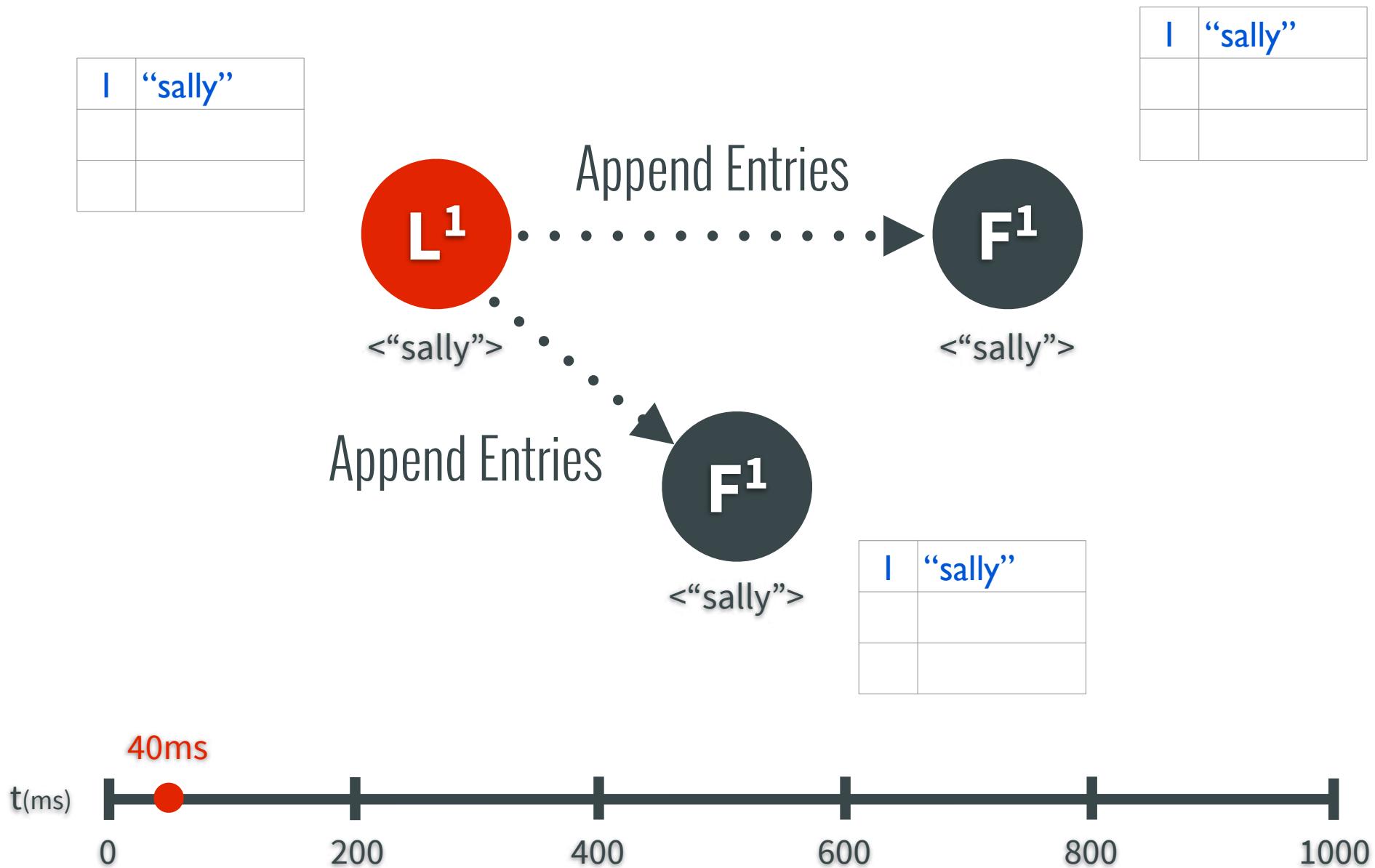
# Log Replication

---



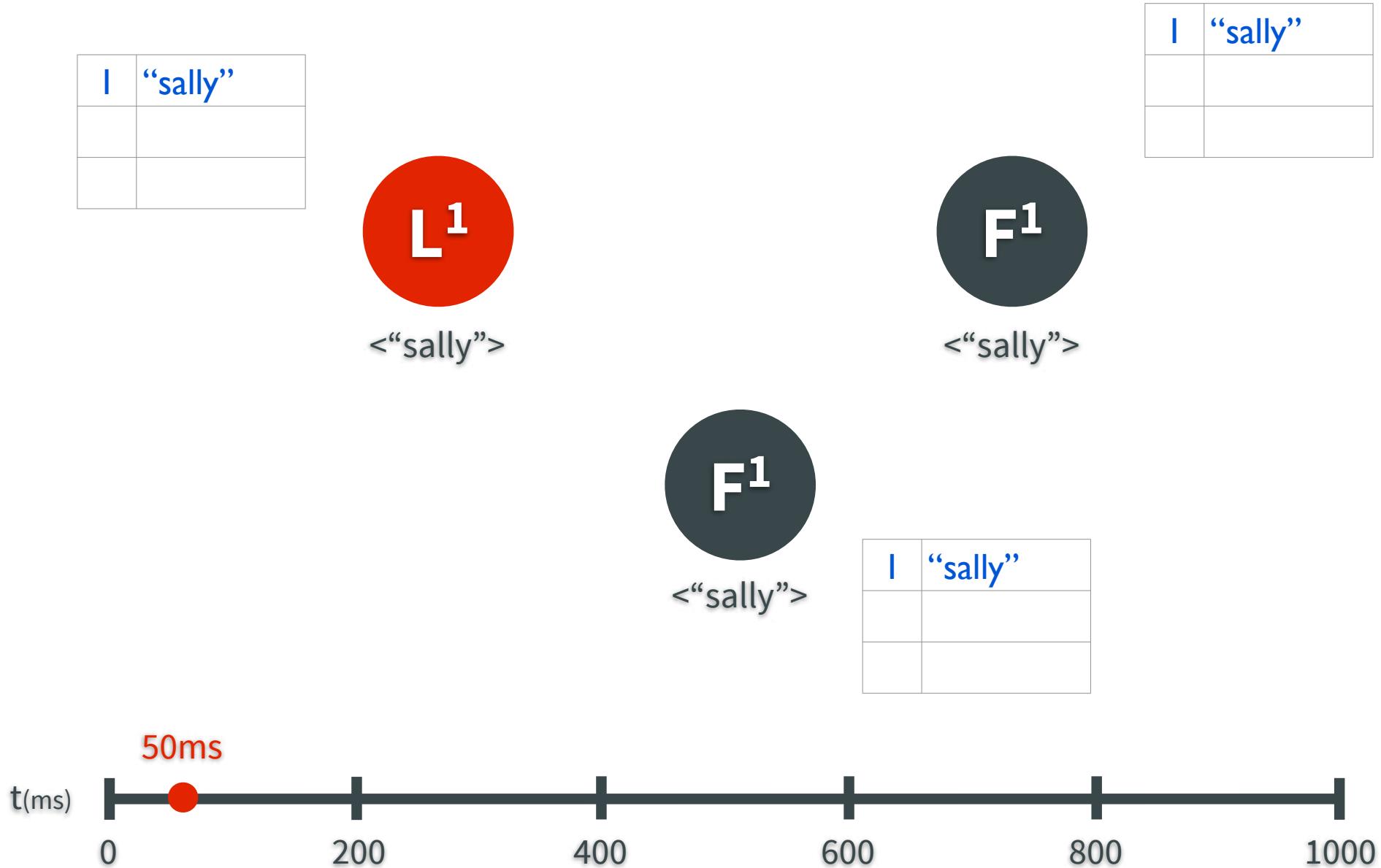
# Log Replication

At the next heartbeat, the leader notifies followers of updated committed entries



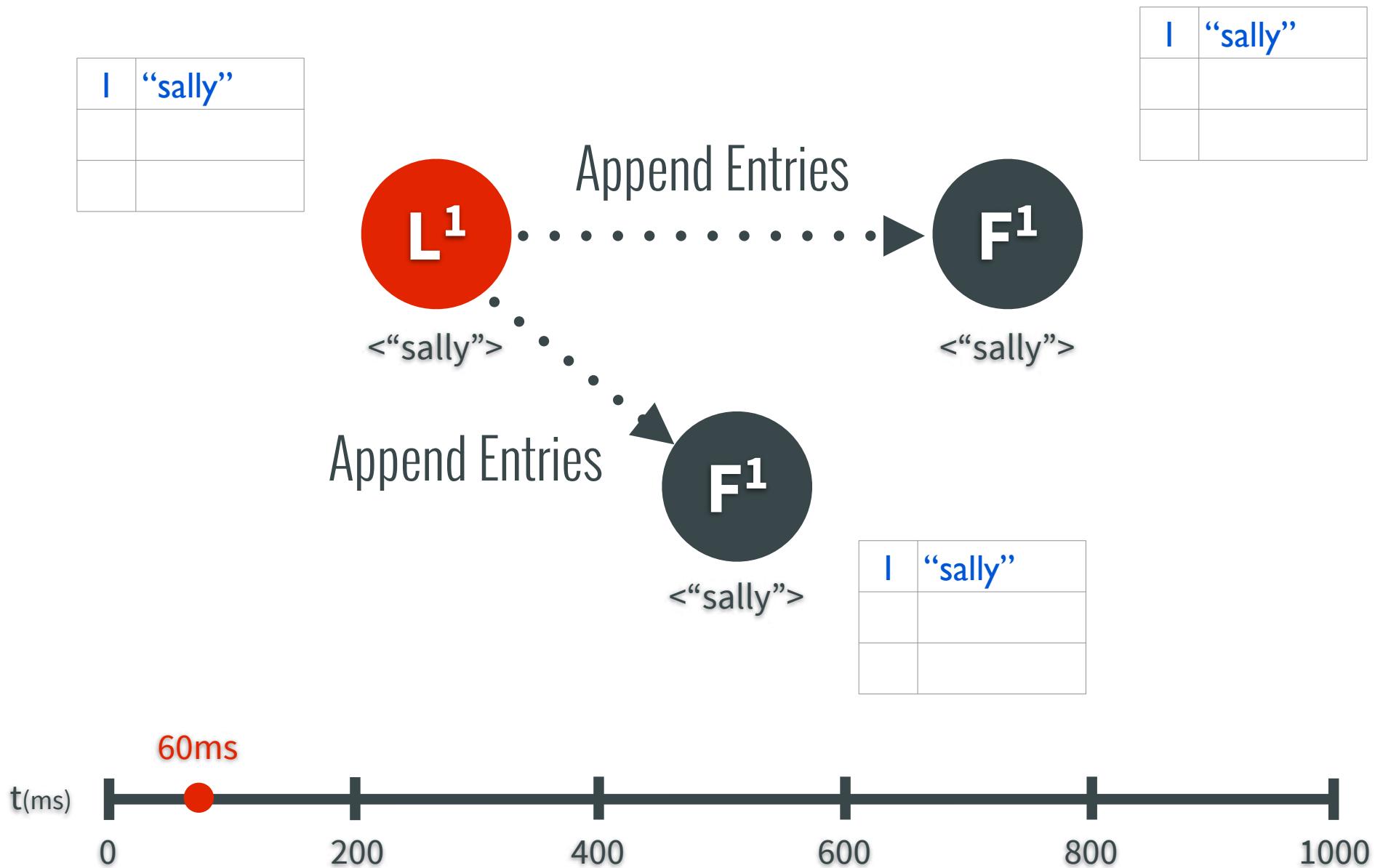
# Log Replication

---



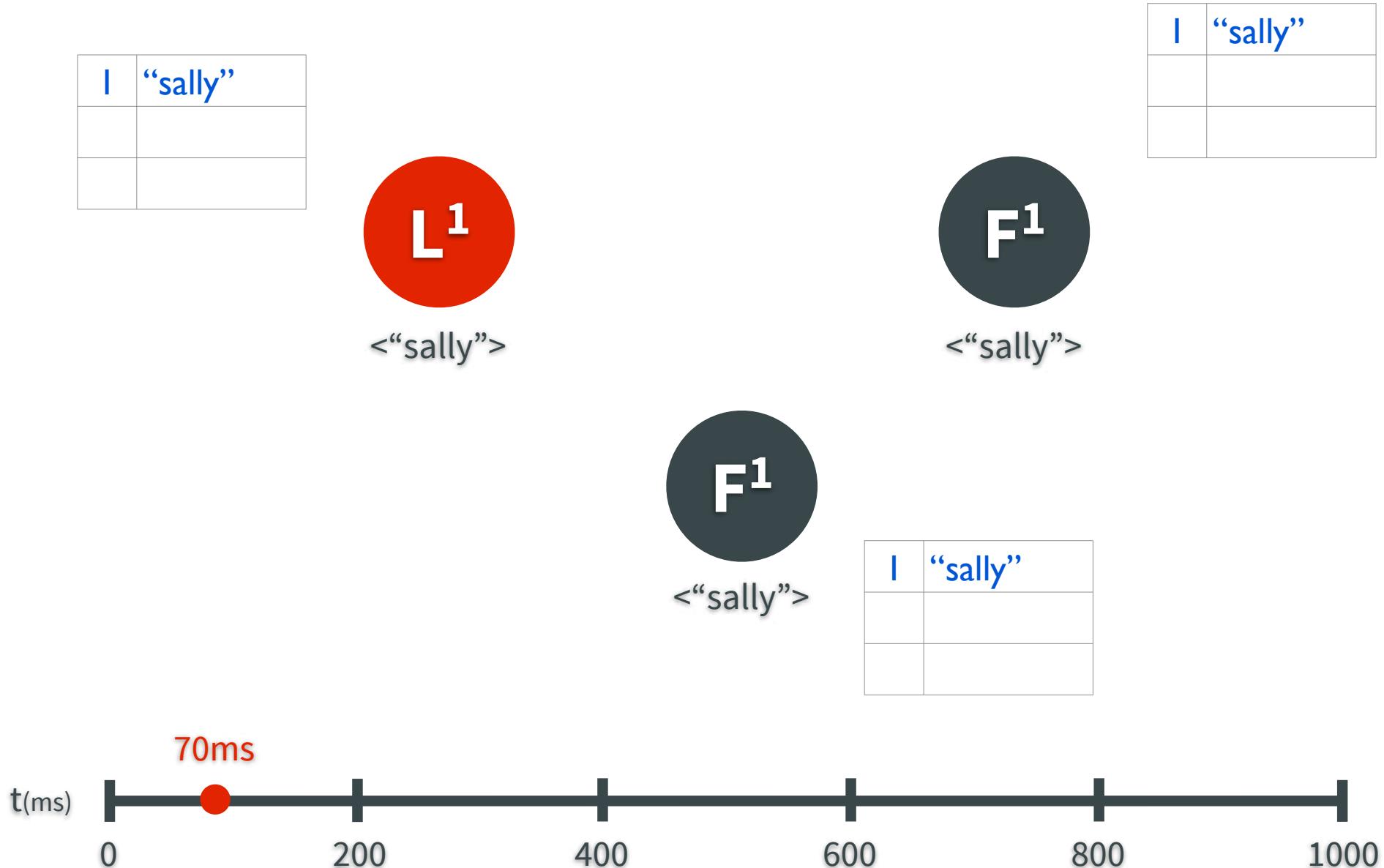
# Log Replication

At the next heartbeat, no new log information is sent



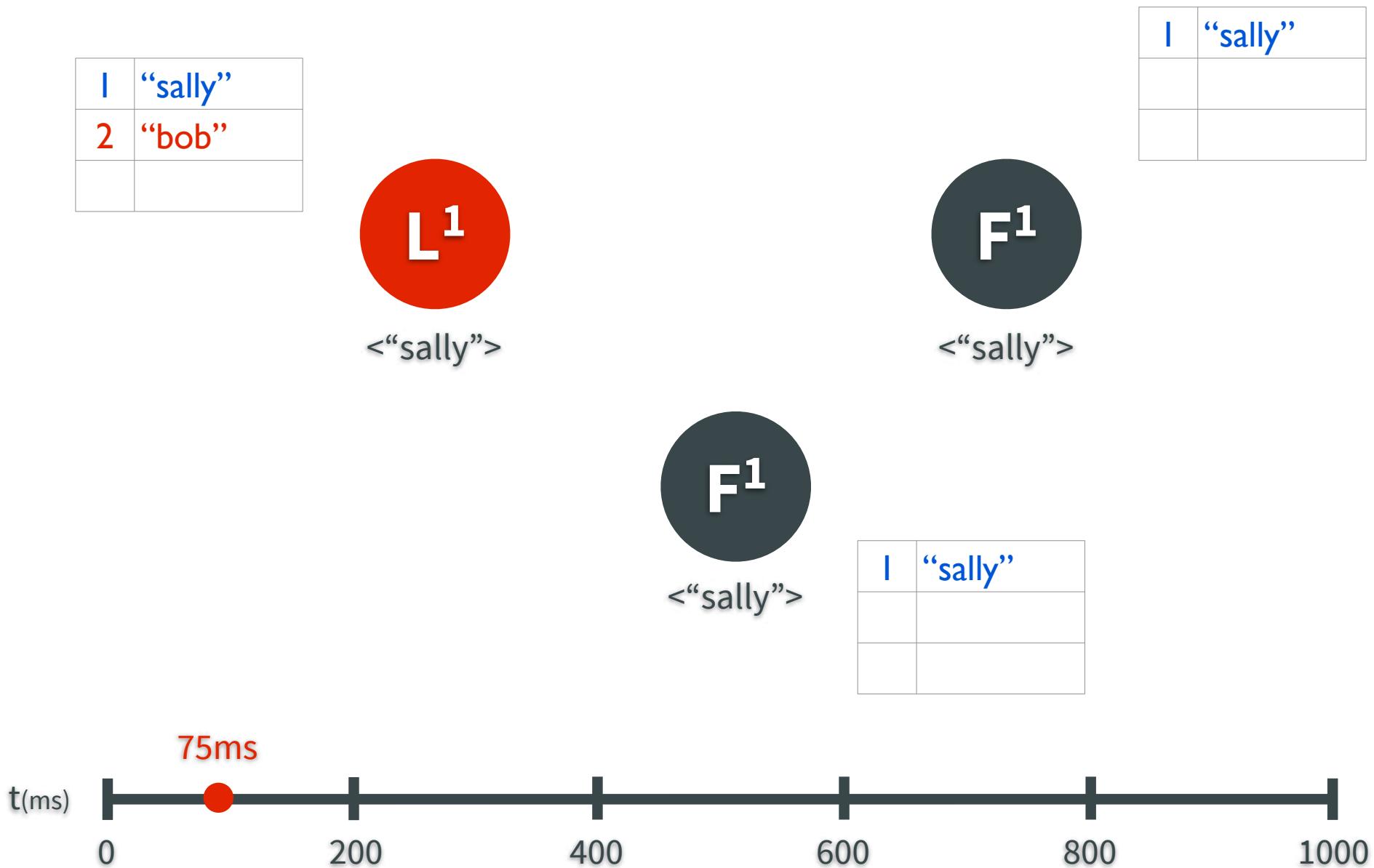
# Log Replication

---



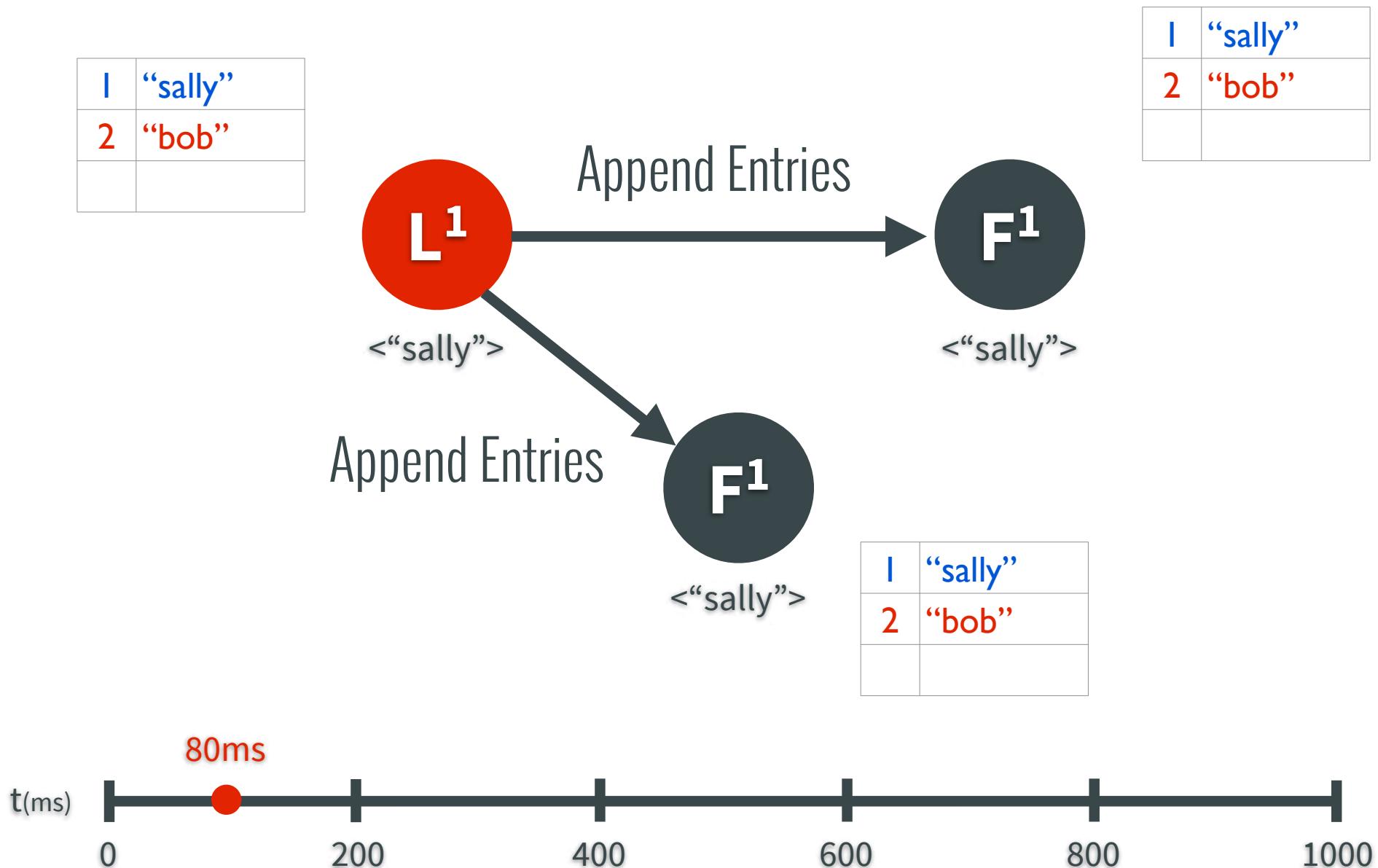
# Log Replication

A new uncommitted log entry is added to the leader



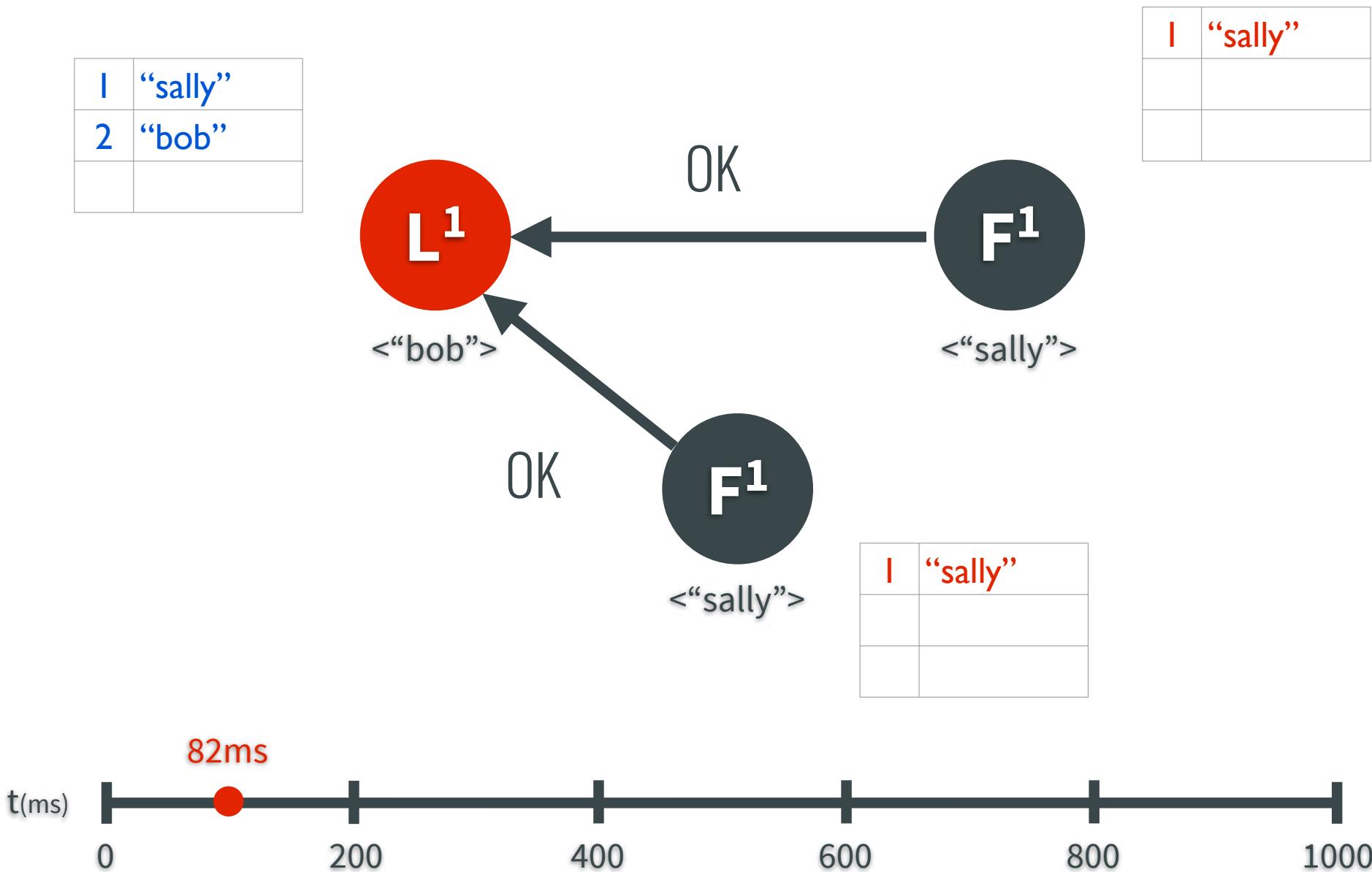
# Log Replication

At the next heartbeat, the entry is replicated to the followers



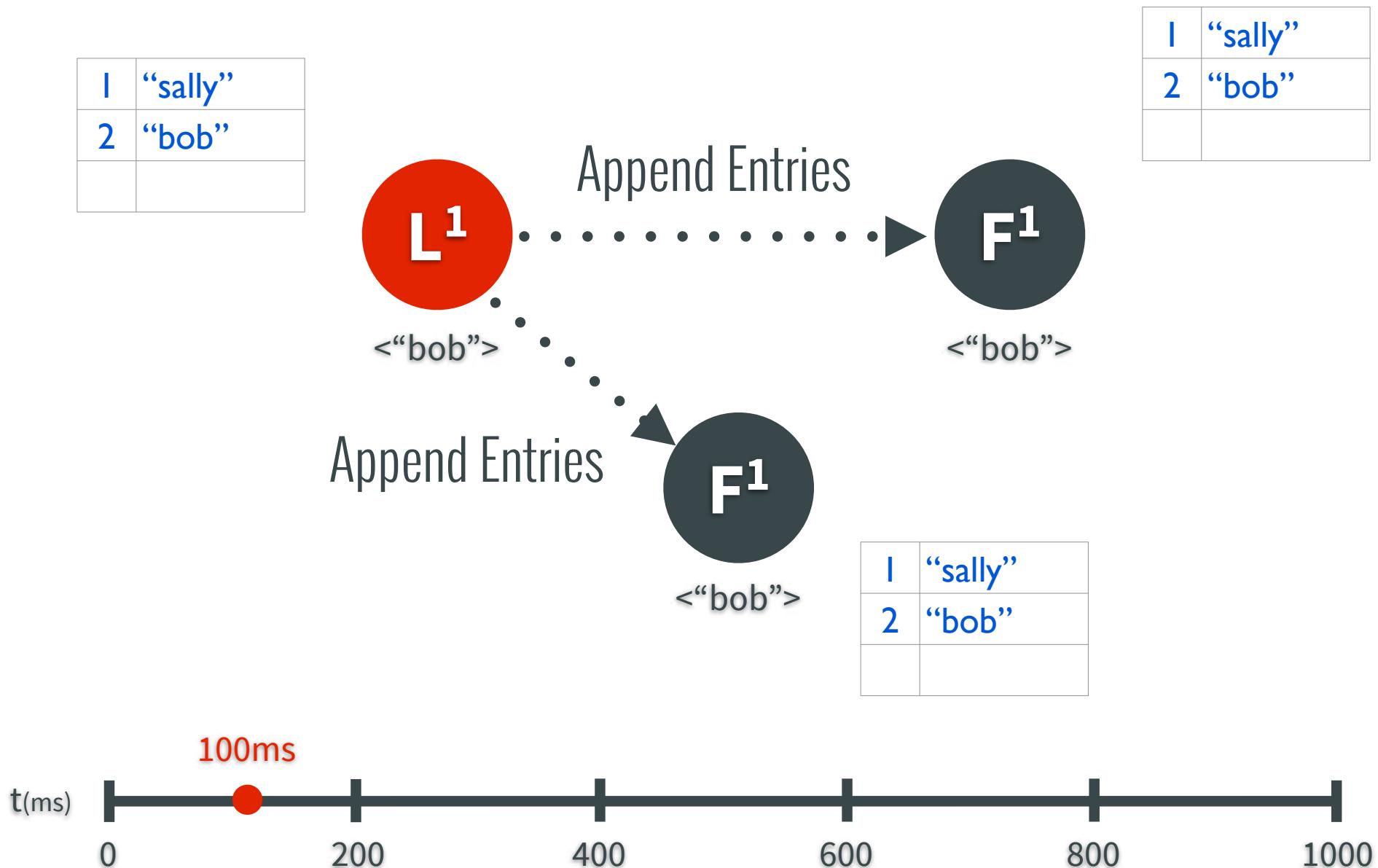
# Log Replication

The entry is committed once the followers acknowledge the request



# Log Replication

At the next heartbeat, the leader notifies the followers of the new committed entry

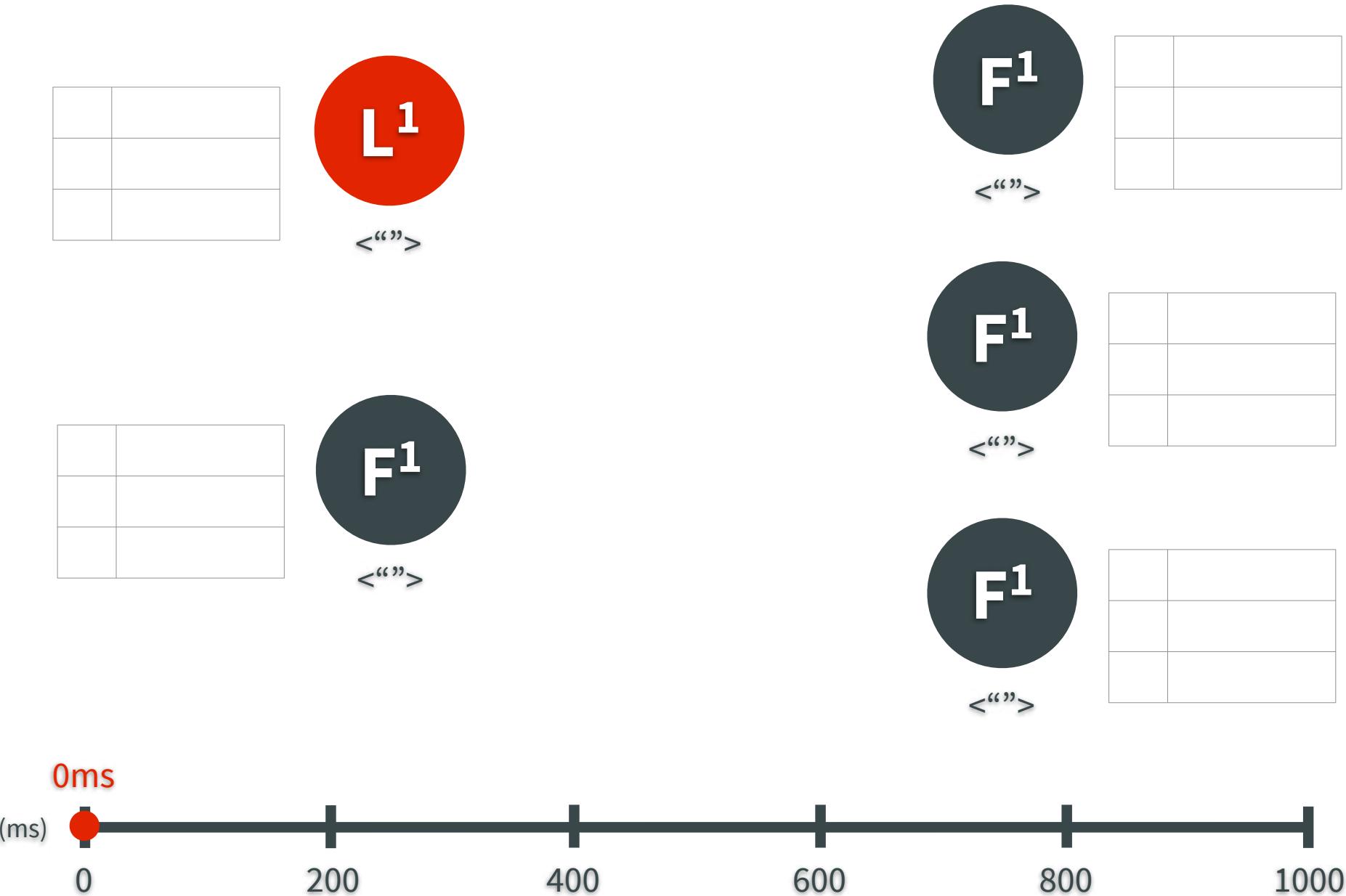


# Log Replication

## (with Network Partitions)

# Log Replication

---



# Log Replication

A new uncommitted log entry is added to the leader

I	“sally”







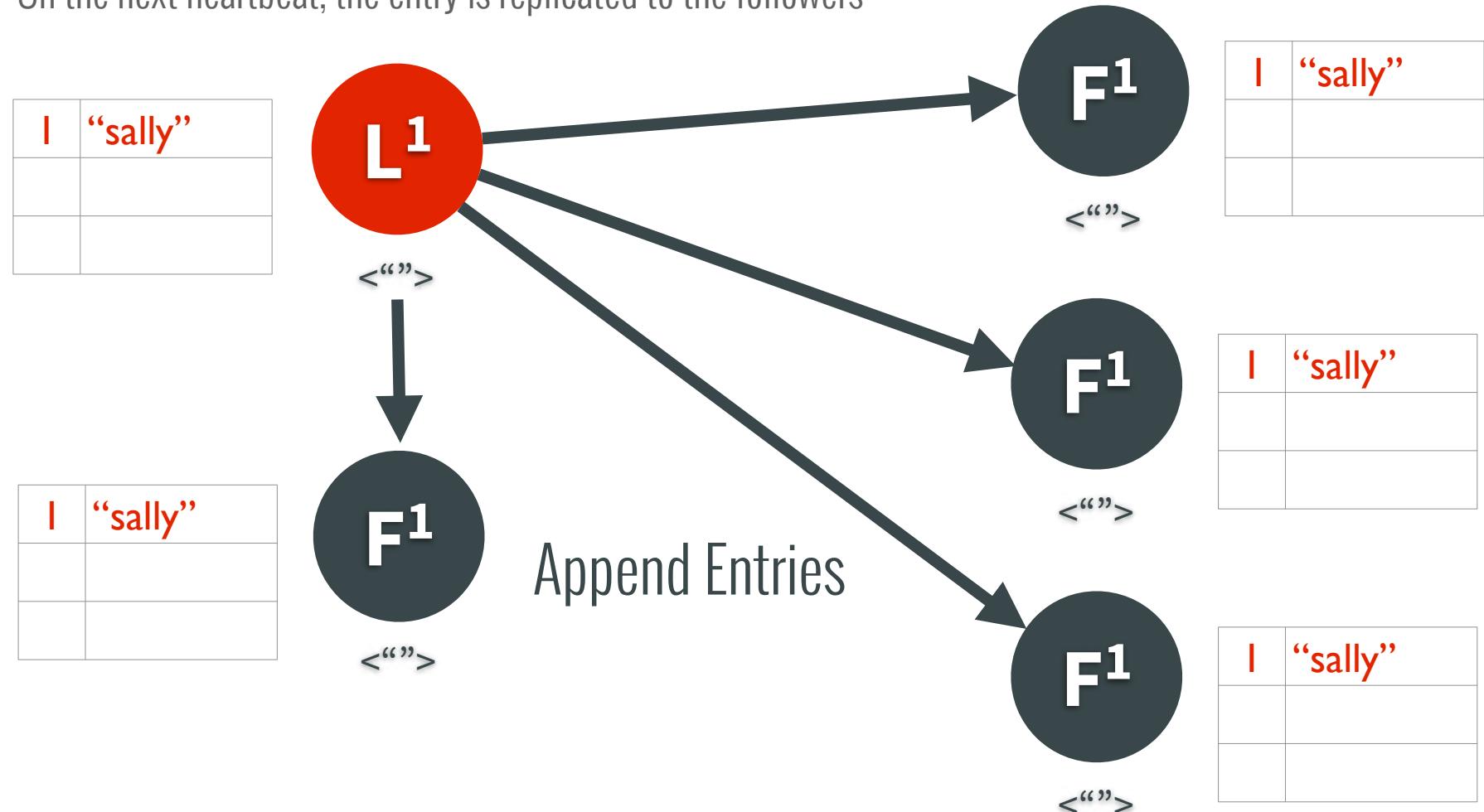






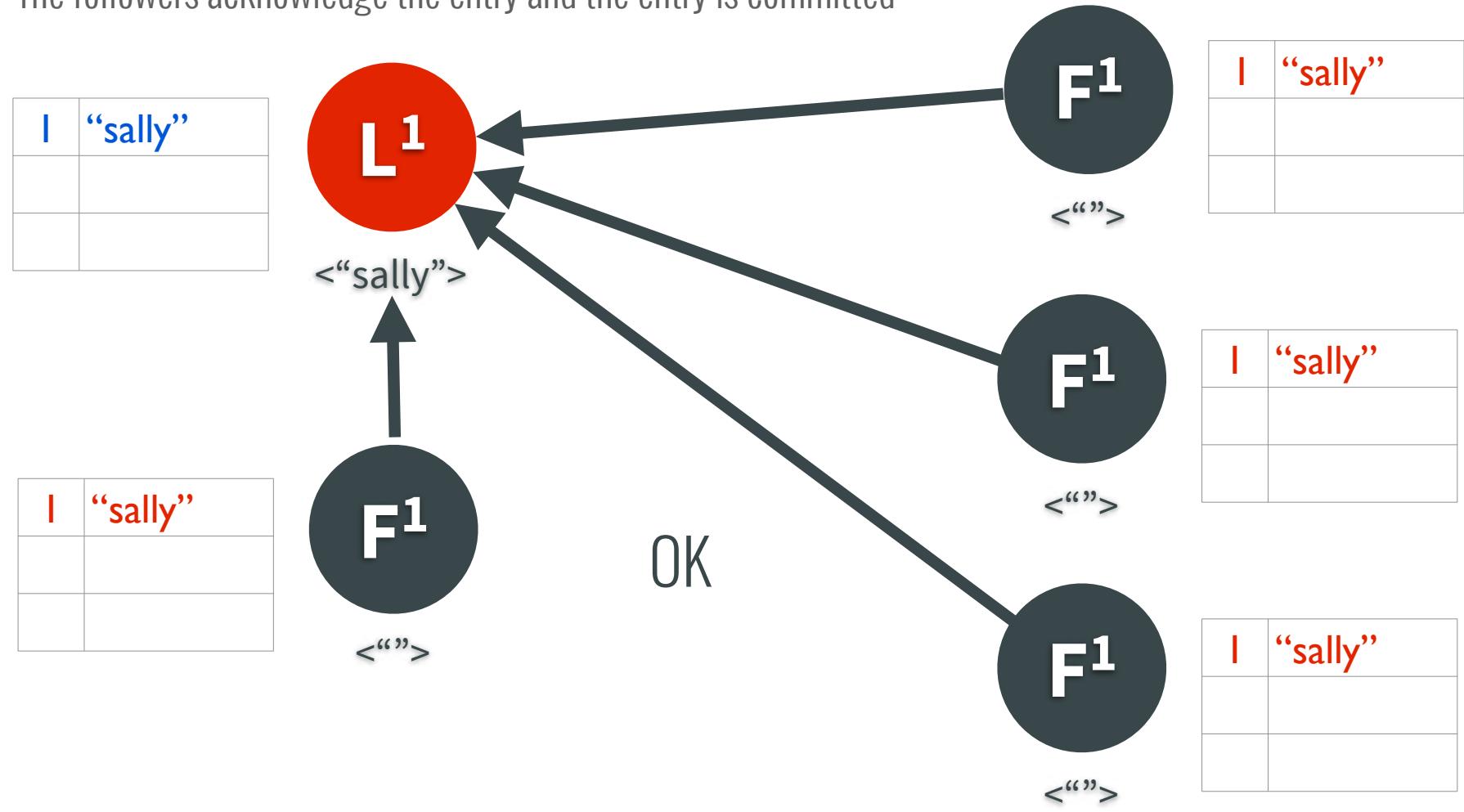
# Log Replication

On the next heartbeat, the entry is replicated to the followers



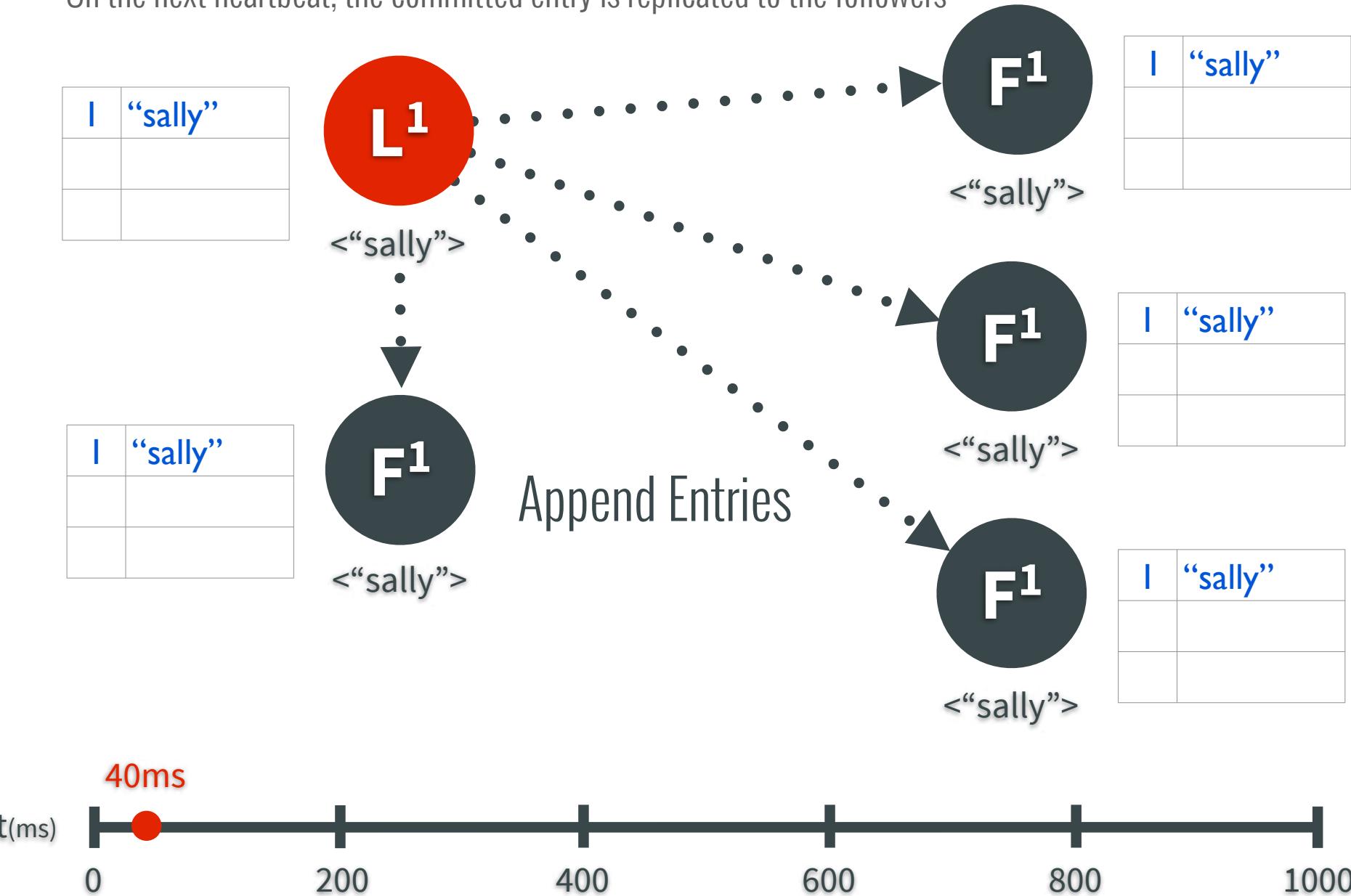
# Log Replication

The followers acknowledge the entry and the entry is committed



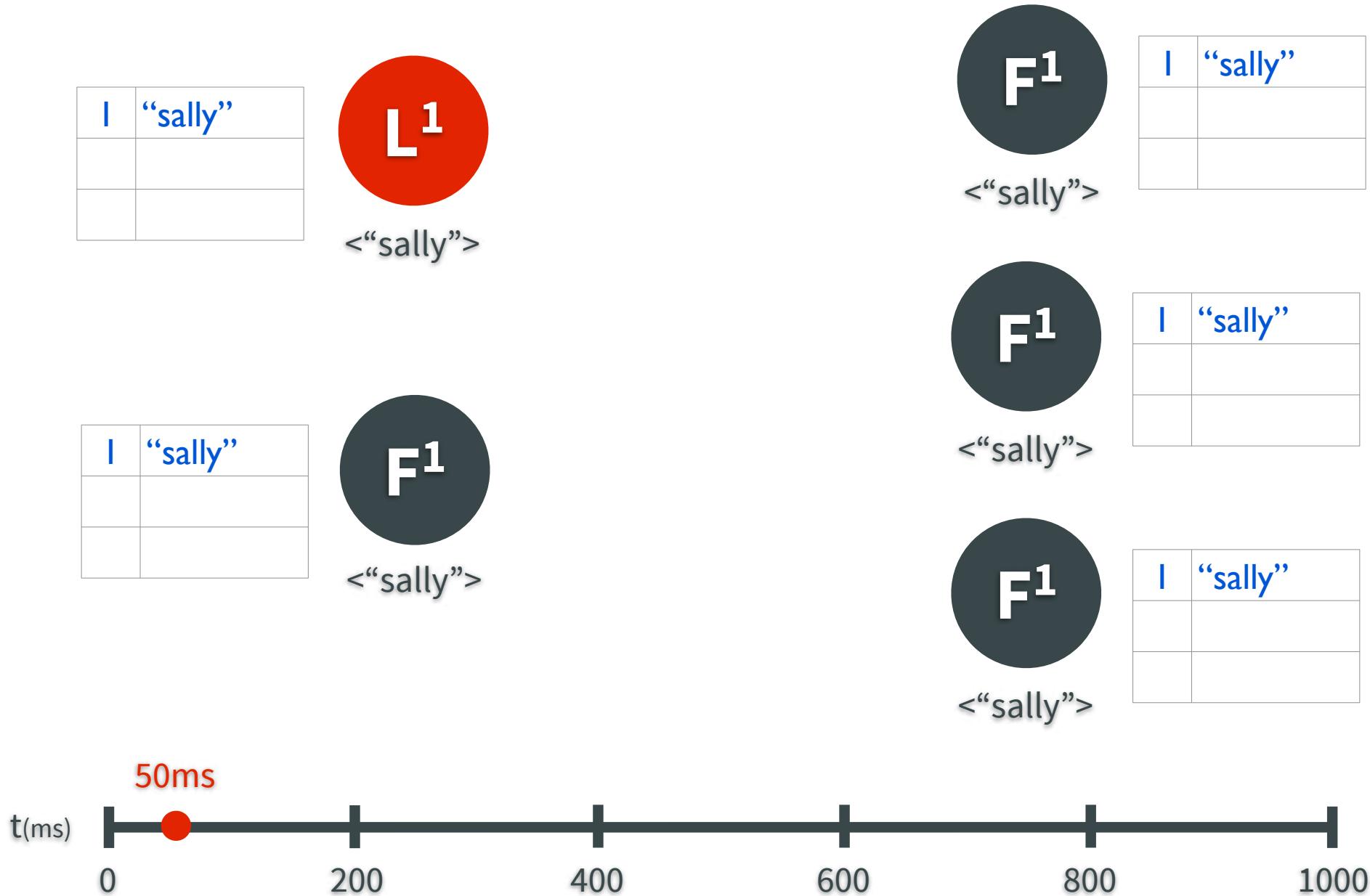
# Log Replication

On the next heartbeat, the committed entry is replicated to the followers



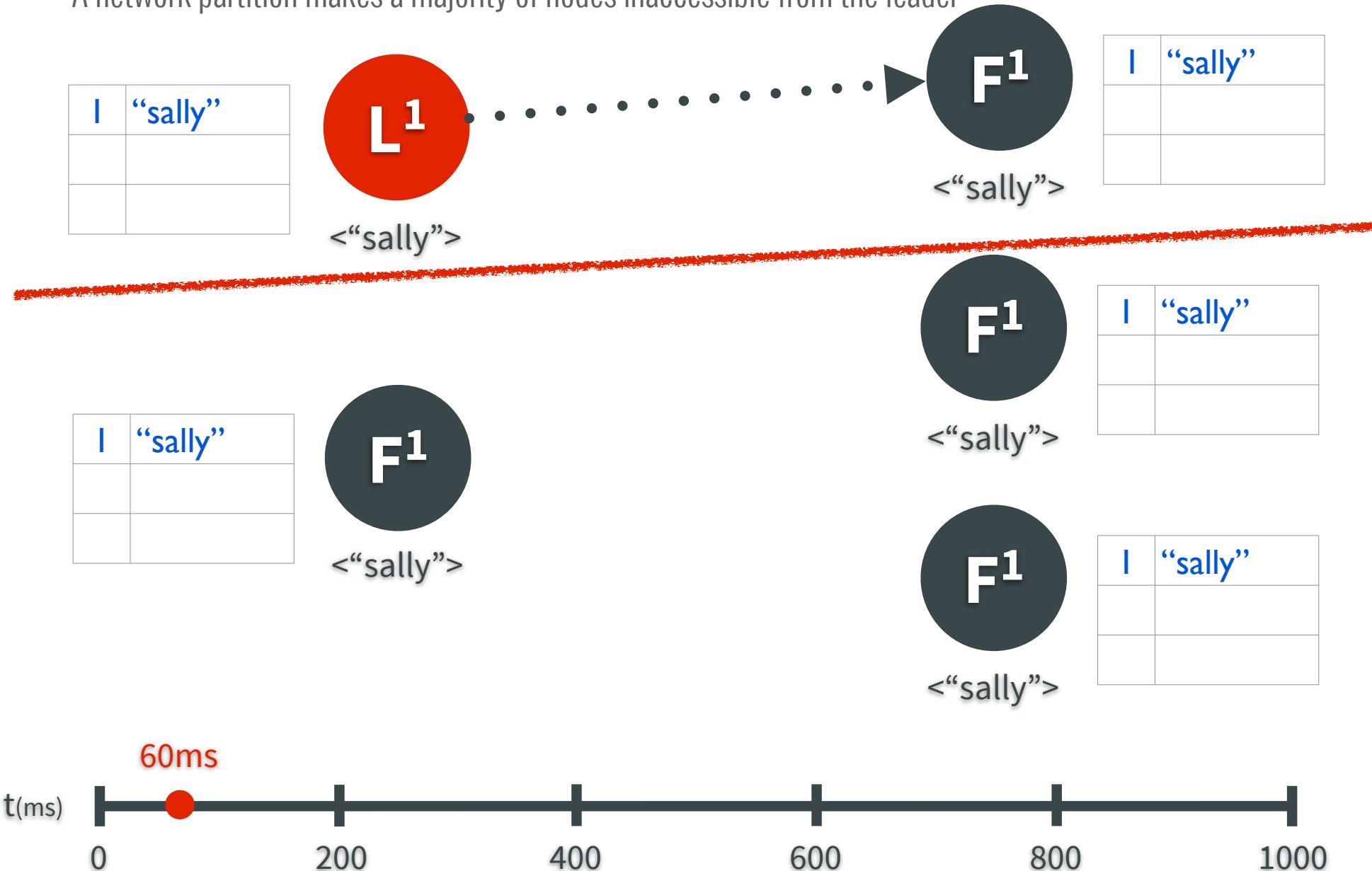
# Log Replication

---



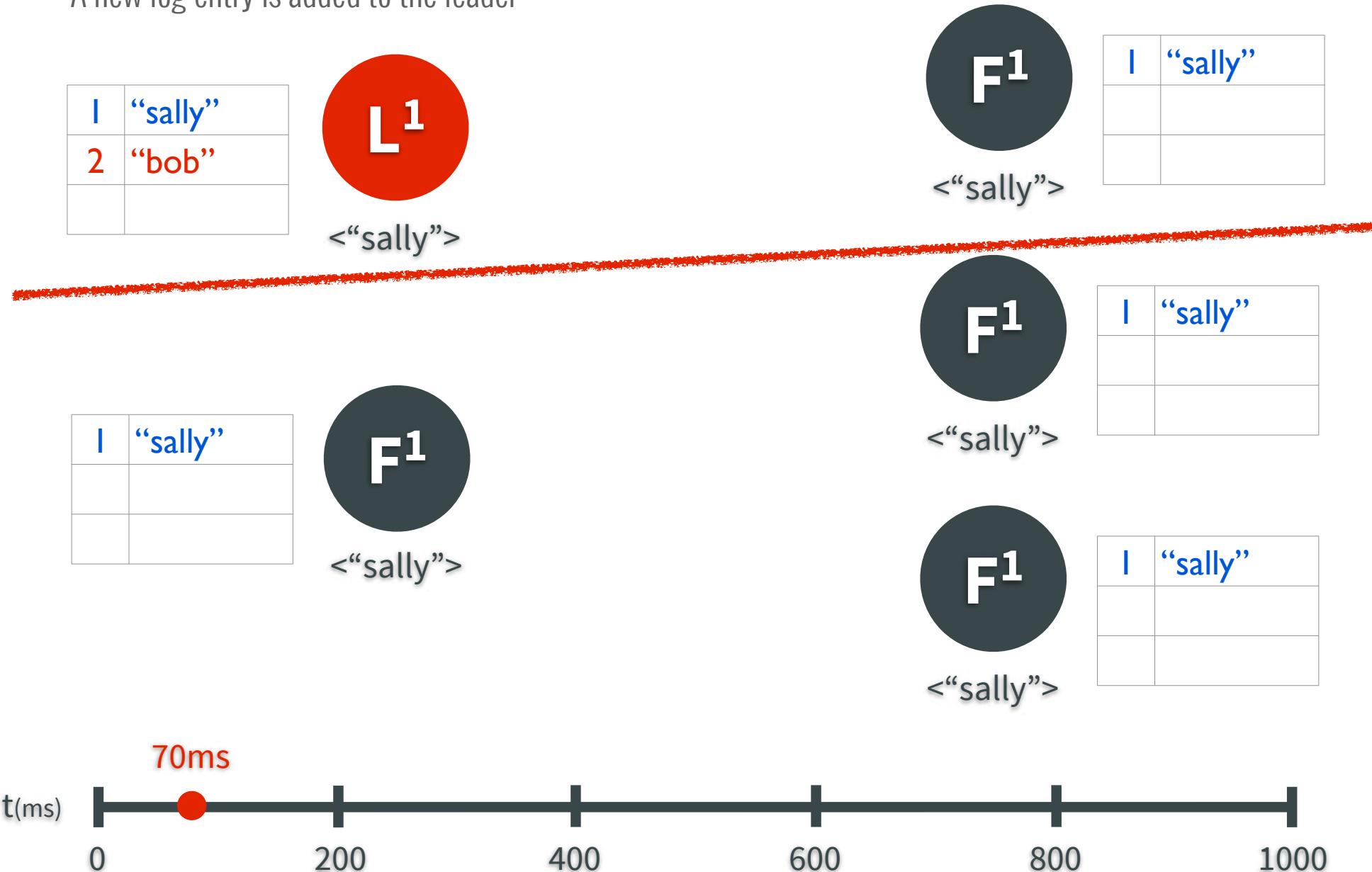
# Log Replication

A network partition makes a majority of nodes inaccessible from the leader



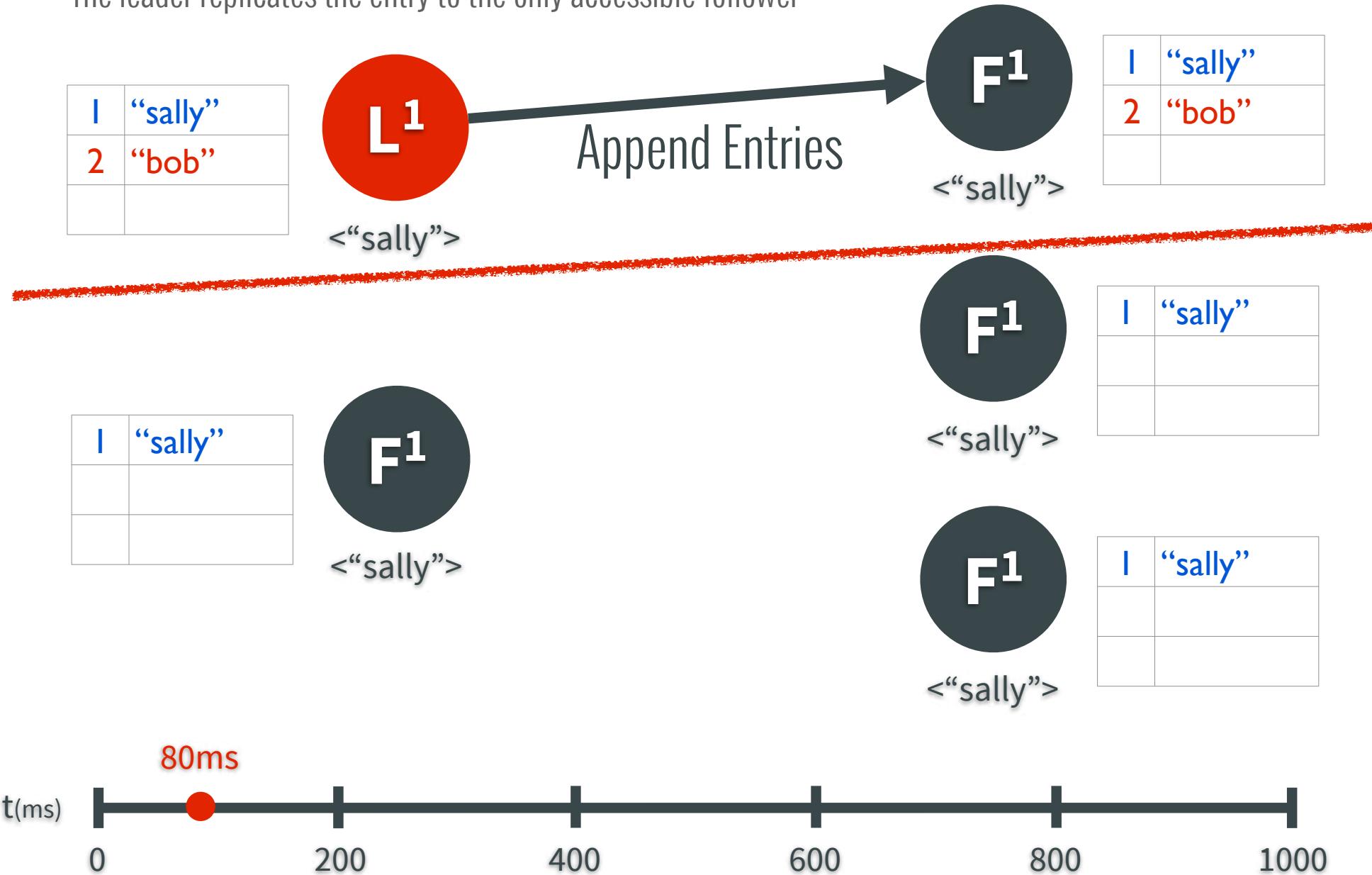
# Log Replication

A new log entry is added to the leader



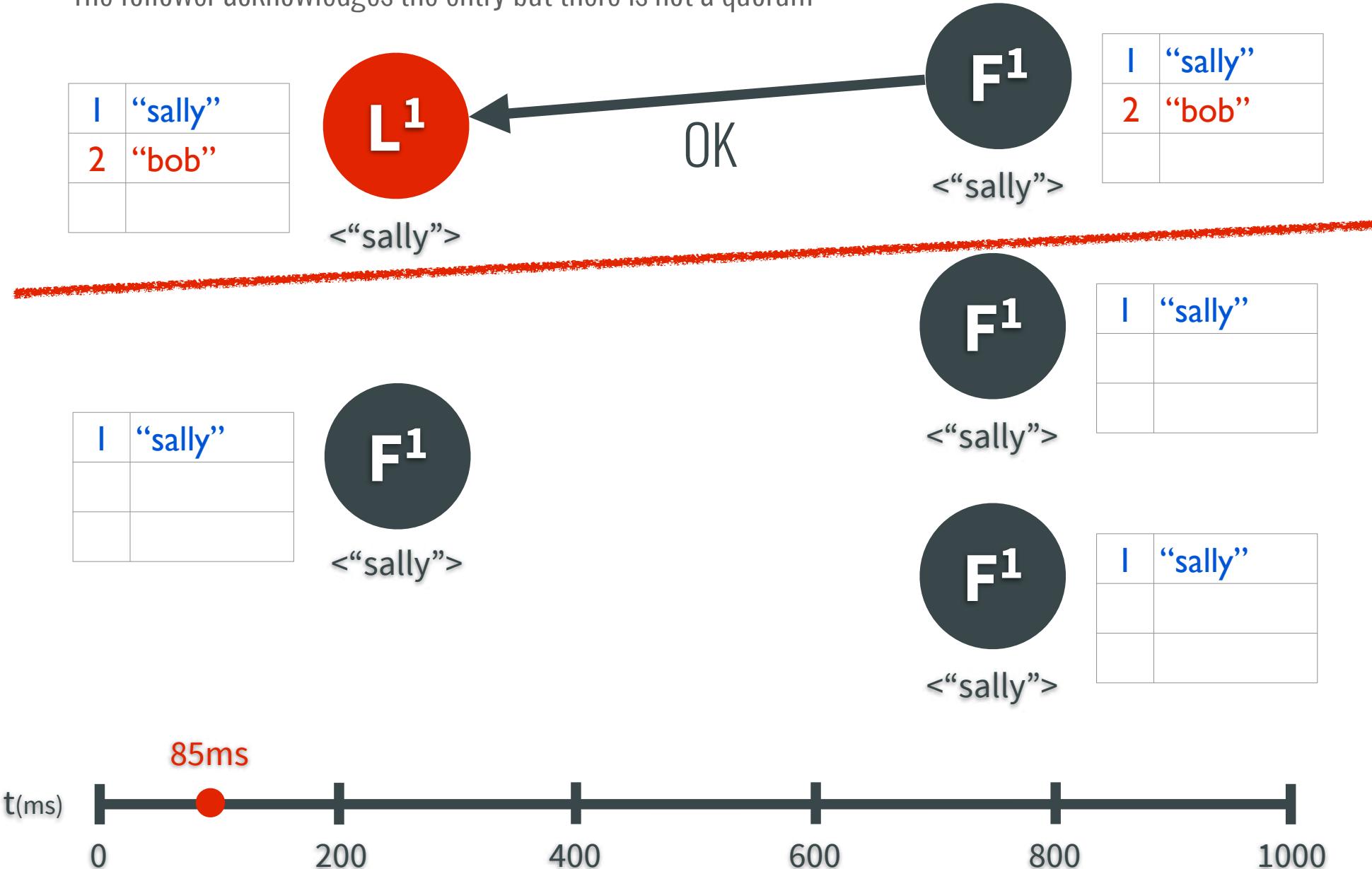
# Log Replication

The leader replicates the entry to the only accessible follower

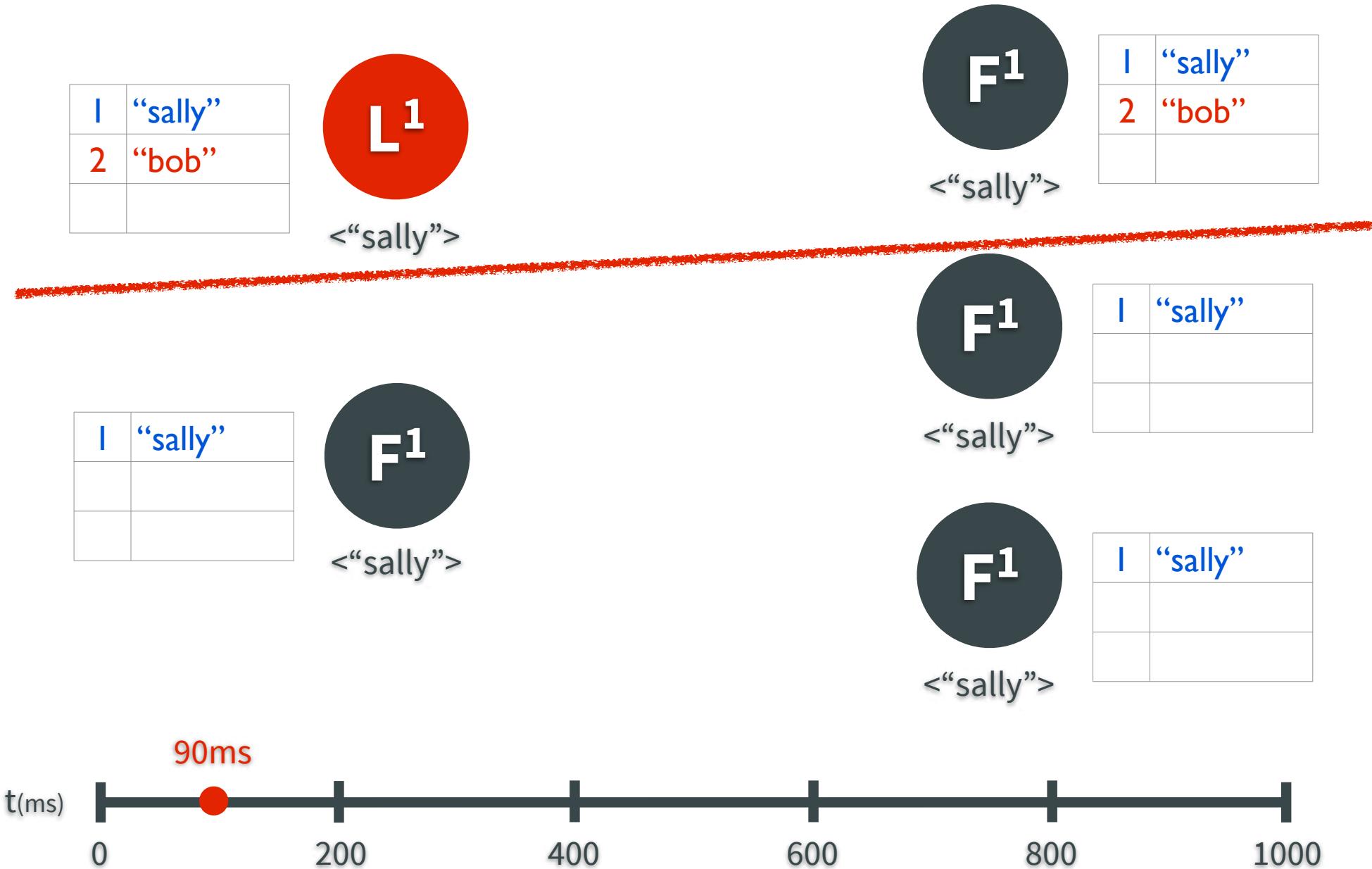


# Log Replication

The follower acknowledges the entry but there is not a quorum

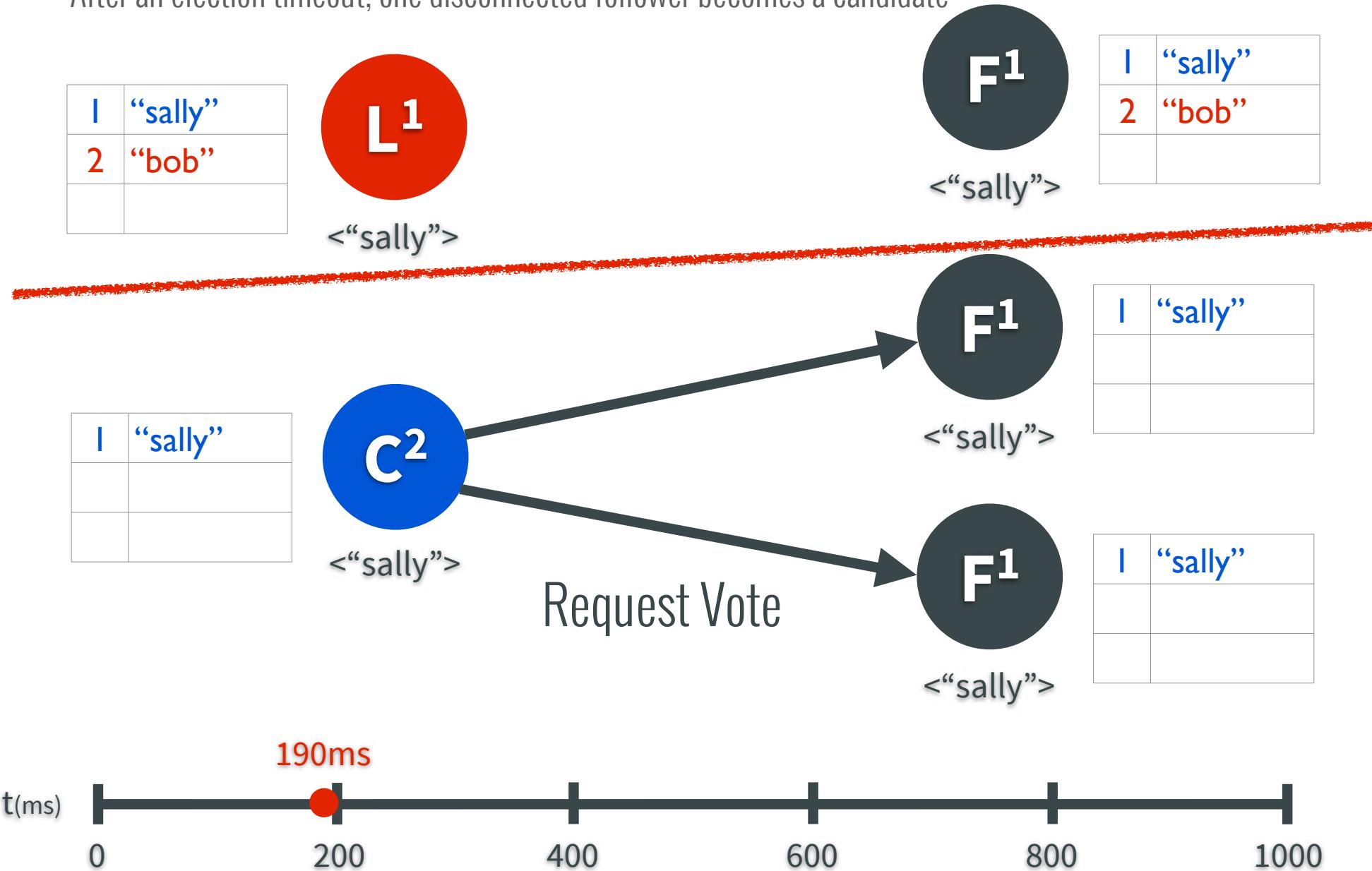


# Log Replication



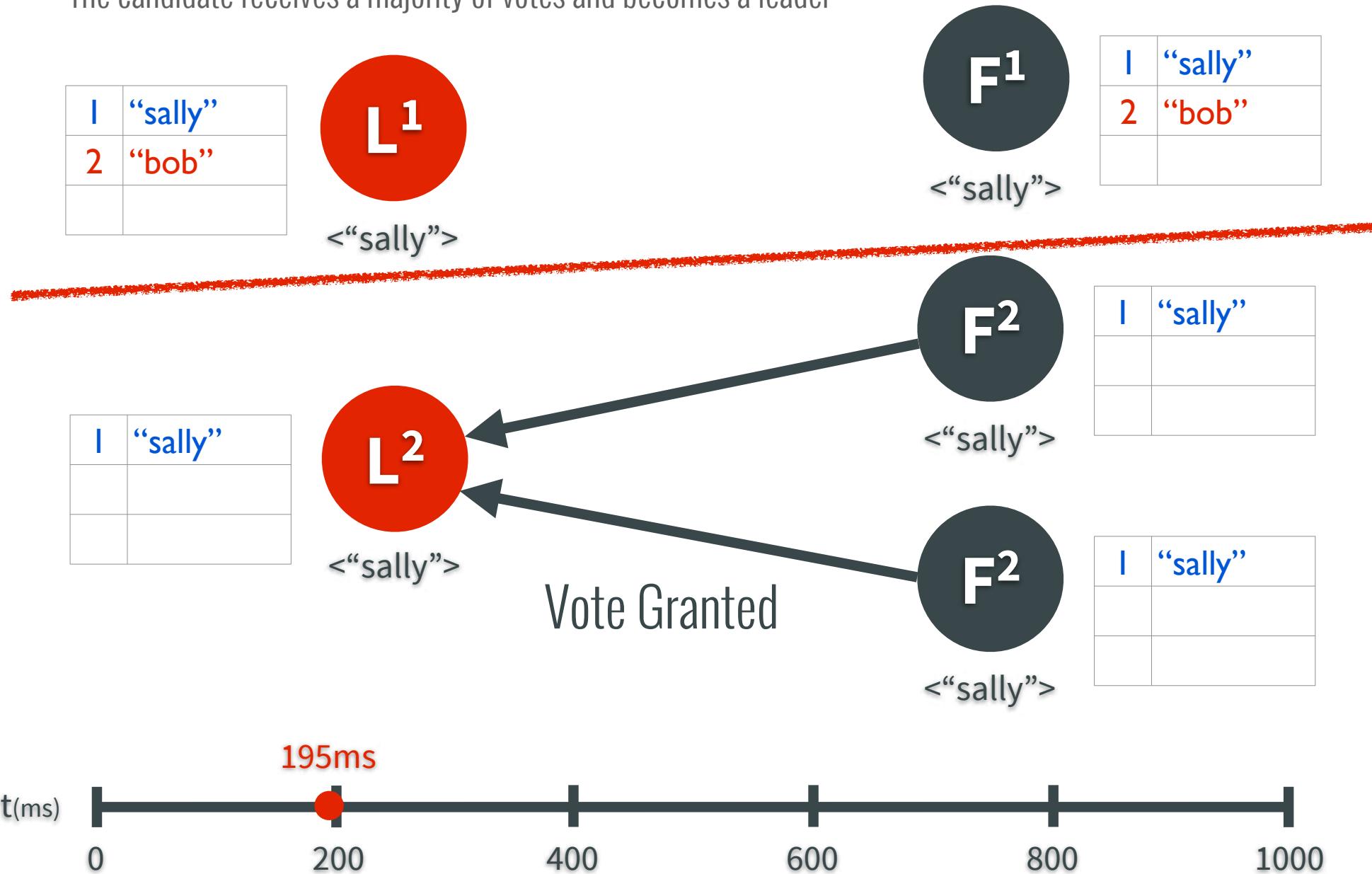
# Log Replication

After an election timeout, one disconnected follower becomes a candidate

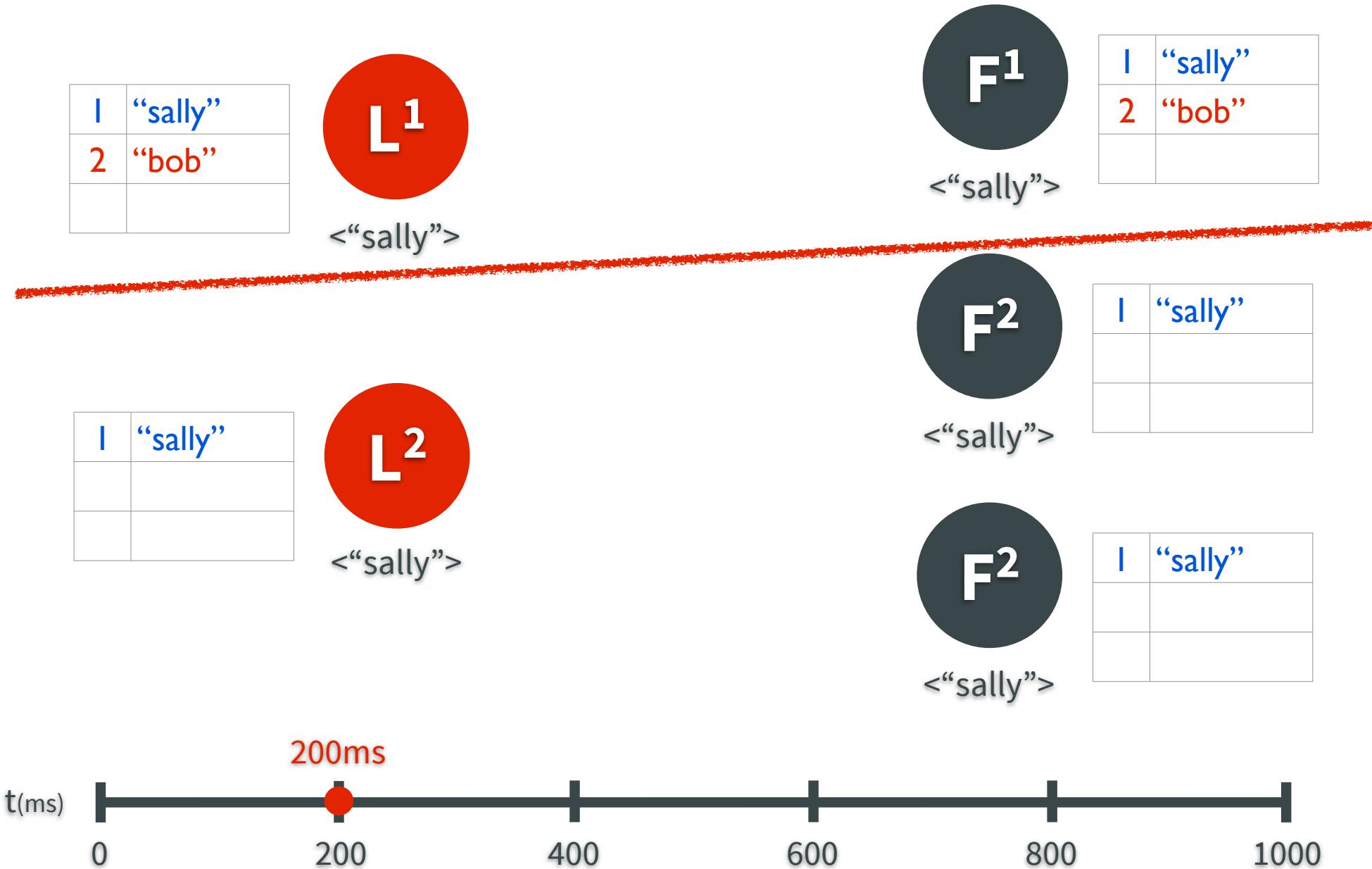


# Log Replication

The candidate receives a majority of votes and becomes a leader

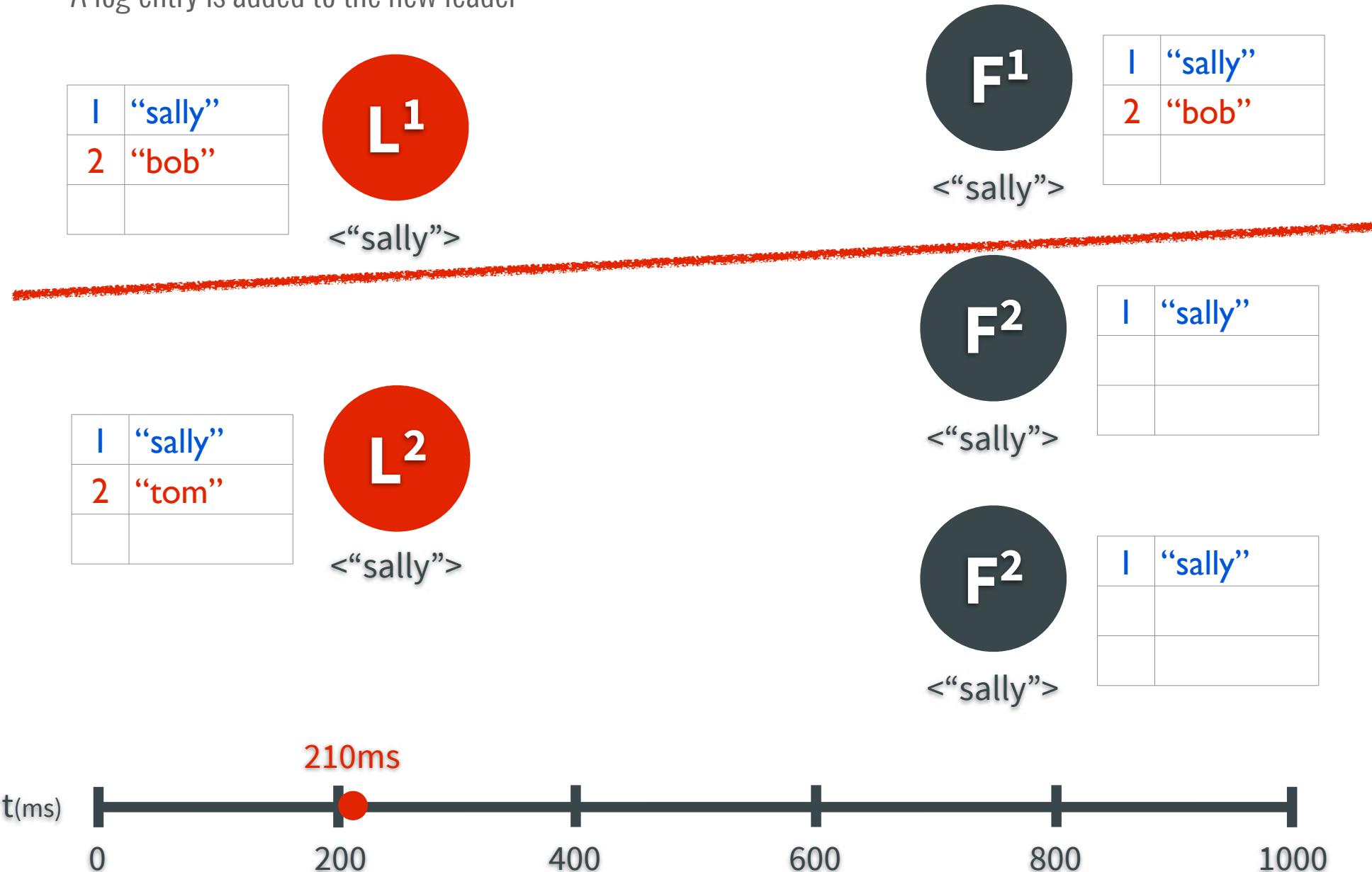


# Log Replication



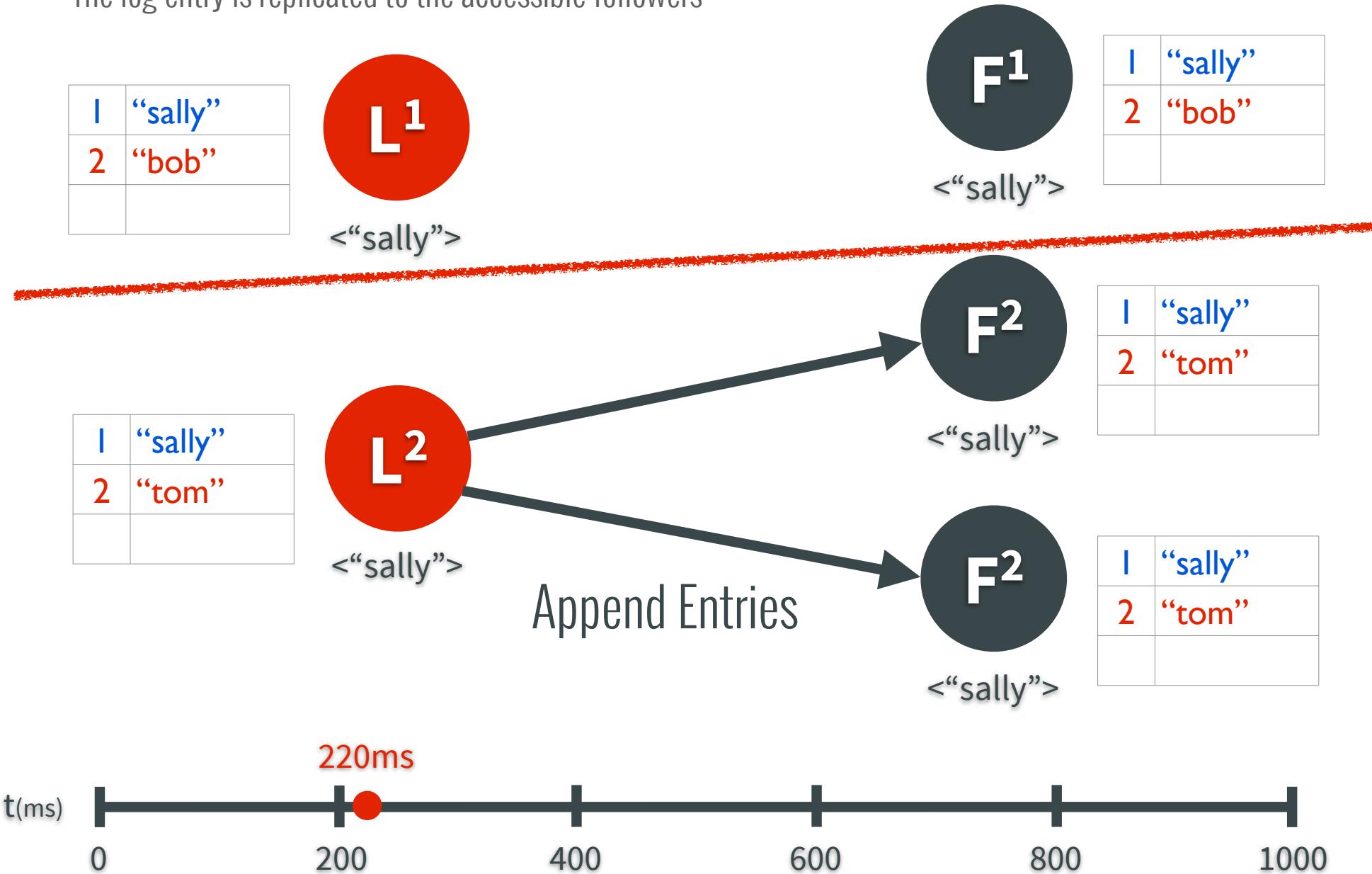
# Log Replication

A log entry is added to the new leader



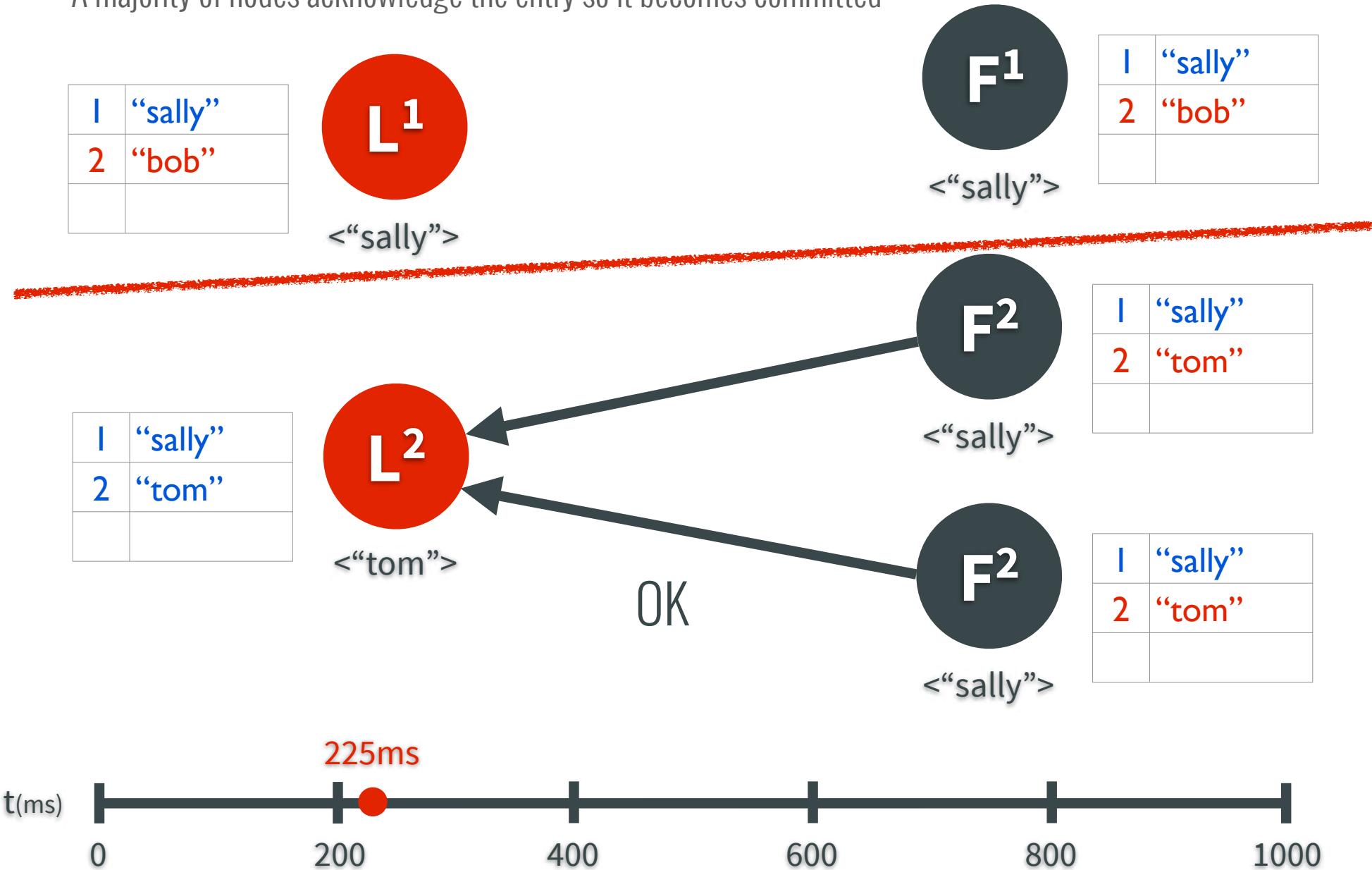
# Log Replication

The log entry is replicated to the accessible followers



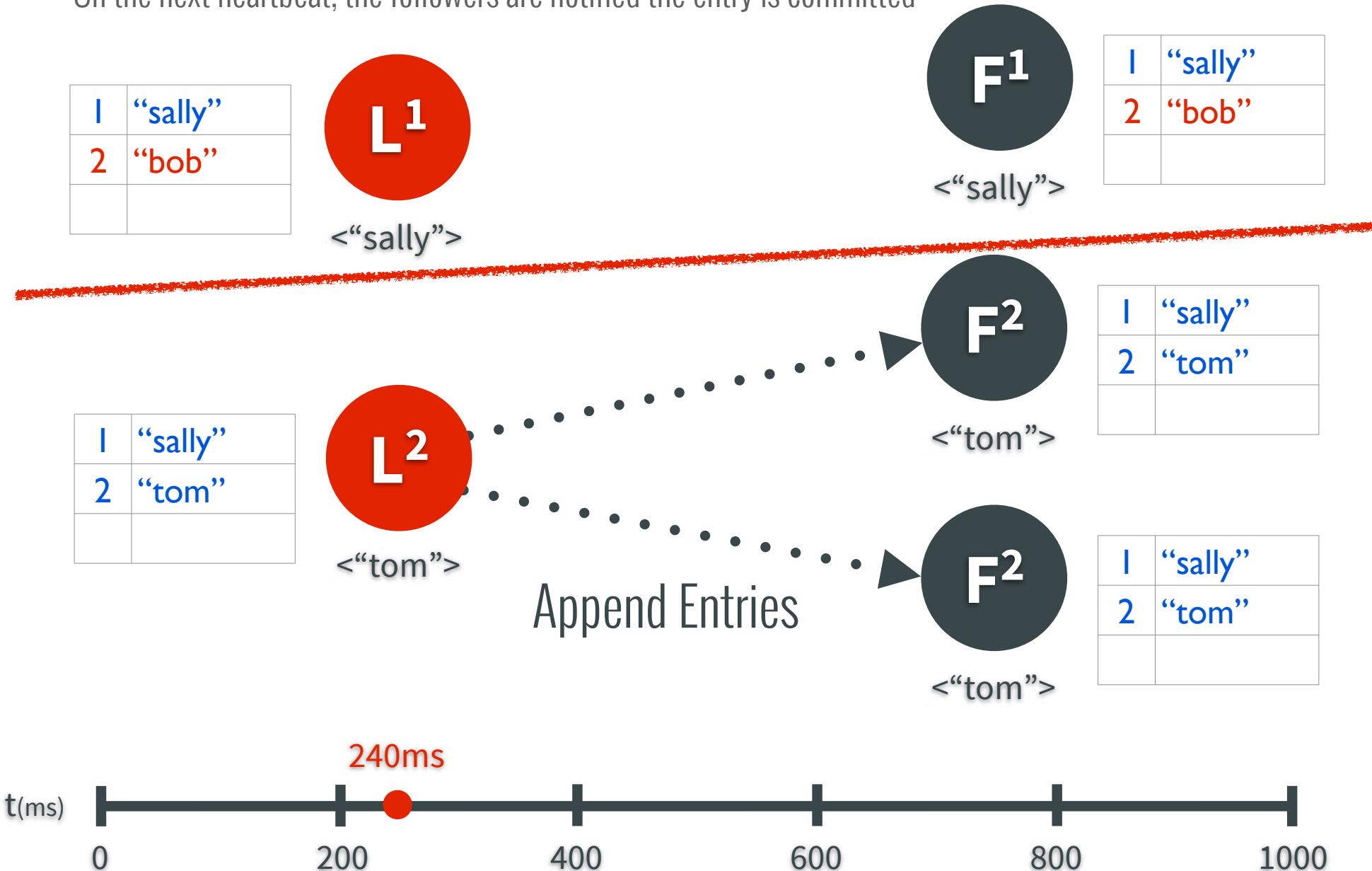
# Log Replication

A majority of nodes acknowledge the entry so it becomes committed

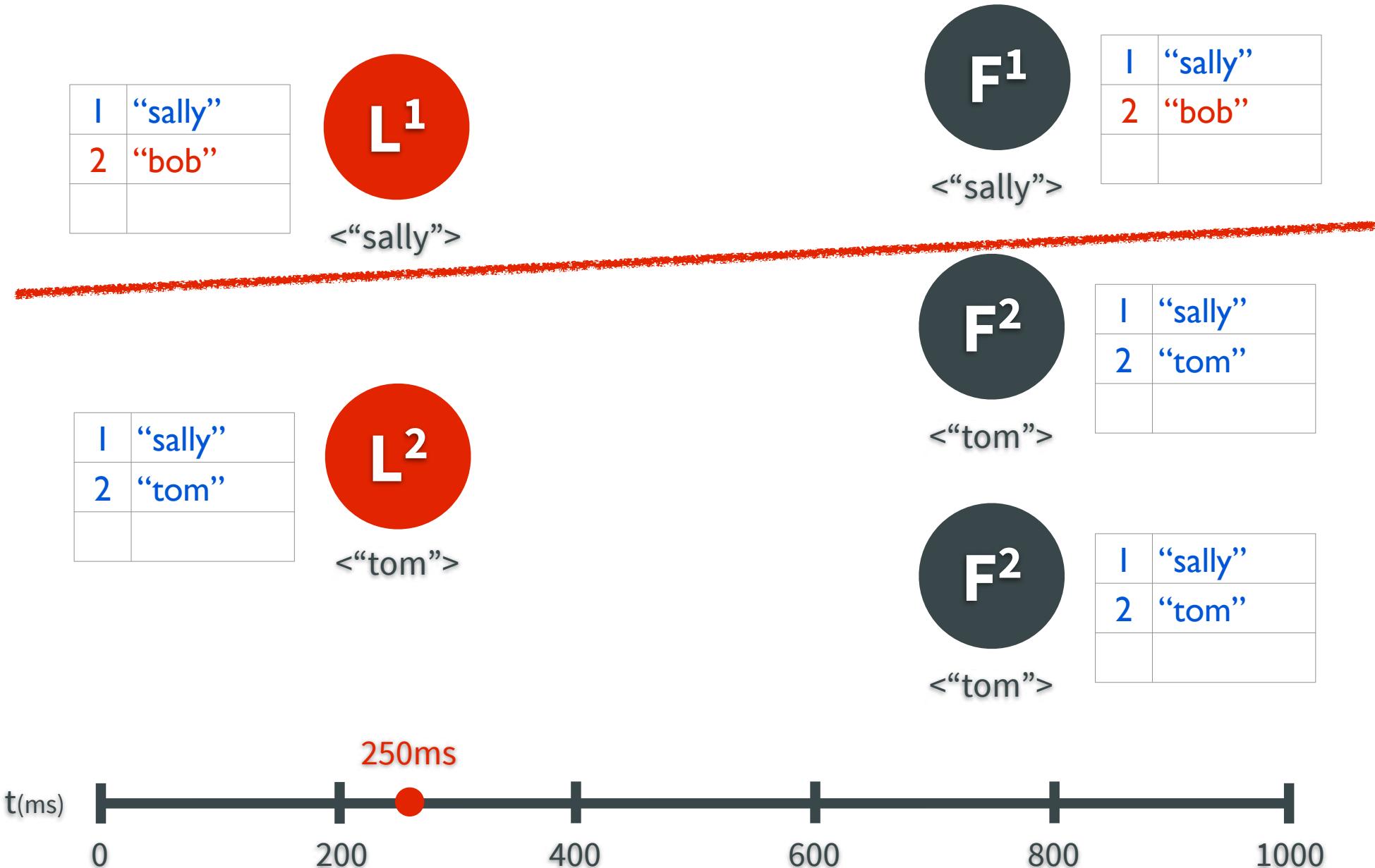


# Log Replication

On the next heartbeat, the followers are notified the entry is committed

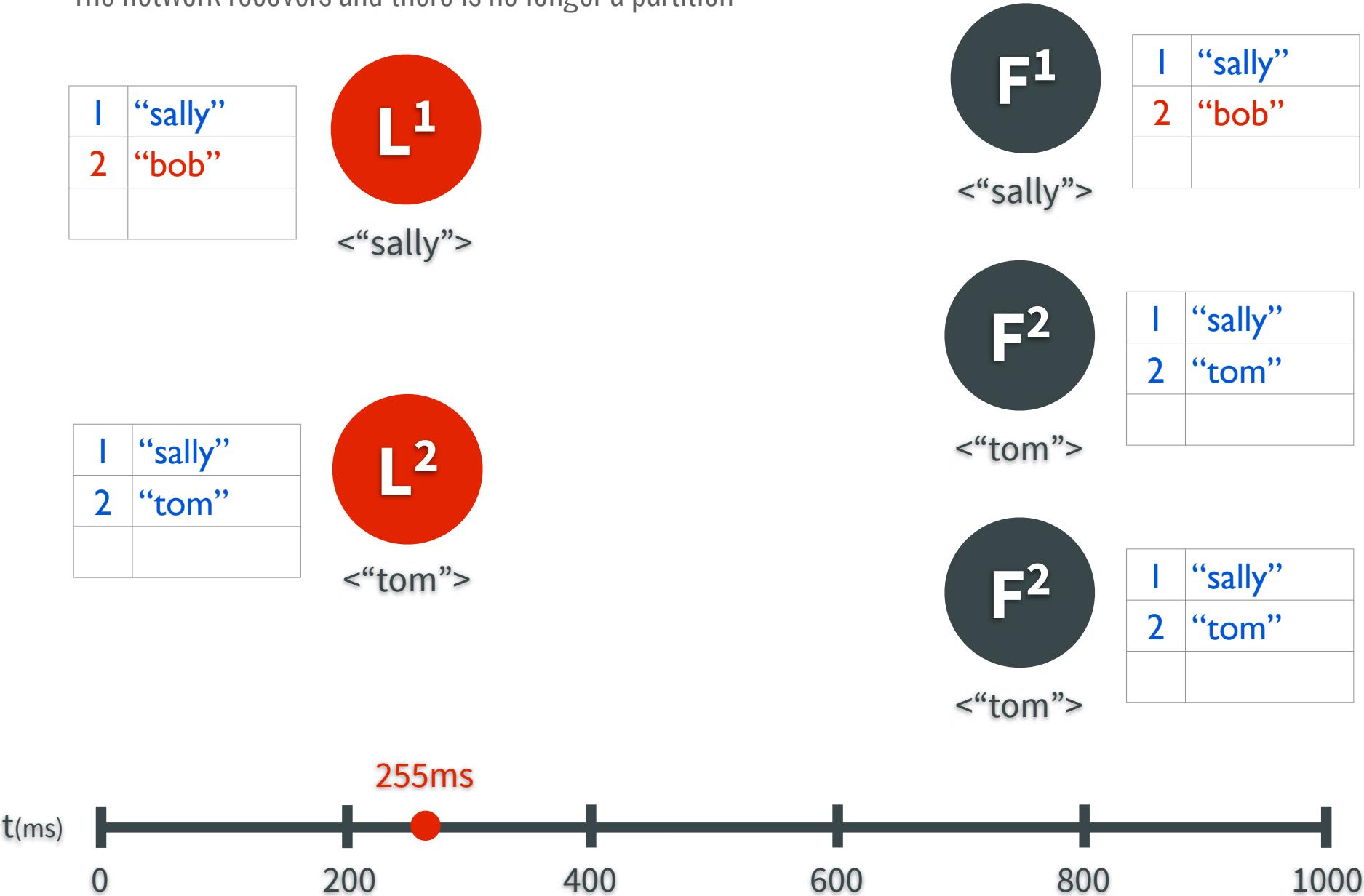


# Log Replication



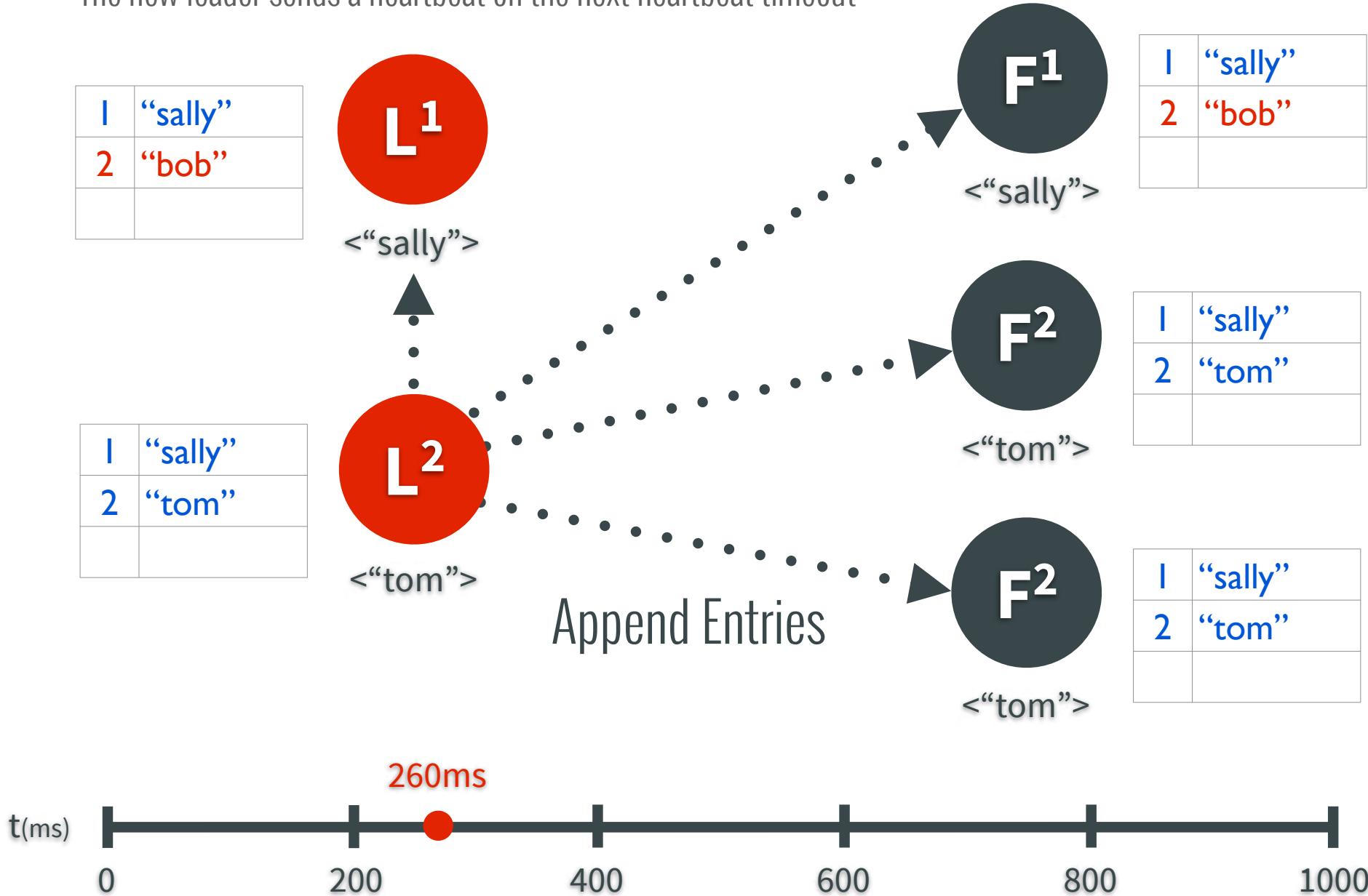
# Log Replication

The network recovers and there is no longer a partition



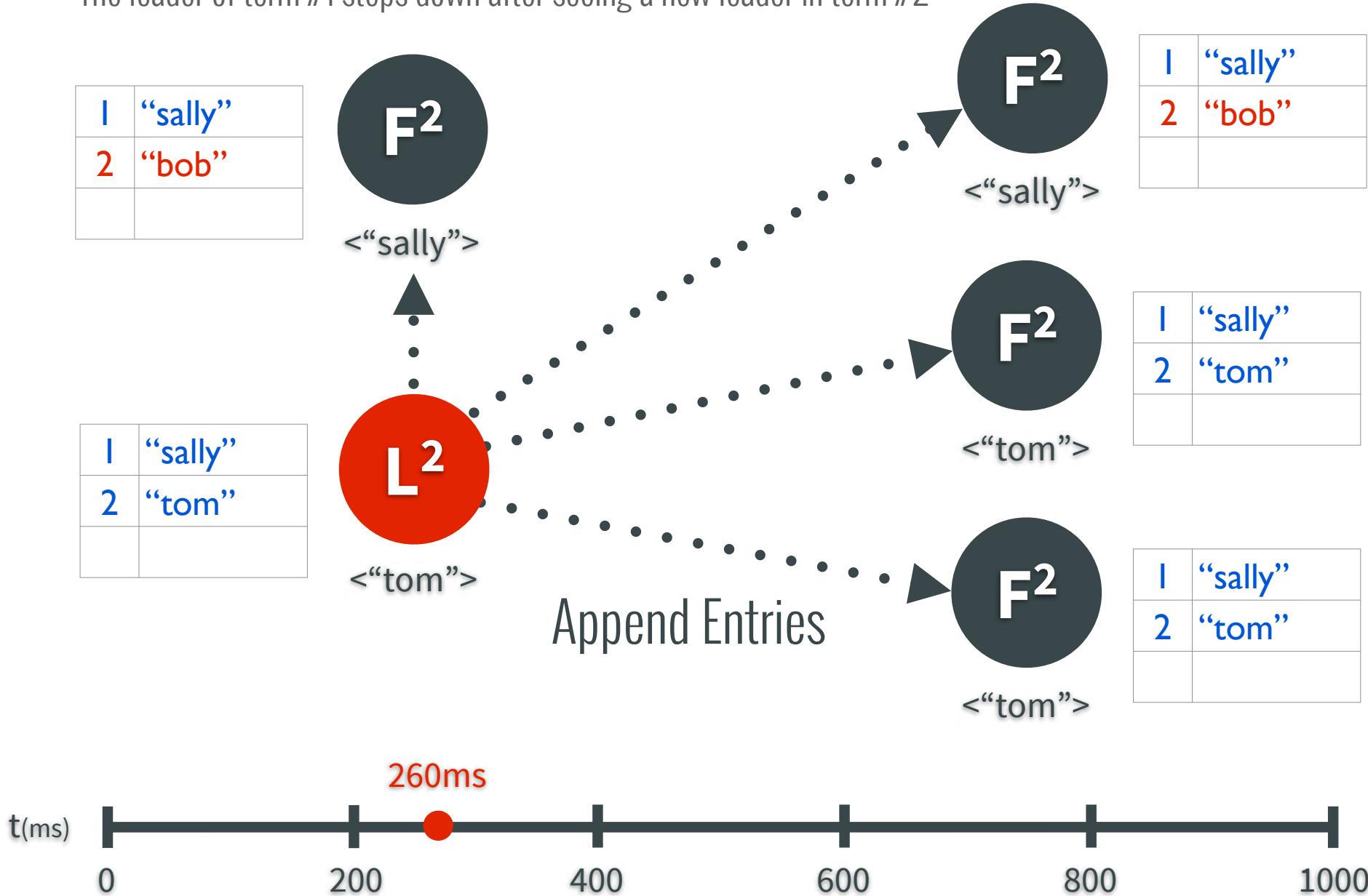
# Log Replication

The new leader sends a heartbeat on the next heartbeat timeout



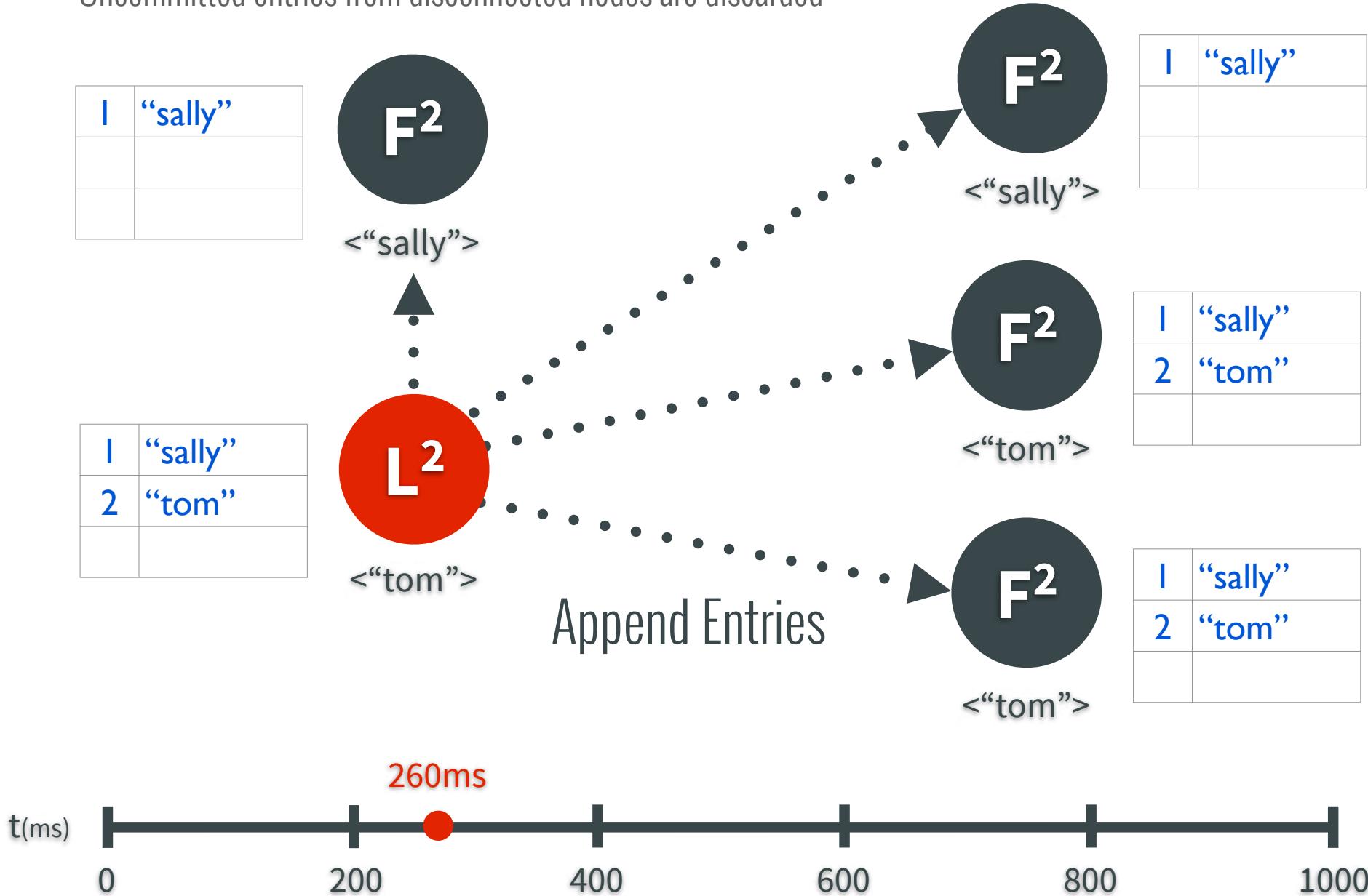
# Log Replication

The leader of term #1 steps down after seeing a new leader in term #2



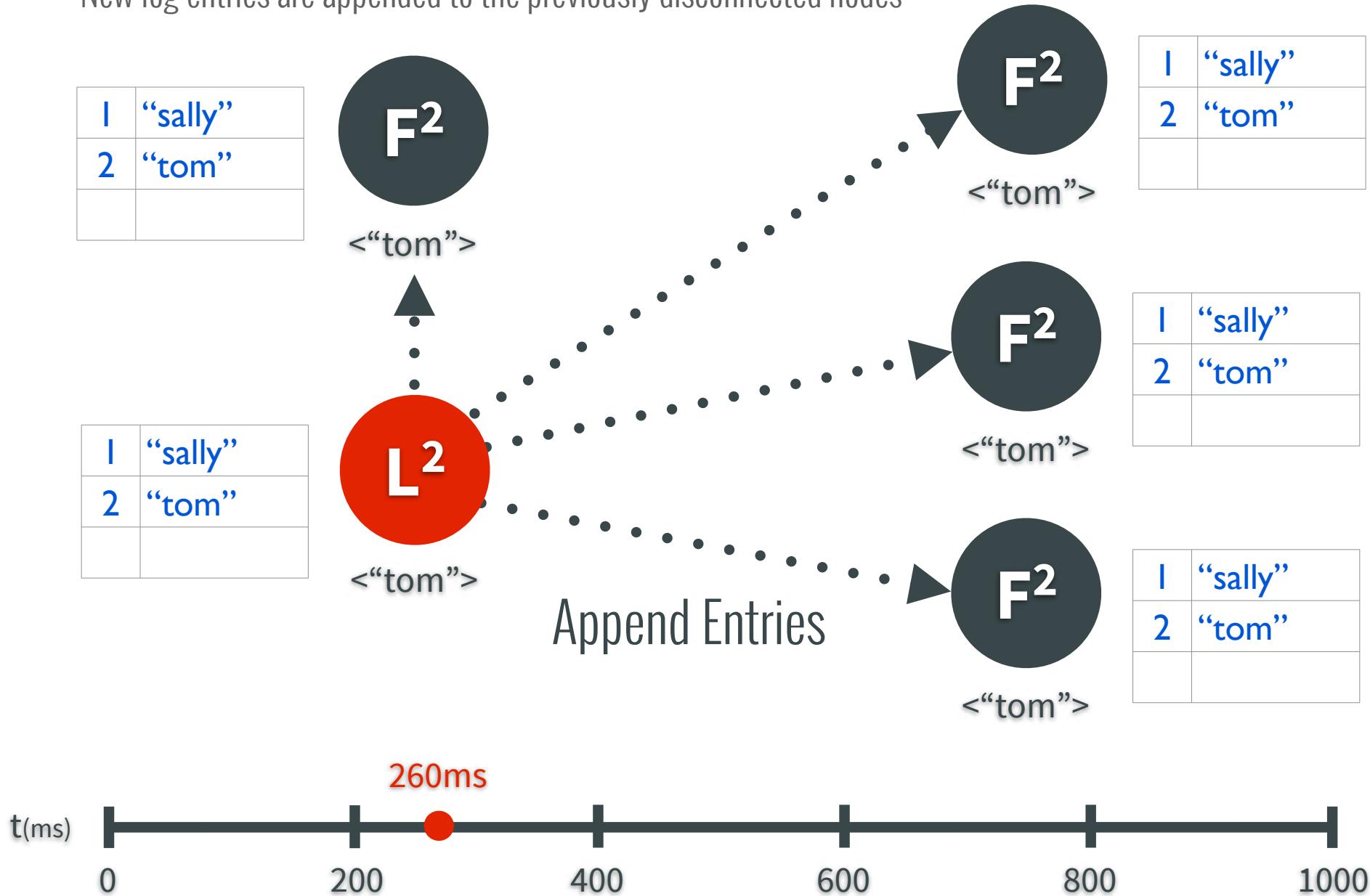
# Log Replication

Uncommitted entries from disconnected nodes are discarded



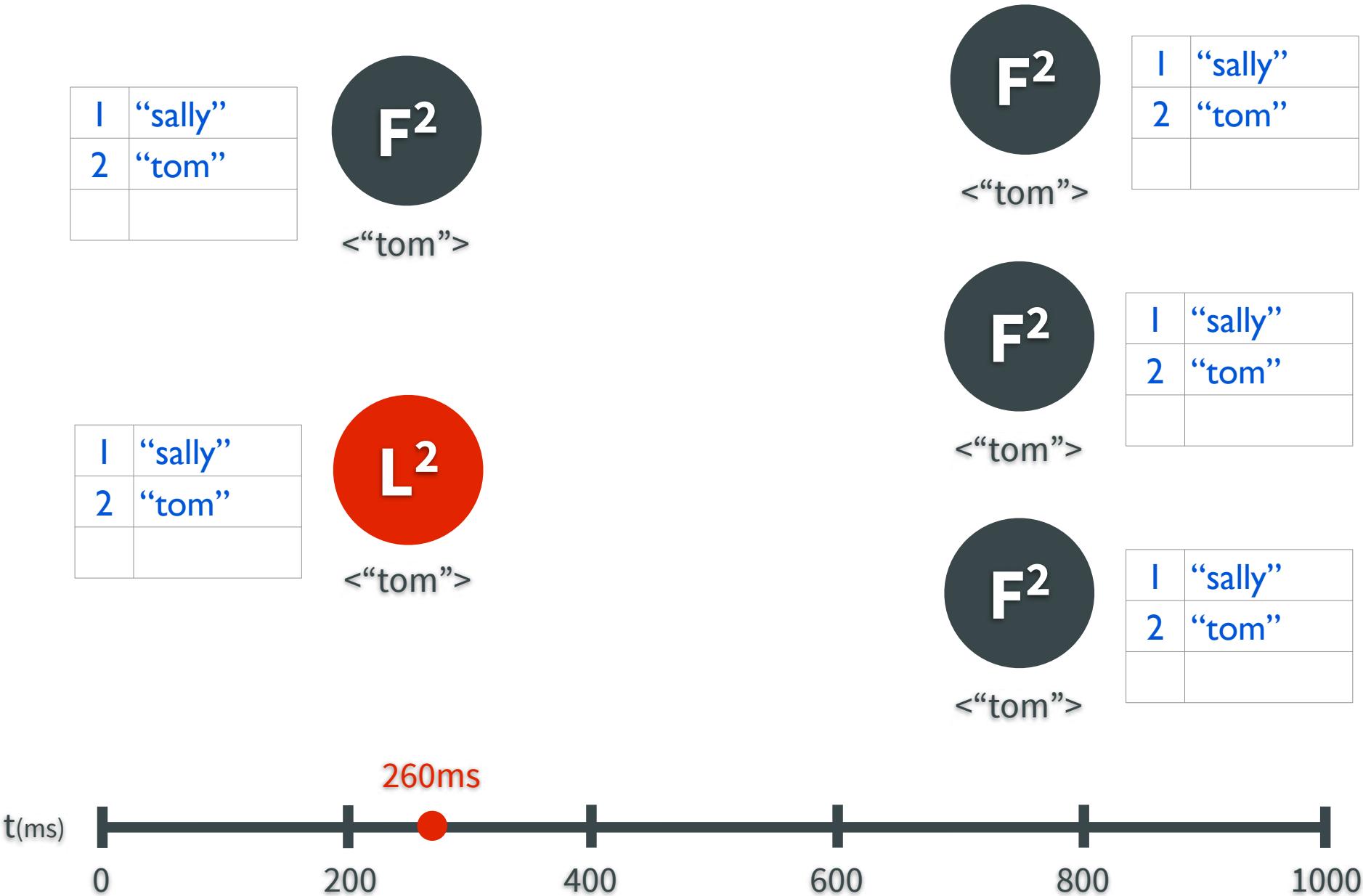
# Log Replication

New log entries are appended to the previously disconnected nodes

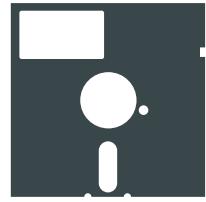


# Log Replication

---



# Log Compaction



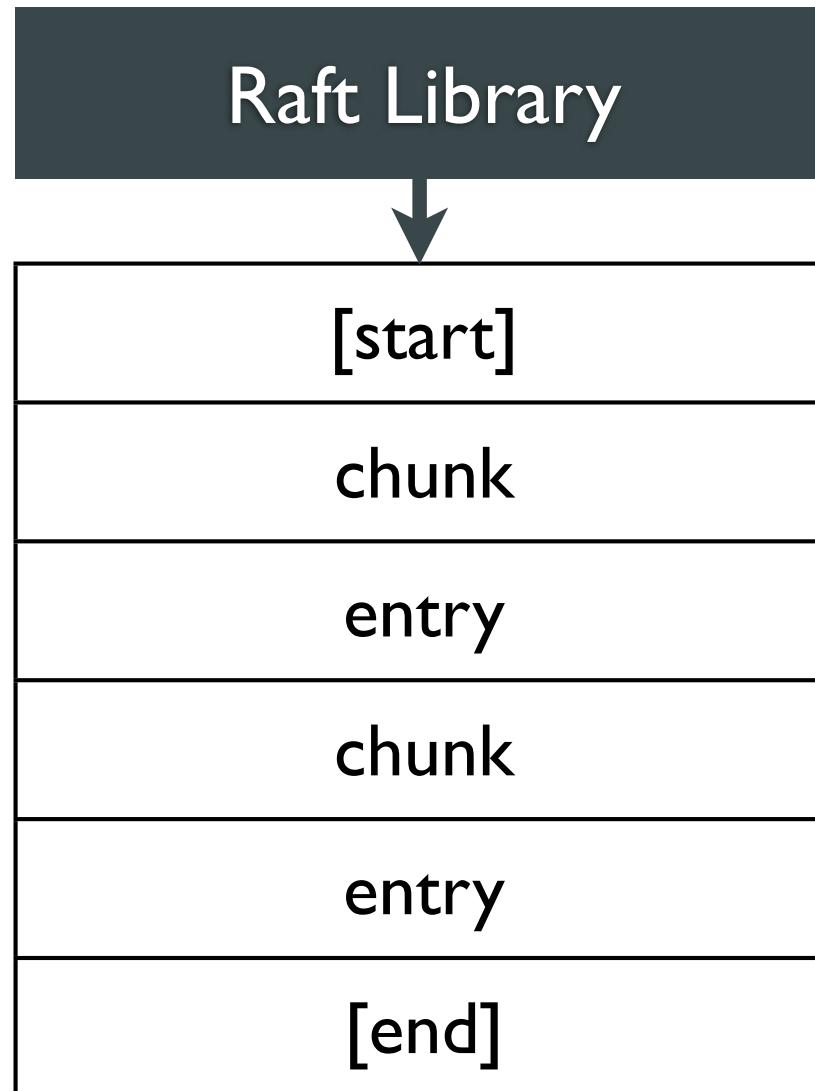
**Unbounded log can grow until  
there's no more disk**



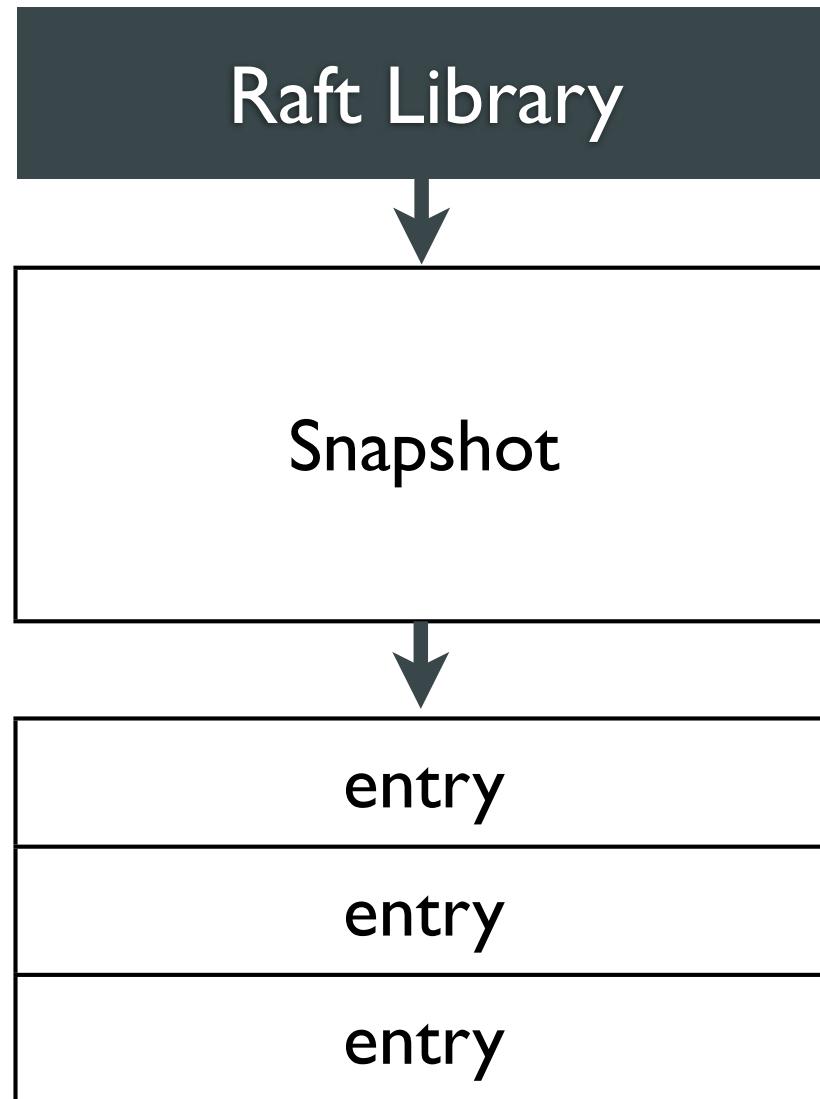
**Recovery time increases  
as log length increases**

# Three Log Compaction Strategies

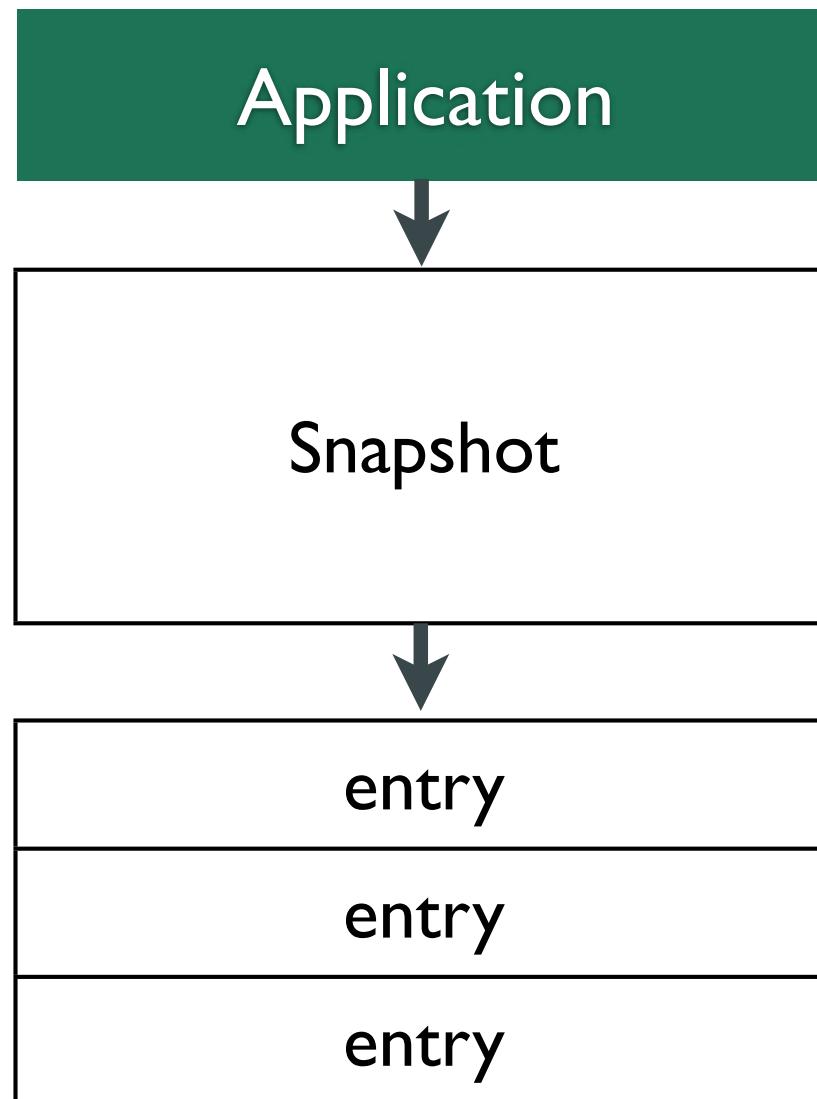
# #1: Leader-Initiated, Stored in Log



## #2: Leader-Initiated, Stored Externally



# #3: Independently-Initiated, Stored Externally



# Questions?

Twitter: @benbjohnson

GitHub: benbjohnson

[ben@skylandlabs.com](mailto:ben@skylandlabs.com)

# Image Attribution

Database designed by Sergey Shmidt from The Noun Project

Question designed by Greg Pabst from The Noun Project

Lock from The Noun Project

Floppy Disk designed by Mike Wirth from The Noun Project

Movie designed by Anna Weiss from The Noun Project

# Distributed Computing

A-04. Apache Zookeeper

# Apache Zookeeper

- Open source project
  - <https://zookeeper.apache.org/>
- Originally developed at Yahoo!
- Paper presented at the USENIX ATC 2010 conference



# A Coordination Service

- Group Membership
  - Add/remove workers/machines
- Leader election
- Dynamic Configuration
- Status Monitoring
- Queueing, barriers, critical sections & locks...

# How ZooKeeper Is Used

# Design Goals

- Multiple outstanding requests
- Read-intensive workload
  - For every *write*, 100s-1000s of *reads*
- General
- Reliable
- Easy to use

# Building Blocks

- 1)Wait-Free Architecture
- 2)Ordering
- 3)Change Events

# Wait-Free

- No locks or other primitives that **stop** a server
- No blocking in the implementation
- Simplifies the implementation
- No deadlocks
- Needs an alternative solution to wait for conditions

# Ordering

- Writes are **linearizable** (i.e., they are executed in the order they are performed)
- Reads are **Serializable** (i.e., you might read stale data)
  - Weaker than linearizable, it's ACID's isolation
- All operations on the same client will be serialized in FIFO order

# Change Events

- Clients can **request to be notified** of changes
- When a change happens, the client gets notified
- They get notification of a change before seeing the result

# Inspiration

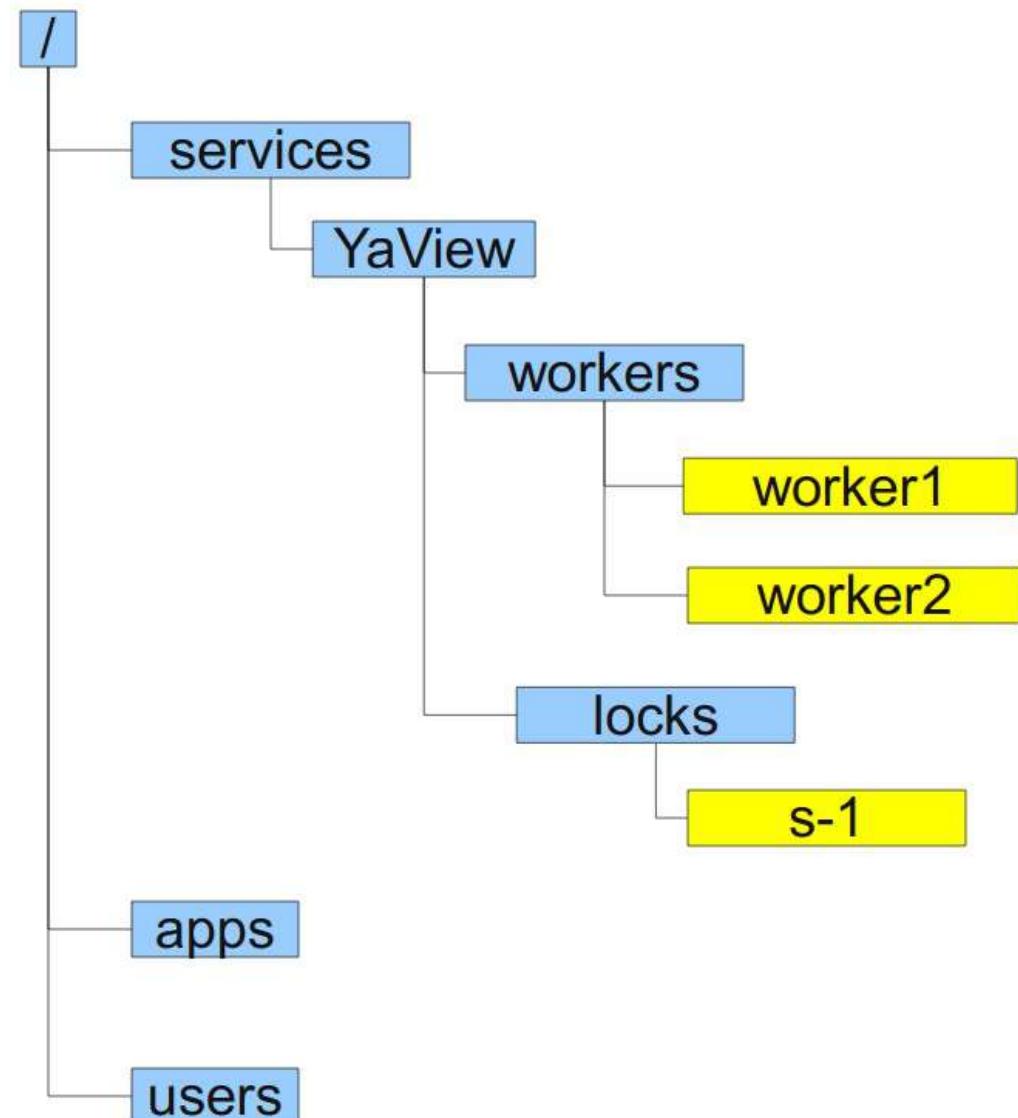
- A **distributed filesystem**
  - Zookeper designers have seen that their engineers were using the NFS filesystem to coordinate their applications
  - It “almost works”--when it fails, it’s ugly though :)
- Works like a filesystem for small pieces of data
  - Plus notification of changes
  - Minus partial read/writes

# API

- `create(path, data, acl, flags)`
- `delete(path, expectedVersion)`
- `setData(path, data, expectedVersion)`
- `getData(path, watch)`
- `exists(path, watch)`
- `getChildren(path, watch)`
- `void sync()`
- `setACL(path, acl, expectedVersion)`
- `getACL(path)`

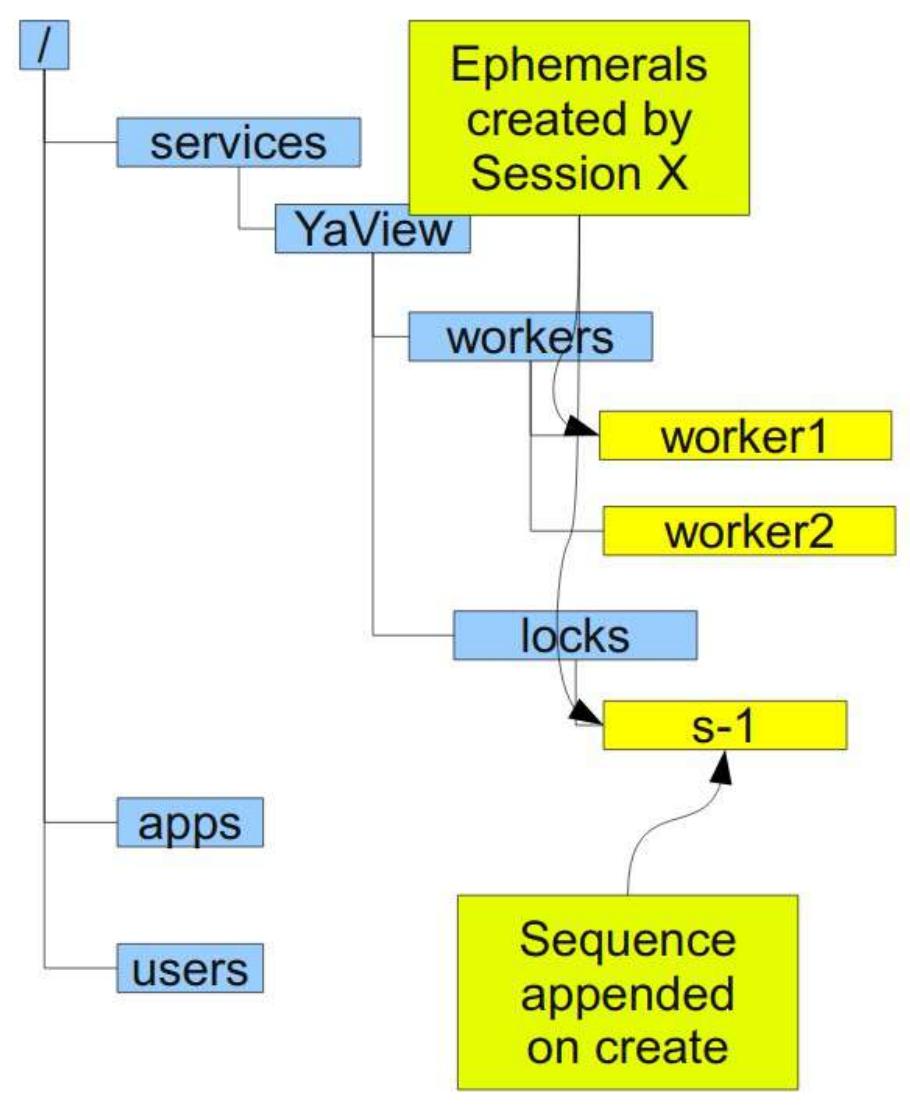
# Data Model

- Hierarchical namespace
- “Znodes” are like both files and directories: they have data and children
- Data is read and written in its entirety



# Create Flags

- **Ephemeral**: znode is deleted when creator fails
- **Sequence**: append a monotonically-increasing counter



# Configuration

- A worker starts and gets the configuration
  - `GetData("/myApp/config", watch=True)`
- Admins change the configuration
  - `setData("/myApp/config", newConf, expectedVersion=-1)`
- Workers get notified of the change and re-run `getData`

# Group Membership

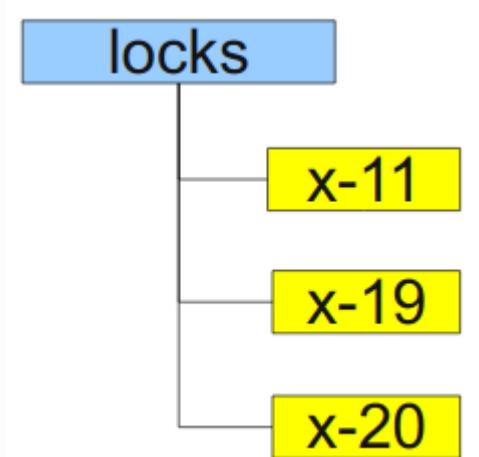
- A worker starts and gets registers itself in the group
  - `create("/myApp/workers/" + my_name, my_info, ephemeral=True)`
- List members
  - `getChildren("myApp/workers", watch=True)`  
worker1  
worker2

# Leader Election

- Check who's the current leader
  - `getData("/myApp/workers/leader", watch=True)`
- If successful, follow the leader
- Else, propose yourself as candidate
  - `create("/myApp/workers/leader", my_name, ephemeral=True)`
  - If successful, you're the leader; otherwise restart
- Since workers are watching changes for the leader, they'll know if the leader fails or changes

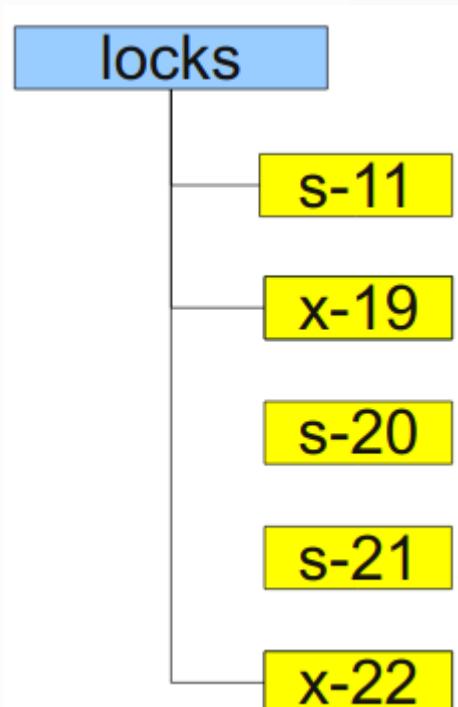
# (Exclusive) Locks

- `id = create("myApp/locks/x-", sequence=True, ephemeral=True)`
- `getChildren("myApp/locks/")`
- If `id` is the first child, lock is mine
- Otherwise:
  - `if exists(last child before id, watch=True)`
    - Wait for the event (when the previous owner releases the lock)
    - Else, return to `getChildren`
- Note that every node only watches the one in their front in the queue :)



# Shared Locks

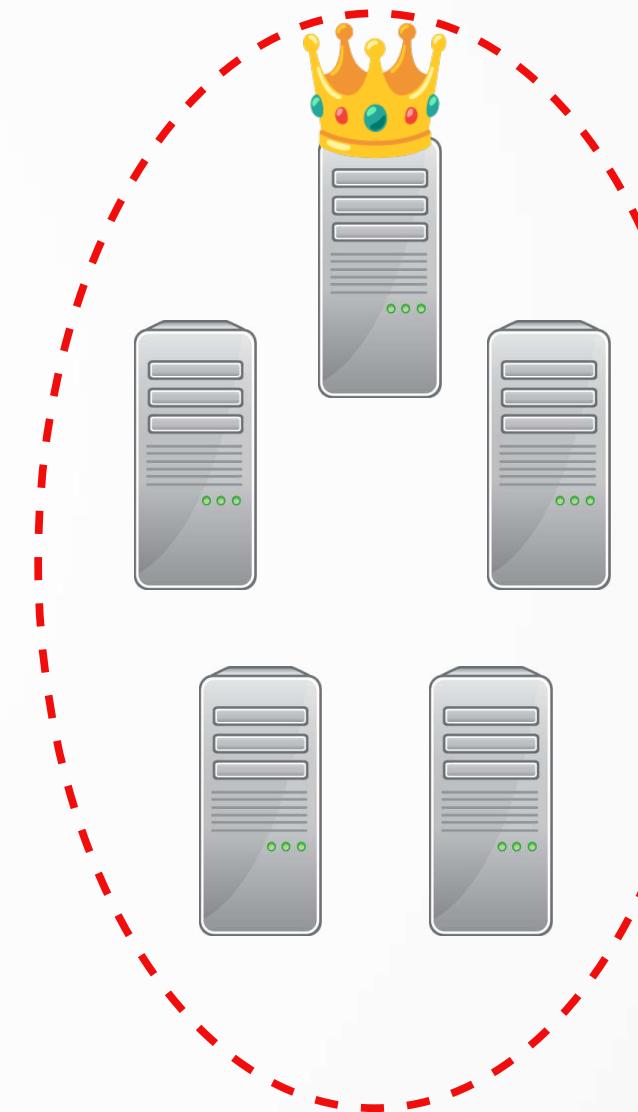
- `id = create("myApp/locks/s-", sequence=True, ephemeral=True)`
- `getChildren("myApp/locks/")`
- If no exclusive “x-” lock before id, go ahead
- Otherwise:
  - if `exists(last "x-" before id, watch=True)`
    - Wait for the event
  - Else, return to `getChildren`



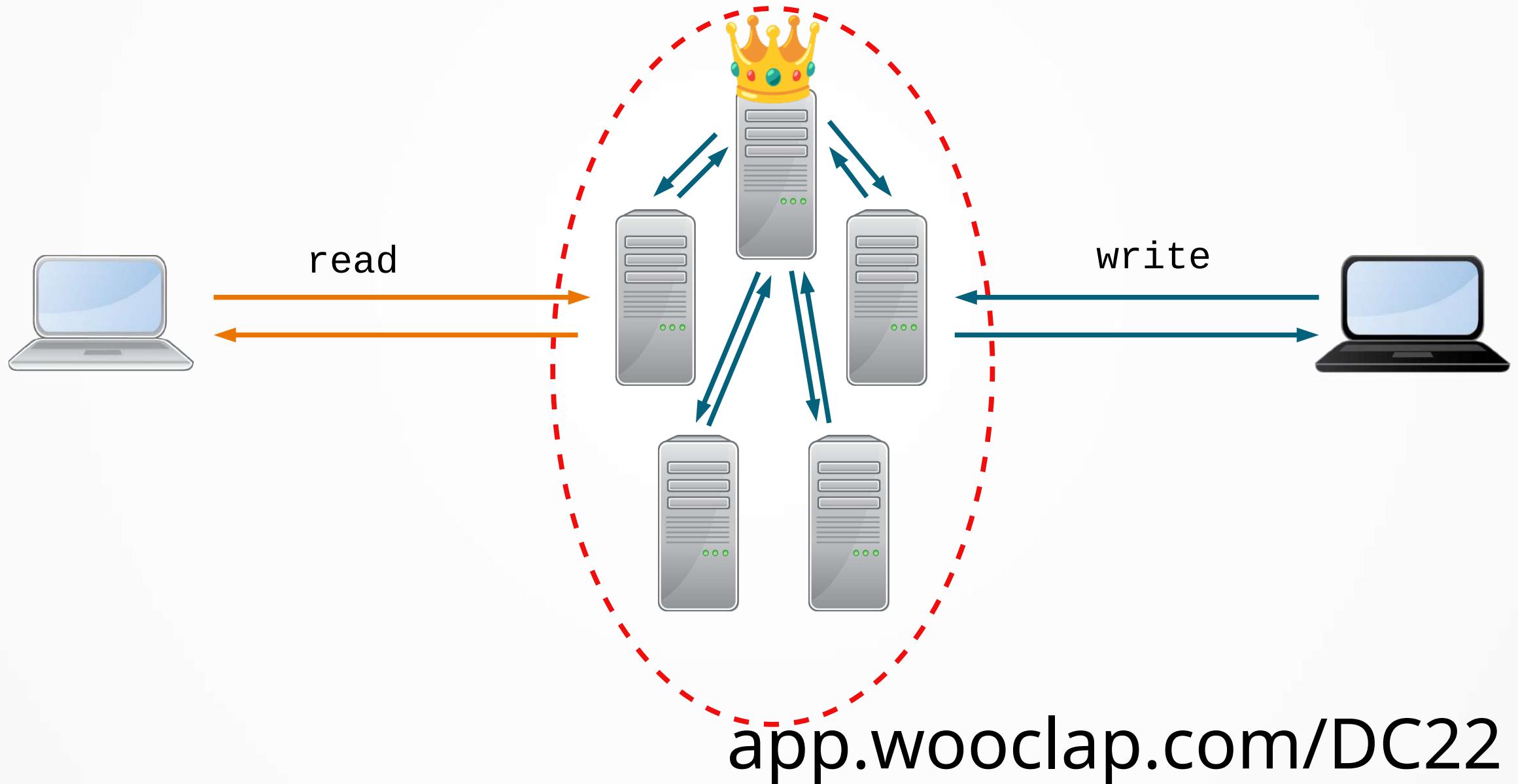
# How Zookeeper Is Implemented

# Architecture

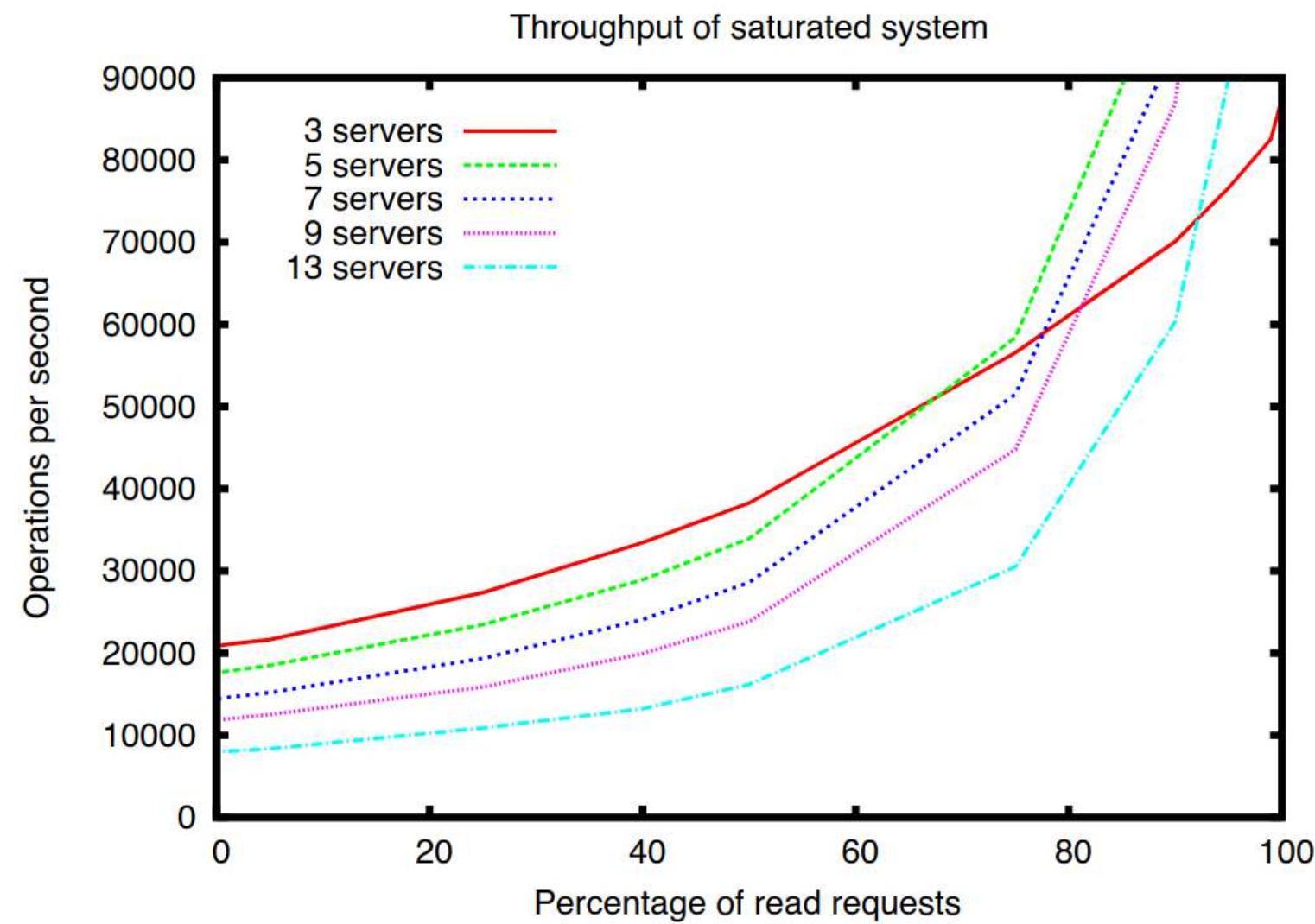
- All servers have a full copy of the state in memory
- Uses a consensus protocol that's similar to Raft
  - Actually, developed before it
- There's a leader
- Update is committed when a majority of servers saved the change
  - As we're used to, we need  $2m+1$  servers to tolerate  $m$  failures



# Reads and Writes



# Operations Per Second



# Distributed Computing

A-05. Google Cloud Spanner

# Google's Problem

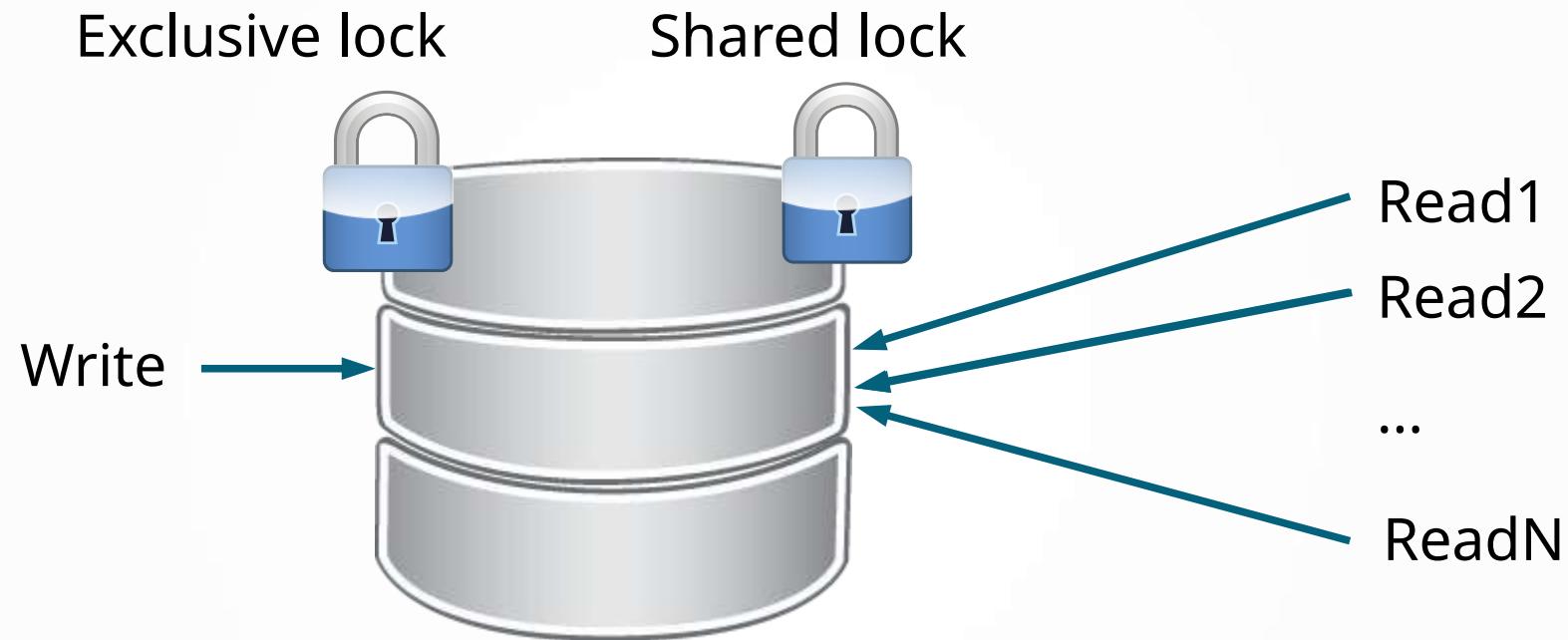
- Google engineers have **several datacenters** distributed across the world
  - **Sharded, replicated** database
- They wanted **linearization** (external consistency)—i.e., transactions are processed **in the same order they are seen in the real world**
- Example use case:
  - I remove an untrusted person X from my friend list
  - I post “my government is repressive”

# Google's Problem (2)

- If we could build a whole log of all transactions...
- But we can't **all** go through Paxos/Raft
  - Remember: each update is sent to all servers in a Paxos instance
- Solution: use Paxos locally and then **resort to clocks**
- Reference: paper, slides & video

# Philosophy

# Locks?



- One write at a time **in the whole system**
- When you're writing somewhere, **you can't read**
- **Unreasonable** for a huge system

# Historical View

- Pieces of information are annotated with time
  - We know **when they got in the system**
  - If they were canceled, **we know when**
- To do this, we need a **super-precise clock!**
  - Typical Internet latencies can be up to **hundreds of milliseconds**
- As we'll see, Spanner will **wait** to handle uncertainties in the clock
  - It's key to **drive uncertainties down**

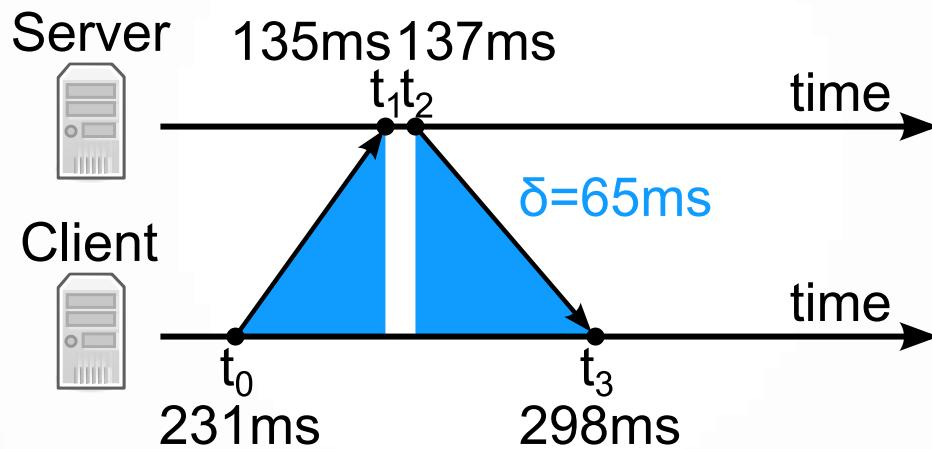
# **What's the Time?**

# TrueTime

- A system to evaluate time with **high reliability**
- Idea here: **bounded uncertainty**
- TrueTime .now( ) returns an interval: *[earliest, latest]*
  - Meaning: time is **between earliest and latest**
- Each datacenter has a set of **time master** machines
  - Regular ones, equipped with GPS
  - **Armageddon masters**, with atomic clocks (!!)

# Talking to a Time Master (1)

- Not all details are given; assuming an approach similar to the **Network Time Protocol (NTP)**:



- Time offset  $\theta = \frac{(t_1 - t_0) + (t_2 - t_3)}{2}$
- Round-trip delay  $\delta = (t_3 - t_0) - (t_2 - t_1)$

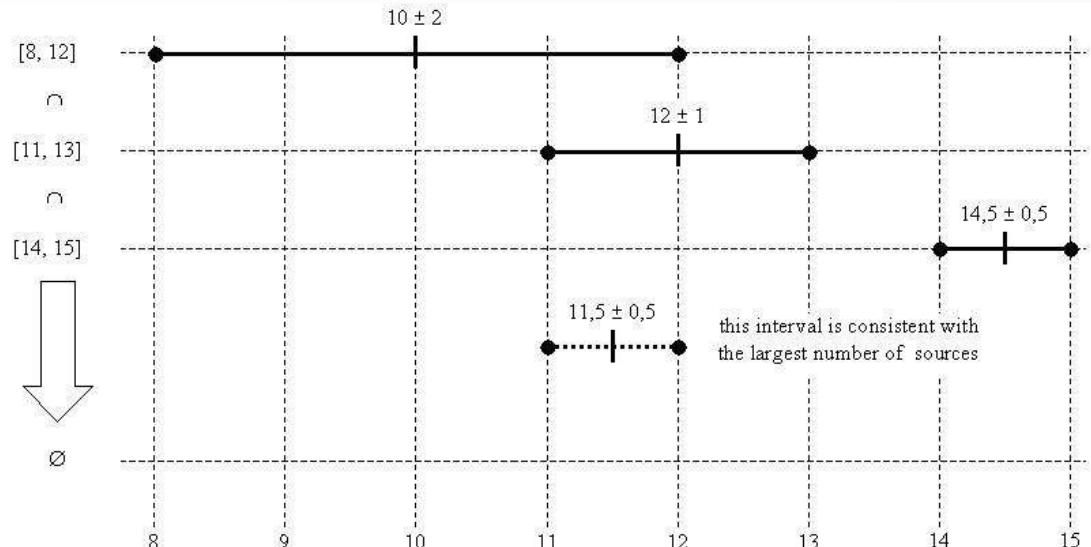
# Talking to a Time Master (2)

- Measurement as described before is repeated, and statistics are extracted
- The output is an interval  $t \pm \varepsilon = [\text{earliest}, \text{latest}]$
- One of the time masters could be wrong: to weed out “liars”, an algorithm is needed
- Problem: given  $n$   $[a_i, b_i]$  intervals, find the sub-interval that is consistent with most cases
  - (In case of a tie, return any of them)

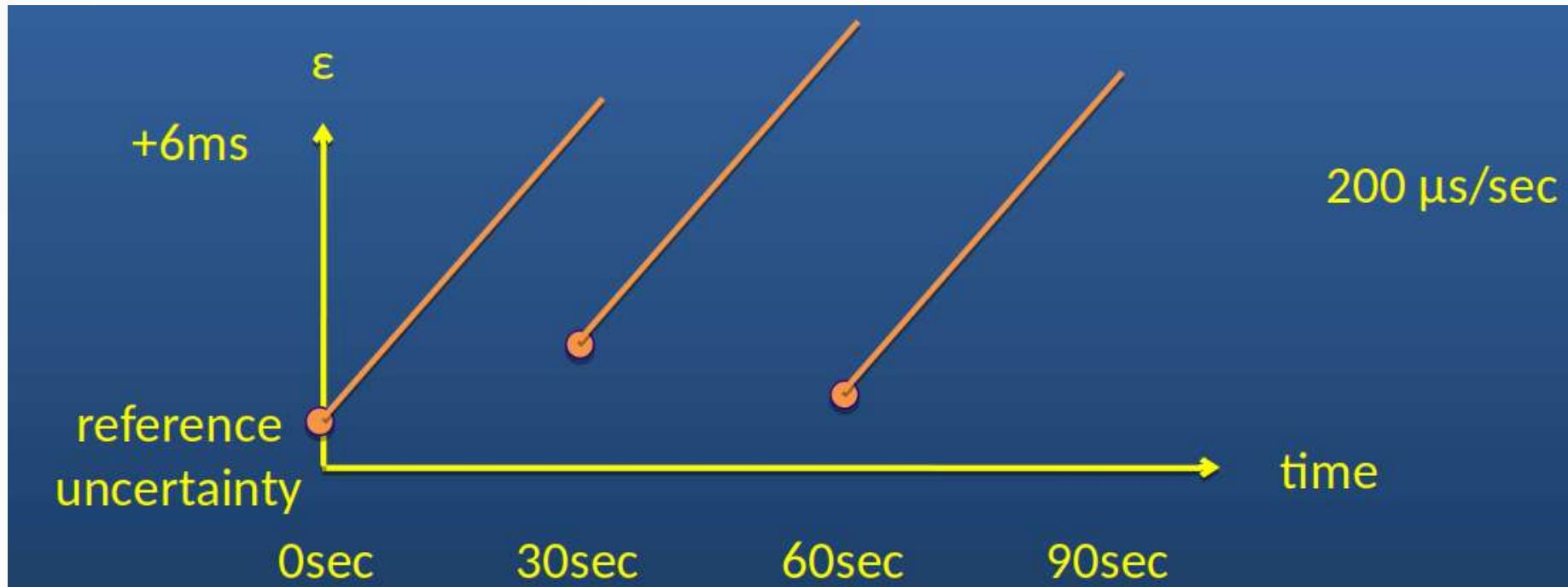
# Intersection Algorithm

(image from [Wikipedia](#), CC BY SA 3.0)

- Create a list containing  $(a, +1)$  and  $(b, -1)$  pairs for each  $[a, b]$  interval and sort them by the first element
- Let's call these values  $(v_i, d_i)$ ;  $d_i \in \{-1, 1\}$
- Compute the cumulative sums  $s_i$  of all  $d_j$  values where  $j \leq i$ 
  - It's the number of intervals overlapping in  $[v_i, v_{i+1}]$
- Find the maximum value  $s_M$ ; the result will be  $[v_M, v_{M+1}]$



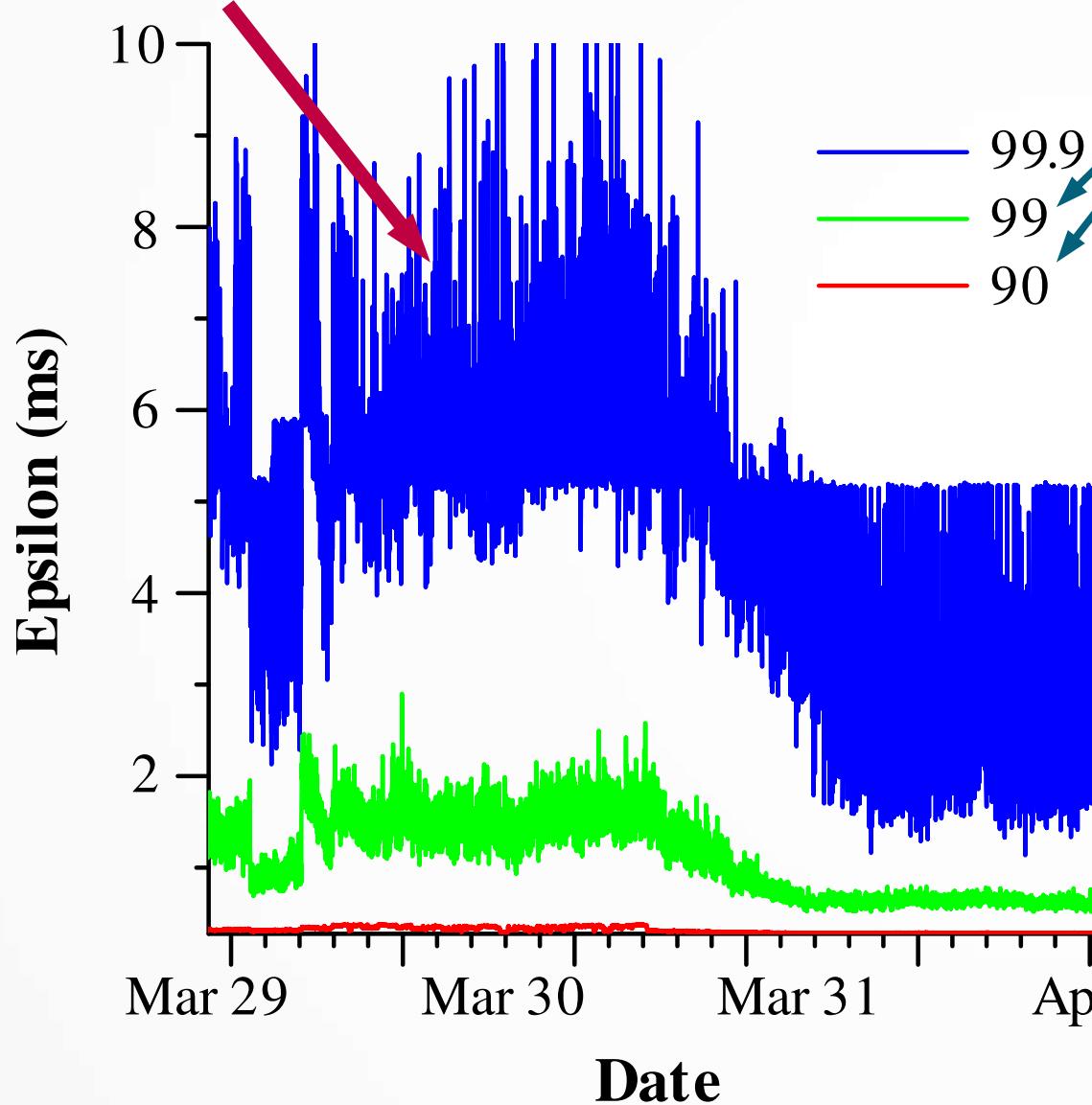
# Catering for Local Clock Error



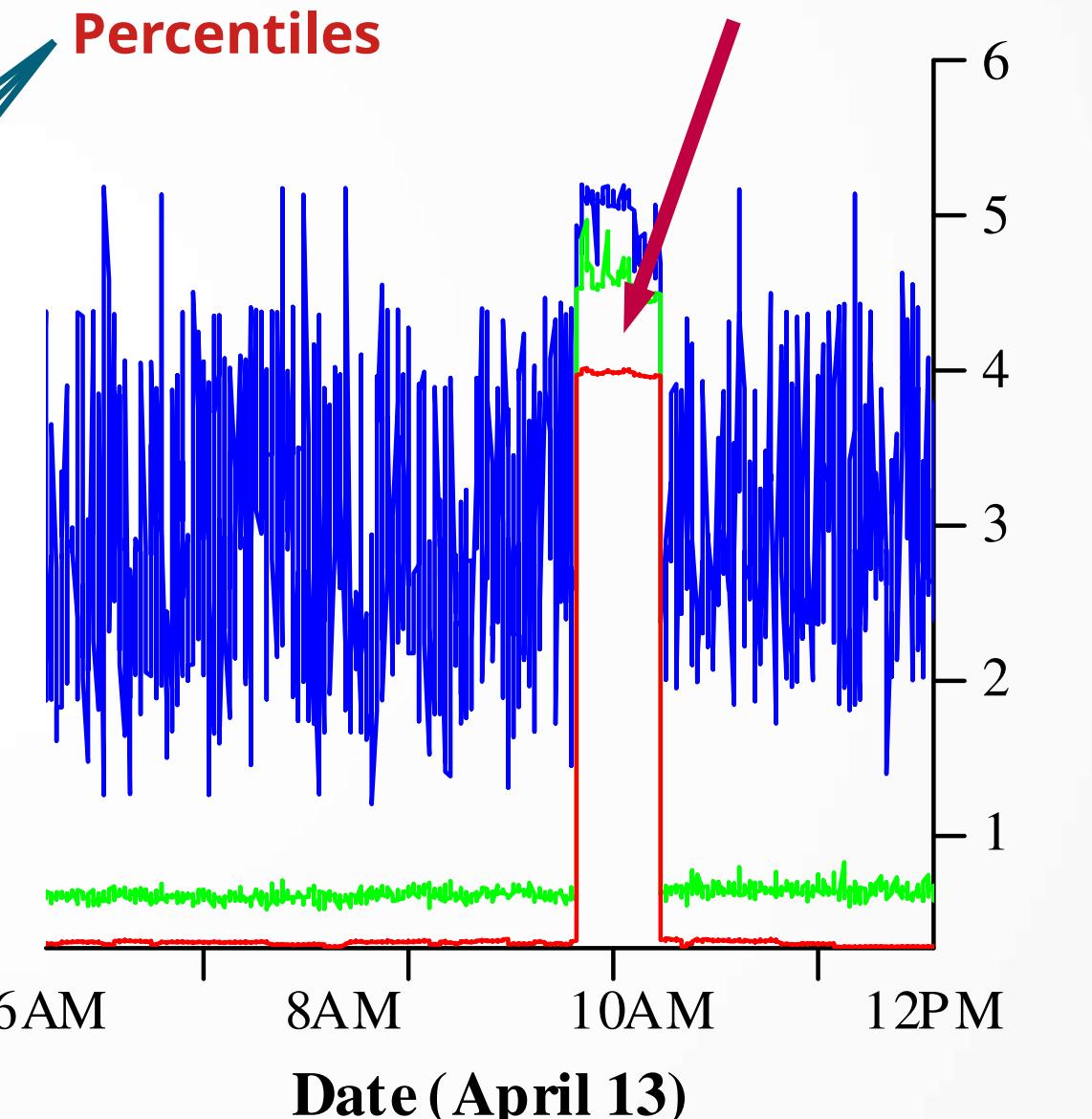
- After the clock is synchronized, the uncertainty grows according to worst-case assumptions on computers' clock drift
- Clocks that “go crazy” are very rare (6 times less than CPUs that do)

# TrueTime Precision

Network congestion



Time server maintenance



# Making Transactions Linearizable

# Version Management

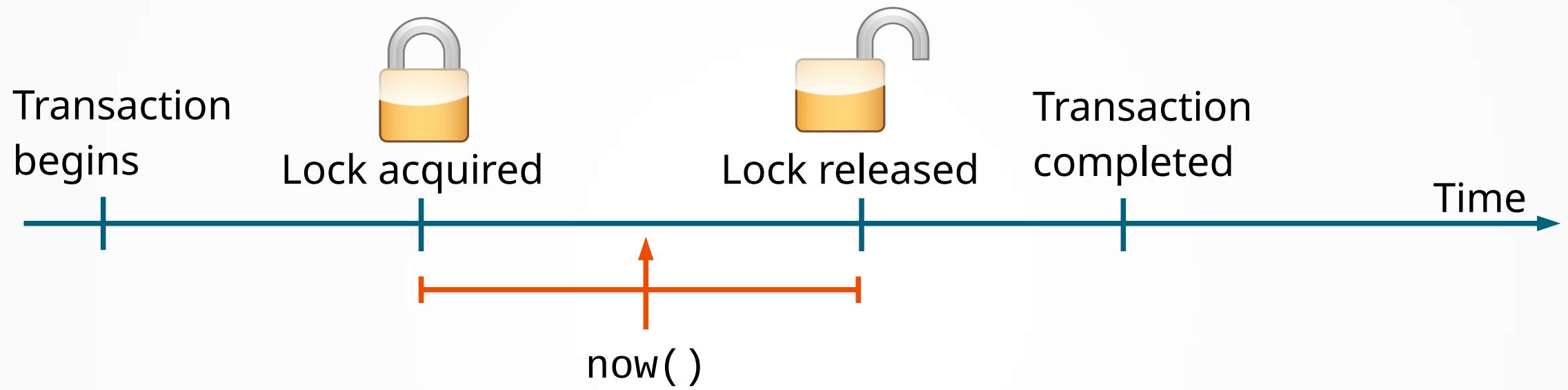
Time	My friends	X's friends	My posts
4	[X]	[me]	
8	[]	[]	
15			["Government bad"]

- X will never read my post:
  - The transactions that removes my friend happens before my complaint
  - A read timestamped before 8 won't see the post
  - With a read timestamped after 15, X won't see my profile
- Even if the write transactions happen in completely different clusters...

# Data Model

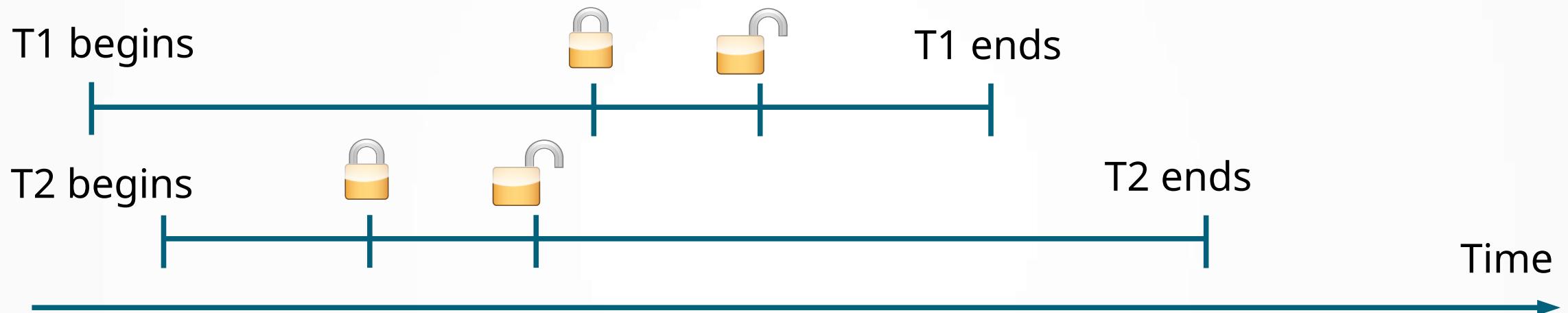
- A key-value store
  - I.e., lookup the value for a given string, as if it was a huge hashtable
  - Holds data like
$$(\text{key:string}, \text{ timestamp:int64}) \rightarrow \text{string}$$
- Nodes responsible of a key in **multiple continents**
  - Use Paxos to get consensus
- Allows asking the value **at a given moment in time**
- SQL-like semantics added afterwards

# Assigning Timestamps to Writes



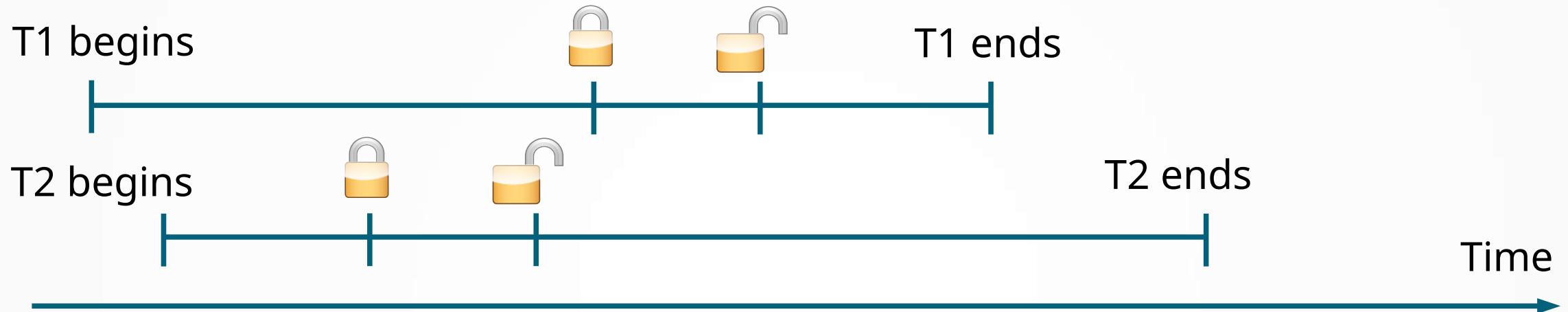
- The timestamp used is **any moment when the lock is acquired**

# Conflicting Transactions



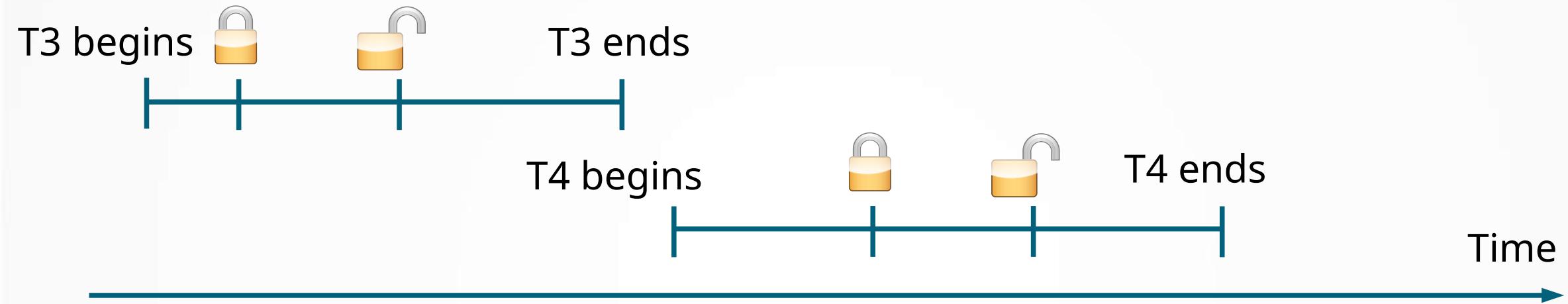
- Two transactions that touch conflicting data will have disjoint locking intervals (that's how locks work!)
- Here, T2 will see data “before” T1, and will be timestamped before it!

# Conflicting Transactions



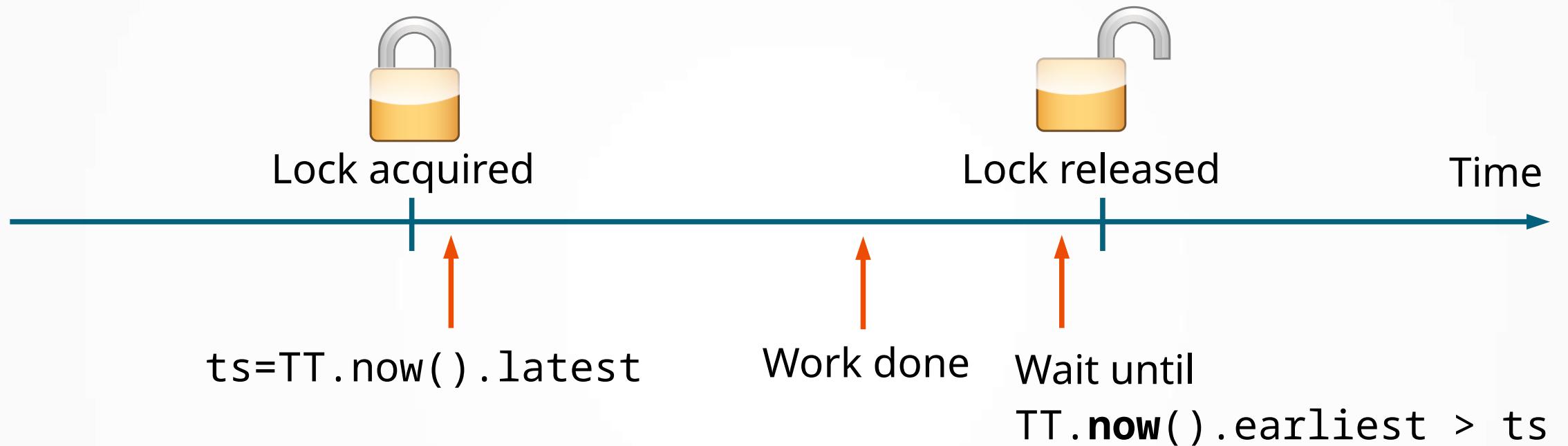
- Two transactions that touch conflicting data will have disjoint locking intervals (that's how locks work!)
- Here, T2 will see data “before” T1, and will be timestamped before it!

# Non-Conflicting Transactions



- If a transaction is executed before another, it will be timestamped before it

# TrueTime to Assign Timestamps



- If the system is afraid of finishing before  $ts$ , it just **waits** until there's no doubt
- If uncertainty on time is too large, **the system gets slowed down**
- Clock uncertainty should be smaller than a transaction length
  - Paxos intercontinental consensus latency: 100s of ms
  - TrueTime latency: generally <10 ms

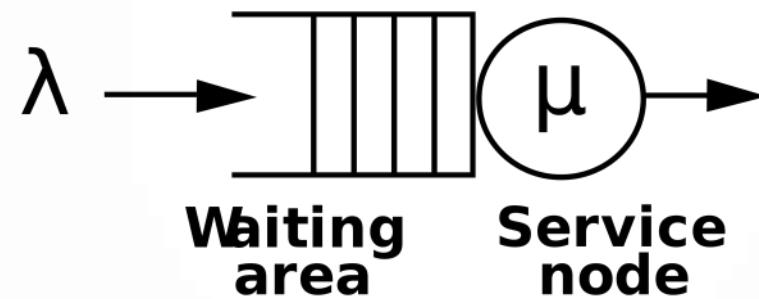
# Distributed Computing

A-06. Queueing and Scheduling

# Queues: The Simplest Model

# A Little Bit of Queueing Theory

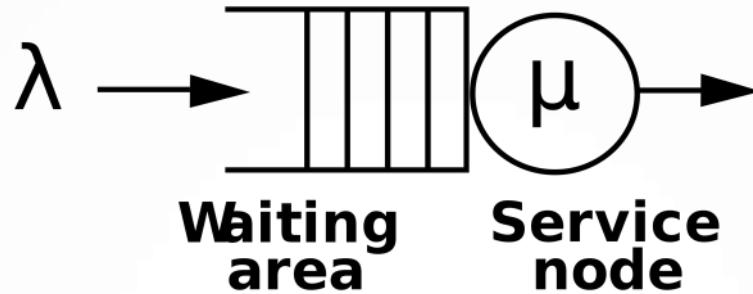
- Let's consider a simple case, for the moment: we have a single server and a queue of **jobs** it has to serve (web pages, computations, ...)



(image by Tsaitgast on Wikipedia, CC-BY-SA 3.0 license)

- $\lambda$  and  $\mu$  are the average frequencies (**rates**) at which jobs respectively join the queue and leave the system when they are complete: in time  $dt$ , the probability a job arrives or leaves when being served are respectively  $\lambda dt$  and  $\mu dt$
- Jobs are served in a First-In-First-Out fashion

# The Simplest Model



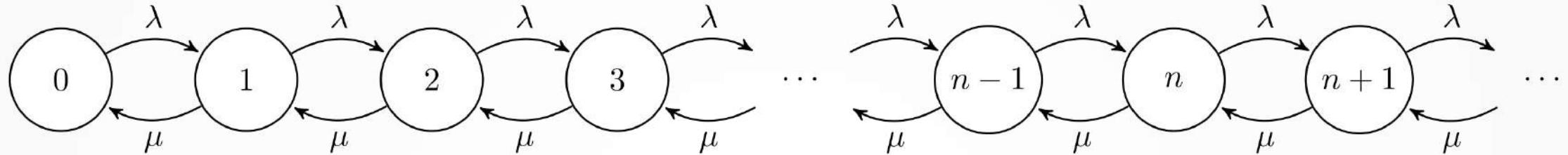
Reference: Harrison & Patel, 1992, chapters 4-5

- This is called **M/M/1 (Kendall notation)** queue because
  - Jobs arrive in a **memoryless** fashion: no matter what happened until now, the probability of seeing a new job in a time unit **never changes**
  - Jobs are served in a **memoryless** fashion: the probability of a job finishing does not depend on how long it's been served, or anything else
  - Just **one** server

# Wait--Why This Model?

- “All models are wrong, but some are useful” (George Box)
- Real systems are **not** like this, but some of the insight **does apply** to real-world use cases
  - The memoryless property makes it **much easier** to derive closed-form formulas
  - (Some of) the insight we get will be **useful**
  - We can compare & contrast with **simulations**
  - And we need to verify whether simulations are representative too

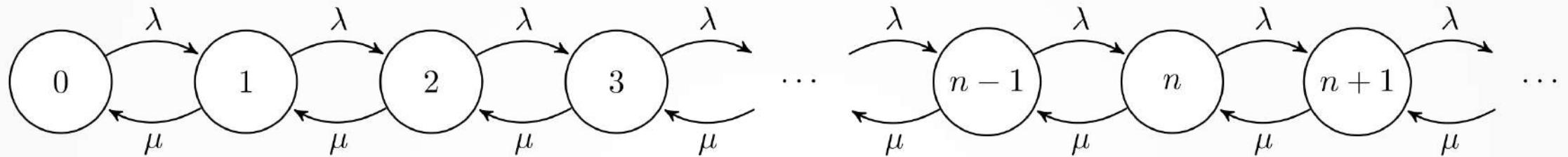
# Let's Analyze M/M/1



(image by Gareth Jones on Wikipedia, CC-BY-SA 3.0 license)

- How does this model behave **on the long run?**
  - What is the probability of having  $x$  jobs in the queue?
  - What is the **amount of time** a typical job will wait before being served?
- We will start by looking at the first question, looking for an **equilibrium probability distribution**

# M/M/1 Equilibrium



- To be an equilibrium we need
    - $\lambda < \mu$ , otherwise the number of elements in the queue will keep growing
    - That in any moment the probability of moving from state  $i$  to  $i+1$  is the same of moving in the opposite direction:

$$\lambda p_i dt = \mu p_{i+1} dt$$

$$p_{i+1} = \frac{\lambda}{\mu} p_i$$

# Some Easy Algebra

- Let's simplify and say  $\mu=1$  (just change the time unit)

$$p_{i+1} = \lambda p_i$$

hence

$$p_1 = \lambda p_0, p_2 = \lambda^2 p_0, \dots, p_i = \lambda^i p_0$$

- Since this is a probability distribution, its sum must be one. We can then solve everything:

$$\sum_{i=0}^{\infty} \lambda^i p_0 = 1; \frac{1}{1-\lambda} p_0 = 1; p_0 = 1 - \lambda; p_i = (1 - \lambda) \lambda^i$$

- The average queue length is

$$L = \sum_{i=0}^{\infty} i p_i = (1 - \lambda) \sum_{i=0}^{\infty} (i \lambda^i) = (1 - \lambda) \frac{\lambda}{(1 - \lambda)^2} = \frac{\lambda}{1 - \lambda}$$

# Little's Law

- Beautifully simple:  $L$  equals the **average time spent in the system  $W$**  times the arrival rate:

$$L = \lambda W$$

- Why? Consider this: if a time unit spent in the system by a job “costs” 1€, then jobs would spend  $W\text{€}$  on average.
- In an average time unit, the system will collect  $L\text{€}$  (because on average  $L$  jobs are in queue); in equilibrium and on average  $\lambda$  jobs will arrive and leave the system, spending a total of  $\lambda W$ .

# So We've Got Our Result

- The average time spent in the system for an M/M/1 FIFO queue is

$$W = \frac{L}{\lambda} = \frac{\left(\frac{\lambda}{1-\lambda}\right)}{\lambda} = \frac{1}{1-\lambda}$$

# Multiple Servers

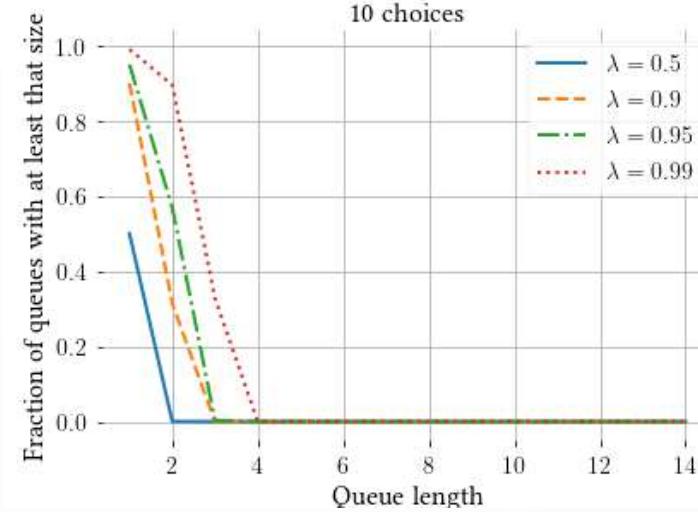
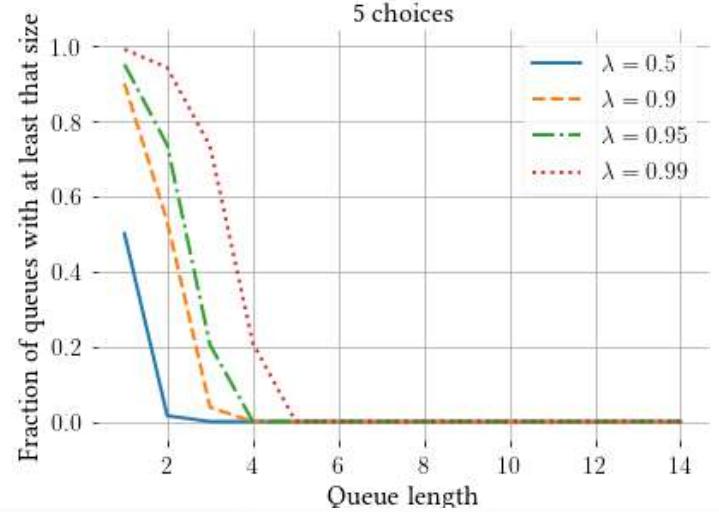
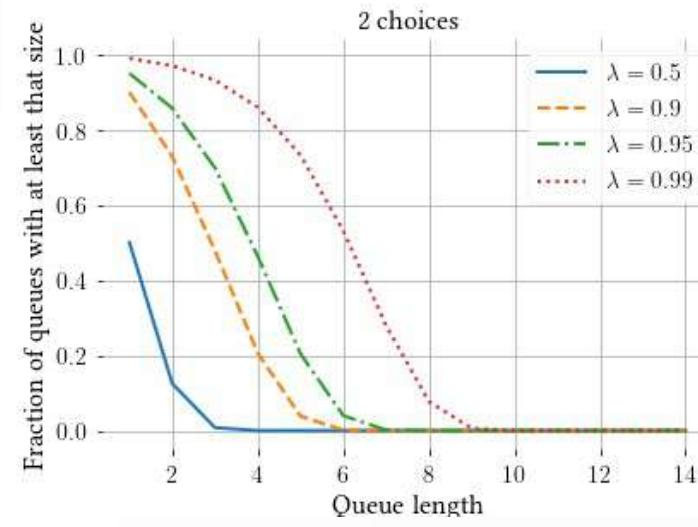
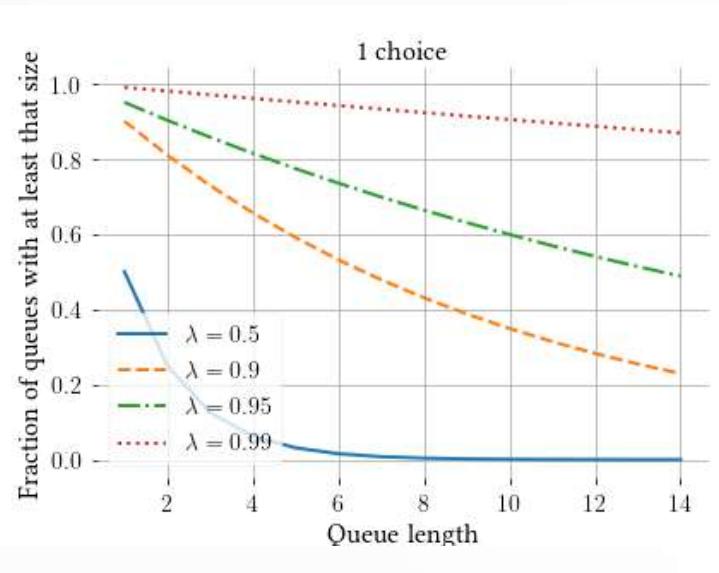
# Multi-Server Version

- Rather than a single server, there are  $n$  servers
- You have a **load balancer** to divide the load between servers
- How to assign load to servers?
  - If randomly, you can handle a load of  $n\lambda$  with the performance of a single server having load  $\lambda$
  - Can we do better?
  - We can assign jobs to the **least loaded server!**

# Supermarket Queueing

- Idea: rather than querying  $n$  servers to find out their queue length, just ask a small number  $d$  of them and assign to the shortest queue
- Mitzenmacher (2001) finds that the fraction of queues with at least  $I$  jobs drops from  $\lambda^i$  to  $\lambda^{\frac{d^i - 1}{d - 1}}$

# Theoretical Queue Length



# Beyond First-In-First-Out

# Preemptive Policies

- In the real world, you very often have **many very small jobs** and **a few extremely large ones**
  - The memoryless property doesn't apply!
- You can **preempt** running jobs: pause them and resume them later
- Practical, widely adopted, solution: **round-robin** scheduling, where you run a job for a given timeslot and then pass the turn to another one
  - If  $n$  jobs are in the queue, each proceeds at speed  $1/n$
  - Often adapted with priorities reflecting job importance

# Size-Based Scheduling

- If you know **for how long** a job will be running (its **size**), you can do something smarter
  - E.g., you may know how long an algorithm will take or how big is the file you're serving
- To minimize the average time spent **W**, the optimal policy is **Shortest Remaining Processing Time** (SRPT), which always serves the job needing the least work to complete
  - Theoretical possibility of **starvation**: large jobs are never served —not really relevant in practice (**Harchol-Balter et al., 2003**)
  - Sketch of optimality proof at the blackboard (**Schrage, 1968**)

# Errors in Size Estimation

- SRPT may behave catastrophically if jobs' size information is incorrect
  - If large jobs are underestimated, their “remaining processing time” goes below 0 and “blocks” the system until they are finished
  - Particularly problematic with very diverse job sizes
- Simple solution: forget about the “remaining” processing time and schedule first the jobs with smallest estimated size!
  - Shown to work well both in simulations and theory
  - Assuming estimation error is proportional to real size

# Conclusions

# Evaluating Distributed Systems

- Mathematical models
  - Good: “hard truths” for the modeled world
  - Bad: needs simplifications: the modeled world is not the real world
- Experiments & measurements on real systems
  - Good: evaluating the “real thing”
  - Bad: overly focused on implementation details, expensive, limited
- Simulations
  - Good: (to some extent) scalable, cheap
  - Bad: trade-off between scalability and precision

# Scheduling For Your System

- Apply the “supermarket” technique, scheduling on the machine with the least jobs in queue
- Beware of FIFO! Consider preemption and round-robin scheduling
- If half-decent size estimates are possible, consider prioritizing shortest jobs
  - Make sure mistakes won’t completely stop everything else
  - If malicious behavior is possible, consider its impact though

# Distributed Computing

A-07. Discrete Event Simulations

# Simulating A System

- We've seen an example of the type of analysis needed to evaluate how a system will behave
- To be theoretically tractable, **simplifying assumptions** must be taken
  - E.g., memoryless/exponential distributions, single server
- We want to look at what happens when we **drop** those assumptions

# Simulations

- To evaluate distributed systems:
  - Mathematical models
    - Good: “hard truths” for the modeled world
    - Bad: needs simplifications: the modeled world is not the real world
  - Experiments & measurements on real systems
    - Good: evaluating the “real thing”
    - Bad: overly focused on implementation details, expensive, limited
  - **Simulations**
    - Good: (to some extent) scalable, cheap
    - Bad: trade-off between scalability and precision

# Discrete Event Simulation

- A queue of events, sorted by the time at which they happen
- A system state
- Iteratively:
  - Select the first event in the queue
  - Update the state
  - Add any new event triggered by this one to the queue
- Repeat until the queue is empty (or maybe a special STOP event is reached)

# Our Code

- Plenty of stuff can be done
- SSE students will extend it and write an assignment
- `discrete_event_sim.py`: library file
- `sir.py`: example complete simulation code
- `mmn_queue.py`: M/M/n simulation to complete

# Goals

- Do what you can in the lab hours
  - 1) Complete `discrete_event_sim.py`
  - 2) Complete the M/M/1 FIFO simulation
  - 3) If you have time, try extending it to M/M/n...
  - 4) ...and writing Supermarket queueing...
  - 5) ...and reproducing the plots in the slides

# Queues

# Hints (1)

- Event queue:
  - Use an efficient data structure (e.g., a priority queue like a binary heap)
  - Pre-populate it with all job arrivals you want to simulate
  - Or, every time you process a job arrival, insert the new one in the queue
  - Remember that job interarrival time is exponential with mean  $1/\lambda$
- State (use a **single** instance of the Simulation class):
  - We'll need arrival and completion time for each job
  - We'll need the FIFO queue(s) on each scheduler

# Hints (2)

- Event processing:
  - Job  $i$  arrives at time  $t$ 
    - We mark this information, e.g., in a mapping that gives us the arrival time of each running job
    - We add the job to the server's FIFO queue
    - We generate an exponentially distributed random variable  $X$  (mean  $\mu = 1$ ); if the FIFO queue is empty and add to the event queue the completion of job  $i$  at time  $t+X$
  - Job  $i$  terminates at time  $t$ 
    - Remove the job from the queue
    - Mark the completion time of  $i$
    - If the queue is not empty, add the completion of the next job at time  $t+X$

# Hints (3)

- Getting queue lengths
  - Create an event, at regular intervals, to get how long queues are and save it for further processing
  - Results will be closer to theory if you wait a while to reach a steady state, and stop when you stop submitting jobs
- Testing
  - Try it out first with small numbers
  - See how it scales, e.g., by doubling #of jobs/simulated time each time; profile your code!
    - Example: `python3 -m cProfile ./mmn_queue [args] | less`

# Hints (4)

- Useful Python modules/functions are:
  - `collections.deque` for the FIFO queue
  - `heapq` for the priority queue
  - `random.exponential` for generating exponentially distributed random variables
  - `matplotlib` for plotting
- Nobody stops you from using your favorite language instead :)

# Multi-Server Version

- Rather than a single server, there are  $n$  servers
- To keep a coherent load, raise the rate of new job generation to  $n\lambda$
- Two strategies for selecting a queue:
  - Random queue choice ( $d=1$ )
  - Considering  $d$  queues and taking the one with the shortest queue (“Supermarket model”)

# Distributed Computing

A-08. Erasure Coding

# Problem Statement

- I have data to save
  - E.g., 100GB
- I can store data on  $N$  different servers
  - Say I pay storage per GB
- I want to be able to recover my data even if  $M$  servers fail and lose my data
- How to minimize the cost?

# Trivial Solution: Replication

- To safeguard myself against  $M$  server failures, I put a copy of all my data on  $M+1$  servers
- Of course, I'm safe if  $M$  servers die
- **Redundancy**—i.e., the ratio between amount of data I store and the amount of original data is  $M+1$ 
  - E.g., for  $M=2$  and 100GB of data, I need 300GB
  - Redundancy: 3

# N=3, M=1: Parity

- I split my data in 2 **blocks**  $B_0$  and  $B_1$ 
  - E.g., N=2 blocks of 50GB each
- I create a **parity** redundant block  $B_R$ 
  - $B_R = B_0 \text{XOR } B_1$
- If I lose one of the blocks  $B_i$ , I can recover it as
  - $B_i = B_R \text{XOR } B_{1-i}$
  - This is because  $x \text{XOR} (x \text{XOR} y) = (x \text{XOR} x) \text{XOR} y = y$
- Redundancy: 1.5
  - E.g., for 100GB, I need 150GB storage
  - Only 100GB to recover all the original data

# M=1, Any N

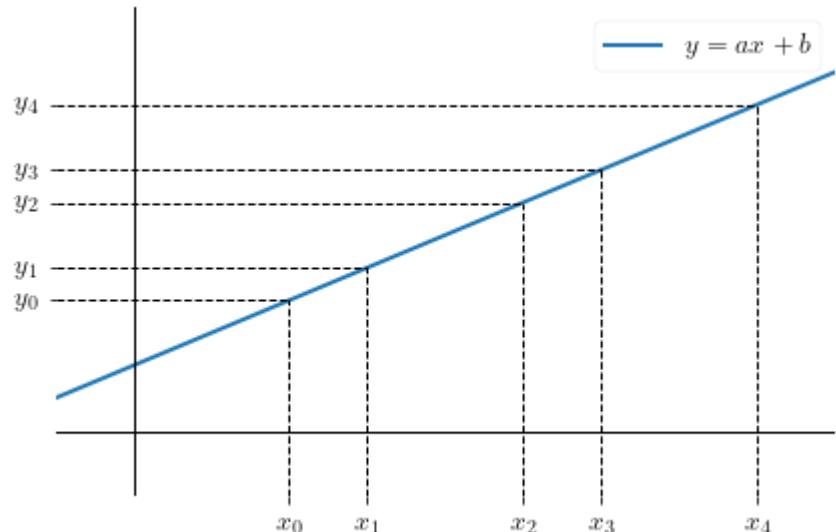
- I split my data in K=N-1 **blocks** of the same size
  - E.g., N=6, 100GB: 5 blocks  $B_0, \dots, B_{K-1}$  of 20GB each
- I create a **parity** redundant block  $B_R$ 
  - $B_R = B_1 \text{XOR } B_2 \text{XOR } \dots \text{XOR } B_N$
- If I lose one of the blocks  $B_i$ , I can recover it as
  - $B_i = B_1 \text{XOR } B_2 \dots \text{XOR } B_{i-1} \text{XOR } B_{i+1} \dots \text{XOR } B_{K-1} \text{XOR } B_R$
  - This is because  $x \text{XOR } (x \text{XOR } y) = y$
  - Here  $y = B_1 \text{XOR } B_2 \dots \text{XOR } B_{i-1} \text{XOR } B_{i+1} \dots \text{XOR } B_{K-1} \text{XOR } B_R$
- Redundancy:  $N/K = N/(N-1)$ 
  - Say N=6, 100GB: redundancy  $6/5=1.2$ , I need 120GB
  - Again, only 100GB to recover all original data

# Erasure Coding Magic: Any N & M

- I **encode** my data in N blocks, each of size  $1/K^{\text{th}}$  of the original data, where  $K=N-M$ 
  - E.g.,  $M=2$ ,  $N=6$ : 6 blocks of size 25GB
- I can decode **any** K of those blocks to recover my original data
  - E.g., any 4 of the  $N=6$  blocks in the example
  - Once again, I just need any blocks totaling 100GB to recover my original data
- Redundancy:  $N/(N-M)$ 
  - 1.5 in the example, 150GB total

# A Trip Into Erasure Coding

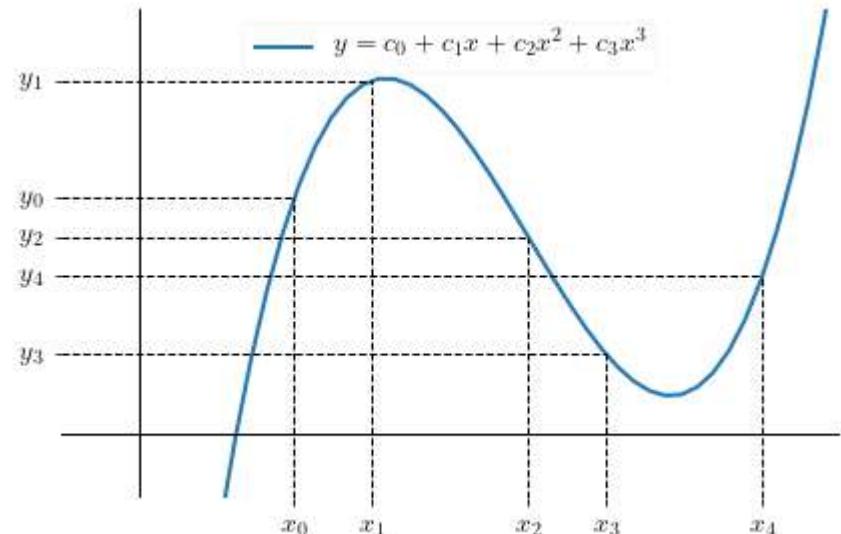
# Any N, M=N-2: Linear Oversampling



- A single straight line connects any two distinct points
- Hence, we can compute  $a$  and  $b$  with any two  $(x_i, y_i)$  pairs
  - If values of  $x_i$  are predetermined, no need to send those

- Message:  $a, b$ 
  - e.g.,  $a=3, b=2$
- Encoded message:  $y_0=f(x_0), y_1=f(x_1), \dots, y_{n-1}=f(x_{n-1})$ 
  - With  $x_i=i: (2, 5, 8, \dots)$
- With any two distinct  $(x_i, y_i)$  pairs, a system of two linear equations and 2 variables:
  - E.g., With  $y_2=8$  and  $y_5=17$ :
    - $ax_2+b=8 \rightarrow 2a+b = 8$
    - $ax_5+b=17 \rightarrow 5a+b = 17$
  - From here, it's trivial to compute the message
- If message and all  $x_i$  are all integers, all  $y_i$  will be too: **no need to handle non-integer math!**

# Any N&M: Polynomial Oversampling



- Any  $K=N-M$  distinct points identify a single polynomial of degree  $K-1$
- Hence, we can find the message with any  $K$   $(x_i, y_i)$  pairs
  - Again, we can agree beforehand what all  $x_i$  values are
- Could sound familiar if you have seen **secret sharing** in cryptography

- Message:  $c_0, \dots, c_{K-1}$
- Encoded message:  $f(x_0), f(x_1), \dots, f(x_{N-1})$
- With any  $n$  distinct  $(x_i, y_i)$  pairs, a system of  $n$  **linear** equations and  $n$  variables
  - $c_0 + c_1x_i + c_2x_i^2 + \dots + c_{k-1}x_i^{k-1} = y_i$
  - The nonlinear part **disappears** because all  $x_i$  values are known

# Polynomial Oversampling: Example

- $N=7, M=3, K=4$
- Sampling points  $x_0, \dots, x_6 = 0, \dots, 6$
- Message:  $(c_0, \dots, c_3); f(x)=c_0+c_1x+c_2x^2+c_3x^3$
- Suppose we lose the other values and remain with  $y_0=f(x_0)=f(0)=4, y_2=40, y_3=100, y_5=364$ . We get the equations
  - $c_0=4$
  - $c_0+2c_1+4c_2+8c_3=40$
  - $c_0+3c_1+9c_2+27c_3=100$
  - $c_0+5c_1+25c_2+125c_3=364$
- We can obtain the original message by solving the equations (e.g., by substitution)
  - Solution:  $(c_0, \dots, c_3) = (4, 2, 4, 2)$
  - Function  $f(x)=4+2x+4x^2+2x^3$
  - Encoded message  $y_0, \dots, y_6 = 4, 12, 40, 100, 204, 364, 592$
- **Problem:** numbers grow in size! If we encode them as bits, numbers in the encoded message will be **bigger** than those in the original one
- Is there a magic way to make sure numbers don't become bigger as we multiply them by powers of  $x$ ?
  - Yes there is! They're called  **finite fields**  (or Galois fields)

# It's OK If You Don't Remember Fields

- Informally: a field is a set in which addition, subtraction, multiplication and division are defined and behave “as in” real and rational numbers
- Disclaimer: in this and the following slides, **bold red** symbols **+**, **-**, **\***, **/**, **0**, **1** and  **$^{-1}$**  refer to operation on the field, and not on the numbers we’re used to
- Properties:
  - **Associativity**
    - $(a+b)+c=a+(b+c)$  and  $(a*b)*c=a*(b*c)$
  - **Commutativity**
    - $a+b=b+a$  and  $a*b=b*a$
  - **Additive & Multiplicative identity**
    - $a+0=a$ ,  $a*1=a$
  - **Distributivity**
    - $a*(b+c)=(a*b)+(a*c)$
  - **Additive and multiplicative inverses**
    - $a+(-a)=0$ ,  $a*(a^{-1})=1$  (*the multiplicative inverse is not defined for 0*)
- **It's useful in our case because we can solve our linear equations** in a field
  - If you think about it, to solve them you do substitutions and or add/multiply/divide the same number from both sides of an equation

# Finite Fields (Galois Fields)

- Fields that have **just a finite number of elements**
- Discovered (invented?) by Évariste Galois (1811-1832)
- They're cool, because we can give a number to each element of the field, and encode those numbers using  $\log_2(n)$  bits for a field of size  $n$ , and do everything as before!



# Integers Modulo A Prime Number Are A Finite Field

- Elements of the set:  $0, 1, \dots, p-1$
- Obvious definition of 0, 1, multiplication, addition and additive inverse
  - $\mathbf{0}=0$ ,  $\mathbf{1}=1$ ,  $-a=-a \bmod p$
  - $a+b=a+b \bmod p$
  - $a*b=ab \bmod p$
  - ***What about the multiplicative inverse?*** We'll see later
- Most properties are easy to prove (please try at home)  
... But what about the multiplicative inverse?

# Multiplication Table Modulo $m$

	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

- $m=7$  (**prime**)
- There's exactly one 1 in every row and column
- In the “modulo 7” field,  $5^{-1}=3$  and  $2^{-1}=4$

	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8
2	2	4	6	8	1	3	5	7
3	3	6	0	3	6	0	3	6
4	4	8	3	7	2	6	1	5
5	5	1	6	2	7	3	8	4
6	6	3	0	6	3	0	6	3
7	7	5	3	1	8	6	4	2
8	8	7	6	5	4	3	2	1

- $m=9$ , non-prime
- Not every value has an inverse
- Not a field

# Proof: There's Exactly One Multiplicative Inverse In Modulo p

- Consider  $a^*b=ab \text{ mod } p$ , with  $a, b$  in  $\{0, 1, \dots, p-1\}$ 
  - It can be zero only if one of  $a$  or  $b$  is **0**, because:
    - $p$  is a prime, so  $ab \text{ mod } p=0$  only if one of  $a$  and  $b$  is a multiple of  $p$
    - but they can't be multiples of  $p$  because they're smaller than  $p$ .
- Consider  $a$  in  $[1, p-1]$  and  $a^*i$  for all  $i$  in  $[1, p-1]$ 
  - For all  $i$  in  $1, 2, \dots, p-1$ ,  $a^*i \neq 0$ , neither  $a$  nor  $i$  are **0**
  - For all  $i \neq j$  in  $1, 2, \dots, p-1$ ,  $a^*i \neq a^*j$ , because:
    - $a^*i=a^*j$  would mean  $a(i-j)=0$  (distributive property)
    - $a \neq 0$  by hypothesis,  $i-j \neq 0$  because  $i \neq j$
  - Conclusion: all the  $p-1$   $a^*i$  values are different, and they can assume only the  $p-1$  values in  $[1, p-1]$ 
    - Each value will be represented exactly once, hence there will be a single unique value of  $i$  such that  $a^*i=1$
    - Hence, that will be the multiplicative inverse  $a^{-1}$

# Our Example, Modulo p

- $N=7, M=3, K=4$  (as before),  $p=7$  (*we need  $p \geq N$* )
- Function  $f(x)=c_0+(c_1*x)+(c_2*x^2)+(c_3*x^3)=c_0+c_1x+c_2x^2+c_3x^3 \text{ mod } 7$
- Sampling points  $x_0, \dots, x_6 = 0, \dots, 6$ 
  - **we're using all of them now, to add more we need to change p**
- Suppose we lose the others and remain with  $y_0=f(0)=3, y_2=1, y_3=5, y_5=3$ . We get the equations
  - $c_0=3$
  - $c_0+2*c_1+4*c_2+c_3=1$
  - $c_0+3*c_1+2*c_2+6*c_3=5$
  - $c_0+5*c_1+4*c_2+6*c_3=3$
- Let's now **solve this!** It's not so hard, you convert every number with its value modulo 7
  - E.g. -2 becomes 5 and 22 becomes 1
  - Inverses:  $1^{-1}=1, 2^{-1}=4, 3^{-1}=5, 4^{-1}=2, 5^{-1}=3, 6^{-1}=6$
  - To divide by  $x$ , you multiply by  $x^{-1}$  left and right

# We Stop Here With Theory

- Almost-practical usage:
  - $N$  is the number of machines that will store your data
  - $M$  is the number of failures you want to tolerate
  - Choose  $p$  not smaller than  $N$
  - Divide your data in  $K=N-M$  blocks of the same size
  - Encode it as a series of values smaller than  $p$ 
    - (find a way, e.g. padding, to make their number a multiple of  $K$ )
  - All elements of the first (original) block will be  $c_0$  coefficients, second block  $c_1$ , and so on
  - Encode and put all the  $y_0$  coefficients in the first encoded block, the  $y_1$  coefficients in the second block, and so on

# There's A Lot More In Coding Theory

- There are finite fields having  $p^m$  elements, any  $m \geq 1$ 
  - But they're harder to explain
  - Of course, computer people in practice use  $2^m$
- You can use coding to do **error correction** in addition to handle erasures
- They're implemented in hardware
- Found everywhere: telecommunications, QR codes, even CD readers from the '90s!
- If you want to play with them, look for a Reed-Solomon library in your favorite programming language

# More Coding Magic

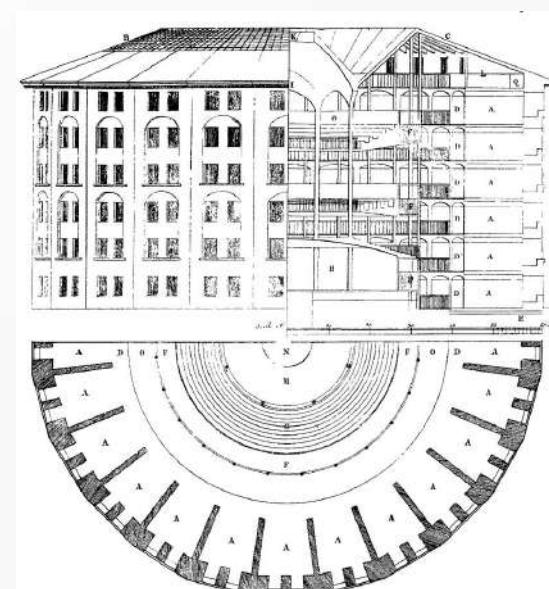
- Approaches that “waste” a bit of space compared to the “perfect” result but give you great properties
  - I.e., to recover 100 GB of data you’ll need a bit more than 100 GB
- **Fountain codes:** you don’t need to choose N to start with
  - Generate as many redundant blocks as you want, as a stream
  - You need a bit more than the amount of the original data to reconstruct it
  - Current standard: RaptorQ, [IEEE RFC 6330](#)
- **Regenerating codes** ([Dimakis et al. 2010](#)): if you lose one or a few blocks you don’t need the original plain-text to recreate them —just download a few encoded blocks and work from them

# Distributed Computing

A-09. Tor

# The Internet Panopticon

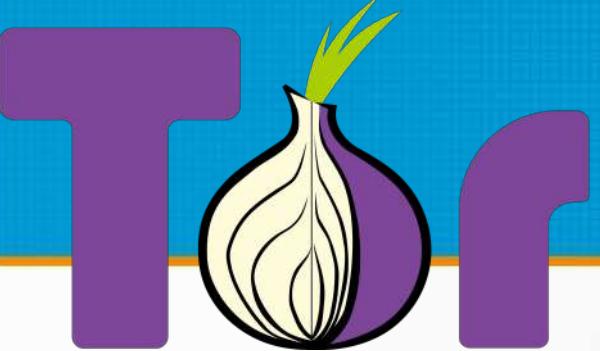
- Tracking is **ubiquitous** on the web: **more than 90%** of the websites do some form of tracking
  - Even **after the GDPR became law** and without user consent
  - Entities that track users **share information**, to get cross-website information
- Mobile apps probably track even more than websites
  - More difficult to investigate
  - More sensors
- Main player: the advertising industry
  - More covert ones: government agencies, malicious entities...



# The Privacy Debate

- How much privacy should people have on the Internet?
- Main argument **against** complete privacy:
  - Crime and terrorism
- Some arguments **for** it:
  - Protecting whistleblowers, activists and journalists
  - Avoiding tracking by totalitarian governments and corporations
  - Avoiding psychological profiling and manipulation
  - Criminals already have access to anonymity
    - Stolen phones, compromised machines

# Tor: The Onion Router

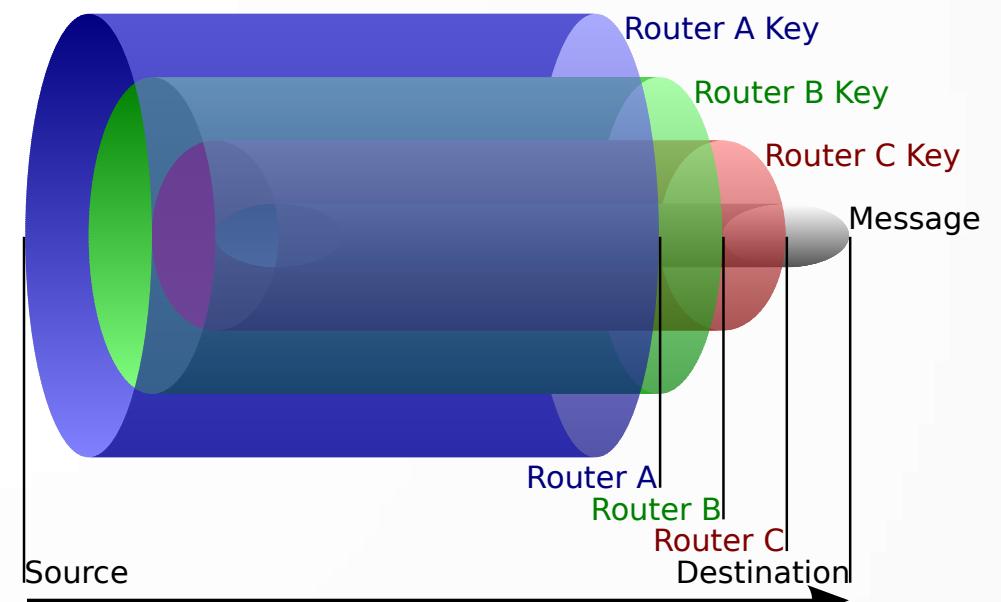


- A project initially funded by the USA government (Office of Naval Research & DARPA)
  - Purpose: protecting intelligence communication online
  - **Onion routing** idea published in mid-90s
- Picked up by the Electronic Frontier Foundation—a non-profit for digital rights—in 2004
  - References: [Tor website](#) & original design paper

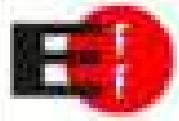
# How Tor Works

# Onion Routing

- **Layers of encryption**, like those of an onion
- I send a message intended to a destination through three routers: A, B and C.
- Each router “peels” one layer of encryption and sends the rest to the next step
- The message finally gets sent by C to the destination, after removing all the crypto layers
- No router knows **both the source and the destination**



*Image by Harrison Neal, CC-BY-SA 3.0*



# How Tor Works: 1



Tor node



unencrypted link



encrypted link

Alice



Step 1: Alice's Tor client obtains a list of Tor nodes from a directory server.

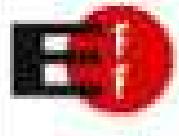


Jane

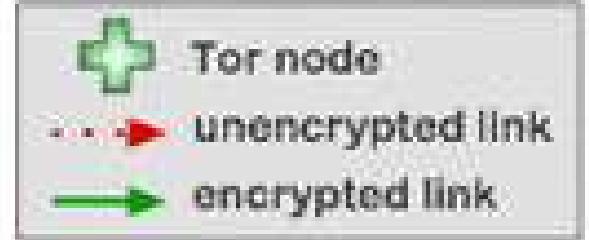


Bob

Dave



# How Tor Works: 2



Alice



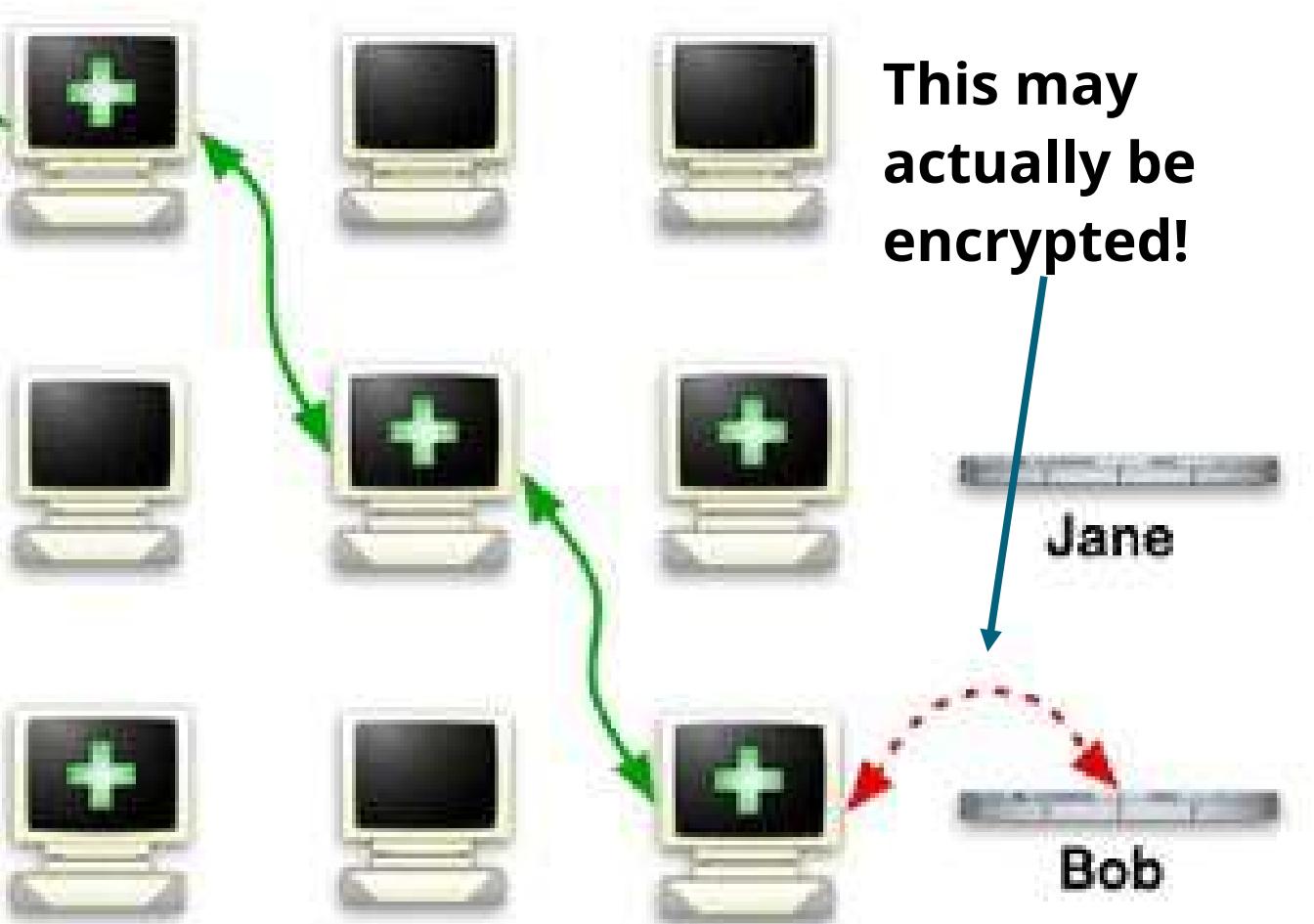
Step 2: Alice's Tor client picks a random path to destination server. **Green links** are encrypted, **red links** are in the clear.

This may actually be encrypted!

Jane

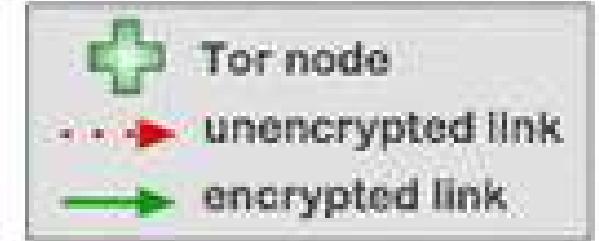
Dave

Bob





## How Tor Works: 3

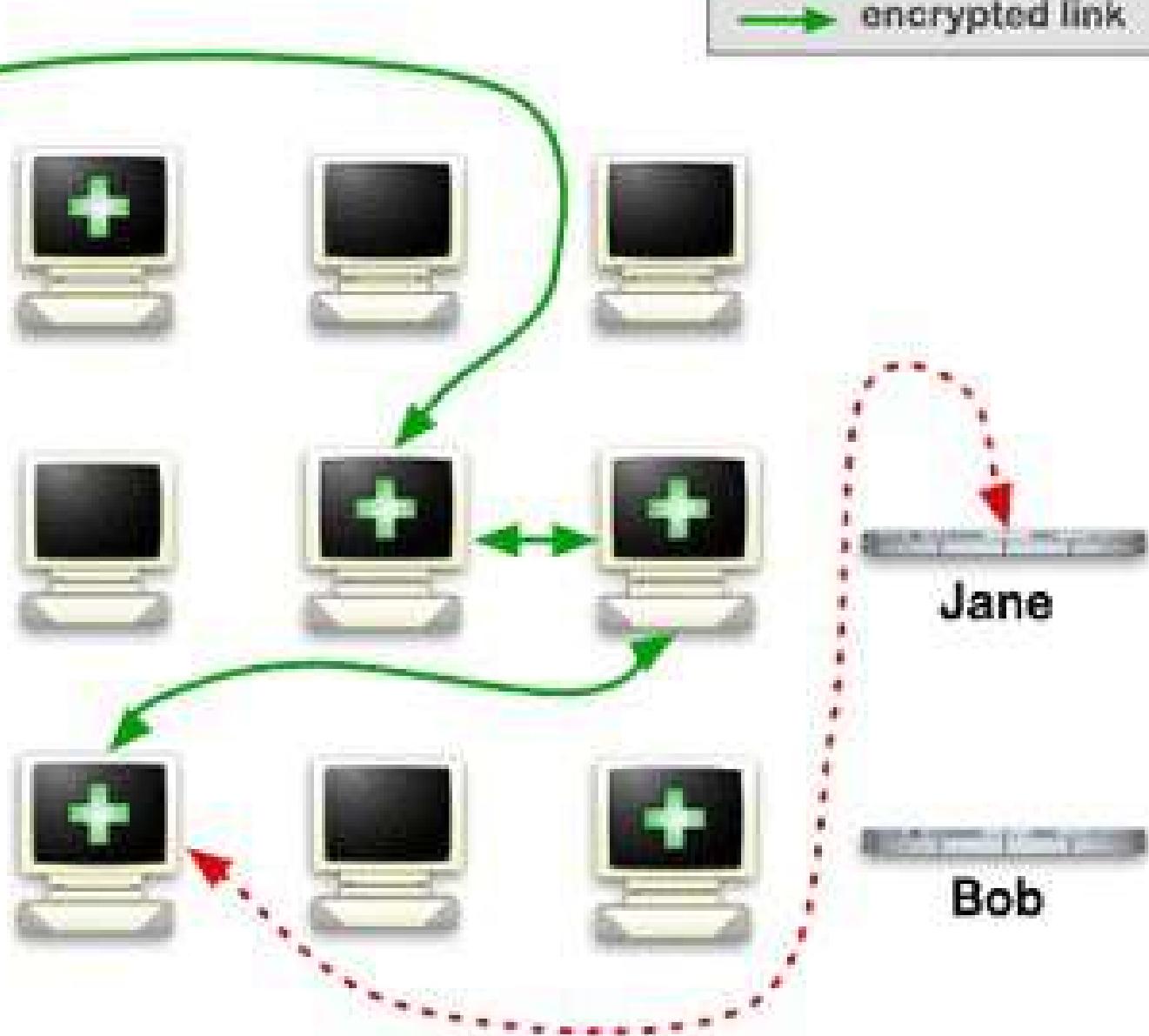


Alice

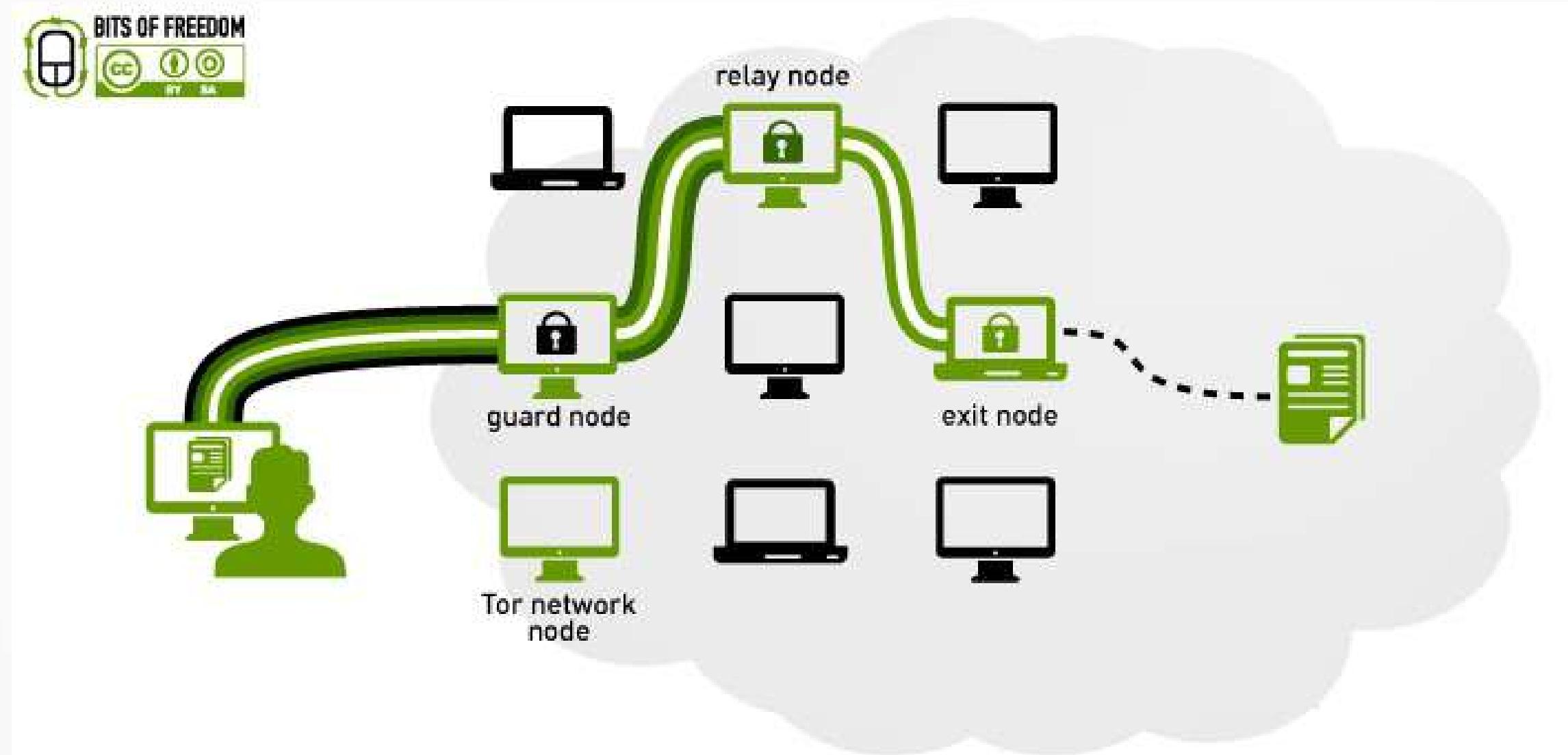


Step 3: If at a later time, the user visits another site, Alice's tor client selects a second random path. Again, green links are encrypted, red links are in the clear.

Dave



# A Bit More Detail



# Security Assumptions

- All the routers you choose shouldn't be **owned by the same attacker**
- Attackers can't see your traffic from **both guard and exit nodes**
  - Otherwise it's easy to correlate your traffic

# Tor Is a SOCKS Proxy

- SOCKS is a protocol that allows encapsulating TCP connections
  - SOCKS support UDP since version 5, but Tor doesn't support UDP connections
- You should only use apps that **will use that proxy server**
- Notably: BitTorrent over Tor isn't a good idea
  - It uses UDP

# Tor Browser

- Using Tor on your everyday browser is not a great idea
  - **Cookies:** the way most websites track you everyday
    - Website can sync cookies to correlate your visits on different websites
  - **Fingerprinting:** specific information on your hardware/software
    - From OS & configuration information to specific characteristics of your hardware
- The Tor Browser is a hardened Firefox designed to look identical for all users and never exit from the proxy

# Bridges

- Tor nodes are **publicly known**
  - Some countries **block access to known Tor nodes**
- **Bridges** exist to allow people to use Tor anyway
  - Non-public
  - People can ask for access to a bridge at a time

# **Onion Services (Darknet)**



# Onion Services: Step 1

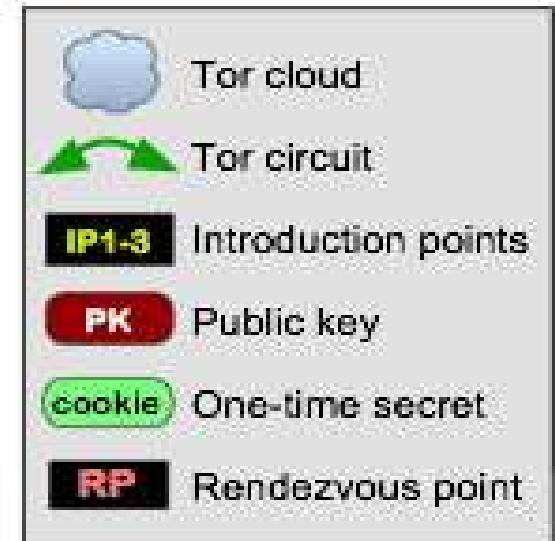
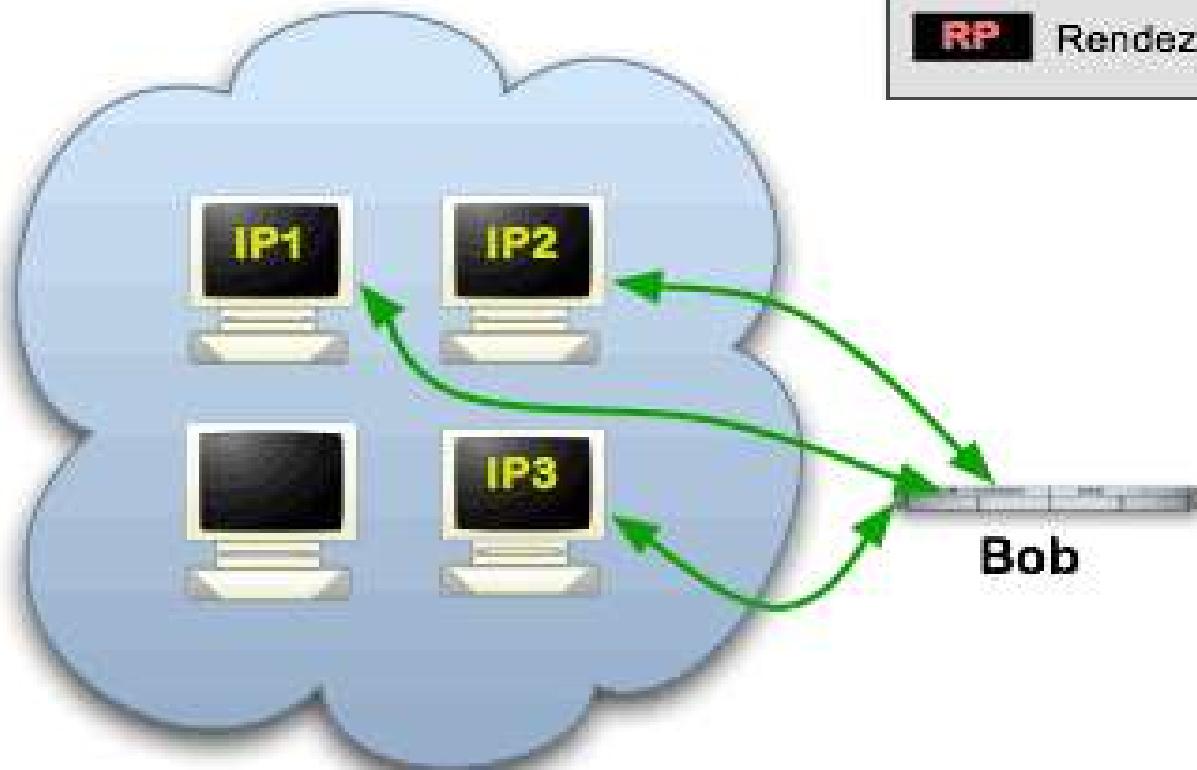
**Step 1:** Bob picks some introduction points and builds circuits to them.



Alice



Bob

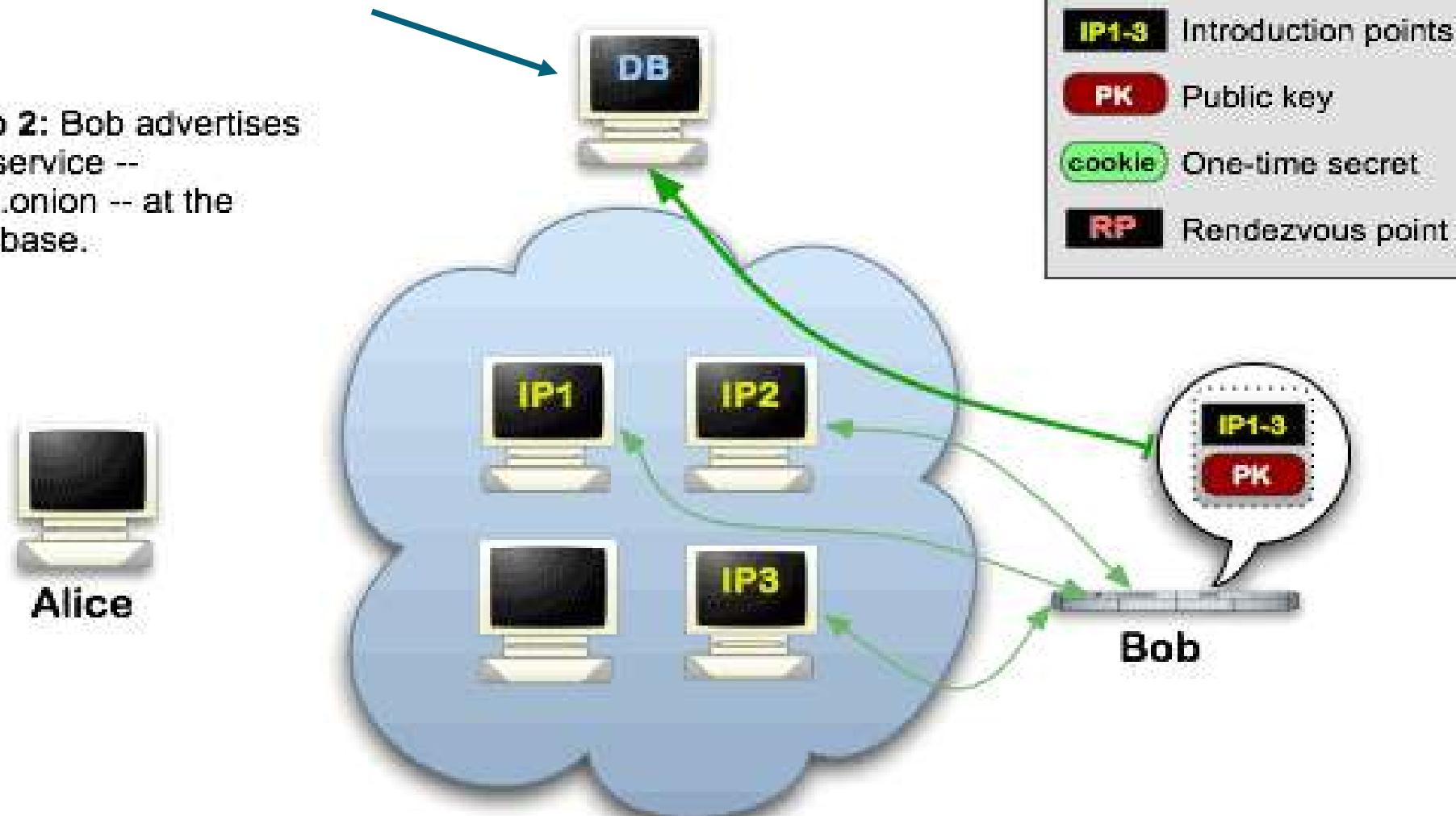




## Onion Services: Step 2

The DB is decentralized... remember the name: **DHT**

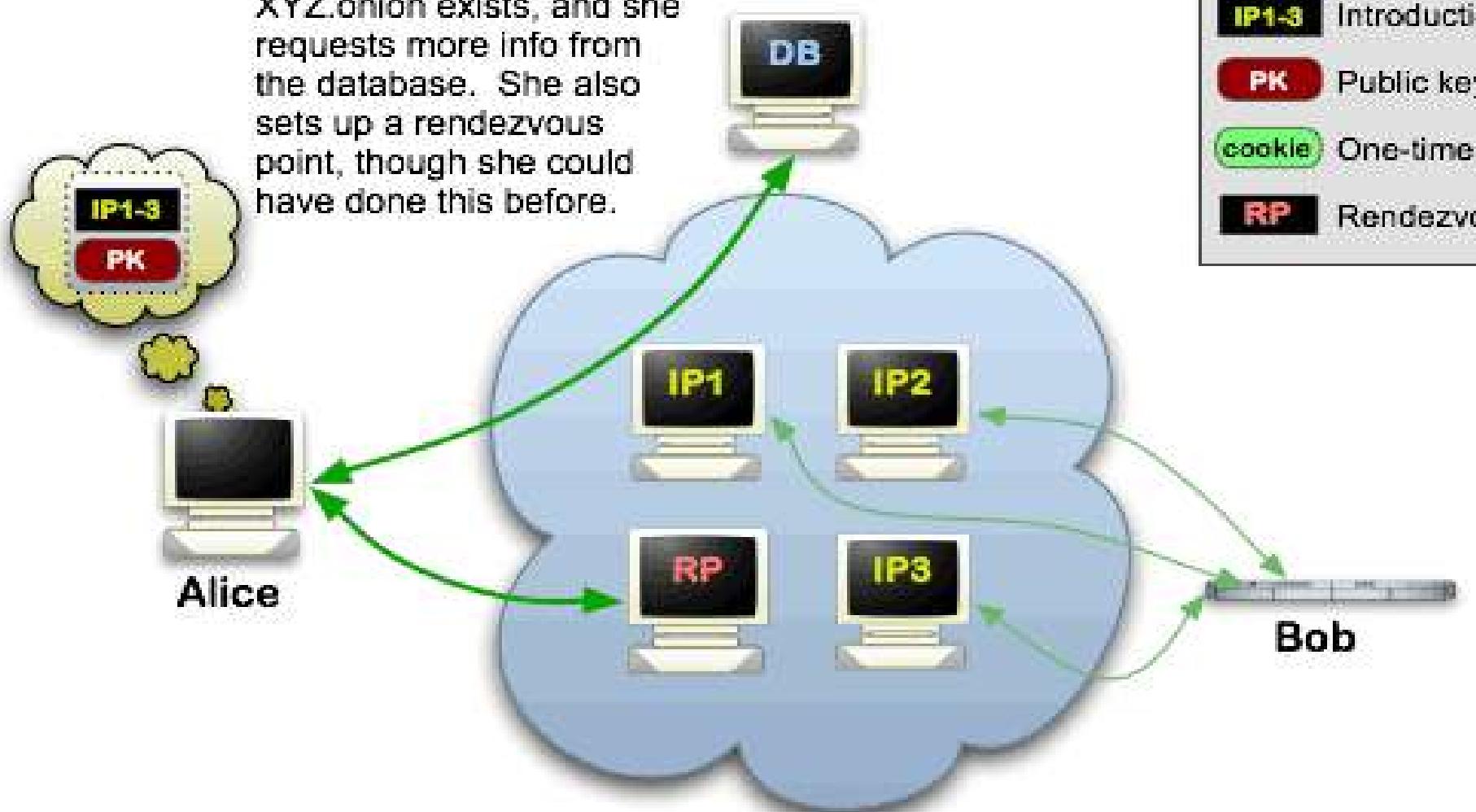
**Step 2:** Bob advertises his service -- XYZ.onion -- at the database.





# Onion Services: Step 3

**Step 3:** Alice hears that XYZ.onion exists, and she requests more info from the database. She also sets up a rendezvous point, though she could have done this before.

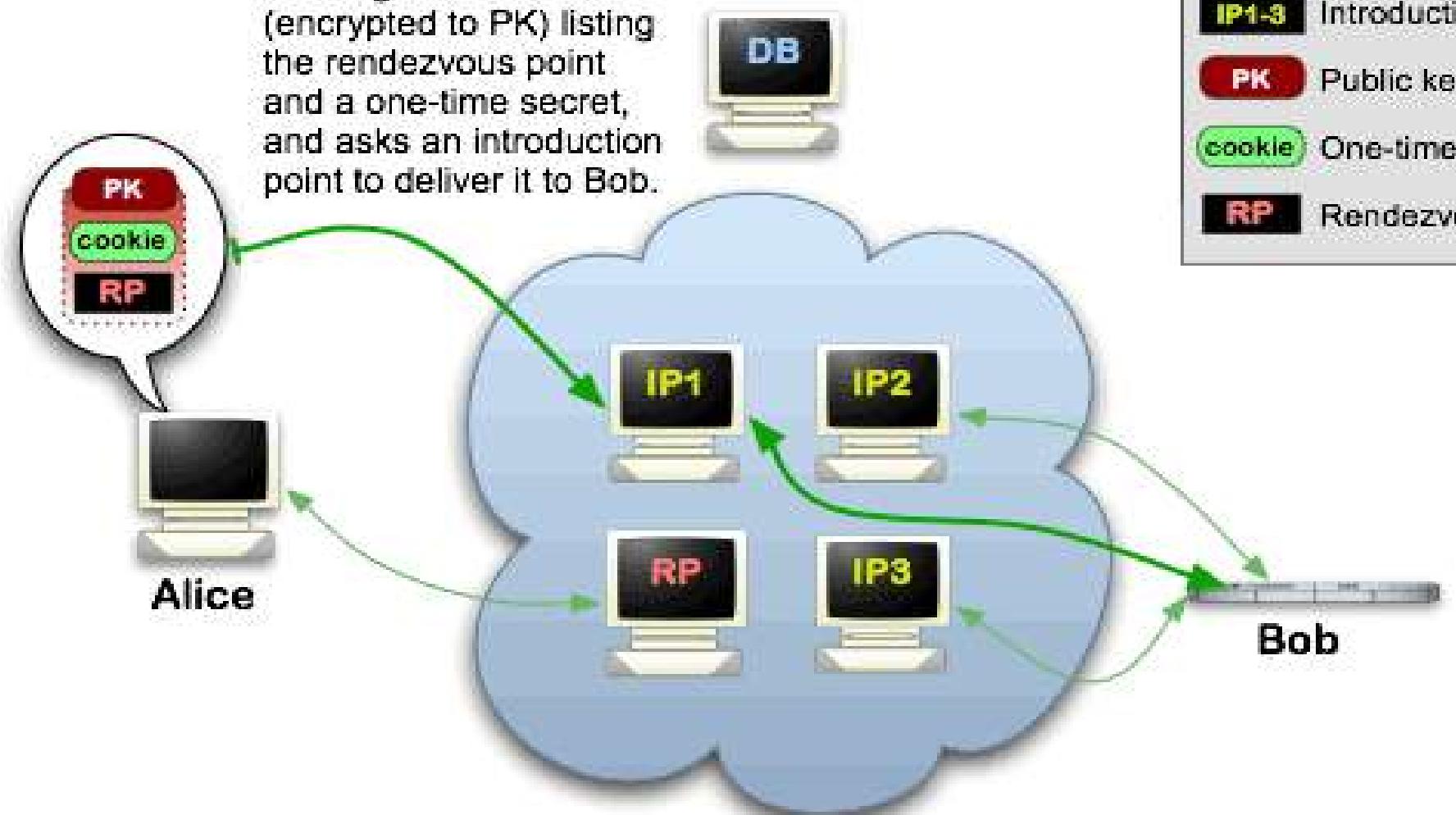


	Tor cloud
	Tor circuit
	IP1-3
	PK
	cookie
	Rendezvous point



# Onion Services: Step 4

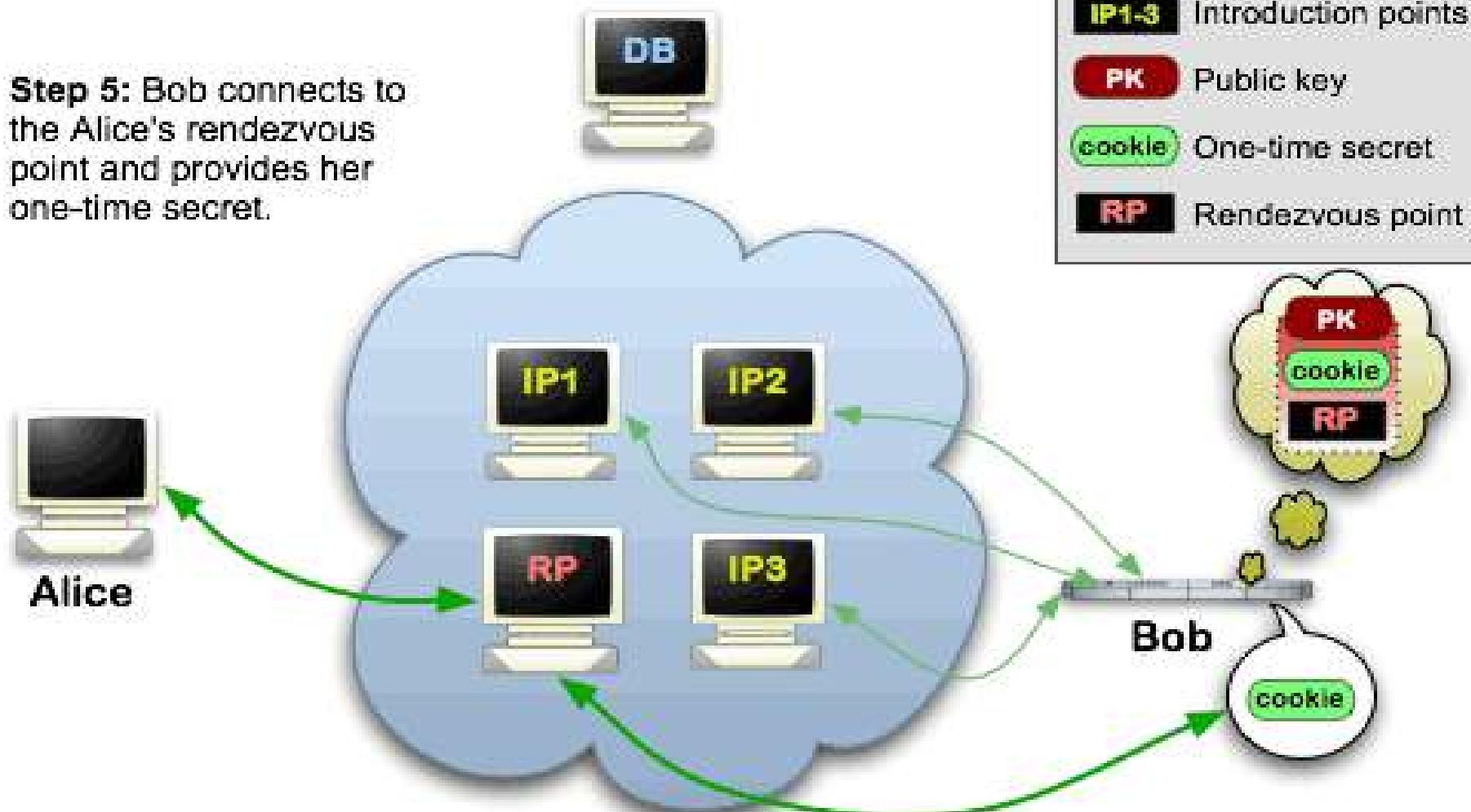
**Step 4:** Alice writes a message to Bob (encrypted to PK) listing the rendezvous point and a one-time secret, and asks an introduction point to deliver it to Bob.





# Onion Services: Step 5

**Step 5:** Bob connects to the Alice's rendezvous point and provides her one-time secret.



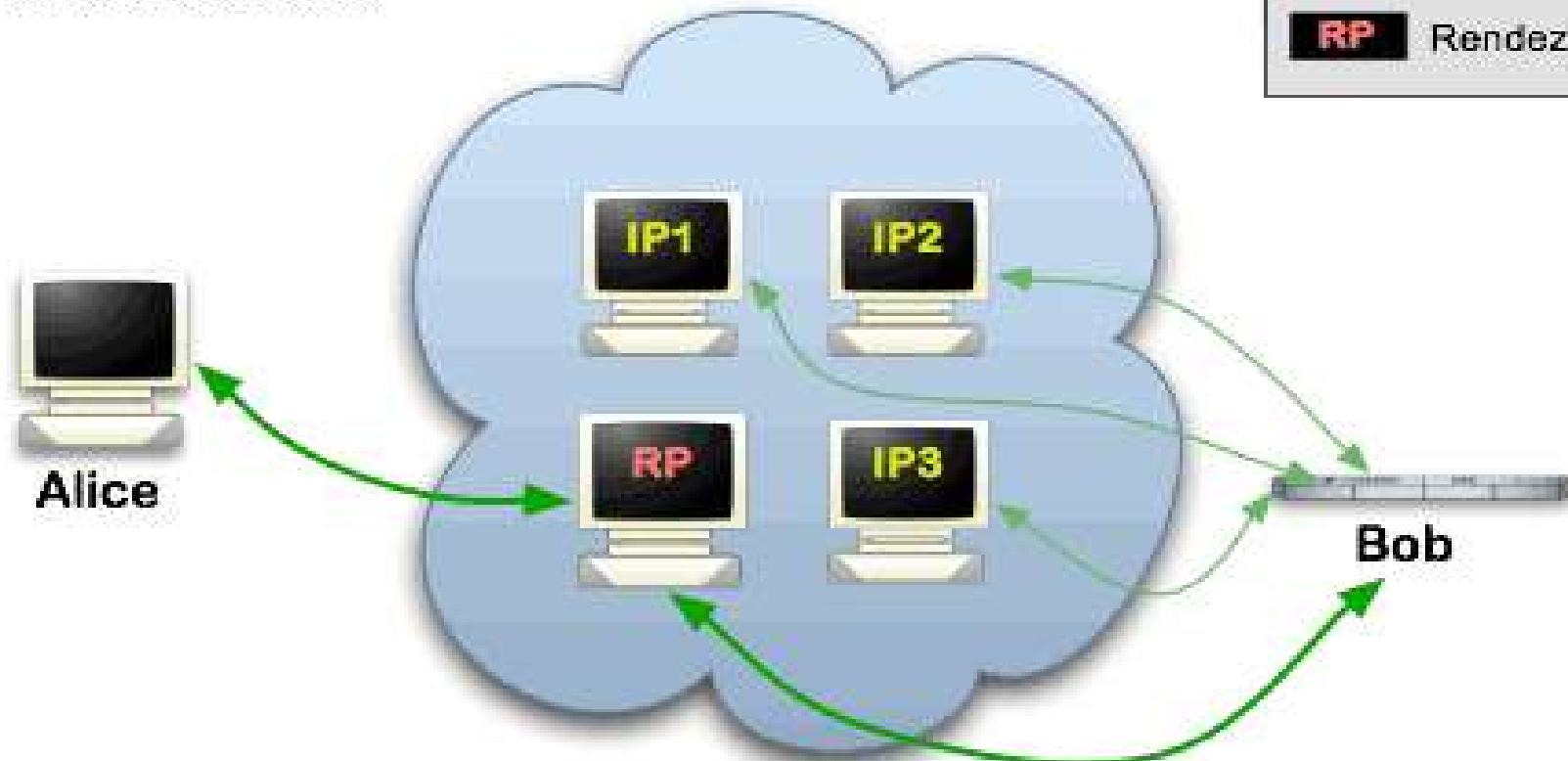


# Onion Services: Step 6

**Step 6:** Bob and Alice proceed to use their Tor circuits like normal.



	Tor cloud
	Tor circuit
	Introduction points
	Public key
	One-time secret
	Rendezvous point



# Numbers

# How Big Is Tor?

- Data from <https://metrics.torproject.org/>
  - Around 5M concurrent users ('20: 2M, '21: 2.2M, '22: 2.5M)
  - ~120k bridge users ('20: 40-50k, '21: ~45k-80k, '22: 75k)
  - ~8k relays ('20-'21: 7k, '22: 6-6.5k)
  - ~2k bridges ('20: 1.8k, 2021: 1.6k, '22: 2.3k)
  - ~300 GB/s aggregate bandwidth ('20-21: 250, '22: 280)
  - ~900 GB/s available ('20: 500, '21: 500-1000, '22: 600)
- Peaks probably caused by technical/political issues

# What Are Onion Services Used For?

- Source:  
*“Cryptopolitik and the Darknet”*  
(Moore & Rid,  
2016)

Category	Websites
None	2,482
Other	1,021
Drugs	423
Finance	327
Other illicit	198
Unknown	155
Extremism	140
Illegitimate pornography	122
Nexus	118
Hacking	96
Social	64
Arms	42
Violence	17
Total	5,205
Total active	2,723
Total illicit	1,547

# Attacks & Defenses

# Tor History and Research

- The security of Tor is **not perfect**
  - We've seen that, by design, powerful attackers can discover information about users
- However, history tells us it's **good enough** in most cases
  - Research looking for its weaknesses
  - Even big agencies like the NSA
  - People were caught because of **mistakes**, not attacking Tor

# Discovering Bridges

- Bridges can be discovered (and censored)
- With a full scan of all the IPv4 addresses
  - in 2013, Durumeric et al. (Zmap) discovered 86% of the Tor bridges
- With deep packet inspection (DPI)
  - E.g., the Great Firewall of China recognizes traffic protocols
- Countermeasure: **obfuscation** (pluggable transports)
  - Together with the bridge address you get a secret; protocols like **obfs4** and **ScrambleSuit** hide your protocol to DPI

# Website Fingerprinting

- A technique to identify which website a user is looking at by looking at the sizes and timing of encrypted packages
- Tor uses messages of a fixed 512 byte size (“cells”)
  - Together with higher latencies, this makes fingerprinting **less efficient**
- Many works use a “closed world” hypothesis
  - “Out of these X websites, which one am I visiting?”
  - The real-world fingerprinting problem is more difficult because **websites are a lot and change frequently**
  - On the other hand, darknet sites are less: attacks to fingerprint them **may actually be more relevant**

# NSA: “Tor Stinks”

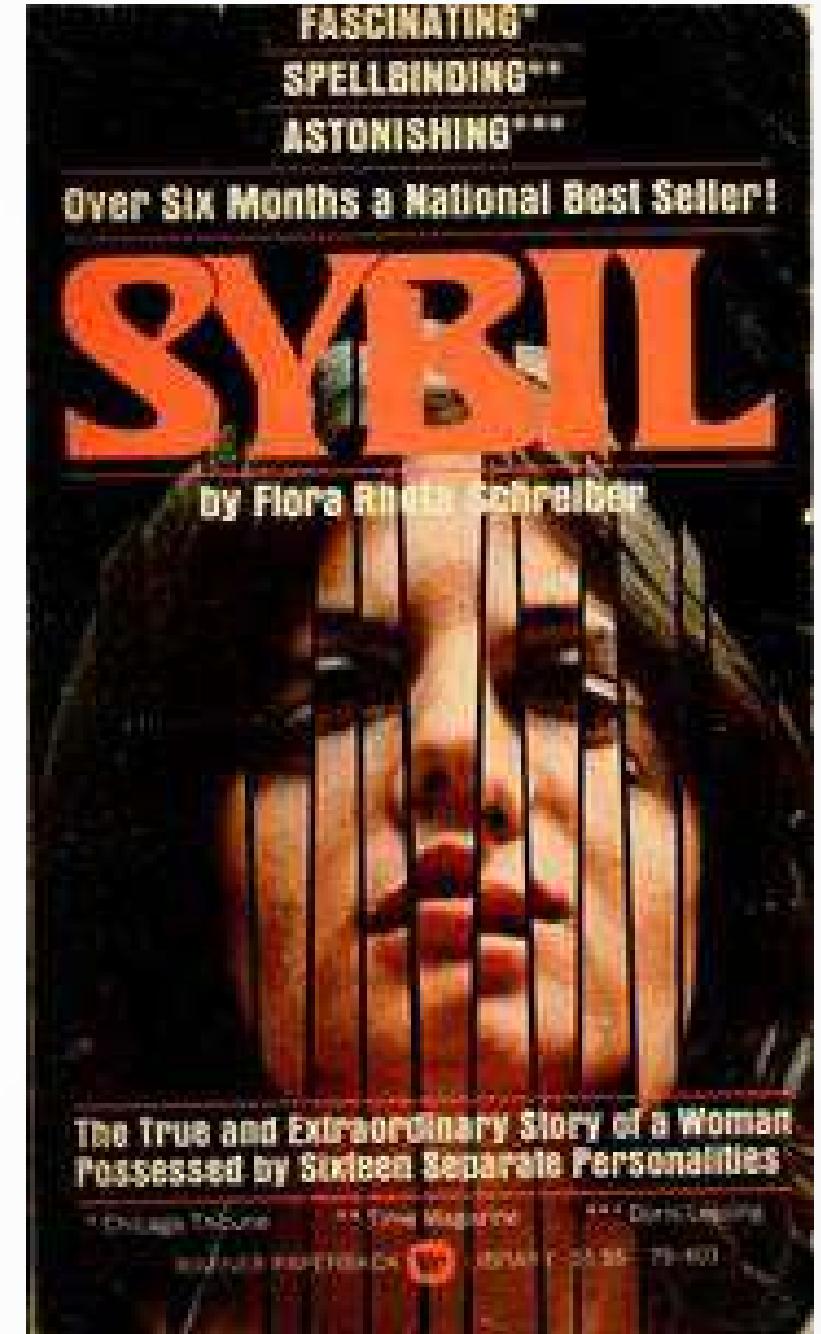
- A 2012 presentation
  - Revealed in 2013 among the Snowden documents
- Limited success in attacking it, through
  - Controlling nodes
  - Vulnerabilities
  - Exploiting errors
- *“We will never be able to de-anonymize all Tor users all the time”*

# Operation Bayonet

- Suggested reading/listening
  - From *Darknet Diaries*, a podcast about computer security
- The story of two darknet services selling illegal goods, seized by the police of two different countries

# Sybil Attack

- Name from a **book** about a woman with 16 personalities
- In P2P: an attacker creates a **very large number of nodes** to **subvert the system**
- Here, it runs many relays, increasing **likelihood of correlating traffic**
- Countermeasure: **fingerprint** node behavior (joining, uptime, ...)
- 2021: a **large attack** (probably state-sponsored) was discovered



# Analysis of Bloom Filters

Matteo Dell'Amico

Distributed Computing

# Outline

1 Probability of False Positives

2 Choosing  $m$  and  $k$

# Outline

1 Probability of False Positives

2 Choosing  $m$  and  $k$

# Probability of False Positives (1)

- Assume hash functions select array positions with equal probability
- We have  $m$  bits,  $k$  hash functions and  $n$  elements in our Bloom filter

# Probability of False Positives (1)

- Assume hash functions select array positions with equal probability
- We have  $m$  bits,  $k$  hash functions and  $n$  elements in our Bloom filter
- The probability that a bit is not set to 1 by a single hash function for a single element is

$$1 - \frac{1}{m}$$

# Probability of False Positives (1)

- Assume hash functions select array positions with equal probability
  - We have  $m$  bits,  $k$  hash functions and  $n$  elements in our Bloom filter
  - The probability that a bit is not set to 1 by a single hash function for a single element is
- $$1 - \frac{1}{m}$$
- For the  $k$  hashing functions, assuming they're independent the probability that a bit is not set to 1 for each of them is

$$\left(1 - \frac{1}{m}\right)^k$$

## Probability of False Positives (2)

- After inserting  $n$  elements, a bit is still set to 0 with probability

$$\left(1 - \frac{1}{m}\right)^{kn}$$

so, it is 1 with probability

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

## Probability of False Positives (2)

- After inserting  $n$  elements, a bit is still set to 0 with probability

$$\left(1 - \frac{1}{m}\right)^{kn}$$

so, it is 1 with probability

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

- 
- We get a false positive when  $k$  random bits are set to 1, hence with probability

$$p_{err} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

## Probability of False Positives (3)

- We can use the fact that

$$\lim_{m \rightarrow \infty} \left(1 - \frac{1}{m}\right)^m = \frac{1}{e}$$

to get, for large  $m$  values

$$p_{err} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k = \left(1 - \left(\left(1 - \frac{1}{m}\right)^m\right)^{kn/m}\right)^k \approx \left(1 - \left(\frac{1}{e}\right)^{kn/m}\right)^k$$

$$p_{err} \approx \left(1 - e^{-kn/m}\right)^k.$$

## Probability of False Positives (3)

- We can use the fact that

$$\lim_{m \rightarrow \infty} \left(1 - \frac{1}{m}\right)^m = \frac{1}{e}$$

to get, for large  $m$  values

$$p_{err} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k = \left(1 - \left(\left(1 - \frac{1}{m}\right)^m\right)^{kn/m}\right)^k \approx \left(1 - \left(\frac{1}{e}\right)^{kn/m}\right)^k$$

$$p_{err} \approx \left(1 - e^{-kn/m}\right)^k.$$

- In real-world cases, the approximation is good and this is the value we'll be using.

# Outline

1 Probability of False Positives

2 Choosing  $m$  and  $k$

# Optimal Number of Hash Functions

- To minimize  $p_{err}$ , the optimal number of hash functions is

$$k = \frac{m}{n} \ln 2$$

(we disregard the fact that that value shouldn't be an integer)

# Optimal Number of Hash Functions

- To minimize  $p_{err}$ , the optimal number of hash functions is

$$k = \frac{m}{n} \ln 2$$

(we disregard the fact that that value shouldn't be an integer)

- Note that with  $k = \frac{m}{n} \ln 2$ , the probability that a given bit is 0 is

$$\left(1 - \frac{1}{m}\right)^{kn} \simeq e^{-kn/m} = e^{-\ln 2} = \frac{1}{e^{\ln 2}} = \frac{1}{2}$$

# Optimal Number of Hash Functions

- To minimize  $p_{err}$ , the optimal number of hash functions is

$$k = \frac{m}{n} \ln 2$$

(we disregard the fact that that value shouldn't be an integer)

- Note that with  $k = \frac{m}{n} \ln 2$ , the probability that a given bit is 0 is

$$\left(1 - \frac{1}{m}\right)^{kn} \simeq e^{-kn/m} = e^{-\ln 2} = \frac{1}{e^{\ln 2}} = \frac{1}{2}$$

- This means that a Bloom filter is most efficient when half of the bits are 0s and half are 1s
- Intuitively, it makes sense: the data structure is carrying **as much information as possible!**

# How Big Given an Error Rate?

- With optimal  $k$ , the false positive rate is

$$p_{err} \approx \left(1 - e^{-kn/m}\right)^k = \left(1 - e^{-\ln 2}\right)^{\frac{m}{n} \ln 2} = \frac{1}{2}^{\frac{m}{n} \ln 2} = \left(e^{-\ln 2}\right)^{\frac{m}{n} \ln 2} = e^{-\frac{m}{n}(\ln 2)^2}$$

# How Big Given an Error Rate?

- With optimal  $k$ , the false positive rate is

$$p_{err} \approx \left(1 - e^{-kn/m}\right)^k = \left(1 - e^{-\ln 2}\right)^{\frac{m}{n} \ln 2} = \frac{1}{2}^{\frac{m}{n} \ln 2} = \left(e^{-\ln 2}\right)^{\frac{m}{n} \ln 2} = e^{-\frac{m}{n}(\ln 2)^2}$$

- If we fix  $p_{err} = \epsilon$ , we get

$$\ln \epsilon = -\frac{m}{n} (\ln 2)^2,$$

hence

$$m = -\frac{n \ln \epsilon}{(\ln 2)^2}$$

## Bits Per Item

- The optimal number of bits per item is

$$\frac{m}{n} = -\frac{\ln \epsilon}{(\ln 2)^2} \approx -2.08 \ln \epsilon$$

- Let's change the base:

$$\ln \epsilon = \frac{\log_{10} \epsilon}{\log_{10} e} \approx 2.30 \log_{10} \epsilon,$$

hence

$$\frac{m}{n} \approx -4.79 \log_{10} \epsilon$$

## Bits Per Item: Interpretation

$$\frac{m}{n} \approx -4.79 \log_{10} \epsilon$$

- For an error rate of 10%,  $\epsilon = 0.1$  and  $\log_{10} \epsilon = -1$ , so we have 4.79 bits per item—**less than one byte per item**

## Bits Per Item: Interpretation

$$\frac{m}{n} \approx -4.79 \log_{10} \epsilon$$

- For an error rate of 10%,  $\epsilon = 0.1$  and  $\log_{10} \epsilon = -1$ , so we have 4.79 bits per item—**less than one byte per item**
- For 1%,  $\epsilon = 0.01$  and  $\log_{10} \epsilon = -2$ , so we have 9.59 bits per item: a bit more than 1 byte per item
- For 0.1%,  $\epsilon = 0.001$  and  $\log_{10} \epsilon = -3$ , we have 14.38 bits per item: **less than 2 bytes per item**
- Every additional 0 in  $\epsilon$  only adds 5 bits per item...

## Bits Per Item: Interpretation

$$\frac{m}{n} \approx -4.79 \log_{10} \epsilon$$

- For an error rate of 10%,  $\epsilon = 0.1$  and  $\log_{10} \epsilon = -1$ , so we have 4.79 bits per item—**less than one byte per item**
- For 1%,  $\epsilon = 0.01$  and  $\log_{10} \epsilon = -2$ , so we have 9.59 bits per item: a bit more than 1 byte per item
- For 0.1%,  $\epsilon = 0.001$  and  $\log_{10} \epsilon = -3$ , we have 14.38 bits per item: **less than 2 bytes per item**
- Every additional 0 in  $\epsilon$  only adds 5 bits per item...
- If you're interested in even more, look up **cuckoo filters** :)

# Distributed Computing

A-10. Search in P2P Systems

# Peer-to-Peer (P2P)

- Distributed applications where nodes (**peers**) play “equal roles”
  - Self-organized and adaptive
  - Distributed and decentralized, hence fault-tolerant
    - An app owned by a single organization **may be shut down**
  - Censorship-resistant (as we’ve seen with Tor)
  - Uses resources that would be wasted otherwise

# P2P Applications (and examples)

- Killer applications:
  - 1999: **file sharing** (Napster)
  - 2008: **cryptocurrencies** (Bitcoin)
  - 2013: **smart contracts**, i.e., “world computer” (Ethereum)
- Other uses:
  - Decentralized chat & audio calling (Skype)
  - Audio streaming (Spotify)
  - Censorship-resistant & private systems (Tor)
  - Cheap file/multimedia stream distribution & tolerance to flash-crowds (BitTorrent)
  - Decentralized private architectures in datacenters (Amazon Dynamo)
  - P2P storage & backup (IPFS)

# Searching

- In a system with potentially millions of peers, how to find a given piece of content?
- It's a non-trivial problem, which balances decentralization with performance

# Napster: Centralized P2P

# 1999: Napster



- File-sharing used mostly for MP3s
- A central index server, to which users uploaded information about their songs
- Solved scalability issues for bandwidth
  - **Core** internet bandwidth was a bigger problem back then
- Legally tricky: uploading copyrighted content was illegal, but what about just **telling where it was?**
- Closed down in 2001 after reaching **26.4M users**

# **Unstructured P2P**

# Getting Decentralized

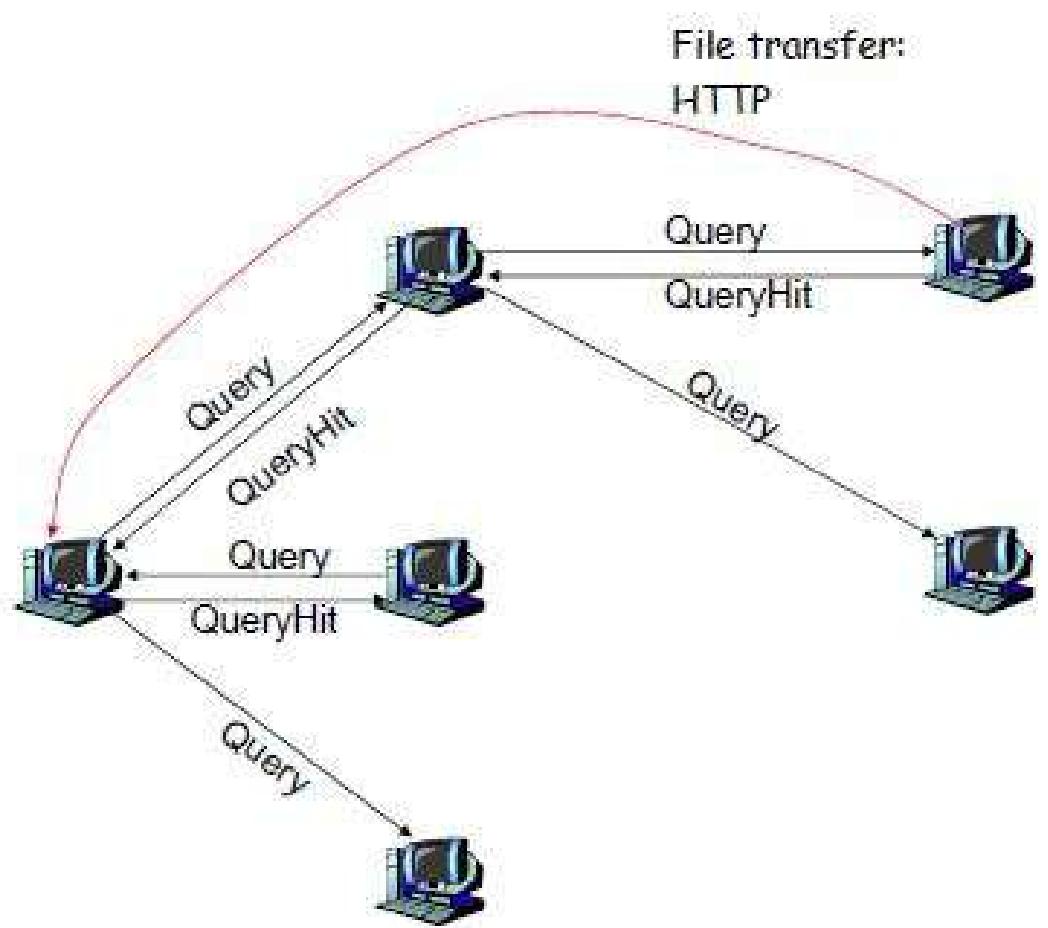
- The centralization of Napster arguably caused its demise:
  - There was a central server to shut down
  - There was a company to sue
- Work started on completely decentralized applications
- Idea: create **overlay networks**, i.e., networks **on top of other networks**
  - i.e., the P2P network is over the TCP/IP one

# 2000: Gnutella

- The first decentralized P2P file-sharing network
- **Overlay network** where each node is connected to a few others (5 by default)
- **Bootstrap**: each node contacts some services (“caches”) to get some nodes to connect with
- If a node is “full” with connections, it will forward the connection request to its neighbors
- New nodes discovered will be saved for the next sessions

# Flooding: Searching Everybody

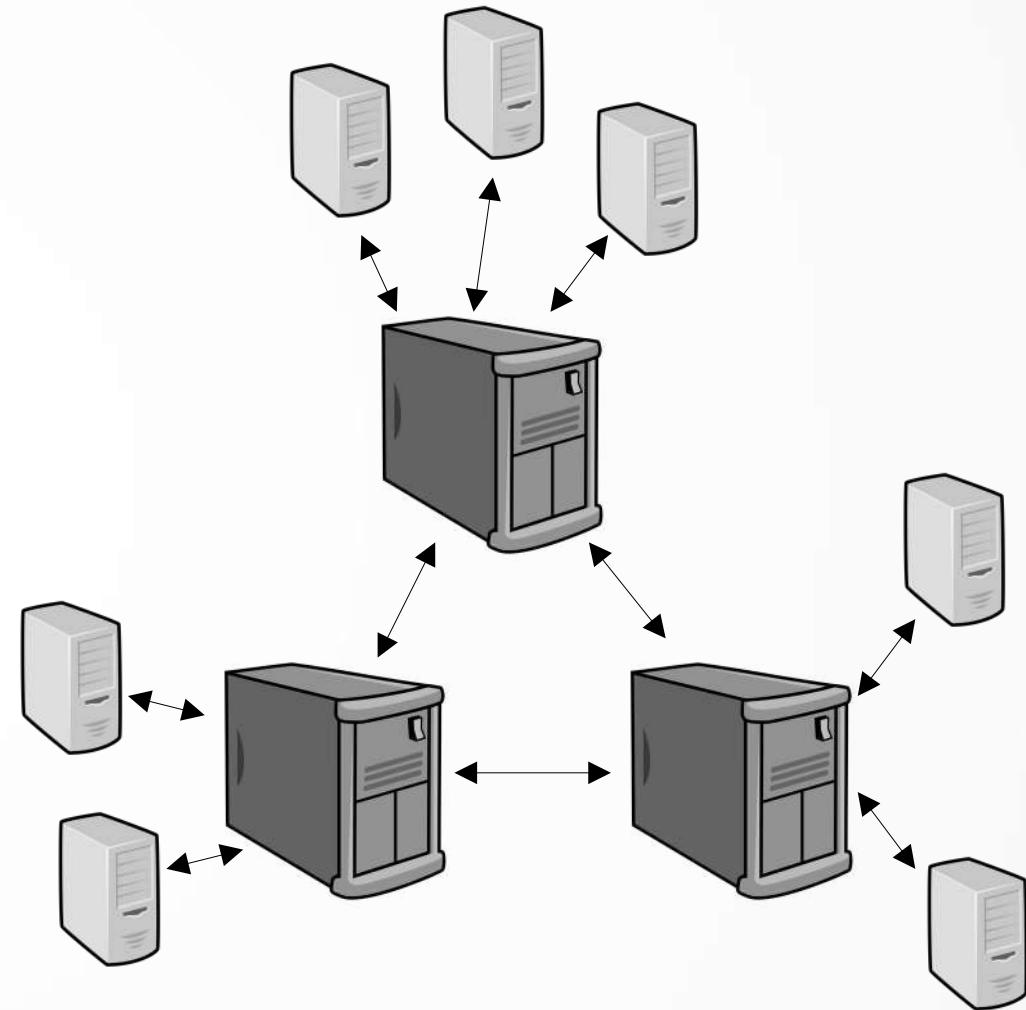
- New query: set a time-to-live (TTL, max 7) and forward it to all neighbors
  - Each of them will decrease the TTL by 1 and forward the query to its neighbors
- Many duplicate queries and very slow, but working to some extent
- However, routing messages could overload machines—especially the weakest ones
- Lots of redundant messages



CC-BY-SA 3.0, Danny Bickson

# 2002: Gnutella 0.6 & Ultrapeers

- There was a big difference between dial-up and ADSL nodes—dial-up ones were often overwhelmed just by sending queries around
- Separation between **leaf nodes** and **ultrapeers**
- Nodes with bandwidth and stability could get upgraded to **ultrapeers**
- Leaf nodes connect to 3 ultrapeers
- Ultrapeers connect to 32+ other ultrapeers
- “Superpeers” were a good design pattern used by several other apps (e.g., Skype)



# Searching with Ultrapeers

- Each node sends to its ultrapeers a “query routing table” (QRT): a representation of the set of files they have
- Ultrapeers aggregate their QRTs and those of all their leaves, and sends the results to all their neighbors
- Queries get sent to a peer only if they have a hope of having the requested file
- These modifications greatly improved scalability
- With higher degree, TTL was lowered to 4

# Bloom Filters

# What do QRTs do?

- QRTs should represent a set of keywords
- If my query is “foo bar” I want to get answers for files that match with **both** keywords
- Maybe you have files that match “foo baz” and “bar qux”—in that case it’s ok to get a false positive: you match both keywords, but don’t have a single file that matches them
- We can have false positives, but no false negatives
- We want QRTs to be **as small as possible**

# Bloom Filters

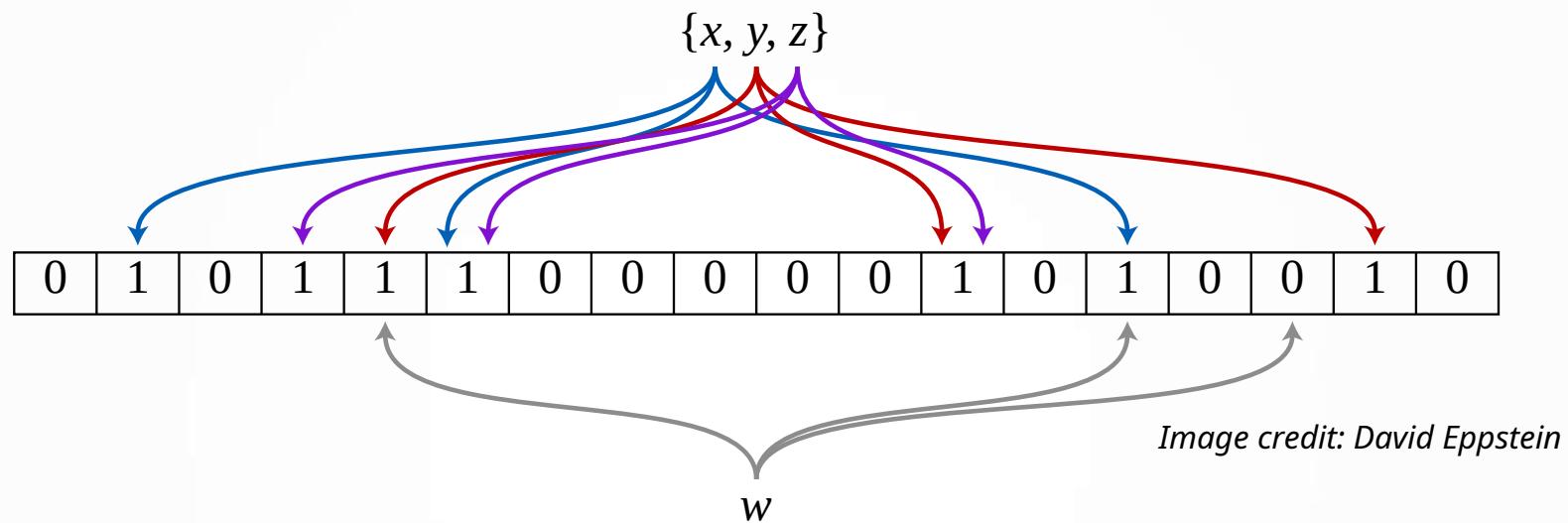
- A data structure invented in 1970
- A data structure for representing sets, giving two methods:
  - **add(x)**: add an element to the set
  - **test(x)**: tell me if  $x$  is in the set
    - There's a possibility for **false positives**
- No way of retrieving the original elements
- On the other hand, **very compact**: use very little space
- Great for our use case!

Space/Time Trade-offs in  
Hash Coding with  
Allowable Errors

BURTON H. BLOOM

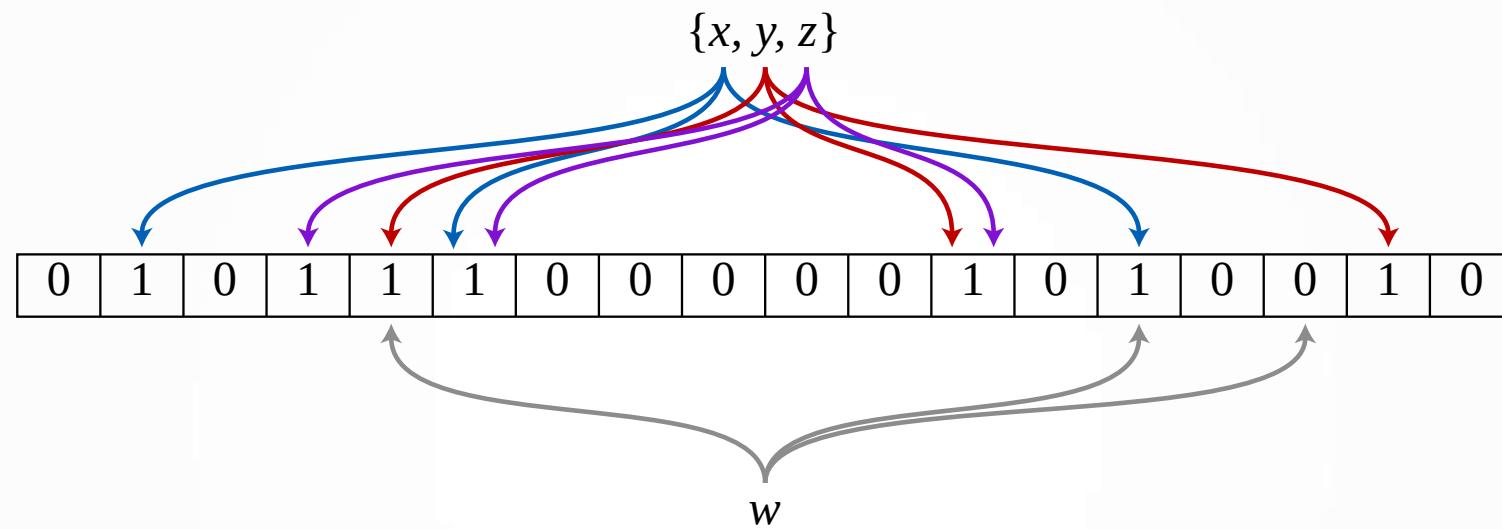
*Computer Usage Company, Newton Upper Falls, Mass.*

# How Bloom Filters Work



- A list of  $m$  bits
- $k$  hash functions mapping elements to value in  $[0, m-1]$
- **Add** sets to 1 the corresponding  $k$  bits
- **Test** verifies if the  $k$  bits are all set to 1

# Bloom Filters: Example



- In this case, we created a filter representing elements  $\{x,y,z\}$ 
  - Set to 1 the corresponding bits
- If we test for any of them, we'll see that all corresponding bits are 1s, so we get a **yes**
- When we test for  $w$ , we get 2 collisions but one bit is 0
  - We're certain  $w$  is not there

# Several Applications

- In databases: keeping a “cache” in RAM before accessing a disk
  - If we **know** an item isn’t on disk, we spare a disk access
- Web caches: avoid storing data requested only once
  - Only cache things at the second time they’re asked

# Bloom Filters: Demo and Analysis

- Bloom filter demo:  
<https://llimllib.github.io/bloomfilter-tutorial/>
- See the other presentation
  - (if you ever tried formulas in LibreOffice, you'd know why)
- Cuckoo filters: <http://bdupras.github.io/filter-tutorial/>

# Distributed Hash Tables

# Better than Superpeers

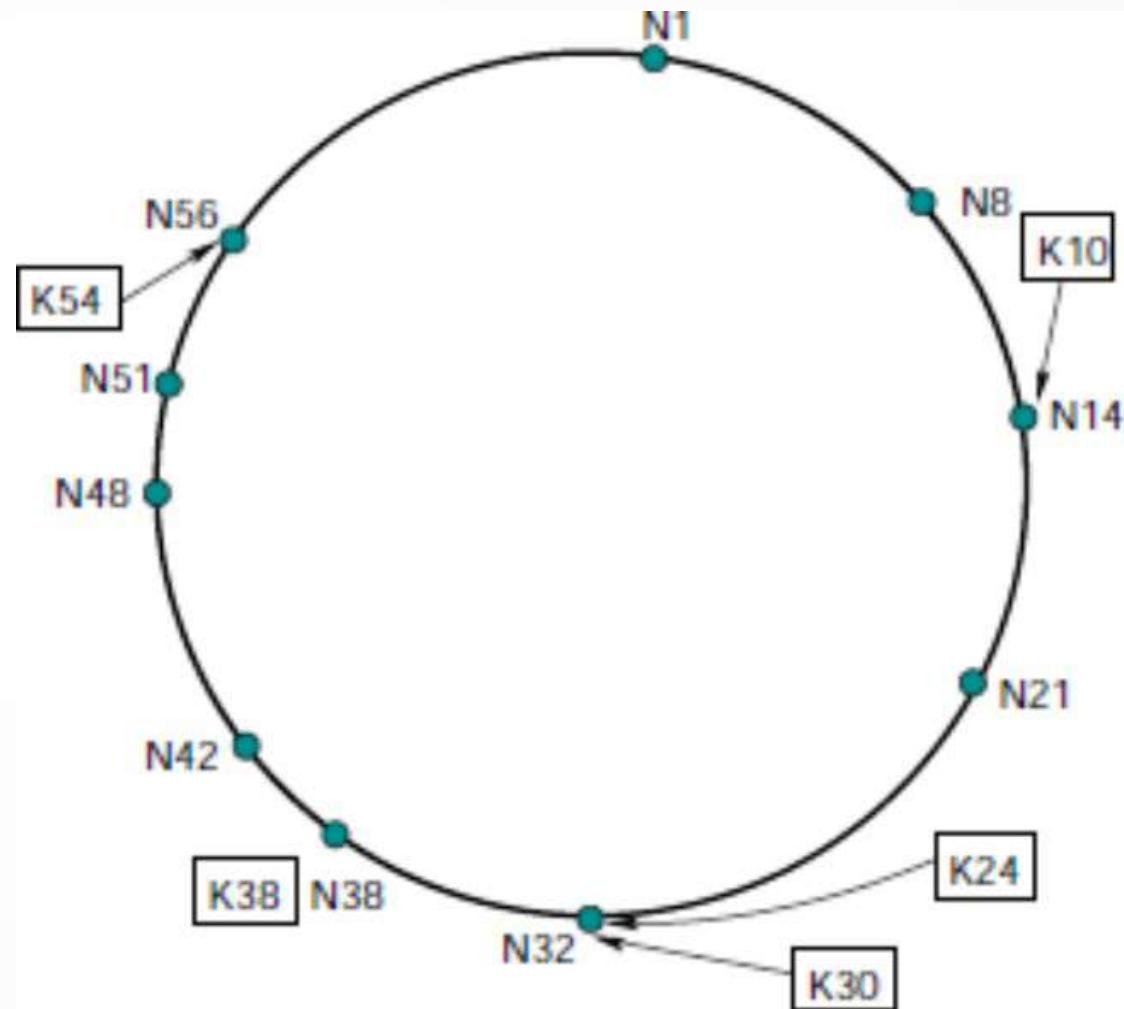
- **Distributed Hash Table (DHT)**: a decentralized system giving **efficient key-value lookups**
- Key idea: a **structured** peer-to-peer overlay
  - We choose **who connects to whom**
  - We use that freedom to **obtain efficient routing!**

# How DHTs Work

- Every peer handles **a portion** of the hash table
  - For redundancy, more than one peer per portion actually
- **Consistent Hashing**: adding or removing peers has a **small impact** on resource allocation (i.e., which data a peer stores)
  - Perfect to handle **churn**: nodes arriving and leaving all the time
- Item  $x$  will be stored on node corresponding to address  $h(x)$

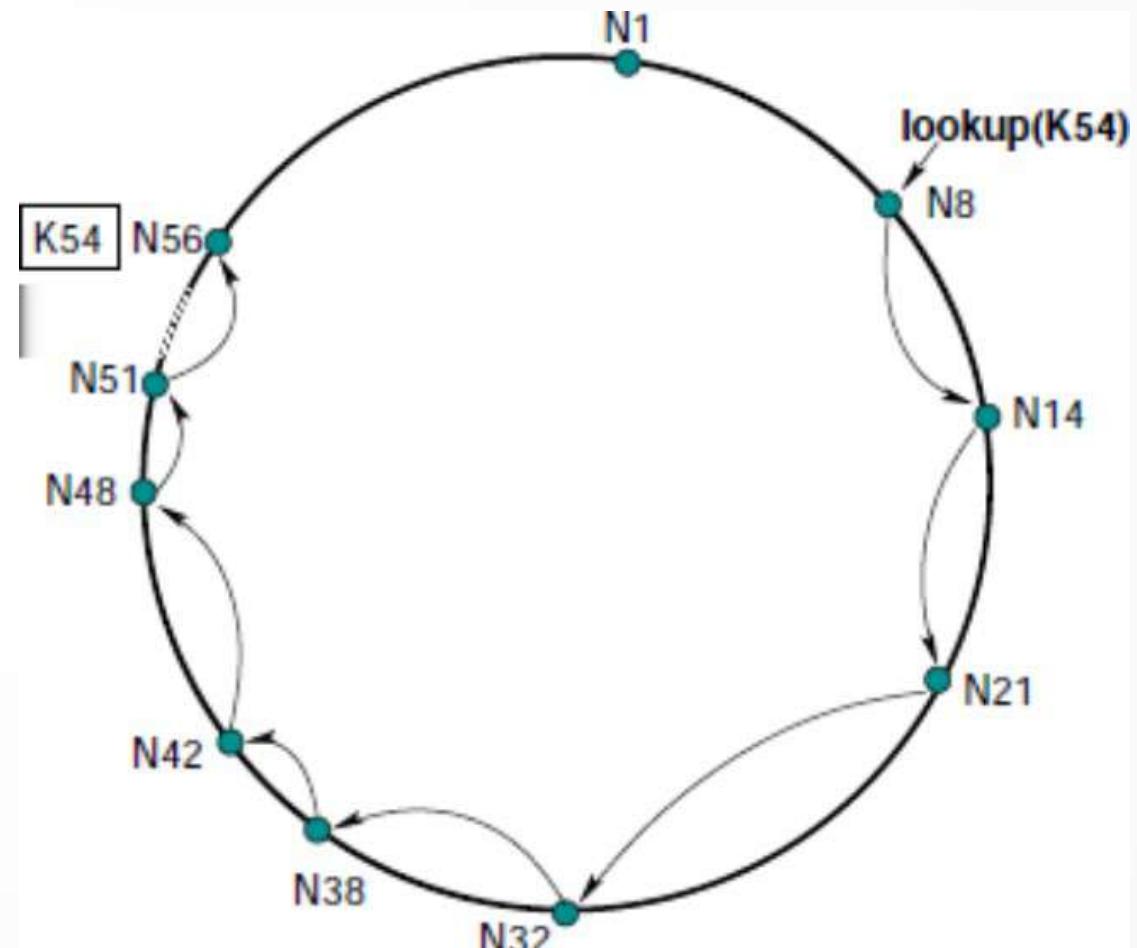
# The Chord Ring

- Reference: paper by Stoica et al.  
(ACM SIGCOMM '01)
- Nodes take random identifiers,  
and get in the ring with a link to  
predecessor and followers
  - In the example: identifiers in  $[0, 63]$
- Item  $x$  get inserted at the first  
peer with hash greater than  $h(x)$ 
  - ...and a few (e.g., 2) predecessors,  
for redundancy



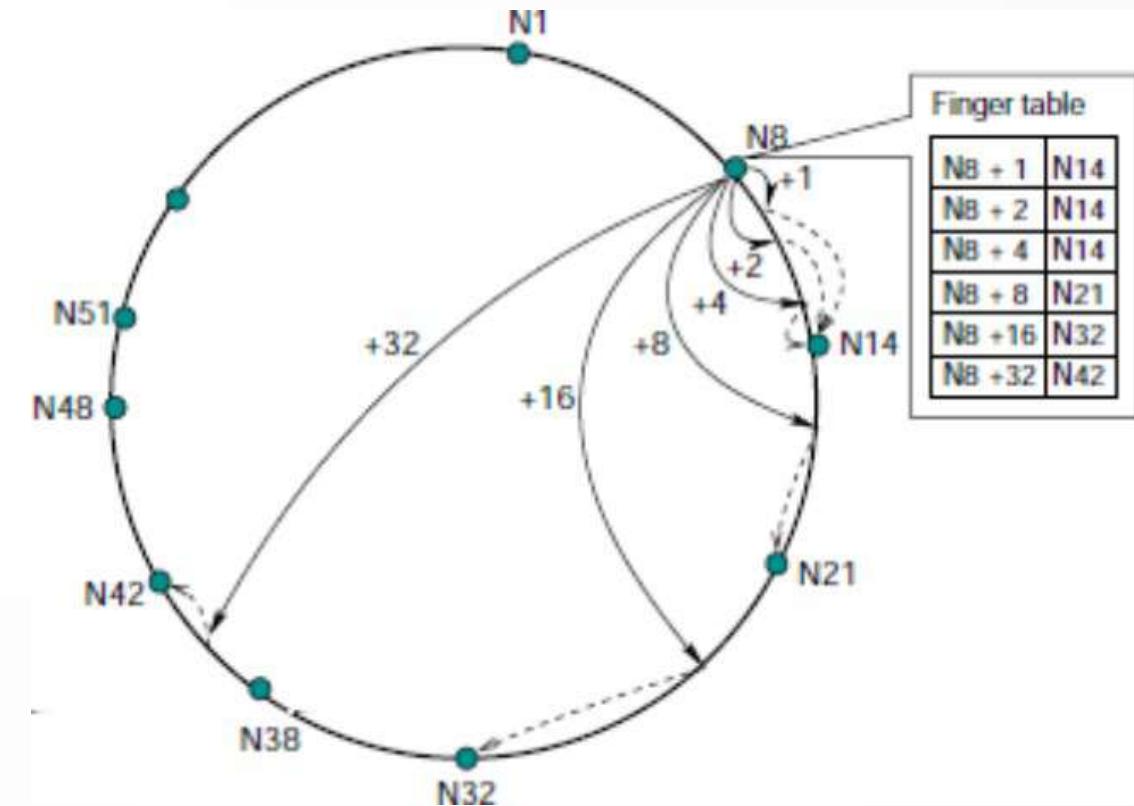
# Chord: Lookup, the Slow Version

- You can get to the node responsible for a given key by following successor link until you get to the destination
- Very slow, and breaks if any node in the middle disappears
- $O(n)$  hops



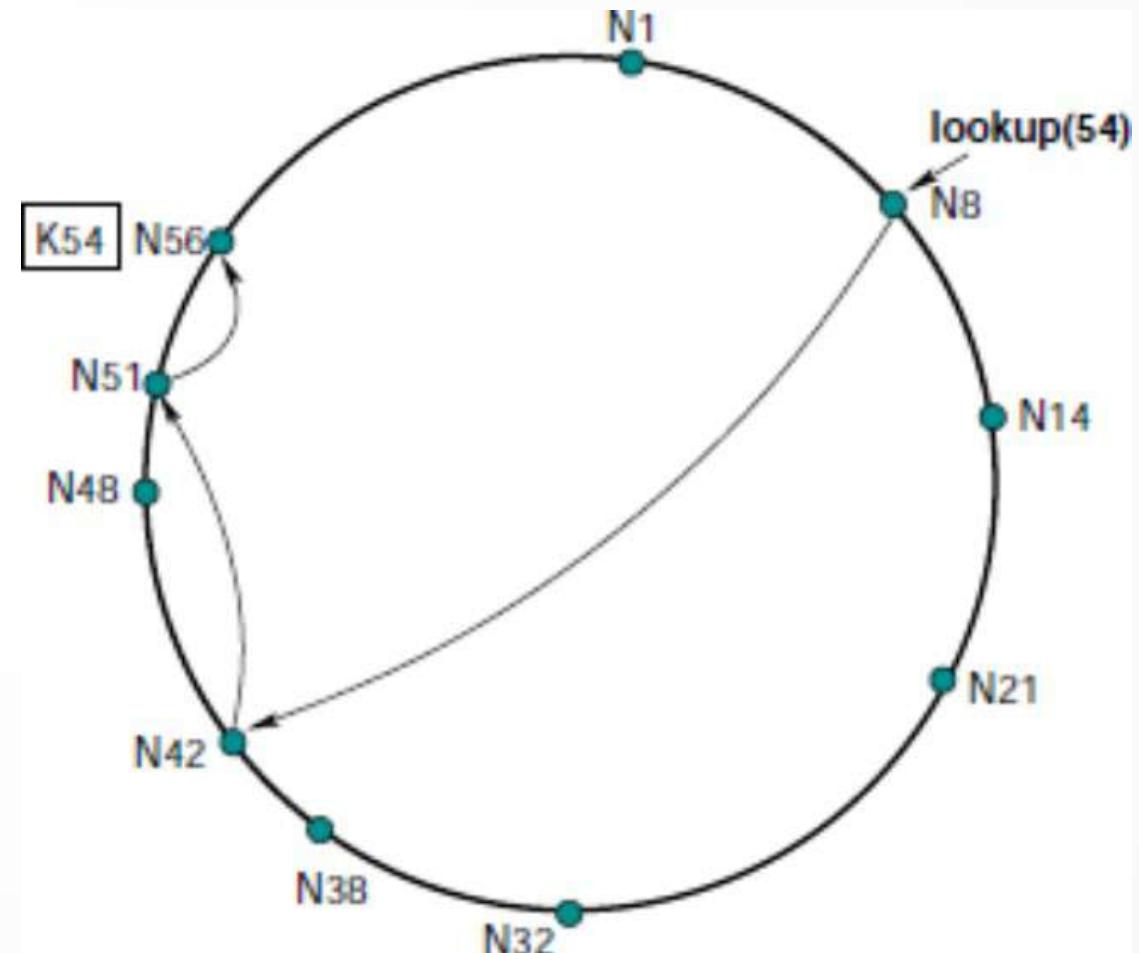
# Chord: Accelerated Lookup

- To get to destination faster, nodes save **shortcuts** ("fingers") to destinations at exponential distances
- In the example, powers of 2
- Node at position  $x$  can get finger at position  $x+y$  by looking it up on the network

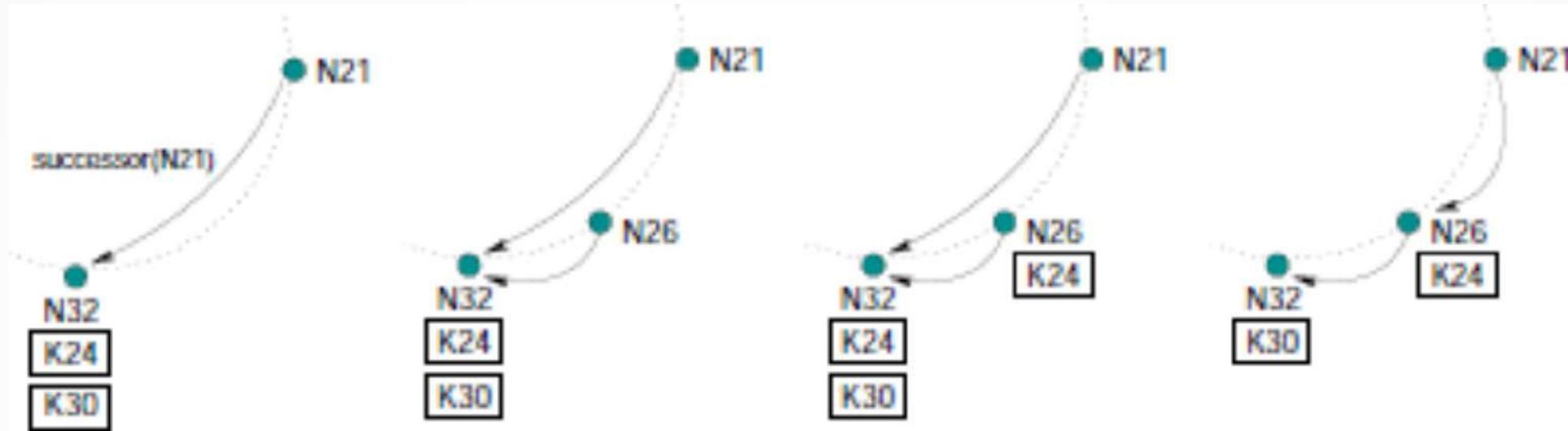


# Chord: Greedy Routing

- If at each step we follow the finger closest—but before—the destination, our lookup is **much faster**
- **How many steps?**
- $O(\log n)$



# Chord: New Nodes

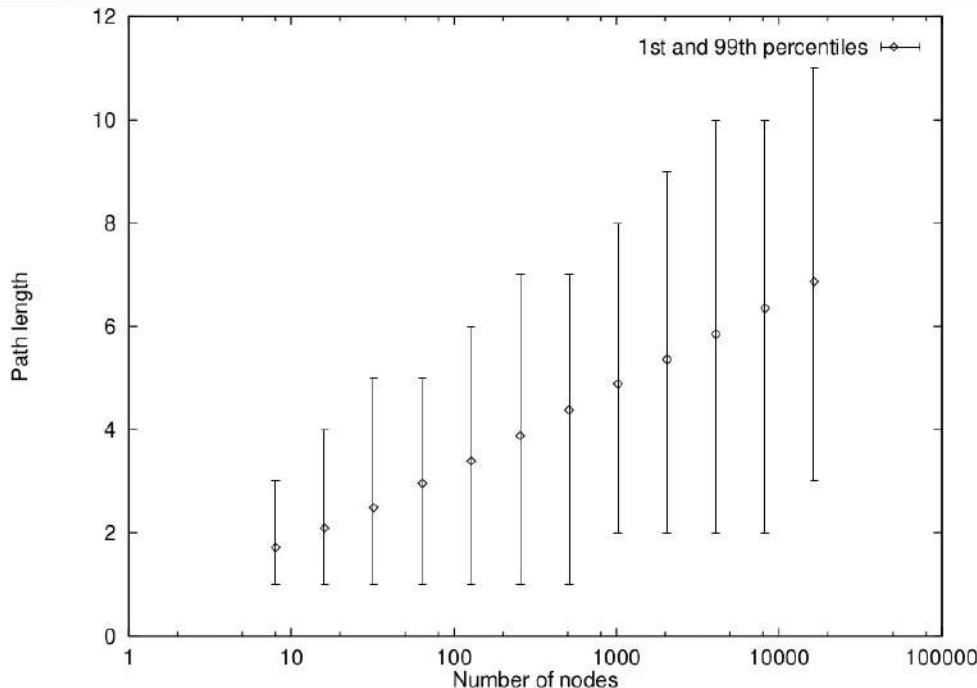


- Adding a new node just requires
  - A lookup
  - Exchanging data with two other nodes

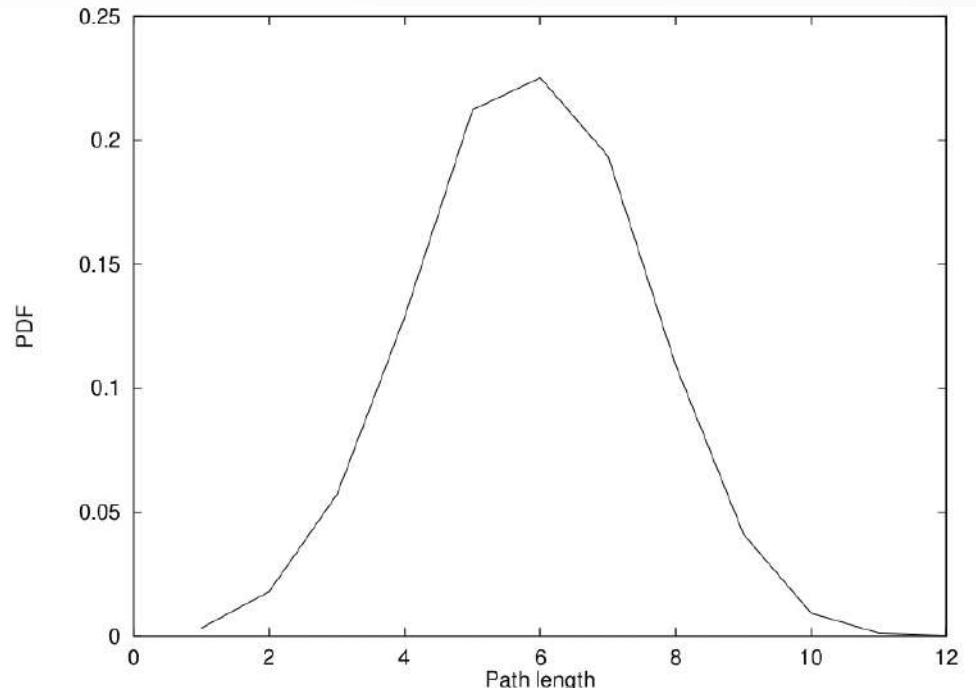
# Chord: Fault Tolerance

- To avoid breaking the successor chain when peers leave the system, nodes keep a list of  $r$  successors
  - If all of them leave the system, the ring can be broken
- A periodic **stabilization** procedure maintains the links
  - Predecessors and successors get pinged, fingers are re-queried

# Chord: Scalability



(a)



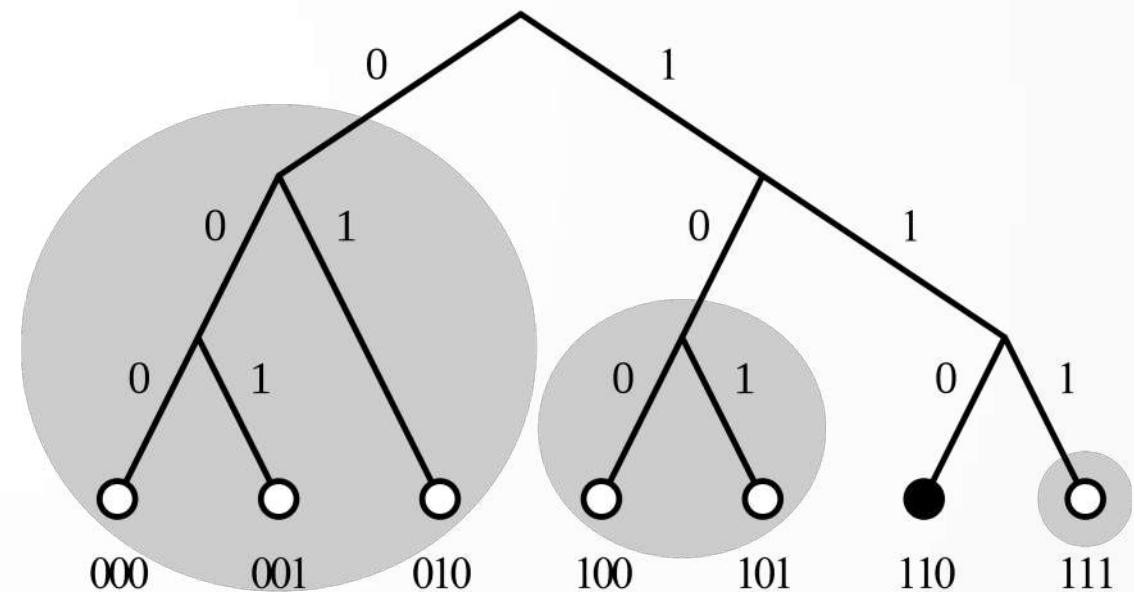
(b)

Figure 10: (a) The path length as a function of network size. (b) The PDF of the path length in the case of a  $2^{12}$  node network.

- Right:  $n=2^{12}=4,096$ ; average path length= $\log_2(n)/2=6$ 
  - Why?

# Kademlia

- Reference: [paper](#)
- The most used DHT in practice (e.g., BitTorrent)
- Very similar idea: logarithmic number of steps to get to destination
- Here, you get a finger table for nodes that have the first 0, 1, ..., k bit in common with you
- Added value: links are symmetrical, so you can exploit information about them when they reach you



# Distributed Computing

A-11. BitTorrent

# What BitTorrent Is

- A **file distribution system**
  - Basically, an alternative to HTTP (or FTP if you're old) for transferring sizeable files
- **File search is not a feature**
- Reference: the delightfully simple paper at P2PECON 2003

# Architecture

# Torrent File

- A .torrent file contains metadata about your file
  - Name
  - Size
  - Hashing information
  - The URL of a **tracker**
    - Or, if you're using the DHT (introduced in 2005), nothing :)

# The BitTorrent Swarm

- **Tracker:** a machine that helps node discover each other
  - Made not necessary by the DHT
- **Seed:** a node with a full copy of the file, just uploading
- **Downloaders:** nodes that have not finished downloading the file
- When downloaders complete a download, they can stay and seed

# Pieces

- Files are cut in **pieces** (256KB by default)
- Hashes of each piece are included in the .torrent file
- A node doesn't report having a piece until it verifies the hash
- Nodes contact each other, asking which pieces they have

# Piece Selection

- Which piece to select?
- **Rarest first:** always try to get the rarest piece in the system first
  - It will be the one that will be most difficult to find later
  - Guarantees that copies will stay around
- Exception: **random first piece**, to get something to upload ASAP

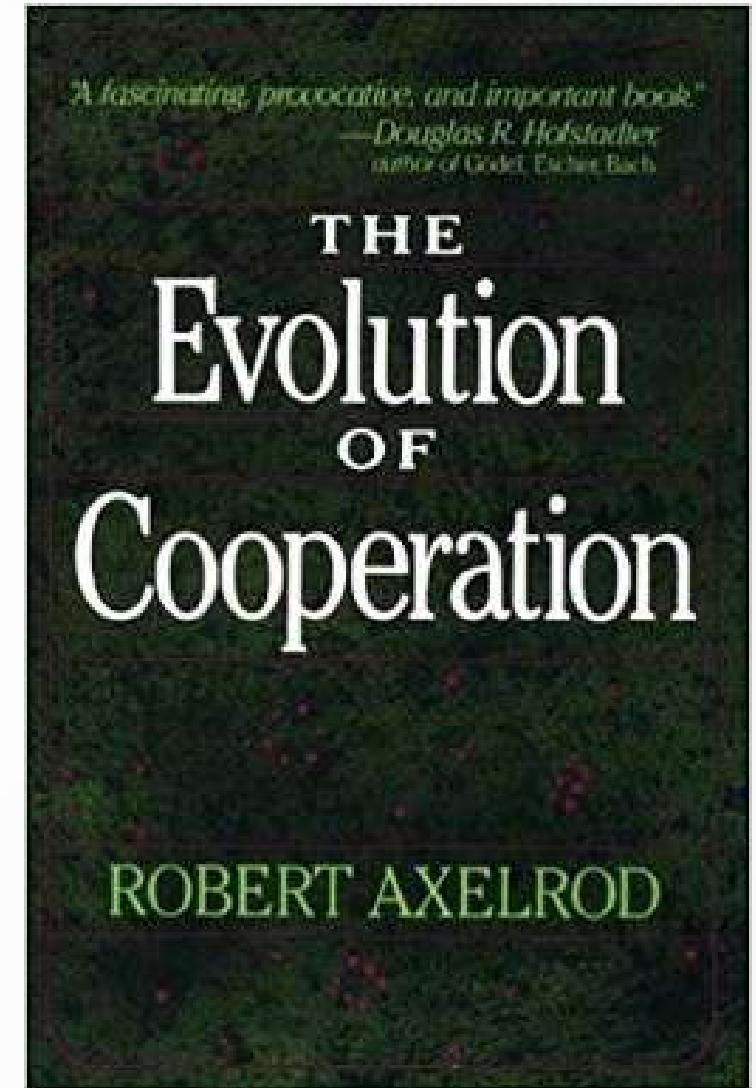
# Incentives to Cooperation

# Tit for Tat

- I behave with you as you did with me
  - “Pan per focaccia”, for the Italian speakers
- Among all the uploads open, most are *choked*—i.e., uploaders don’t send data through them
  - By default, 4 connections are unchoked
  - **Optimistic unchoke**: every 30s, give an upload slot to a random node
  - The others go to the peers that are sending data faster
- Result: to get fast downloads, you need to **upload fast**

# The Evolution of Cooperation

- Tit for Tat was probably inspired by a book on game theory
- In some cases, when rational entities have multiple reciprocal interactions (“iterated prisoner’s dilemma”), cooperation emerges as a successful strategy
- Tit-for-Tat is a bare-bone version of the concept of reputation: if you have a good reputation, I’ll be more friendly to you



# Distributed Computing

## A-12. Eventual Consistency

# Remember ACID and CAP?

- ACID: Atomicity, Consistency, Isolation, Durability
- CAP: Consistency, Availability, Partition Tolerance
  - Can't have all, choose two
- If you have a partition you can
  - Choose consistency (CP) and lose availability
  - Choose availability (AP) and lose consistency
  - Or some hybrid
- In the beginning of the course we've seen CP systems: if Paxos/Raft are partitioned, the minority will stop working
- Now, armed with what we've seen till now, we'll delve in AP systems

# **Eventual Consistency**

# Reference

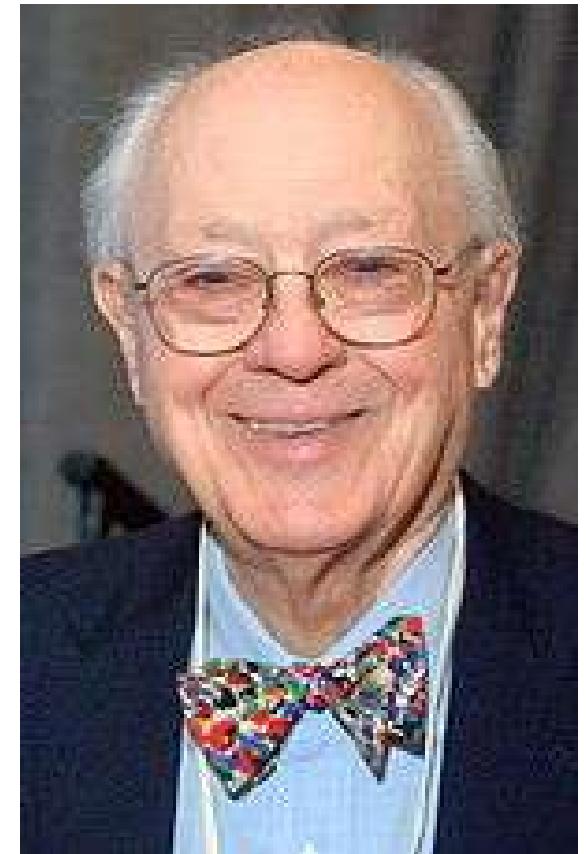
- Eric Brewer's talk **NoSQL: Past, Present, Future**, 2012.

# NoSQL before SQL

- Charles Bachmann, 1973 Turing Award
- IDS (Integrated Data Store),  
navigational database

```
get department with name='Sales'  
get first employee in set department-employees  
until end-of-set do {  
    get next employee in set department-employees  
    process employee  
}
```

Image by YMS at Wikimedia Commons, CC-BY 2.0



# 1970s: Relational vs UNIX

- Relational: top-down approach
  - Easy-to-grasp abstraction, one API does it all (SQL)
  - Declarative language, user doesn't care about optimizing
  - Data outlasts implementations
  - Transactions
- UNIX: Bottom-up
  - Few, simple, efficient mechanisms
  - Compose tools

# Two World Views

- Relational
  - Clean model, ACID transactions
  - Two kinds of developers: DB authors and SQL programmers
  - Do one important thing, do it well
- Systems
  - Bottom up, new modules to add functionality
  - One kind of programmer
  - Flexible systems that can grow to do new things

# Brewer's Story

- 1996-1998: build a search engine and a proxy cache
  - Didn't use a DBMS: custom servers on top of file systems were faster
  - Because DBMS's features cost performance
- 1997: ACID vs. BASE (Basically Available, Soft State, Eventual Consistency)
  - Not well received: people liked ACID
- 1999: CAP Theorem
- Mid-00's: Eventually consistent systems start to get used

# Partition Mode

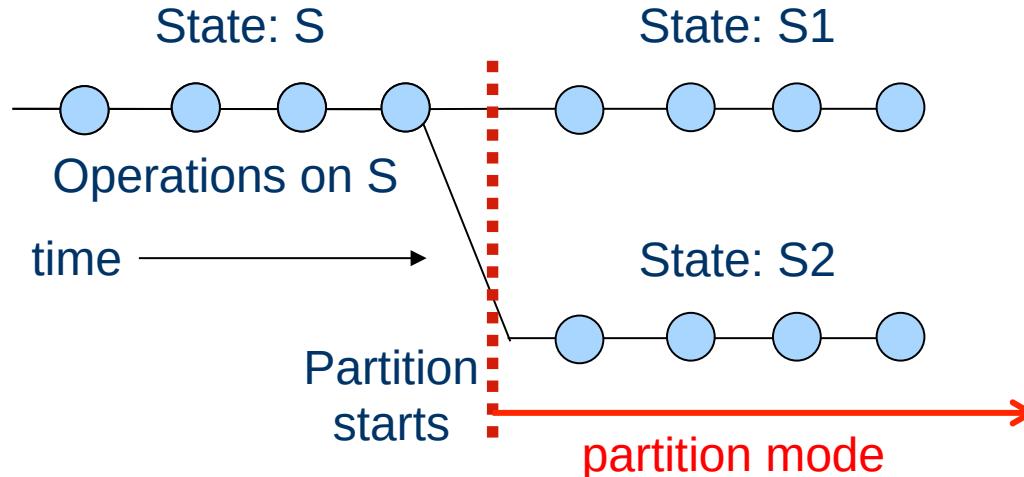
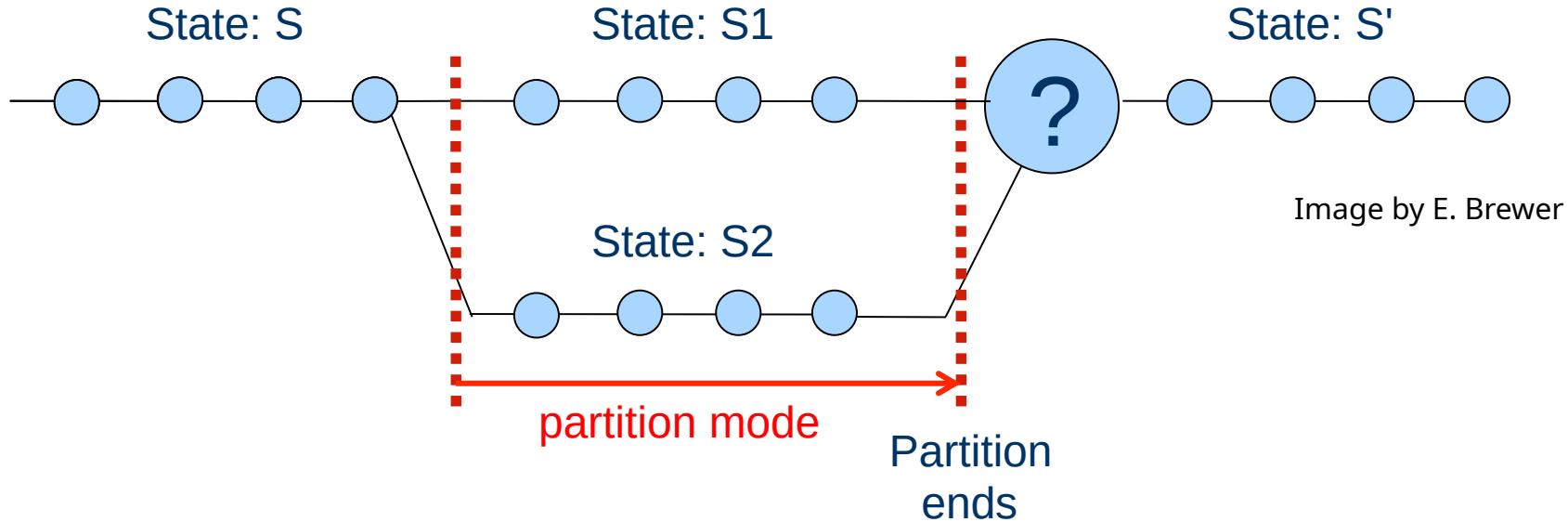


Image by E. Brewer

- Special mode the system can detect
  - Allow commits?
  - Give output?
  - Allow it to impact the outer world?

# Partition Recovery (1)



- How to get back to a consistent state afterwards?
  - Last writer wins?
  - Rules that depend on what has been done?

# Partition Recovery (2)

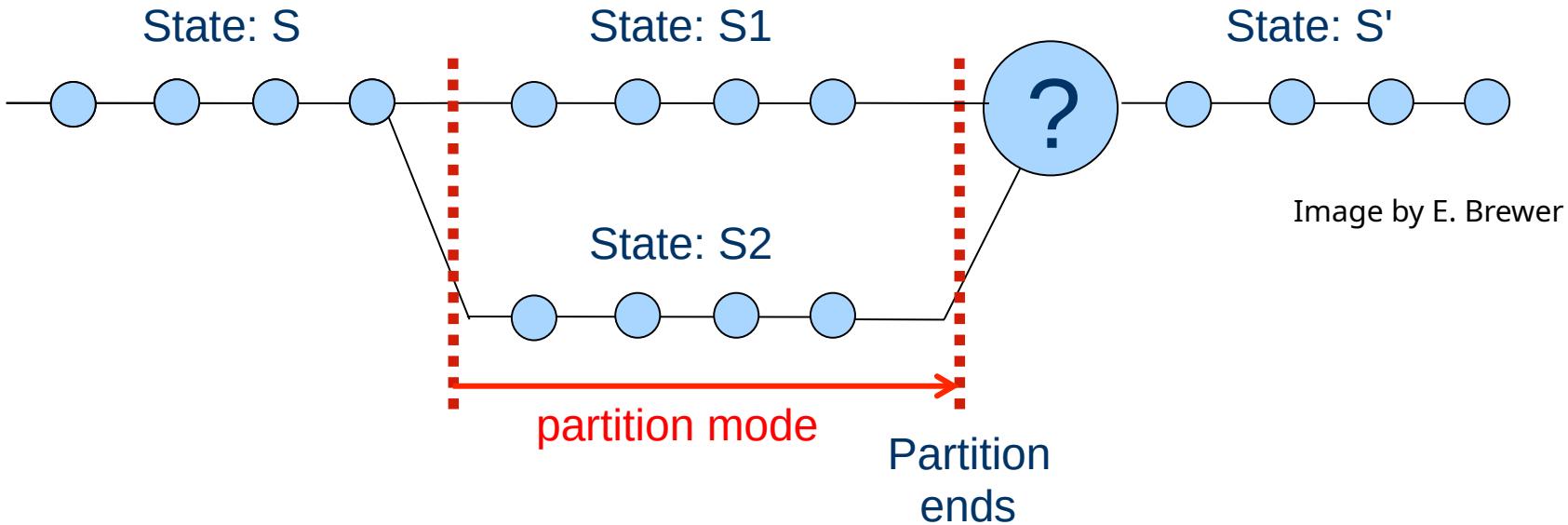


Image by E. Brewer

- Detect and repair mistakes
  - What have we done wrong?
  - How do we deal with it?

# ATMs and “Stand In” Time

- ATMs do keep operating when isolated from the network
  - “Partition mode”, indeed
  - Operations are **commutative**: increment, decrement
- Limit damage
  - E.g., give out at most 200€ per person
- Partition recovery:
  - Detect errors (balance below zero)
  - Compensate (overdraft penalty)

# Eventually Consistent Systems

- Three key issues to address:
  - Detect partitions
  - Define how “partition mode” works
  - Define how to do recovery
- The real world is eventually consistent!
  - There are “consistency rules” (laws, contracts, ...)
  - You see problems (inconsistency detection)
  - You compensate for it (money, ...)

# Amazon Dynamo

# References

- DeCandia et al.  
*Dynamo: amazon's highly available key-value store.* ACM SOSP 2007.
- Lakshman and Malik.  
*Cassandra - A Decentralized Structured Storage System.* ACM LADIS 2009.
- Cassandra documentation: *Dynamo*.

# About Dynamo

- Origin: handling shopping carts at Amazon
- Availability affects income! As available as possible
  - Trade off with consistency
- Born as a key-value store
  - Later evolved as a more complete DB, as usual
- Uses techniques seen in all the course
- Cassandra (Facebook, now Apache) has a very similar architecture

# Requirements

- “Always writeable”
  - You can always add an item to your shopping cart
  - Can write in partition mode
- *User-perceived consistency*
- Guaranteed latency measured at percentile **99.9**
- Parameters to tune cost, consistency, durability and latency
- Scale **out** to tens of thousands of servers
  - 2007: tens of millions requests, >3M checkouts in a single day

# Key Idea

- Chord in a datacenter (nodes are servers)
  - Consistent hashing: adding/removing one node at a time is cheap
- Completely decentralized
- Each item is replicated in the N nodes “after” a given key in the ring
  - Those nodes are called “preference list”
  - Replication guarantees durability

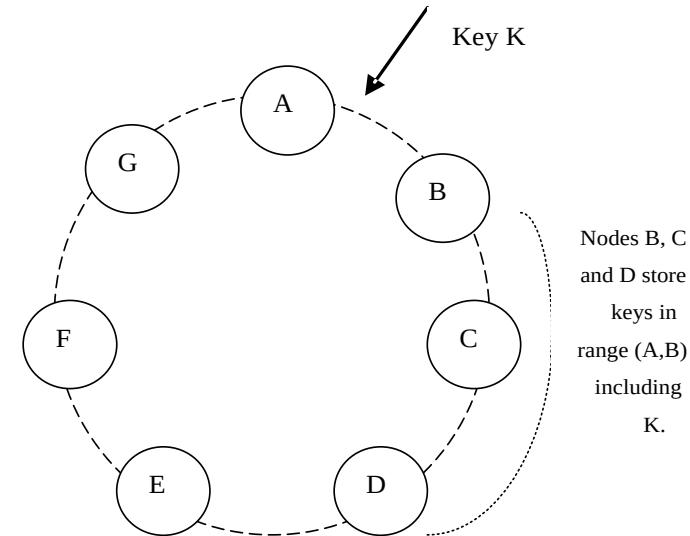


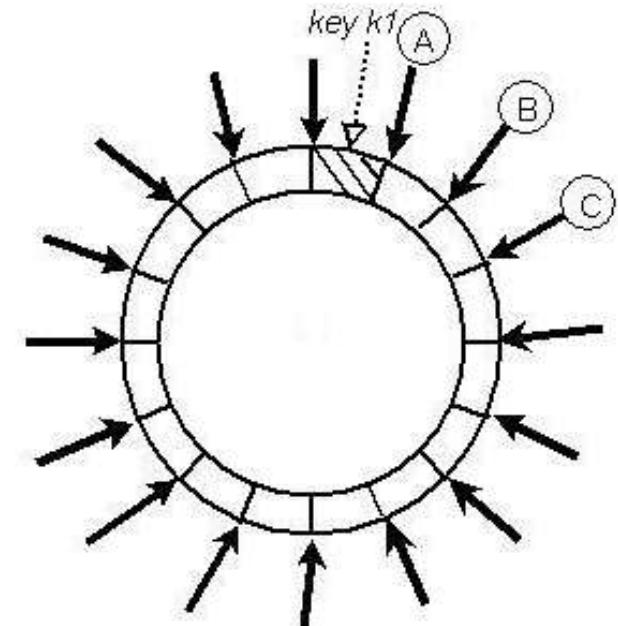
Image from DeCandia et al., SOSP '07

# Lookup

- Unlike Chord, every node knows the **full partition table**
  - One hop to get to any piece of data
  - Routing table updated via a **gossiping** algorithm: every node periodically exchanges information with a random set of other nodes
  - Gossiping also handles **failure detection**
- Optionally, the client knows the routing table as well
  - Data can be directly asked to the node having it

# Faster Partitioning

- Split the hashing space in Q partitions
- They get distributed equally between nodes
- When nodes join, they “steal” partitions from other nodes
- When they leave, they are redistributed to other nodes
- Transferring partitions **doesn't require random disk accesses**



# API

- `get(key) → [value], version_info`
- `put(key, value, version_info)`
- `get()` returns a **list** of values
  - May be more than one in case of inconsistencies
  - Will be handled by the client
- `version_info` is passed to the subsequent put to solve some inconsistencies
  - If something is created from scratch, `version_info` is null

# Sloppy Quorum

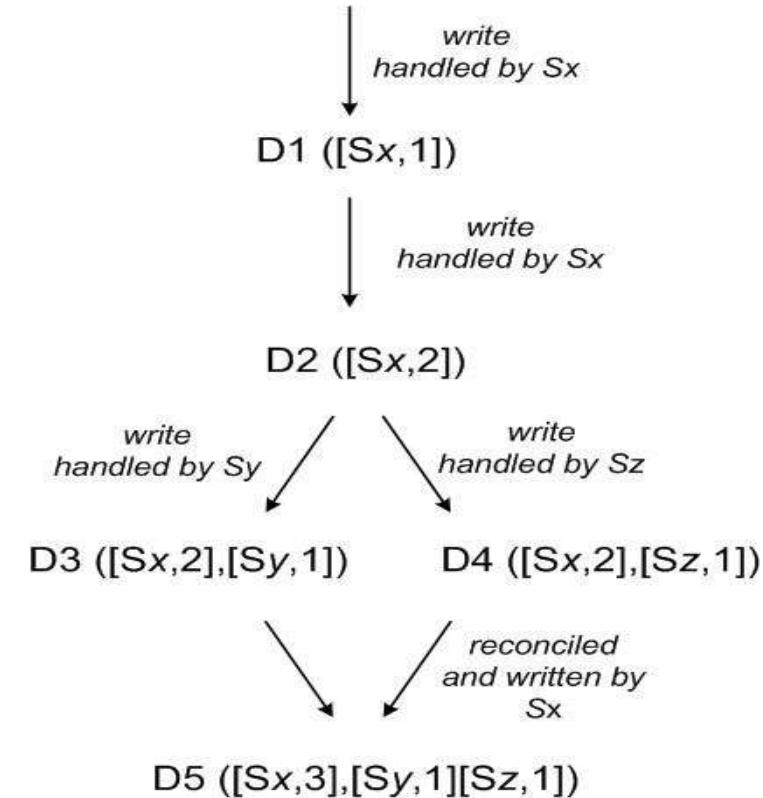
- Three configurable parameters:
  - N: number of copies for each piece of data (often, 3)
  - R: number of reads to get a successful read
    - Low R: fast read, high R: consistent
  - W: number of writes to get a successful write
    - Low W: fast write, high W: consistent
- If  $R+W > N$ , it is sort-of like a consensus algorithm, i.e., high consistency
  - Except failures

# Example configurations:

- N=3, R=2, W=2 (default)
  - Consistent & durable
- N=x, R=1, W=x
  - Slow writes, fast reads (great for read-intensive workloads)
- N=R=W=1
  - Cache (e.g., web cache)

# Solving (Some) Inconsistencies: Vector Clocks

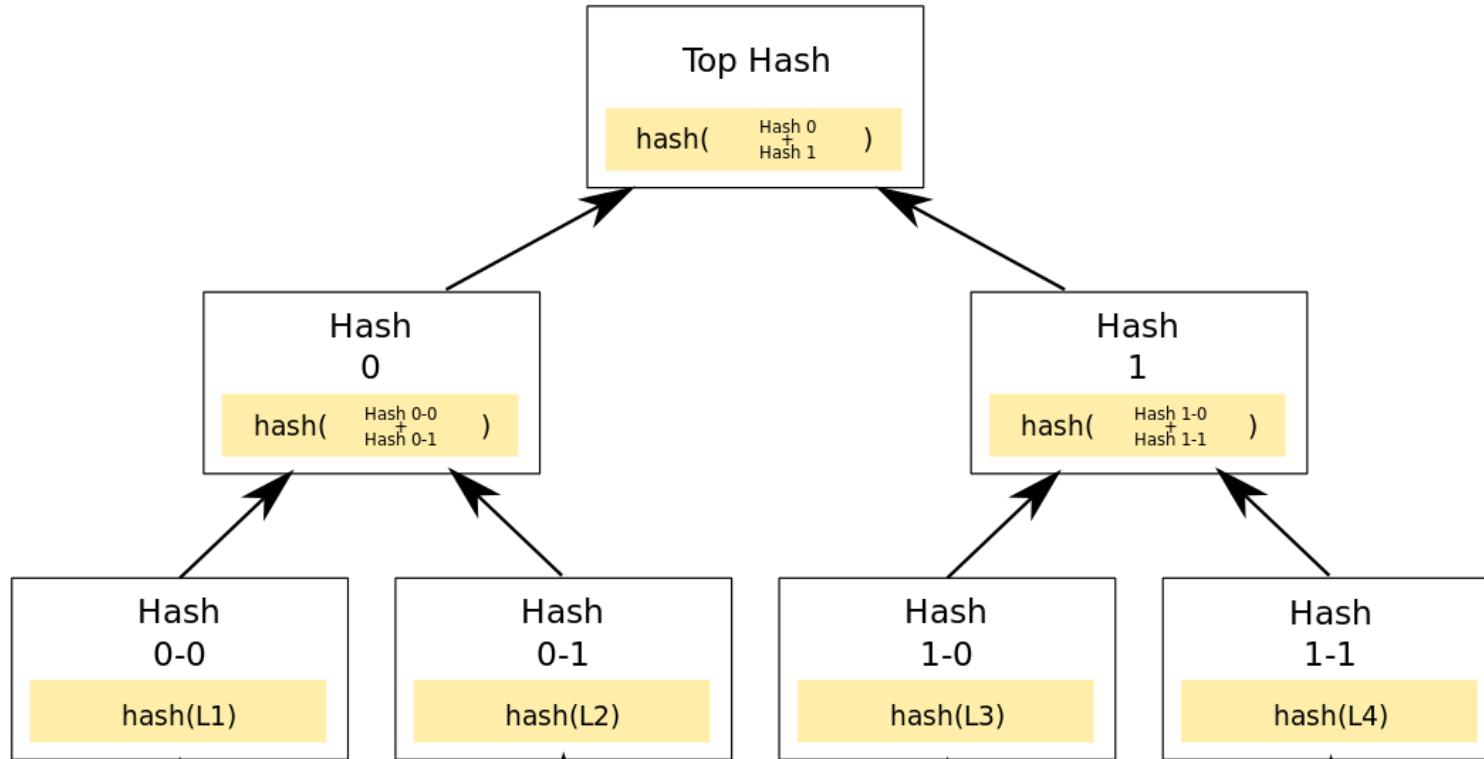
- `version_info` gets a counter value for each machine they have passed through
  - Idea from 1986 (Ladin and Liskov)
- One copy supersedes another if counters are not smaller for each machine
- Otherwise, they're independent and we ask the client what to do



# Failures

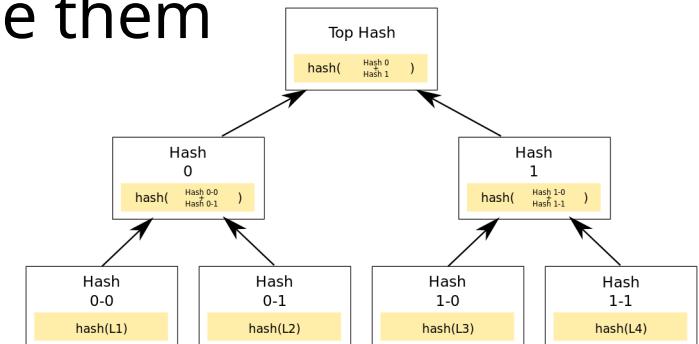
- When machines go offline, it's considered transient
  - Permanent addition or removal is an administrator action
- Reads and writes spill over to the first machine in the ring after the N that should handle them by default
- When the machine comes back online, updates are reported to it
- Can create some rare inconsistencies even when  $R+W > N$

# Merkle Trees



# Anti-Entropy

- Merkle trees are used to compare data between nodes that store replicas of a partition
- If the root is different, compare the children to find out which half is different, and so on recursively
- Fast way to spot differences & reconcile them



# Client-Side Reconciliation

- If everything else fails, the client is presented with more than one return value
- What to do looks depends on the application
  - Amazon cart policy: in doubt, leave stuff it in the cart!
  - We've seen what an ATM would do
- Can be reminiscent of exception handling
- Rare: Amazon reports 0.06% cases of more than 1 value returned

# Other Optimizations

- Buffered writes: wait for a few writes to be committed before writing to the disk
  - Performance/consistency tradeoff
- Throttling background operations
  - Slow down gossip/maintenance when many requests are around
- Let coordinate read/writes to nodes who are responding fastest
  - Additional load balancing

# Some numbers from the paper

- Tens of thousands of machines
- Tens of millions of requests, >3M checkouts in a day
- Response time below 400ms at 99.9 percentile (avg below 40)
- In 99.94% cases, requests return exactly one version
- 99.9995% successful responses without timeouts
  - Equivalent to **2.5 minutes** of unavailability in a **year**

# What About Cassandra?

- Project by Facebook, now handled by the Apache foundation
- Very similar architecture
- Zookeeper for routing table and seeds
- Rack-aware & datacenter-aware data placement
  - Again uses Zookeeper to elect a leader and coordinate it
- No vector clocks, just get a timestamp, and the latest wins
- Lightly-loaded nodes get “migrated” on the ring

# Distributed Computing

A-13. MapReduce

# Big Data

- What does this mean?
  - Web
  - Physics/Astronomy/Finance data...
- 3 Vs:
  - Volume
  - Velocity
  - Variety
- Realization: **data often counts more than the algorithms used**

# The MapReduce Programming Model

- A programming model inspired by
  - Functional programming
  - Bulk Synchronous Parallelism (BSP)
- Execution frameworks
  - For large-scale data processing
  - Designed to run on “commodity hardware”
    - i.e., medium-range servers

# References

- *MapReduce: Simplified Data Processing on Large Clusters*—see [paper](#) by Dean and Ghemawat (Google) at USENIX OSDI '04
- The [lesson](#) in the course by Pietro Michiardi (EURECOM); and figures thanks to him. Thanks!
- [Data-Intensive Text Processing with MapReduce](#), book by Jimmy Lin and Chris Dyer

# Principles

# Scale Out, Not Up

- Many “commodity servers” are preferable to few high-end ones
  - Cost grows more than linearly with performance
- In some cases, a big enough server just doesn’t exist
  - Google: **estimated** at 15 Exabytes in 2013  
(15,000,000,000,000,000 bytes)
  - Internet Archive: **50 PetaBytes** (2014)

# Looking at the Bottlenecks

- In many workloads, **disk I/O is the bottleneck**
  - Reading from a HDD: around 100 MB/s
  - SSDs: 400MB/s—6.5GB/s
  - Ethernet: 100-10,000Mb/s—i.e., 12.5 to 1250 MB/s
  - RAM: ~20 GB/s (DDR4), ~40-50GB/s (DDR5)
- We want to **read from several disks at the same time**

# Avoiding Synchronization

- **Shared Nothing** architecture:
  - Independent entities, with **no common state**
  - Synchronization introduces latencies, and it is difficult to implement without bugs
  - A goal is to **minimize sharing**, so that we synchronize **only when we need to**

# Failures Are the Norm

- When you have a cluster that's big enough, failures are **the norm, not the exception**
  - Hardware, software, network, electricity, cooling, natural disasters, attacks...
  - Cascading failures when the failure of a service makes another unavailable
  - Most failures are transient (data can be eventually recovered)

# Move Processing to the Data

- High-Performance Computing (HPC):
  - Distinction between performance & storage nodes
  - **CPU-intensive** tasks: computation is the bottleneck
- **Data-Intensive** workloads:
  - Network (if not the disks) is generally the bottleneck
  - We want to process the data **close to the disks where it resides**
  - **Distributed filesystems** are necessary, and they need to **enable local processing**

# Process Data Sequentially

- Data is **too large to fit in memory**, so it's **on disks**
- We've seen 100MB/s for a HDD—that's for **sequential reads**
  - Disk seeks for random disk access make everything **much slower**
- Consider a 1TB DB with  $10^{10}$  100-byte records
  - Updating 1% of the records with random access will require around **a month** (seek latency ~30ms)
  - Rewriting all records will require around **5-6 hours** (at 100 MB/s)
- There's a big advantage in **organizing computation for sequential reads**

# What MapReduce is For

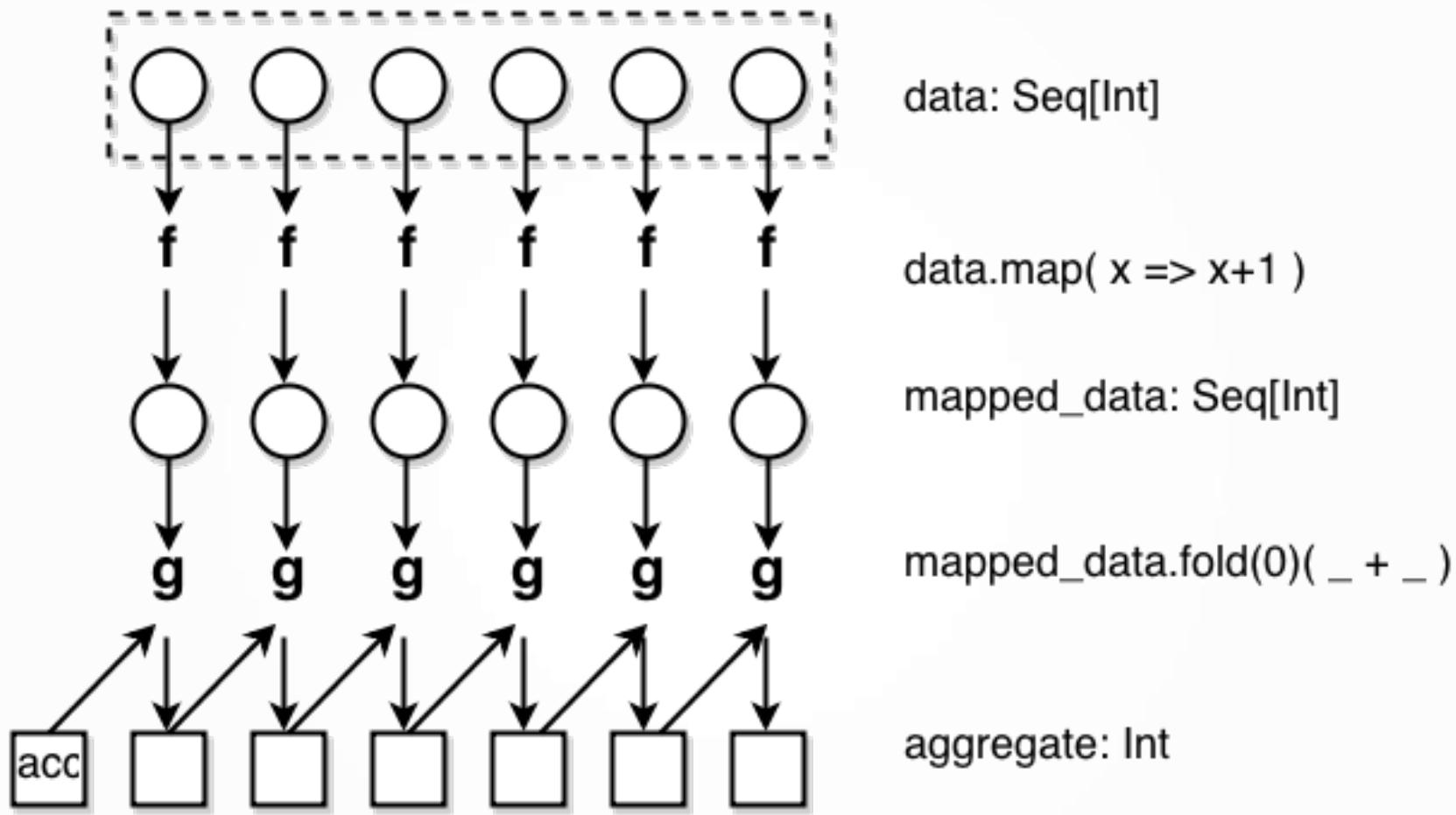
- Batch processing involving (mostly) **full scans** of a dataset
- Data collected elsewhere and copied to a distributed filesystem
- Examples:
  - Compute PageRank, a score for the “reputation” of each page on the Web
  - Process a very large social graph
  - Train a large machine learning system
  - Log analysis

# Scalability Goals

- In two dimensions:
  - **Data**: if we double the data size, the same algorithm should ideally take around **twice** as much the time
  - **Resources**: if we double the cluster size, the same algorithm should ideally run in around **half** the time
- **Embarassingly parallel** problems: shared-nothing computations that can be done separately on fragments of the dataset
  - E.g., convert data items between formats, filter, etc.
- Exploit having **embarrassingly parallel sub-problems**

# Programming Model

# Functional Programming Roots



- **Map and reduce (or fold):** higher-order functions
  - Accepting functions as arguments

# Functional Map

- Takes a sequence as input
- Apply a single function  $m$  to each element of your dataset
- Produce a new sequence as output
- Example:  $\text{map}(\text{neg}, [4, -1, 3]) = [-4, 1, -3]$

# Functional Reduce

- Given a list  $l$  with  $n$  elements, an initial value  $v_0$  and a function  $r$ , the output we can compute
  - $v_1 = r(v_0, l_0)$
  - $v_2 = r(v_1, l_1)$
  - ...
  - $v_n = \text{reduce}(r, v_0, l) = r(v_{n-1}, l_{n-1})$
- For example,  $\text{sum}(l) = \text{reduce}(\text{add}, 0, l)$
- Can be seen as an **aggregation operation**

# The MapReduce Model

- Dean and Ghemawat, engineers at Google, discovered that their scalable algorithms followed this pattern
  - A “map” part where original data is transformed, on the machines that were originally holding the data
  - A “reduce” part where the first results are aggregated
- The MapReduce framework facilitated writing programs with this style
- Implemented as free/opensource in Apache Hadoop MapReduce (originally developed at Yahoo!)

# MapReduce Map Phase

- Processes data **where it's read**
  - Filter what's not needed so you don't **waste network bandwidth sending it**
  - **Transform data** (e.g., convert to the format that's best for your computation)
  - Unlike the functional *map*, this always creates key/value pairs
  - **Embarassingly parallel**: each “fragment” of input determines its own output, alone

# Shuffle Phase

- Data gets **grouped by key**, so that we get a sequences of all values **mapped to the same key**
  - Handled by the execution framework (Hadoop, Google MapReduce), so programmers don't have to do anything
  - Yet, there are optimizations possible visible to the users
  - Data gets moved on the network
  - If data is well distributed along keys, work is well distributed between machines

# Reduce Phase

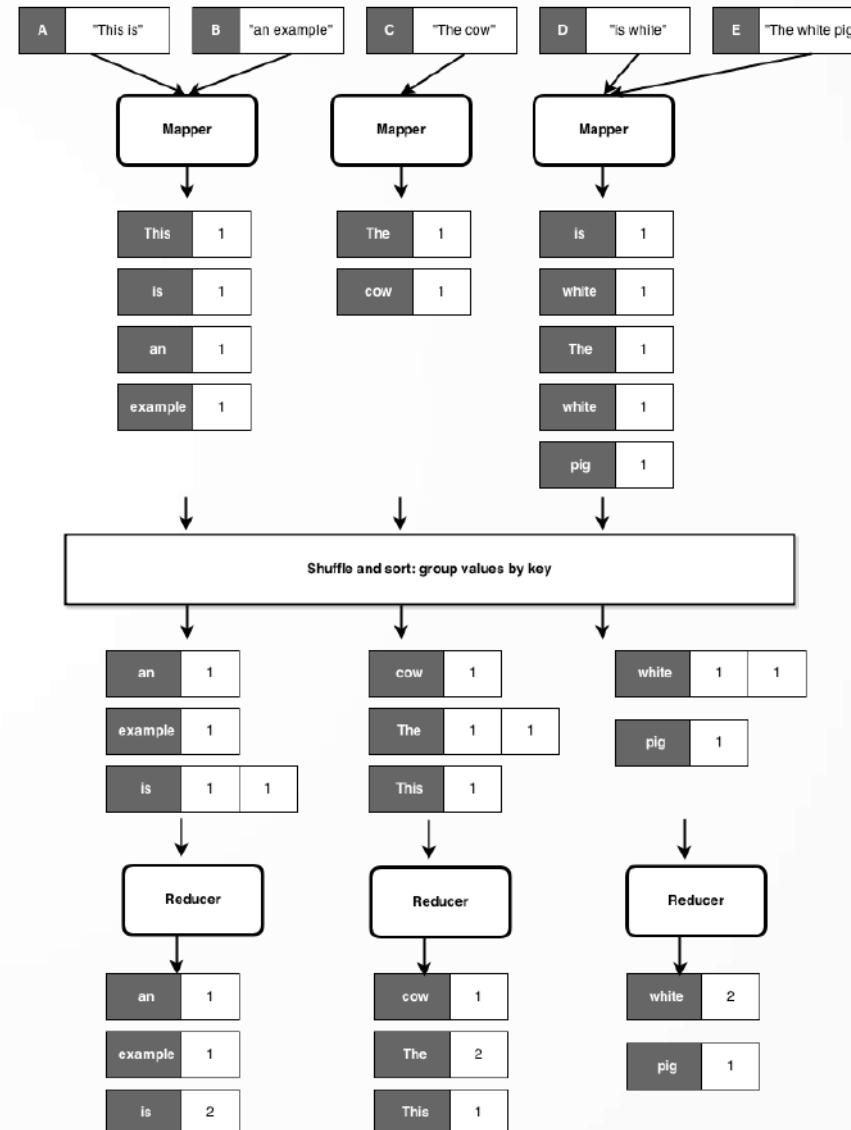
- **Reduce phase:** an **aggregation operation**, defined by the user, is performed on all elements having the same key
- The output is written on the distributed filesystem
- This output can be an **input to a further map-reduce step**

# WordCount: MapReduce's Hello World

```
def map(text):  
    for word in text:  
        emit word, 1
```

```
def reduce(word, counts):  
    emit word, sum(counts)
```

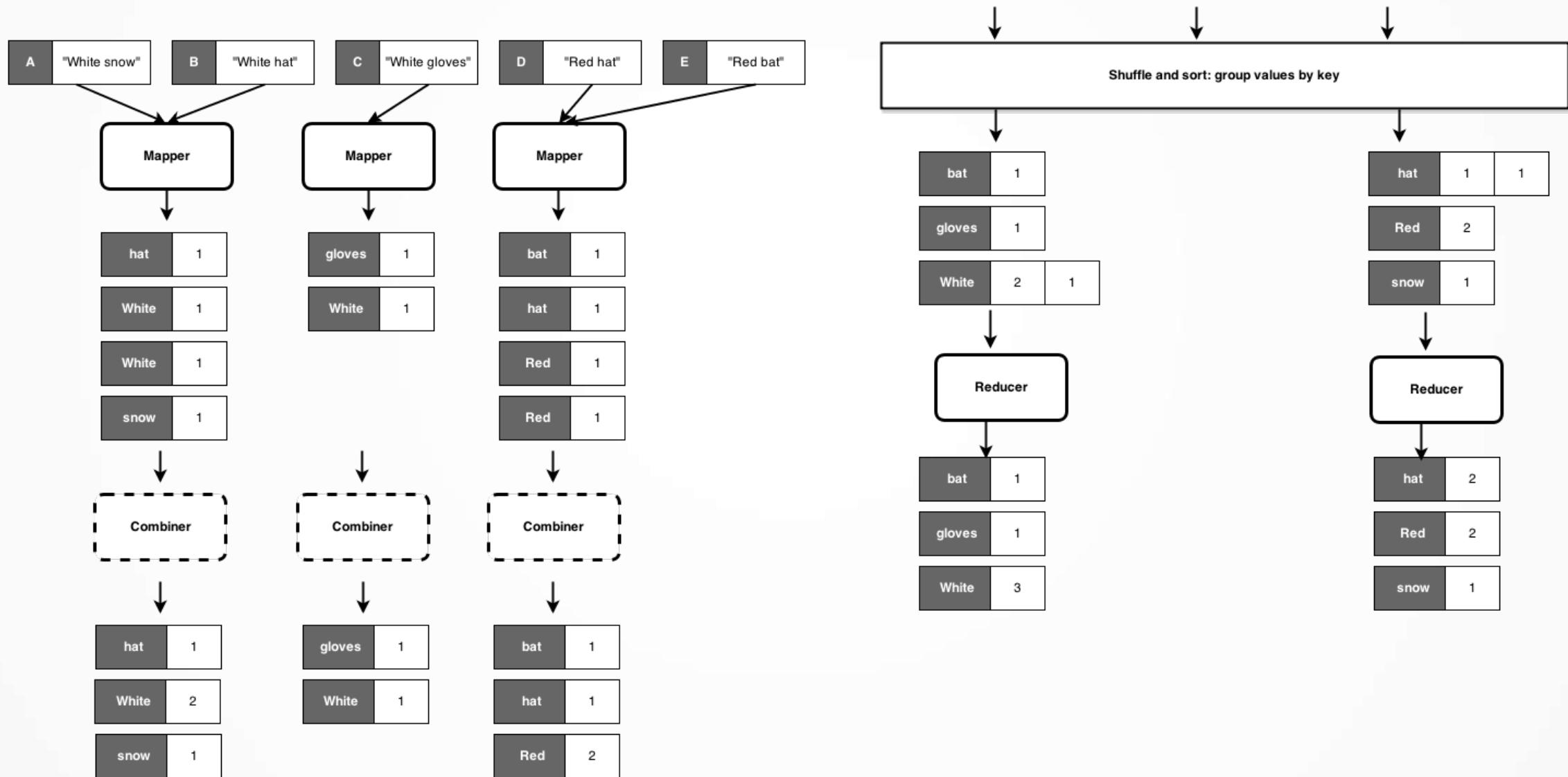
- We'll implement it in a simulated framework
- Run it on the **Moby Dick** text



# Combiners

- A way to reduce the amount of data before sending it over the network
- They are “mini-reducers”, run on mapper machines to pre-aggregate data
- In Hadoop they’re **not guaranteed to be run**, so the algorithm must be correct without them
- We’ll write a combiner for our WordCount

# Combiners in WordCount



# Exercise

- Write a “MapReduce program” for computing the per-team mean “overall” stat of players in **FIFA 21**
- Then, try adding a combiner
  - Note:  $\text{mean}(1,2,3,4,5)$  **is not**  $\text{mean}(\text{mean}(1,2,3), \text{mean}(4,5))$
  - Hence, the combiner cannot output partial means
  - The algorithm should work without combiners
  - Hint: try outputting  $(k, (\text{sum}, n\_items))$  from mappers and combiners

# What Can We Do With MapReduce?

- “Everything”
  - Trivially, we could send everything to a single reduce function and compute anything there
  - Would scale terribly, of course
- The question becomes: what can we do **efficiently**?
  - It’s about finding scalable solutions
  - This is non-trivial! It’s about optimizing computation, communication, and sharing costs well
- Many algorithms require **multiple rounds** of MapReduce

# Patterns

# Co-Occurrence Matrices

- Problem: building a co-occurrence matrix
  - $M$ , a square  $n \times n$  matrix, where  $n$  is the number of words
  - A cell  $m_{ij}$  contains the number of times word  $w_i$  occurs **in the same context** of  $w_j$  (e.g., appear in the same sliding window of  $k$  words)
- A building block for more complex manipulations
  - E.g., Natural language processing (NLP)
  - Similar problem: recommender systems
    - “*Customers who buy X often also buy Y*”

# Is the matrix too large?

- $M$  has size  $n^2$ : it can become very big quickly
- English: hundreds of thousands of words
  - i.e., tens of billions of cells
- Other use cases: billions
  - i.e., forget about it
- Most of those cells will anyway have a value of 0
  - Let's just compute the nonzero values!

# The Pairs Approach

- Use **complex keys**: when the mapper encounters  $w_1$ , close to  $w_2$ , it will emit the  $((w_1, w_2), 1)$  pair, meaning “I've found the  $(w_1, w_2)$  pair once”
- From there on, it really looks similar to WordCount :)
- Let's do it as an exercise!

# The Stripes Approach

- Say we have words  $[b, c, b, d]$  in the context around word  $a$
- The mapper will return  $(a, \{b: 2, c: 1, d: 1\})$ : we associate to the key  $a$  a mapping to all the words corresponding non-empty columns in a matrix row
- The combiner and the reducer will aggregate each of the stripes

# Distributed Computing

A-14. Hadoop Design

# Apache Hadoop

- If you don't work at Google, Hadoop is the software suite you're likely to use if you have a large dataset
- Free-Open Source, Apache License
- Handled by the Apache Foundation
- Based on Java
- A large ecosystem
- We'll see the parts that deal with MapReduce

# Credits

- Again, thanks to Pietro Michiardi of EURECOM. Many diagrams thanks to him.

# **HDFS: the Hadoop Distributed FileSystem**

# Move Computation to the Data

- We have seen that MapReduce is based on
  - Performing a **map** phase wherever data is read
  - A **shuffle** phase to move around processed data
  - A **reduce** phase to aggregate it
- HDFS is designed to **enable** this kind of computation
  - For nodes that do **both** storage and computation
  - Inspired by GFS, the Google correspondent
  - See the **paper** for more information

# HDFS Principles

- Large datasets, that can't be stored on a single machine
  - Each “file” is partitioned in several machines
  - Network-based, with all the complications
  - Failure-tolerant
- One distributed filesystem design, tailored to
  - Read-intensive workloads (many reads for a write)
  - Throughput, not latency (sequential reads)
  - Commodity hardware

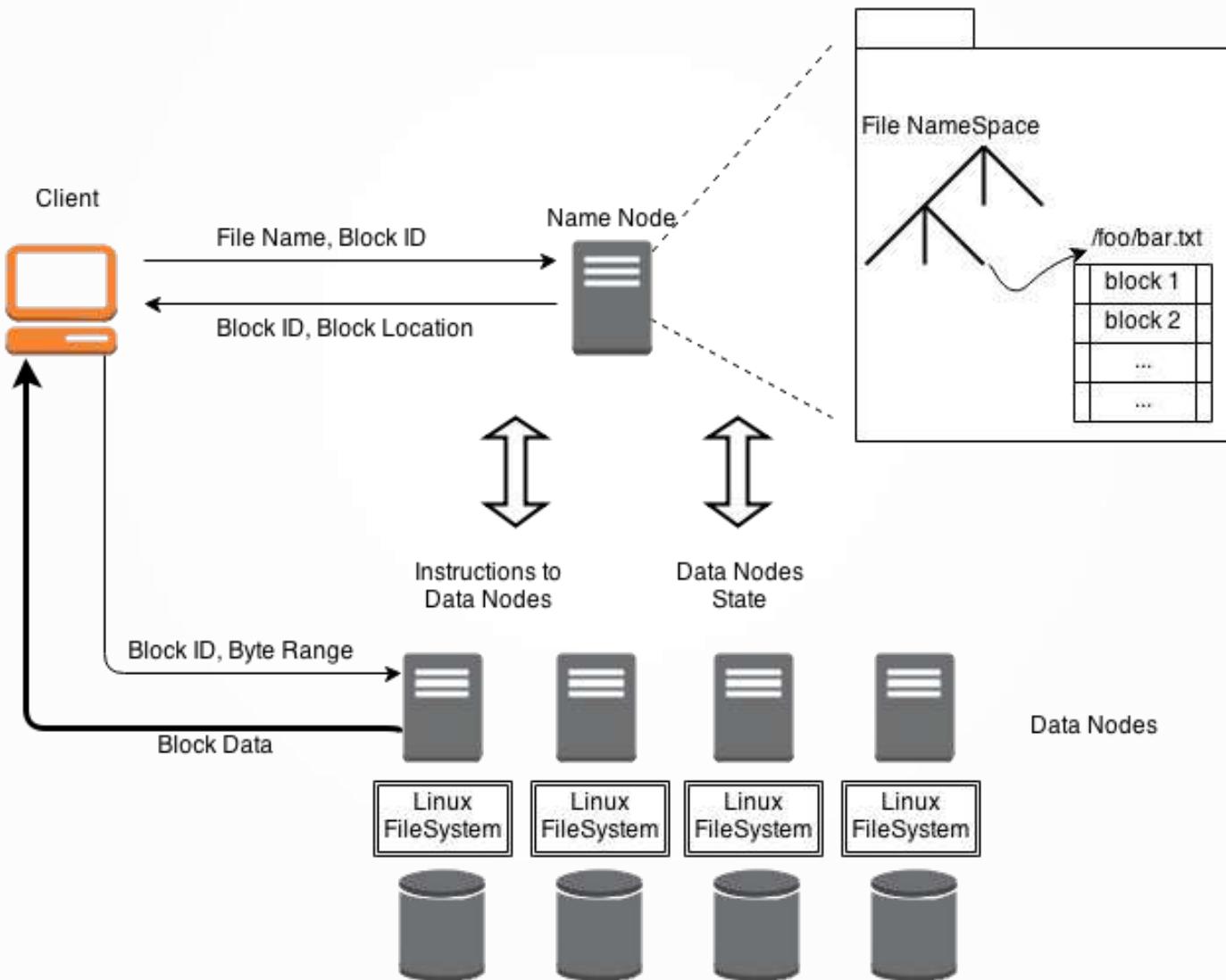
# HDFS Blocks

- Big files are broken in chunks
  - Big chunks! Default is 128 MB
  - Unrelated to space used on disks (a 1MB file doesn't use 128 MB): blocks are stored as files on the native filesystem
- Blocks are **replicated** in different machines
  - Q: Why not using **erasure coding**?
  - A: Because you can run processing right away (map!)
- Q: Why are blocks so large?
  - A1: To make seek times small compared to read. Consider 10ms seek and 100MB/s read, for a 100MB block seek time is 1%
  - A2: To ease handling metadata. We'll see right away.

# HDFS Nodes

- **NameNode:** keeps metadata in RAM
  - Directory tree, and index of blocks per file (around 150B/block)
  - Metadata is around **1M** times smaller than the dataset – 1GB of RAM can index 1PB of data
  - The load of NameNode is kept manageable exactly because there aren't that many blocks
  - Writes are written in an atomic and synchronous way on a **journal**
  - It's a good idea to put this journal somewhere on the net
- **Secondary NameNode**
  - Receives copies of the edit log from the NameNode
  - When the primary is down, the system uses it and stays read-only
  - If the journal is on the network, we can switch the secondary to primary
- **DataNode:** store data, heartbeat to the NameNode with the list of their blocks

# HDFS Architecture

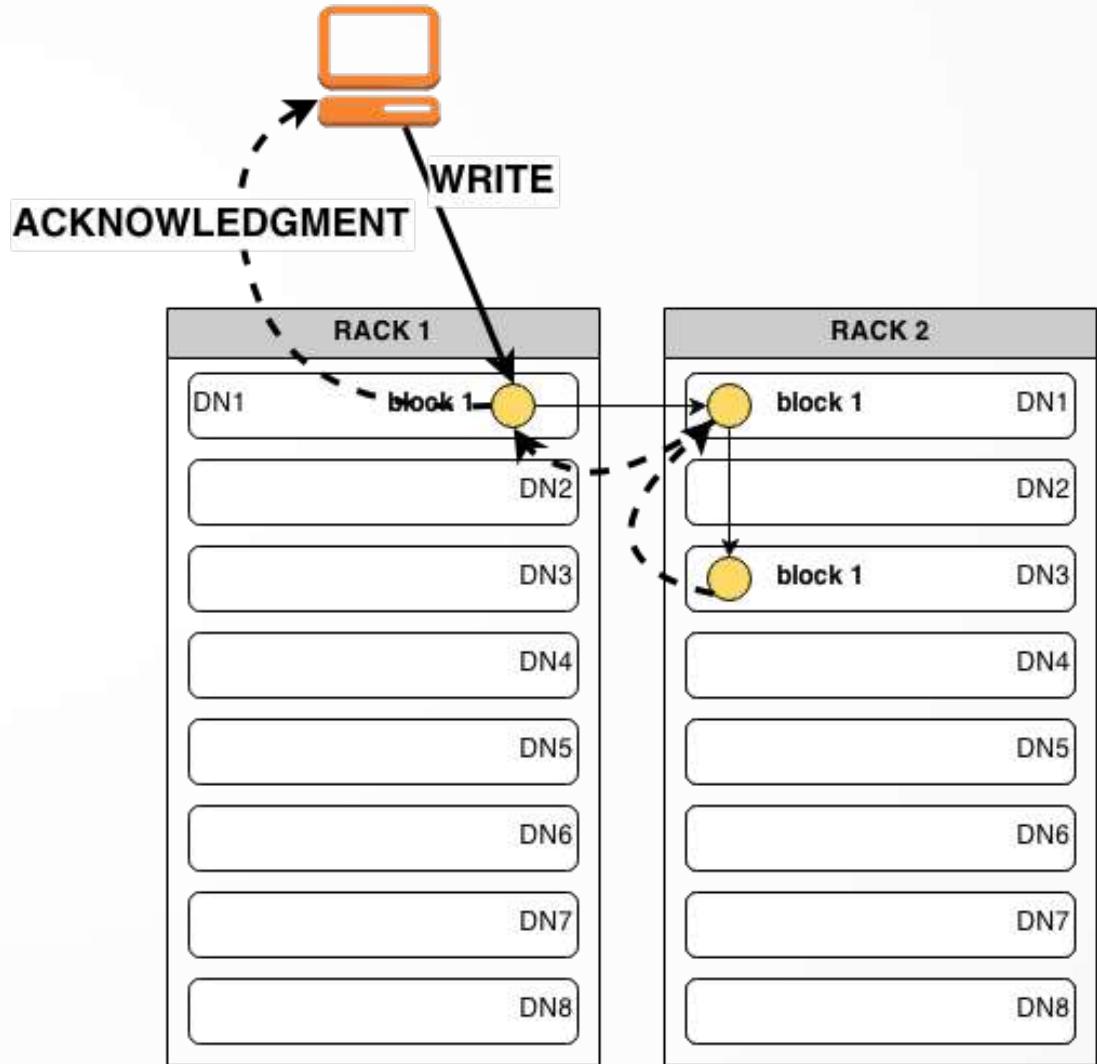


# History of a File Read

- Note: the client is often a machine **in the same cluster**
- Get the block locations from the NameNode
- Obtain a set of DataNodes, sorted by proximity to the client
  - i.e.: first the same node, then the same rack
- If MapReduce is reading, the data will be in the very same machine
  - Data is read in Map tasks
  - There are corner-case exceptions

# History of a File Write

- Client asks the NameNode for  $k$  Datanodes (default  $k=3$ )
- **Pipeline** replication: the first datanode will make a copy to the second, and that one to the third
- Default: first replica off-rack, second replica in the same rack of the first
  - Tradeoff between reliability and cluster bandwidth



# Scheduling

# A Job Is Made of Tasks

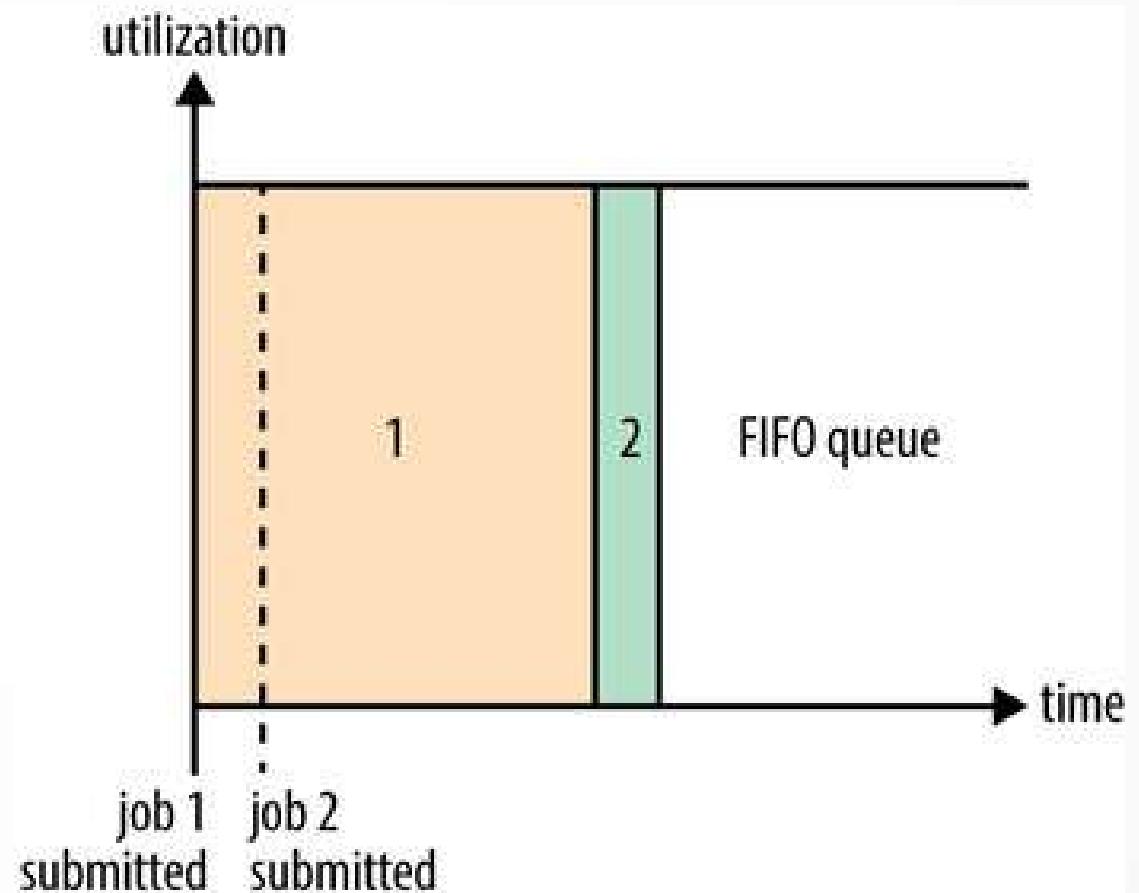
- A **MapReduce** job
  - Runs on a set of blocks specified by the programmer
  - Has one phase each of map, shuffle & reduce
- Map and reduce phases are divided in **tasks**
  - Tasks are single-machine, independent job
  - The only “holistic” part is in the shuffle phase
- Each machine runs a configurable number of tasks
  - Often: one task per CPU, so they don’t slow each other down

# Map and Reduce Tasks

- **Map:** by default, **1 HDFS block → 1 *input split* → 1 task**
  - The scheduler will do whatever is possible to run the tasks on a machine having that block
  - Map tasks are usually quick (a few seconds), unless they perform unusually large computation
- **Reduce:** Number of reduce tasks is user-specified
  - Keys are partitioned **randomly** based on hash values: one task will handle several keys
  - Users can override this and write a **custom partitioner** (useful to handle skewed data)
  - Reduce tasks have very variable runtime, depending on what they do

# Schedulers: FIFO

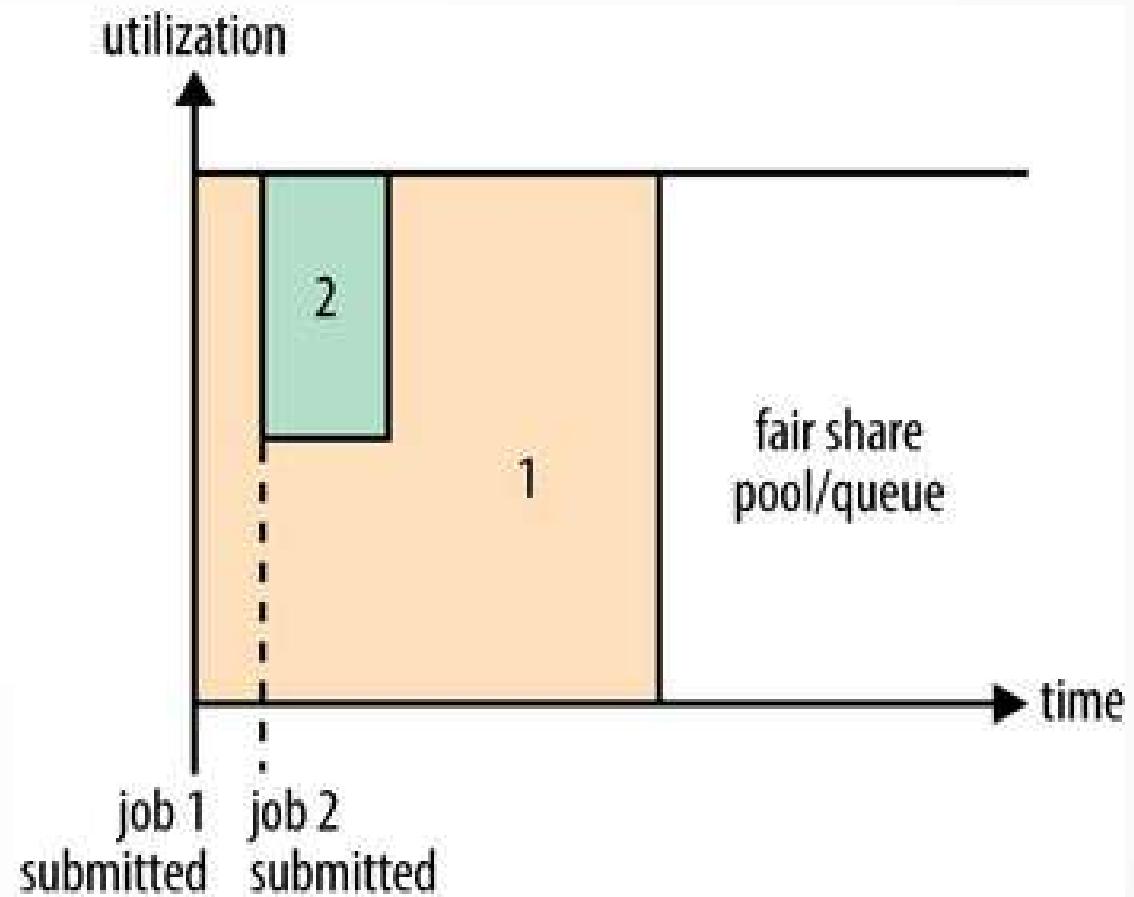
- Priority of jobs is their arrival time
- As soon as a machine is free, it's given the first pending task by the first job
- Penalizes small jobs which can **wait forever** when very large jobs are there



(from *Hadoop: the Definitive Guide*)

# Schedulers: Fair

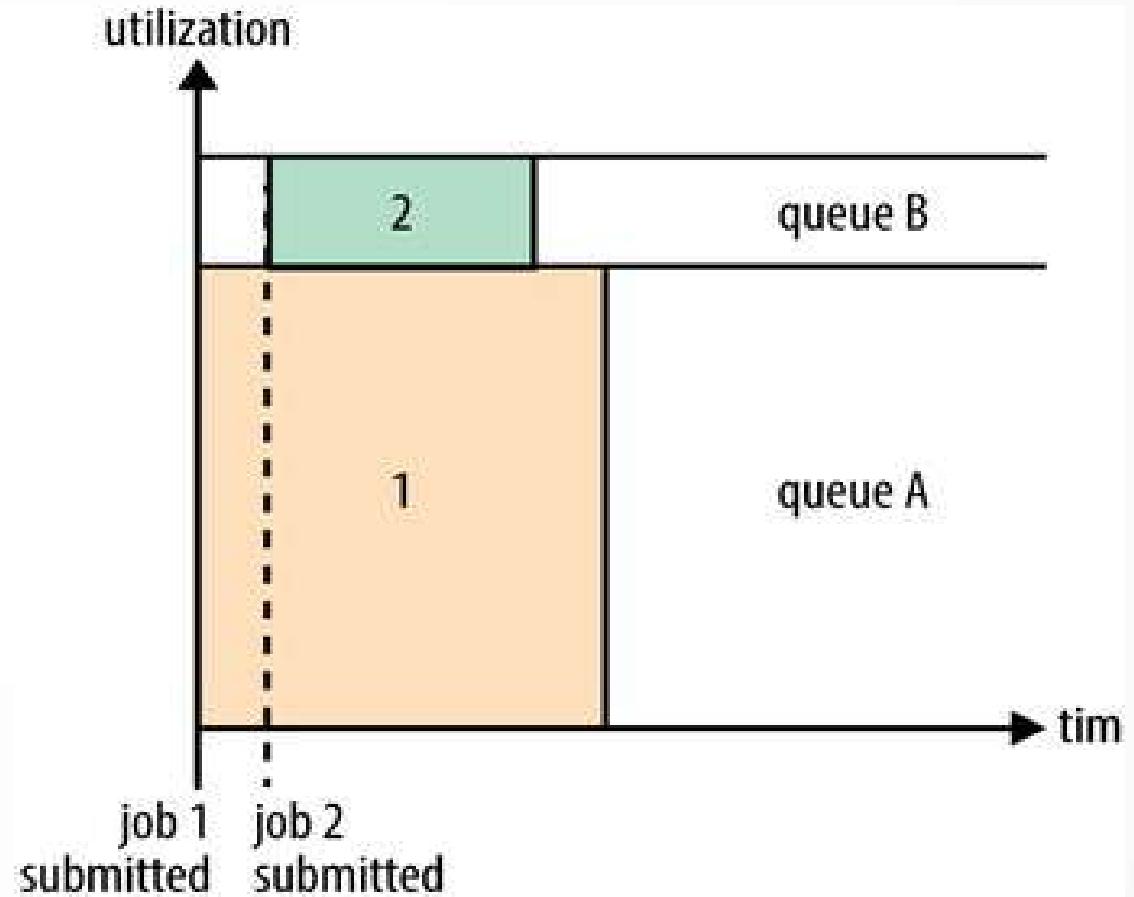
- Priority: give precedence to active jobs with **least running tasks**
- Results in each job having roughly the same amount of work done in a given moment
- Conceptually very similar to processor-sharing and/or round-robin schedulers
- Can be configured to **kill running tasks** to free up space for jobs
- You can't prioritize jobs in a queue



(from *Hadoop: the Definitive Guide*)

# Schedulers: Capacity

- Creates “virtual clusters” with a queue each and a dedicated amount of resources
- Can be used to make sure that organizations/application have access to a reasonable amount of computing power
- There is elasticity possible: if a queue leaves unused resources, they can be used by another queue



(from *Hadoop: the Definitive Guide*)

# What About Shortest Job First?

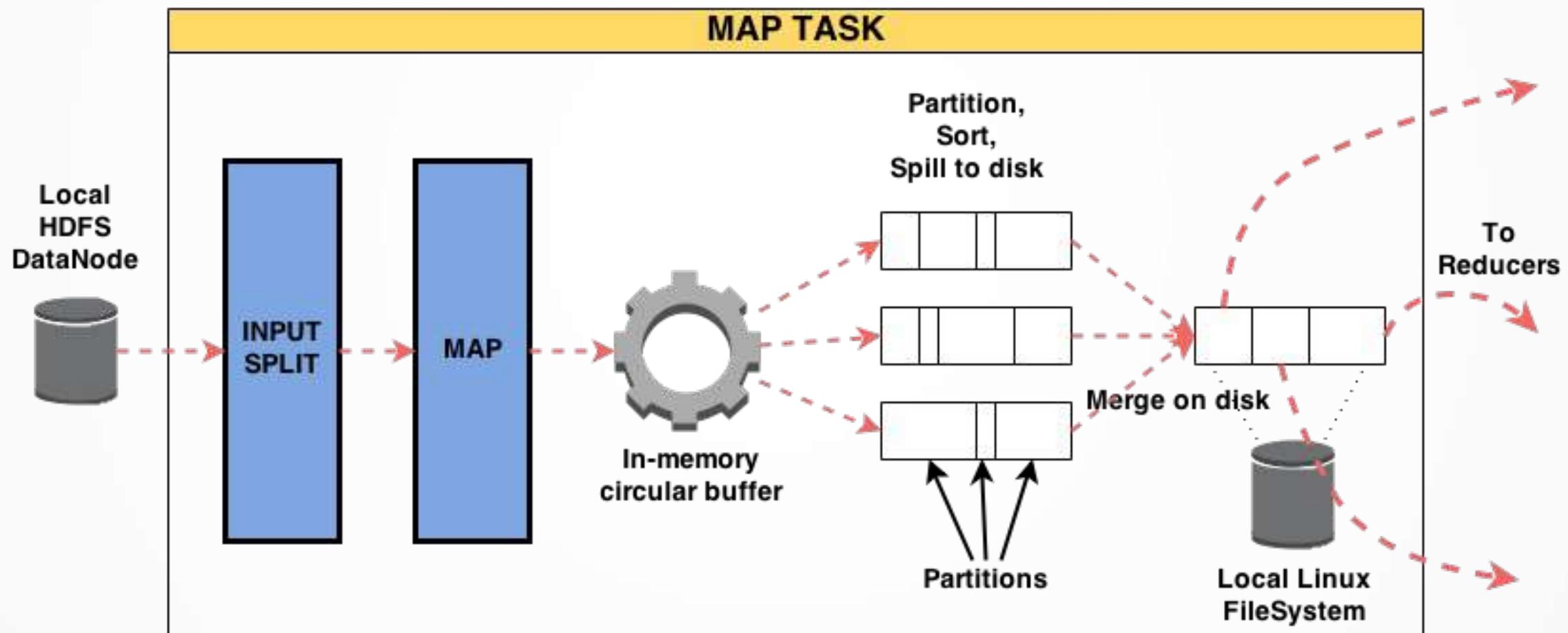
- Maybe, with your assignments, you've already seen that letting shortest jobs first ahead can be great for a loaded system
  - In particular for real-world jobs
- The problem is that you generally **don't know** how long a job will need to run
- There's a big potential here to improve performance, but system designers are conservative people & you don't know when the system will be problematic

# Handling Failures

- Failures are **common**: software & hardware problems
- If a **task fails**, it's retried a few times (e.g., 4)
  - After that, by default the job is marked as failed
- If a **task hangs** (no progress), it's killed and retried
- If a **worker machine** fails, the scheduler notices the lack of heartbeats and removes it from the worker pool
- If the **scheduler** fails—Zookeeper can be used to set up a backup and keep it updated

# Shuffle (& Sort) Phase

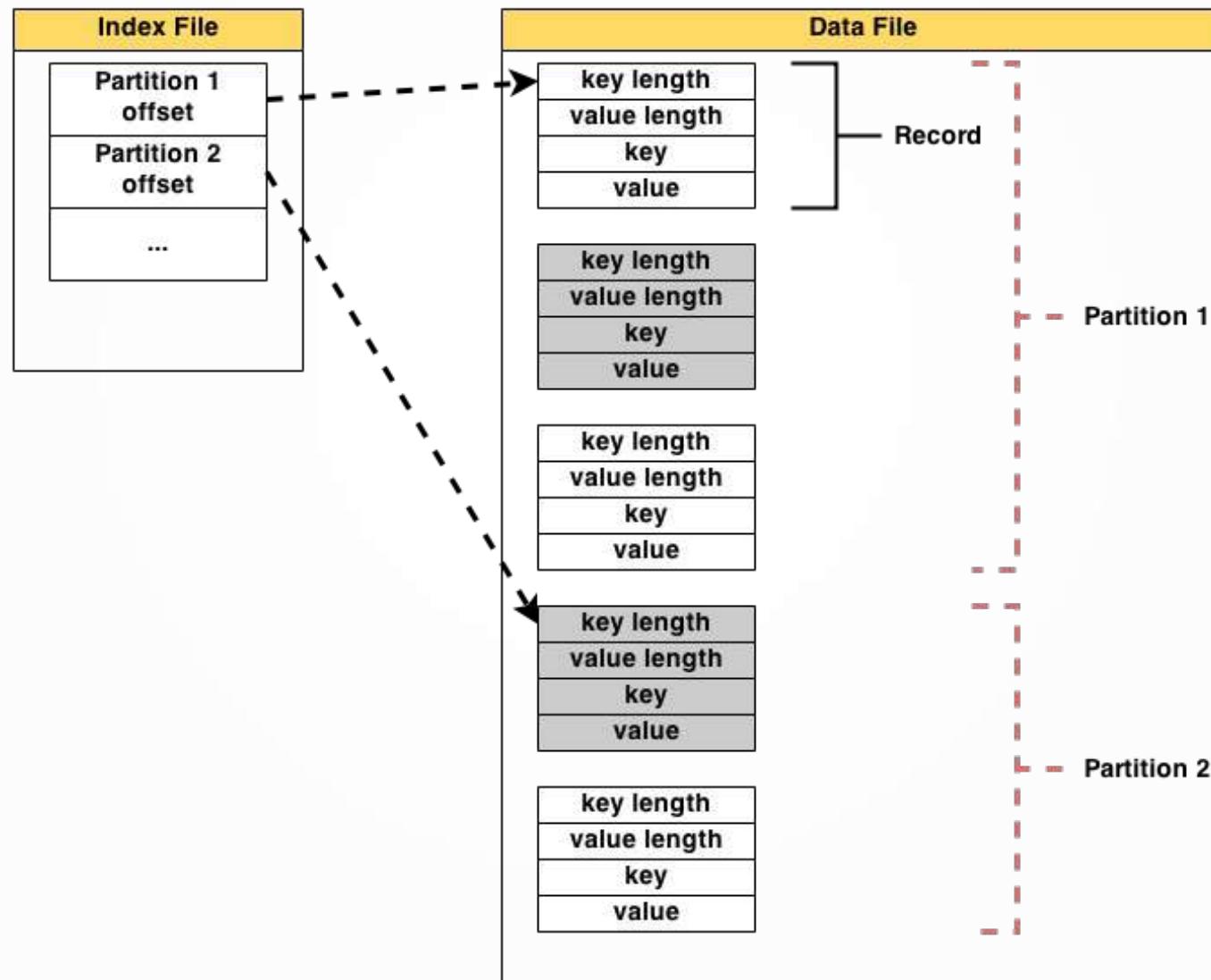
# Shuffle: Map Side



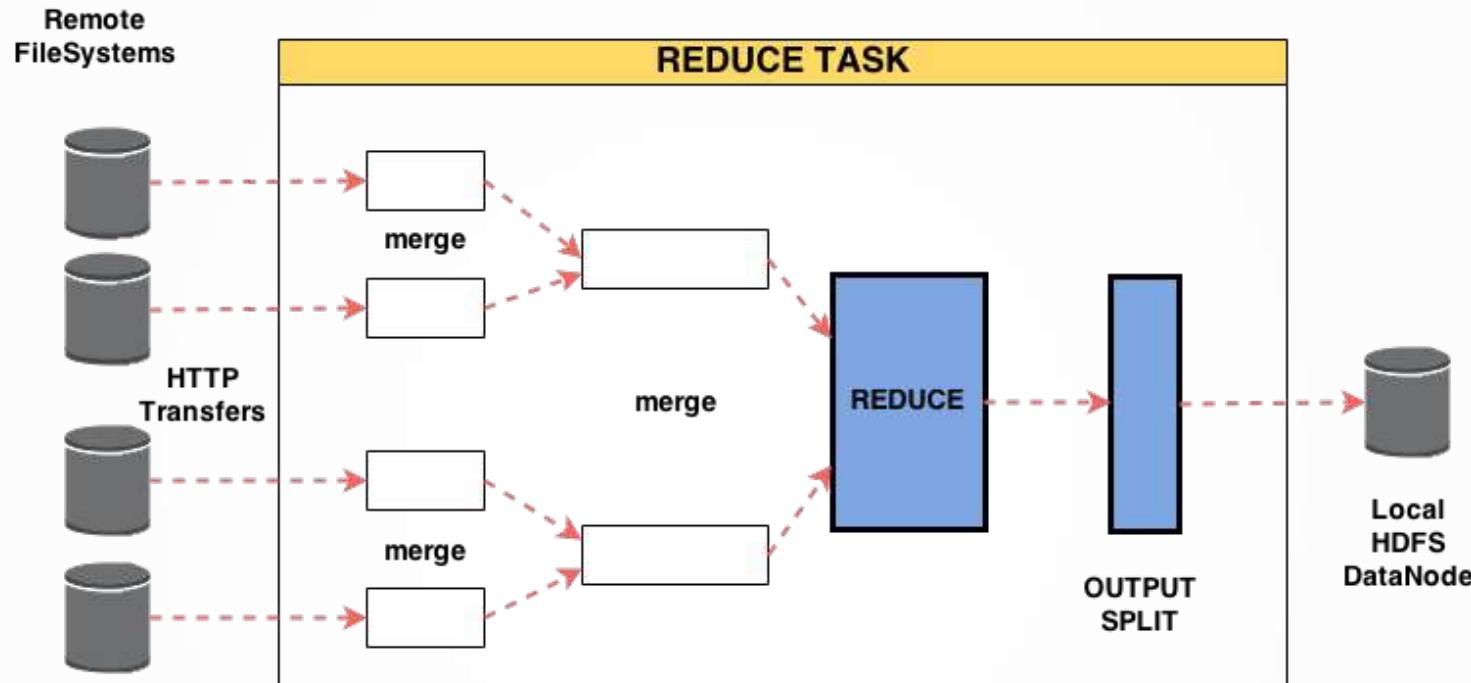
# Map Side: Description

- The output of map stays in a buffer in memory
  - Default buffer size: 100 MB
- When the buffer is filled, it's partitioned (by destination reducer), sorted and saved to disk
  - Additional guarantee in Hadoop: reduce keys are always sorted
- At the end of a map phase, spills are merged and sent to reducers
- Combiners are run right before spilling to disk. **Why?**

# A Look at A Spill File



# Shuffle: Reduce Side



- Reducers **fetch** data from mappers and run a merge
  - Mappers don't delete data right after it's sent to reducers. **Why?**
- We're essentially running a distributed mergesort
- Output is saved (& replicated) on HDFS

# Distributed Computing

A-15. Apache Spark

# References

---

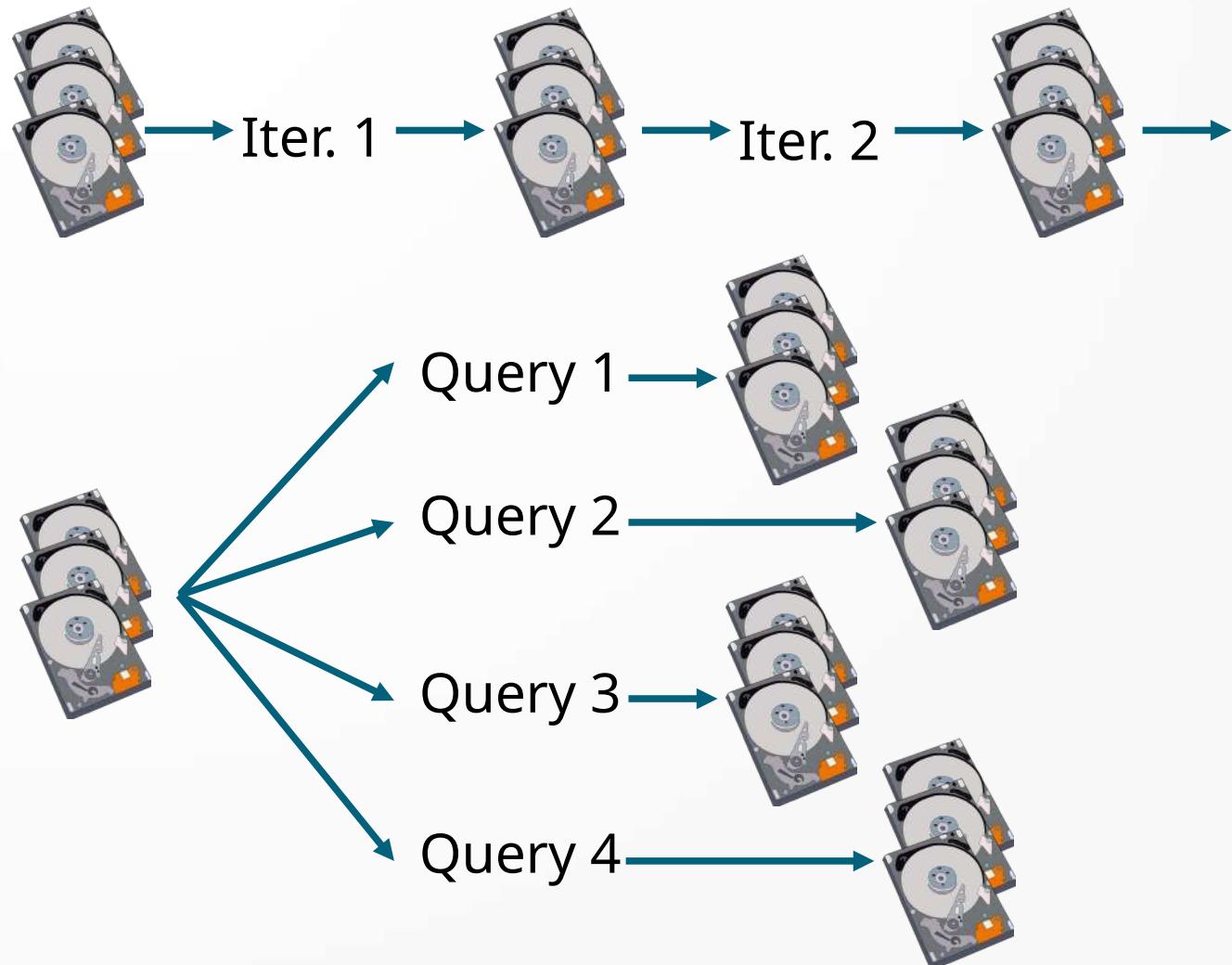
- The paper and presentation by Zaharia et al. At USENIX NSDI 2012: “*Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*”
- The spark website: <https://spark.apache.org>

# MapReduce: Virtues & Shortcomings

- MapReduce has been a big improvement for “big data” on large clusters of unreliable machines
- However, it’s less than perfect for important use cases
  - Multi-stage applications (e.g., iterative machine learning, graph processing)
  - Interactive ad-hoc queries
- Several specialized frameworks were designed to handle these cases

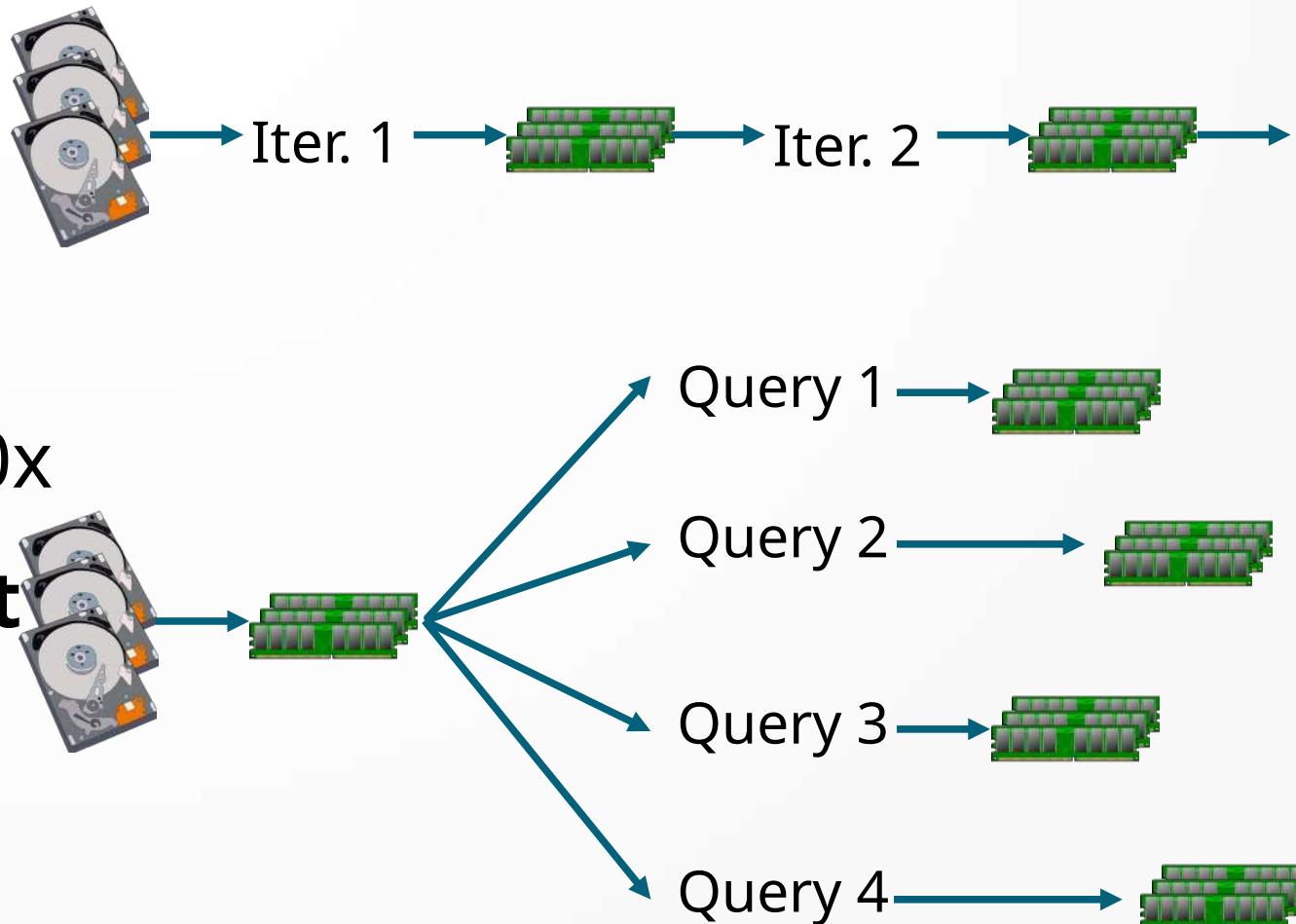
# Everything On Disk

- Each MapReduce job (i.e., something that has “one shuffle phase”) has to **read and write from disk**
- This is the solution to deal with unreliability: write everything on disk and replicate it
- However, this is **slow**



# In-Memory Processing

- What if things could be kept in RAM without the need of touching the disk every time?
- Huge speedup—10 to 100x
- The goal of Spark is to **get both this and fault tolerance**



# **Resilient Distributed Datasets (RDDs)**

# Resilient Distributed Datasets

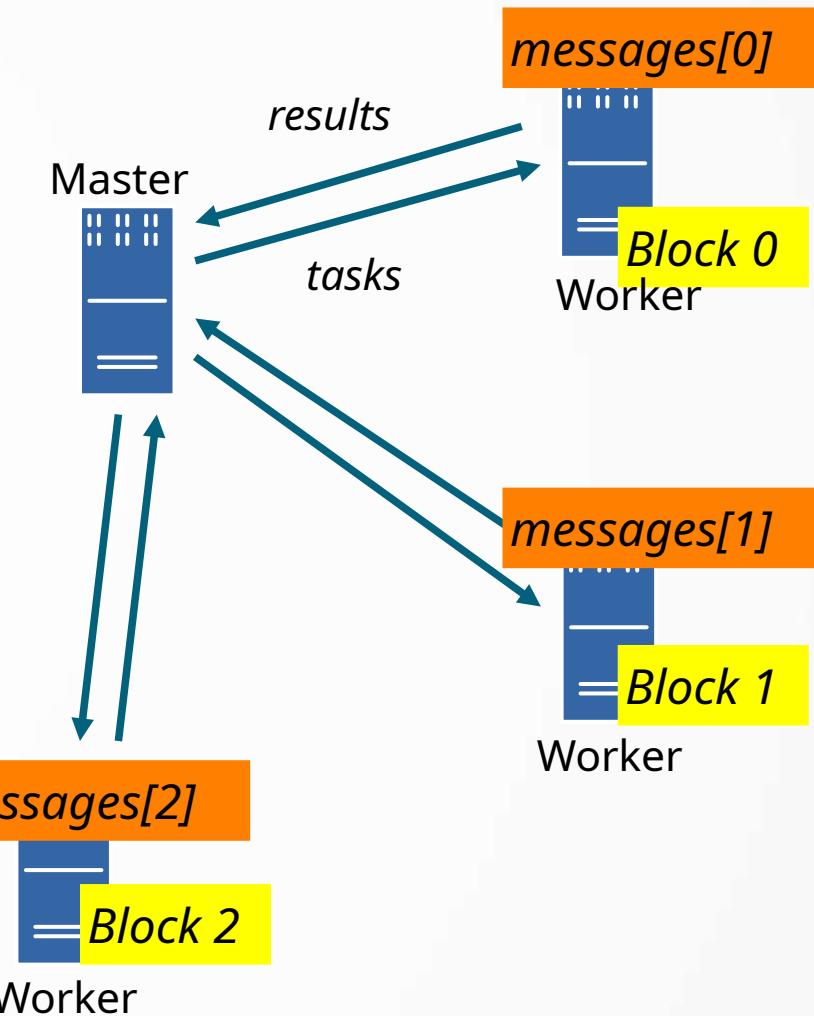
- **Immutable, distributed** in-memory data structures
  - **Partitioned** among the machines in the cluster
- Only built through **coarse-grained** operations that process the whole RDD
  - Based on functional programming (map/filter/join...)
- Fault recovery performed using **lineage**
  - A log of all the operations done to get there
  - Recover lost partitions by **recomputing what's missing**
  - No cost if nothing fails

# RDDs vs. Databases

- Databases & key-value stores handle **small updates** and **store everything on disk**
  - They're good for small modifications (transactions) that don't modify most of the state
- RDDs are efficient for **large operations**

# Example: Log Mining

```
lines = spark.textFile("hdfs://...")  
  
def is_error(line):  
    return line.startswith('ERROR')  
errors = lines.filter(is_error)  
  
def get_message(line):  
    return line.strip().split()[1]  
messages = errors.map(get_message)  
messages.persist()  
  
messages.filter(lambda m: 'foo' in m).count()  
messages.filter(lambda m: 'bar' in m).count()
```



# Fault Recovery

- RDDs tracks **lineage** (i.e. dependencies) for each block

```
messages=textFile(...) \  
    .filter(lambda x: 'error' in x) \  
    .map(lambda x: x.split()[1])
```

