

# Rafting Trip

(AKA: Distributed Systems)

David Beazley  
(@dabeaz)

<https://www.dabeaz.com>

# Saw Recently...



**David Crawshaw**  
@davidcrawshaw

Follow



The longer you spend building and running distributed systems, the more effort you put into finding ways to avoid distributing systems.

**Martin Thompson** @mjpt777

After years of working on distributed systems I still keep being surprised by how easy it is miss potential outcomes. The state space is too vast for the human brain.

10:13 AM - 28 May 2019

126 Retweets 483 Likes



15

126

483



# This Week

- We attempt to implement a project from MIT's distributed systems class (6.824)

**6.824 - Spring 2020**

## **6.824 Lab 2: Raft**

**Part 2A Due: Feb 21 23:59**

**Part 2B Due: Feb 28 23:59**

**Part 2C Due: Mar 6 23:59**

---

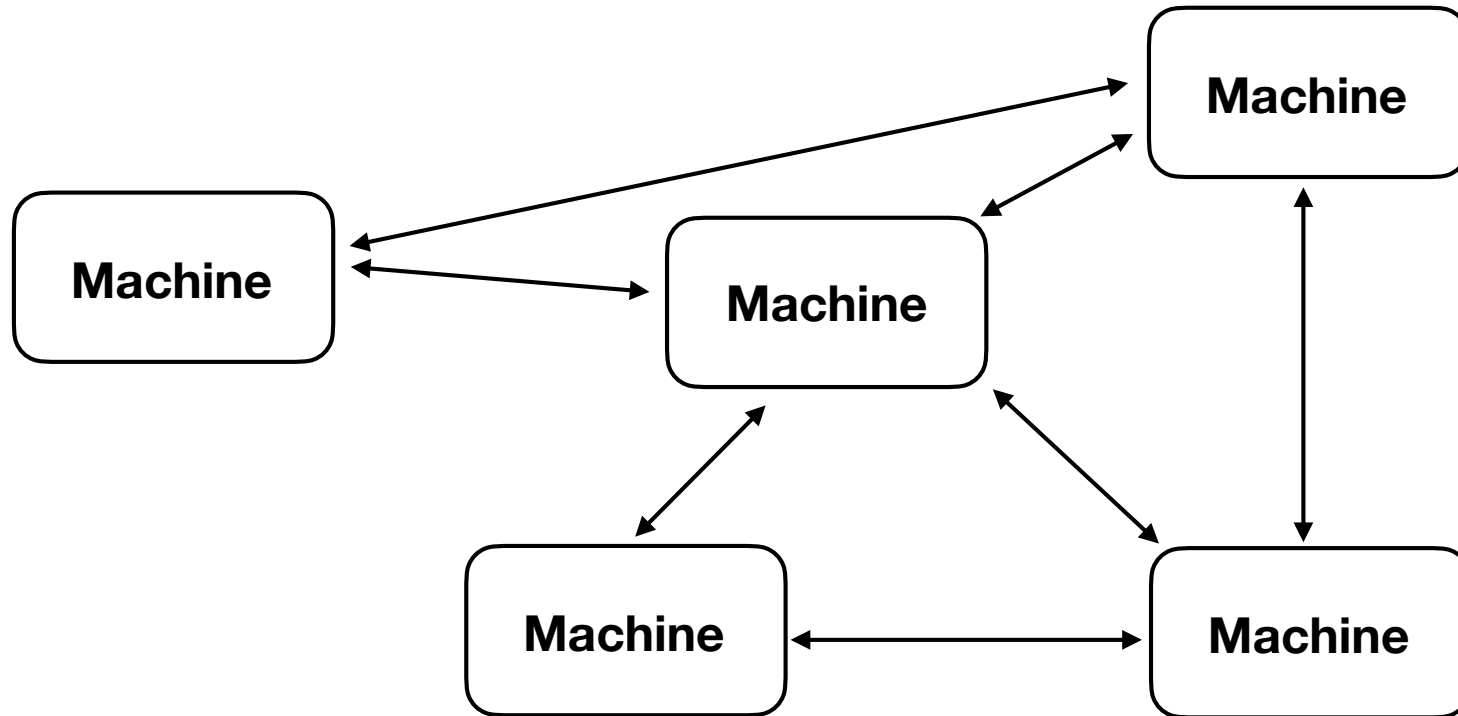
### **Introduction**

This is the first in a series of labs in which you'll build a fault-tolerant key/value storage system. In this lab you'll implement Raft, a replicated state machine protocol. In the next lab you'll build a key/value service on top of Raft. Then you will “shard” your service over multiple replicated state machines for higher performance.

A replicated service achieves fault tolerance by storing complete copies of its state (i.e., data) on multiple replica servers. Replication allows the service to continue operating even if some of its servers experience failures (crashes or a broken or flaky network). The challenge is that failures may cause the replicas to hold differing copies of the data.

# High Level View

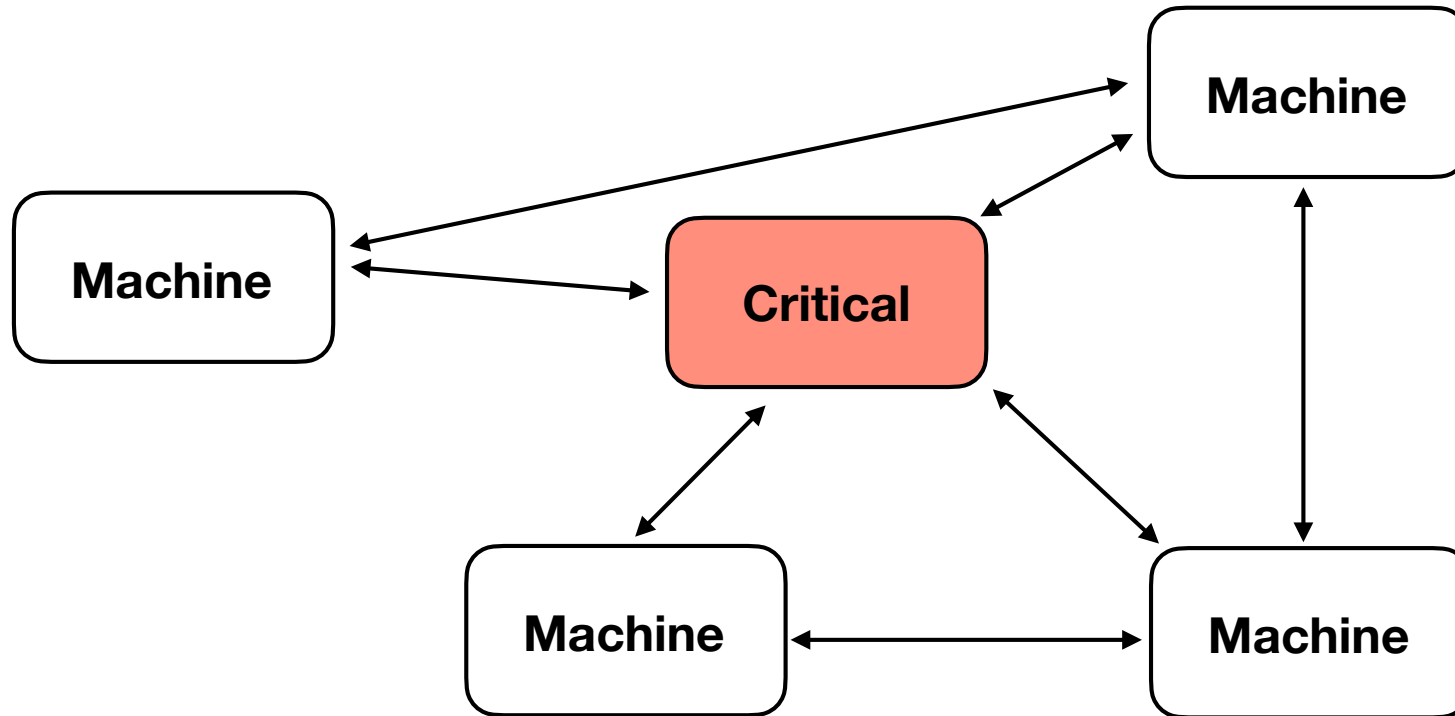
- Distributed Computing



- Machines/services communicating over a network

# Our Focus: Reliability

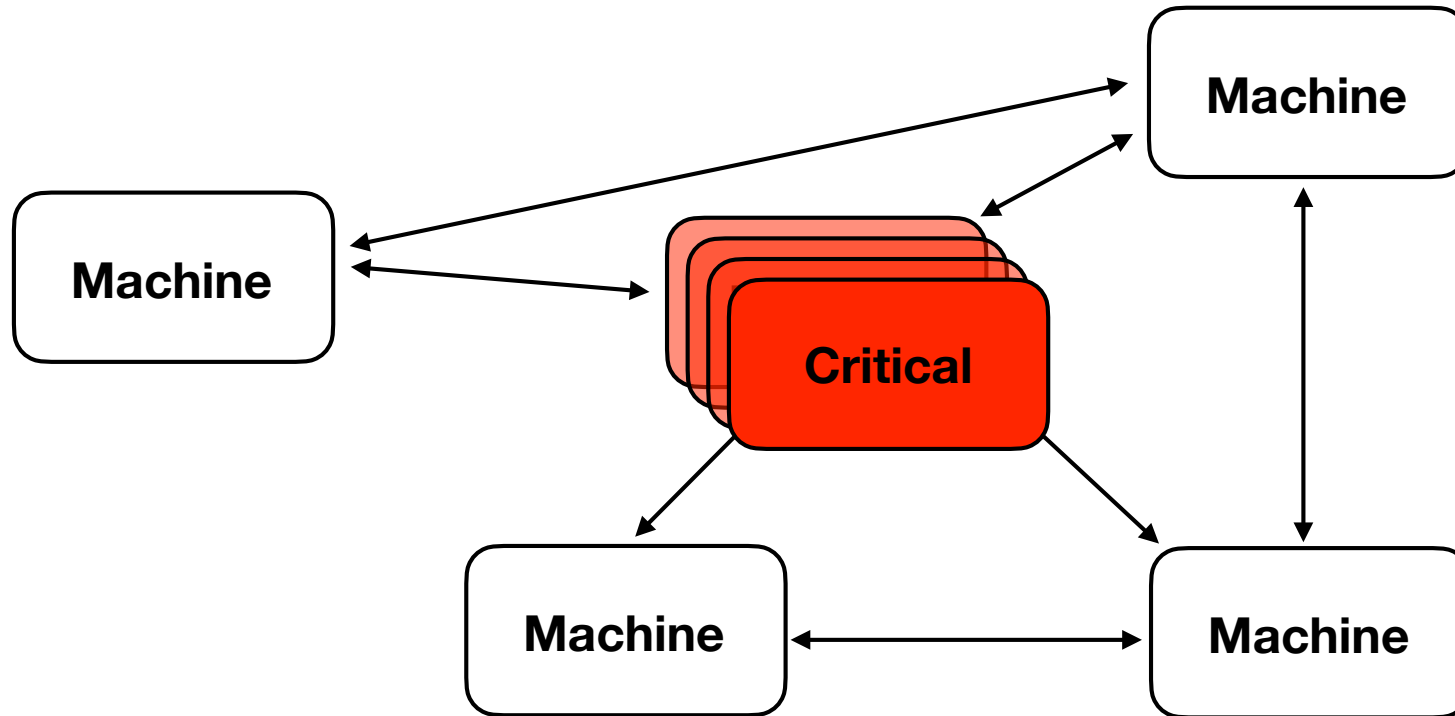
- Certain services are more critical than others



- Example: Centralized registries, database, etc.

# Our Focus: Reliability

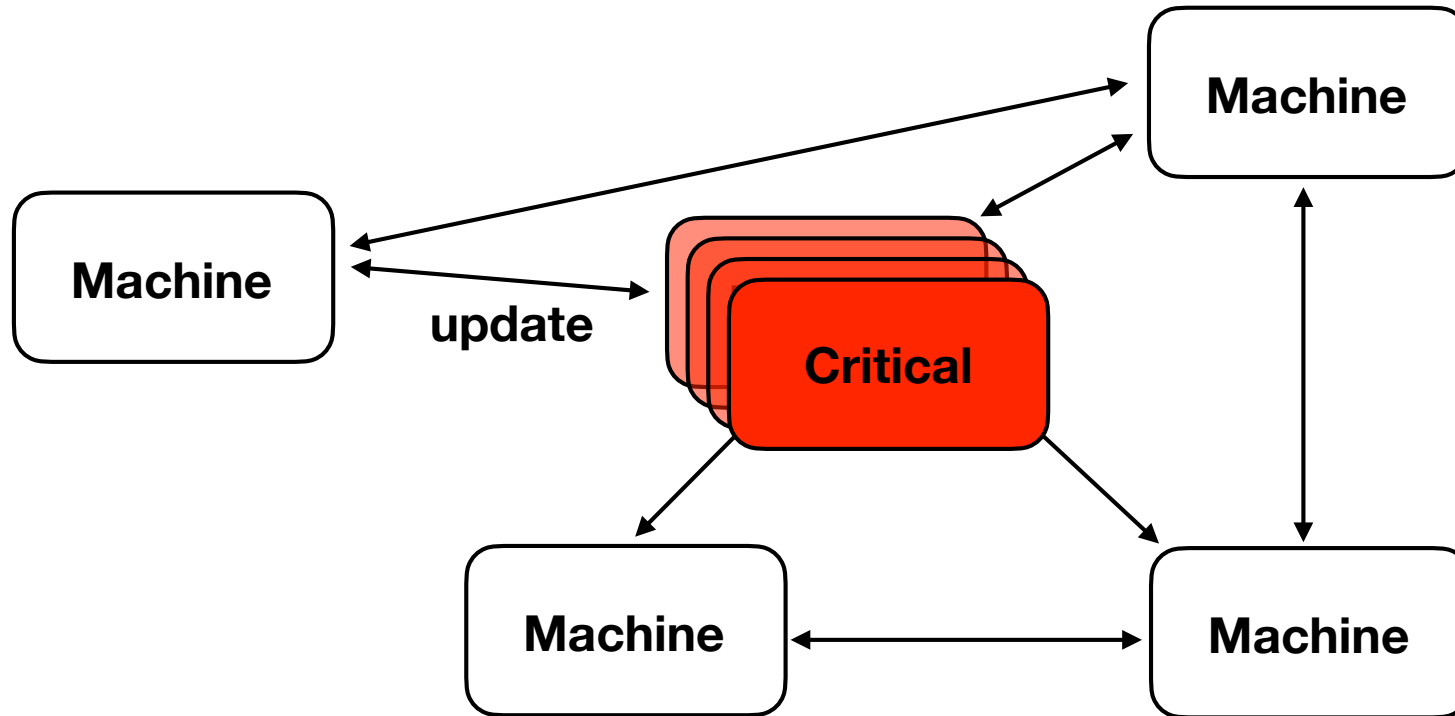
- Obvious Solution: Replication!



- Whew! Crisis averted via redundancy.

# A Problem

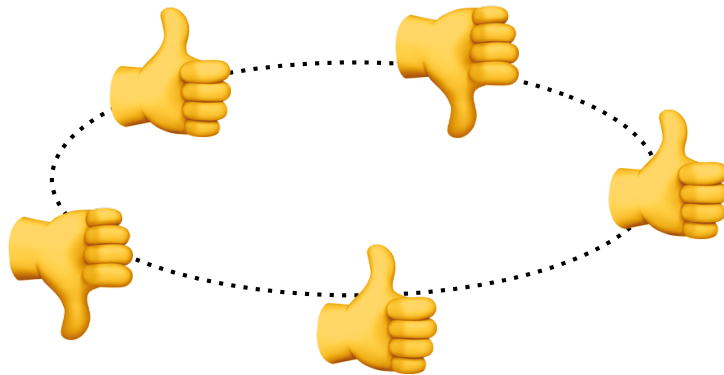
- Updates!



- How do you simultaneously update the state on all replicas at once (laws of physics, time, etc.)

# Solution: Consensus

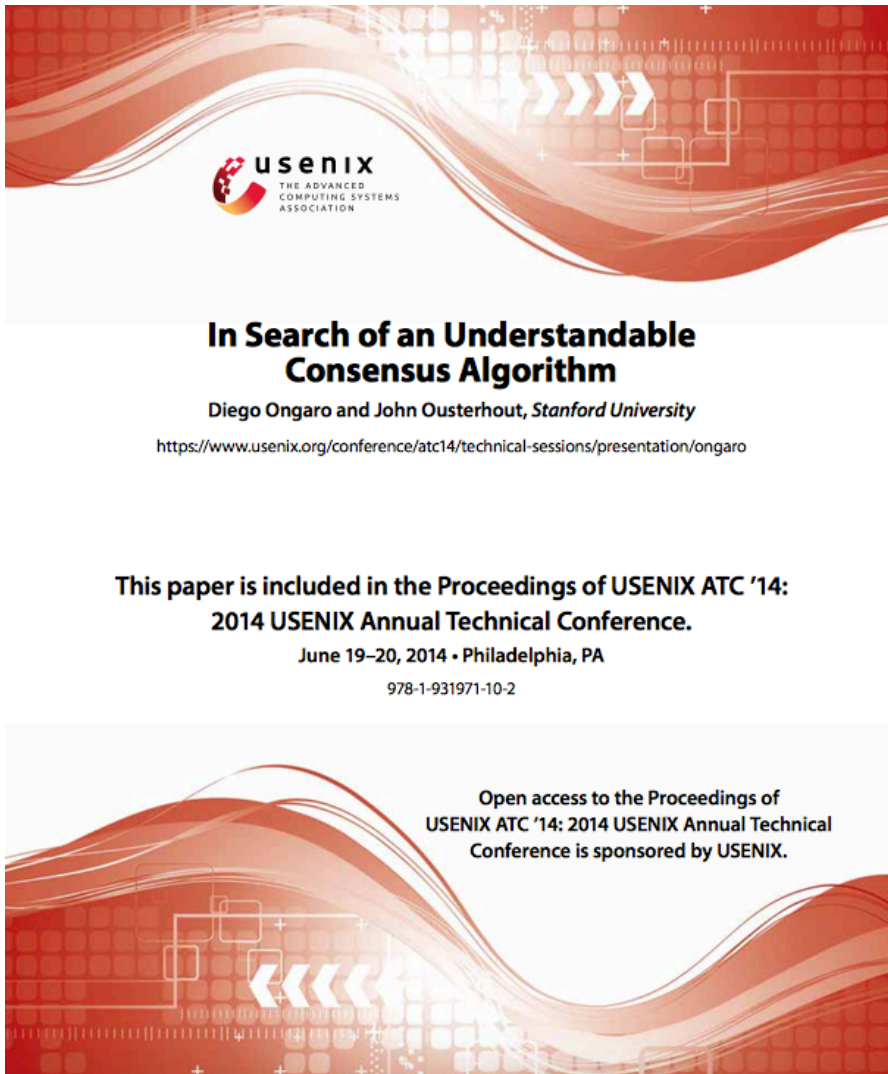
- Replication requires a mechanism for agreeing on the correct "state" of the system. Machines work together.



- This is one of the fundamental hard problems of distributed computing
- Algorithms: Distributed Consensus



# Our Challenge



- Raft Algorithm
- Distributed Consensus
- Published @ 2014 USENIX ATC
- Claim: "Understandable"

## What is etcd?

# Real World Raft Failure!

On November 2, 2020, Cloudflare had an [incident](#) that impacted the availability of the API and dashboard for six hours and 33 minutes. During this incident, the success rate

## 2020-11-02 14:44 UTC: etcd Errors begin

The rack with the misbehaving switch included one server in our etcd cluster. We use [etcd](#) heavily in our core data centers whenever we need strongly consistent data storage that's reliable across multiple nodes.

In the event that the cluster leader fails, etcd uses the [RAFT](#) protocol to maintain consistency and establish consensus to promote a new leader. In the RAFT protocol, cluster members are assumed to be either available or unavailable, and to provide accurate information or none at all. This works fine when a machine crashes, but is not always able to handle situations where different members of the cluster have conflicting information.

We are very sorry for the difficulty the outage caused, and are continuing to improve as our systems grow. We've since fixed the bug in our cluster management system, and are continuing to tune each of the systems involved in this incident to be more resilient to failures of their dependencies. If you're interested in helping solve these problems at scale, please visit [cloudflare.com/careers](https://cloudflare.com/careers).

# Why Raft?

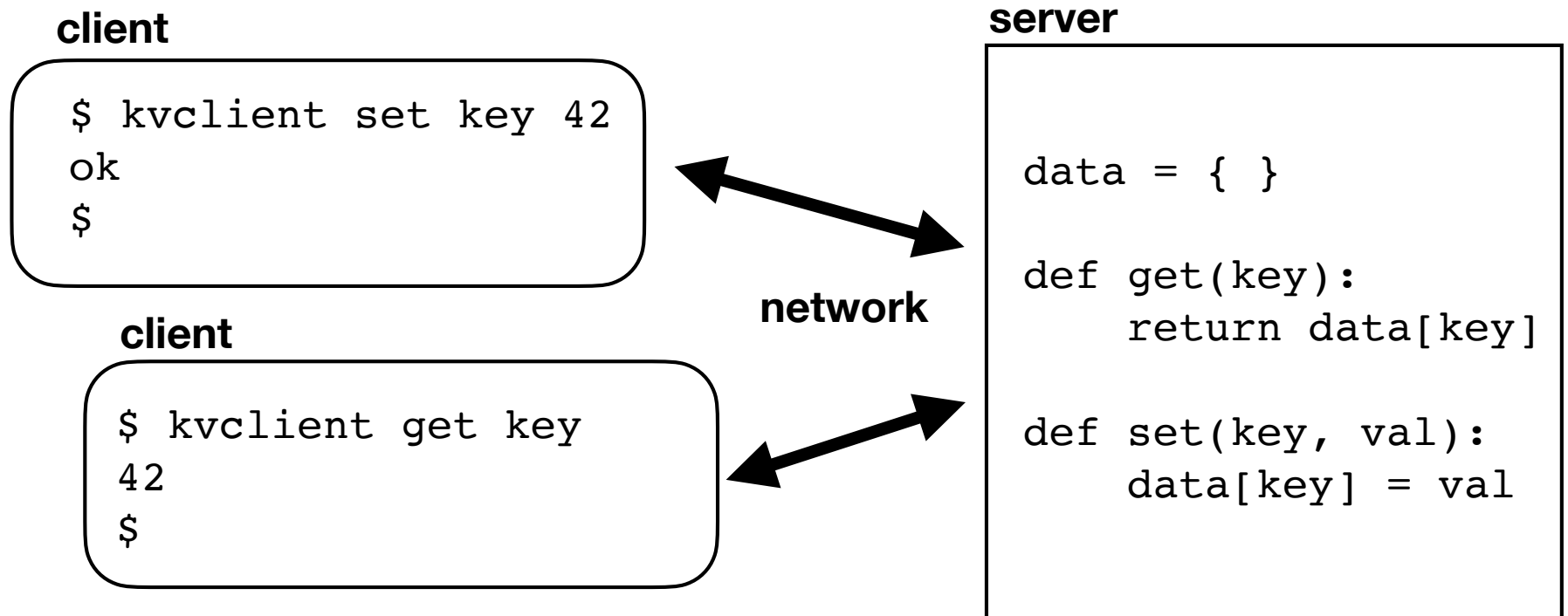
- There are many facets of distributed computing. However, distributed consensus might be one of the hardest.
- Fault tolerance: Cost of failure is high.
- Non-trivial: Many moving parts. Not an "echo server."
- Challenging: concurrency, networks, testing, etc.
- Solving it involves problems that transcend the algorithm

# Core Topics

- Messaging and networks
- Concurrency
- State machines and event driven systems
- Software architecture/OO
- Testing/validation

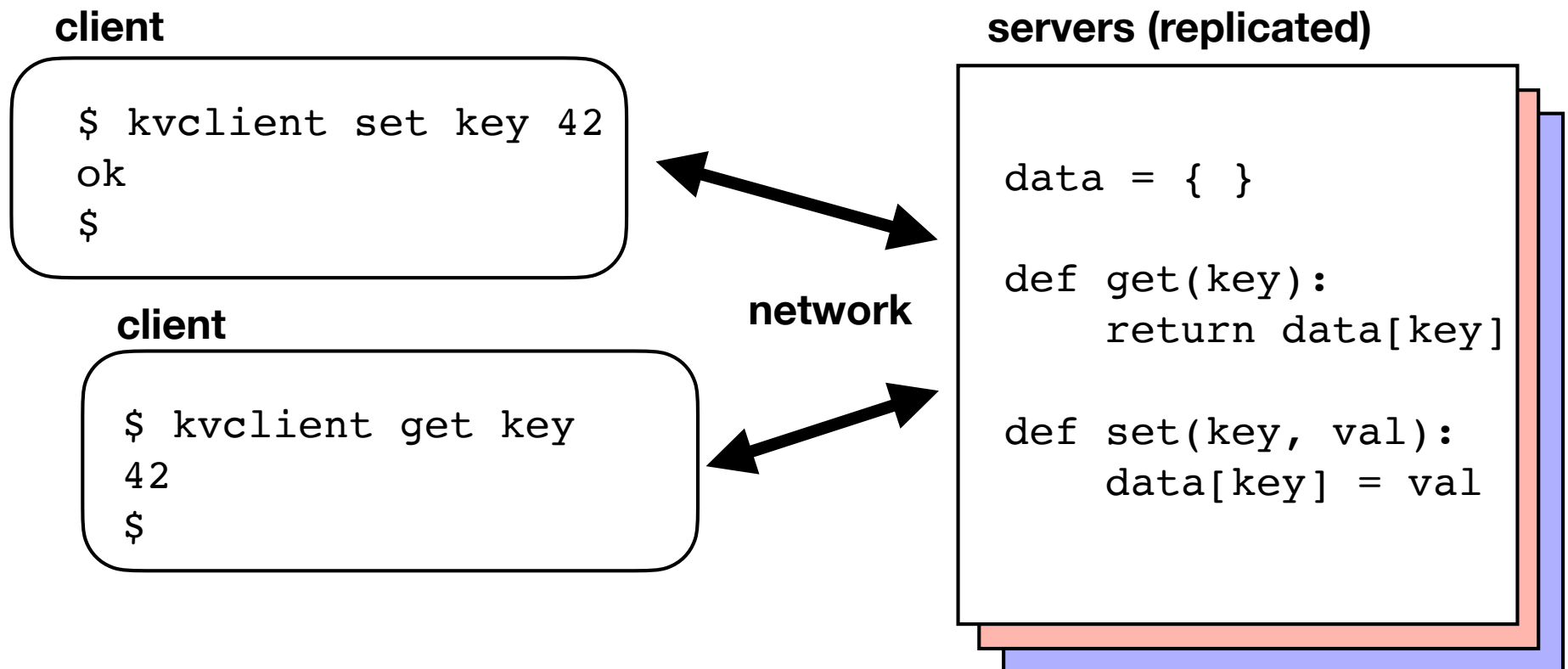
# The Project

- We're going to build a distributed key/value store
- In a nutshell: A networked dict (e.g., redis, memcache)



# The Problem

- Making it fault-tolerant
- Always available, never lose data



# How it Works

- Raft is an algorithm that solves the replication problem
- I will attempt to explain how in a few slides
- There are a few central ideas



# Transaction Logs

- Raft manages a transaction log (i.e., a "list")

**server**

```
data = { }  
  
def get(key):  
    return data[key]  
  
def set(key, val):  
    data[key] = val
```

**log**

```
...  
set x 3  
set y 1  
set y 9  
set x 2  
set x 0  
...
```

- Log keeps an ordered record of state changes
- If you replay the log, you get back to the current state.
- Not a new idea: Databases do this.

# Replication

- Fault tolerance achieved by replicating the transaction log

1 2 3 4 5 6 7 8

**(Leader) Server 0**

**Server 1**

**Server 2**

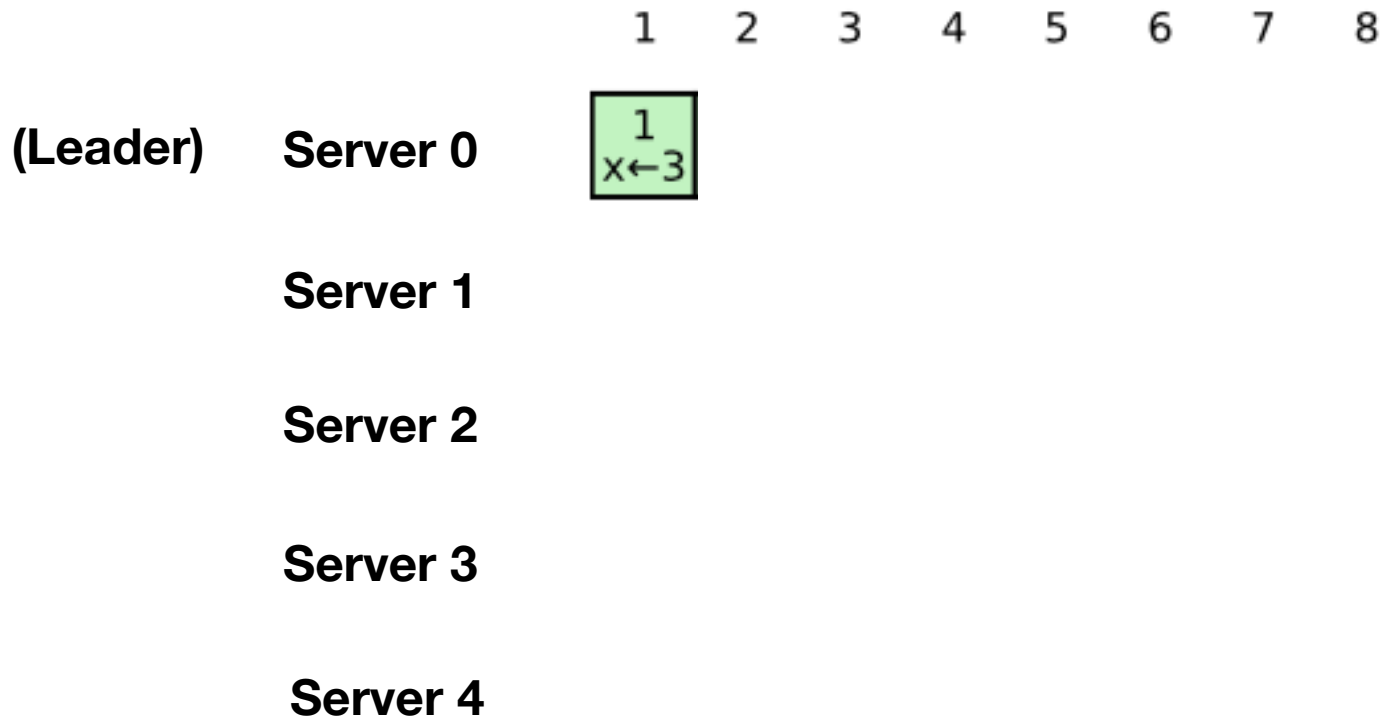
**Server 3**

**Server 4**

- Log entries are first added to the leader

# Replication

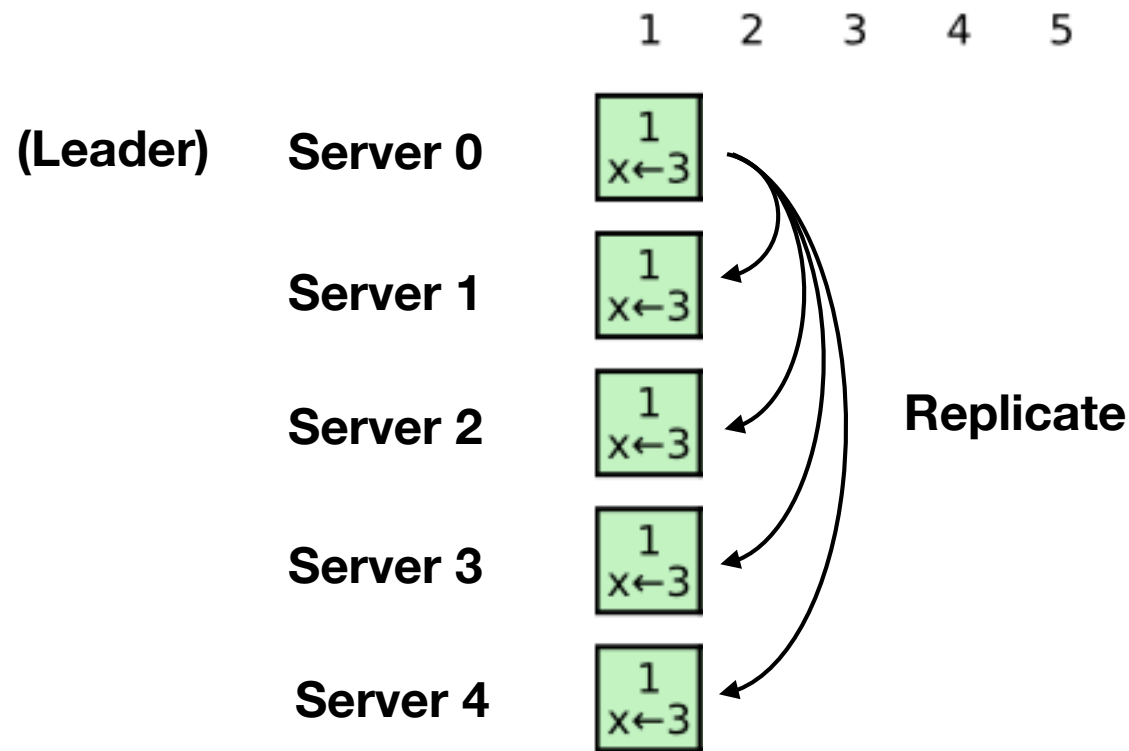
- Fault tolerance achieved by replicating the transaction log



- Log entries are first added to the leader

# Replication

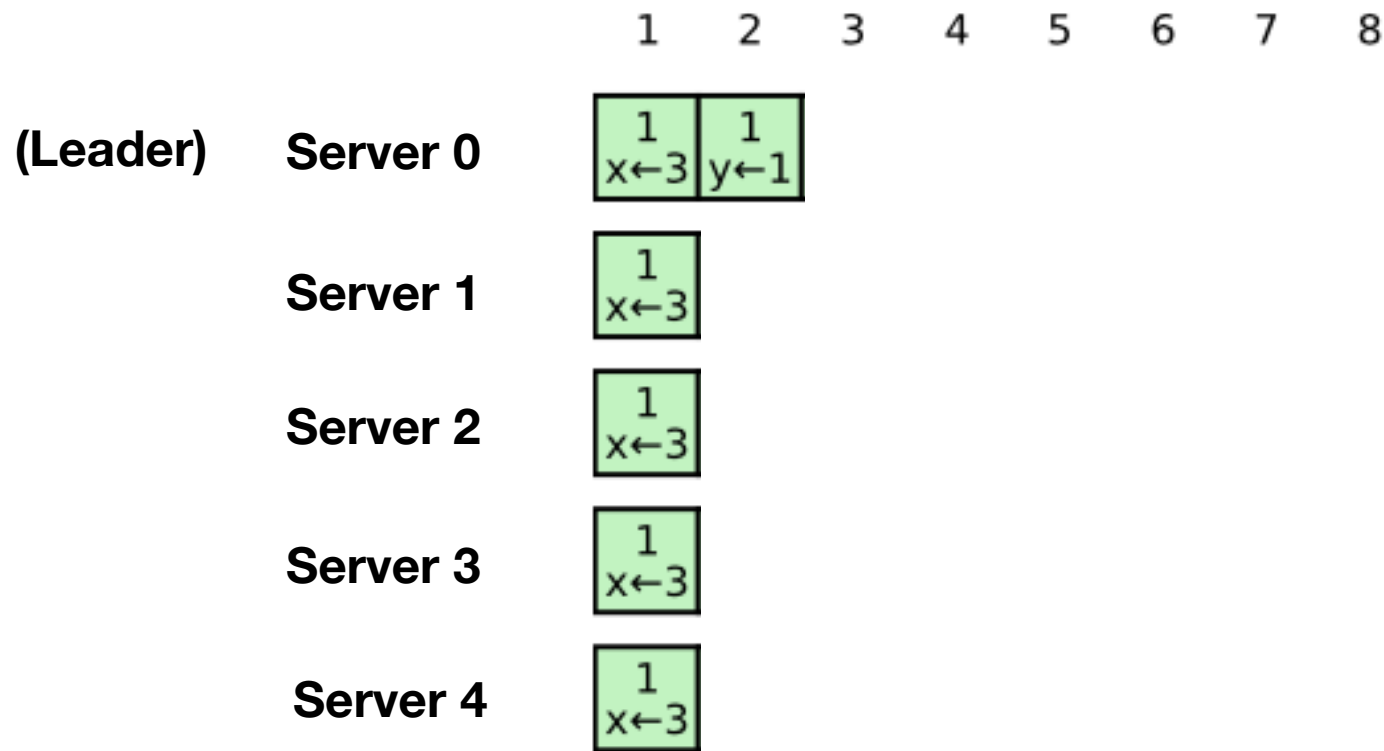
- Fault tolerance achieved by replicating the transaction log



- Leader replicates to other servers (followers)

# Replication

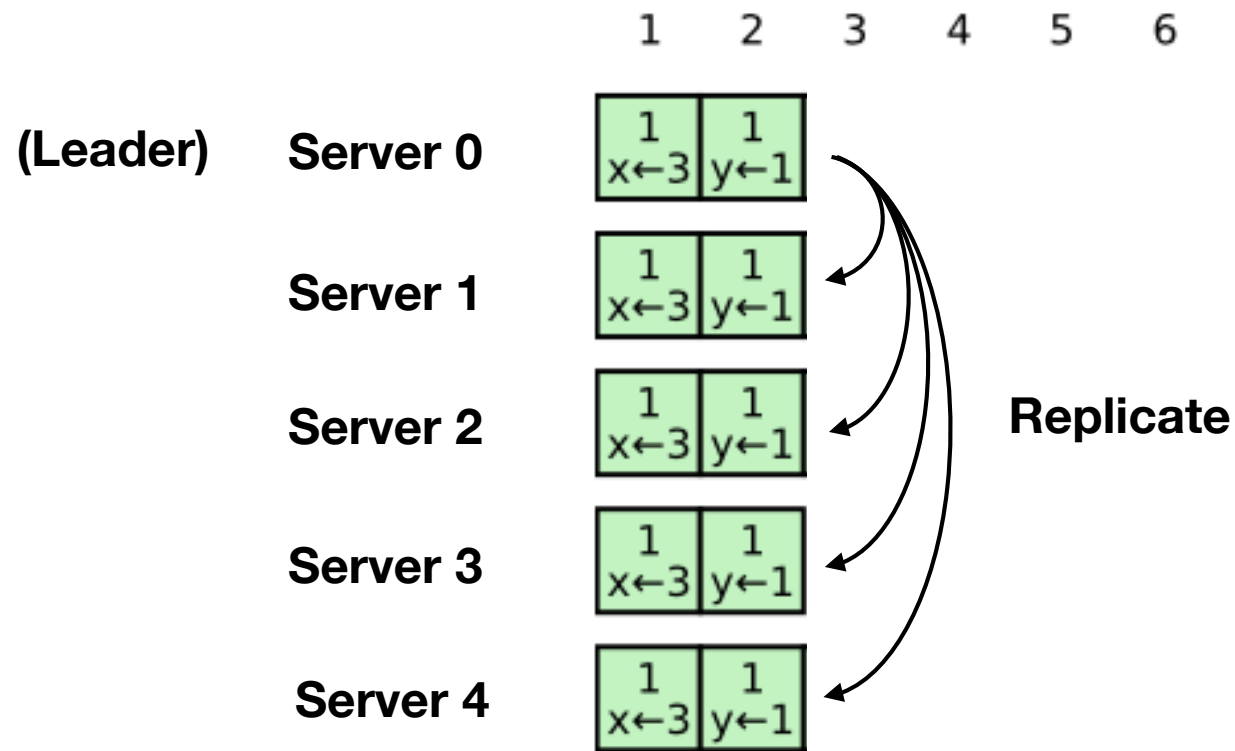
- Fault tolerance achieved by replicating the transaction log



- This continues for more log entries

# Replication

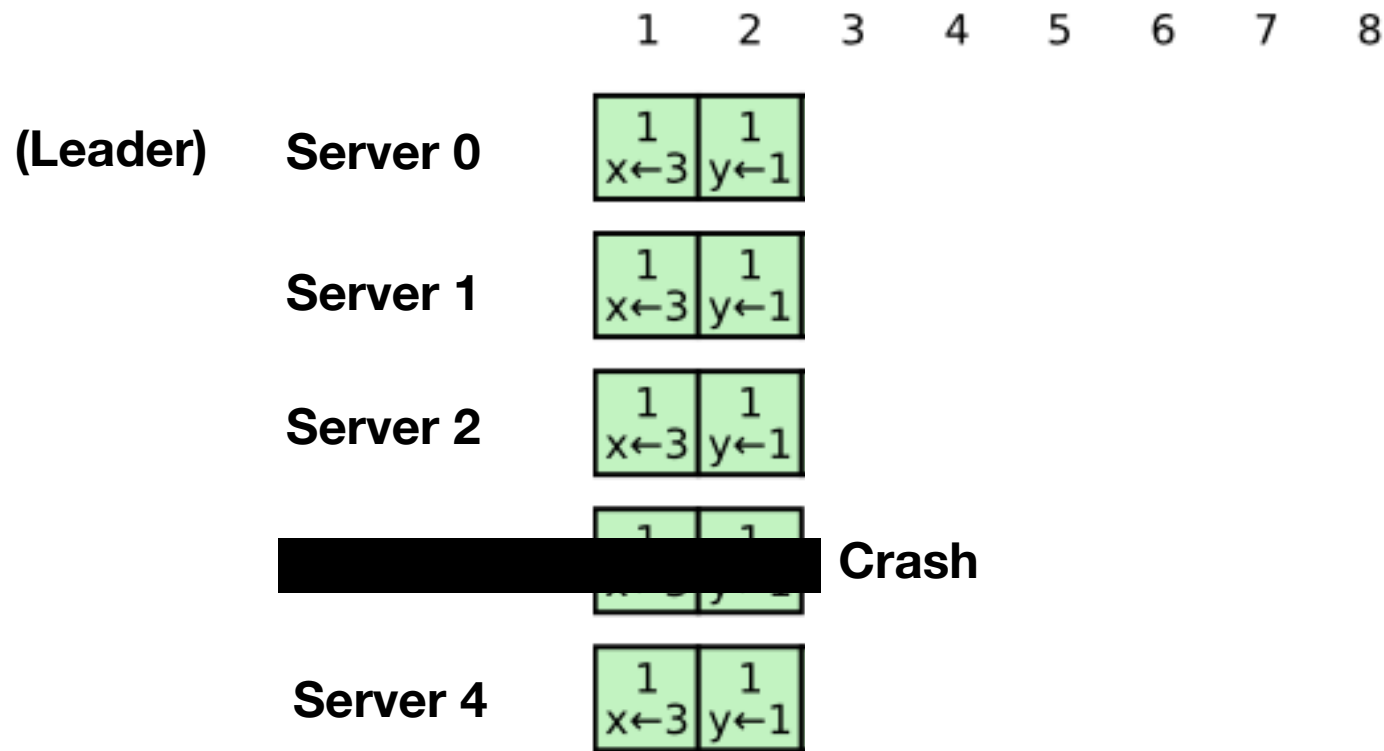
- Fault tolerance achieved by replicating the transaction log



- This continues for more log entries

# Replication

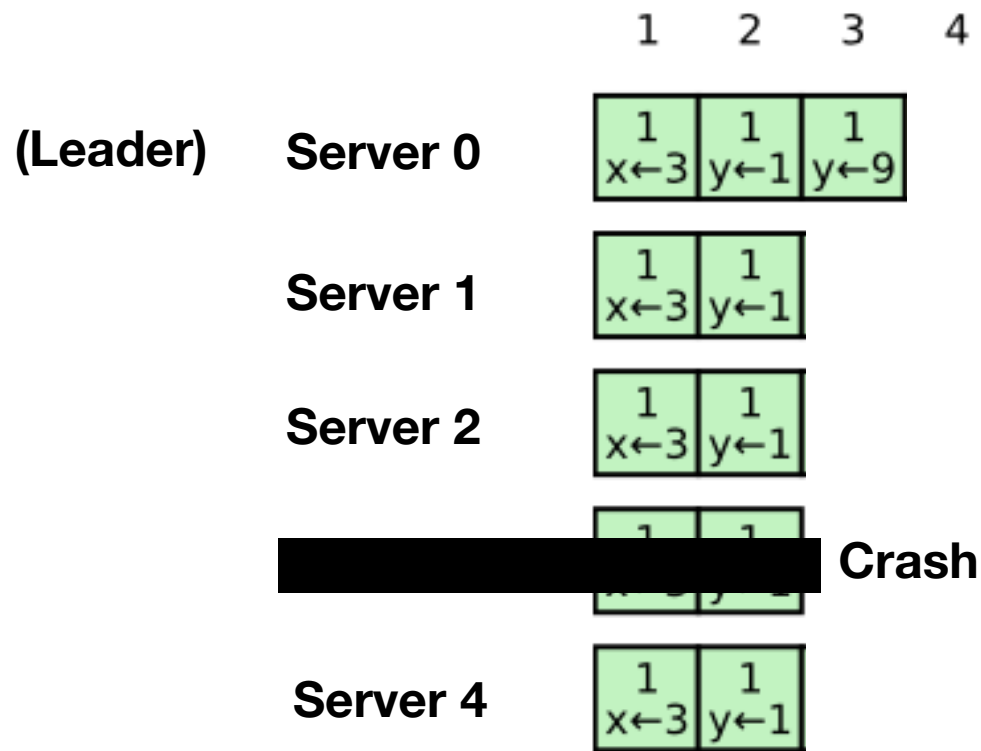
- Fault tolerance achieved by replicating the transaction log



- If a machine crashes, cluster keeps operating

# Replication

- Fault tolerance achieved by replicating the transaction log

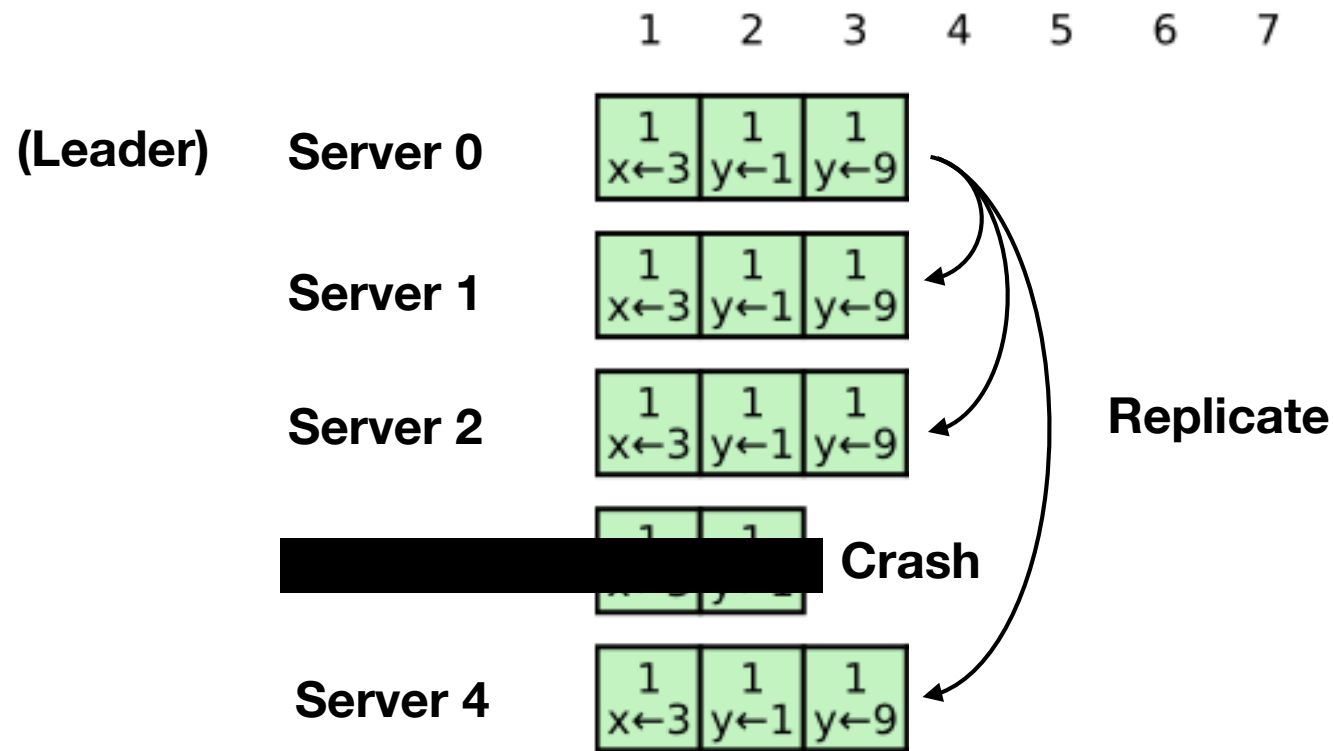


- If a machine crashes, cluster keeps operating



# Replication

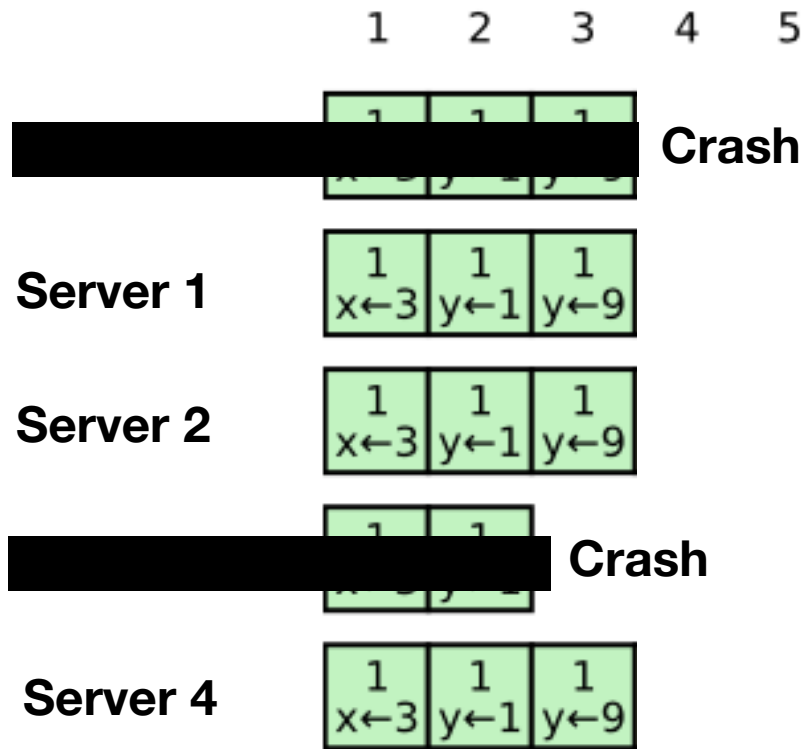
- Fault tolerance achieved by replicating the transaction log



- If a machine crashes, cluster keeps operating

# Replication

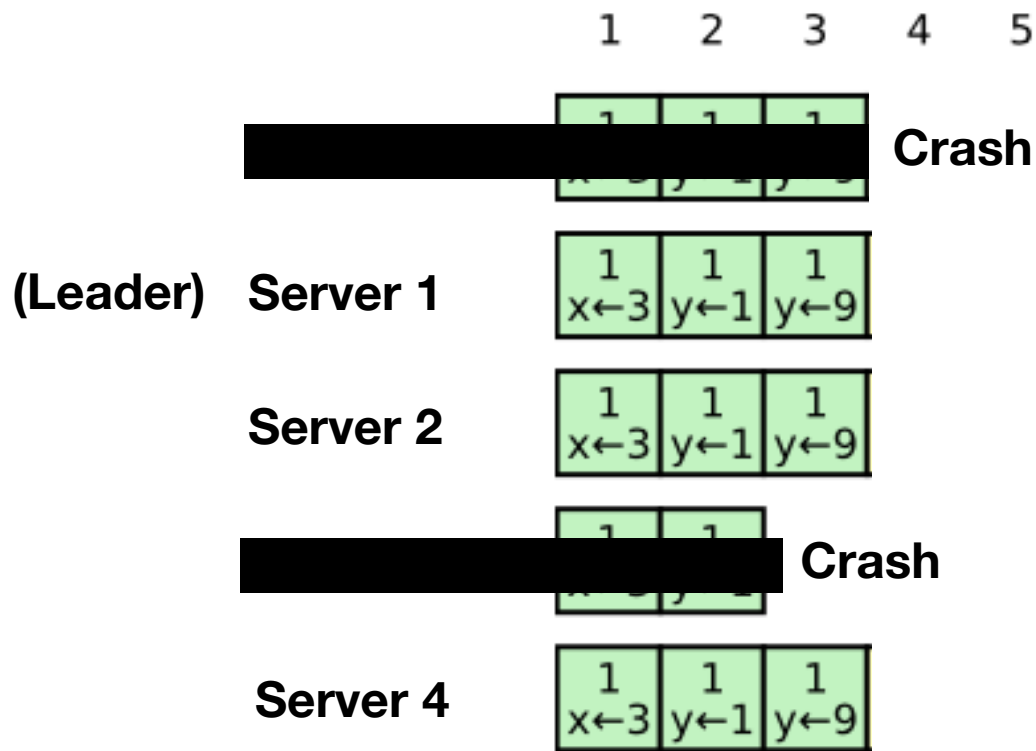
- Fault tolerance achieved by replicating the transaction log



- If leader crashes, a new leader is elected

# Replication

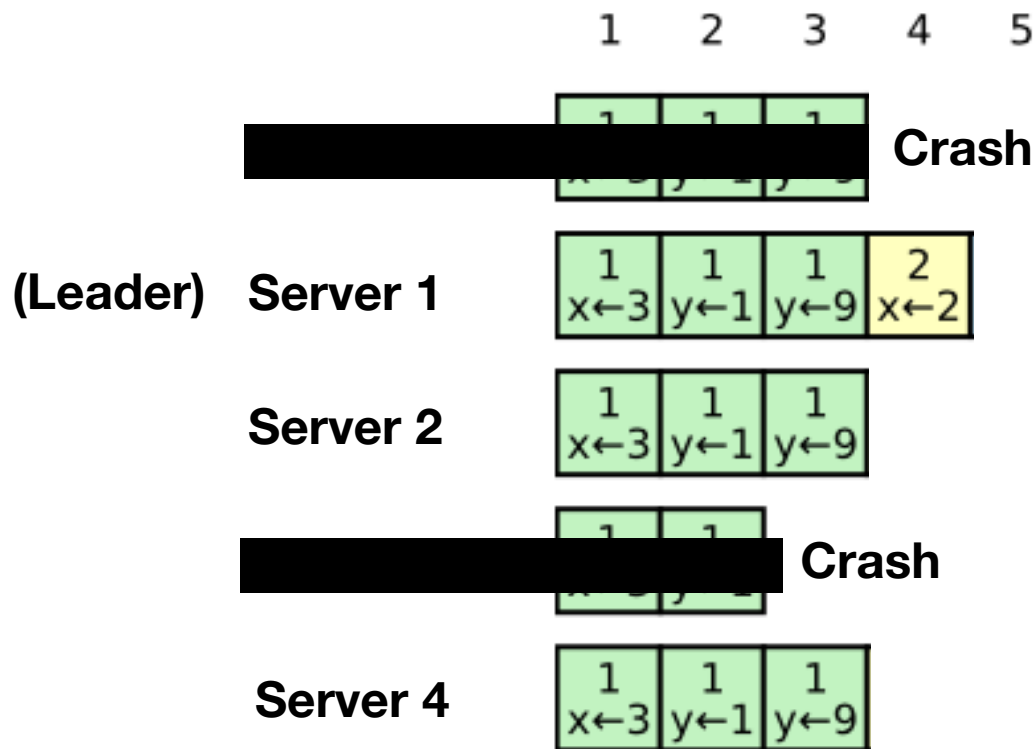
- Fault tolerance achieved by replicating the transaction log



- If leader crashes, a new leader is elected

# Replication

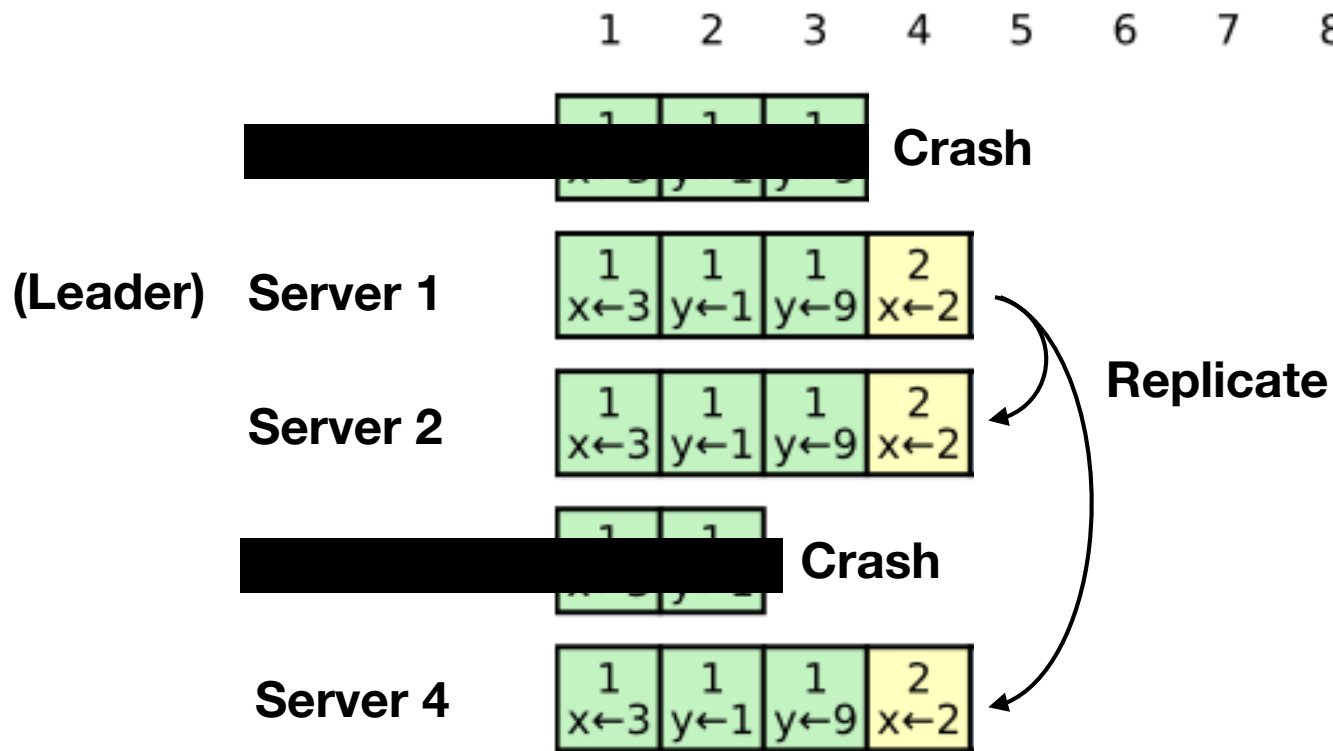
- Fault tolerance achieved by replicating the transaction log



- If leader crashes, a new leader is elected

# Replication

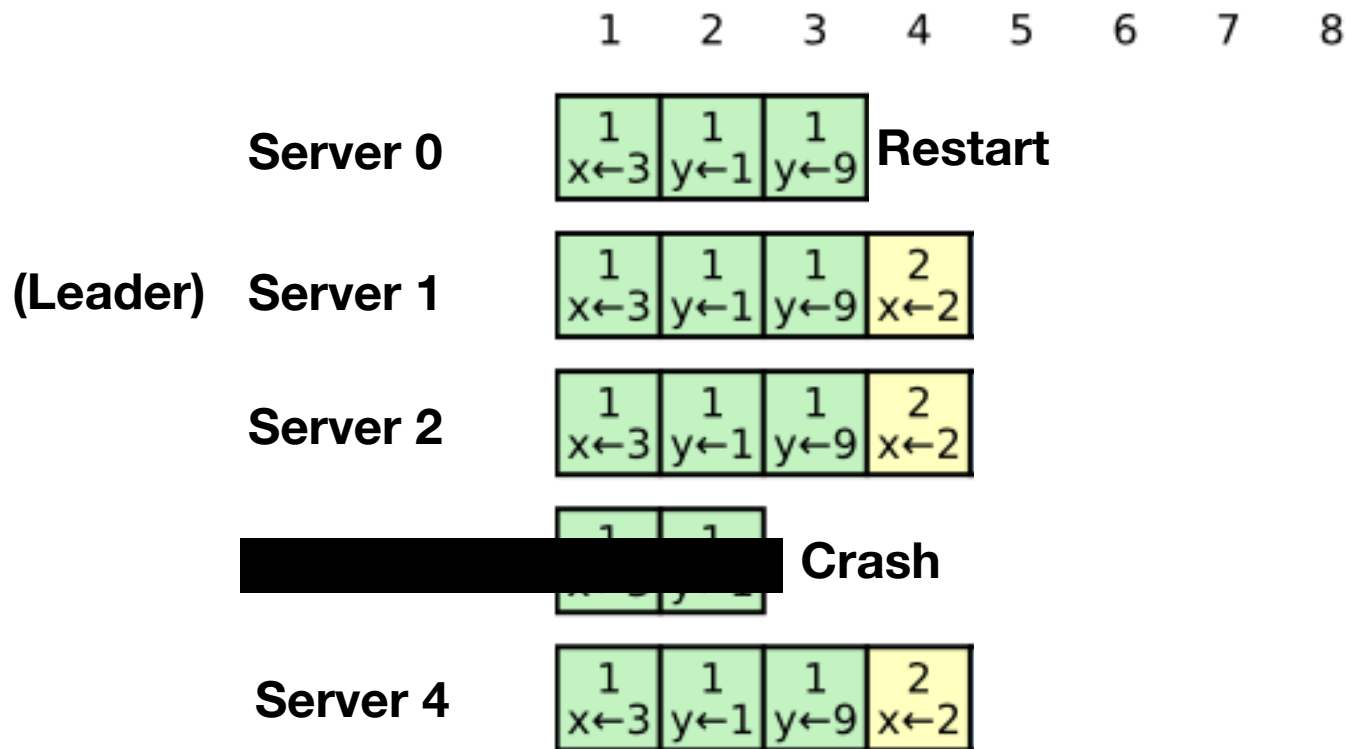
- Fault tolerance achieved by replicating the transaction log



- If leader crashes, a new leader is elected

# Replication

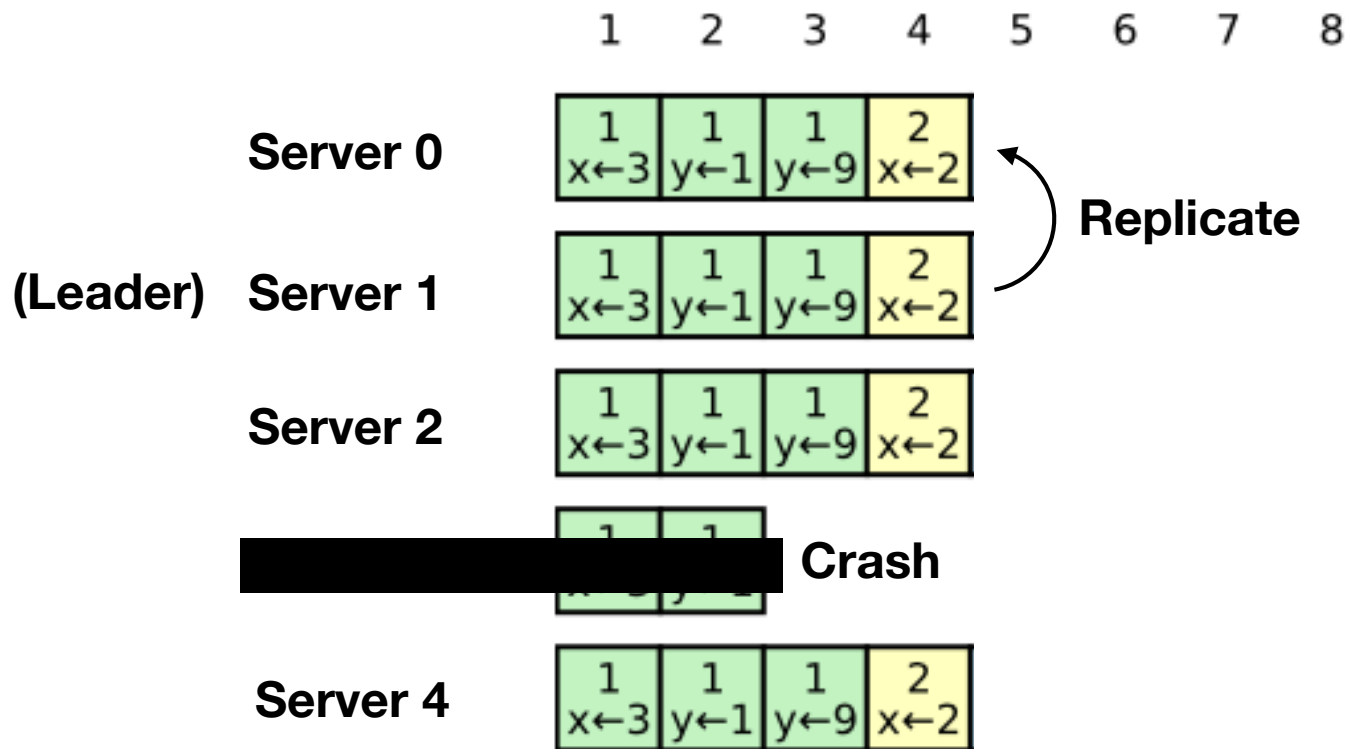
- Fault tolerance achieved by replicating the transaction log



- Restarted machines get updated by leader

# Replication

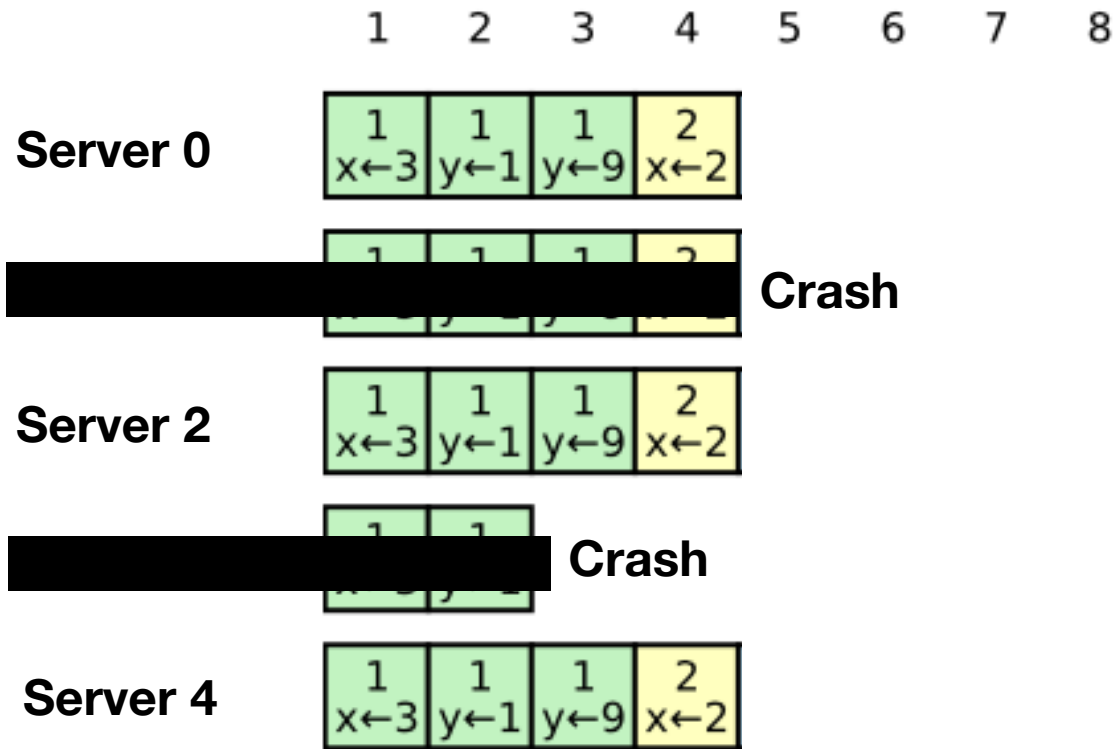
- Fault tolerance achieved by replicating the transaction log



- Restarted machines get updated by leader

# Replication

- Fault tolerance achieved by replicating the transaction log

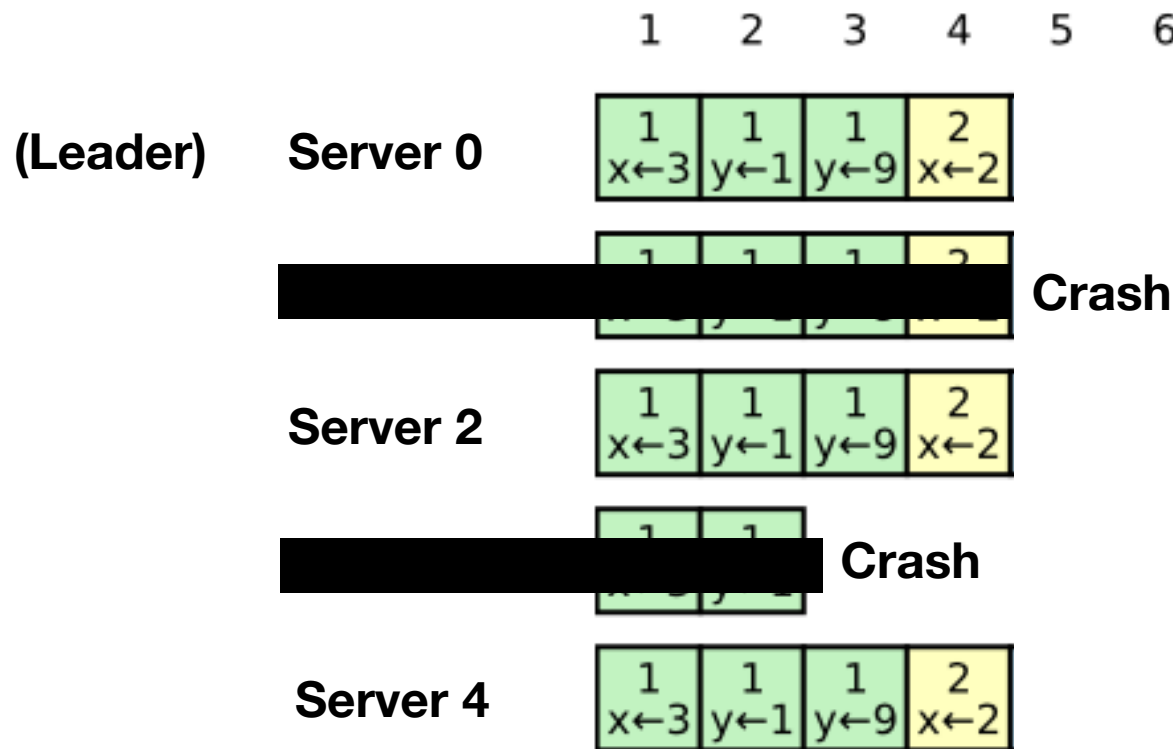


- This continues even if new leader crashes



# Replication

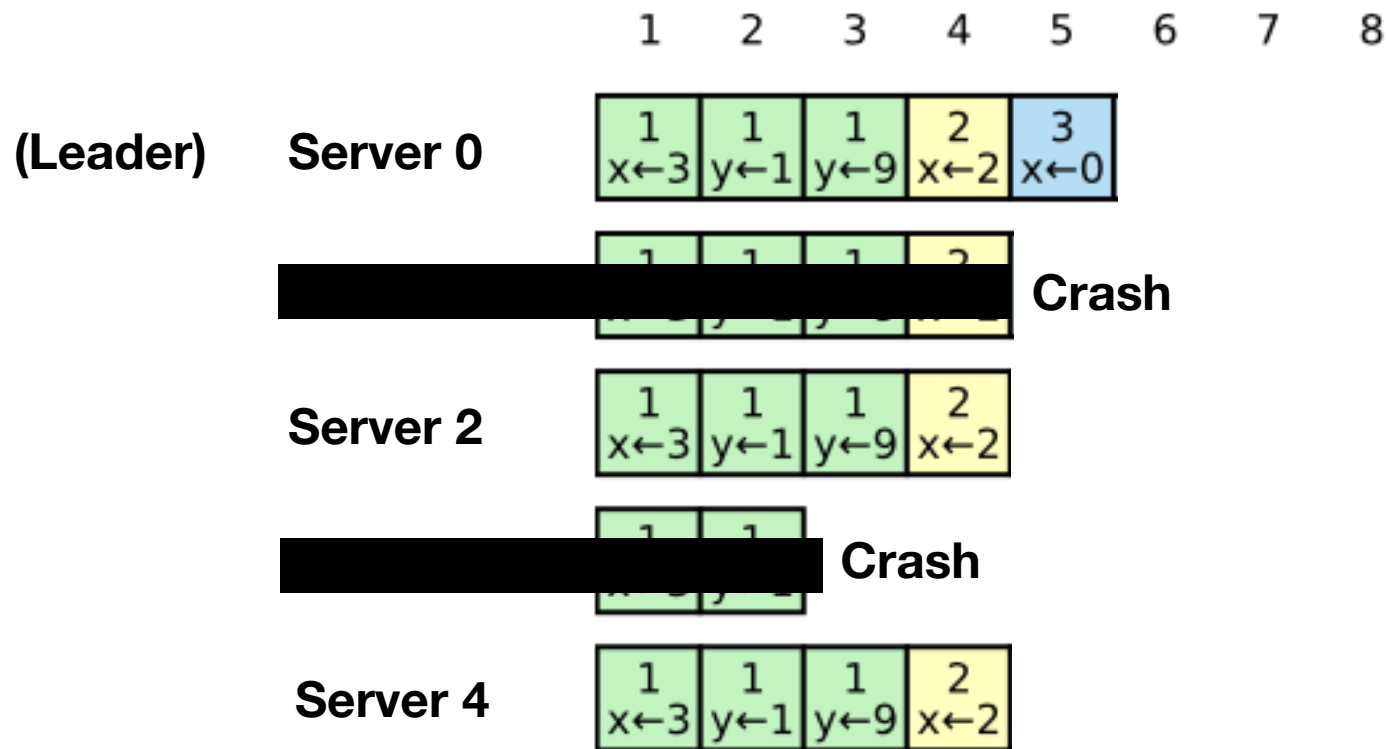
- Fault tolerance achieved by replicating the transaction log



- If there is a quorum, new leader will be elected

# Replication

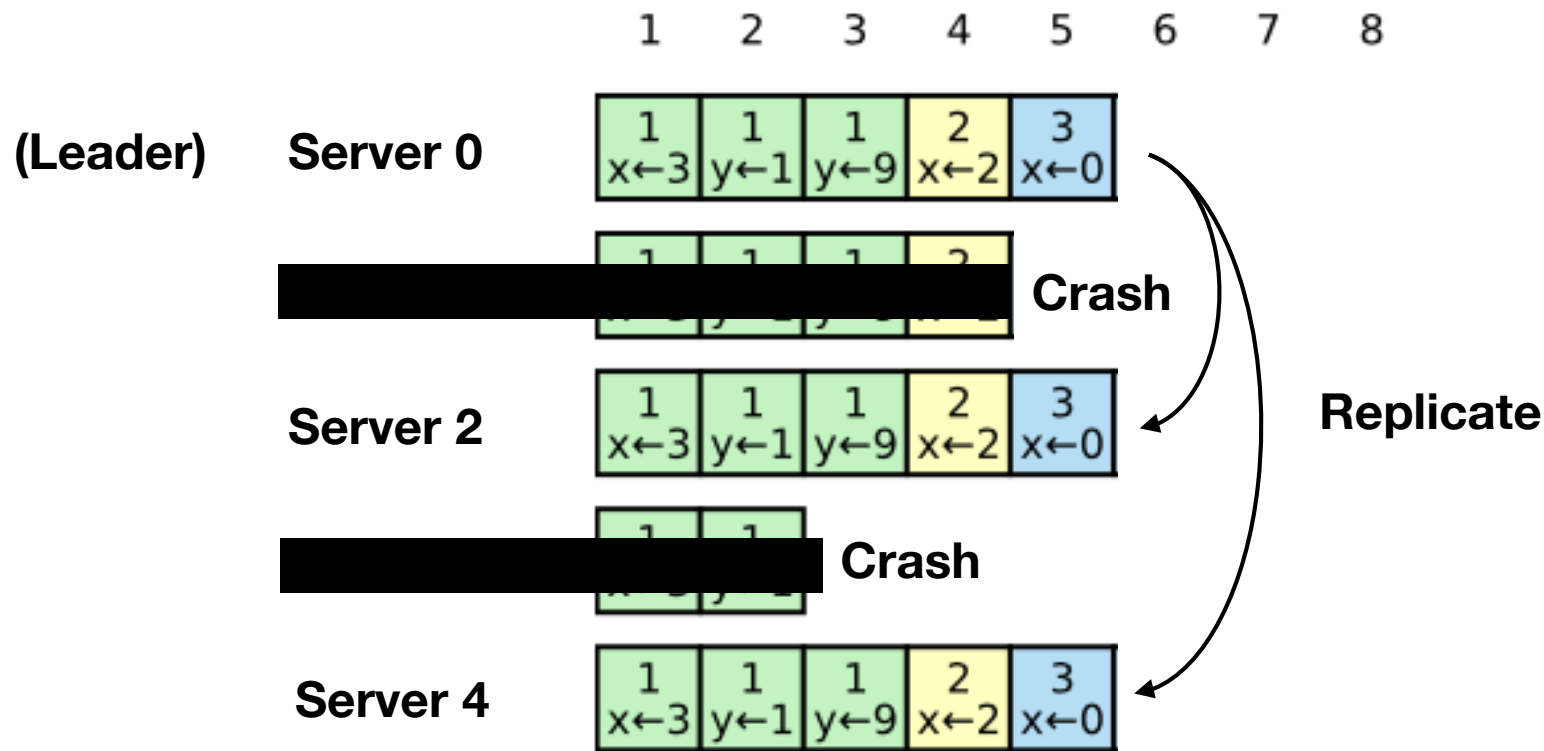
- Fault tolerance achieved by replicating the transaction log



- Operation continues as long as majority are running

# Replication

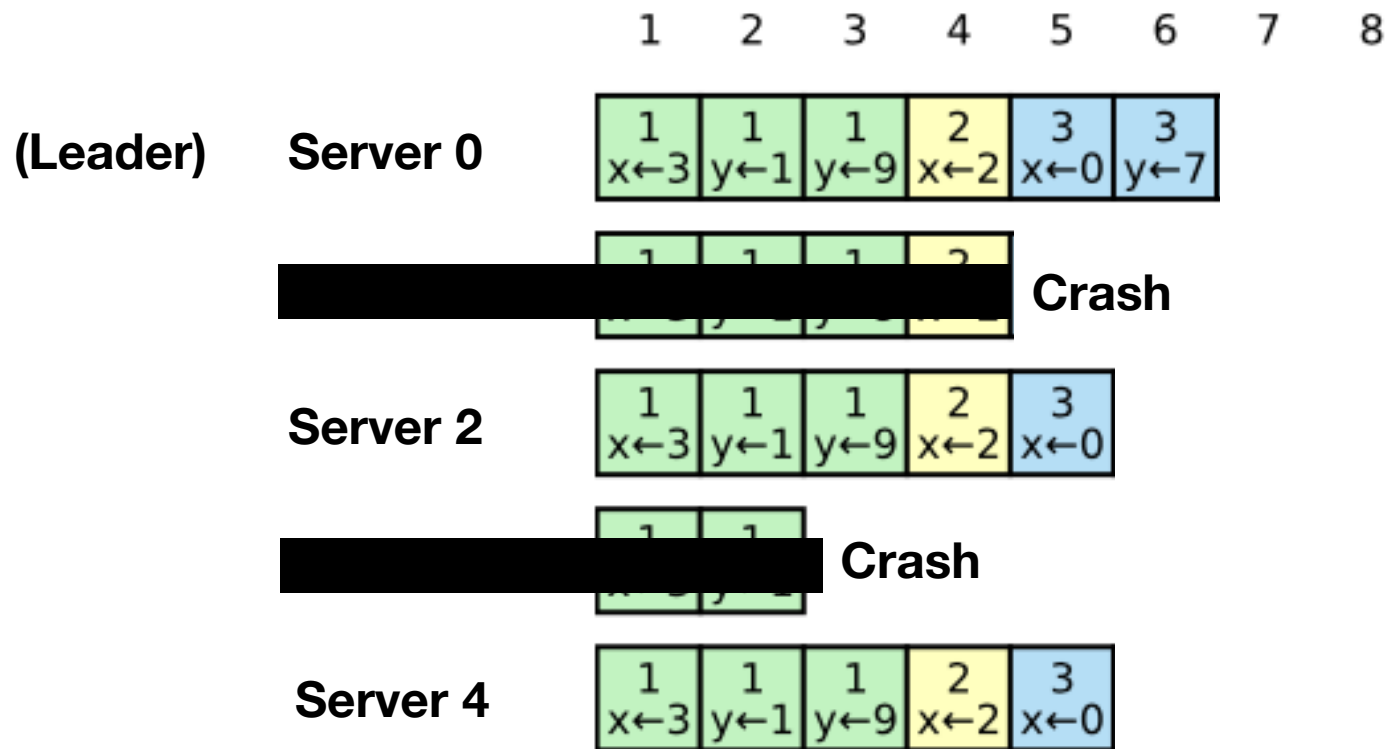
- Fault tolerance achieved by replicating the transaction log



- Operation continues as long as majority are running

# Replication

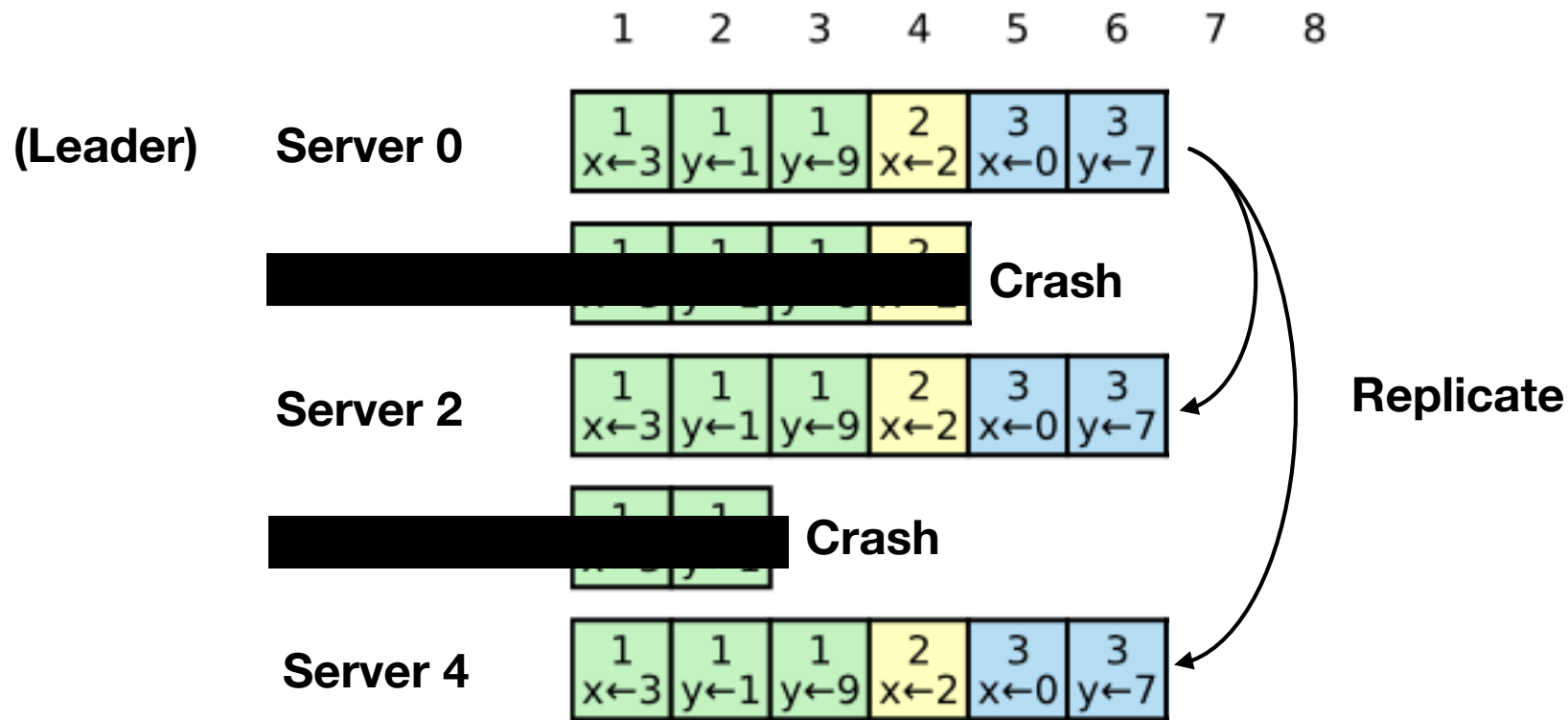
- Fault tolerance achieved by replicating the transaction log



- Operation continues as long as majority are running

# Replication

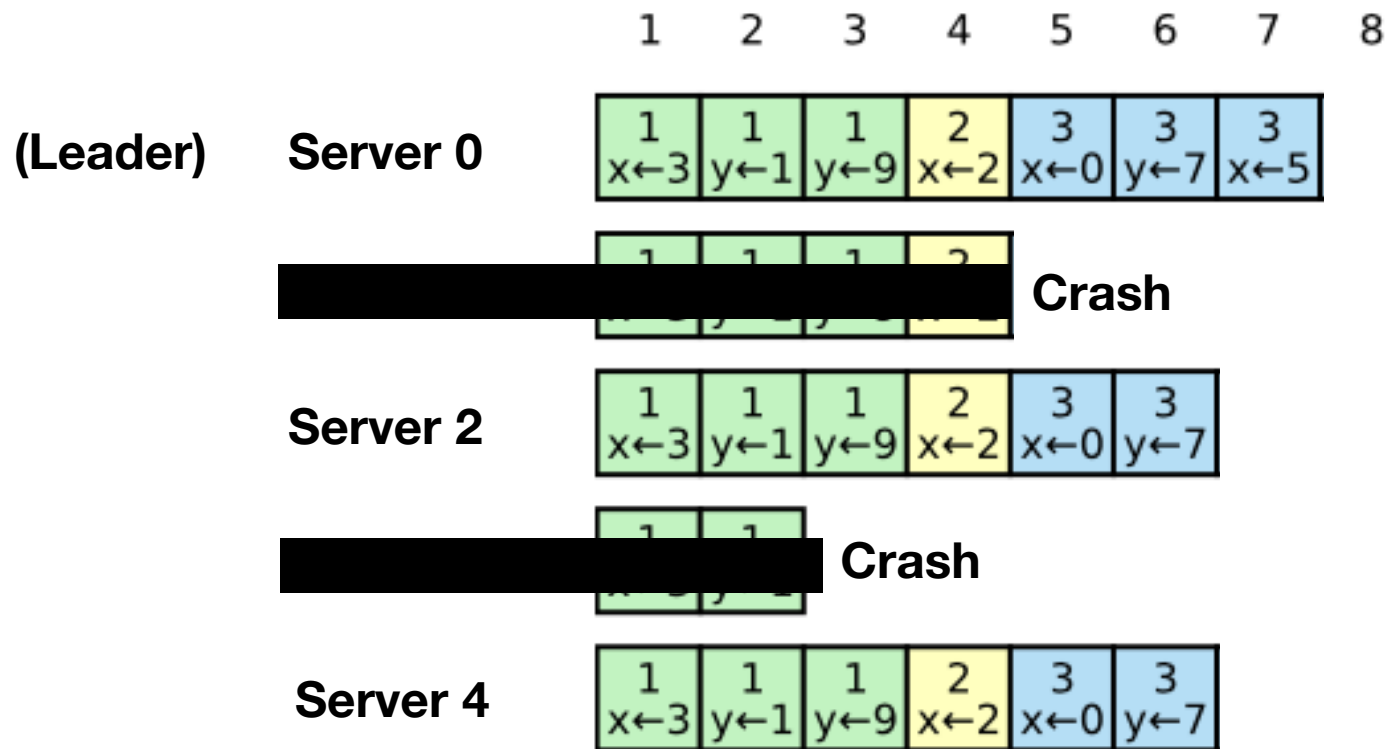
- Fault tolerance achieved by replicating the transaction log



- Operation continues as long as majority are running

# Replication

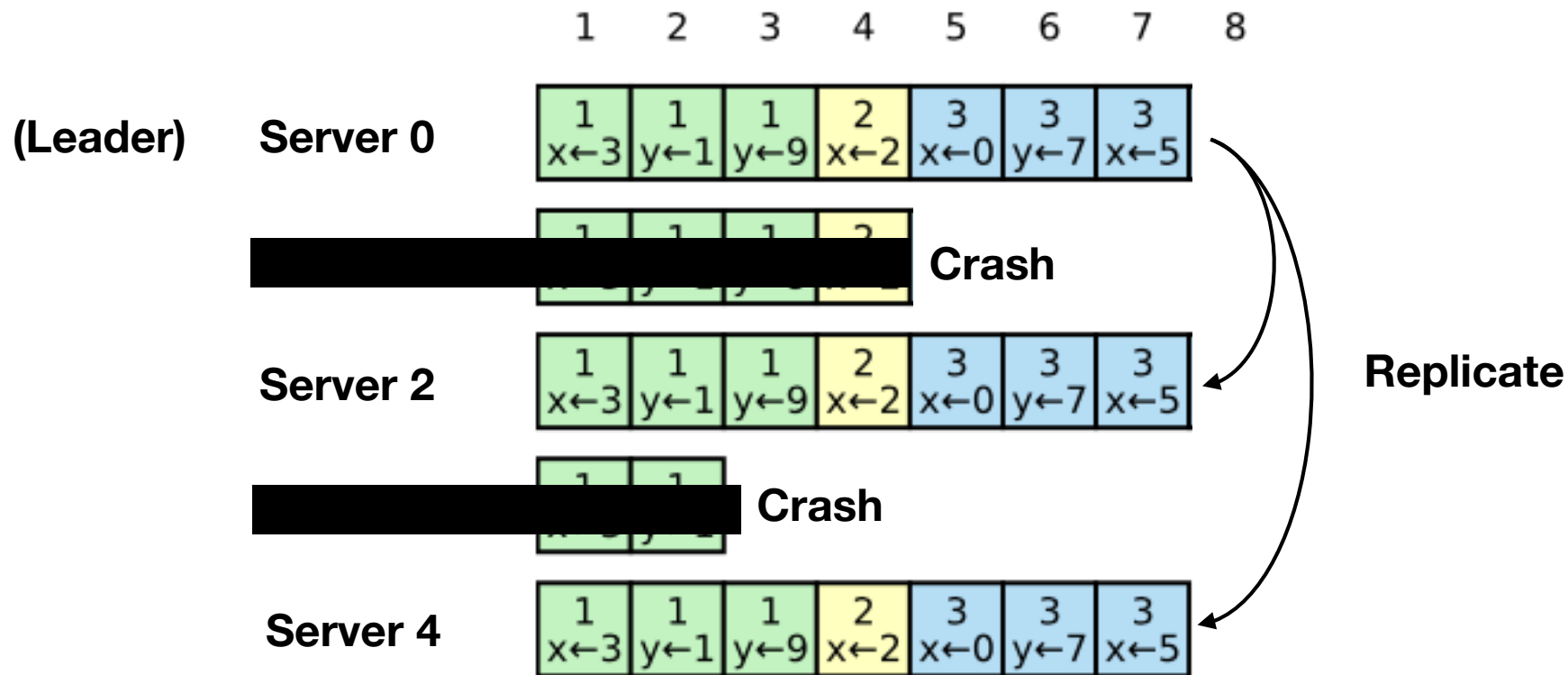
- Fault tolerance achieved by replicating the transaction log



- Operation continues as long as majority are running

# Replication

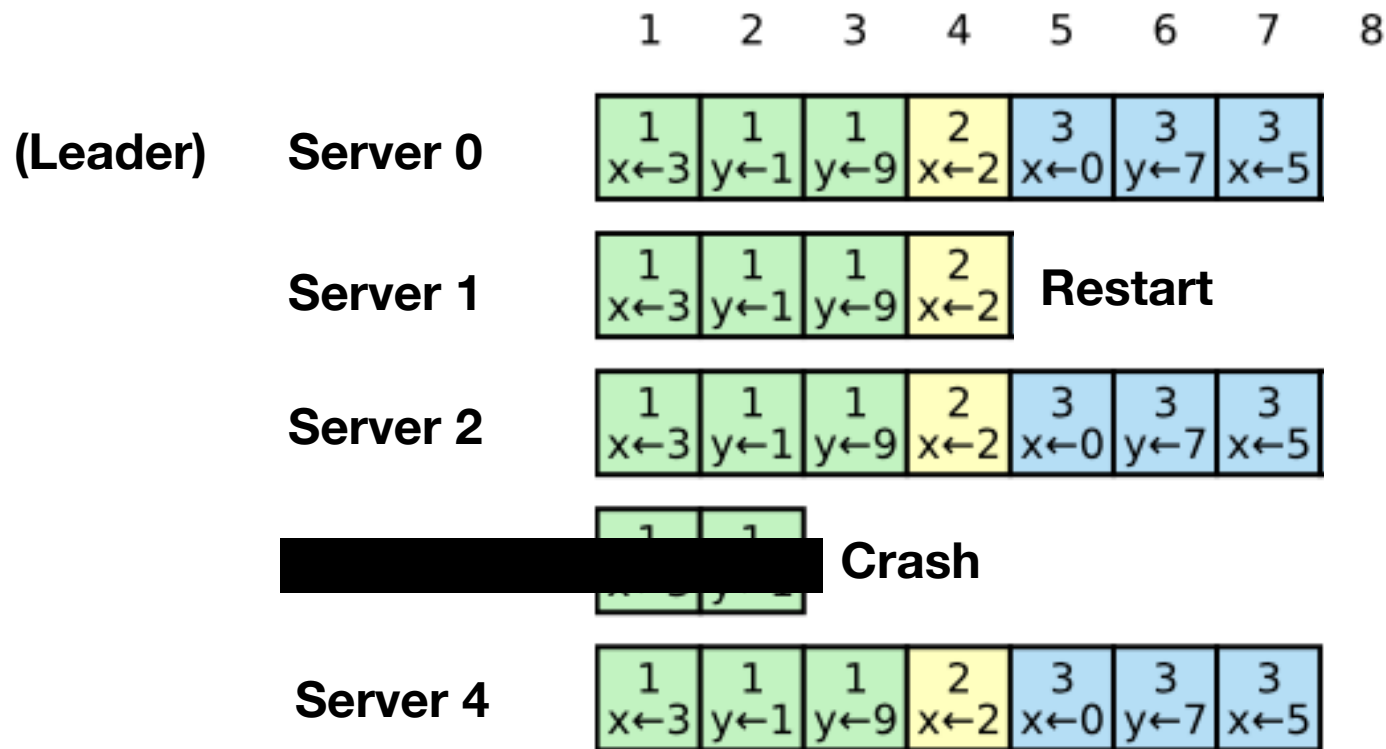
- Fault tolerance achieved by replicating the transaction log



- Operation continues as long as majority are running

# Replication

- Fault tolerance achieved by replicating the transaction log

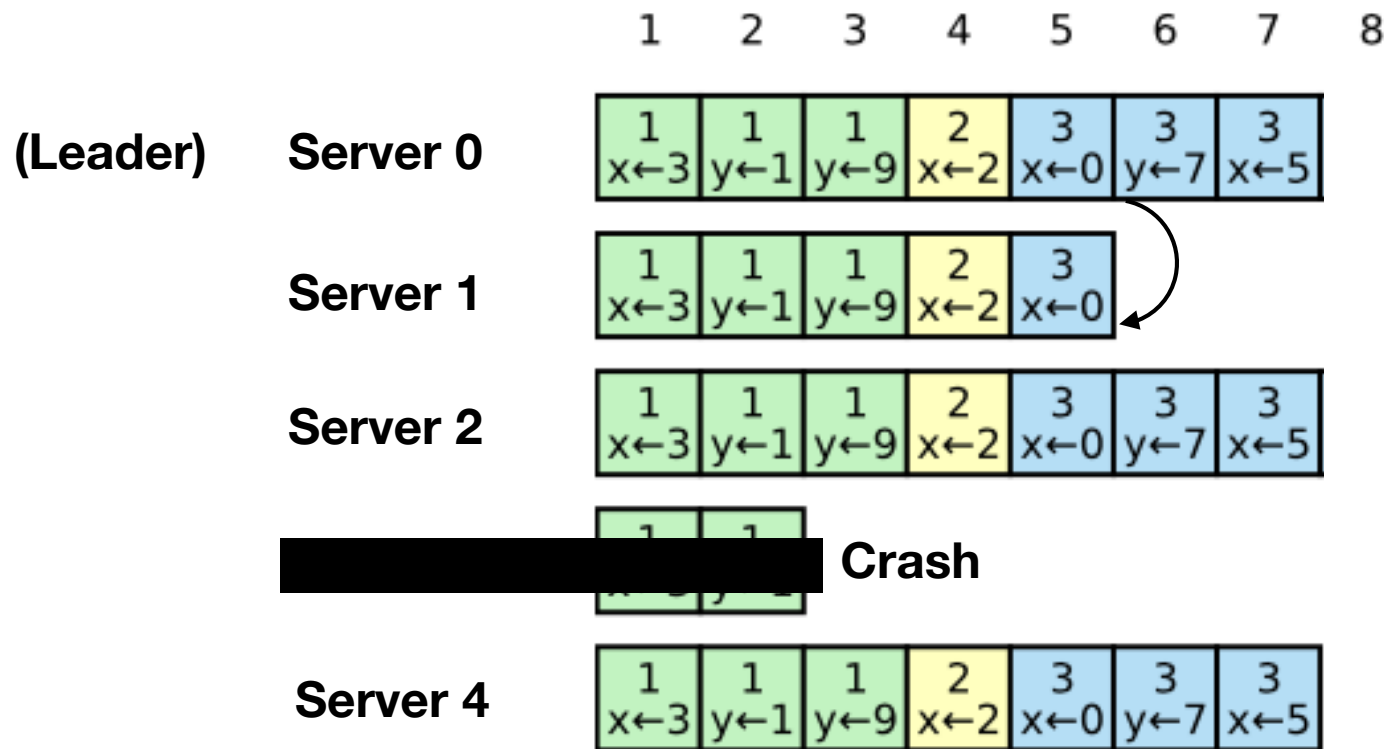


- Again, restarted machines will be updated



# Replication

- Fault tolerance achieved by replicating the transaction log



- Again, restarted machines will be updated

# Replication

- Fault tolerance achieved by replicating the transaction log

|          | 1                     | 2                     | 3                     | 4 | 5 | 6 | 7 | 8 |
|----------|-----------------------|-----------------------|-----------------------|---|---|---|---|---|
| Server 0 | 1<br>$x \leftarrow 3$ | 1<br>$y \leftarrow 1$ | 1<br>$y \leftarrow 9$ |   |   |   |   |   |
| Server 1 | 1<br>$x \leftarrow 3$ | 1<br>$y \leftarrow 1$ | 1<br>$y \leftarrow 9$ |   |   |   |   |   |
| Server 2 | 1<br>$x \leftarrow 3$ | 1<br>$y \leftarrow 1$ | 1<br>$y \leftarrow 9$ |   |   |   |   |   |
| Server 3 | 1<br>$x \leftarrow 3$ | 1<br>$y \leftarrow 1$ |                       |   |   |   |   |   |
| Server 4 | 1<br>$x \leftarrow 3$ | 1<br>$y \leftarrow 1$ | 1<br>$y \leftarrow 9$ |   |   |   |   |   |

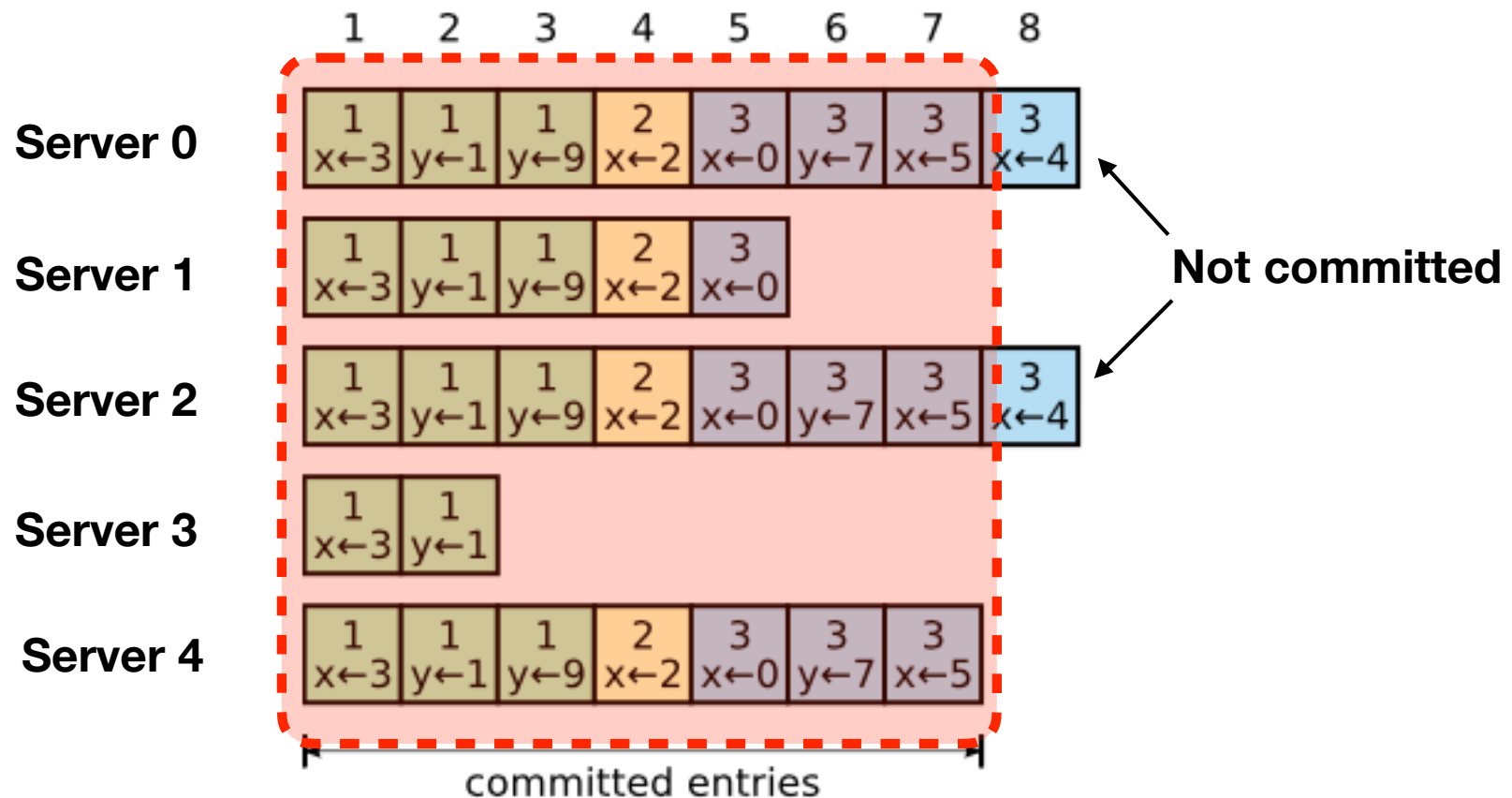
**Aside:**

**Replicated Logs == Raft**



# Majority Rules

- Transactions are "committed" by achieving consensus



- Consensus means replication on a majority (not all)

# Complications

- There are many failure modes
  - Leaders can die
  - Followers can die
  - The network can die
- Yet, it all recovers and heals itself. For example, if a follower dies, the leader will bring the restarted server back up to date by giving it any missed log entries.
- But, there is a lot of book-keeping and subtle detail

# The Plan

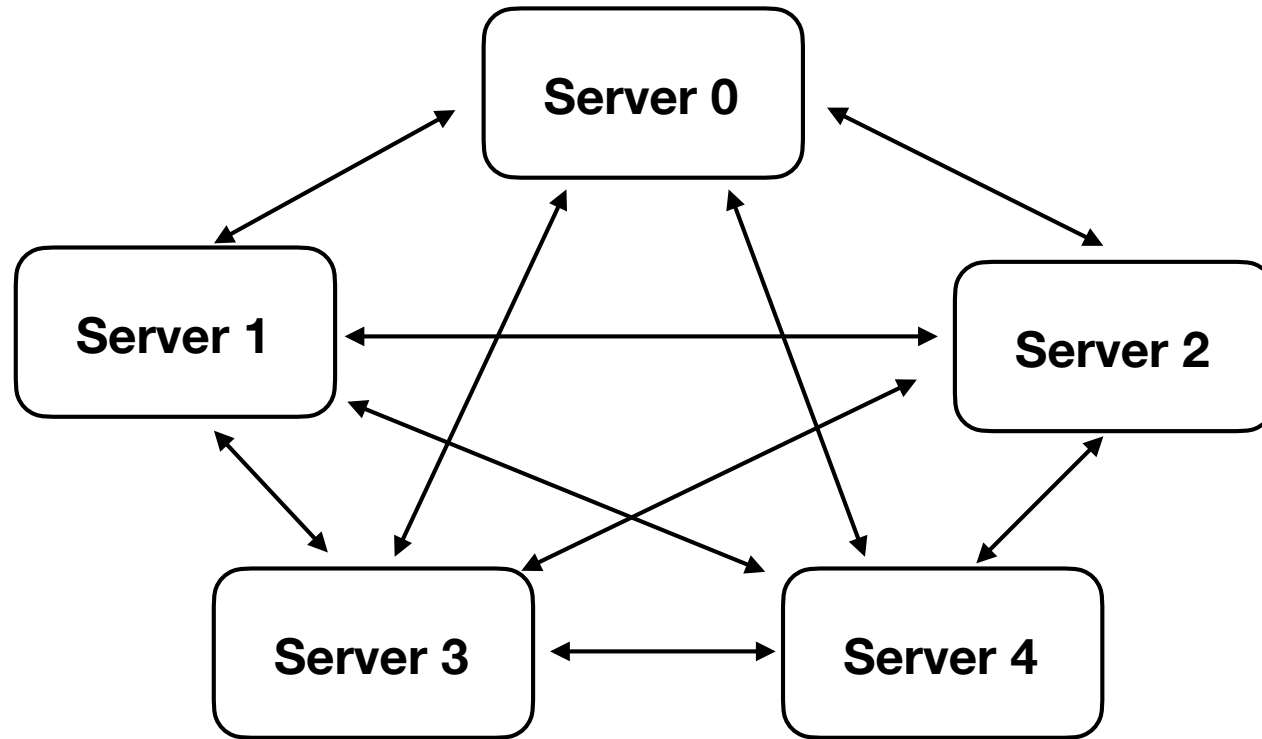
- We will start with some foundational topics
  - Technical: Messages, RPC, Event Systems, etc.
  - Design: Problem modeling
- There will be a lot of open-coding (work on Raft)
- Key to success: TAKE. IT. SLOW.
  - Read/study the problem.
  - An hour of thinking is better than a day of debugging

Part 1

# Foundations

# Raft and Networks

- Raft involves a cluster of identical servers



- They exchange messages over a network

# Raft Configuration

- You'll need to maintain a network config

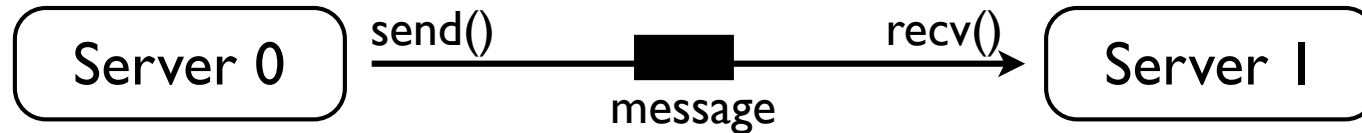
```
# raftconfig.py
```

```
SERVICES = {  
    0: ('123.45.67.89', 15000),  
    1: ('123.45.67.100', 15000),  
    2: ('123.45.67.113', 15000),  
    3: ('123.45.67.114', 15000),  
    4: ('123.45.67.192', 15000),  
}
```

- This is the cluster of machines that run Raft
- For project purposes, hardcode this somewhere

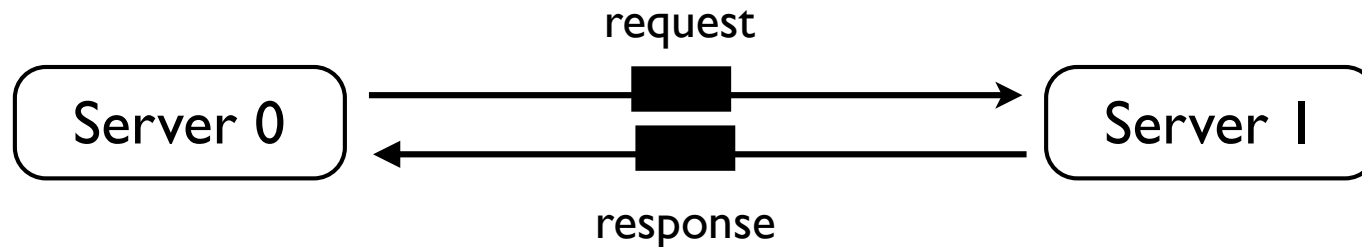


# Message Passing



- A message is a discrete packet of bytes
- Indivisible (treated as a single object)
- Mental model: email

# Remote Procedure Call



- A function call over the network (2 messages)

```
# client
def func(x, y, z):
    msg = encode(x, y, z)
    send_message(msg)
    resp = recv_message()
    return decode(resp)

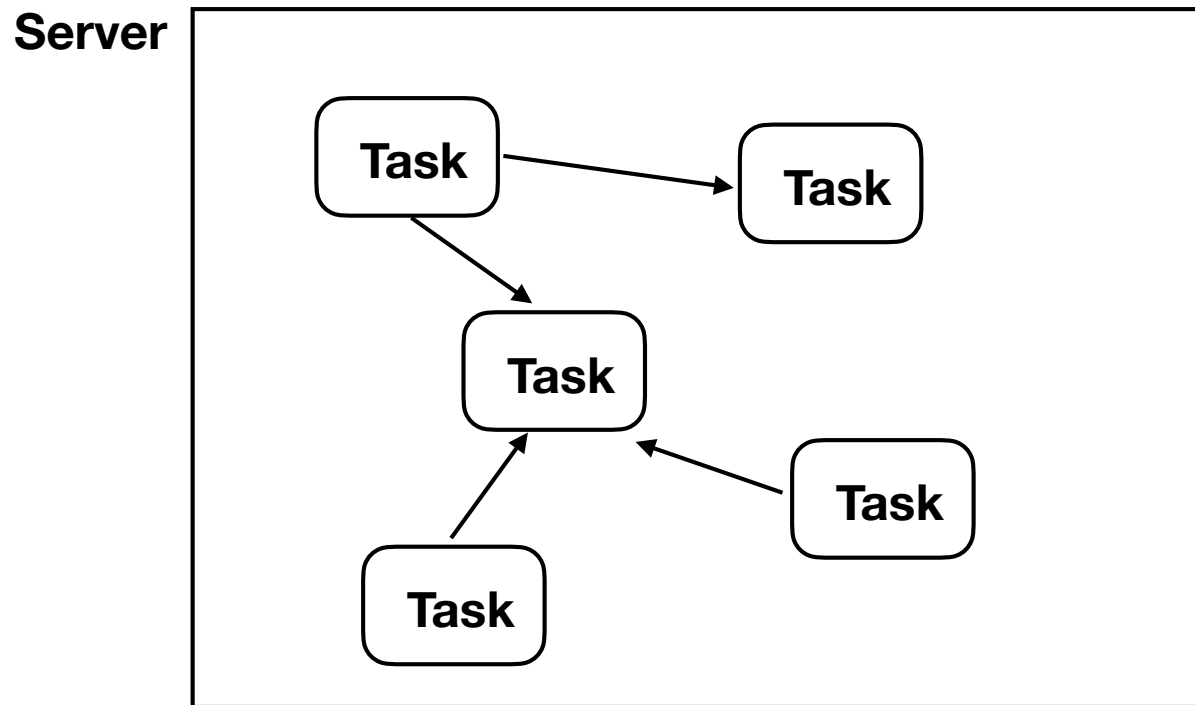
# server
def func_handler():
    msg = recv_message()
    x, y, z = decode(msg)
    result = func(x, y, z)
    resp = encode(result)
    send_message(resp)

def func(x, y, z):
    ...
    return result
```

Dotted arrows indicate the flow of messages: one from `send_message(msg)` in the client to `recv_message()` in the server handler, and another from `send_message(resp)` in the server handler to `recv_message()` in the client.

# Concurrency

- Server may have many internal tasks that need to operate concurrently



- Event monitoring, timers, message handling, etc.

# Threads

- For internal concurrency, use threads

```
# Some function to launch in a thread
def func(x, y, z):
    ...

def main():
    t = threading.Thread(target=func, args=(1,2,3))
    t.start()
    ...
    t.join()    # Optional (wait for thread to finish)
```

- Once launched, thread runs independently
- Can wait for it (but that's about it)

# Commentary

- Programming with threads is a big topic.
- Threads share memory program resources.
- Coordination of mutable state is difficult
- Typically involves locking, synchronization, etc.

# Thread Queues

- Threads can coordinate with queues
- Idea: Each thread is an "independent task"
- Threads send messages to each other
- No shared state other than the queues

# Queue Example

```
from queue import Queue
```

```
def producer(q):  
    for i in range(10):  
        q.put(i)  
        time.sleep(1)  
    q.put(None)
```

```
def consumer(q):  
    while True:  
        i = q.get()  
        if i is None:  
            break  
        print("Got:", i)
```

```
q = Queue()  
threading.Thread(target=producer, args=(q,)).start()  
threading.Thread(target=consumer, args=(q,)).start()
```

# Project I

- Warmup exercise
- Implement a simple key-value store

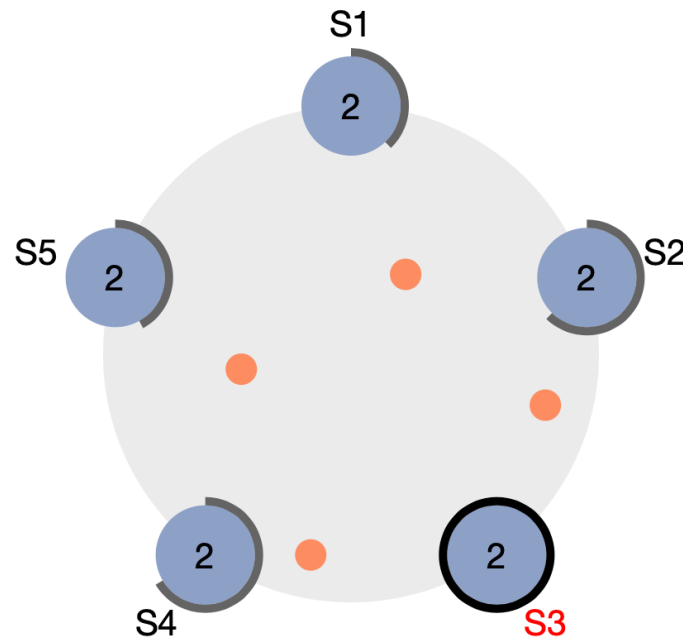


Part 2

# Event Driven Programming

# Raft Visualization

- Go to <https://raft.github.io>
- How would you describe what you see?



# Raft Servers

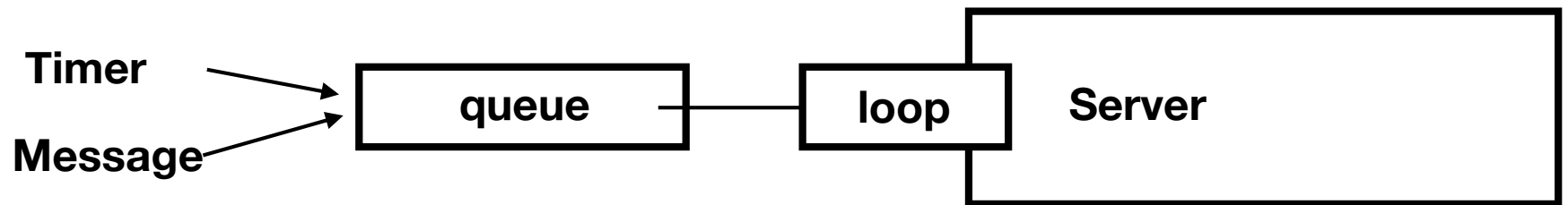
- Servers operate independently
- Handle incoming messages
- Also manage timers/timeouts
- The exact sequencing is unpredictable
- It's not easy to describe with sequential steps

# Event Driven Model

- Raft servers are better conceptualized through an "event model"
- Events
  - Receive a message
  - Timer expiration
- Actions only take place in response to events
- Question: How to implement?

# Event Serialization

- Events can be serialized via queues

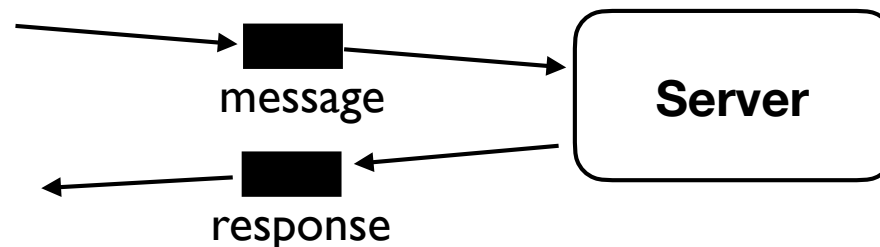


```
while True:
    evt = queue.get()
    if isinstance(evt, Timer):
        handle_timer(evt)
    elif isinstance(evt, Message):
        handle_message(evt)
```

- Server consists of event handlers
- One event at a time!

# Server Actions

- In response to events, servers carry out some kind of action
- Might be some internal operation
- Typically, a response message is sent



# Project 2

- Implement a simple event-driven system
- Traffic Light Problem.
- Involves control of many elements all at once.

Part 3

# Starting with Raft



# How to Start?!?

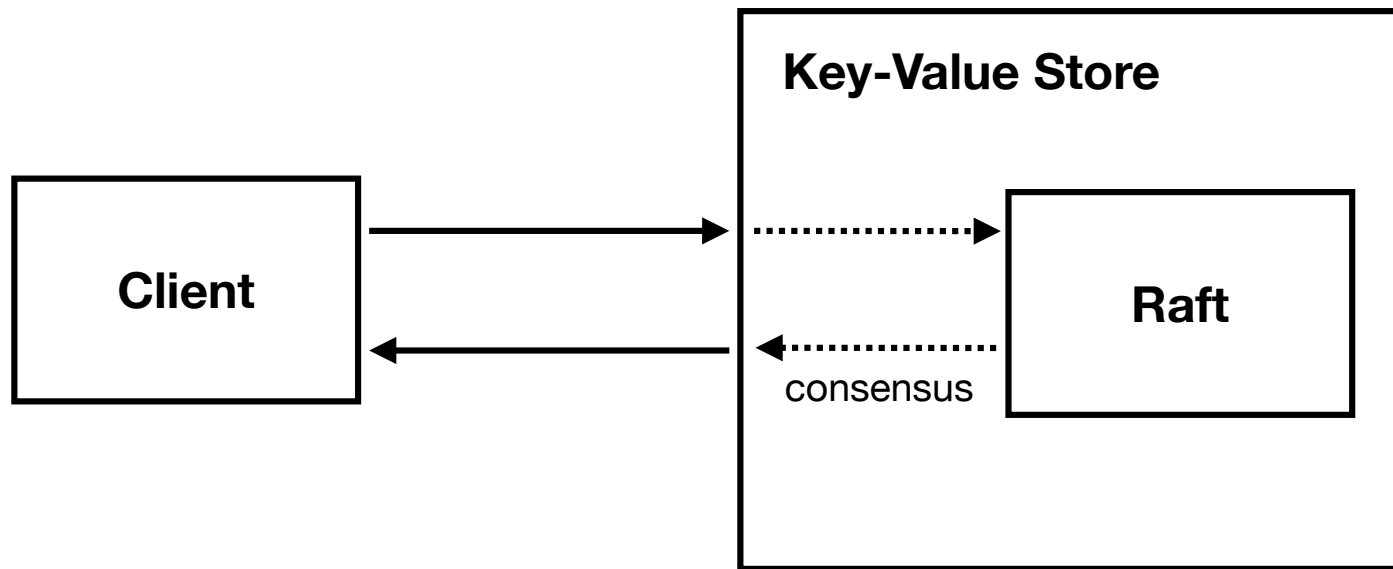
- How does one start to work on Raft?
- There are many facets to it
- An issue: Everything is interconnected
- It is difficult to make forward progress

# Disclaimer

- I do NOT claim to know the "best" way to start with this project.
- However, I know that the project requires a few critical "working" parts to get anywhere
  - Needs to interact with an application.
  - Needs a log
  - Needs messaging.

# Raft Application

- Raft is a module that serves an application



- It is embedded in the application
- An internal implementation detail

# Option 1: Application

- Start with the application
- Figure out the Raft embedding
- Focus on the flow of data through the system

# The Log

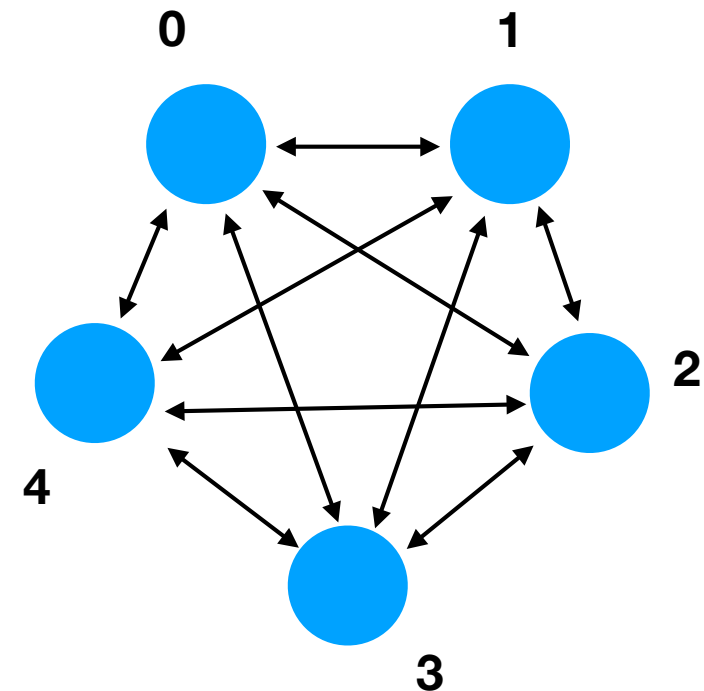
- Raft is based on the idea of keeping a replicated transaction log
- In fact, the whole algorithm is focused on this one problem (log replication)

# Option 2 : Algorithm

- Focus on the Raft algorithm itself
- Specifically, log replication
- Approach it from a high-level
- Make it something that can be tested

# Messaging

- Raft involves a cluster of servers
- Servers send messages



# Option 3 : Systems

- Focus on the systems-level components
- Server configuration
- Networking/Messaging
- Logging/monitoring
- Build the environment in which Raft will run



# Project

- Read section 5 and 8 of the Raft paper
- Contemplate the possible "starting point" for a coding project
- Start sketching out some ideas in code
- Could just be empty classes/functions
- Goal is to better understand the problem

## Part 4

# The Log

# Transaction Log

- Raft is based on the idea of keeping a replicated transaction log
- The log records the state changes on some (unspecified) arbitrary system
- In event of a crash: You replay the log to get back to a consistent state

# Example: KV-Store

## Operations

```
kv.set( 'foo' , 42 )  
kv.get( 'foo' )  
kv.set( 'bar' , 13 )  
kv.set( 'foo' , 23 )  
kv.set( 'spam' , 100 )  
kv.get( 'spam' )  
kv.delete( 'foo' )
```

## Transaction Log

```
( 'set' , 'foo' , 42 )  
( 'set' , 'bar' , 13 )  
( 'set' , 'foo' , 23 )  
( 'set' , 'spam' , 100 )  
( 'delete' , 'foo' )
```

- Note: Log only needs to record state changes

# Commentary

- The log is probably the MOST important part of understanding the Raft algorithm
- The whole point of the algorithm is to maintain and to replicate the log.
- Everything is about the log.
- Everything.
- The log.

# Replication

- What does it mean to "replicate?"
- For Raft, the log is identical on all servers
- Same entries, same position.
- The same.

# The Log is (is not?) a List

- Is the transaction log just a list?

```
log = [ ]
```

```
log.append(entry1)
```

```
log.append(entry2)
```

```
...
```

- Answer: It's complicated.
- Conceptually, it's a list, but it needs to be a "fault-tolerant" list.
- Imagine if the "append" could fail somehow.

# Problem: Persistence

- Raft assumes the log lives on non-volatile storage

```
def append_entry(log, entry):  
    # write entry to NV storage  
    ...  
    # Only return when actually stored  
    return success
```

- The issue: What if a server dies due to catastrophic system failure (power-loss, hardware fault, kernel crash, etc..)

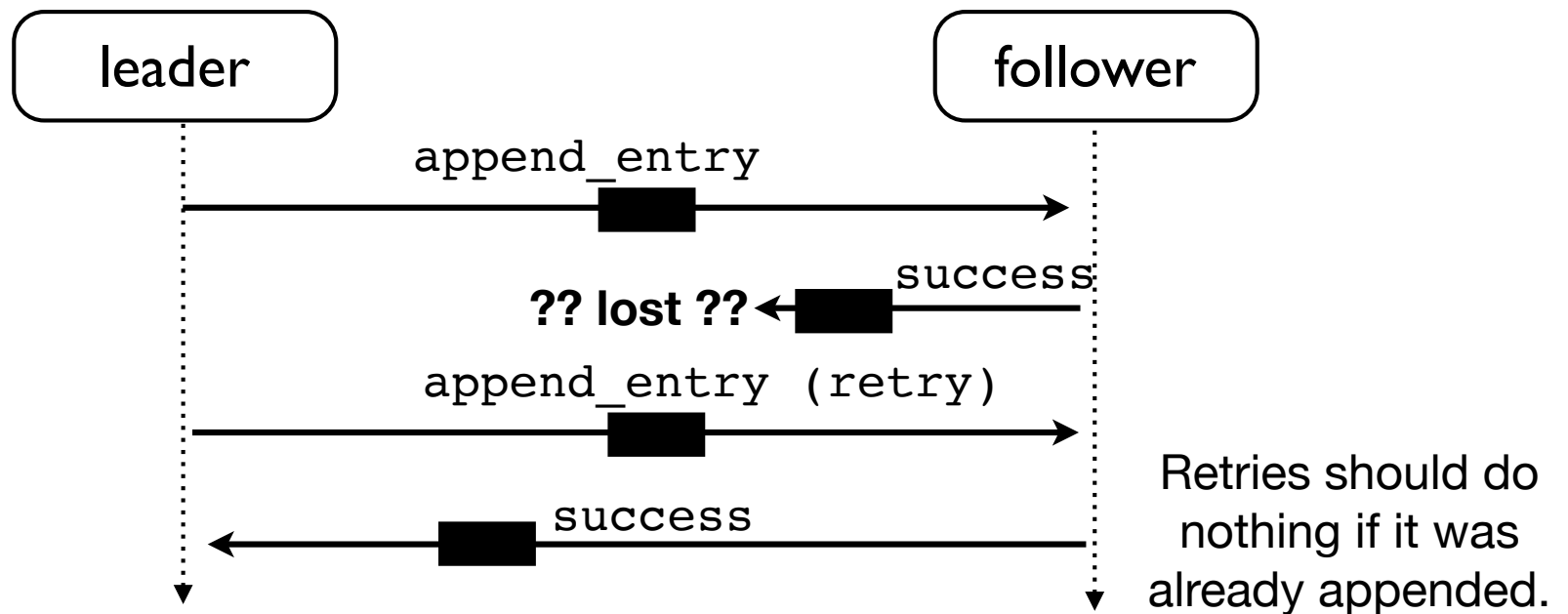


# Problem: Repetition

- Appends must be repeatable ("idempotent")

```
append_entry(log, entry)
append_entry(log, entry) # Does nothing (already appended)
```

- The issue: Network faults/retries



# How to solve?!?

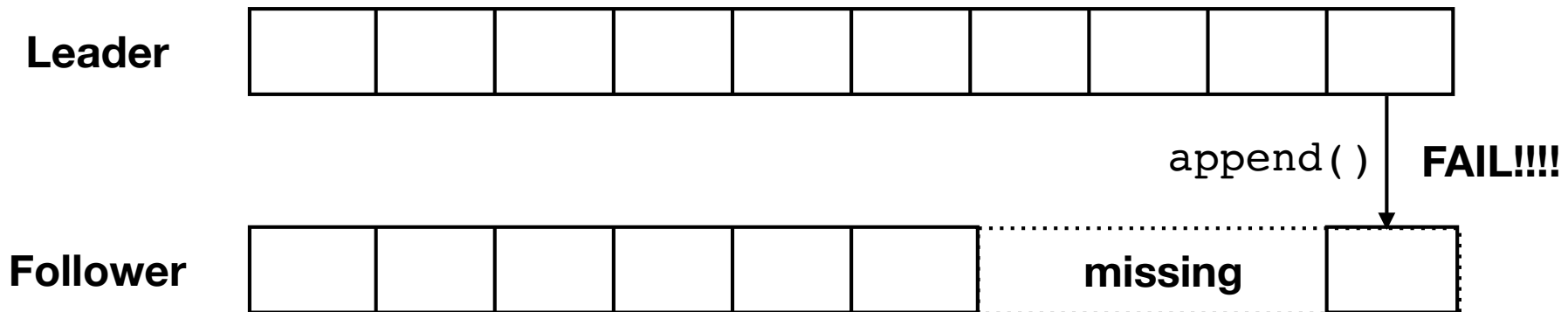
- Make appends specify an exact position (index)

```
def append_entry(log, index, entry):  
    if len(log) > index:  
        log[index] = entry          # Replace existing entry  
    elif len(log) == index:  
        log.append(entry)          # Add new entry at end  
    else:  
        # ????
```

```
>>> log = [ 'x', 'y' ]  
>>> append_entry(log, 2, 'z')  
>>> append_entry(log, 2, 'z')      # Duplicate  
>>> log  
[ 'x', 'y', 'z' ]  
>>>
```

# Problem: Gaps

- What happens if a "follower" disappears for awhile and then comes back??



- This is NEVER allowed.
- The log is continuous. NO. GAPS. EVER.

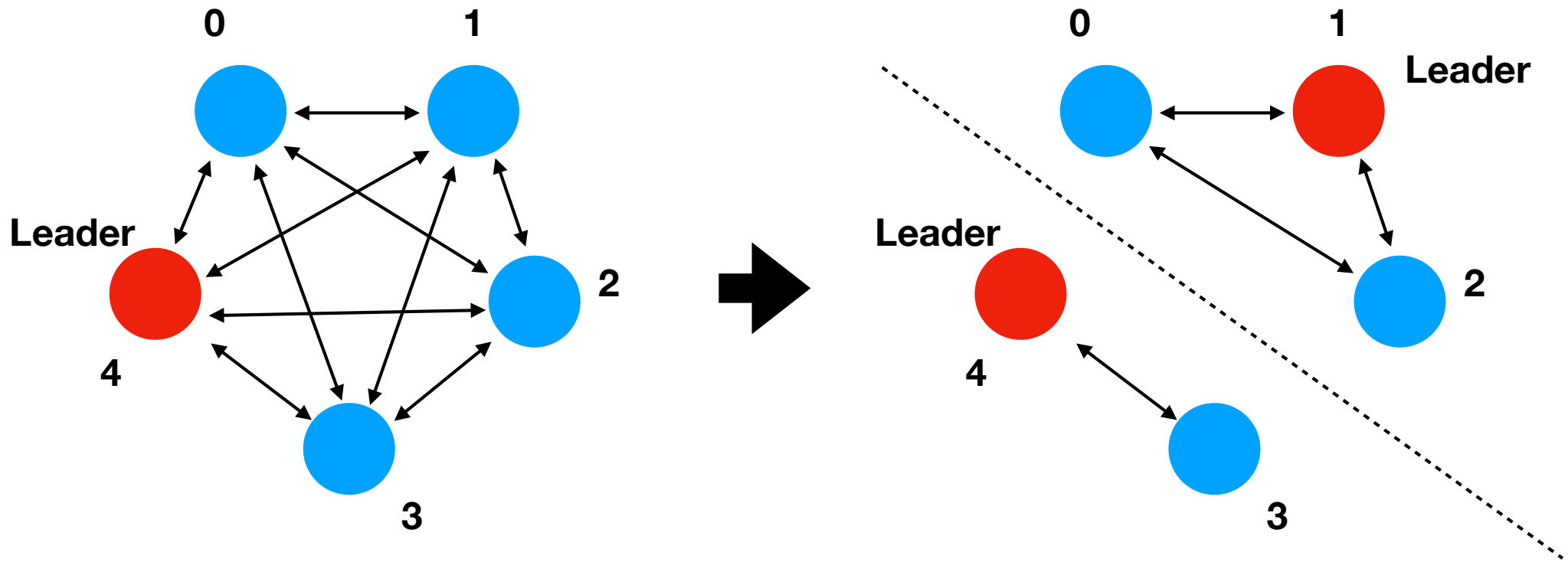
# How to solve?!?

- Report success/fail for each append

```
def append_entry(log, index, entry):  
    if len(log) > index:  
        log[index] = entry  
        return True  
    elif len(log) == index:  
        log.append(entry)  
        return True  
    else:  
        return False      # Would result in gap  
  
>>> log = [ 'x', 'y' ]  
>>> append_entry(log, 2, 'z')  
True  
>>> append_entry(log, 10, 'w')  
False  
>>>
```

- Fail indicates missing data

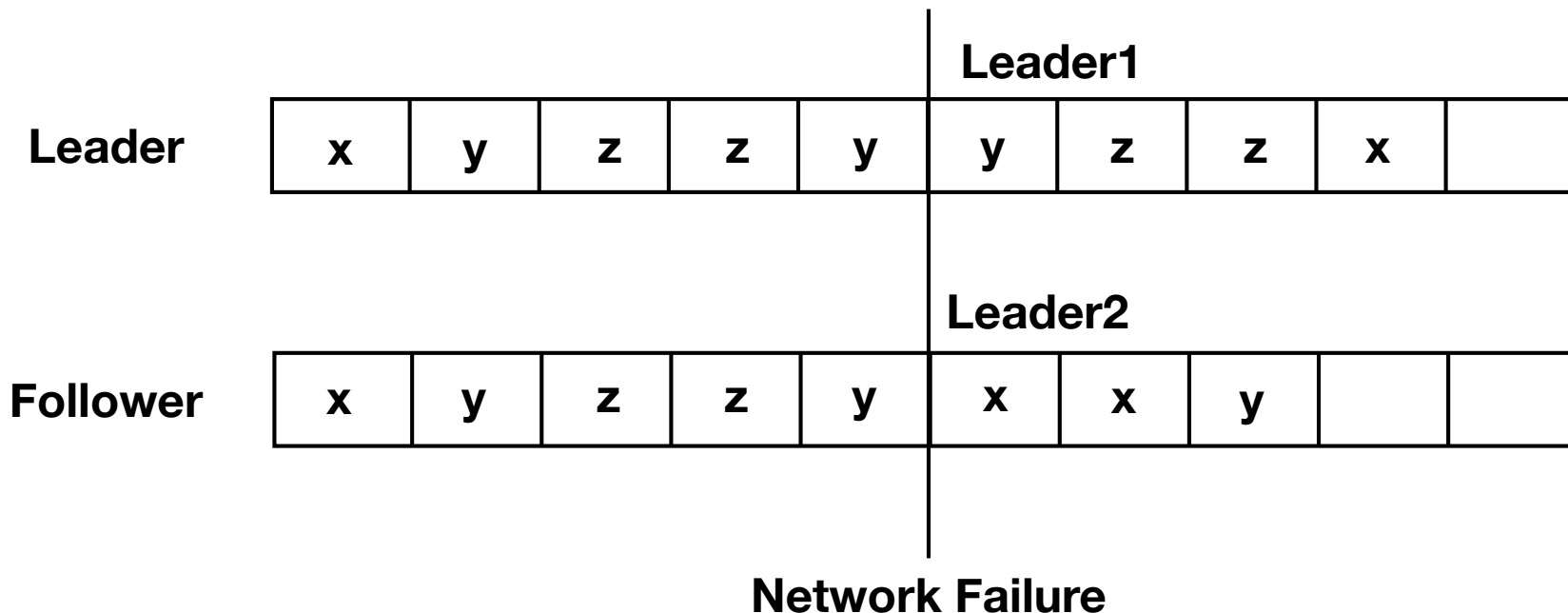
# Problem: Divergence



- Raft could experience a network partition that results in two "leaders."
- But the ex-leader may not know it's cut off.

# Problem: Divergence

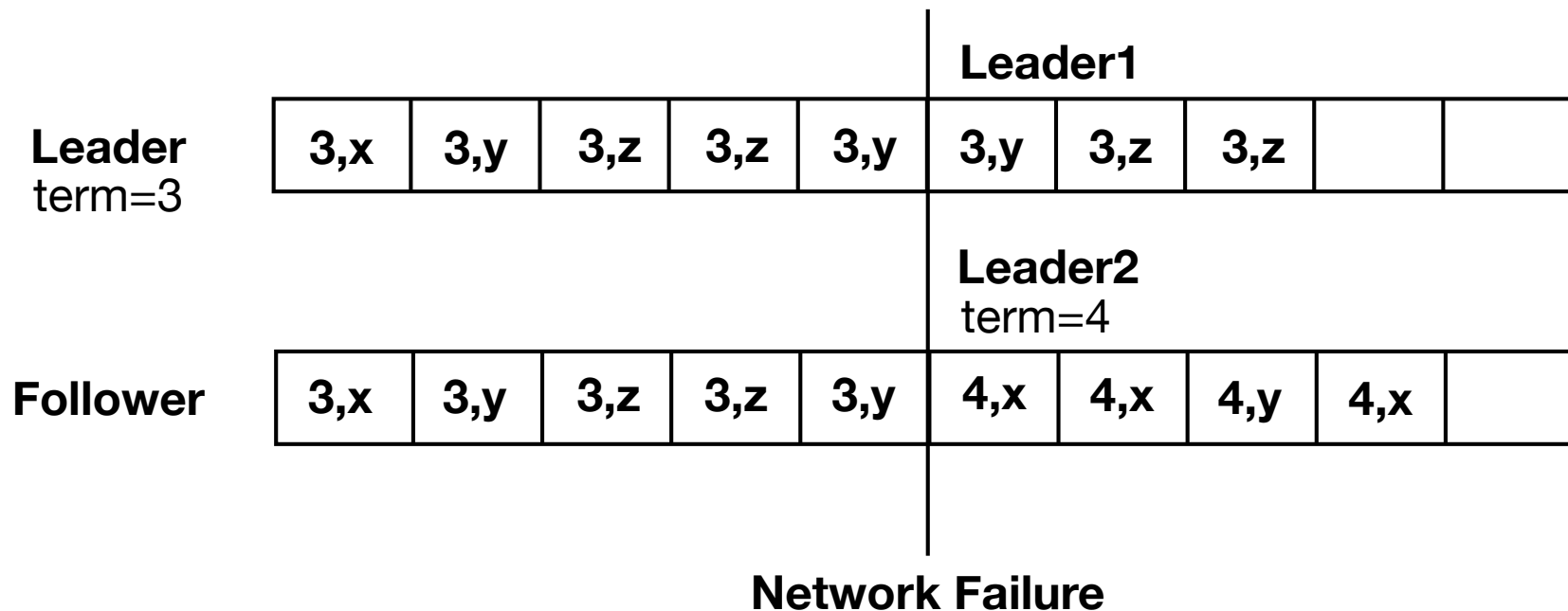
- Two "leaders" might cause a log divergence



- Because of the failure, they don't see each other
- Practical issue: What happens when it heals?

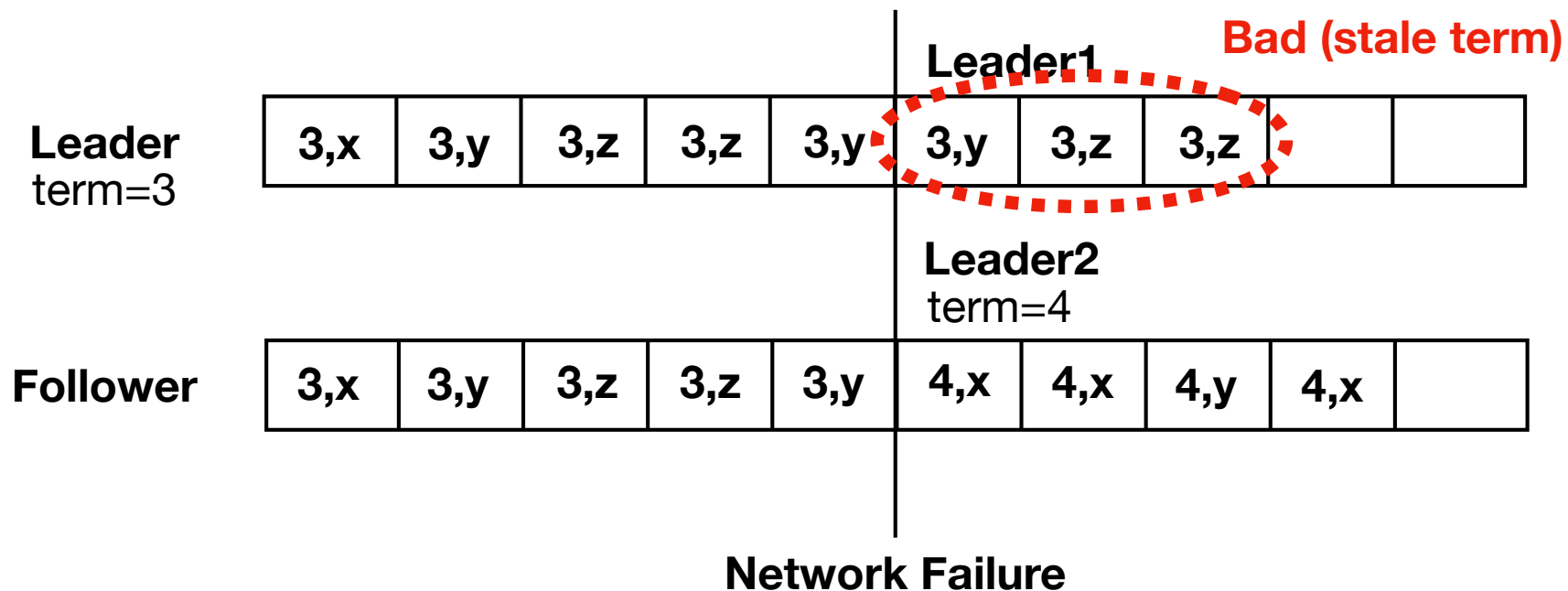
# Solution: Terms

- There is a monotonically increasing term number
- Incremented on every leader election
- The term is recorded in the log



# Solution: Terms

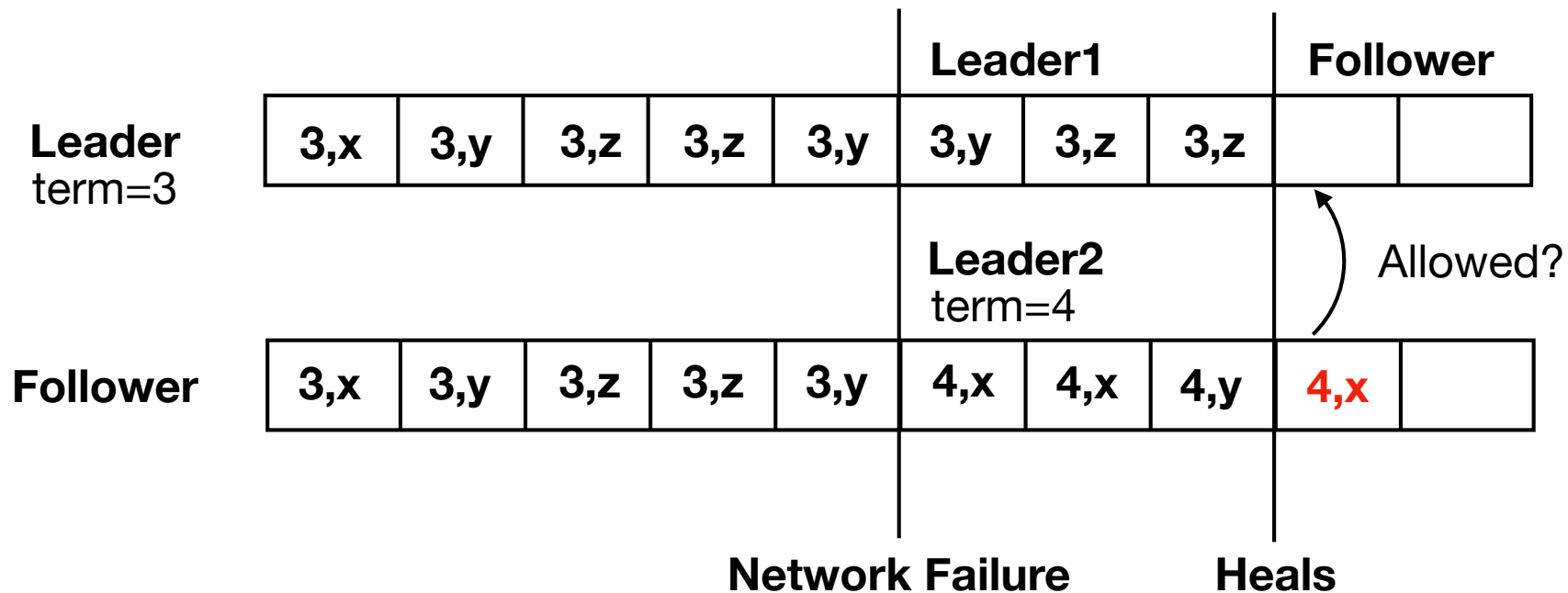
- The term can be used to detect bad log entries
- Remember--every log entry must be identical





# Log Matching

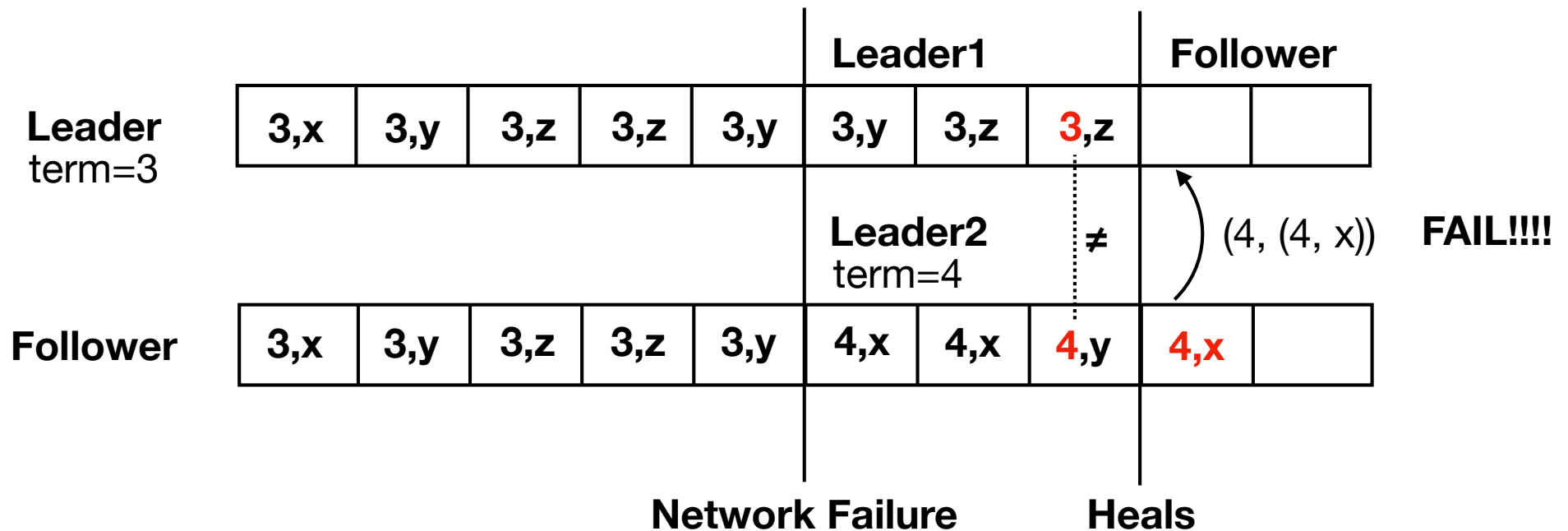
- The log is never allowed to have gaps, but consider this tricky scenario after healing



- Can the new leader just append onto old leader?

# Log Matching

- Appends always require the term number of the preceding log entry to be given

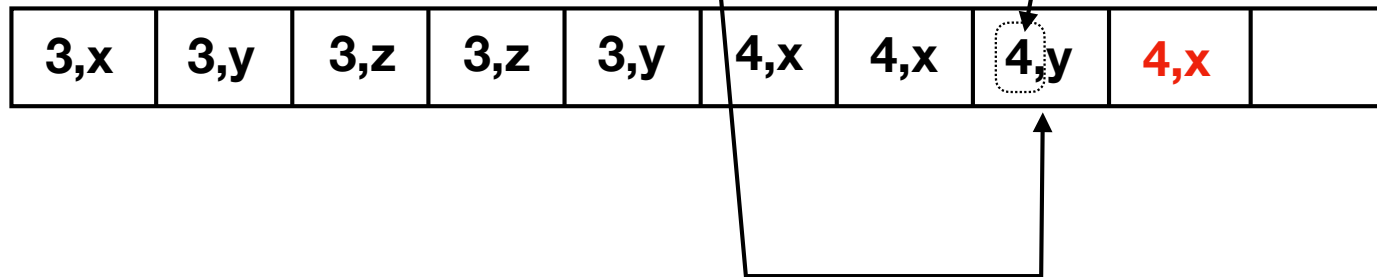


- This enforces "log continuity" (log rejects appends that don't connect properly)

# Solving Log Matching

- Log appends must link to prior entry

```
def append_entry(log, prev_index, prev_term, entry):  
    ...
```



- Can't append to the log unless you provide correct information about the previous entry

# Log Implementation

- Log Appends are encapsulated into a single operation called "AppendEntries"
- It's a list append with conditions
  - Idempotent (repeated attempts ok)
  - No gaps/holes
  - Matching of terms
- Allows multiple entries to be appended

# Project 4

- Implement the Raft log as a stand-alone object
- Write tests to verify its behavior
- Should be usable on its own--no dependency on the network or any other part of Raft
- If appending to the log is broken, then everything is broken.

Part 5

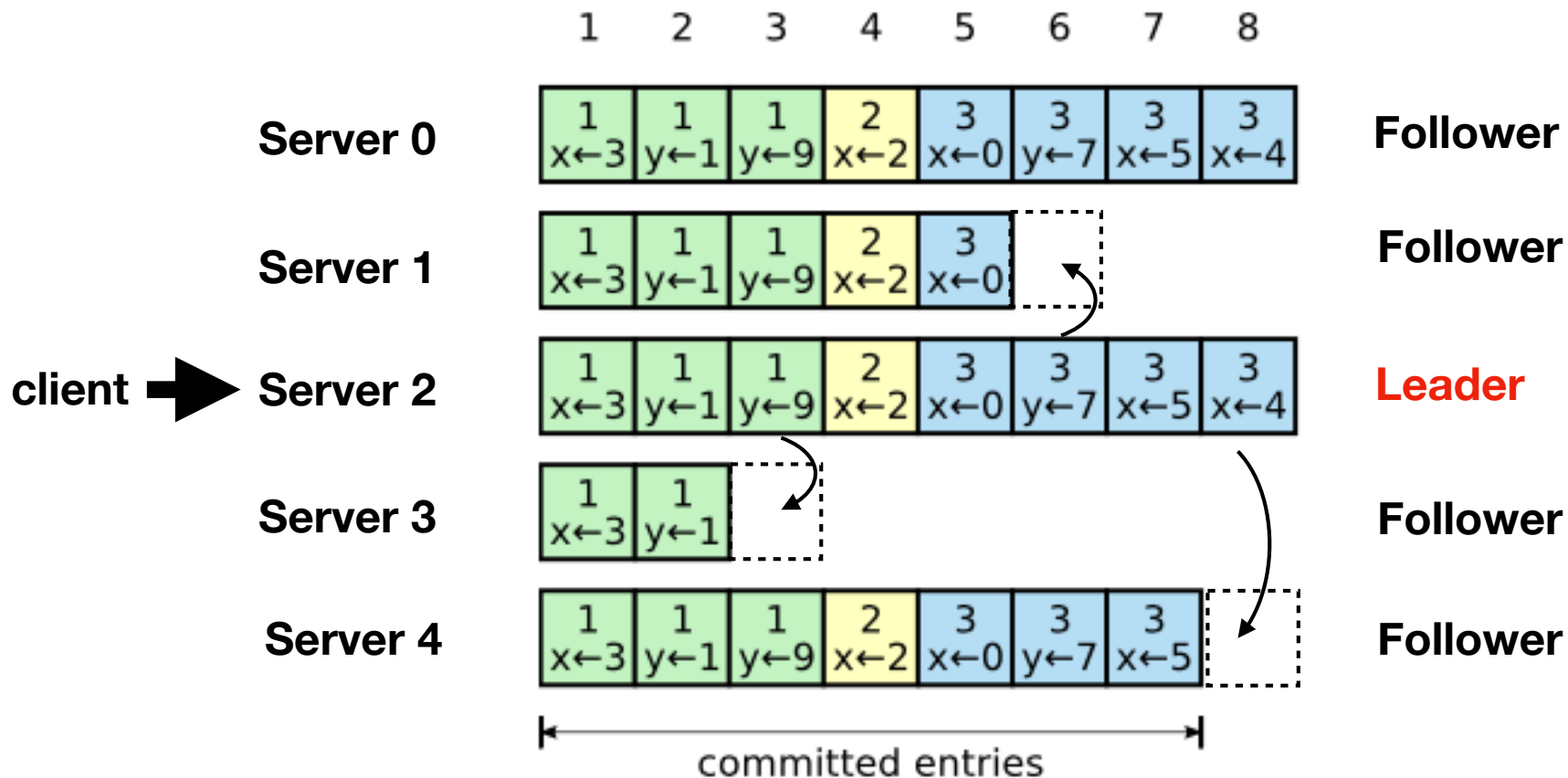
# Log Replication

# Raft Leadership

- In Raft, one server is designated the "leader"
- All client transactions are with the leader
- The role of the leader is to update followers

# Follower Update

- The leader works to bring all followers up to date.



- It does this by sending messages (AppendEntries)



# Comments

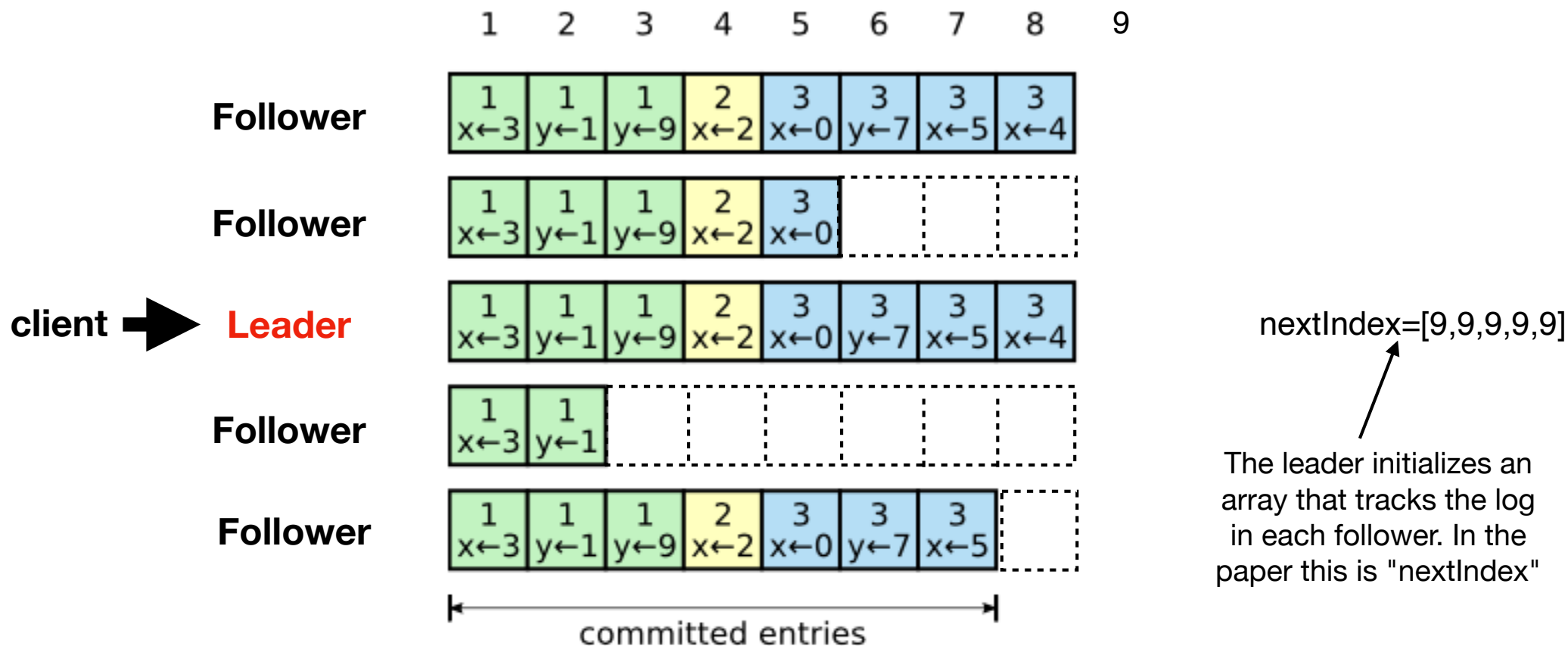
- Followers are passive.
- They receive messages from the leader and respond to the leader. NEVER with each other.
- If there is no leader, a new leader is elected from one of the followers (later).

# Problem

- How does a newly designated leader know anything about the state of the followers given that there has been no prior communication?
- A new leader knows nothing.
- Why it matters: The leader has to issue log updates to followers. But, what is updated?

# Initial Leader State

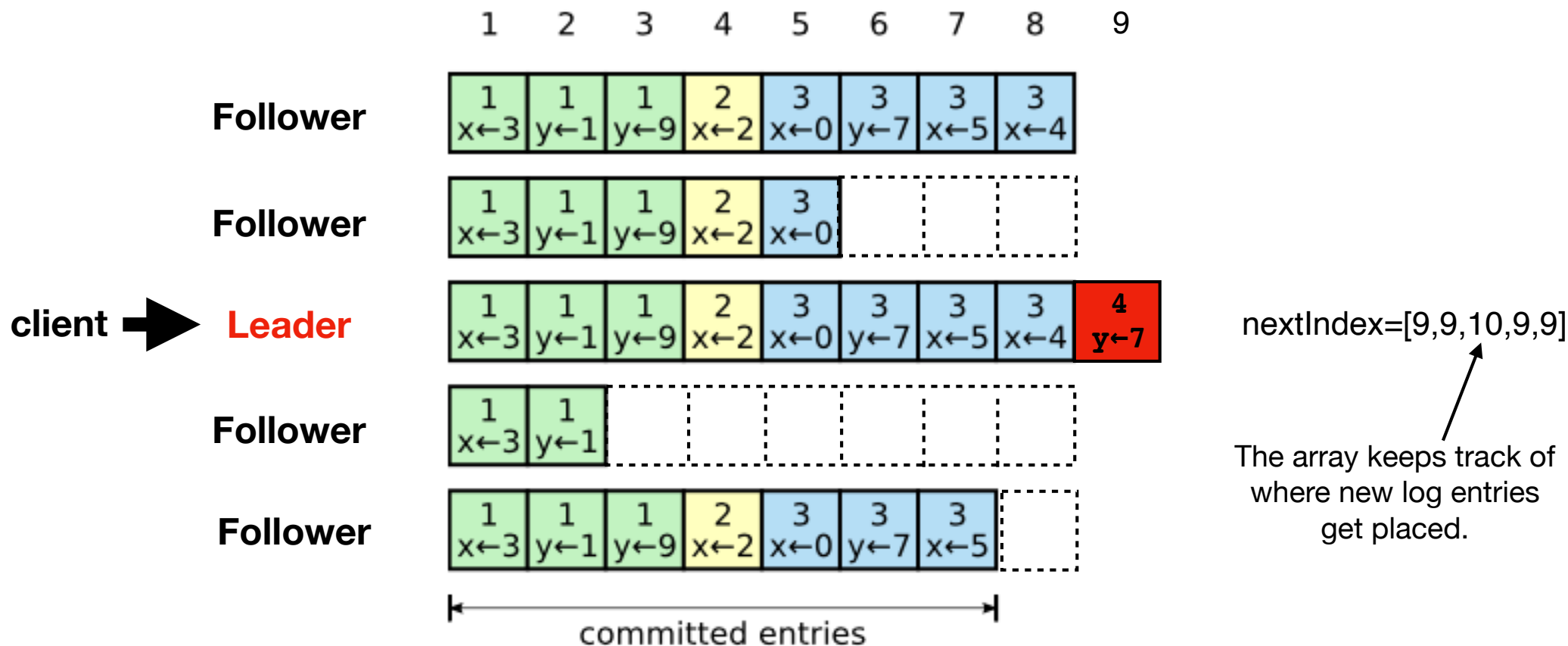
- Leaders start by assuming that all followers have the entire log (same information as leader)



- May or may not be true (leader doesn't know)

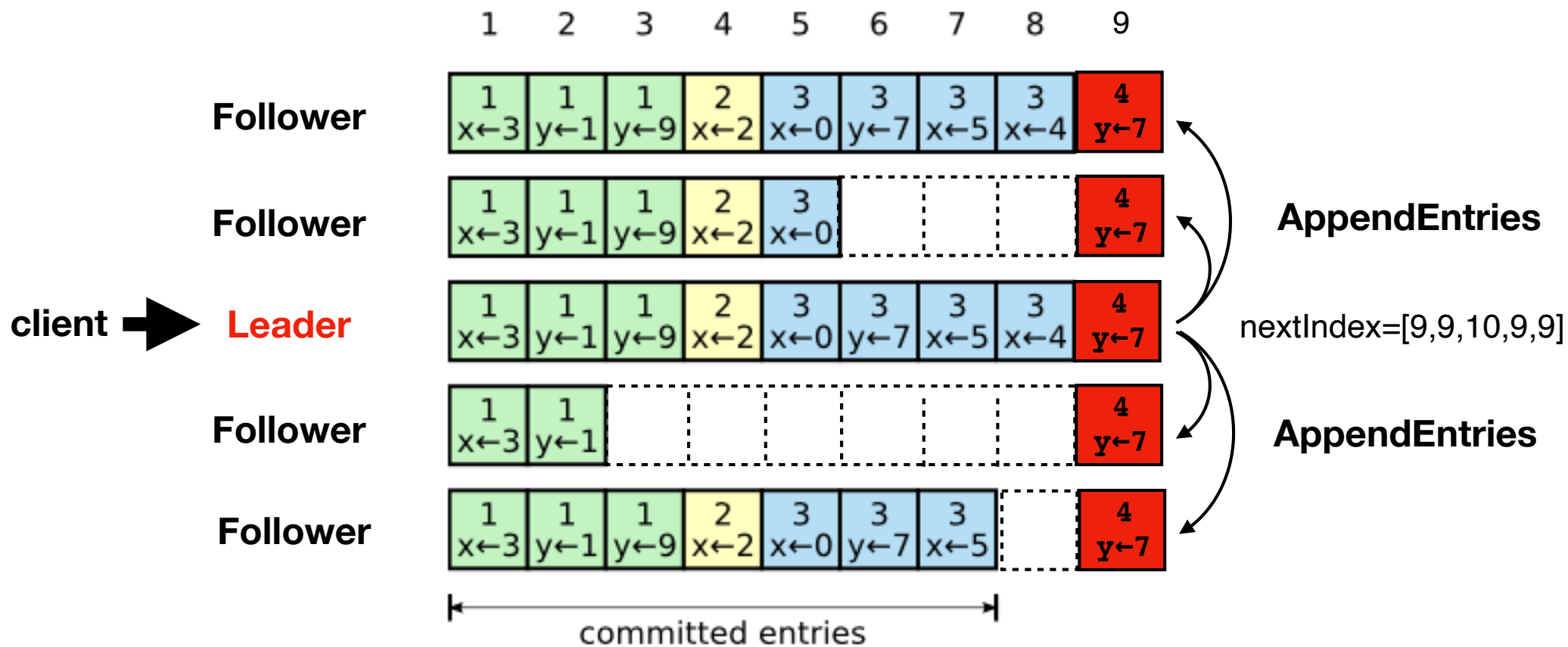
# New Appends

- New entries go on end of leader log



# Append Replication

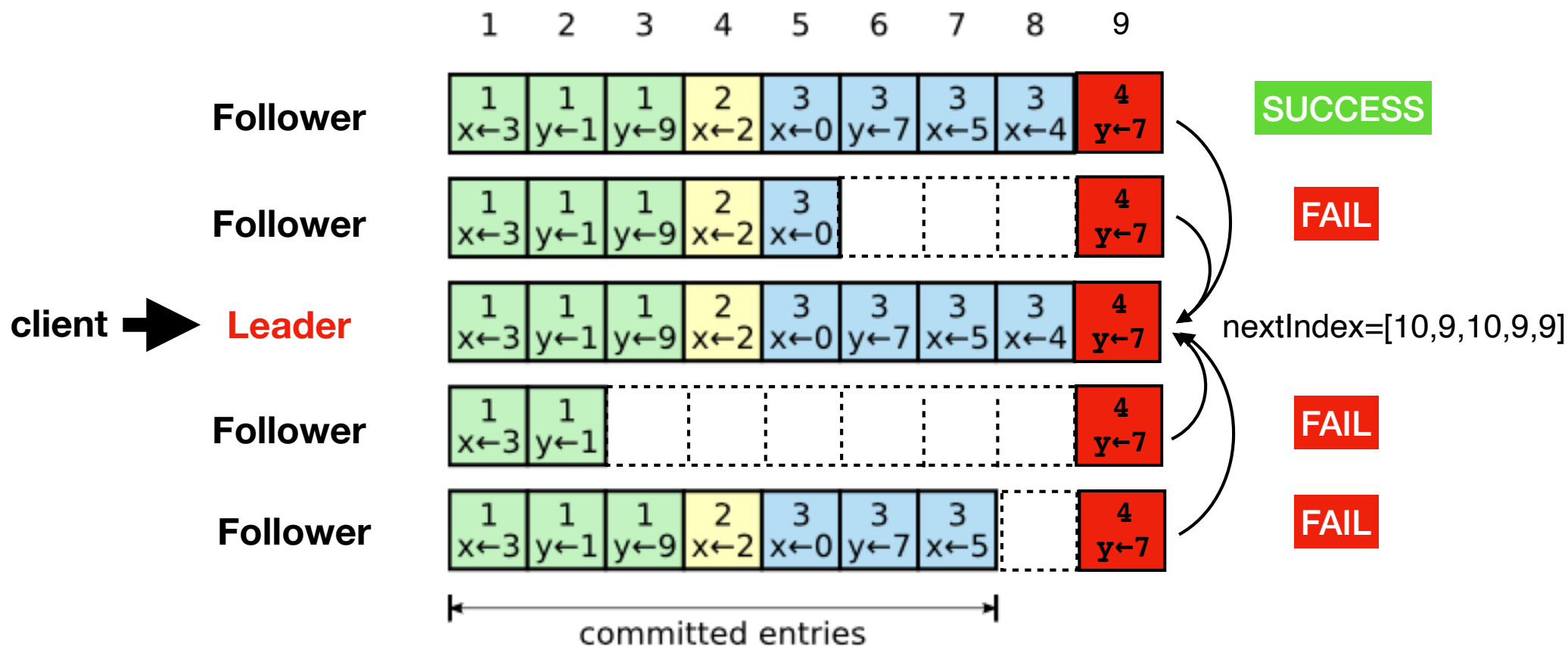
- Leader replicates to the followers



- This is a network message

# Response Handling

- AppendEntries will succeed or fail (no gaps)



- This is reported back (AppendEntriesResponse)

# Backtracking

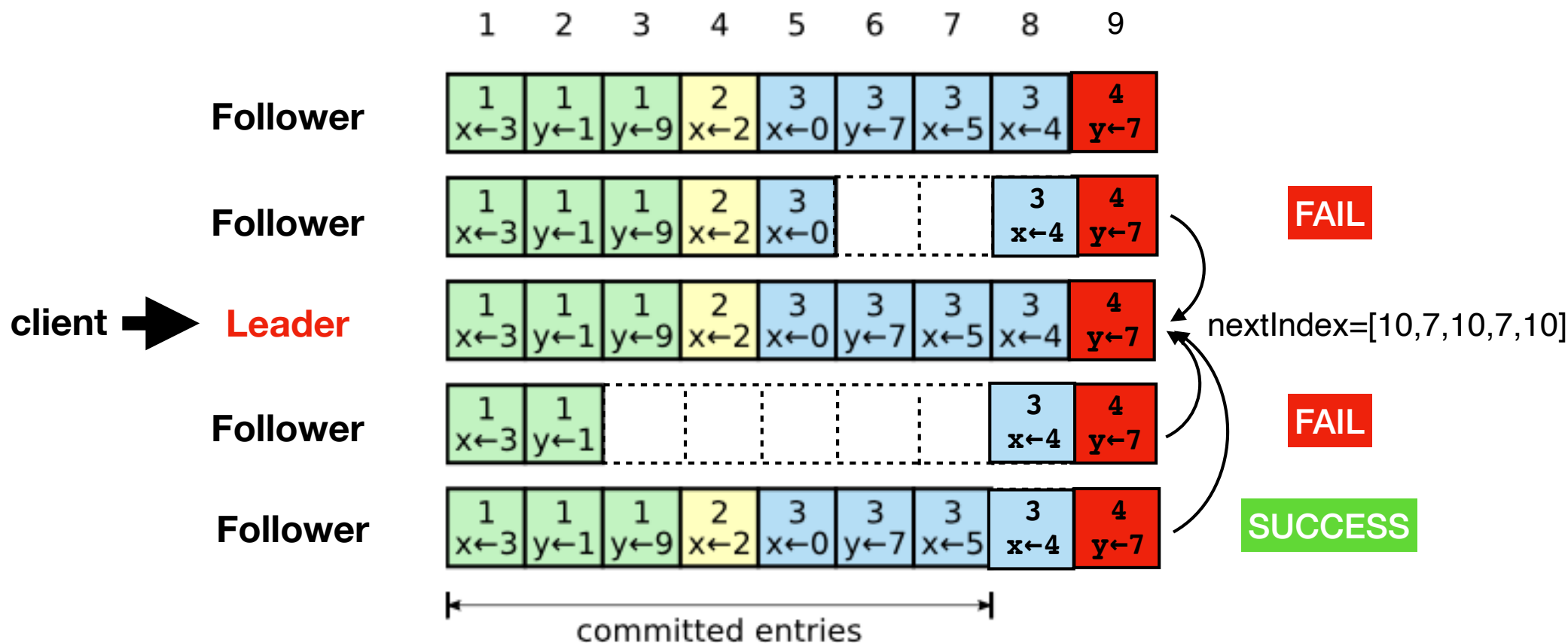
- Leader retries failures with earlier log entries



- Again, followers report back with success

# Backtracking

- This process repeats itself

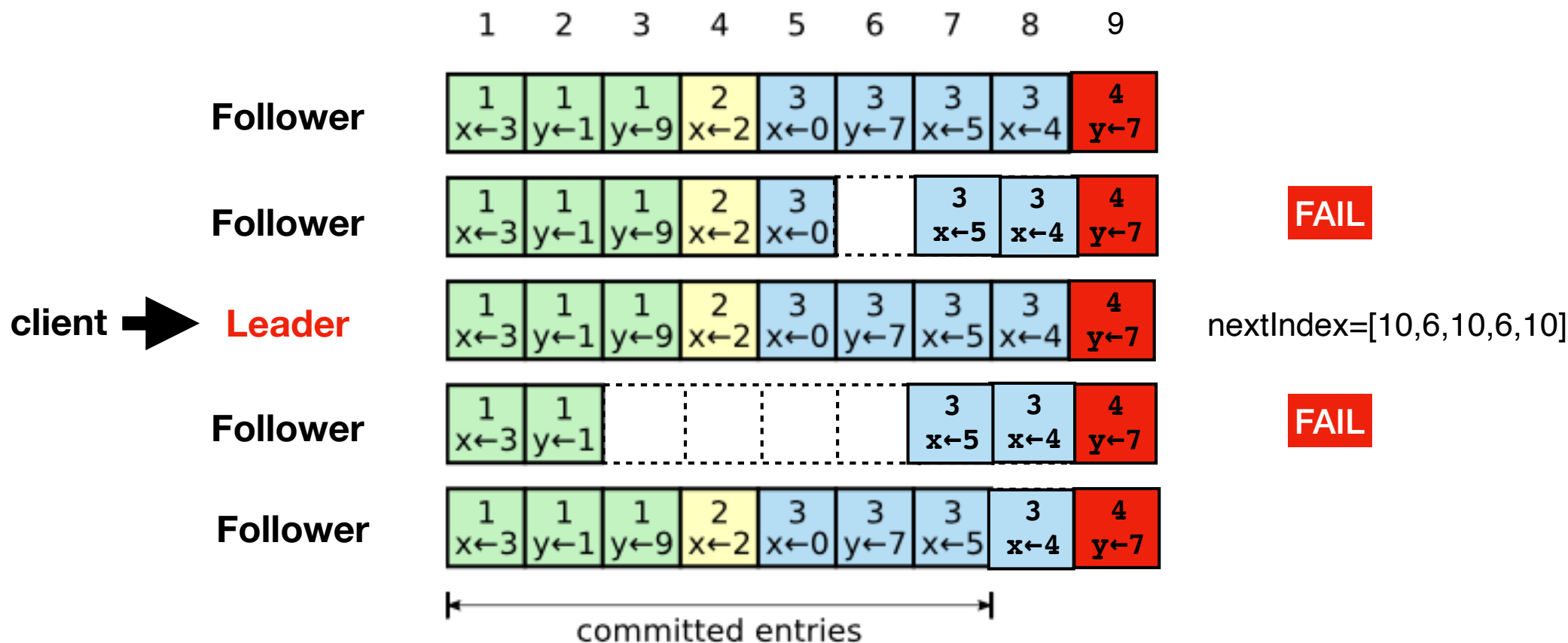


- Leader keeps working backwards until success



# Backtracking

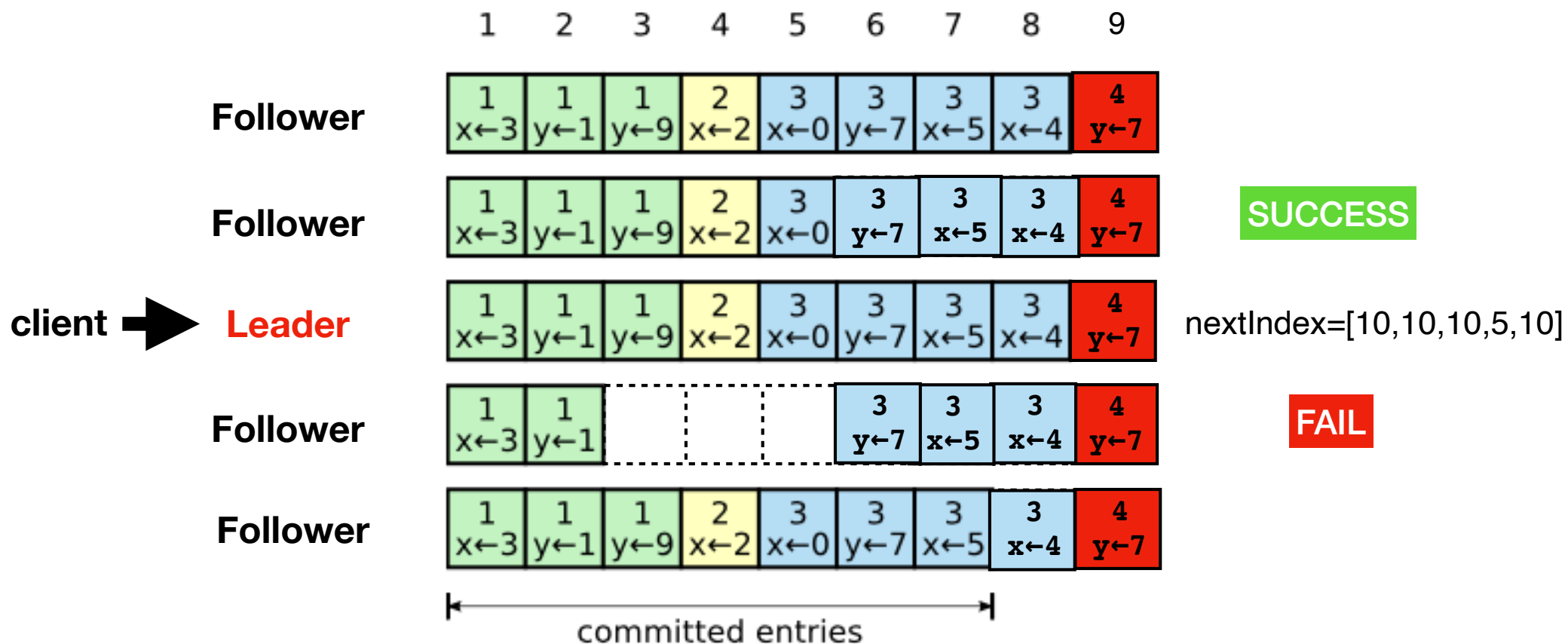
- This process repeats itself



- Leader keeps working backwards until success

# Backtracking

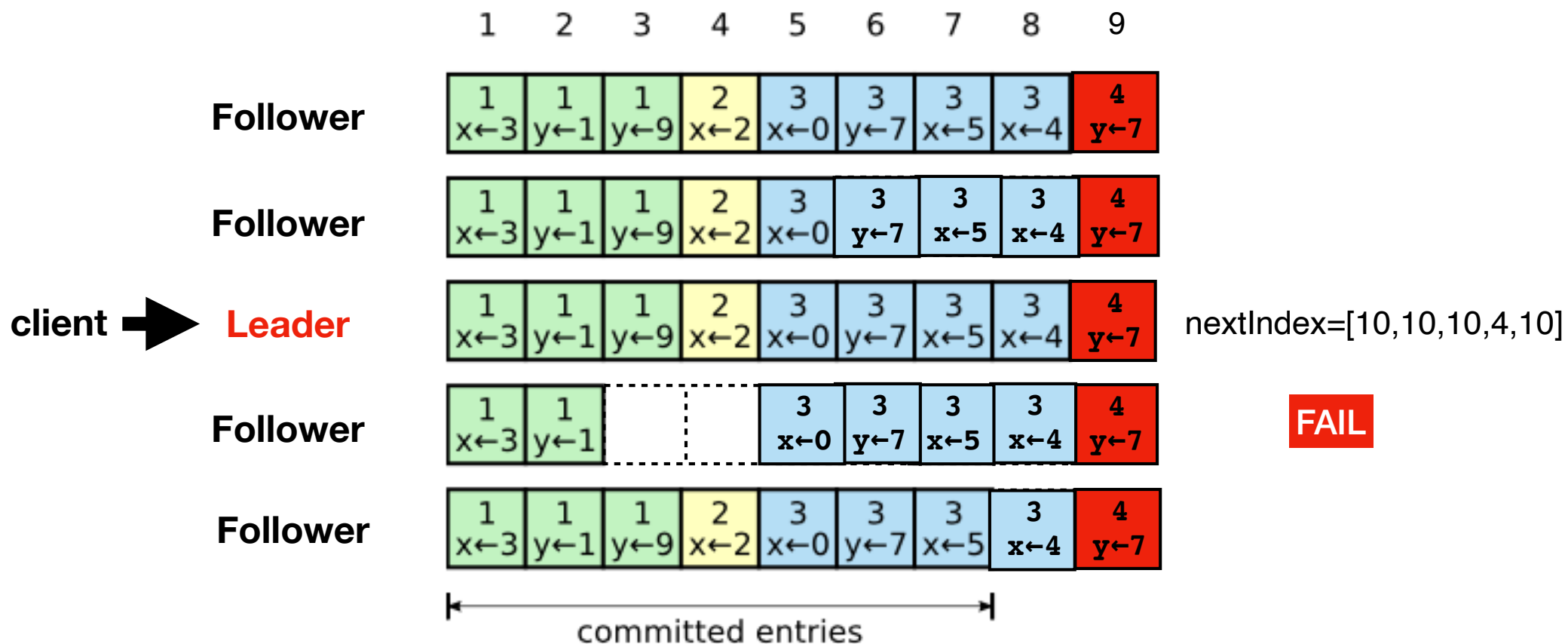
- This process repeats itself



- Leader keeps working backwards until success

# Backtracking

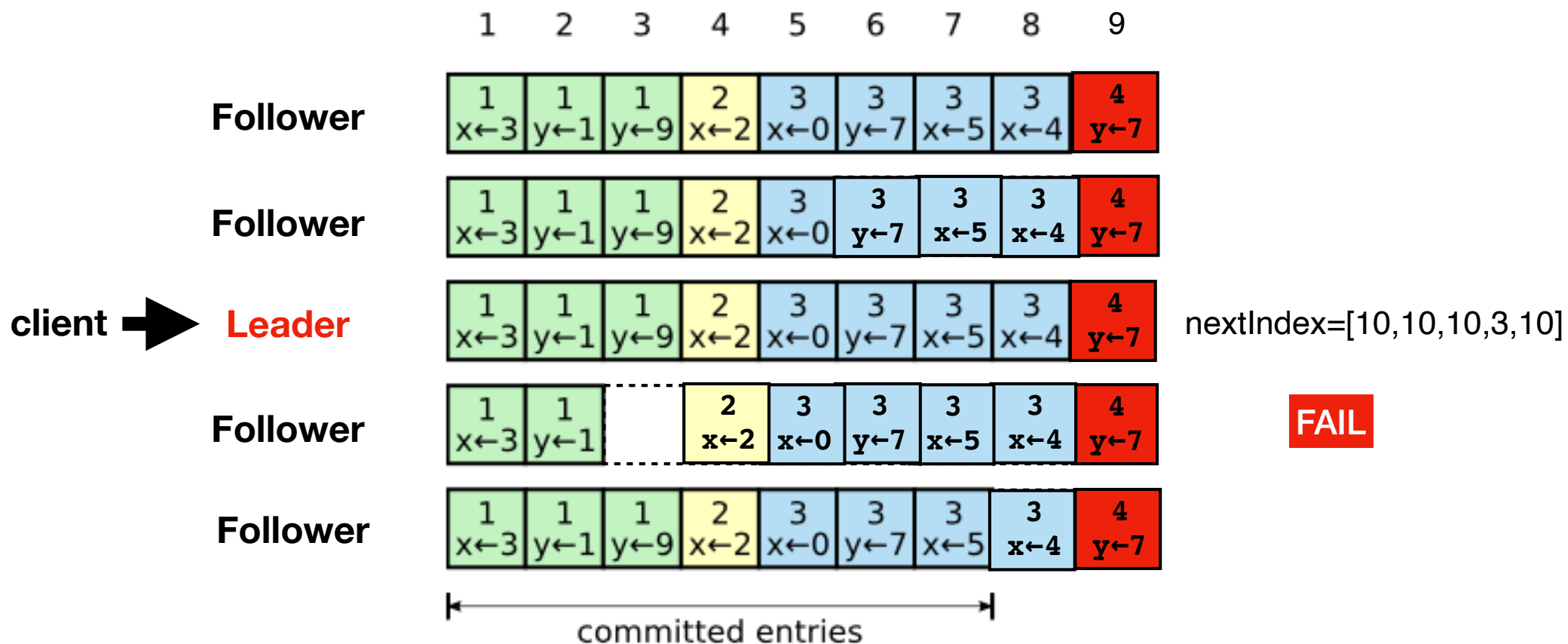
- This process repeats itself



- Leader keeps working backwards until success

# Backtracking

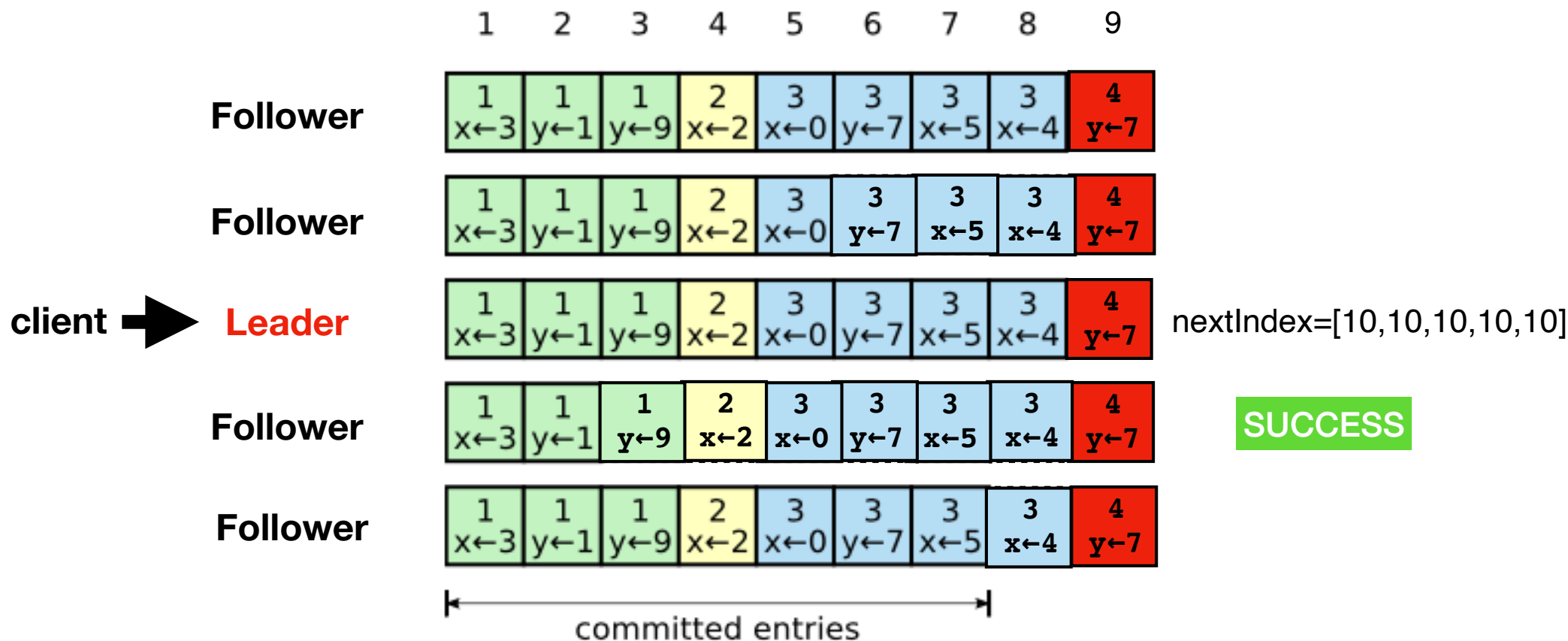
- This process repeats itself



- Leader keeps working backwards until success

# Backtracking

- This process repeats itself



- Leader keeps working backwards until success

# Commentary

- There is some book-keeping
- The leader needs to track the state of each follower independently
- But, follower states are initially unknown so it has to be determined from message responses
- Leader still has to keep the other followers up-to-date during this process (there could be new entries arriving. A lagging follower isn't allowed to block progress on everyone)

# Project 5

- Implement log replication
- Question: Can it be tested? How?
- Comment: This is a difficult part of the project. Involves multiple servers and complex interactions. Take it slow.

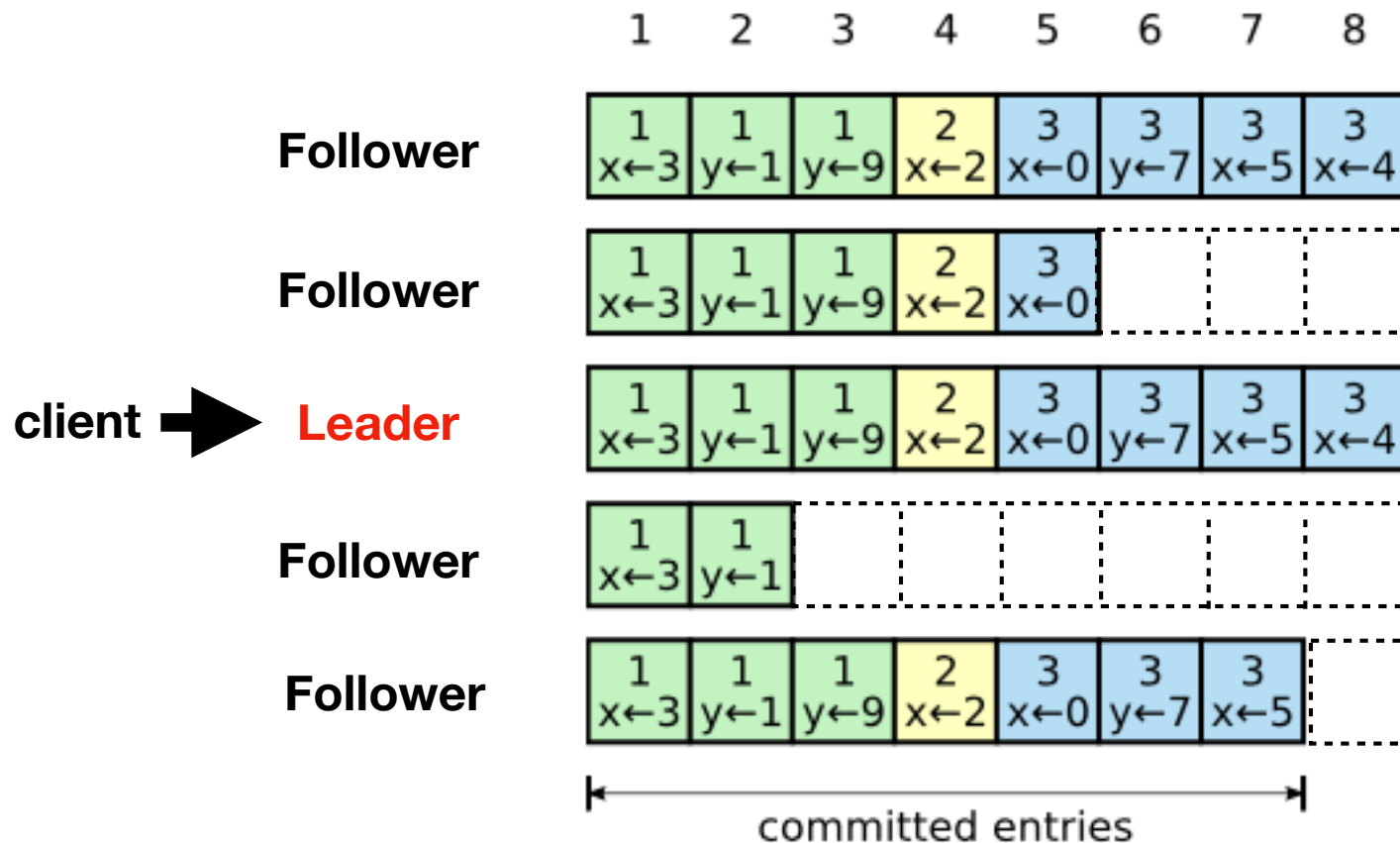
**Part 6**

# Consensus



# Consensus

- The goal is to reach consensus. Log entries must be replicated on a majority of servers



- When consensus is reached -- entries committed

# Applying the Log

- When log entries are committed, it means that a server can "apply the log" to its state machine
- Basically, it means that the application can "do the thing" the client actually requested

## Client

```
kv.set('foo', 42)
```

## Key-Value Server

```
data = { }
```

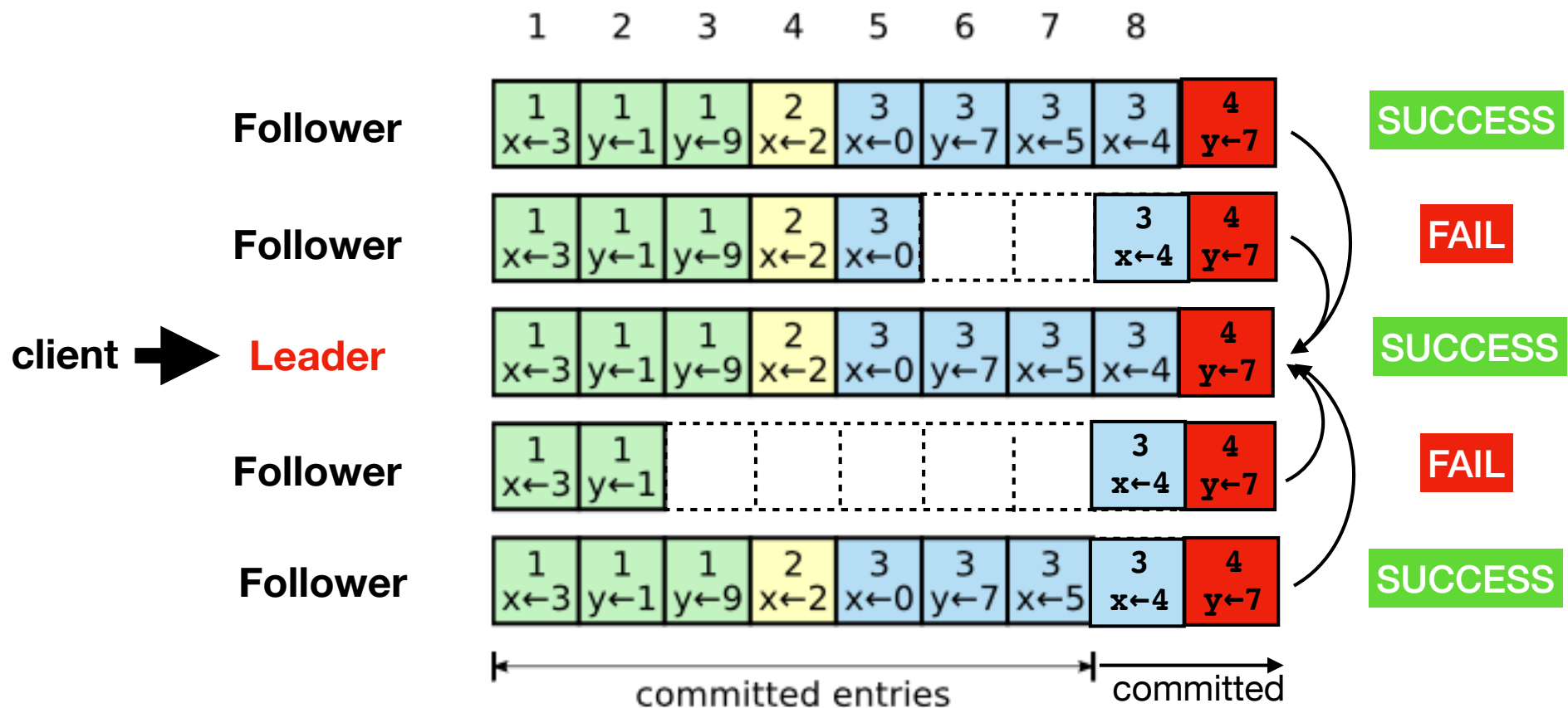
```
kv.set('foo', 42) —————→ def set(key, value):  
                                entry = ('set', key, value)  
                                append_entry(entry)  
                                while not committed:  
                                    wait  
                                # DO IT  
                                data[key] = value
```

# Consensus on Leader

- The leader communicates with followers
- Followers inform the leader about their log state
- Note: There is an error in Figure 2 (missing info)
- From this, the leader can determine consensus

# Consensus on Leader

- The leader updates its commit index by watching responses to AppendEntries



- Need success on a majority (not all)

# Problem: Followers

- Followers don't communicate
- So, how do they know when consensus has been reached?
- Solution: The leader tells them

# Consensus on Followers

- The leader commit index is always sent



- Followers watch the index

# Project 6

- Implement consensus
- Figure out how consensus integrates with the higher-level application (i.e., key-value store)

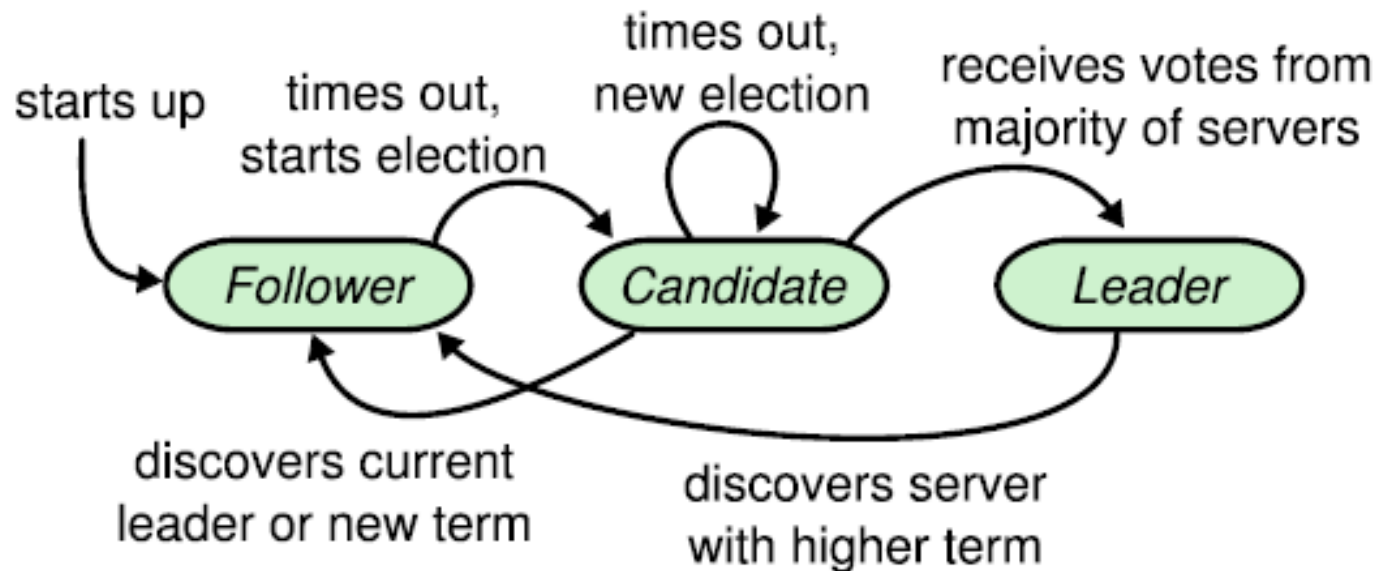
**Part 7**

# Leader Election



# Raft Operational Roles

- Servers in Raft operate in different roles



- A major complexity concerns the transitions between the different roles

# The Roles

- Follower: Passively listen for messages
- Candidate: Has called a leader election. Awaiting for votes from other servers.
- Leader: Sends AppendEntries updates. Interacts with clients.

# Terms

- Raft divides time into terms
- It is an ever-increasing integer value
- Only increased by candidates
- In any given term, there is only one leader
- There might be no leader (two candidates for a given term with a split-vote)

# Terms and Messages

- All network messages include the term
- This is used as a leader-discovery mechanism
- Some universal rules:
  1. Receiving any message with a higher term than yourself makes you a follower
  2. Discard all received messages with a lower term than yourself (out of date)

# Time Management

- The leader sends AppendEntries messages to all followers on a periodic timer (heartbeat)
- Followers wait for leader messages on a slightly randomized timeout
- If no leader message, follower calls an election (sends a RequestVote message to all servers)

# Rules For Voting

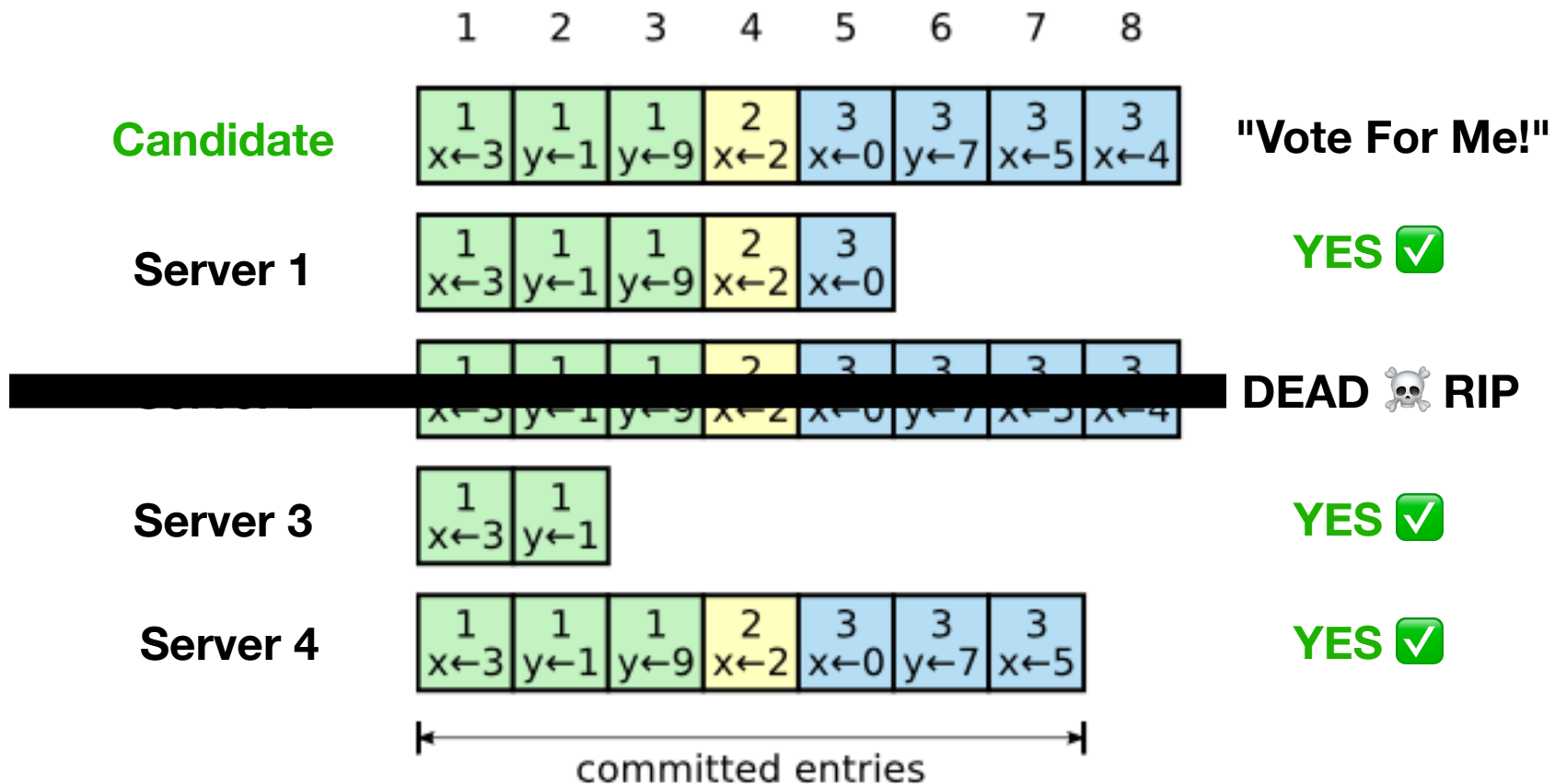
- A server may only vote for one candidate in a given term (subsequent requests denied)
- This is for dealing with split votes
- It's possible that two followers could promote to candidate at the same time. They'd have the same term number, but might not be able to get a quorum.

# Rules For Voting

- A server may not vote for a candidate if the candidate's log is not as up-to-date as oneself.
- Required reading: Section 5.4.1 of Raft Paper
  - Grant vote if last entry in candidate's log has a greater term than myself.
  - If last log entry has same term as myself, grant vote if candidate's log is longer.
- This is subtle. Great care required.

# Voting Example

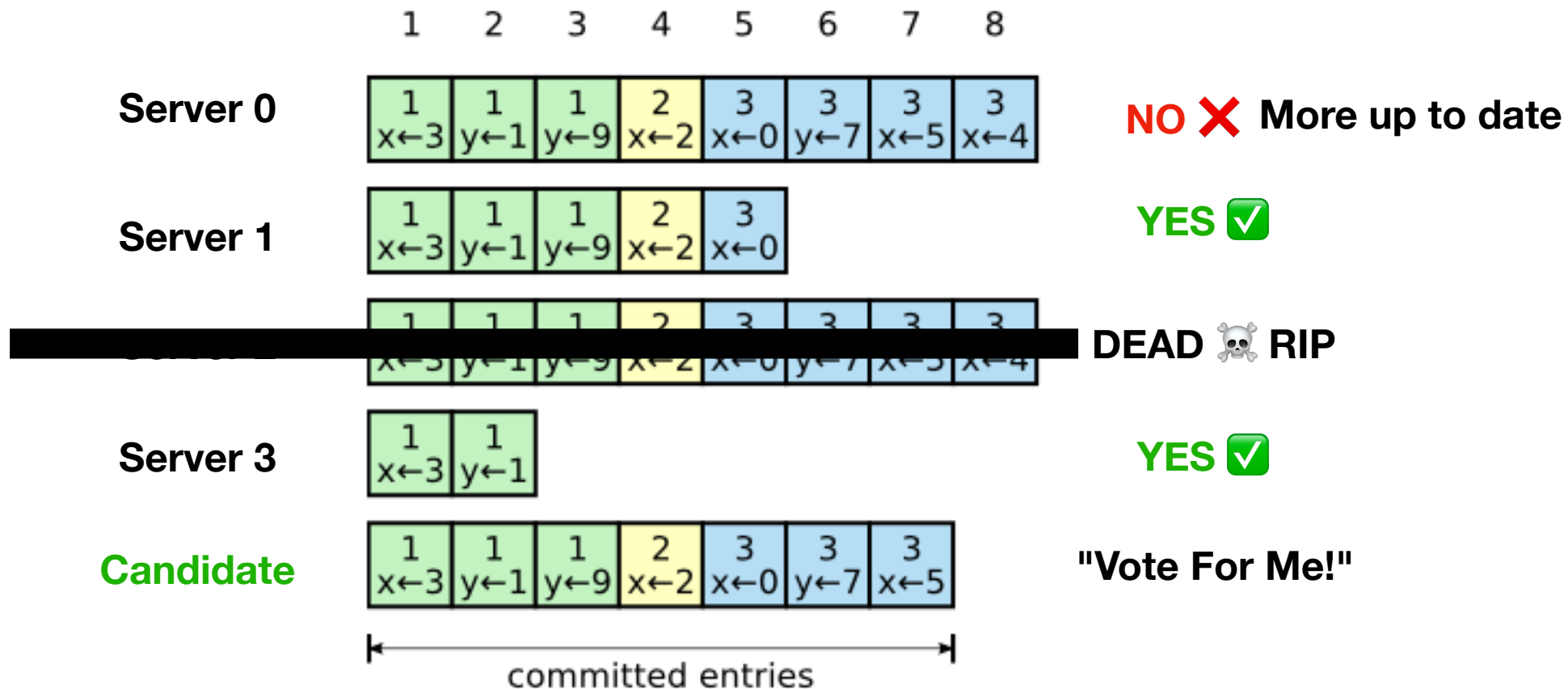
- Servers will vote for candidate with longer log/newer term





# Voting Example

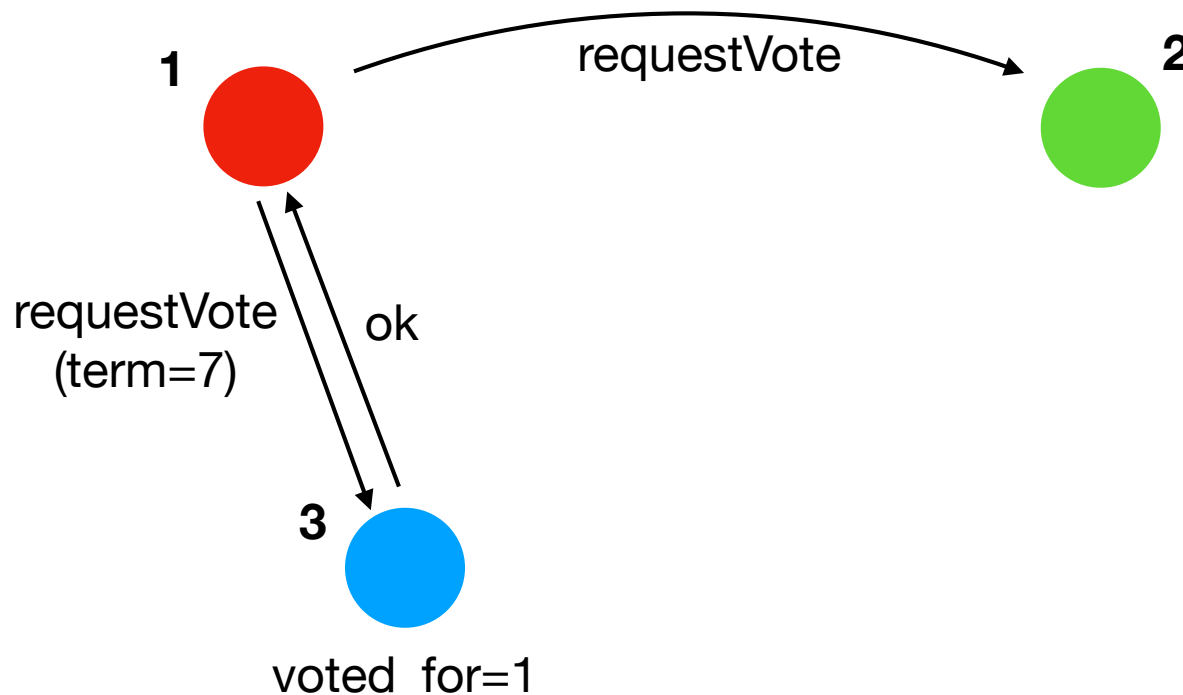
- But a server with a shorter log still might win



- The winner will always have all committed entries

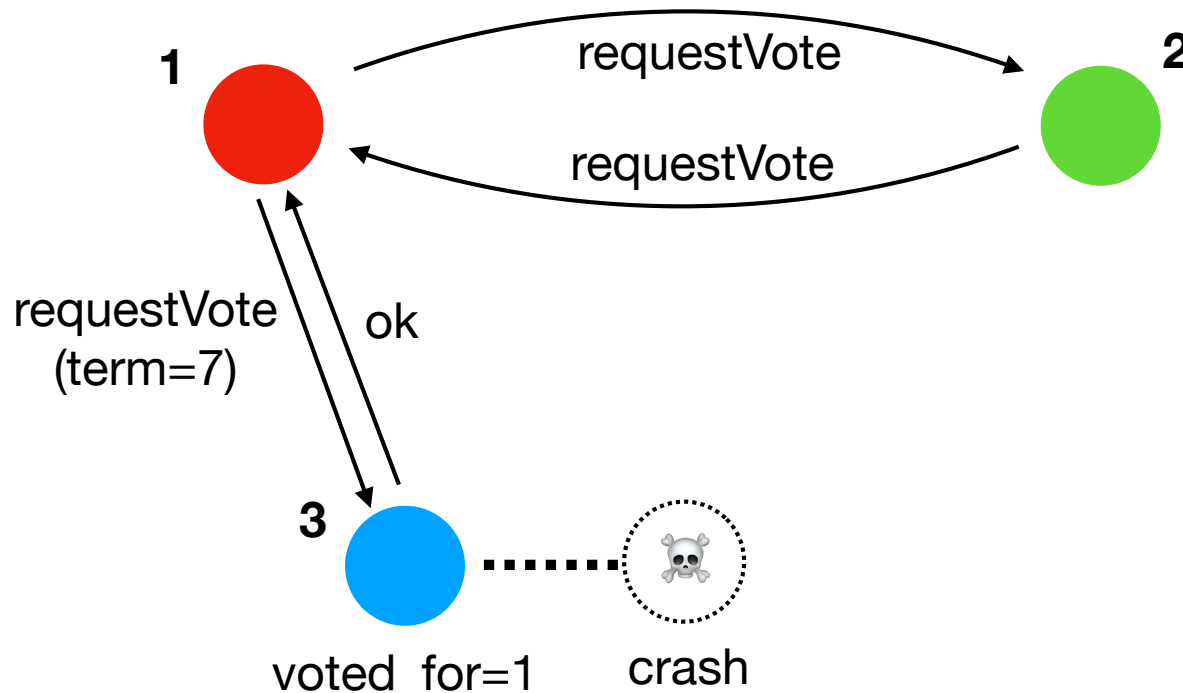
# Non-volatile Storage

- Certain parts of Raft require non-volatile storage---for example, voting.
- A crashed/restarted server never votes twice!



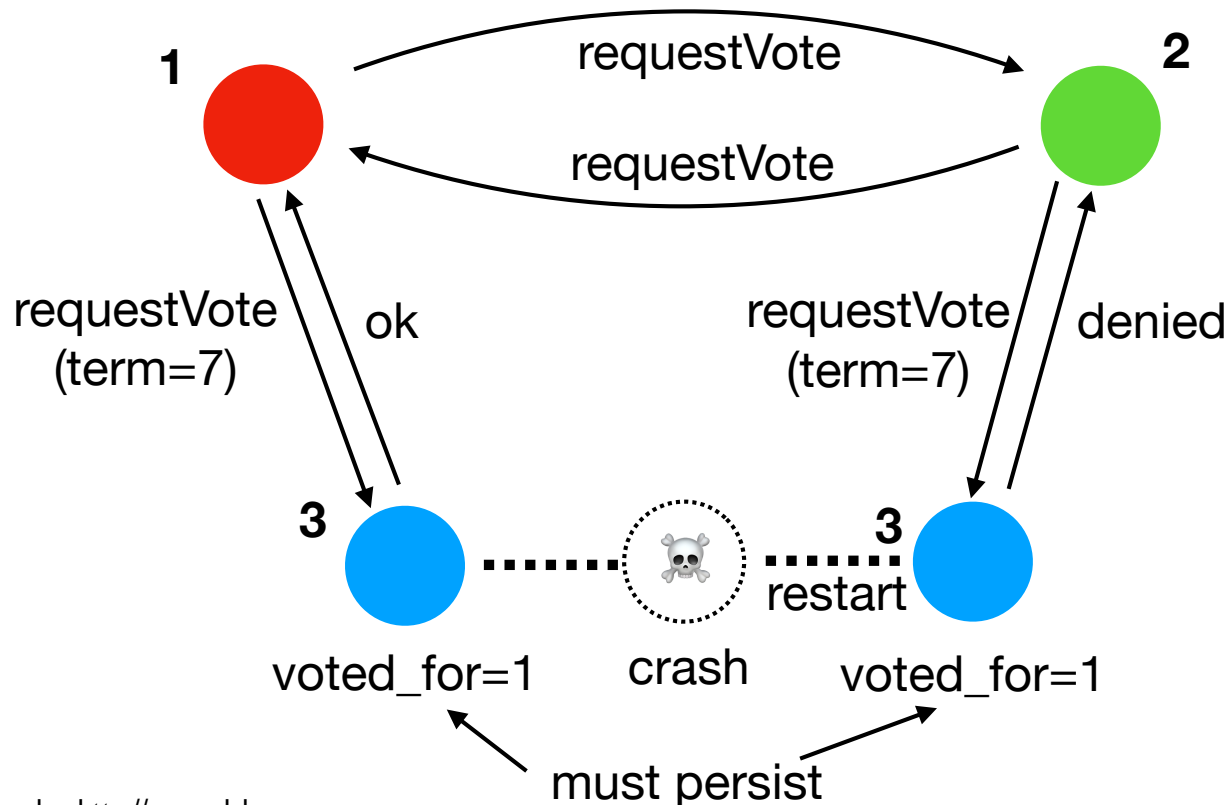
# Non-volatile Storage

- Certain parts of Raft require non-volatile storage---for example, voting.
- A crashed/restarted server never votes twice!



# Non-volatile Storage

- Certain parts of Raft require non-volatile storage---for example, voting.
- A crashed/restarted server never votes twice!



# Project 7

- Implement leader election
- Can it be done in a way that can be tested?

**Part 8**

# What Can Go Wrong?

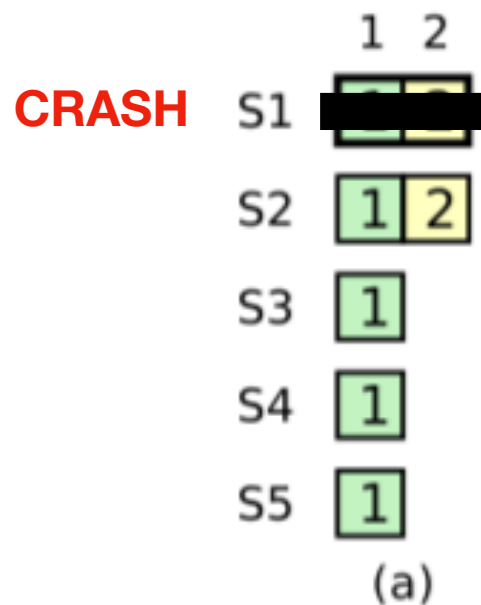
# "Figure 8"

- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



# "Figure 8"

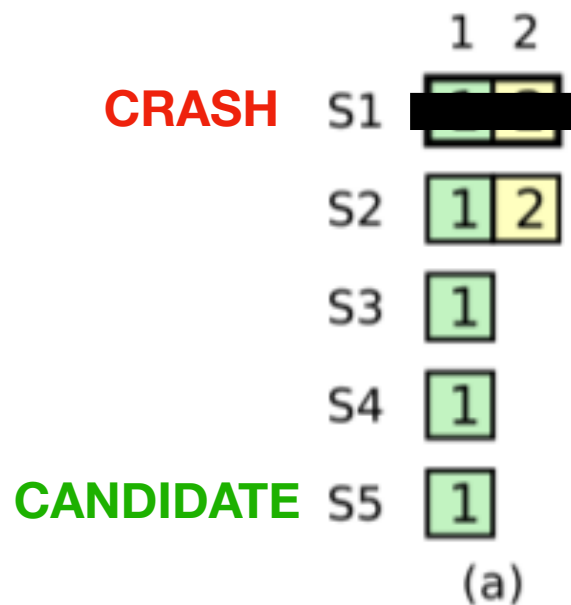
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority





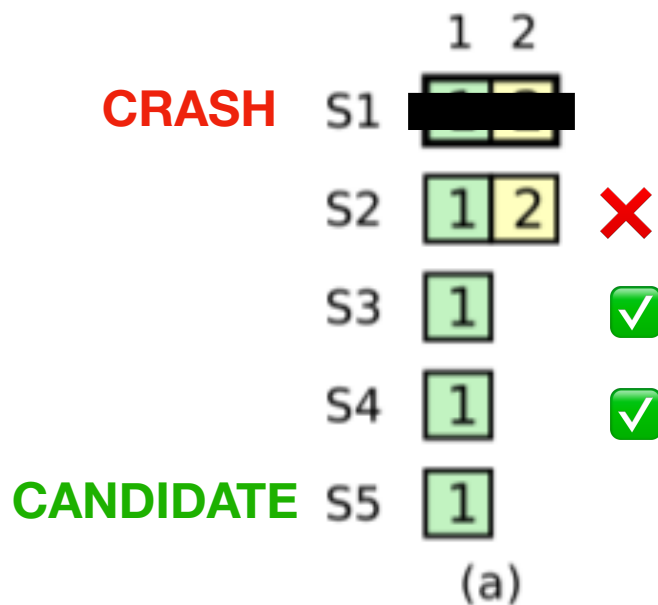
# "Figure 8"

- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



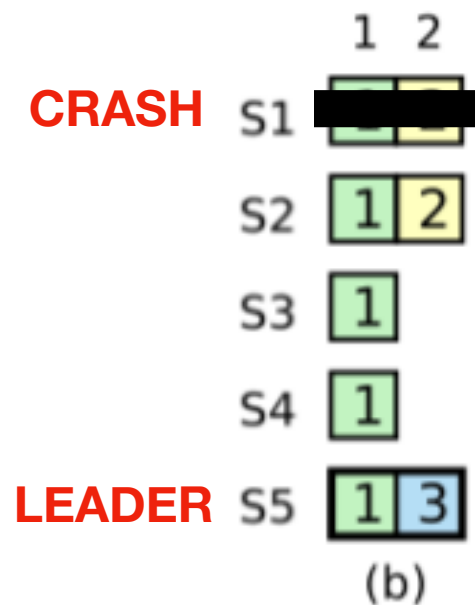
# "Figure 8"

- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



# "Figure 8"

- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



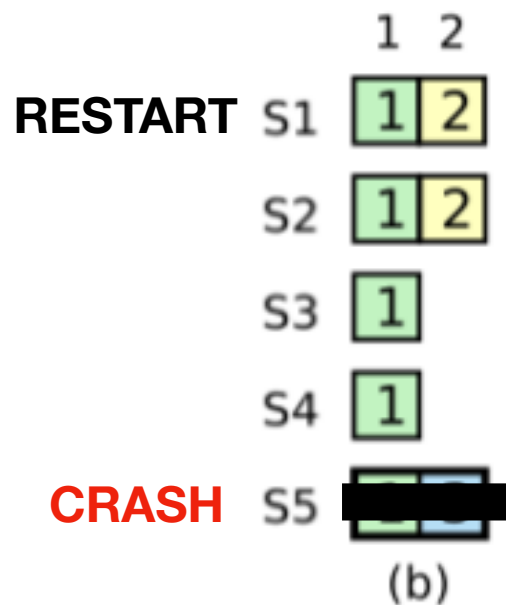
# "Figure 8"

- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



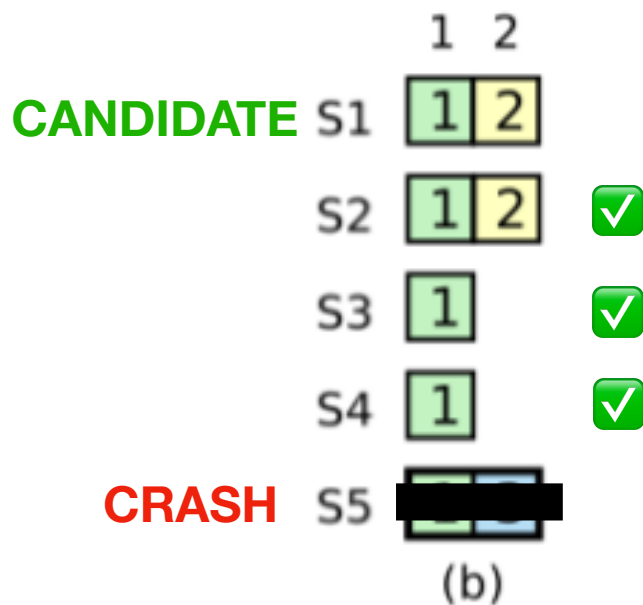
# "Figure 8"

- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



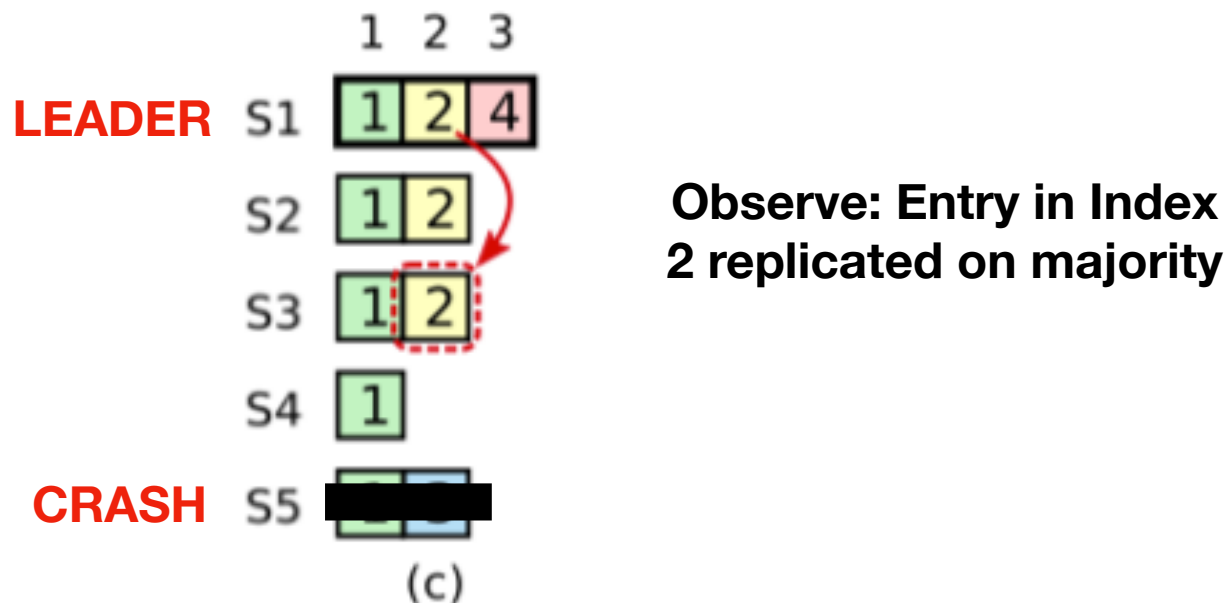
# "Figure 8"

- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



# "Figure 8"

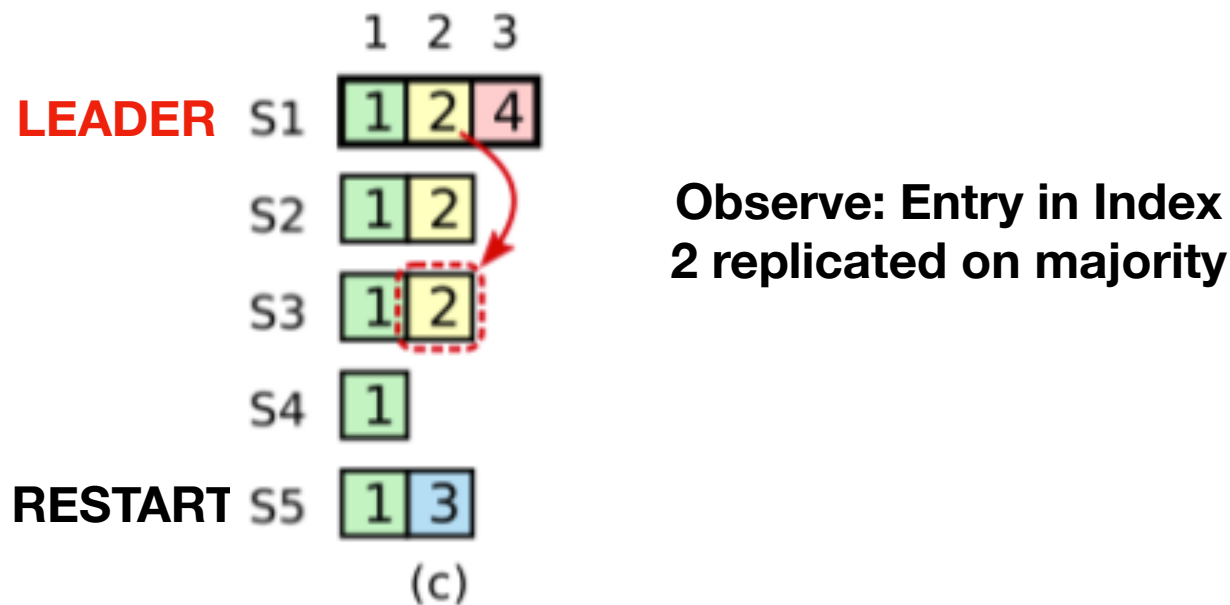
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



- QUESTION:** Can the leader call index 2 "committed"?

# "Figure 8"

- A newly elected leader can never commit entries from the previous leader--even if replicated on majority

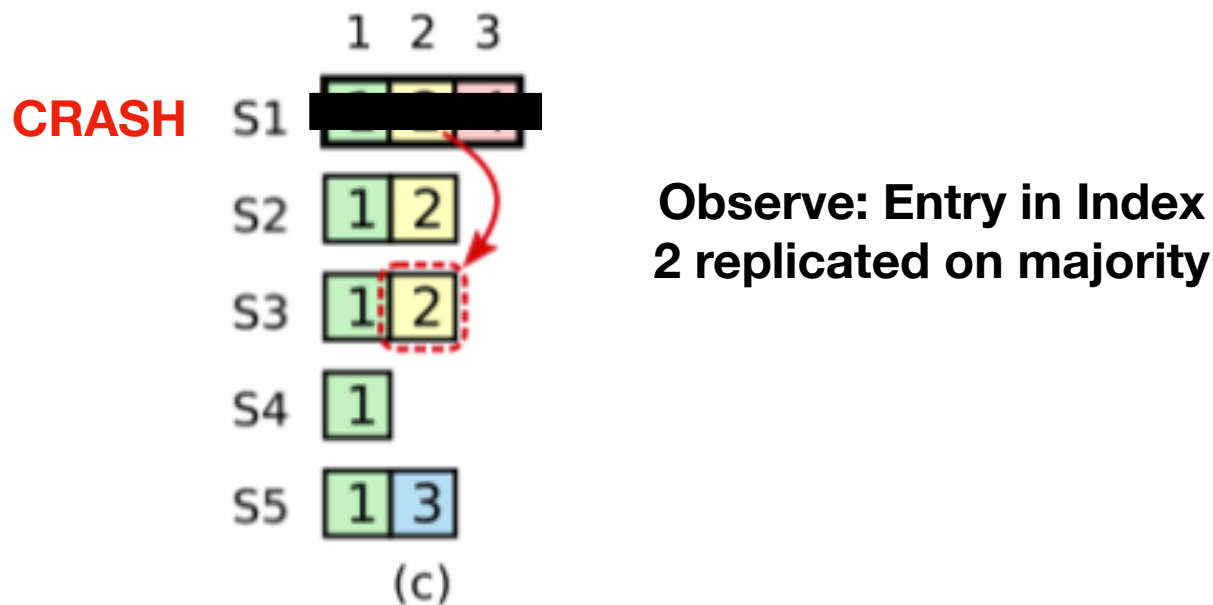


- QUESTION: Can the leader call index 2 "committed"?



# "Figure 8"

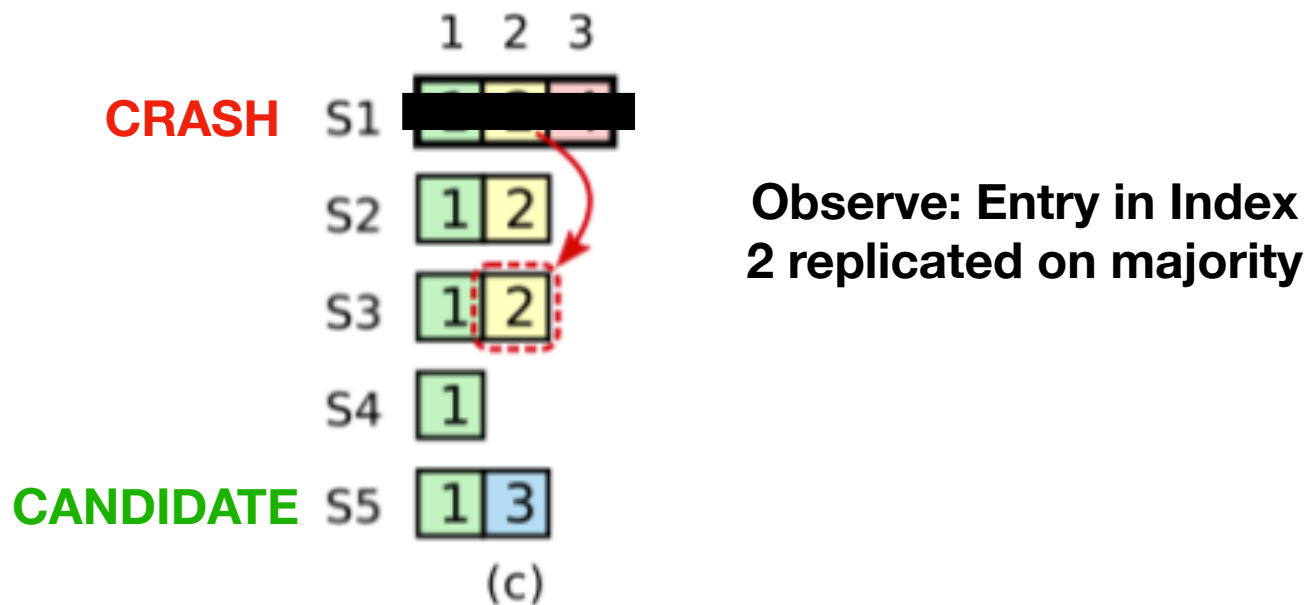
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



- QUESTION: Can the leader call index 2 "committed"?

# "Figure 8"

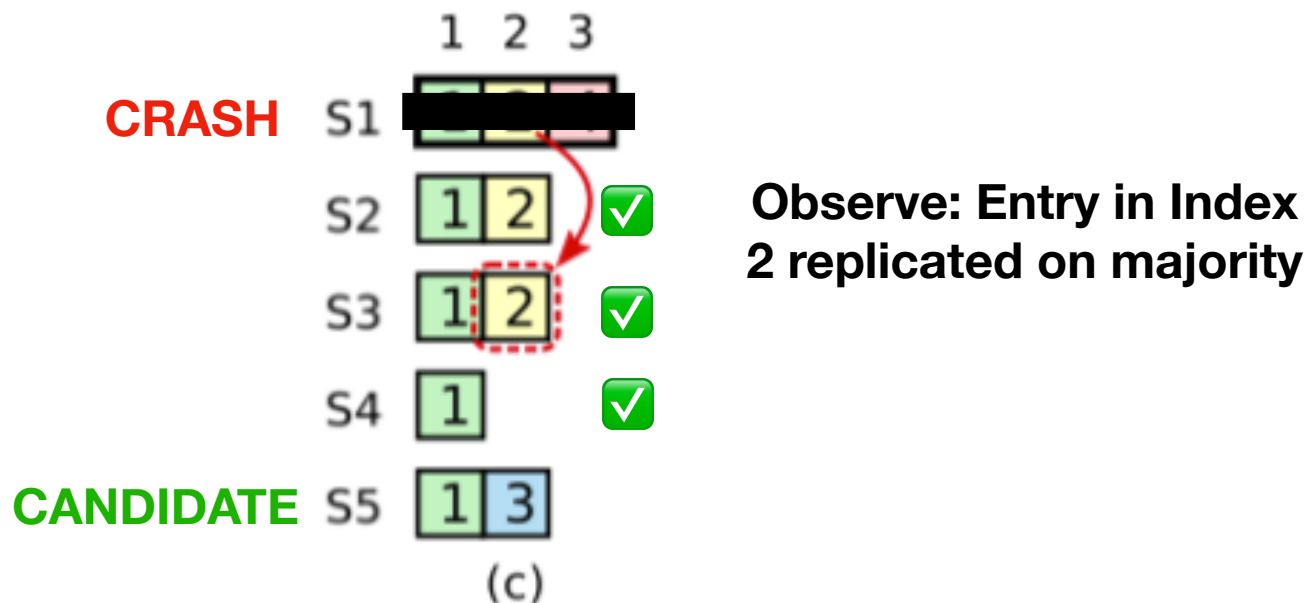
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



- QUESTION: Can the leader call index 2 "committed"?

# "Figure 8"

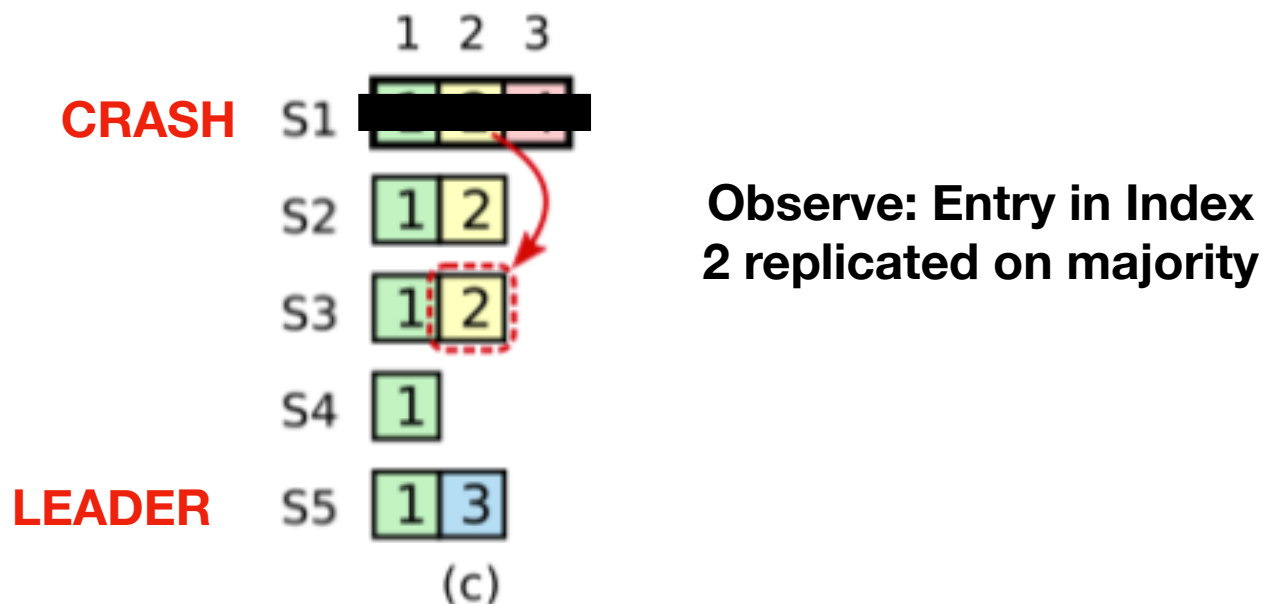
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



- QUESTION: Can the leader call index 2 "committed"?

# "Figure 8"

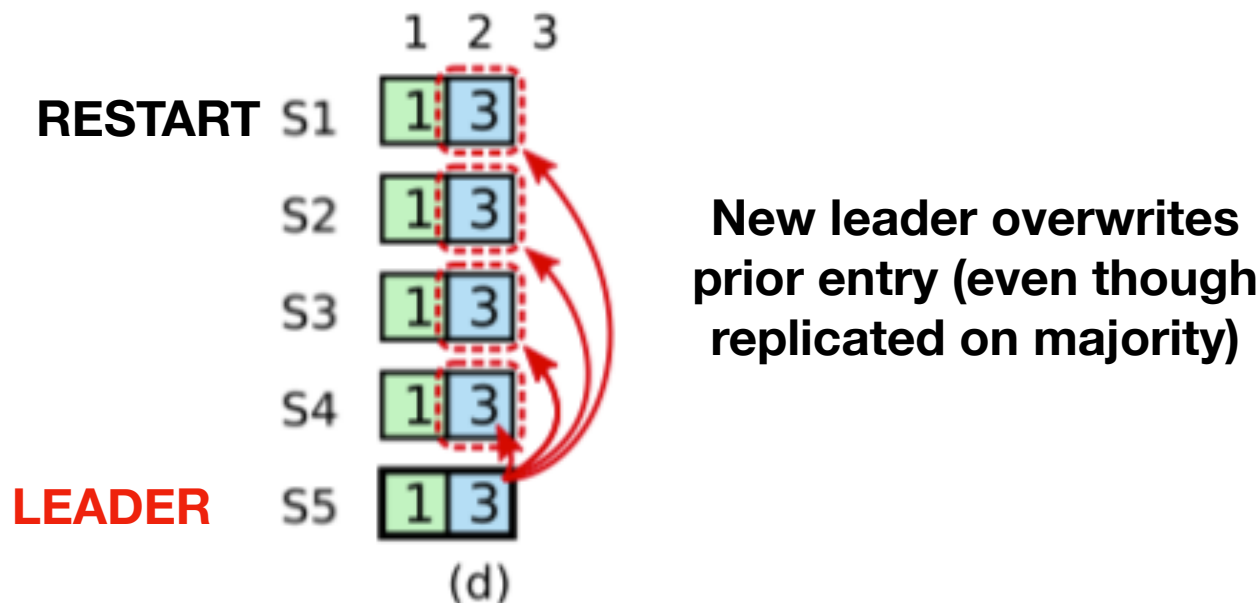
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



- QUESTION: Can the leader call index 2 "committed"?

# "Figure 8"

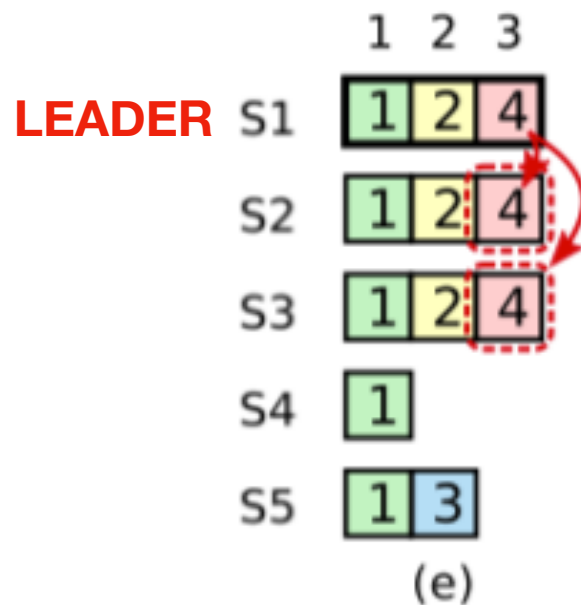
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



- ANSWER: NO!

# "Figure 8"

- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



It is safe to commit earlier entries once the leader has committed an entry from its own term

- If later term committed, then S5 can't win election.

# A Checklist

- Get message with newer term : Become follower
- Message with older term: Ignore
- Vote for only one candidate per term
- Grant vote if candidate log is at least as up to date
- Leader never commits entries from prior leaders until an entry from its own term is committed.

# Project 8

- How would you know that it's not safe for a leader to commit entries from earlier log terms?
- Can you write an assertion/invariant to detect this problem during runtime?
- Can you write a test that discovers the problem?



**Part 9**

# Model Checking

# Thought

- Systems modeling/testing is hard
- Can it be formalized in some way?
- Can an algorithm be proven?

# Systems as State Machines

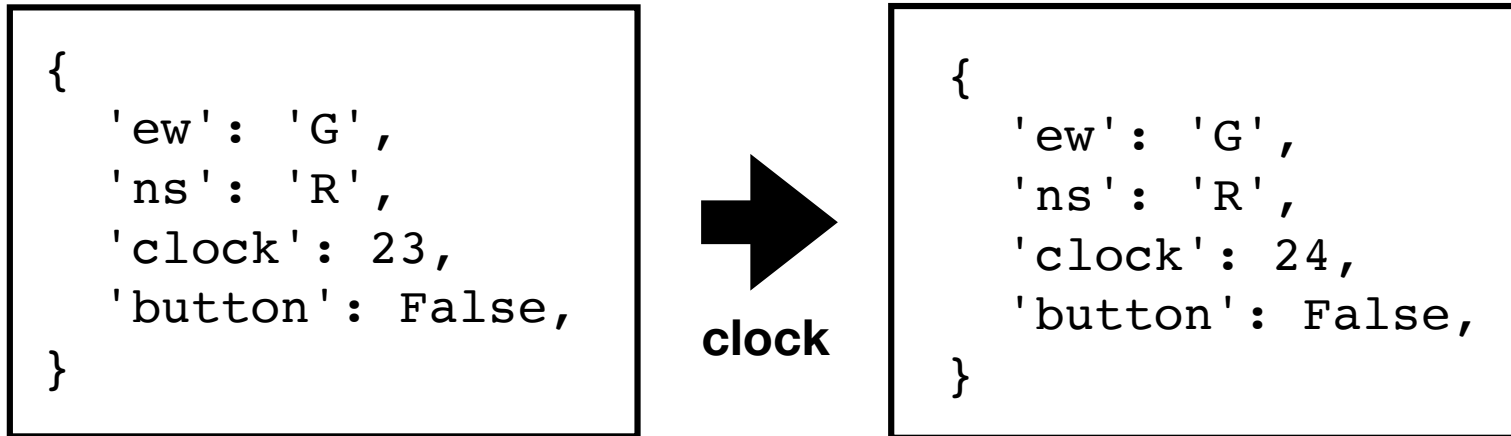
- System is comprised of "state"

```
{  
    'ew': 'G',  
    'ns': 'R',  
    'clock': 23,  
    'button': False,  
}
```

- The state is a data structure (e.g., a dict)

# Sequencing

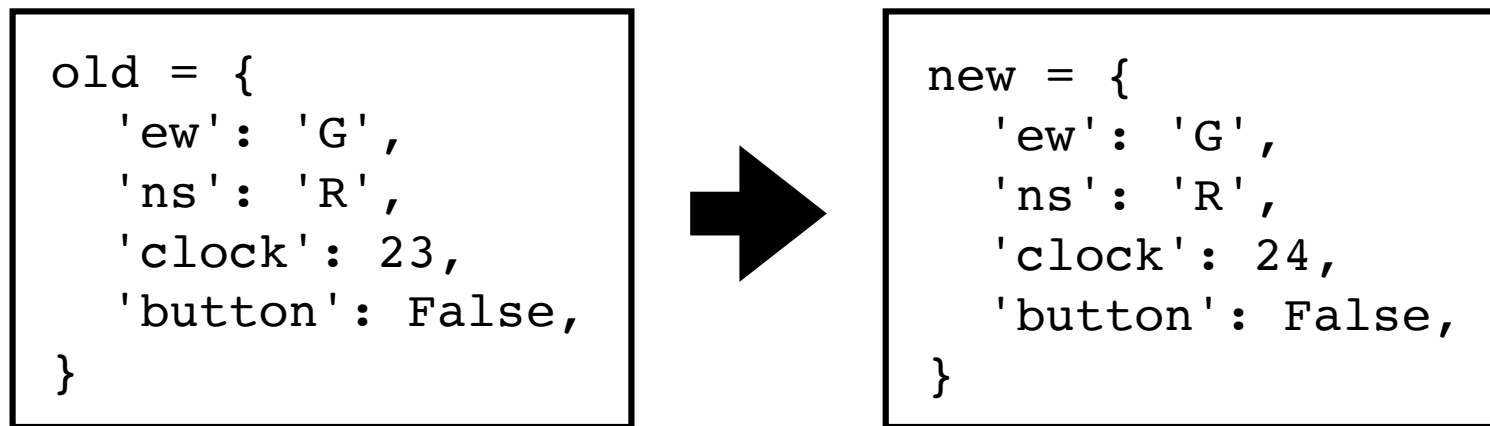
- State machines execute via events



- Events cause a state change

# Modeling State Changes

- A state change is an update to the values



- It's a function: state -> state

```
new = dict(old, clock=old['clock']+1)
```

- New state is old values + updated values

# Modeling a State Machine

- Can implement a "next state" function

```
def next_state(state, event):
    if (state['ew'] == 'G'
        and state['ns'] == 'R'
        and event == 'clock'
        and state['clock'] < 30):
        return dict(state, clock=state['clock'] + 1)
    if (state['ew'] == 'G'
        and state['ns'] == 'R'
        and event == 'clock'
        and state['clock'] == 30):
        return dict(state, ew='Y', clock=0)
    if (state['ew'] == 'Y'
        and state['ns'] == 'R'
        and event == 'clock'
        and state['clock'] < 5):
        return dict(state, clock=state['clock'] + 1)
    ...
```

# Modeling a State Machine

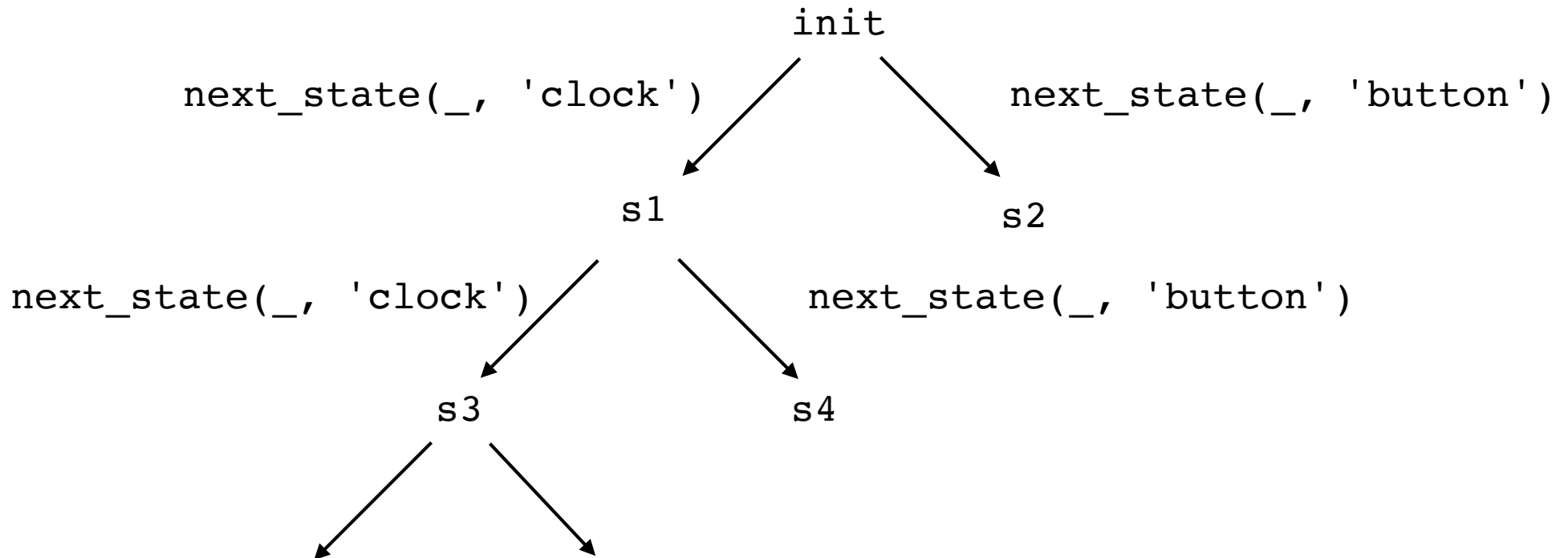
- Example:

```
>>> init = { 'ew': 'G', 'ns': 'R', 'clock': 0, 'button': False }
>>> next_state(initial, 'clock')
{ 'ew': 'G', 'ns': 'R', 'clock': 1, 'button': False}
>>> next_state(_, 'clock')
{ 'ew': 'G', 'ns': 'R', 'clock': 2, 'button': False}
>>> next_state(_, 'clock')
{ 'ew': 'G', 'ns': 'R', 'clock': 3, 'button': False}
>>>
```

- This kind of formalism allows you to "turn the crank" on a state machine (more mathematical)

# Simulating a State Machine

- A next-state function can form the basis of a state-state simulation



- Generate every event and watch it unfold



# Simulation Goals

- Simulation might uncover fatal flaws in the logic
- Deadlock: A state where no further progress can be made (all events produce no state change)
- Violation of invariants (example: a traffic light showing green in all directions).
- Running a simulation might make testing easier since it's decoupled from the runtime environment (i.e., sockets, threads)

# Project 9

- Can you implement a model for the traffic light problem and test it via simulation?
- A program that explores every possible configuration of the state machine and verifies certain invariants or behaviors
- No dependence on the system (sockets, etc.).

# TLA+

<https://lamport.azurewebsites.net/tla/toolbox.html>

- A tool for modeling/verifying state machines
- It is based on a similar mathematical foundation
- And there is a TLA+ spec for Raft
- The spec is useful in creating an implementation, but you must be able to read it
- A demonstration follows...

# TLA+ Definitions

- Definitions are made via ==

```
Value == 42  
Name == "Alice"
```

- There are some primitive datatypes

```
23      \* Integers (note: this is a comment)  
TRUE    \* Booleans  
"G"     \* Strings
```

- Operators (like a function)


```
Square(x) == x*x
```

# Boolean Logic

- AND, OR, NOT operators

|        |            |
|--------|------------|
| A /\ B | \* AND (^) |
| A \/ B | \* OR (v)  |
| ~A     | \* NOT (¬) |

- Grouping by indentation (implies parens)

|                         |  |                          |
|-------------------------|--|--------------------------|
| /\ a                    |  | ( a                      |
| /\ b                    |  | and b                    |
| /\  \/ c > 10  /\ d = 0 |  | and ((c > 10 and d == 0) |
| \/ c > 20 /\ d = 1      |  | or (c > 20 and d == 1))  |
| /\ e                    |  | and e)                   |

# TLA+ State Variables

- State is held in designated variables

```
VARIABLES ew, ns , clock, button
```

- Variables are initialized in a special definition

```
Init == /\ ew = "G"  
        /\ ns = "R"  
        /\ clock = 0  
        /\ button = FALSE
```

- This is the "start" state

# TLA+ Next State

- Next state is expressed as a math formula

```
Next == \/  
        /\ ew = "G"  
        /\ ns = "R"  
        /\ clock < 30  
        /\ clock' = clock + 1  
        /\ UNCHANGED <<ew, ns, button>>
```

```
\/  
        /\ ew = "G"  
        /\ ns = "R"  
        /\ clock = 30  
        /\ clock' = 0  
        /\ ew' = "Y"  
        /\ UNCHANGED <<ns, button>>
```

...

# TLA+ Next State

- The first part of each "rule" are conditions

```
Next == \/  
    /\ ew = "G"  
    /\ ns = "R"  
    /\ clock < 30  
    /\ clock' = clock + 1  
    /\ UNCHANGED <<ew, ns, button>>  
  
    \/  
    /\ ew = "G"  
    /\ ns = "R"  
    /\ clock = 30  
    /\ clock' = 0  
    /\ ew' = "Y"  
    /\ UNCHANGED <<ns, button>>  
  
    ...
```



# TLA+ Next State

- The last part of each rule indicate state change

```
Next == \/  
  /\ ew = "G"  
  /\ ns = "R"  
  /\ clock < 30  
  /\ clock' = clock + 1  
  /\ UNCHANGED <<ew, ns, button>>  
  
  \/  
  /\ ew = "G"  
  /\ ns = "R"  
  /\ clock = 30  
  /\ clock' = 0  
  /\ ew' = "Y"  
  /\ UNCHANGED <<ns, button>>
```

- State changes written:  $var' = expression$

# Big Picture

- TLA+ is NOT an implementation language
- There is no "runtime" in which you make a working state machine or process events
- It is purely a simulation tool
- Goal is to uncover flaws in your reasoning about how systems work and how the parts interact.

# Bigger Picture

- There is a TLA+ spec for Raft.
- Take a look at formal Raft TLA+ spec

<https://github.com/ongardie/raft.tla>

- Can you make any sense of it?
- Example: What happens when an "AppendEntries" message is received?

Part 10

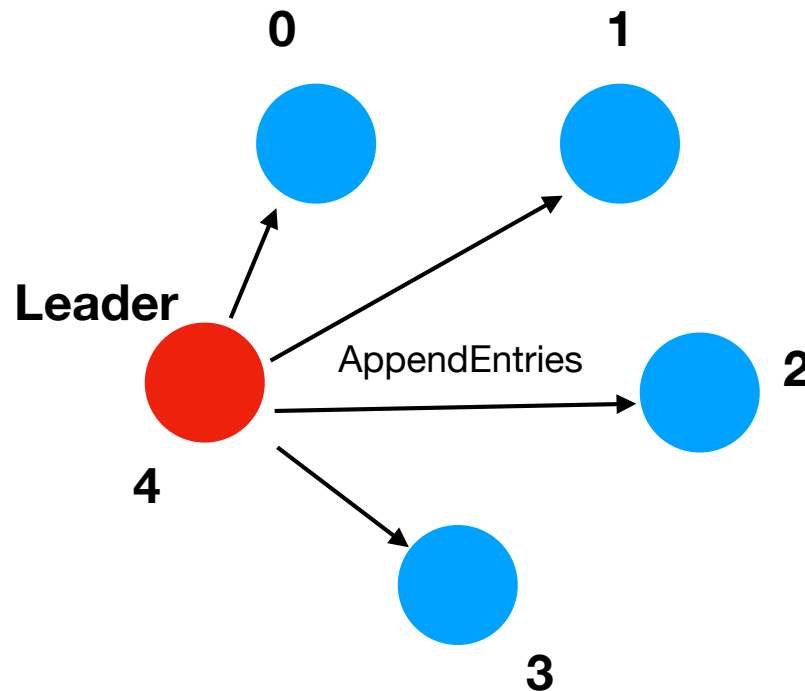
# Timing/Systems

# Heartbeats and Timeouts

- Raft relies on two timing mechanisms
  - Leader heartbeat
  - Election timeout
- There are some subtle facets to both

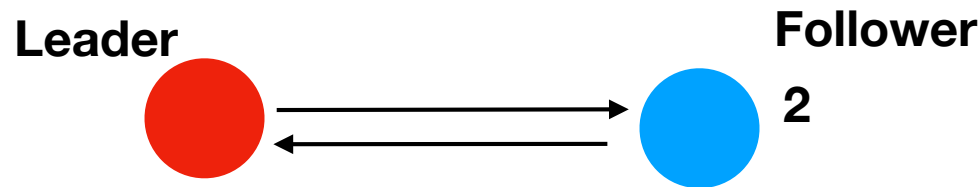
# Heartbeats

- AppendEntries is the leader heartbeat
- Sent on periodic interval even if empty



# Heartbeat Timing

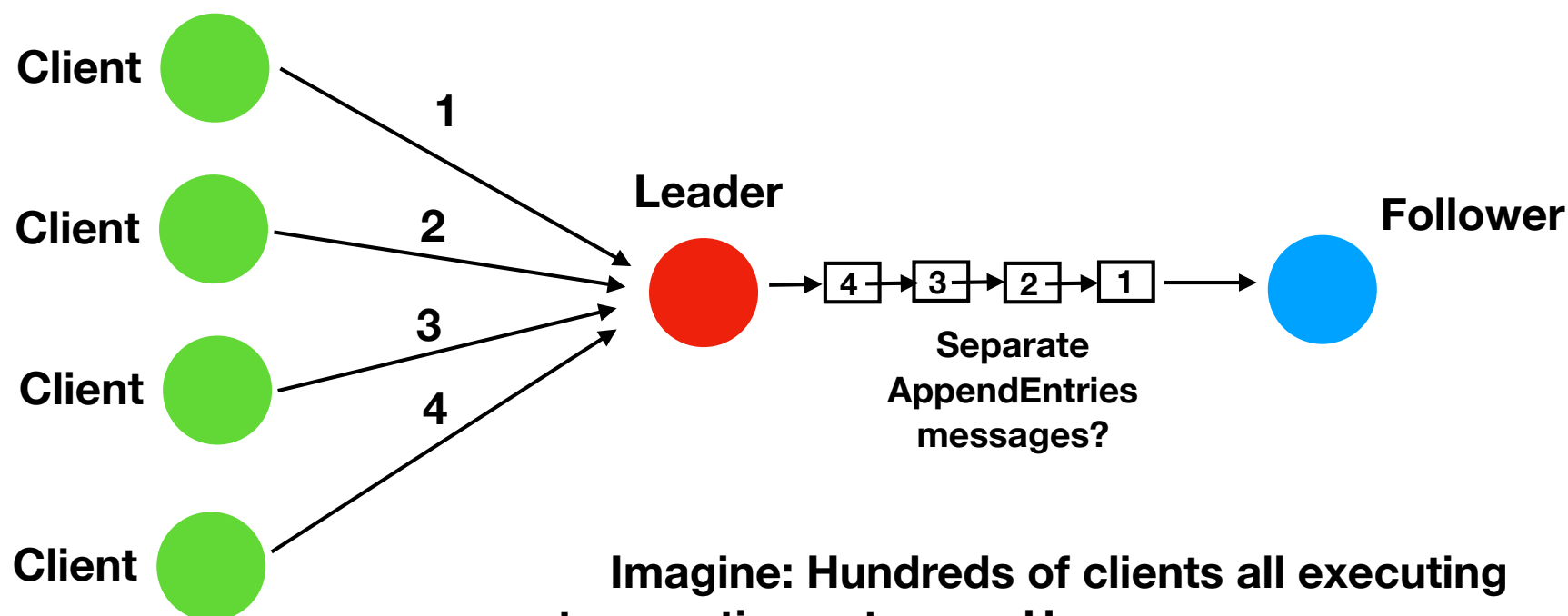
- Timing based on round-trip "ping time"



- There is a request/response cycle
- It doesn't make much sense to send a new `AppendEntries` to the follower before hearing back from the previous request (the new request would duplicate the last request).

# Clients and Heartbeats

- Q: Does the leader immediately send AppendEntries for every client transaction?
- Scenario:

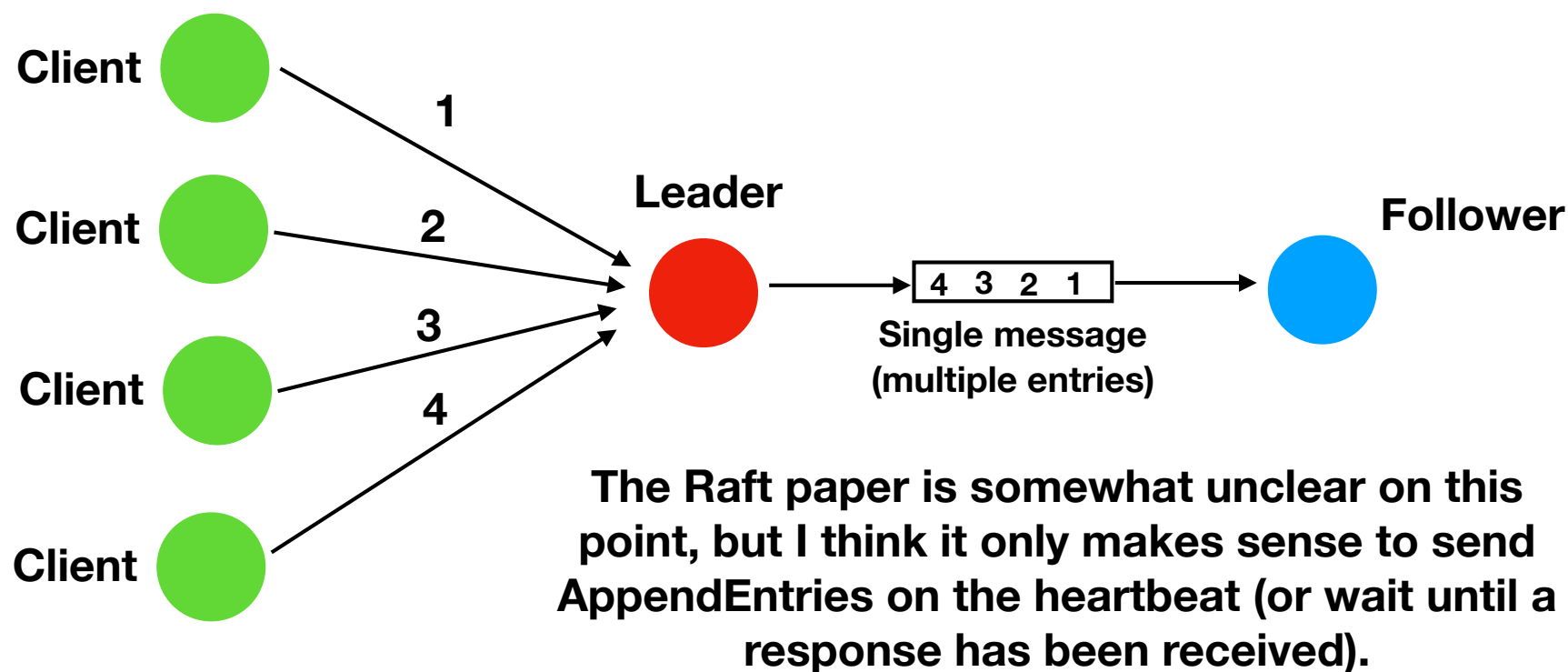


**Imagine: Hundreds of clients all executing transactions at once. How many messages are generated in the Raft network?**



# Clients and Heartbeats

- Or does the leader accumulate entries and only send a message on the heartbeat?
- Scenario:



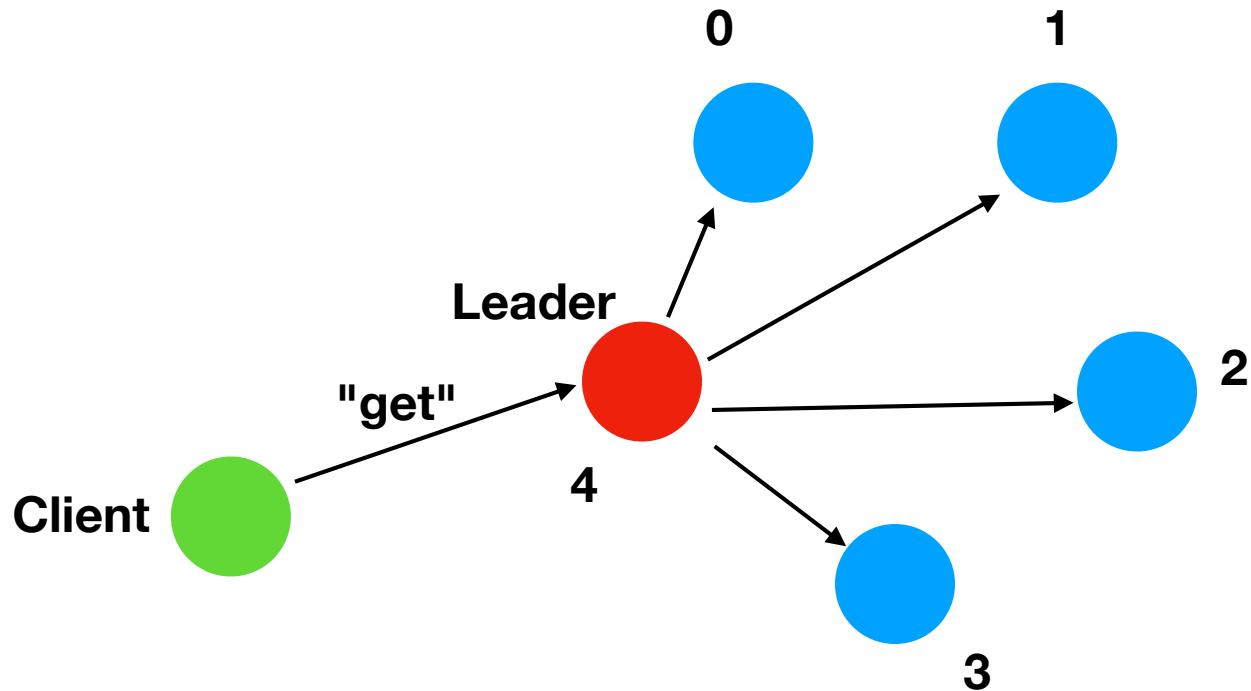
# Discussion

- In Raft paper, heartbeat is on order of 0.5 - 20ms
- Response time to client would be bounded by this in some way (i.e., a client making a request would have to wait at least this long to know if consensus was reached)
- However, multiple client requests could happen in parallel (could have 100s of pending requests that all go at once).

# Further Discussion

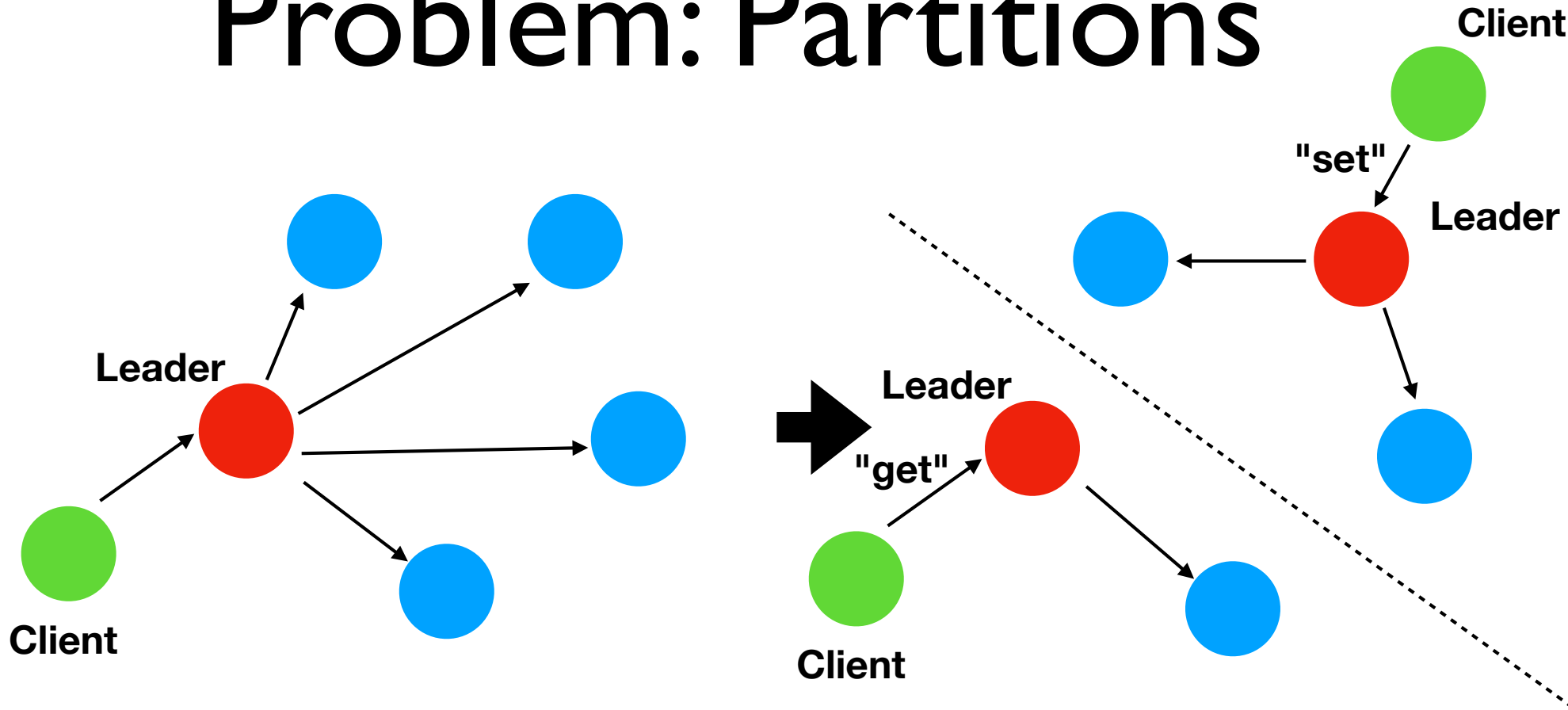
- Is the heartbeat a global timer that applies to all followers at once?
- Or does the leader maintain a different heartbeat timer for each follower?
- The Raft paper leaves this unanswered. We just know that each follower should regularly receive `AppendEntries`.

# The Trouble with Reads



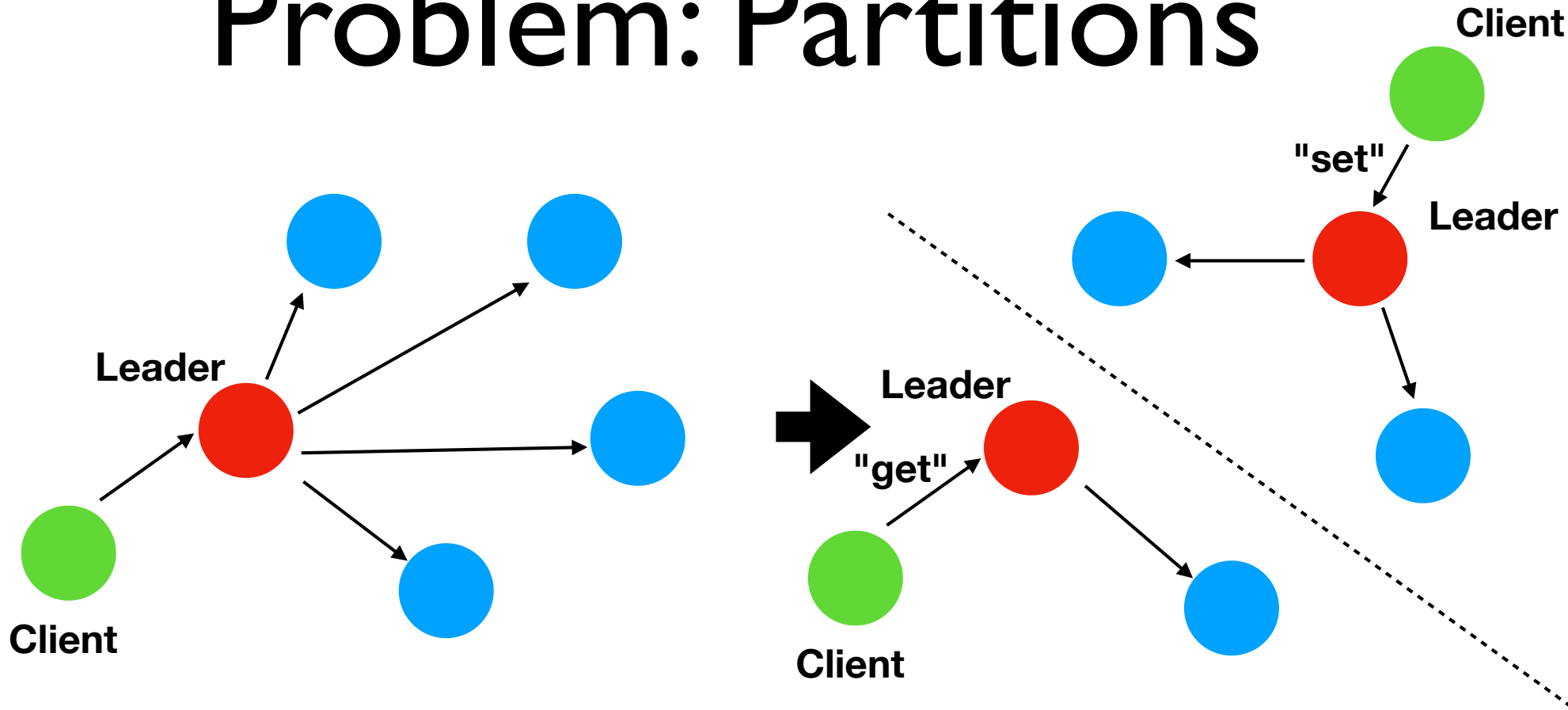
- Question: Are "reads" a transaction in the log?
- They don't modify state. Is it critical?

# Problem: Partitions



- A partition results in two "leaders"
- Former leader doesn't know it's cut off
- What about its clients?

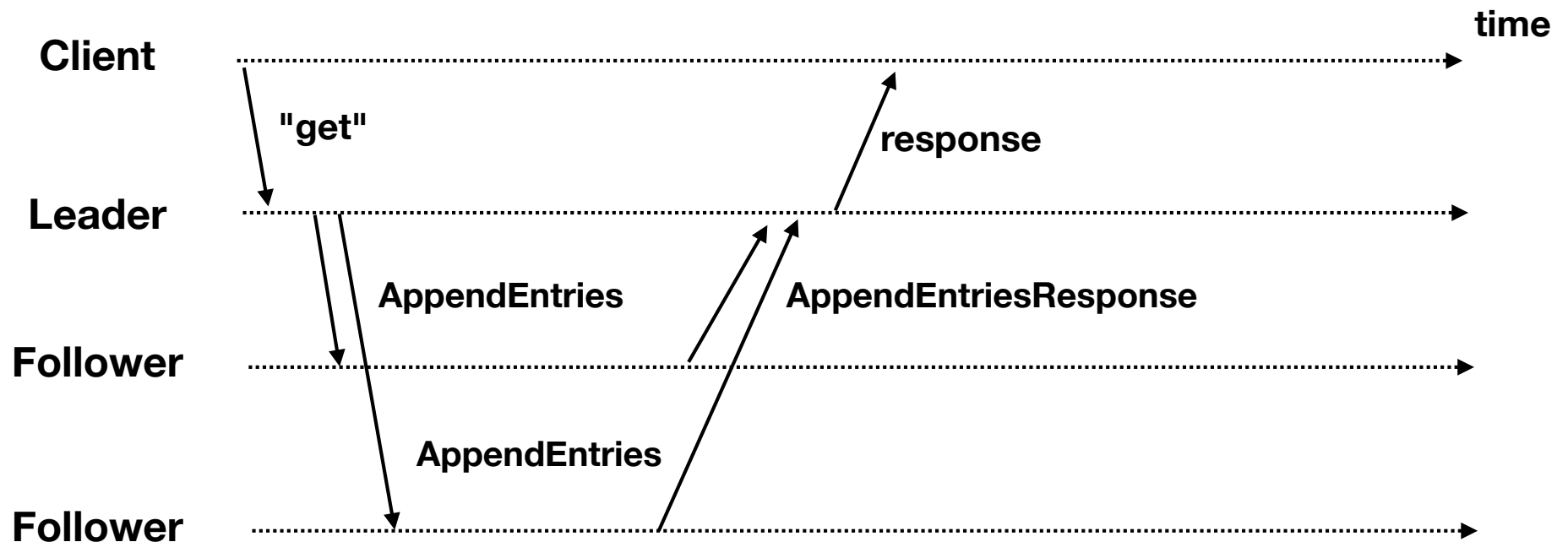
# Problem: Partitions



- A client might read stale data off the old leader
- This is (probably) bad.

# Solution

- Leader is not allowed to respond to any (read) request until it has exchanged an AppendEntries message with a majority of the cluster after it received the request



# An Extra Problem

- How does a new leader know what log entries are committed?
- There is a guarantee that the leader has all committed entries, but at startup the leader doesn't actually know what they are (yet)
- But there's this extra twist that a new leader can't commit entries from a previous leader without committing an entry from its own term (section 5.4.2, Figure 8)



# Solution

- Newly elected leaders immediately append a "no-op" into the log
- Once committed, leader knows what has been committed and can respond to read requests

# Project 10

- Build the Raft timer mechanisms
- How does one determine leadership for read/get requests?
- Work on Raft systems implementation

**Part 11**

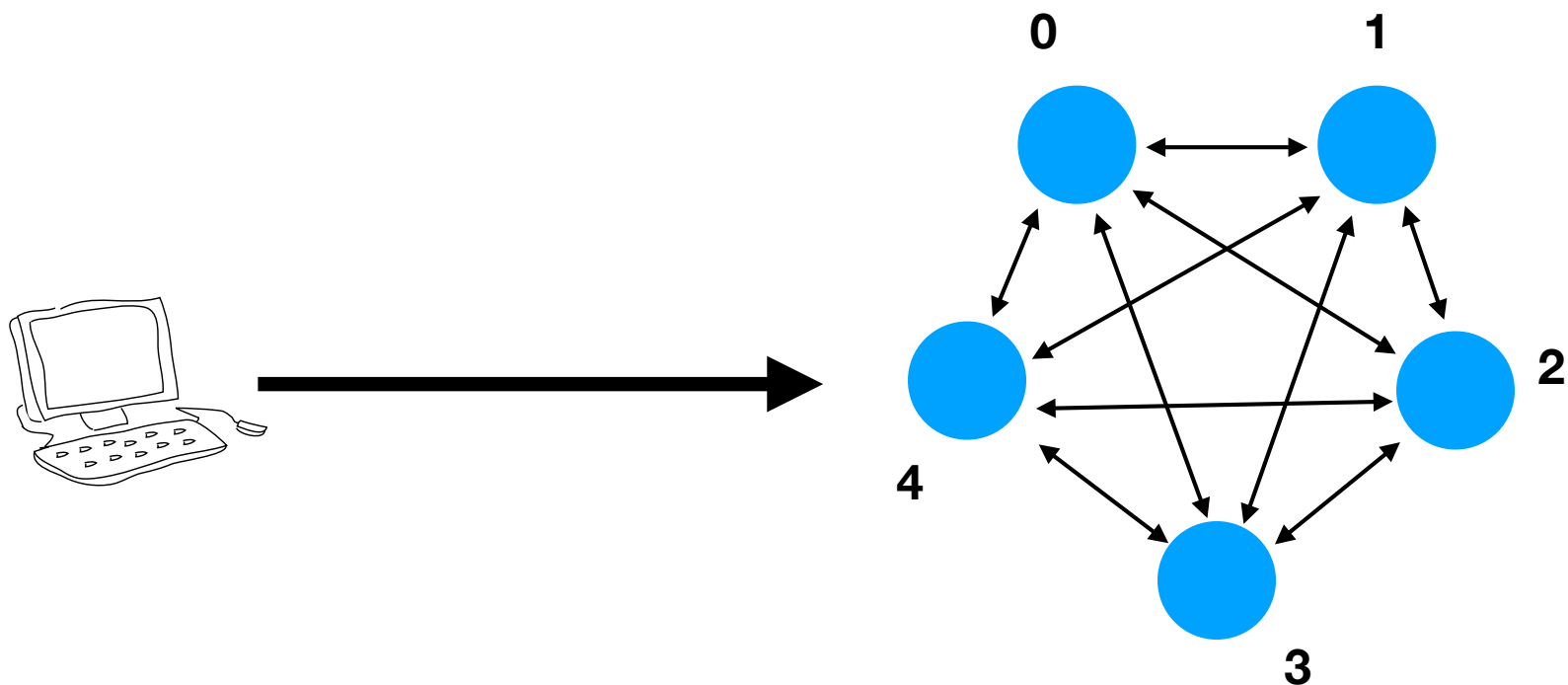
# The Application

# System Implementation

- At some point, we have to turn Raft into an operational "system" of some sort
- Must deal with other details
- Client-Raft interface

# Client-Connection

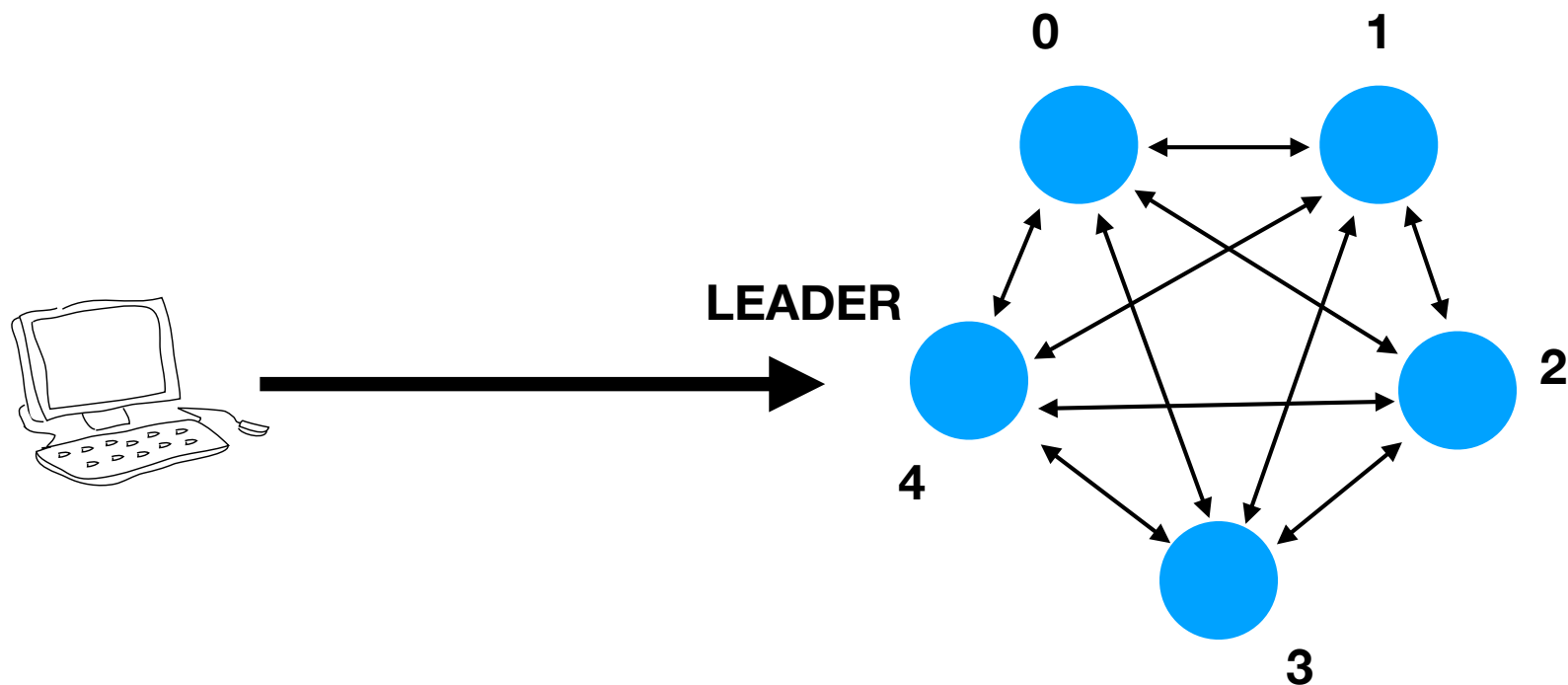
- At some point, a client must talk to Raft



- For example, to implement a key-value store

# Strong Leader

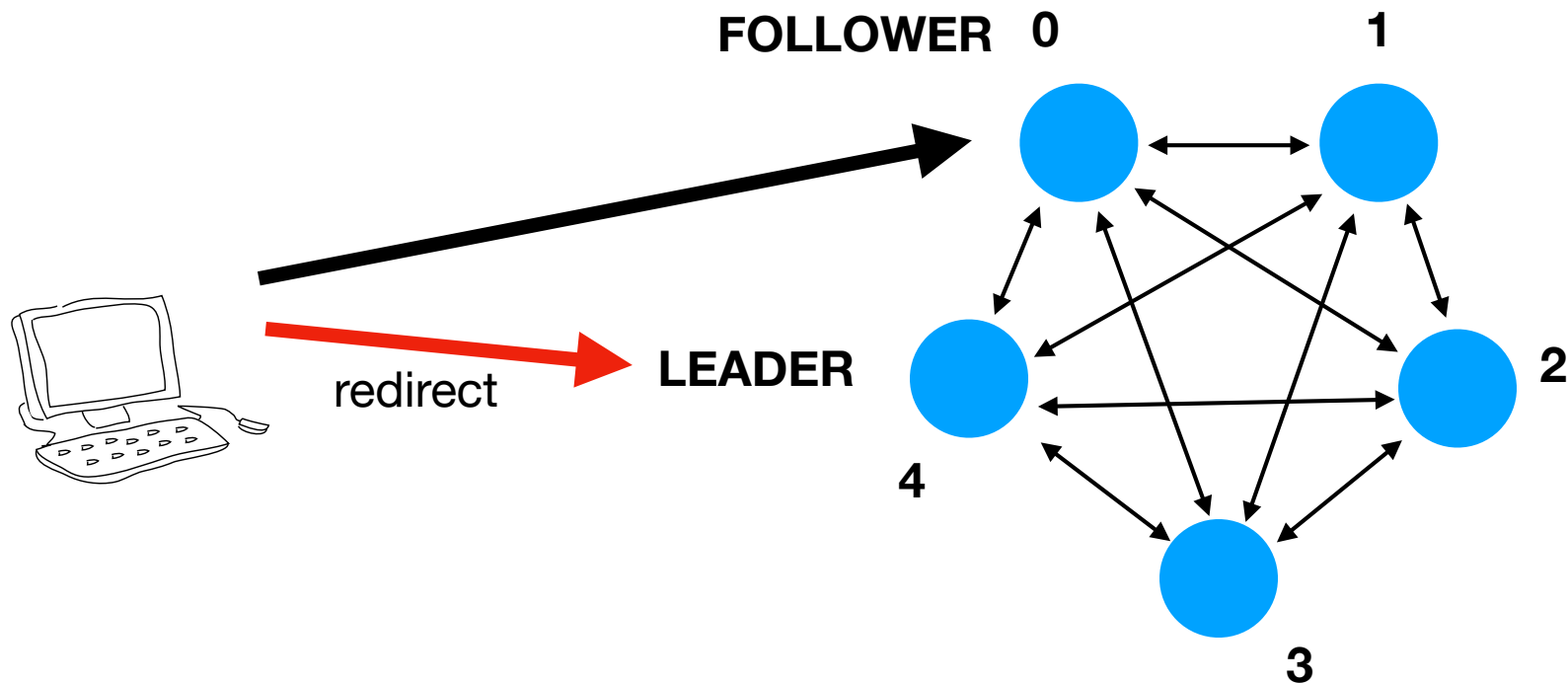
- Clients only talk to the leader



- Remember: Everything happens via leader

# Redirects

- If connected to a follower, it can redirect



- A follower could also just drop the connection

# Finding the leader

- Client connects to a random server. It's either the leader or it redirects the client to leader
- Alternative: Client just tries servers in order until it finds the leader
- In normal operation, the leader won't change very often--if you're deploying Raft, you're going to try and run it on "reliable" hardware



# Client Responses

- The leader only responds to a client when the request has been committed
- Meaning: replicated on a majority of servers
- In the paper: "Applied to the state machine" means that an entry was committed and that a leader can respond.

# Handling Leader Crashes

- Clients must implement a timeout.
- If no response from leader within a given time period, retry the request (on a different server)

# Duplicate Requests

- It's possible that a client request could be received twice by Raft
- Scenario: Log entry gets committed, but the leader crashes before it can respond to client
- Client retries the same request and it gets executed again.
- One solution: Use unique serial numbers on client requests, don't re-execute requests if already executed.

# Project I I

- Build the Raft-Application interface
- Implement a fault-tolerant Key-Value store