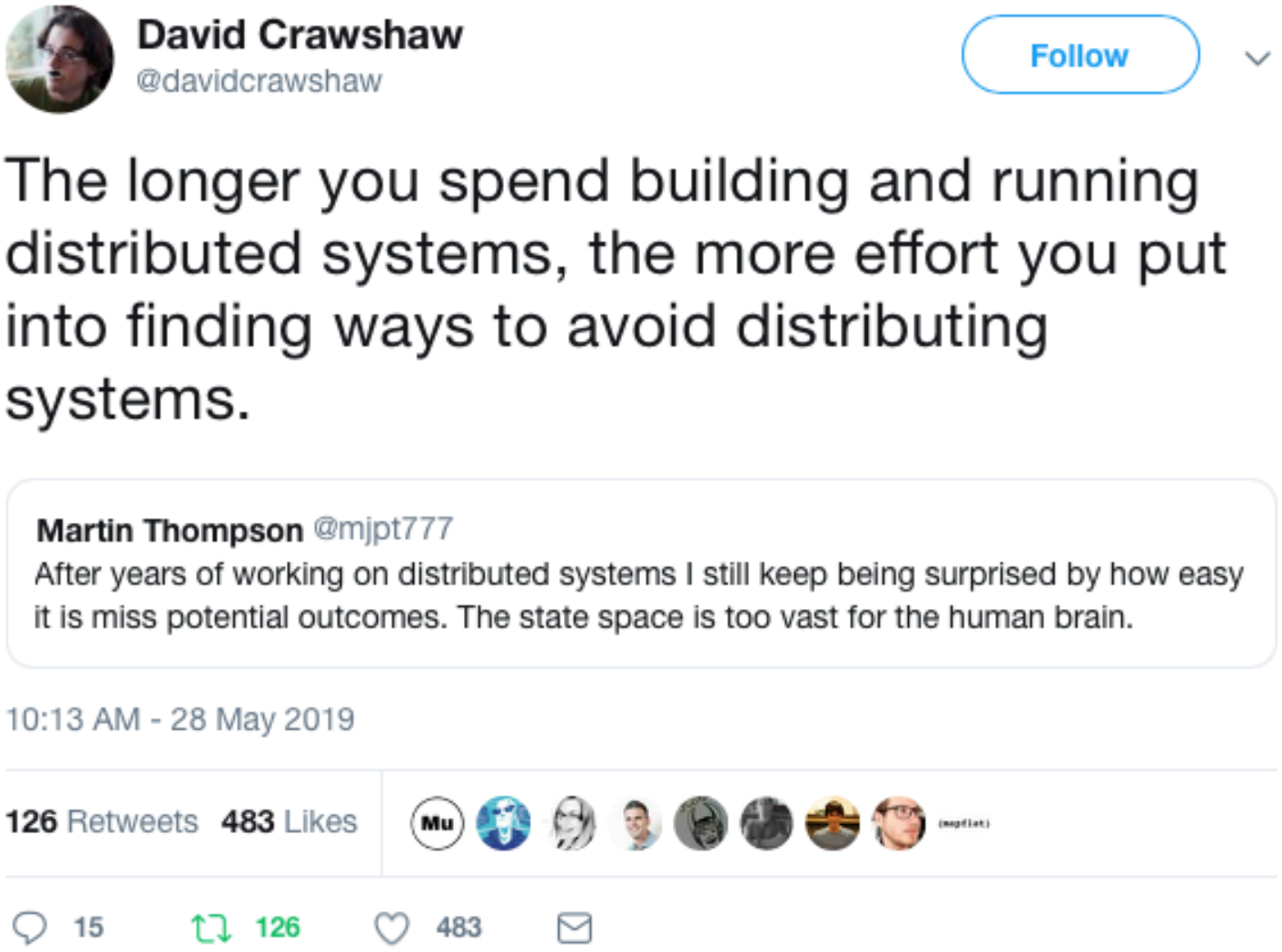
Rafting Trip

(AKA: Distributed Systems)

David Beazley
(@dabeaz)
https://www.dabeaz.com

Saw Recently...



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

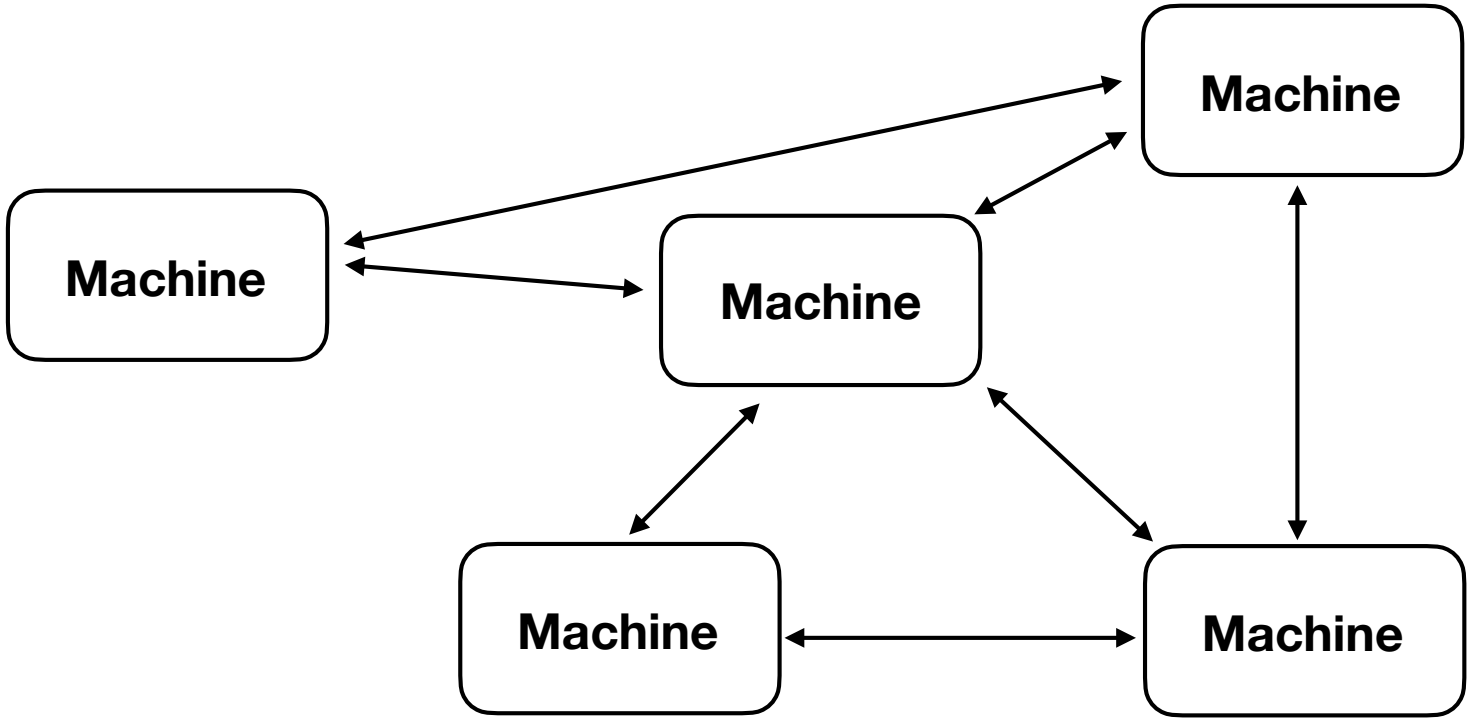
Introduction

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



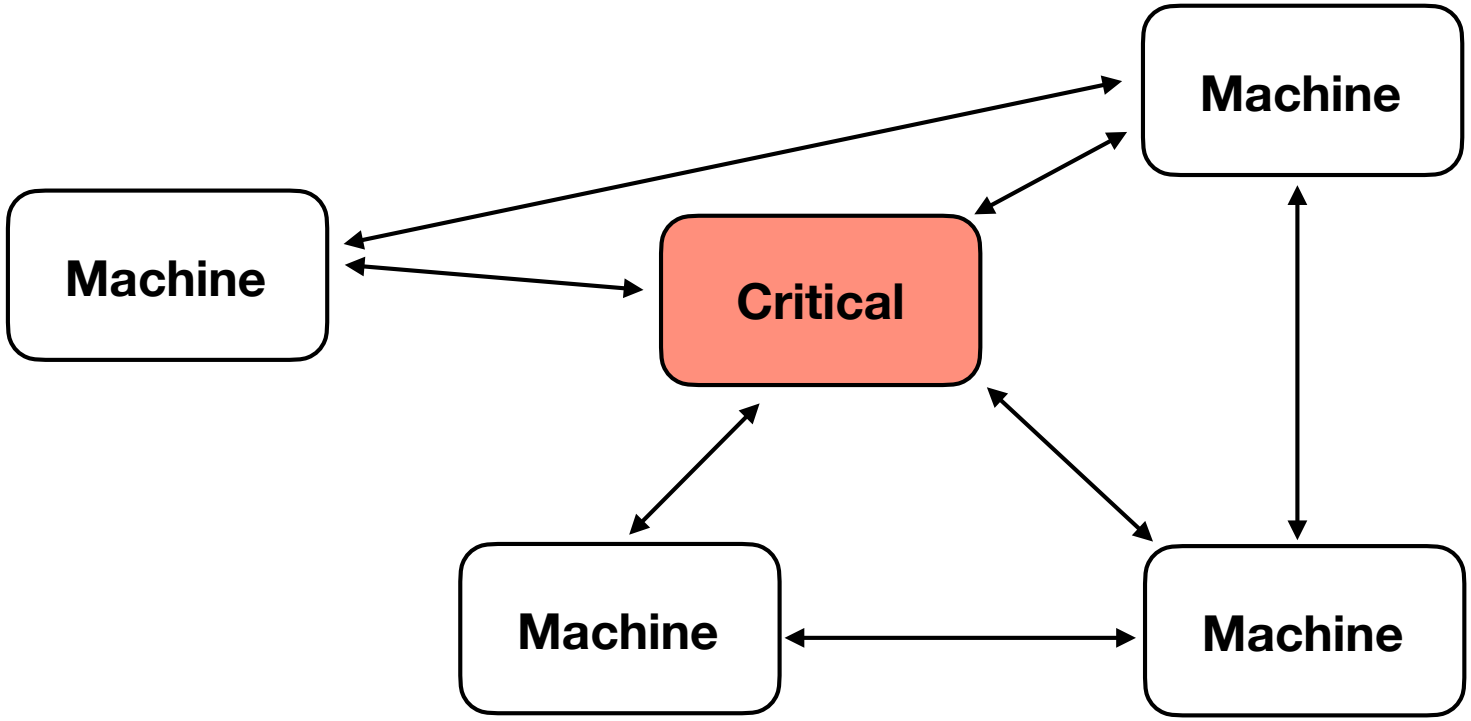
- Machines/services communicating over a network

Motivations

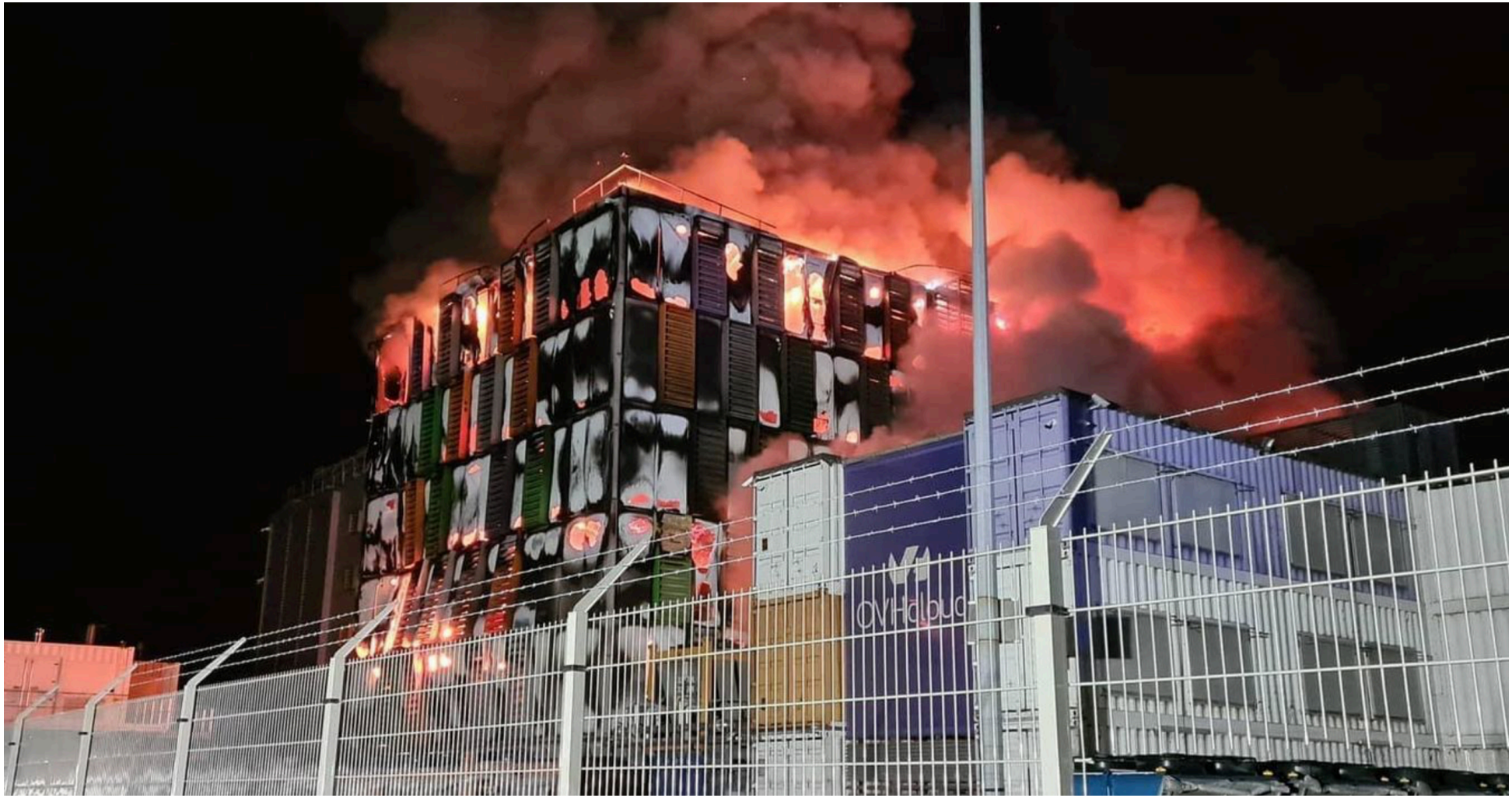
- Performance : Parallel Computing
- Modularity : Microservices, Web services, etc.
- Reliability : Fault Tolerance, Redundancy

Our Focus: Reliability

- Certain services are more critical than others

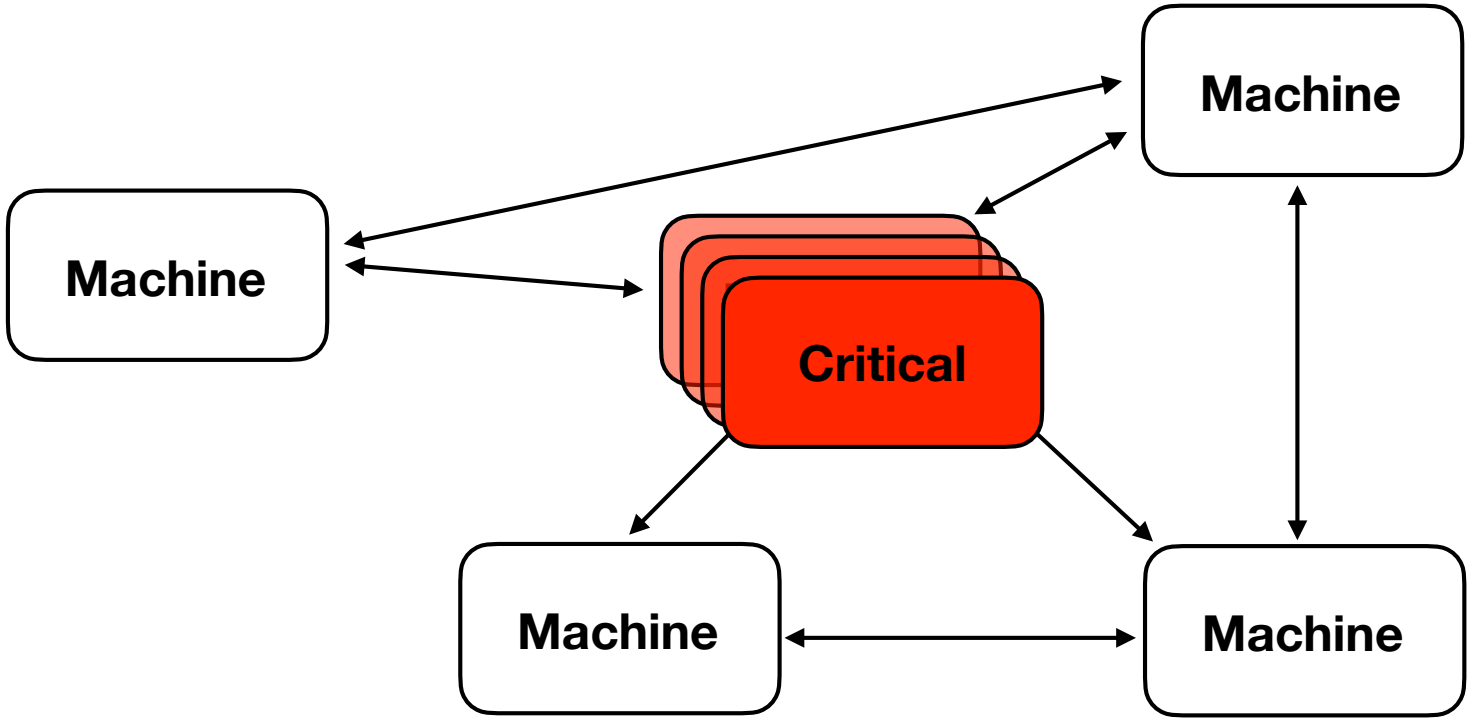


- Example: Centralized registries, database, etc.



Our Focus: Reliability

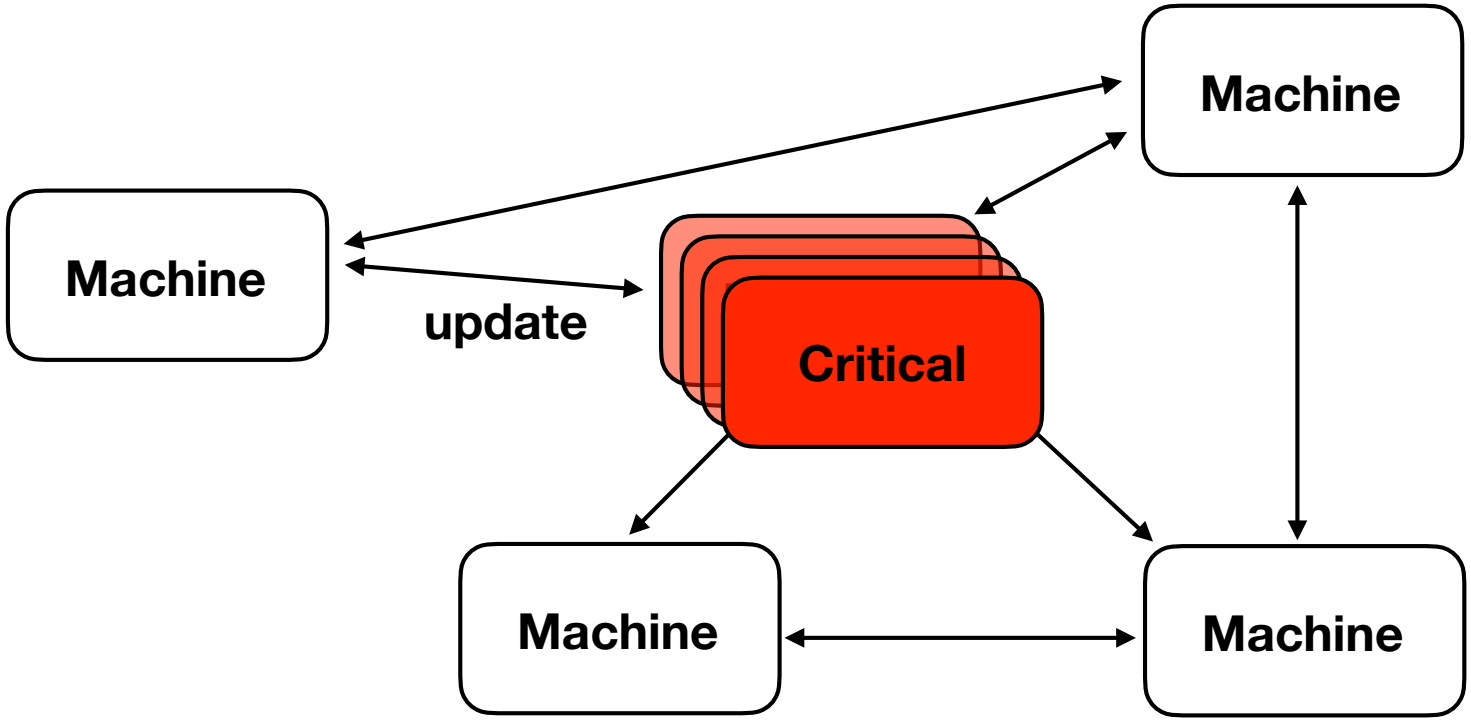
- Obvious Solution: Replication!



- Whew! Crisis averted via redundancy.

A Problem

- Mutable state

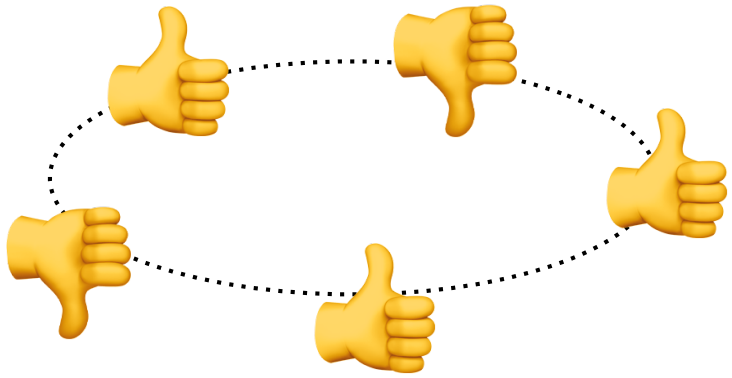


- It is impossible to simultaneously update the state on all replicas at once (laws of physics, time, etc.)

- So, how do you deal with that?

Solution: Consensus

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



- This is one of the fundamental hard problems of distributed computing

- Algorithms: Distributed Consensus

Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems

Brian M. Oki
Barbara H. Liskov

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

(1988)

It's not a "new" problem

Abstract

One of the potential benefits of distributed systems is their use in providing highly-available services that are likely to be usable when needed. Availability is achieved through replication. By having more than one copy of information, a service continues to be usable even when some copies are inaccessible, for example, because of a crash of the computer where a copy was stored. This paper presents a new replication algorithm that has desirable performance properties. Our approach is based on the primary copy technique. Computations run at a primary, which notifies its backups of what it has done. If the primary crashes, the backups are reorganized, and one of the backups becomes the new primary. Our method works in a general network with both node crashes and partitions. Replication causes little delay in user computations and little information is lost in a reorganization; we use a special kind of timestamp called a viewstamp to detect lost information.

Paxos

- Perhaps the most cited algorithm (Leslie Lamport)
    - First published (1989), First Journal Article (1998, submitted 1990)
    - Notable for having a formal verification
- Problem: Translating Paxos into an actual implementation is notoriously hard (mathematical, inscrutable "details")

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.

- Diego Ongaro

Our Challenge

In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout, *Stanford University*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

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

Real World Raft Example

What is etcd?

Project

etcd is a strongly consistent, distributed key-value store that provides a reliable way to store data that needs to be accessed by a distributed system or cluster of machines. It gracefully handles leader elections during network partitions and can tolerate machine failure, even in the leader node.

Applications of any complexity, from a simple web app to Kubernetes, can read data from and write data into etcd.

Technical overview

etcd is written in Go, which has excellent cross-platform support, small binaries and a great community behind it. Communication between etcd machines is handled via the Raft consensus algorithm.

Latency from the etcd leader is the most important metric to track and the built-in dashboard has a view dedicated to this. In our testing, severe latency will introduce instability within the cluster because Raft is only as fast as the slowest machine in

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.

Why Raft?

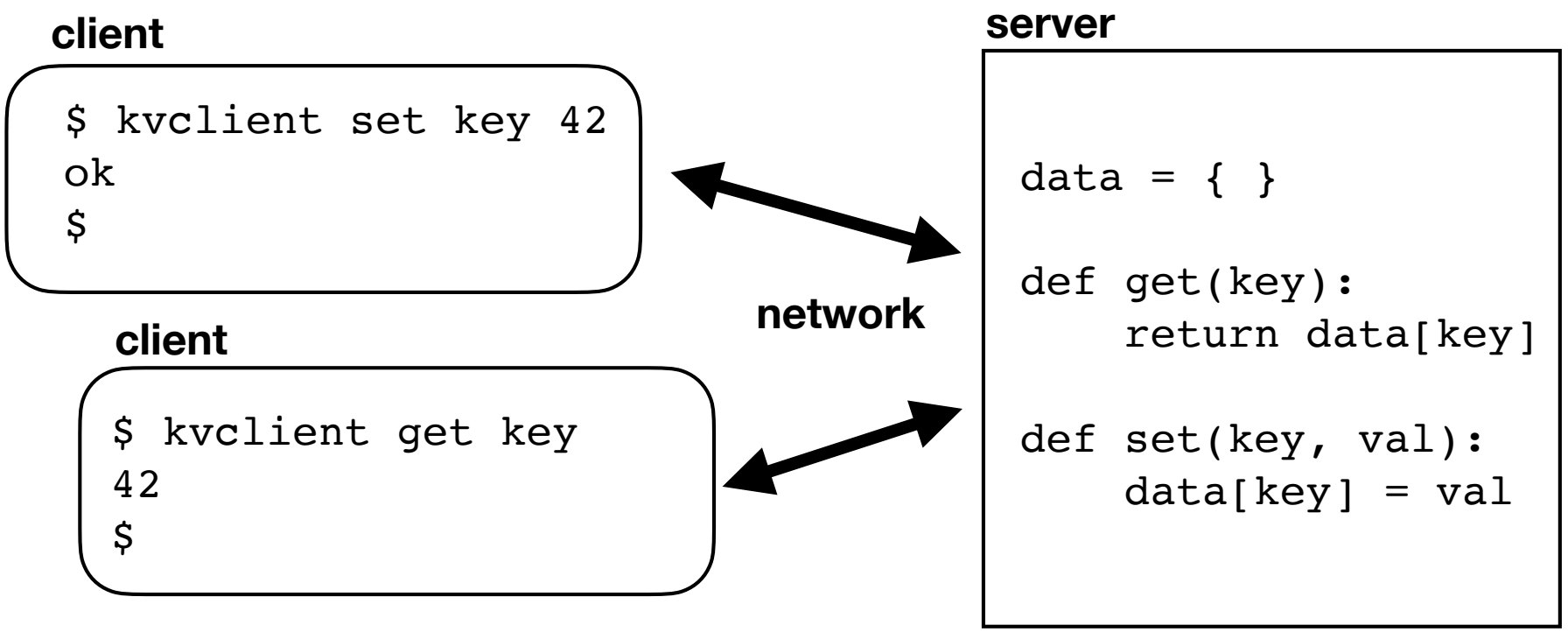
- There are many facets of distributed computing. However, distributed consensus might be one of the hardest.
- Fault tolerance: Cost of failure is high.
- Non-trivial: Many moving parts. Not an "echo server."
- Challenging: concurrency, networks, testing, etc.
- Solving it involves problems that transcend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

Core Topics

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

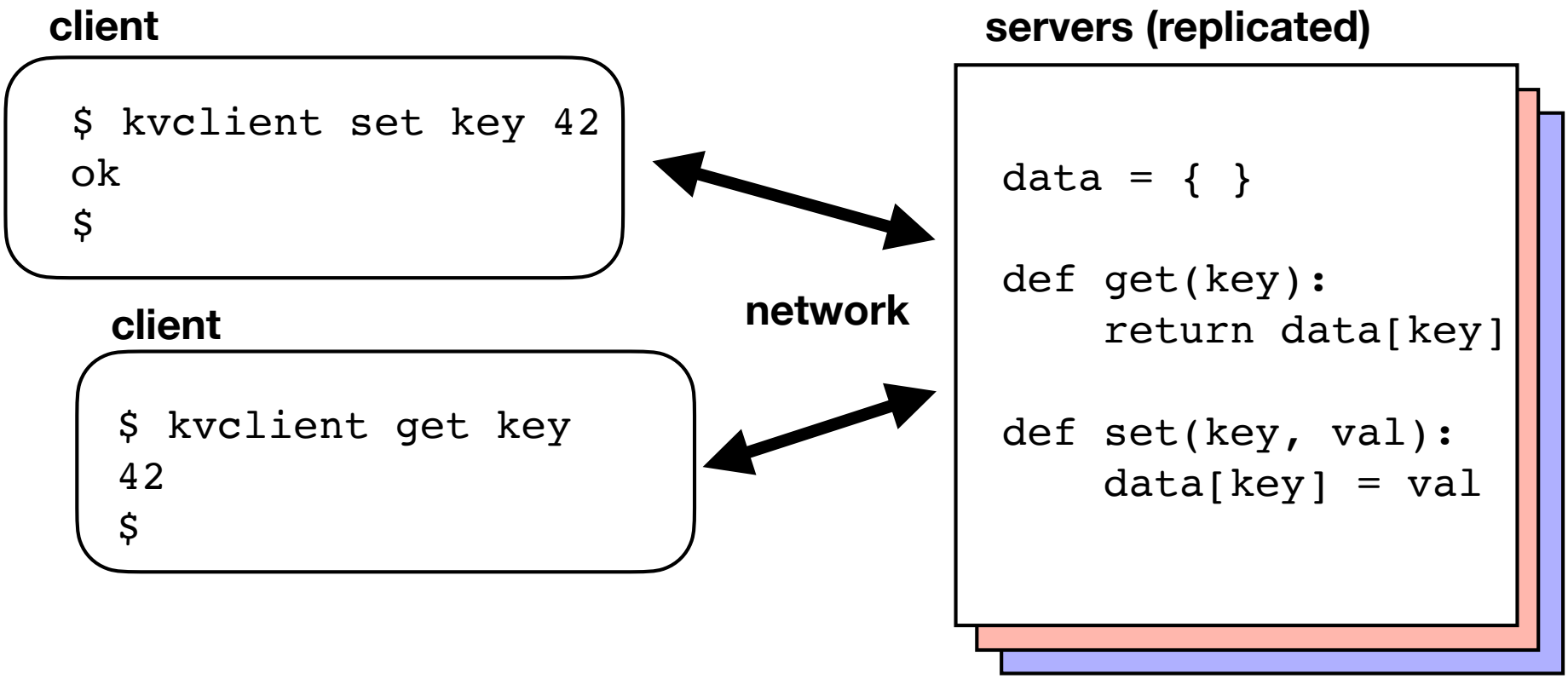
The Project

- 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

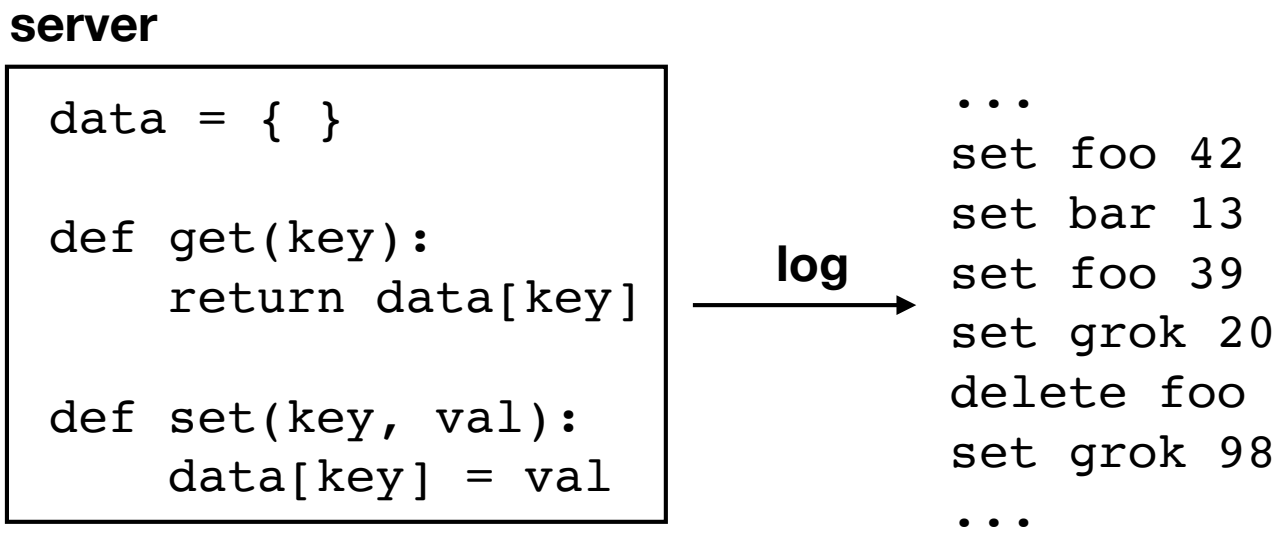


Raft

- 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

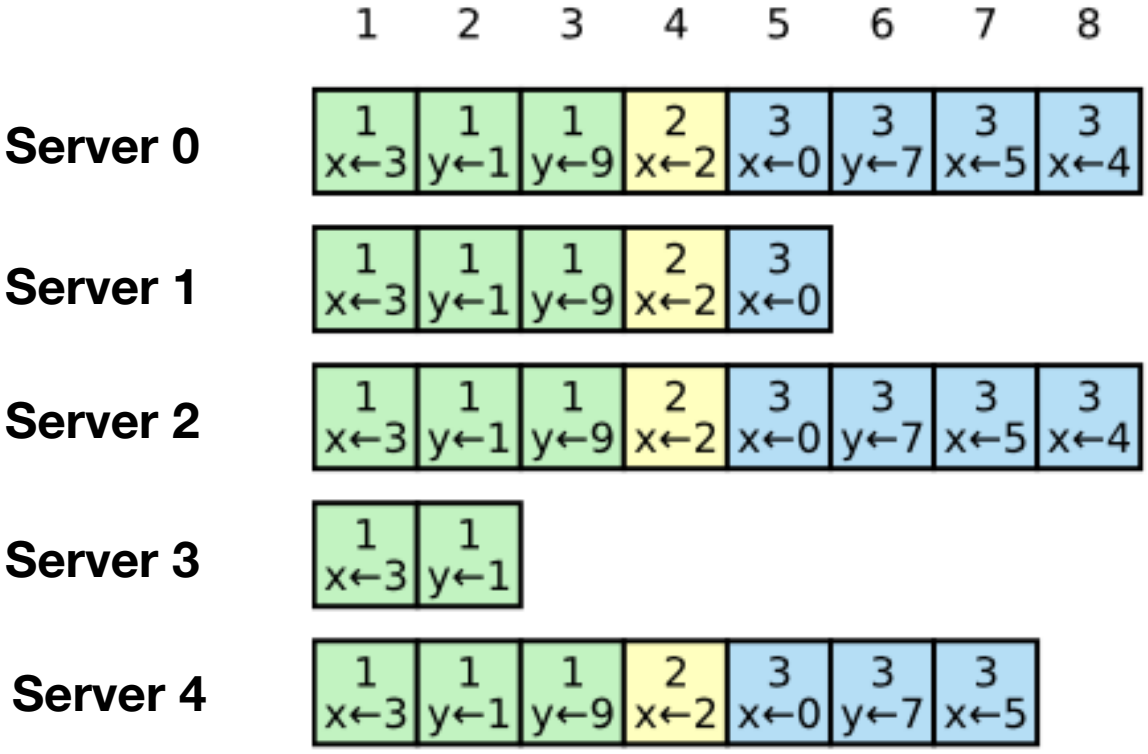
- Raft manages a transaction log



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

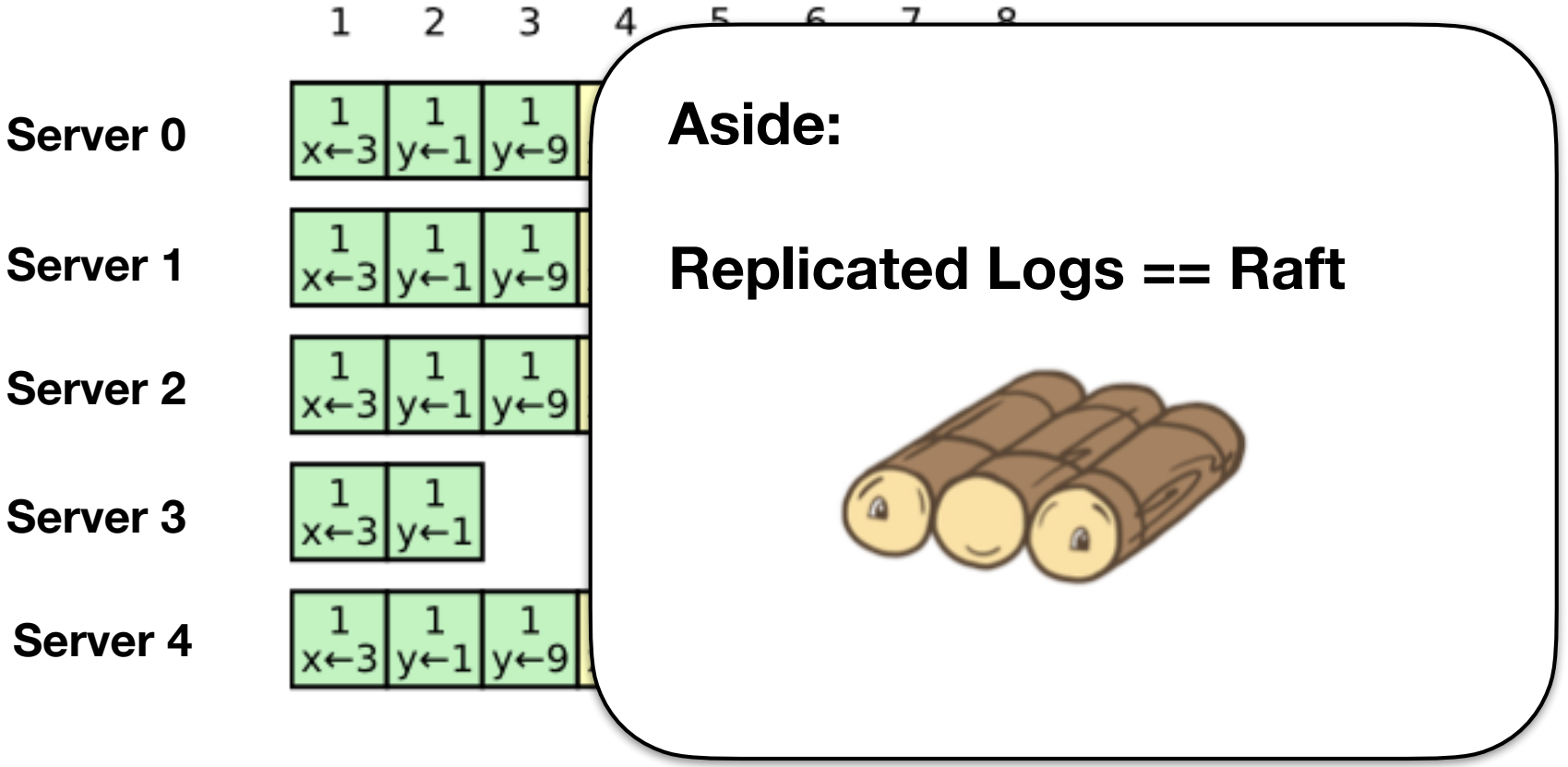
Replication

- Fault tolerance achieved by replicating the transaction log



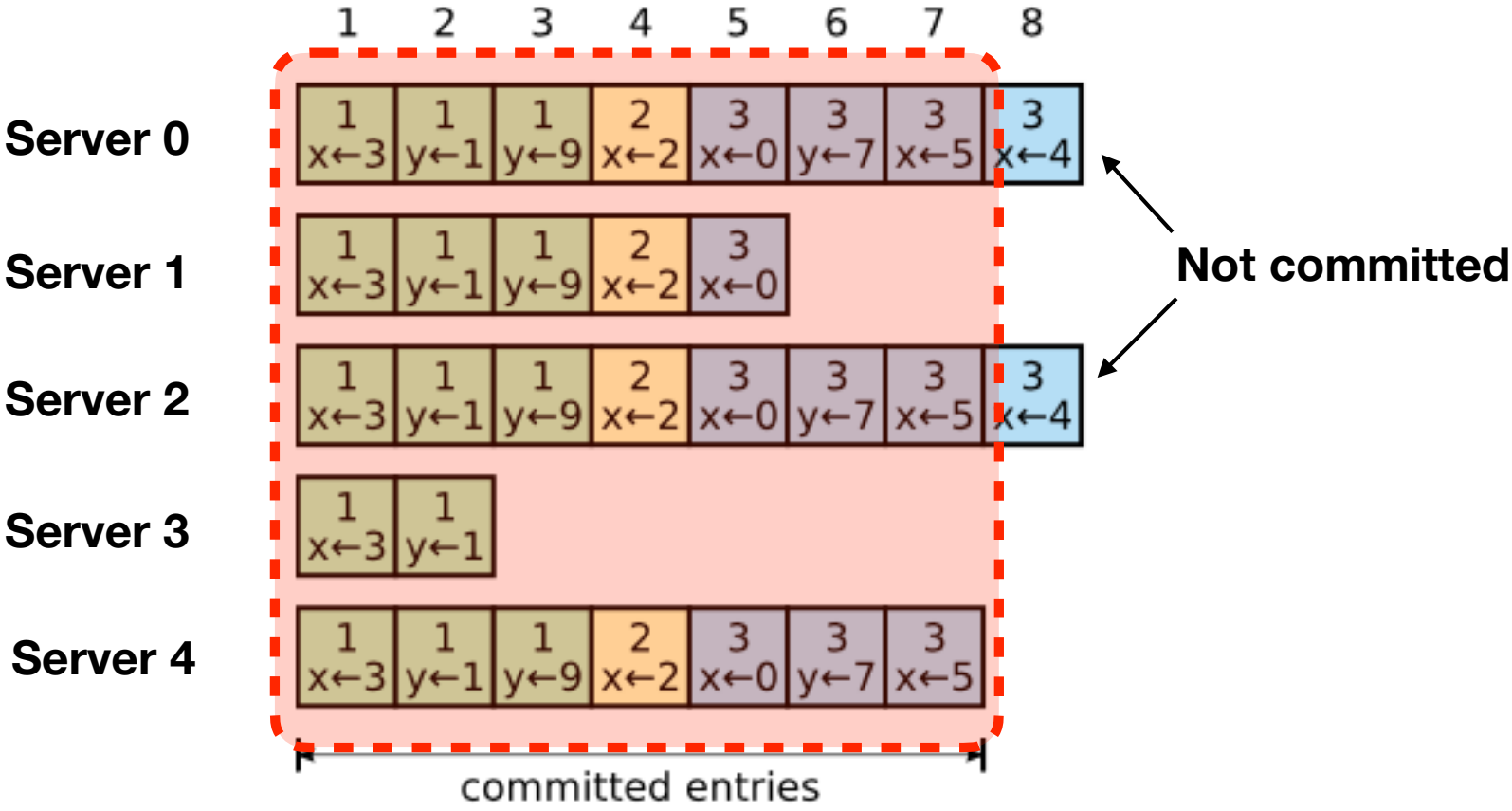
Replication

- Fault tolerance achieved by replicating the transaction log



Majority Rules

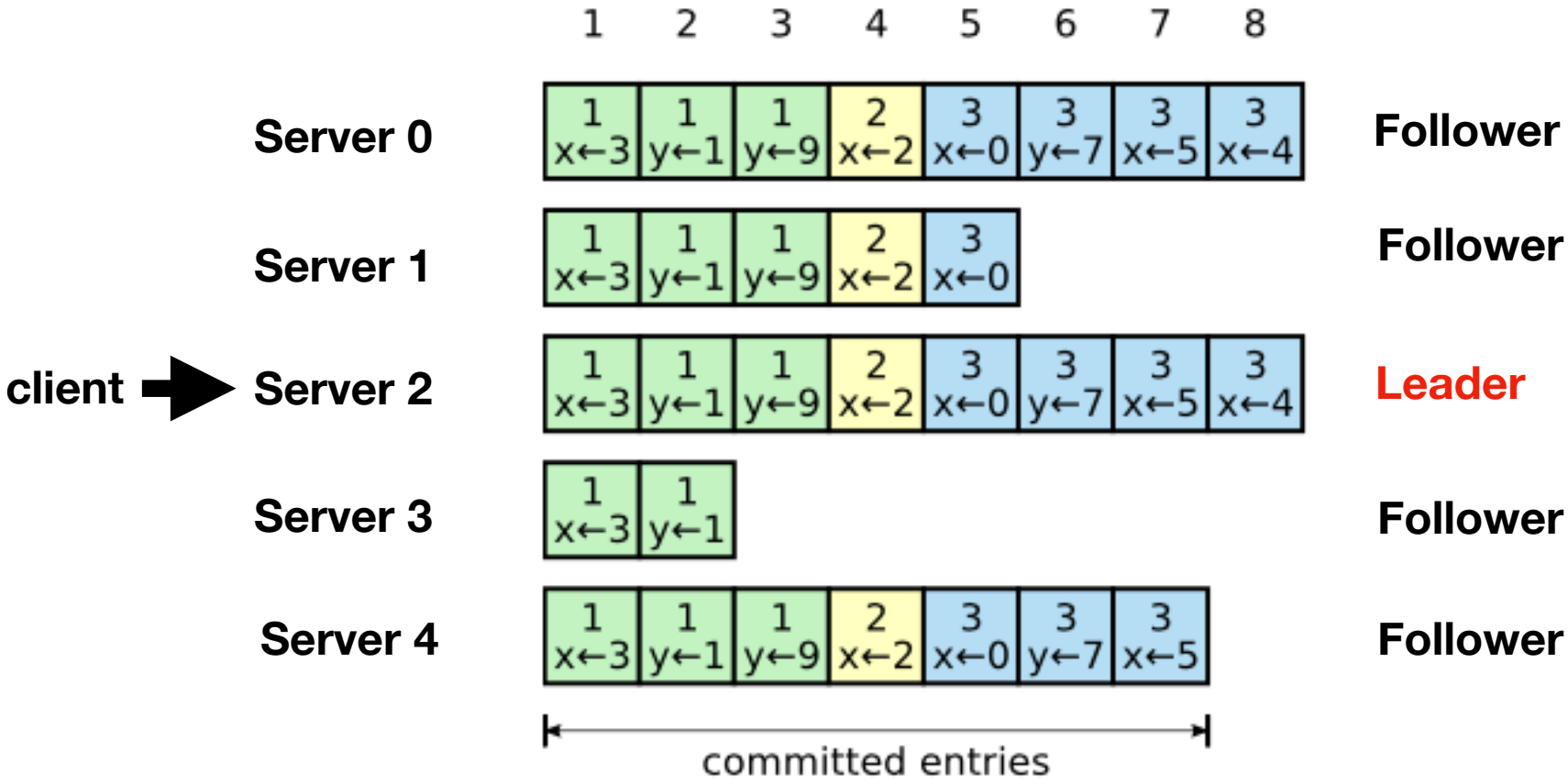
- Transactions are "committed" by achieving consensus



- Consensus means replication on a quorum of machines

A Strong Leader

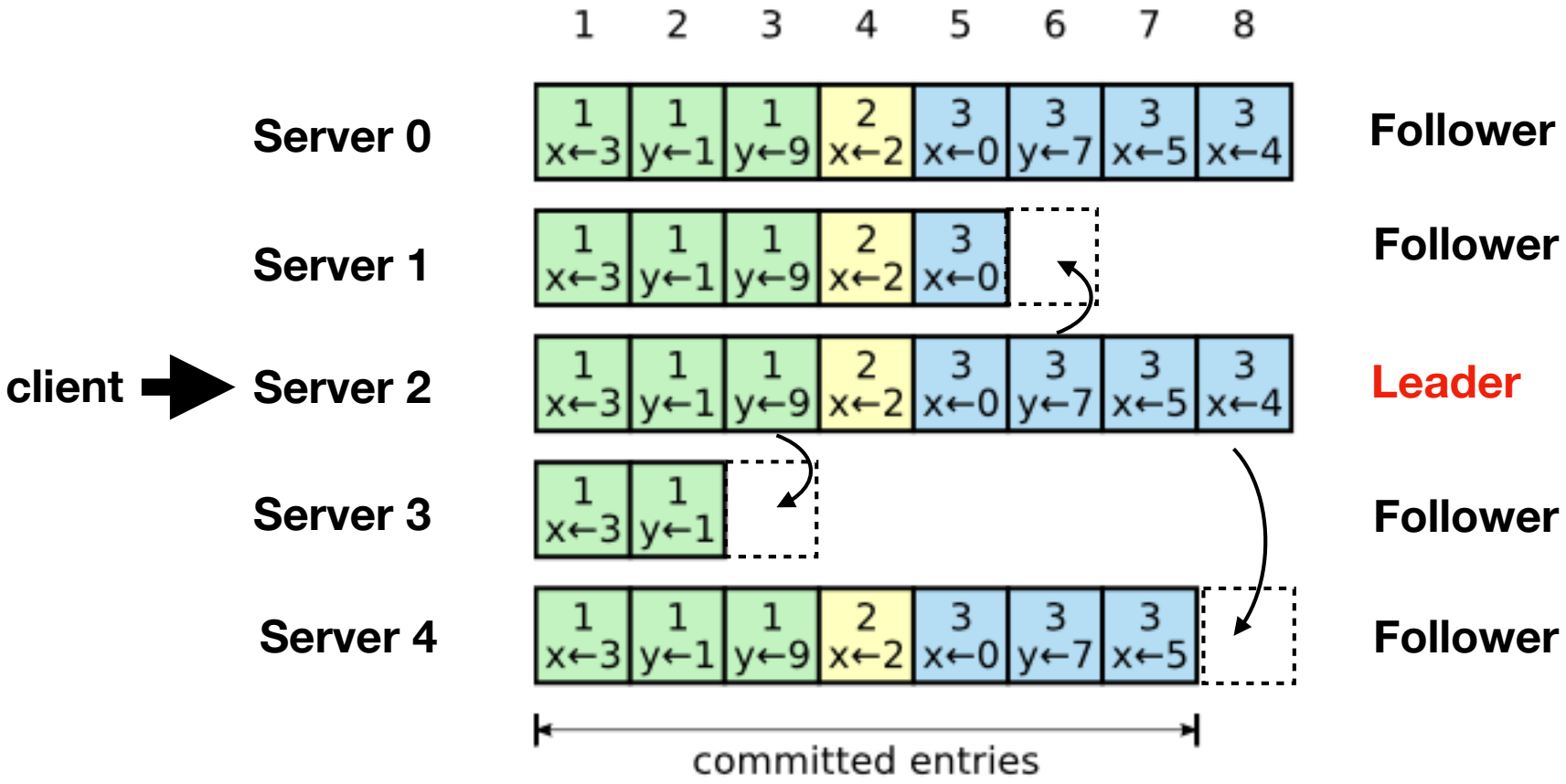
- All actions are coordinated by one and only one leader



- The leader is elected by all of the servers

Follower Update

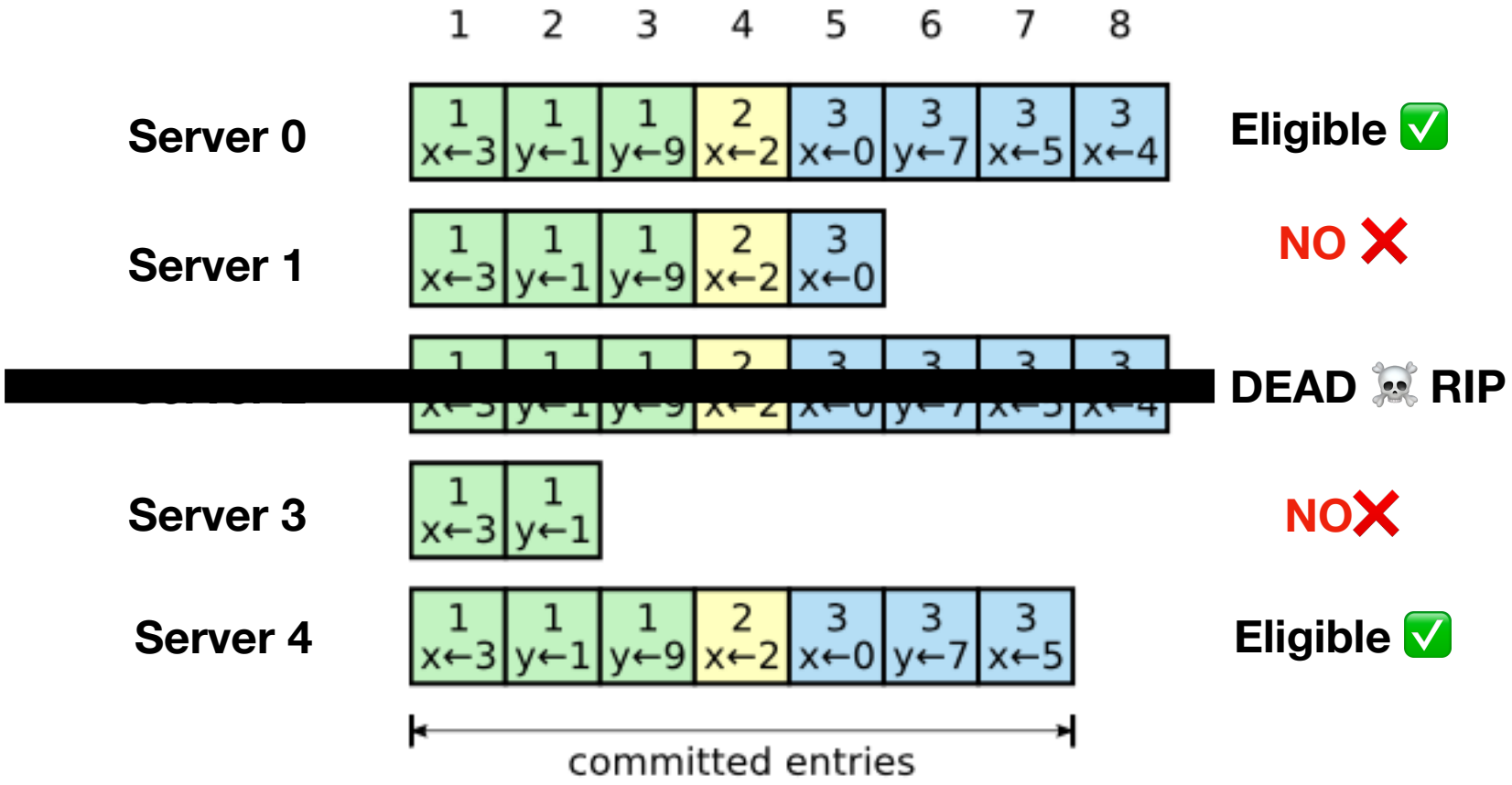
- The leader works to bring all followers up to date.



- But, there is a risk the leader could die (the whole point)

Leader Election

- If a server has a complete committed log, it can lead



- <u>Any</u> quorum always has at least one such member

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

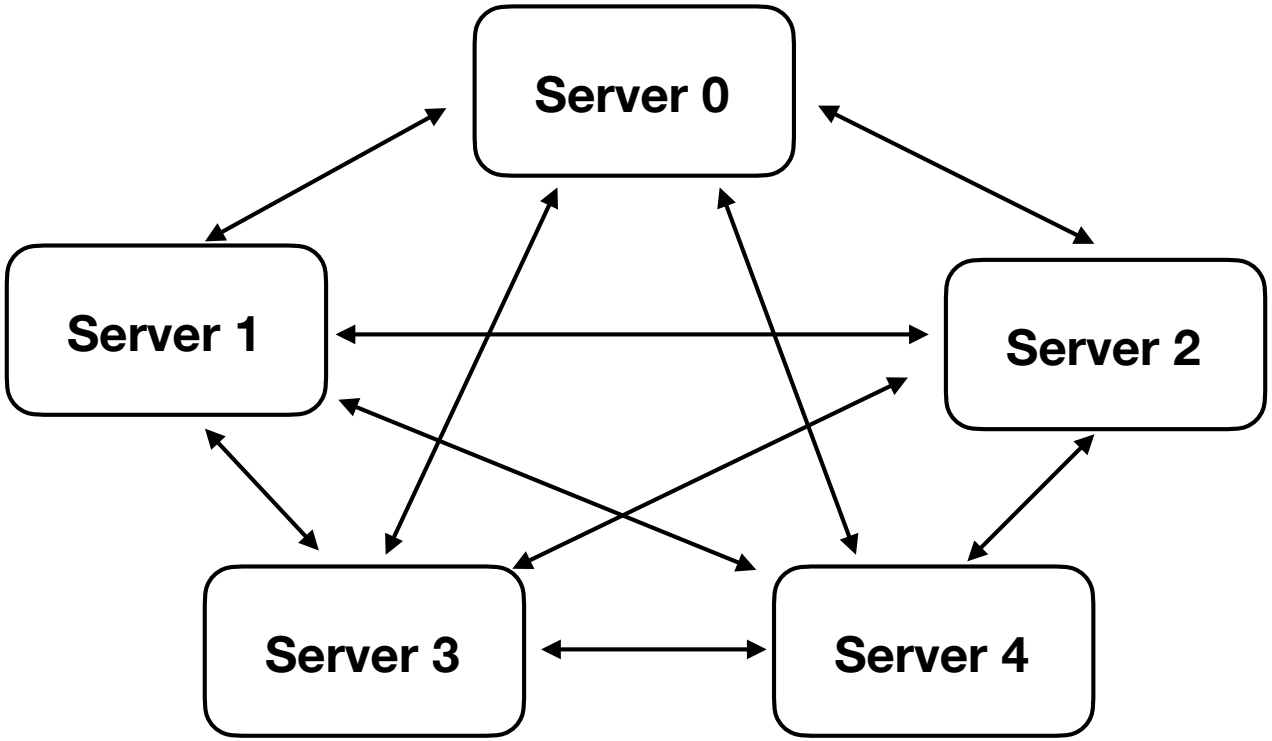
- We will start with some foundational topics
    - Technical: Sockets, messages, concurrency
    - Design: Problem modeling
- There will be a lot of open-coding (work on Raft)
- Key to success: TAKE. IT. SLOW.
    - Read/study the problem.
    - An hour of thinking is better than a day of debugging

Part 1

Foundations

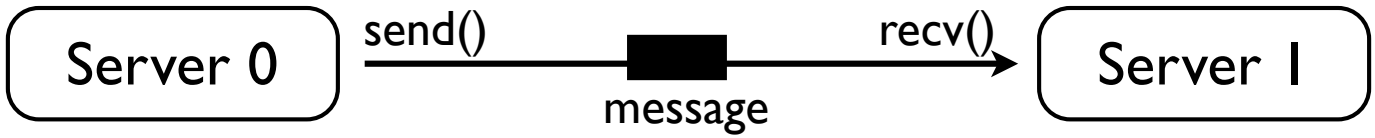
Raft and Networks

- Raft involves a cluster of identical servers



- They exchange messages over a network

Message Passing



- Servers run independently/concurrently
- Servers send and receive messages
- A message is a discrete packet of bytes
- Indivisible (treated as a single object)

Concurrency

- In production, servers are different machines
- Each server runs an independent process
- No shared state
- Only coordination is through messages

Communication

- Low level library: sockets
- Setting up a listener (server)

```
# Set up a listener

sock = socket(AF_INET, SOCK_STREAM)
sock.bind(("", 12345)
sock.listen(5)
client, address = sock.accept()
```

- Connecting as a client

```
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(("localhost", 12345))
```

Data Transport

- Receiving raw data on a socket

```
fragment = sock.recv(maxsize)
if not fragment:
    print("Connection Closed")
else:
    # Process message fragment
    ...
```

- Sending raw data on a socket

```
while data:
    nsent = sock.send(data)
    data = data[nsent:]

# Alternative
sock.sendall(data)
```

- Note: Both of these assume partial data (might have to assemble into a final message)

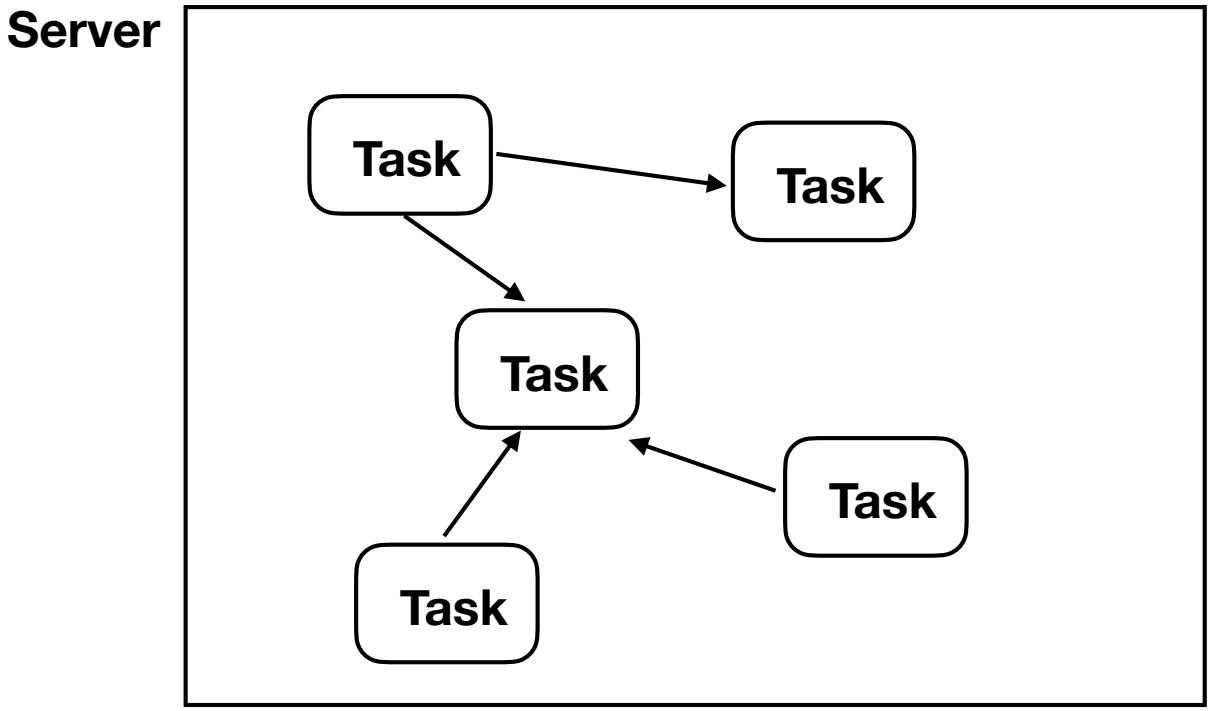
Message Encoding

- Common: size-prefixed byte vector

- No interpretation of the bytes (opaque)
- Payload could be anything (text, JSON, etc.)
- Messages are indivisible (no fragments)

Internal Concurrency

- Server may have many internal tasks that also 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.

Prefer Queues

- It is easier to program threads using queues
- Idea: Each thread is an "independent task"
- Threads coordinate by sending messages
- 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 1

- Implement messaging
- Implement a simple key-value store
- Implement the control software for a traffic light

Part 2

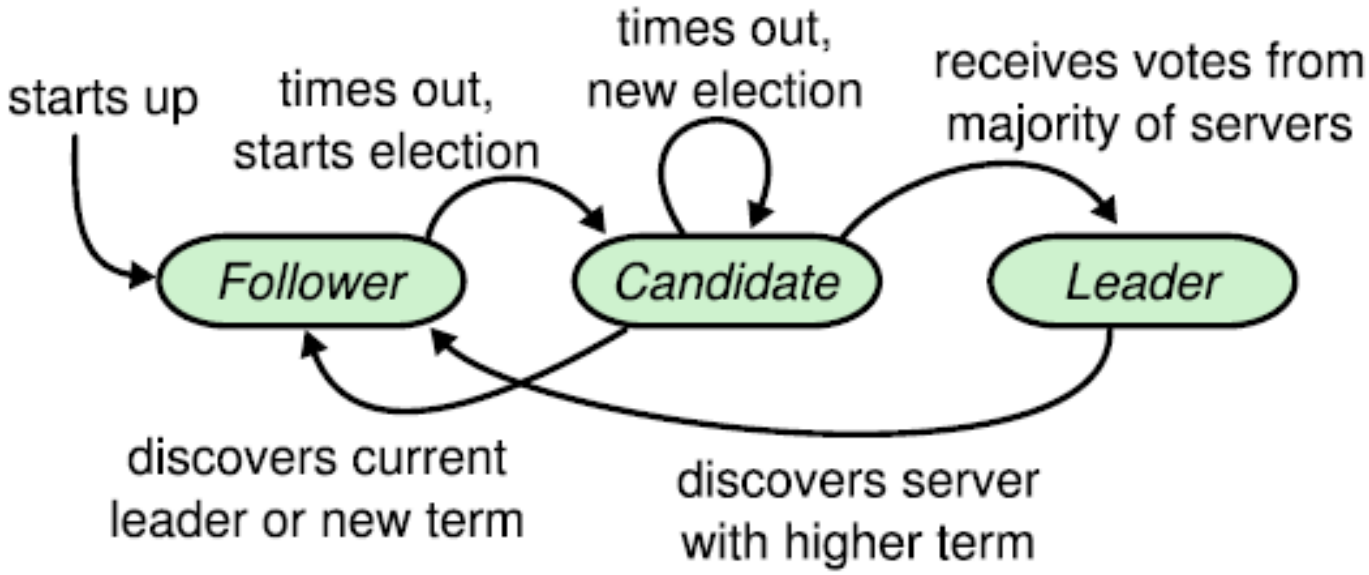
System Modeling

The Raft "System"

- Raft is a collective "system" of parts
  - Multiple servers
  - Messages between servers
  - Timers
  - Concurrency
- The entire system is driven by "events"

Raft Operational States

- Servers in Raft operate in different roles



- A major source of complexity concerns the transitions between roles and associated events

Runtime Environment

- The runtime environment is complex
- Example:
  - 5 independent servers/processes
  - Fully interconnected via sockets
  - Each server running 12+ threads
- Testing and debugging is punishing

Modeling/Simulation

- How to tame the system complexity?
- One approach: Start by implementing a "simulation" or "model" of Raft
- Single process, no system dependencies (i.e., threads, sockets, timers, etc.)
- But, what does that look like?

Project 2

- How to eliminate system dependencies?
- Refactor the control software for the "traffic light" to have no system dependencies
- Make it something that allows for isolated testing/experimentation

Part 3

Model Checking

Thought

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

Systems as State Machines

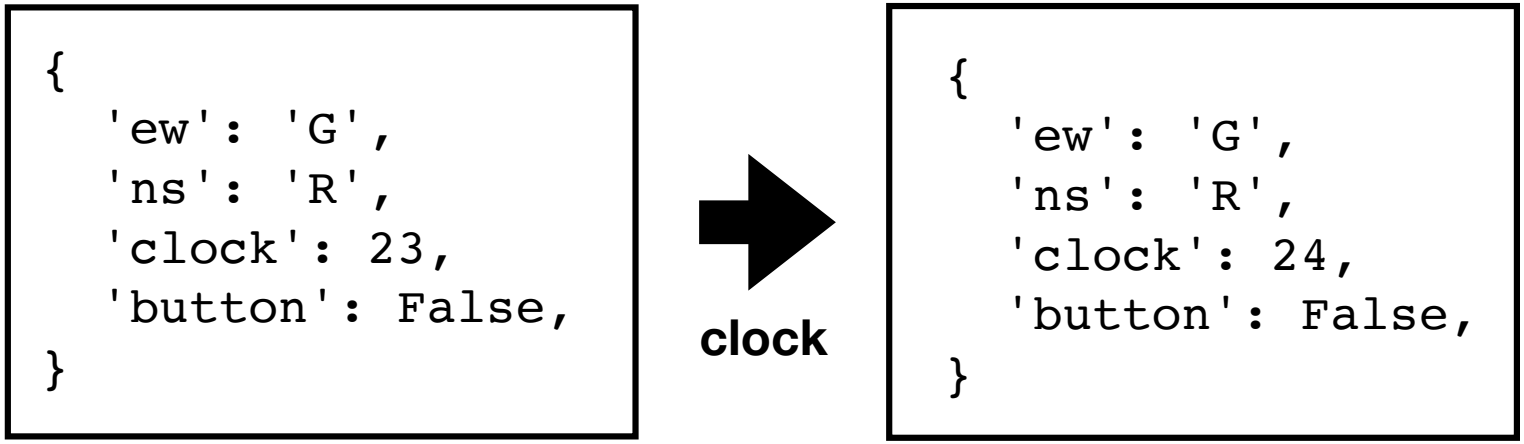
- System is comprised of "state"

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

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

Sequencing

- State machines execute via events

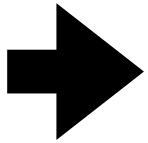


- Events cause a state change

Modeling State Changes

- A state change is an update to the values

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



```
new = {
   'ew': 'G',
   'ns': 'R',
   'clock': 24,
   'button': False,
}
```

- It's a function: state -> state

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

- New state is old values + updated values

Expressing 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)
    ...
```

Expressing a State Machine

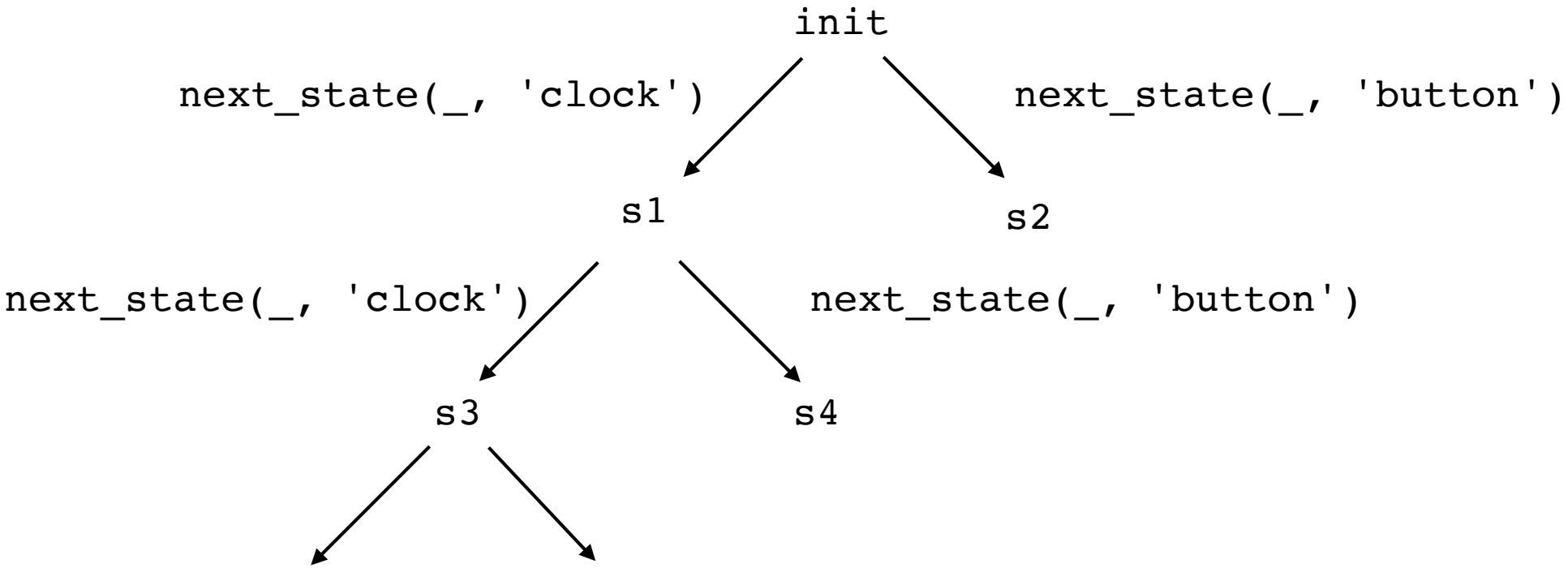
- Example:

```
>>> 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 3

- Can you modify the traffic light code so that it's testable 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 (∨)
~A            \* NOT (¬)
```

- Grouping by indentation (implies parens)

```
/\ a
/\ b
/\ \/ c > 10 /\ d = 0
   \/ c > 20 /\ d = 1
/\ e
```

```
(a
 and b
 and ((c > 10 and d ==  0)
      or (c > 20 and d == 1))
 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*

Working with Multiples

- If working with multiple things, use tuples

```
Lights == 1..5              \* 5 Traffic lights

Init == /\ ew = [ n \in Lights |-> "G" ]
        /\ ns = [ n \in Lights |-> "R" ]
        /\ clock = [n \in Lights |-> 0 ]
        /\ button = [n \in Lights |-> FALSE ]

Next(i) == \/ /\ ew[i] = "G"
              /\ ns[i] = "R"
              /\ clock[i] < 30
              /\ clock' = [clock EXCEPT ![i] = clock[i] + 1 ]
              /\ UNCHANGED <<ew, ns, button>>

           \/ /\ ew[i] = "G"
              /\ ns[i] = "R"
              /\ clock[i] = 30
              /\ clock' = [clock EXCEPT ![i] = 0]
              /\ ew' = [ew EXCEPT ![i] = "Y"]
              /\ UNCHANGED <<ns, button>>
```

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?

Interlude

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
    - Need a "log"
    - Need messaging.
- So, we're going to start with that.

The Path Forward

- Step 1: The transaction log
- Step 2: Message passing
- Step 3: Log Replication (multiple servers)
- Step 4: Consensus/Application interface
- Step 5: Leader Election
- Step 6: Timing
- Step 7: Systems programming

Emphasis on Modeling

- Our approach is going to emphasize modeling and simulation as a first step
- Goal is to understand the algorithm.
- We'll work out bugs in our understanding.
- Later: Will introduce systems elements

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.

The Log is (is not?) a List

- Is the transaction log just a list?

```
log = []

log.append(entry1)
log.append(entry2)
...
```

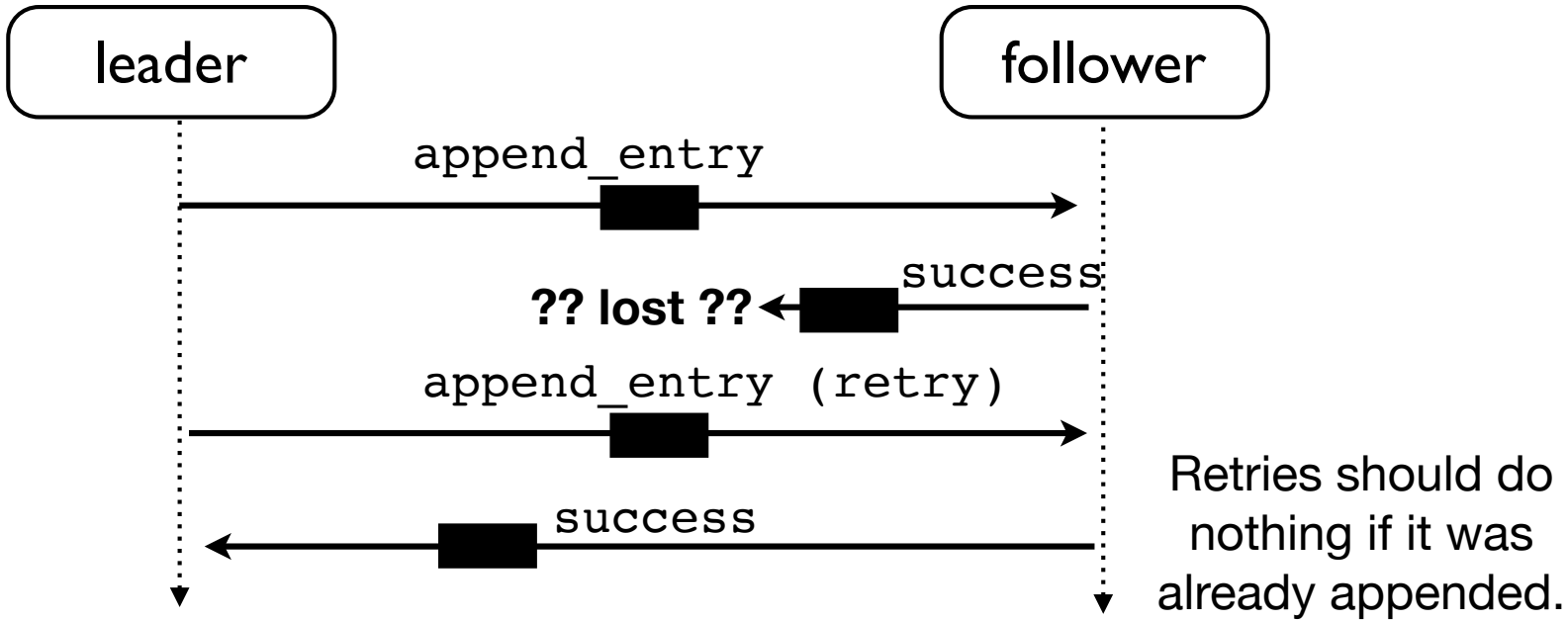
- Answer: It's complicated.
- Conceptually, it's a list, but it needs to be a "fault-tolerant" list.

Problem: Idempotence

- Appends must be "idempotent"

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

- The issue: Network faults/retries



Solving Idempotence

- Log appends always specify an exact position where the entry gets placed

```
def append_entry(log, index, entry):
    if len(log) > index:
        log[index] = entry
    elif len(log) == index:
        log.append(entry)
    else:
        raise IndexError("No holes!")
```

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

Log Matching Property 1

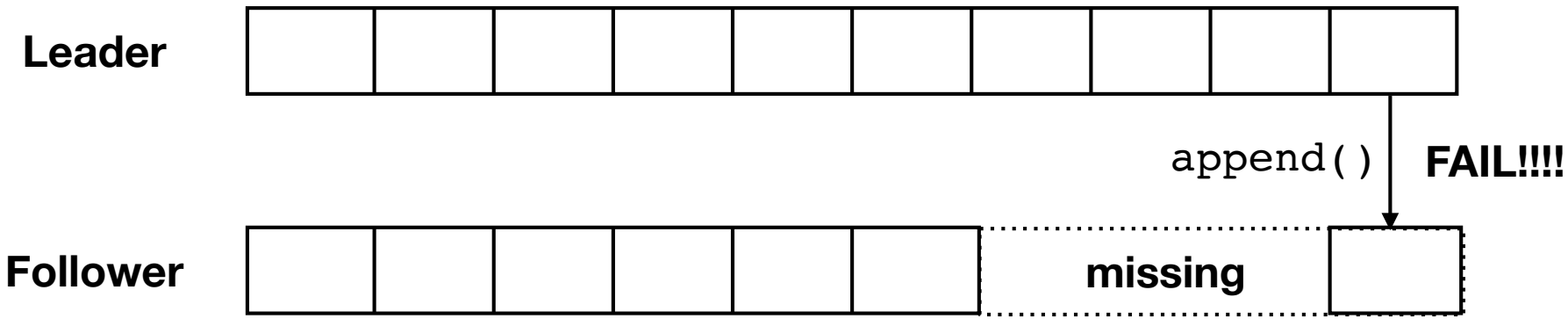
- The log entry stored at a given index must be stored at the same index in all logs

```
if i == j:
   assert log1[i] == log2[j]
```

- Critically important: Logs in Raft are supposed to identical across all servers.
- The algorithm works to enforce this.

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.

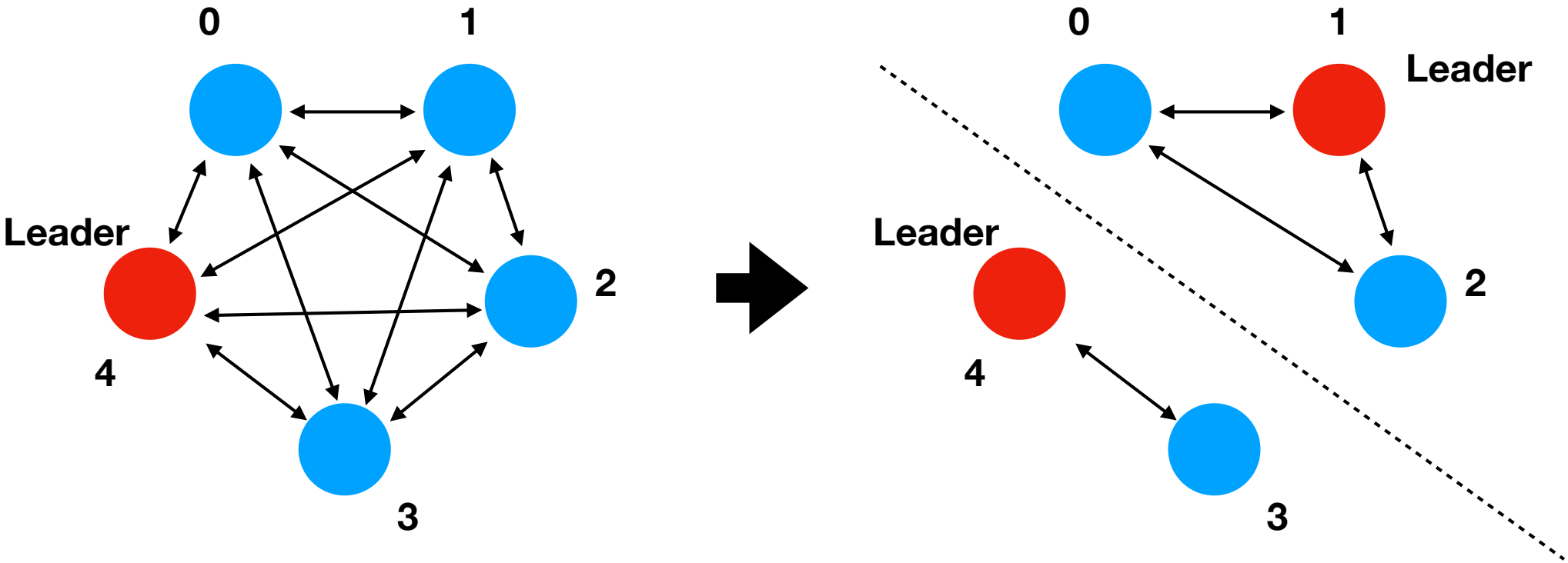
Solving Gaps

- Log appends must be allowed to fail

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

>>> log = [ 'x', 'y' ]
>>> append_entry(log, 10, 'z')
False
>>> append_entry(log, 2, 'z')
True
>>> log
['x', 'y', 'z']
>>>
```

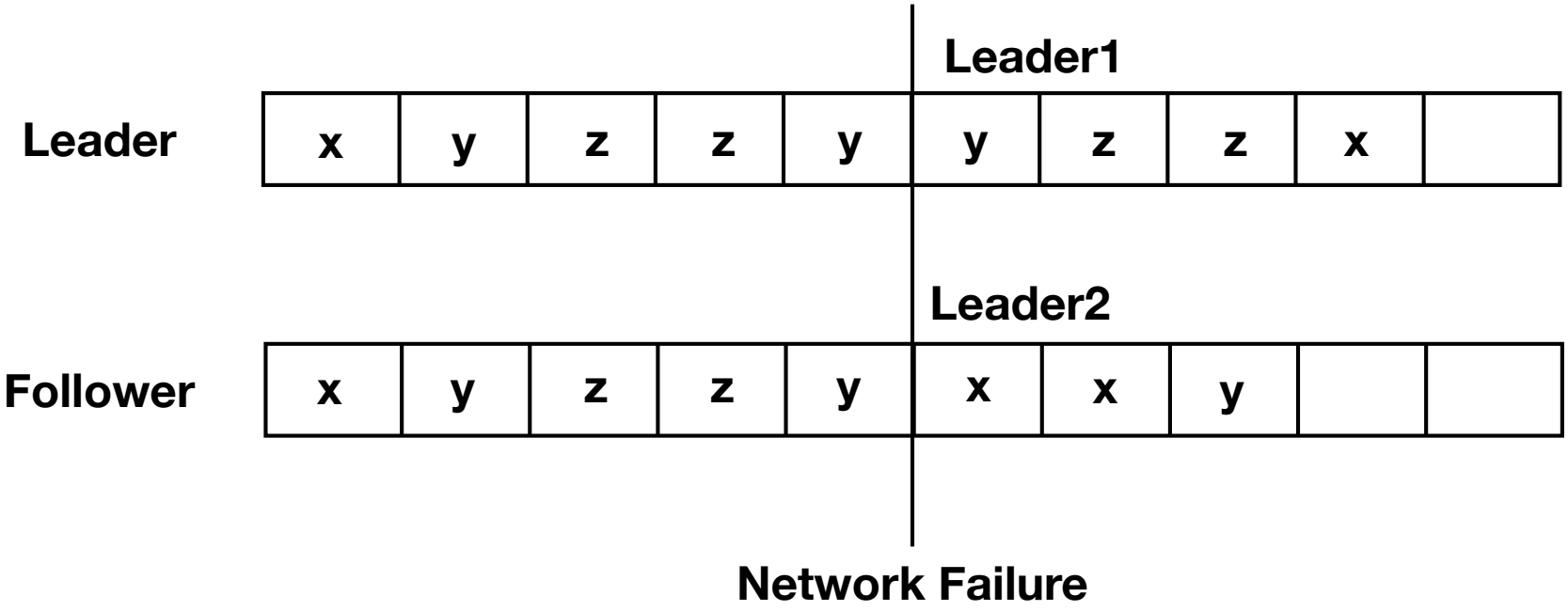
Problem: Divergence



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

Problem: Divergence

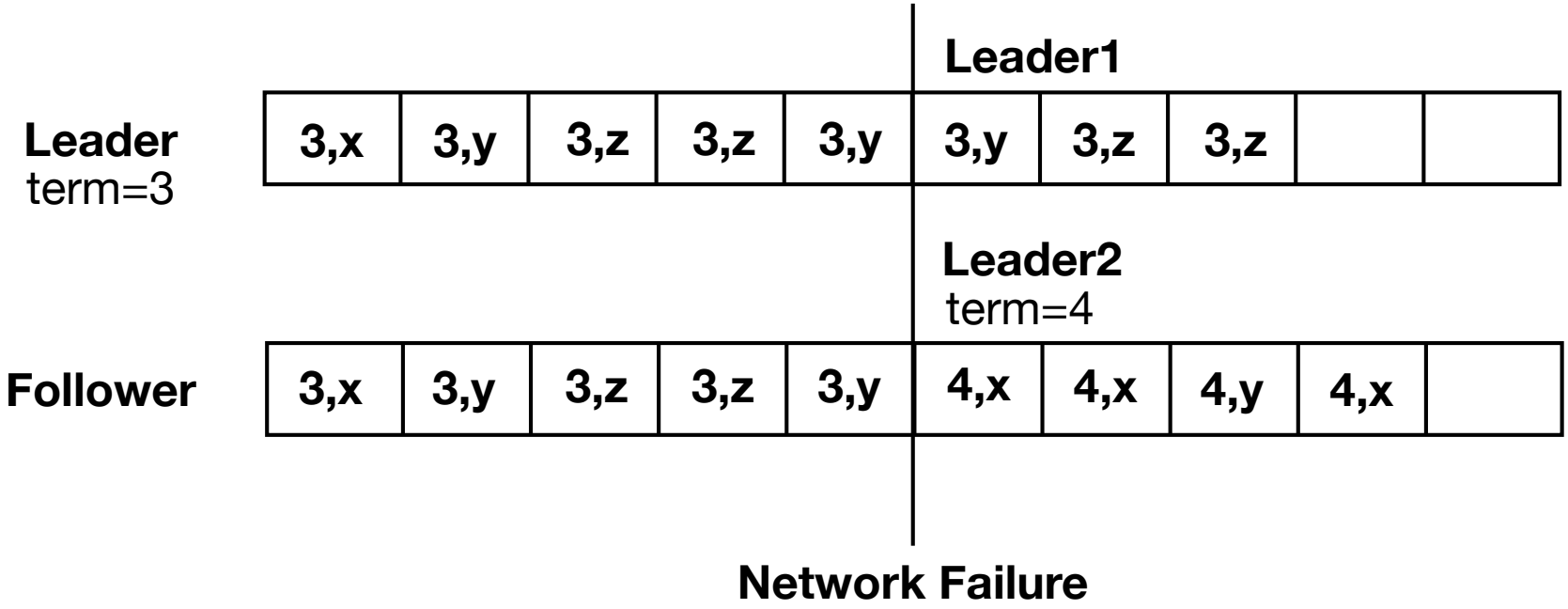
- Two "leaders" might cause a log divergence



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

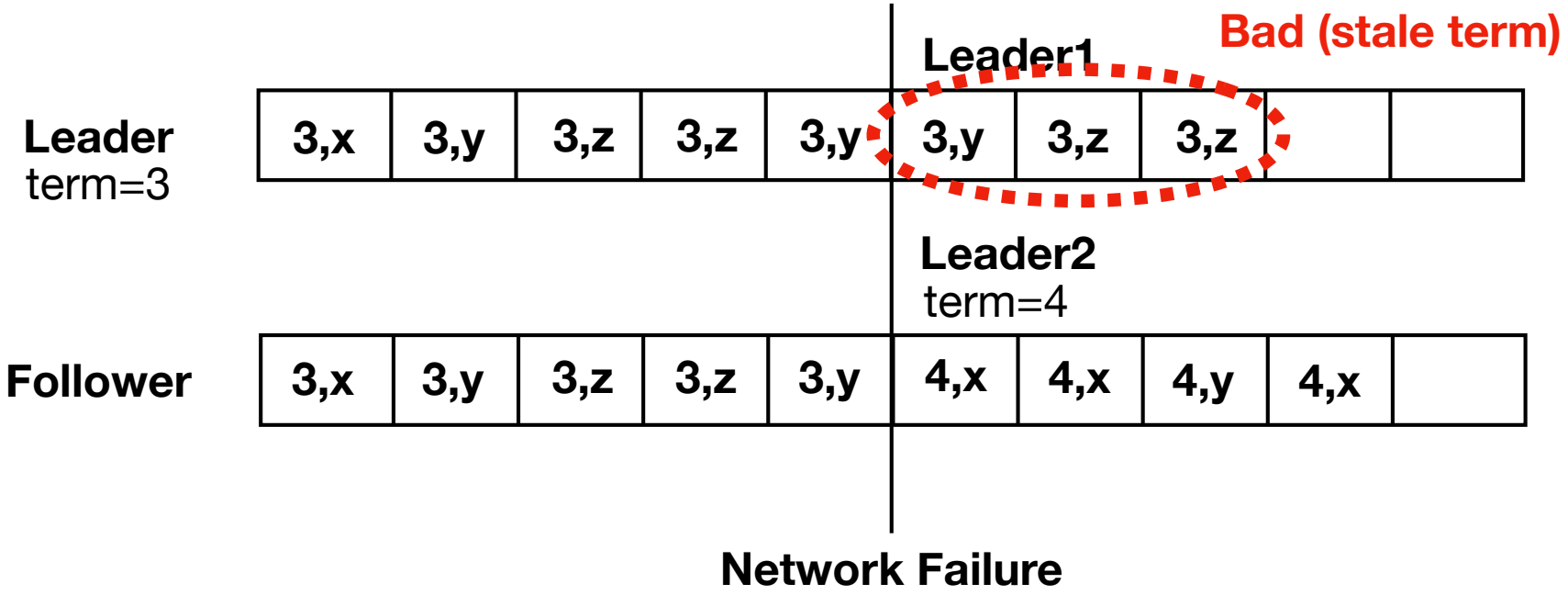
Solution: Terms

- 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
- Critical: Bad entries are never "committed"



Log Matching Property 2

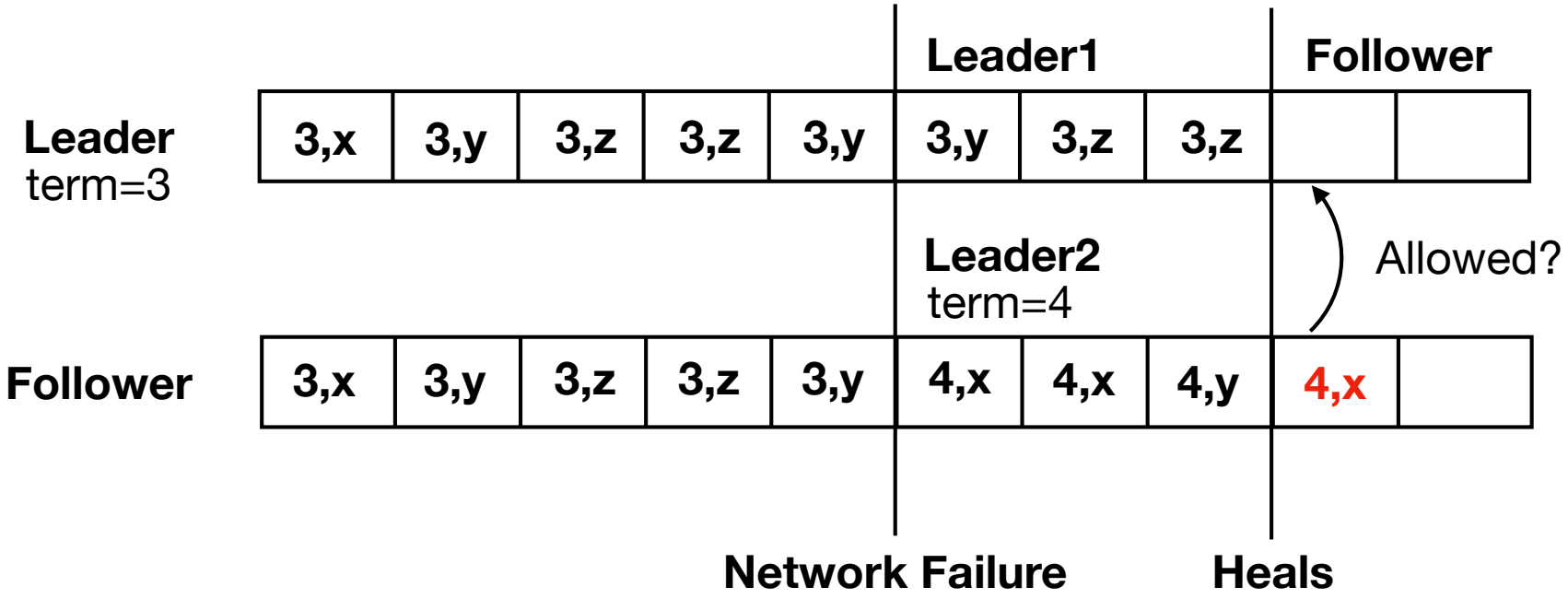
- If entries in different logs have the same index and term, then the logs must be identical in all preceding entries

```
if i == j:
   assert all(log1[n] == log2[n] for n in range(i))
```

- Critically important: This is a constraint that the log must enforce at all times.

Log Matching

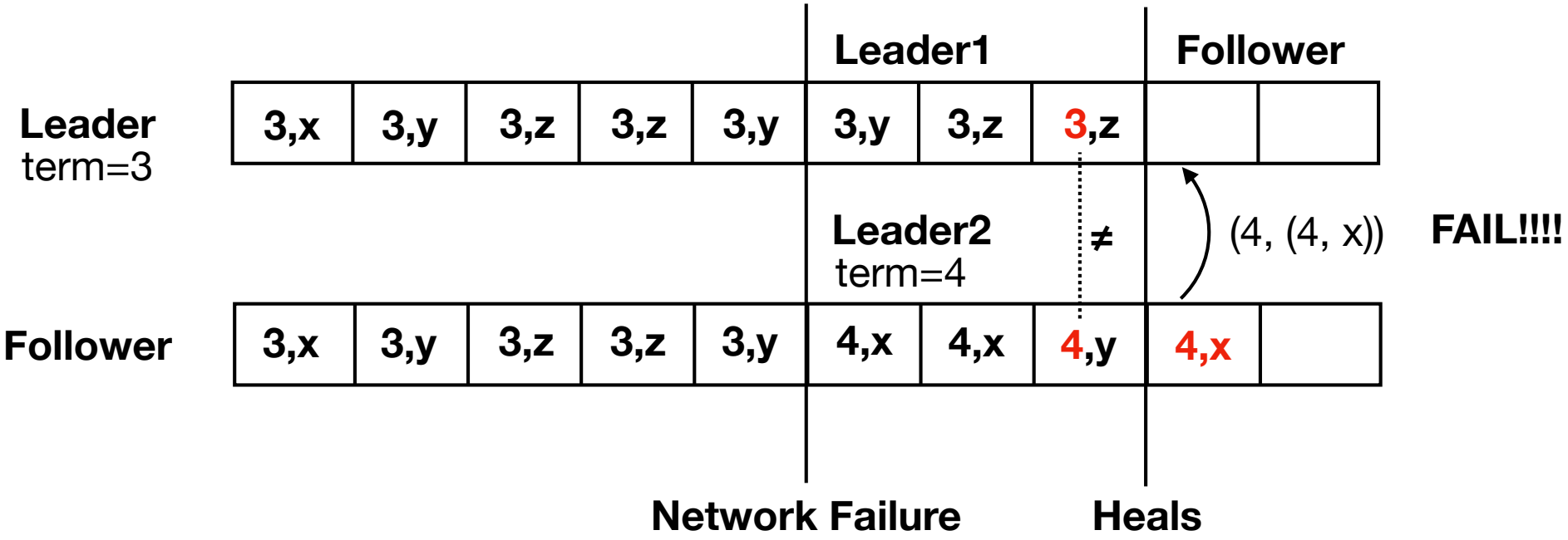
- The log is never allowed to have gaps, but consider this tricky scenario after healing



- 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

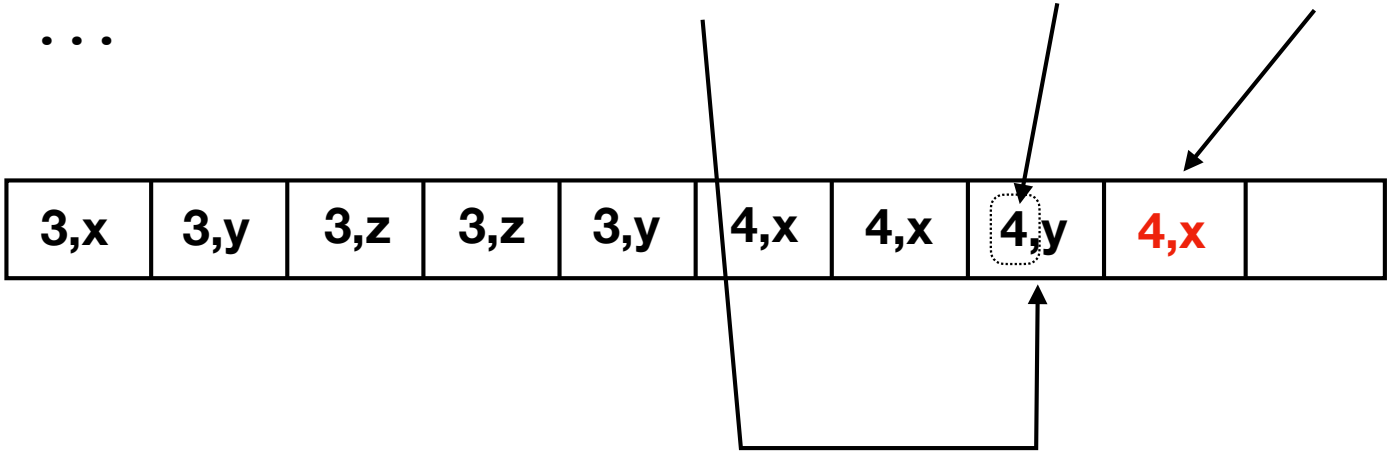


- This enforces "log continuity" (current leader will detect bad entries from a wayward leader)

Solving Continuity

- Log appends must link to prior entry

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



- You 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 is basically a list append with conditions
  - Idempotent (repeated attempts ok)
  - No gaps/holes
  - Must enforce the log matching properties
- Allows multiple entries to be appended

Project 4

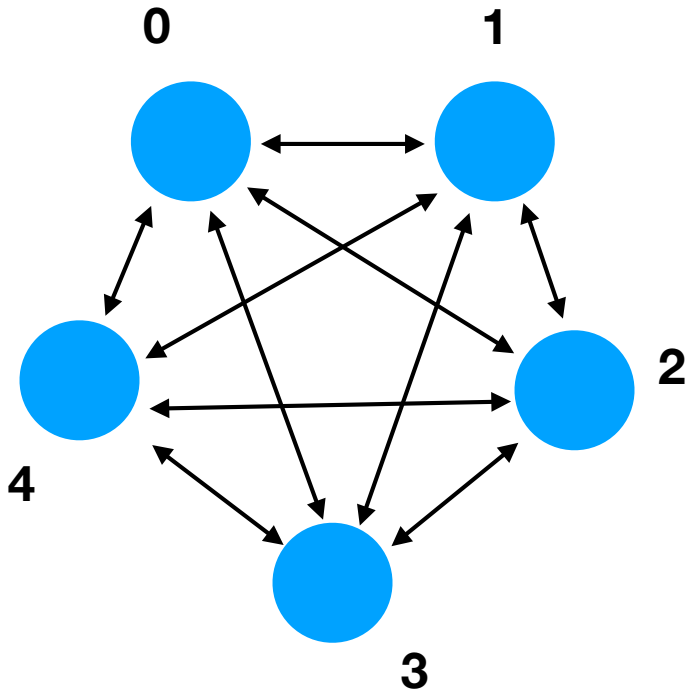
- Implement the Raft log as a stand-alone object
- Write tests to verify its behavior
- Should be usable on its own--no dependency on the network or any other part of Raft
- If appending to the log is broken, then everything is broken. It's critical for this part to be absolutely correct.

Part 5

The Raft Network

Raft Networking

- It is a cluster of interconnected servers



- Usually an odd number. Known in advance.
- Number each server (0-4).

Raft Networking

- Create a high-level abstraction layer

```
# Create the Raft network
net = RaftNetwork(5)   # A cluster of 5 nodes

# Create a node
node0 = net.create_node(0)

# Send a message to other servers
node0.send(1, "Hello from 0")
node0.send(2, "Hello from 0")

# Receive a message (from anyone)
msg = node0.receive()
```

- Allow messages to be sent
- Allow messages to be received

Raft Networking

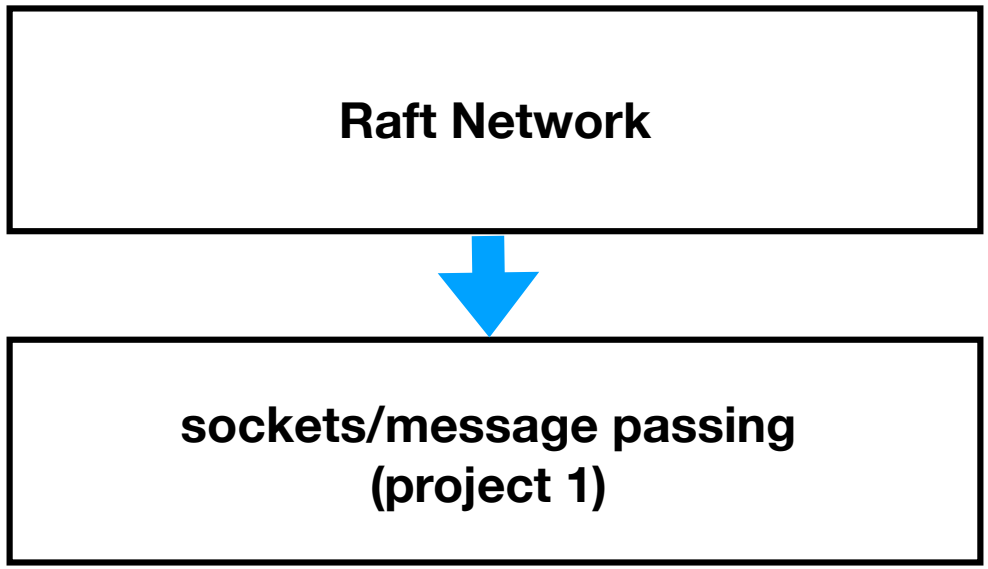
- Important features of Raft messaging
  - Message delivery is asynchronous (the sender does NOT wait for a receiver to actually get the message)
  - Messages can be lost or dropped. For example, if a server is offline. Messages are NOT queued. Throw them away.
  - Every server is connected to every other server (fully interconnected).

Raft Networking

- Debugging/Development Considerations
  - Instrument your networking layer with controls that can be used to simulate network failure
  - Example: simulating a dead link, delayed messages, partitioned networked, etc.
- Have debug logging
- Think about testing

Abstraction Layers

- Eventually the Raft network will sit on top of low-level socket/messaging code



- We're not doing that to start. We're going to create a simulated network instead.

Project 5

- Work on the Raft network abstraction
- Model/simulation that allows messaging
- Should be independent of system layer (no sockets, no threads, etc.).

Part 6

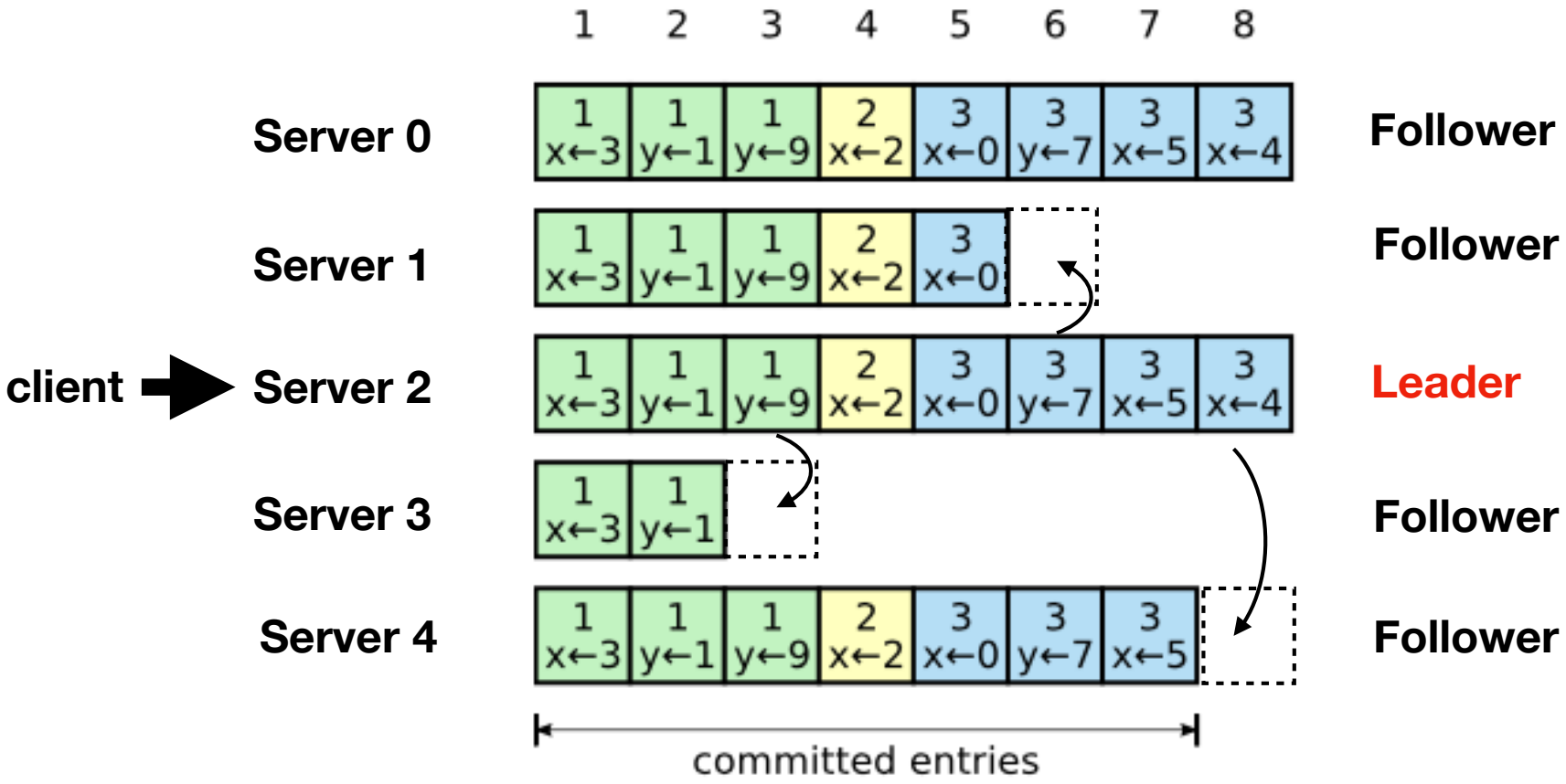
Log Replication

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

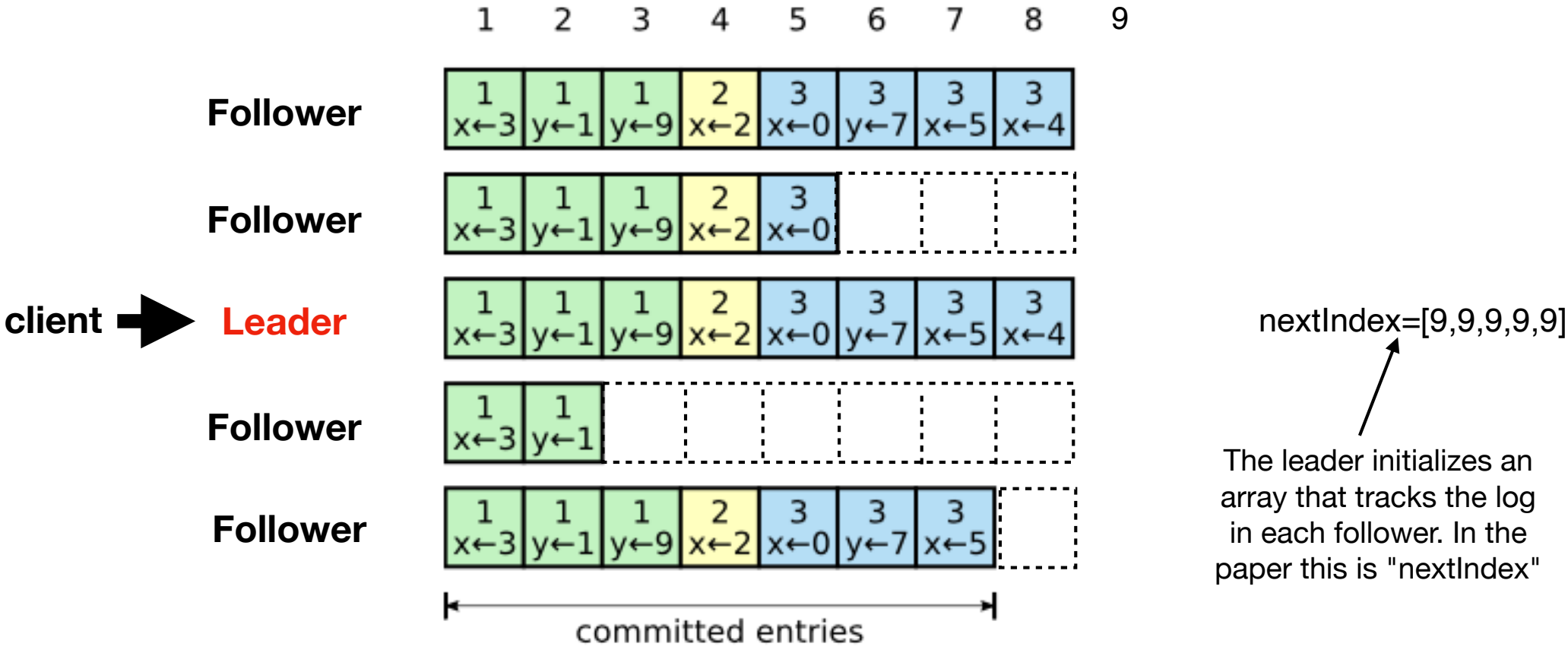
- Followers are passive.
- They receive messages from the leader and respond to the leader. NEVER with each other.
- If there is no leader, a new leader is elected from one of the followers (later).

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

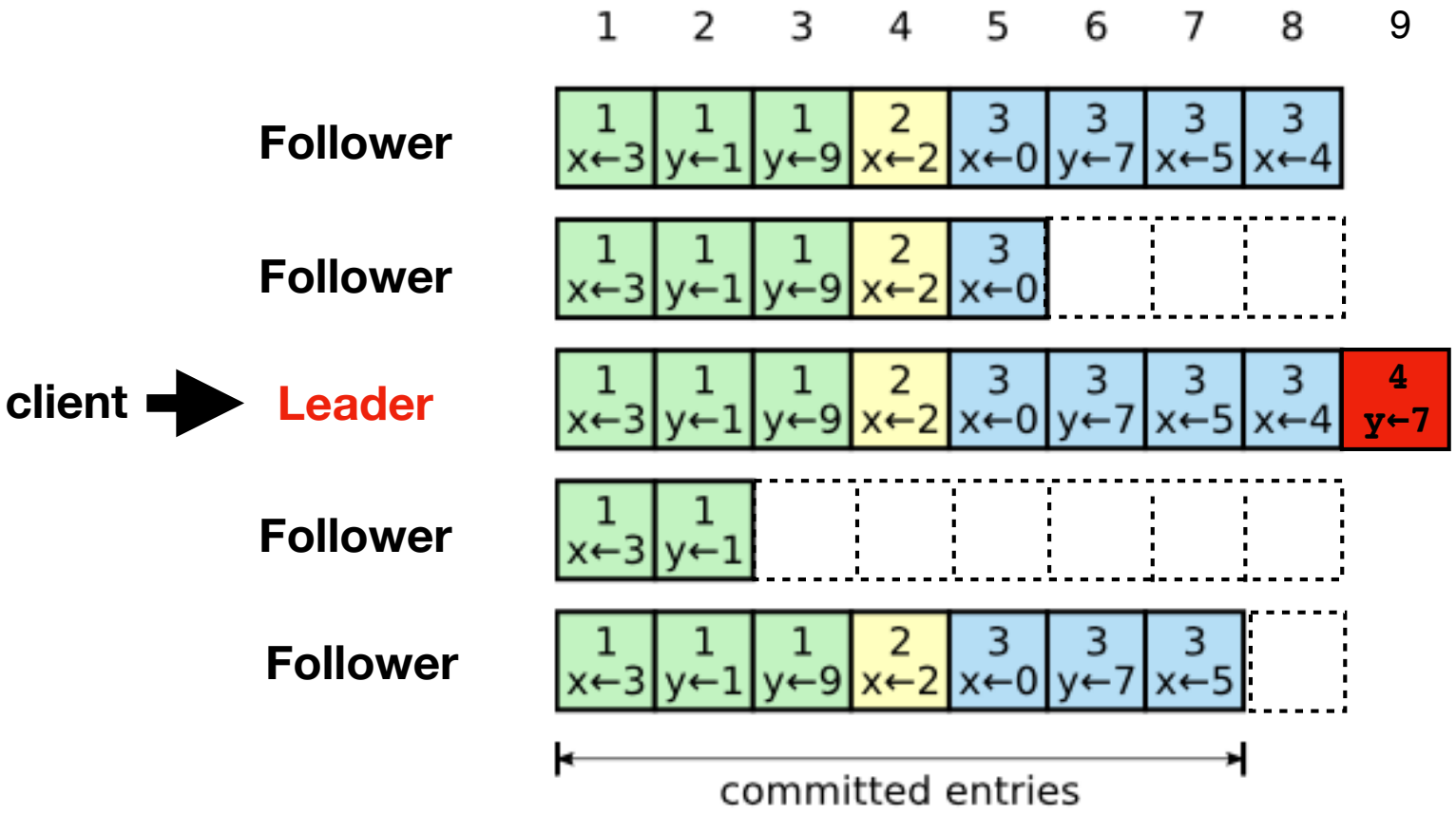
- Leaders start by assuming that all followers have the entire log (same information as leader)



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

New Appends

- New entries go on end of leader log

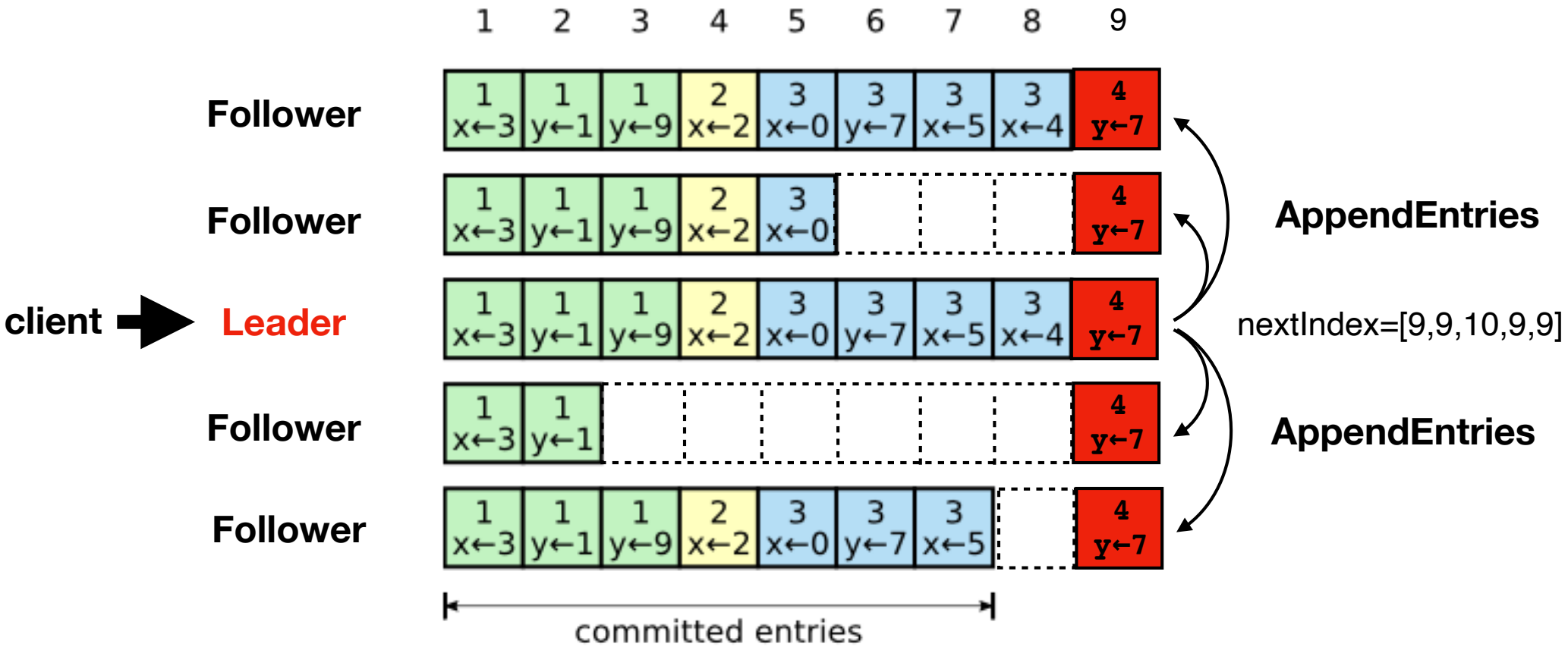


nextIndex=[9,9,10,9,9]

The array keeps track of where new log entries get placed.

Append Replication

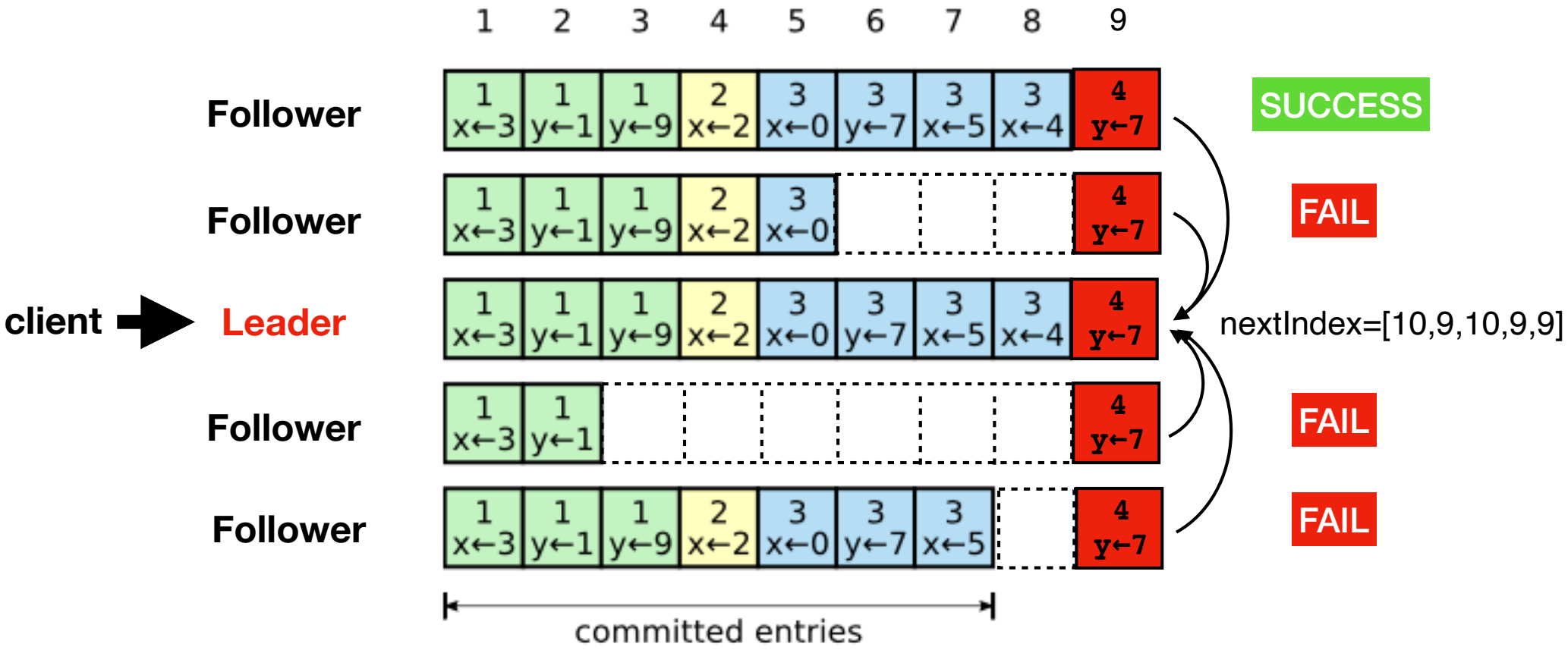
- Leader replicates to the followers



- This is a network message

Response Handling

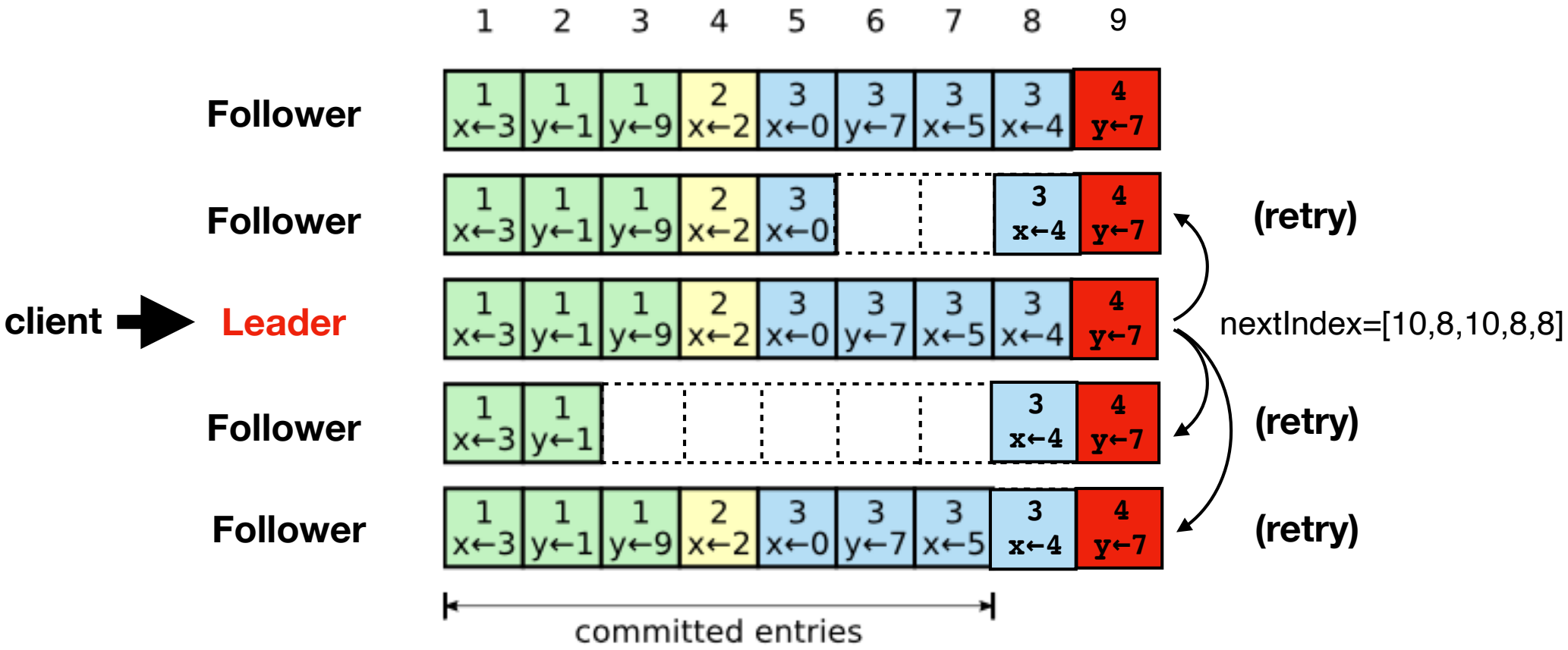
- AppendEntries will succeed or fail (no gaps)



- This is reported back (AppendEntriesResponse)

Backtracking

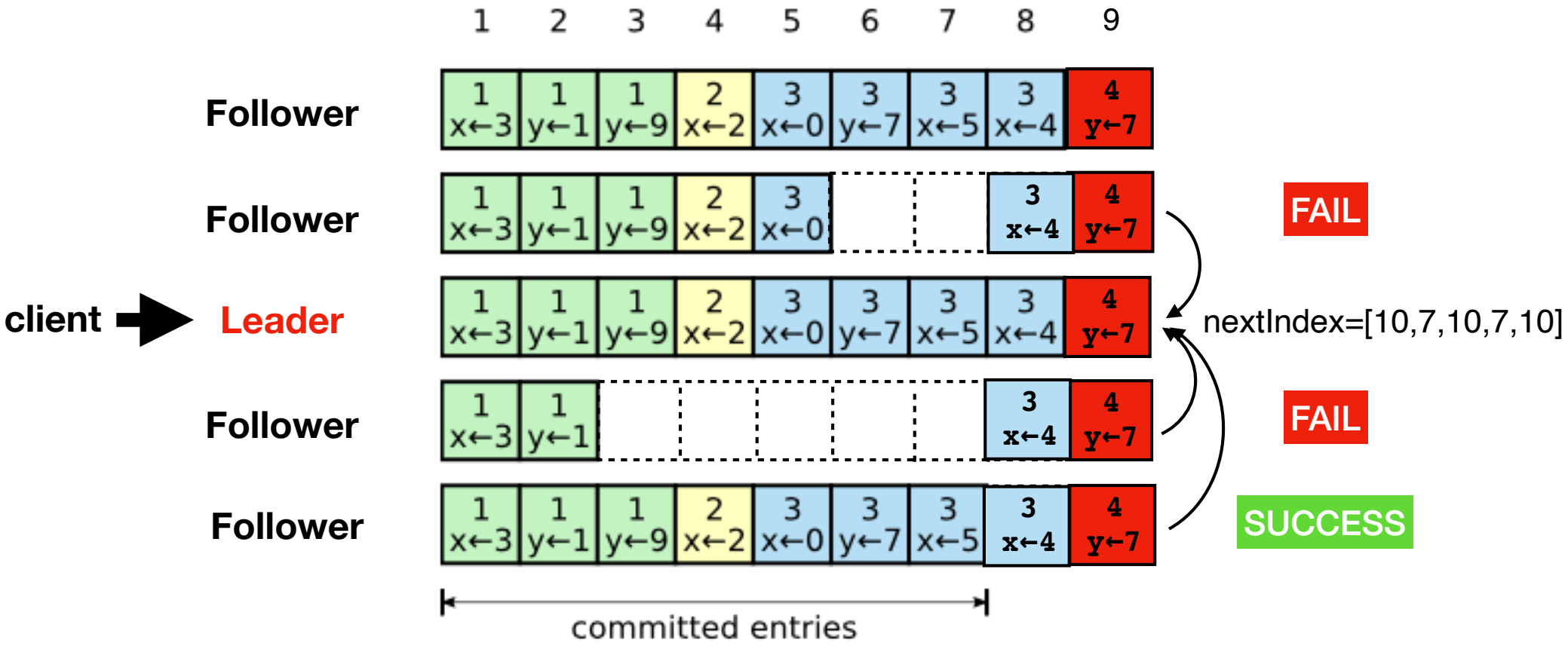
- Leader retries failures with earlier log entries



- Again, followers report back with success

Backtracking

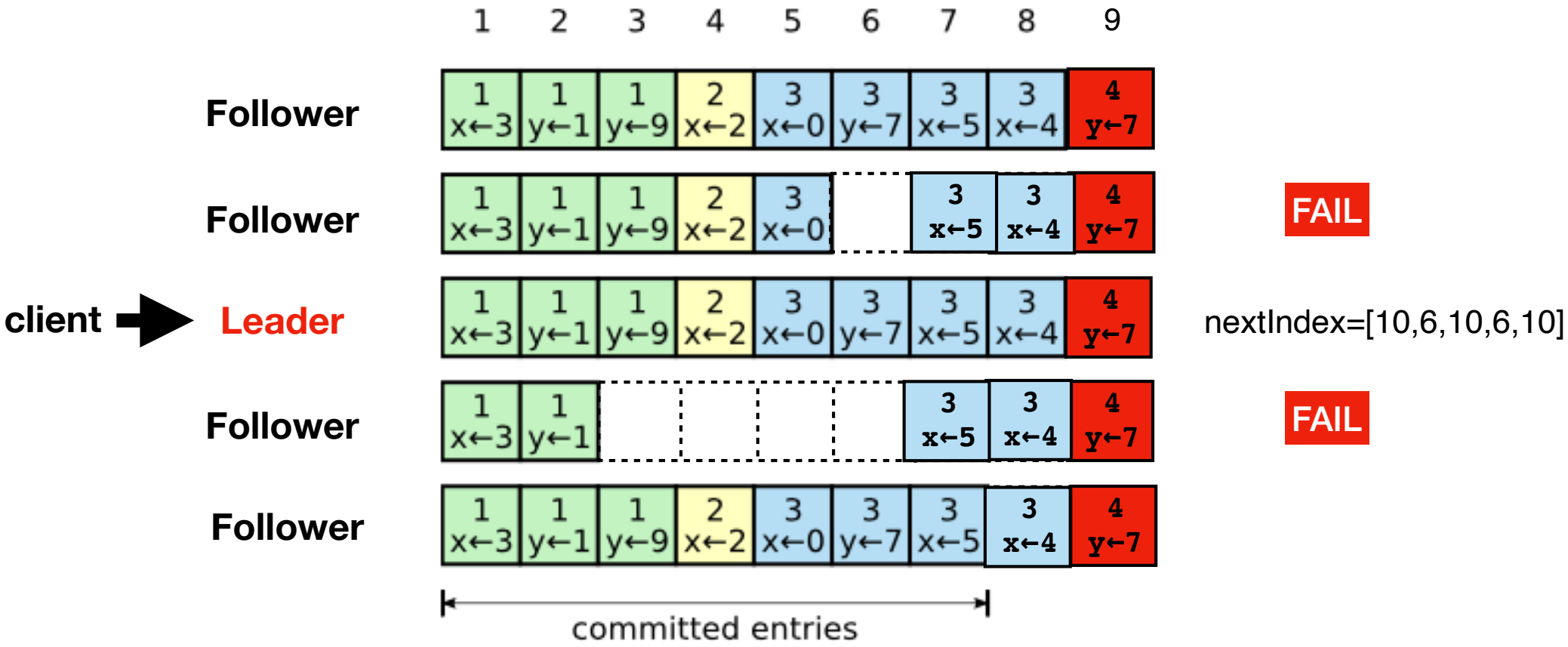
- This process repeats itself



- Leader keeps working backwards until success

Backtracking

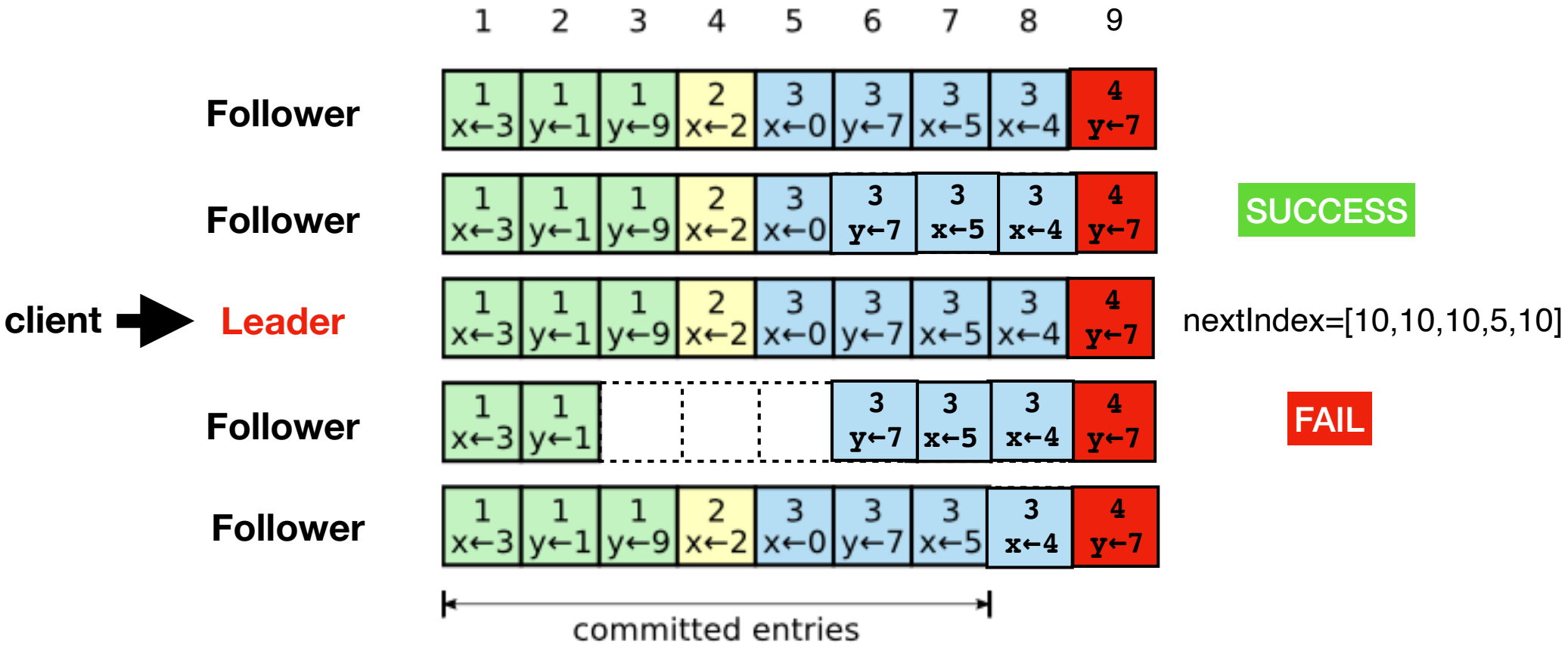
- This process repeats itself



- Leader keeps working backwards until success

Backtracking

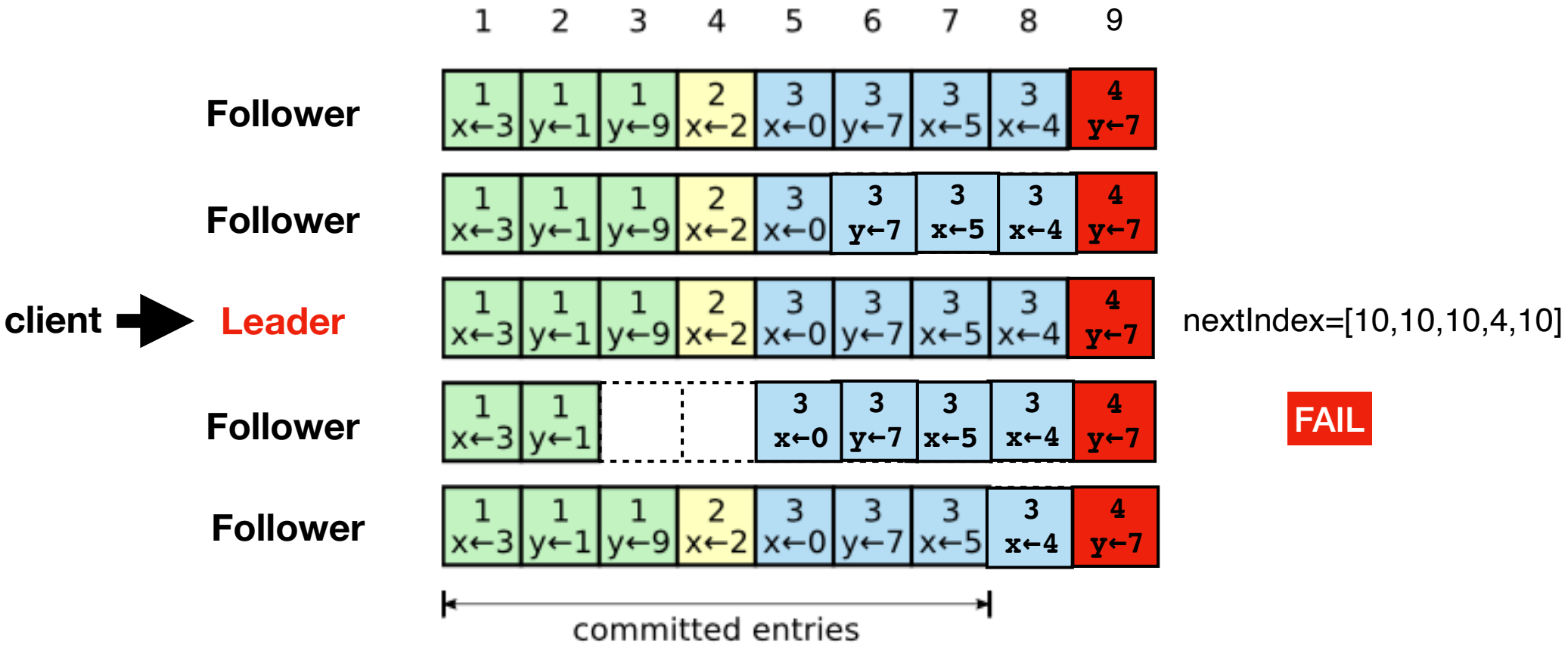
- This process repeats itself



- Leader keeps working backwards until success

Backtracking

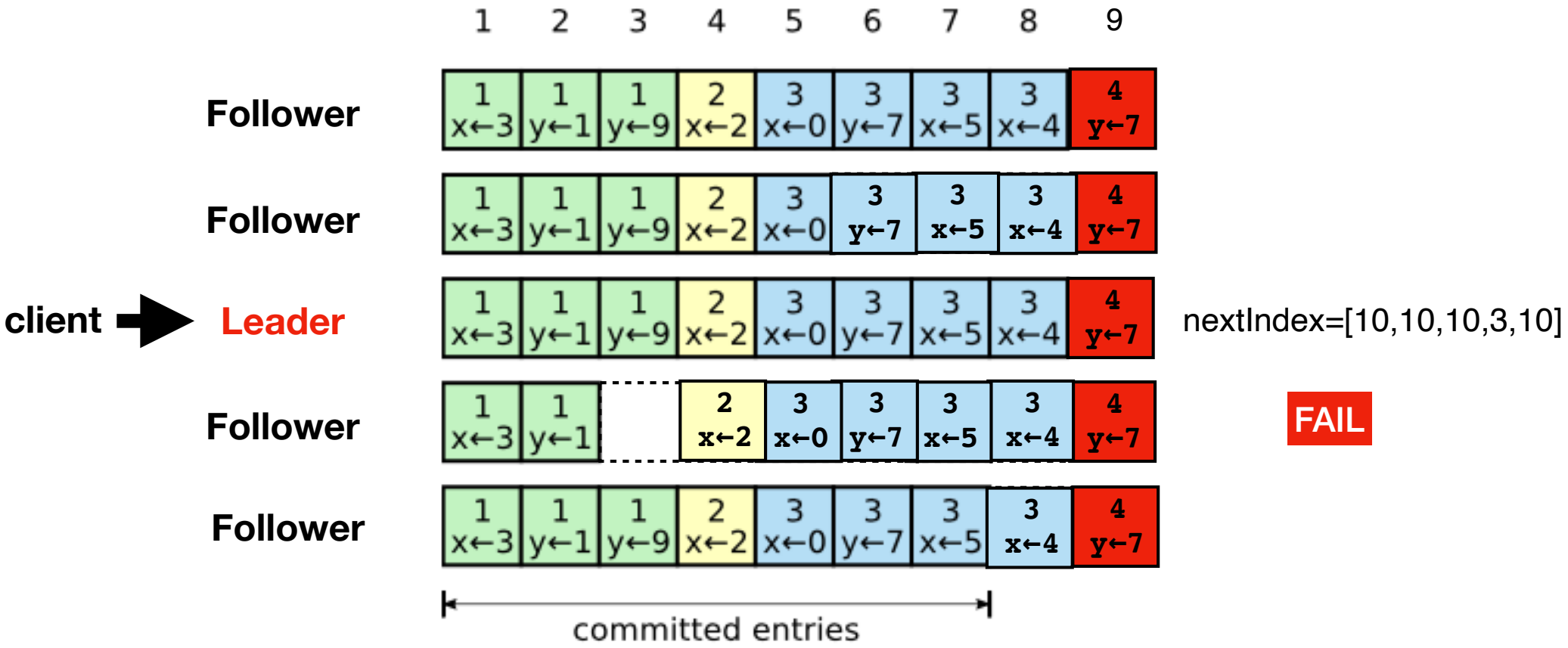
- This process repeats itself



- Leader keeps working backwards until success

Backtracking

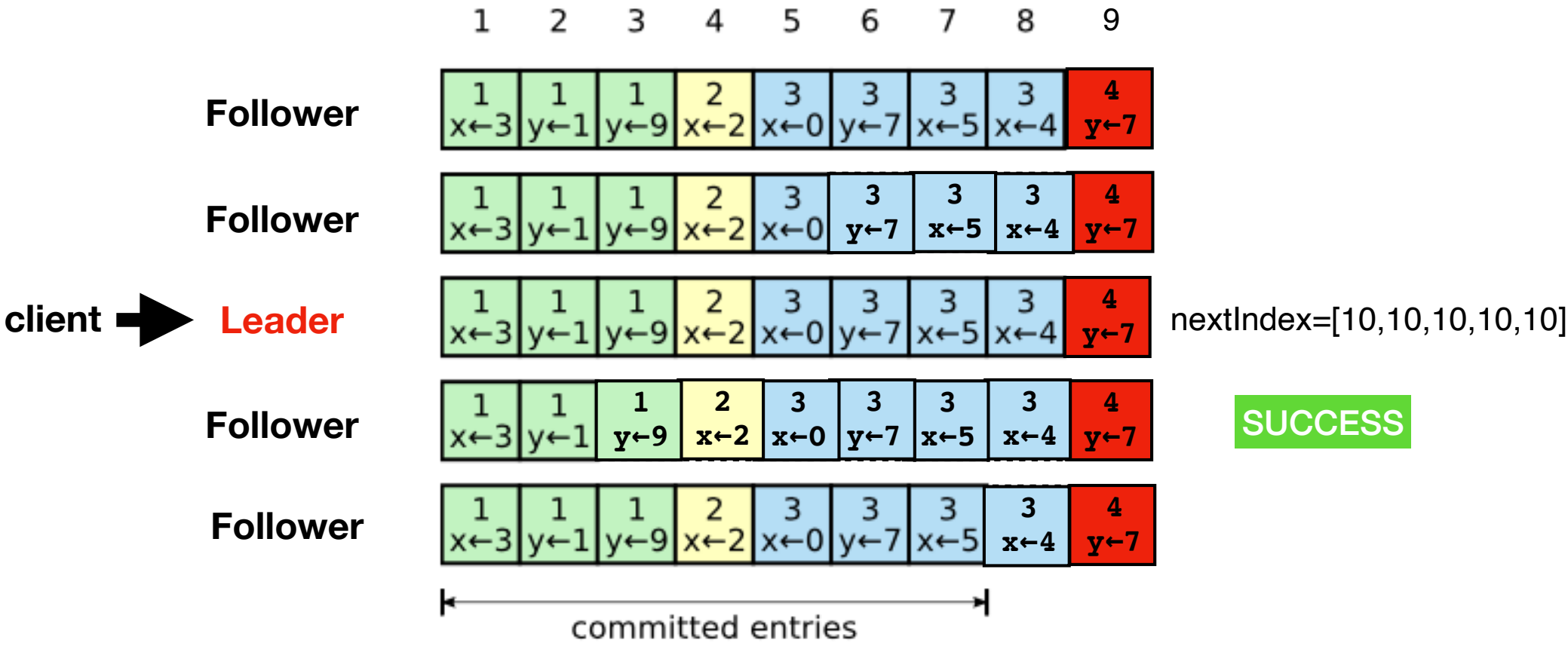
- This process repeats itself



- 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 unknown so it has to be worked out based on 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 6

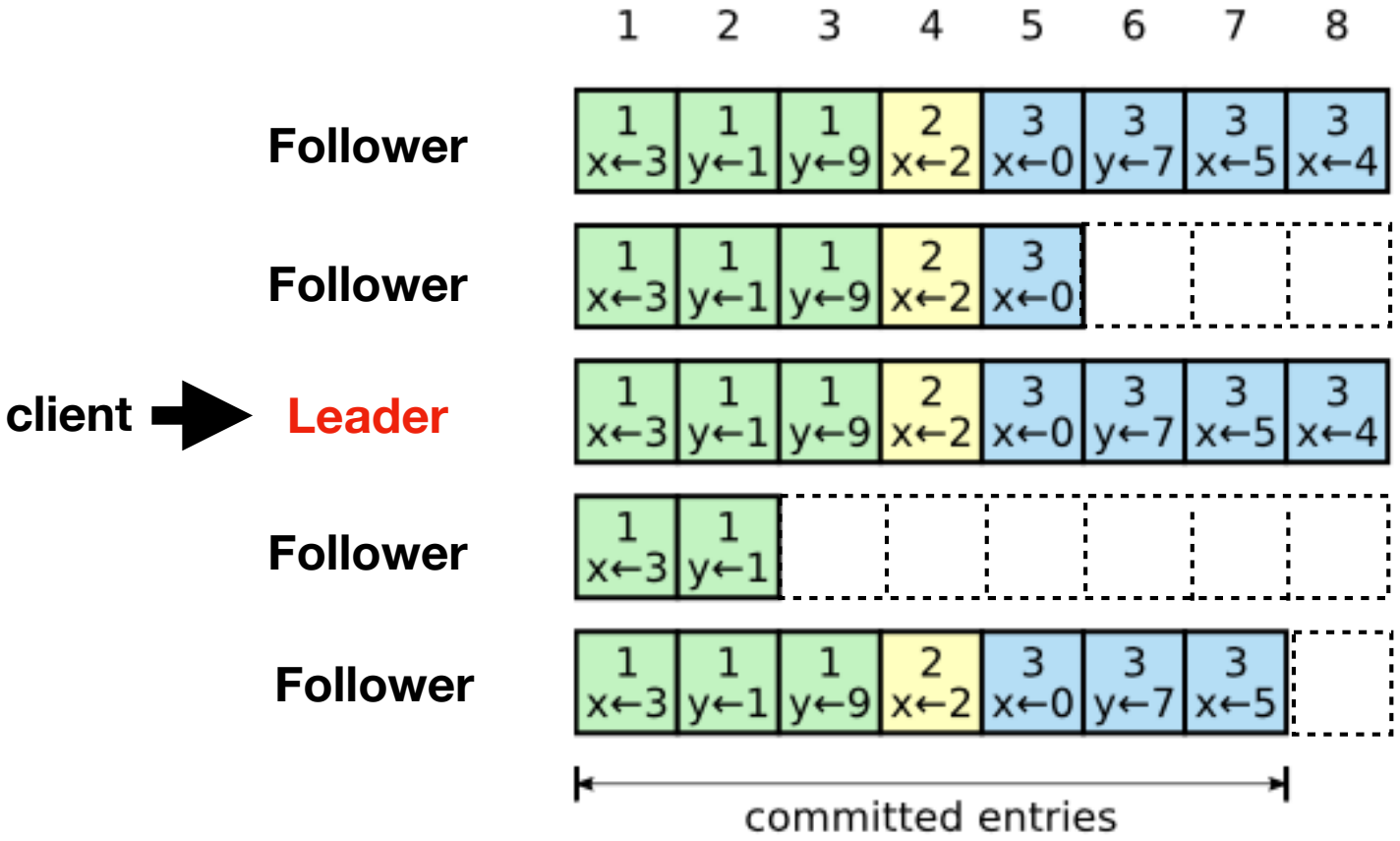
- Implement log replication via AppendEntries
- Designate one server as leader (manually)
- Implement a method to add an entry to its log
- Have the leader replicate to the followers

Part 7

Consensus

Consensus

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



- When consensus is reached -- entries committed

Consensus on Leader

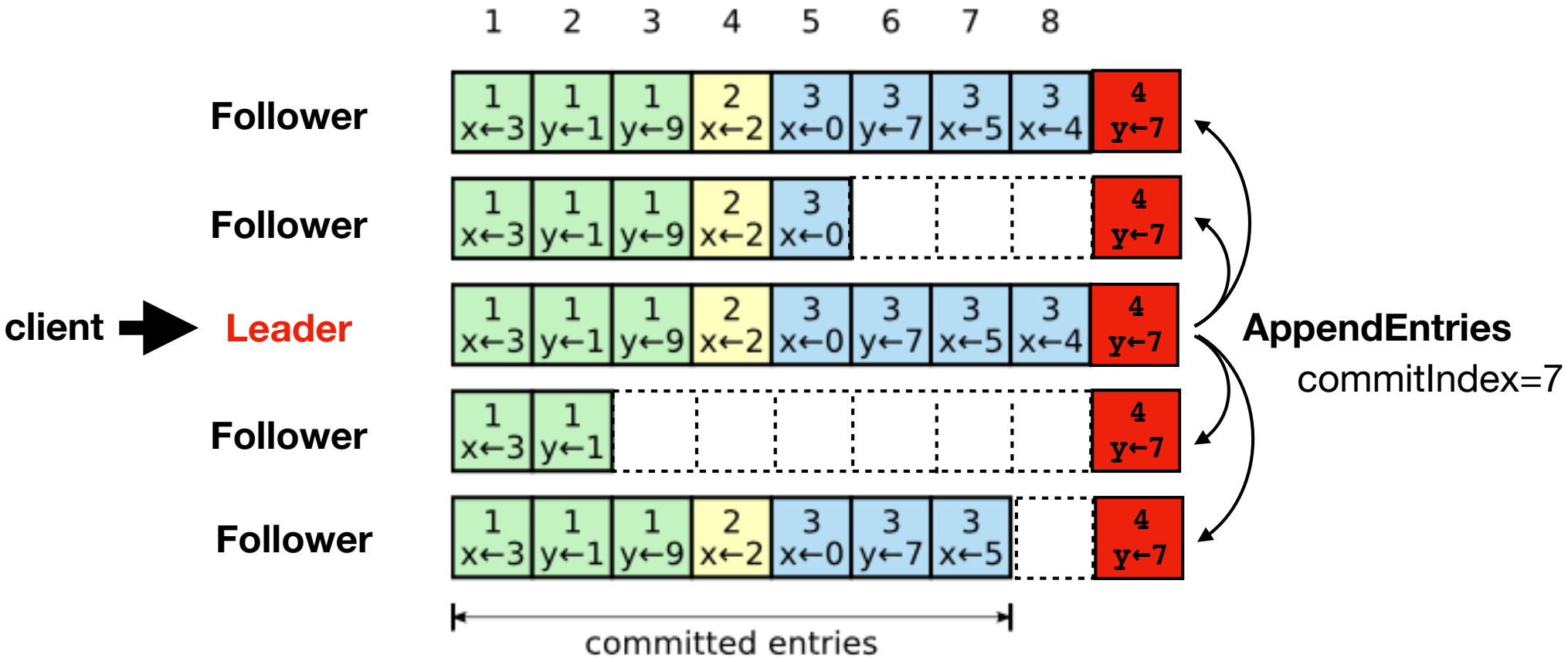
- The leader communicates with followers
- Followers inform the leader about their log state
- Note: There is an error in Figure 2 (missing info)
- From this, the leader can determine consensus

Problem

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

Consensus

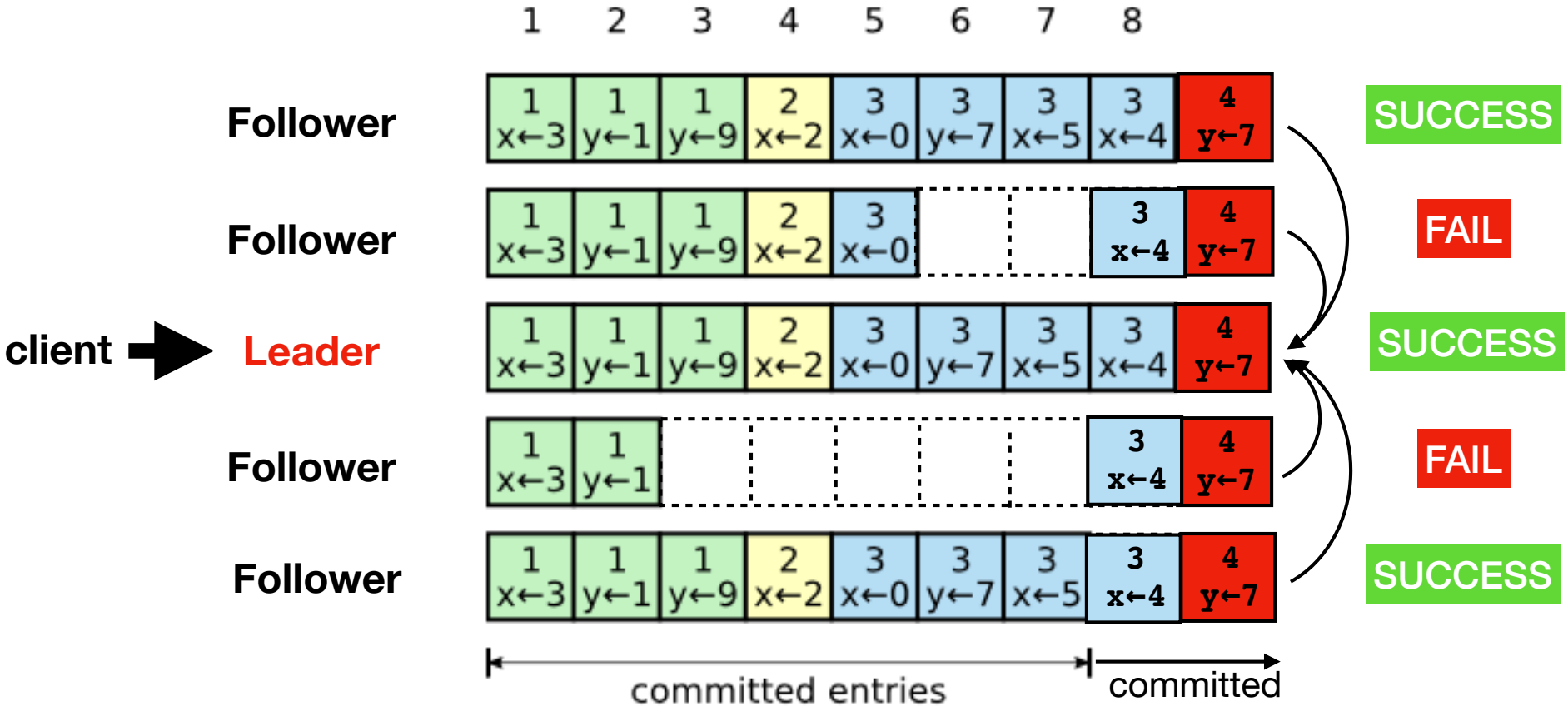
- The leader commit index is always sent



- Followers watch the index

Consensus

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



- Need success on a majority (not all)

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 each server can "do the thing" the client actually requested

Client

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

Key-Value Server

```
data = { }

def set(key, value):
    entry = ('set', key, value)
    append_entry(raft, entry)
    while not committed:
        wait
    # DO IT
    data[key] = value
```

There are some other complicated issues here, but will resolve in project

Project 7

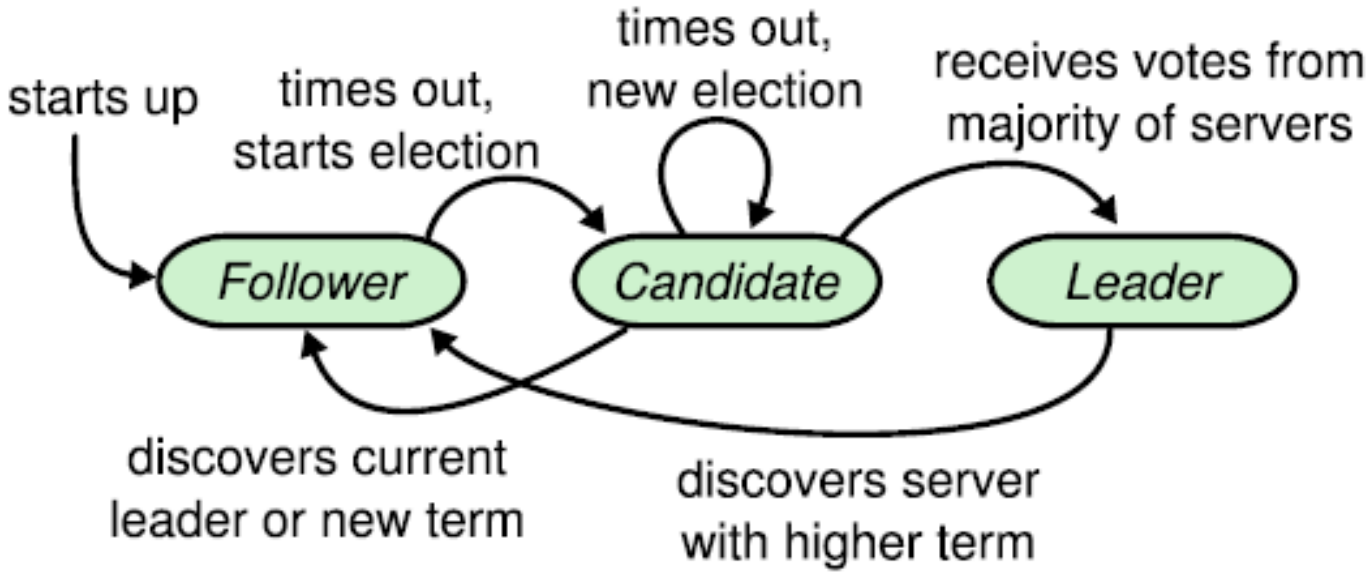
- Implement consensus
- Leader determines consensus
- Leader communicates consensus to followers
- Leader/followers "apply the state machine"

Part 8

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

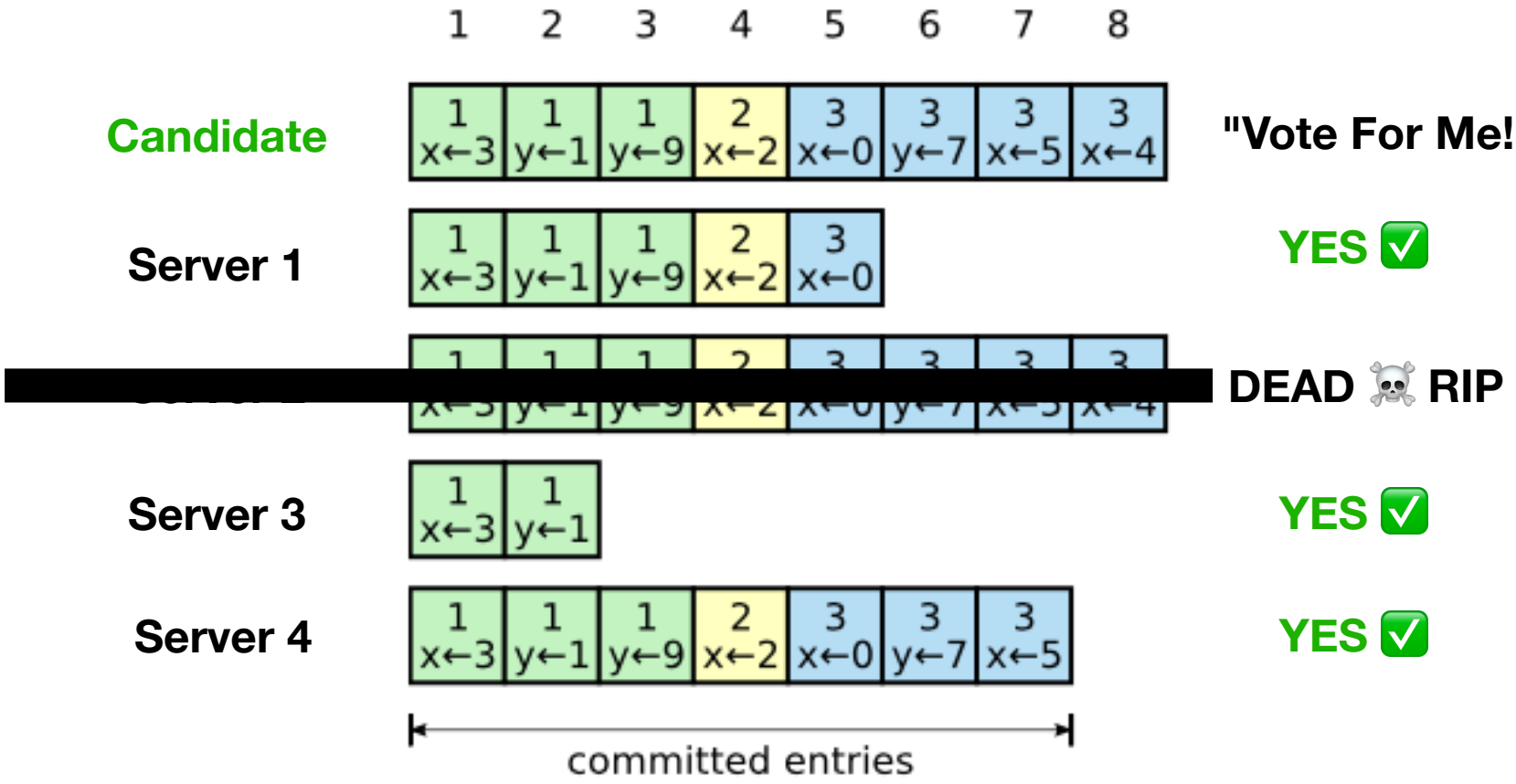
- A server may only vote for one candidate in a given term (subsequent requests denied)
- This is for dealing with split votes
- It's possible that two followers could promote to candidate at the same time. They'd have the same term number, but might not able to get a quorum.

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.

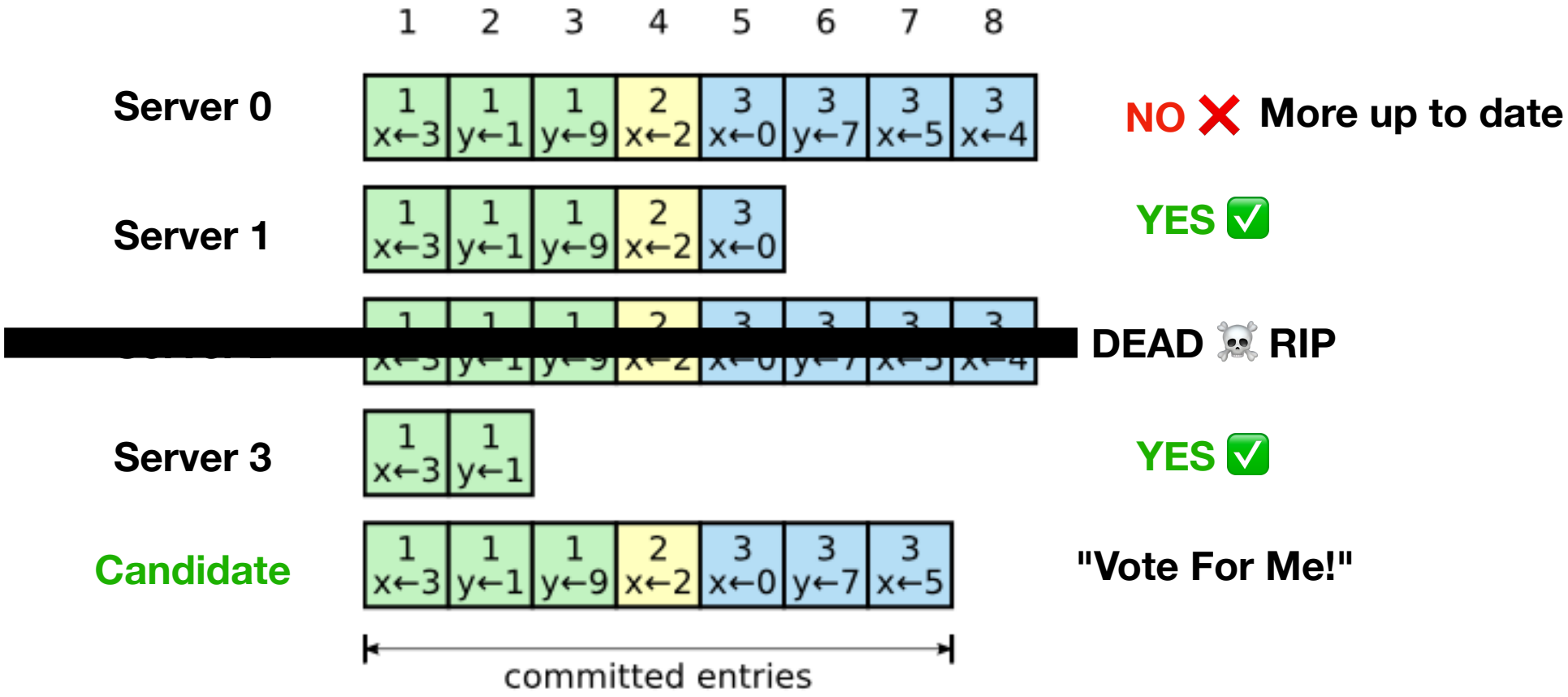
Voting Example

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



Voting Example

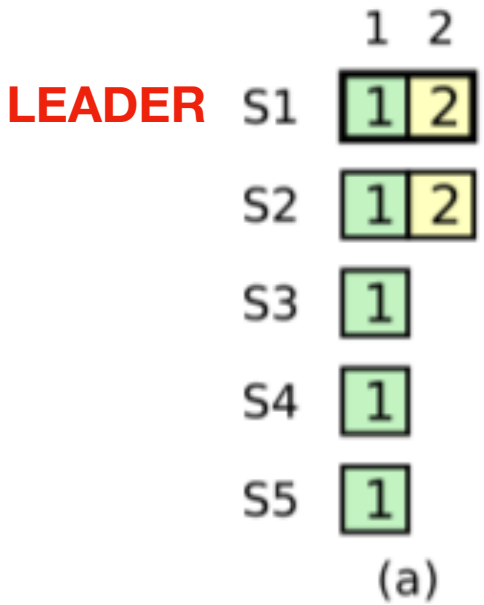
- But a server with a shorter log still might win



- The winner will always have all committed entries

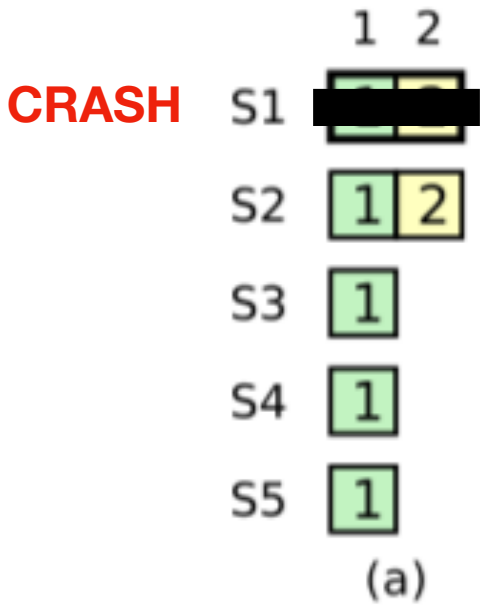
"Figure 8"

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



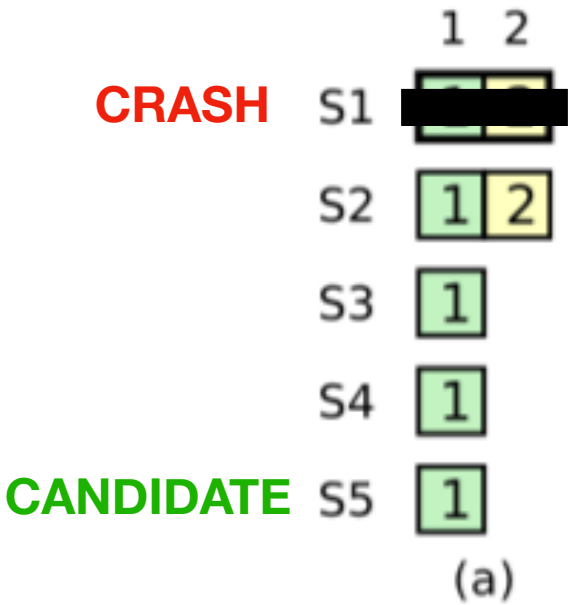
"Figure 8"

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



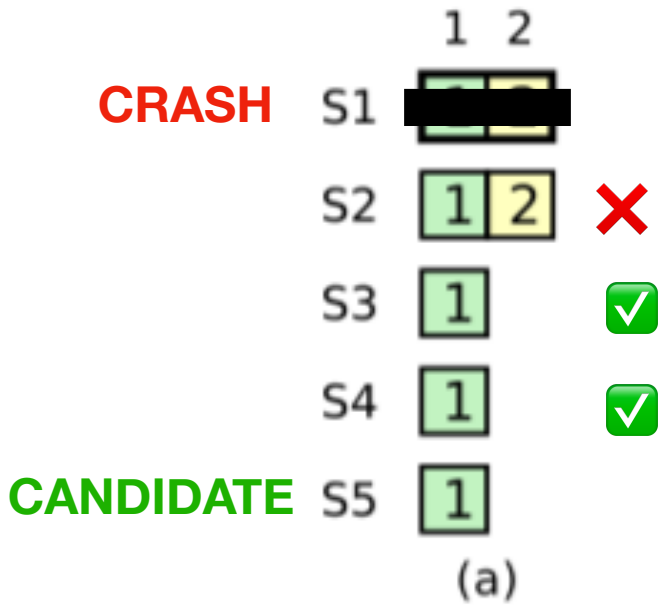
"Figure 8"

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



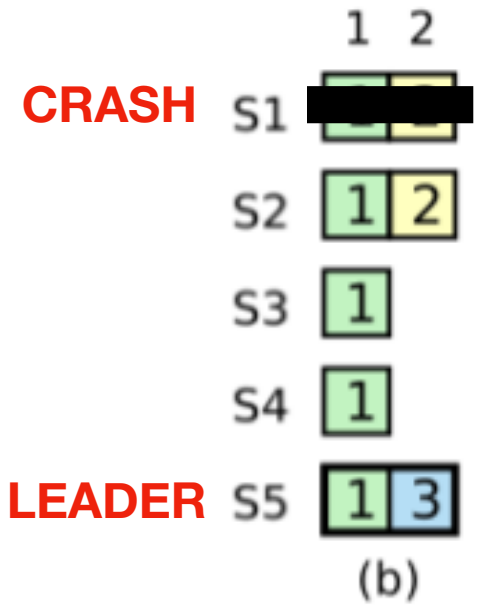
"Figure 8"

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



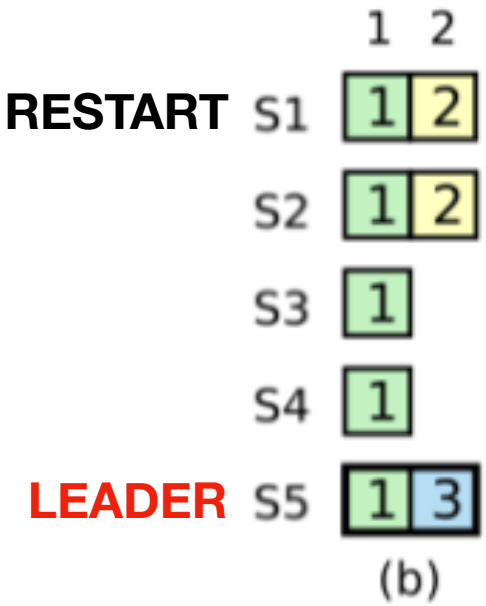
"Figure 8"

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



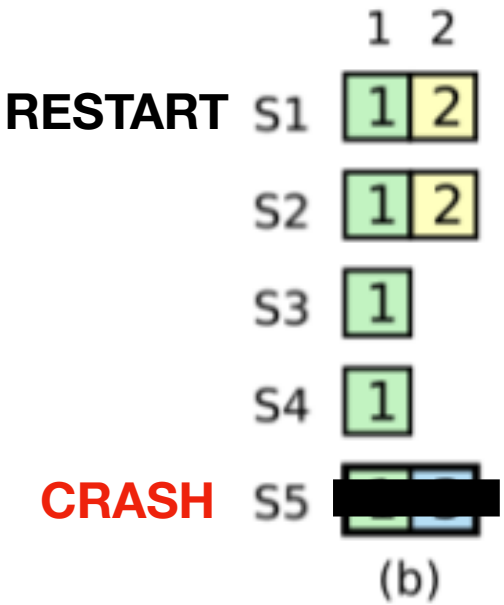
"Figure 8"

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



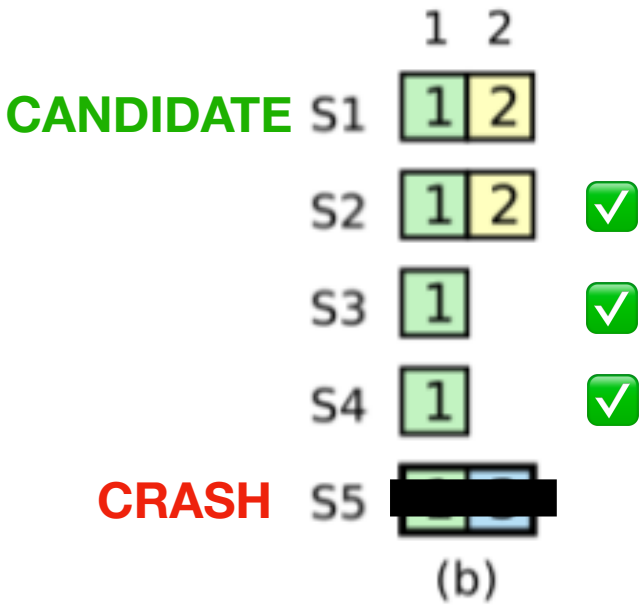
"Figure 8"

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



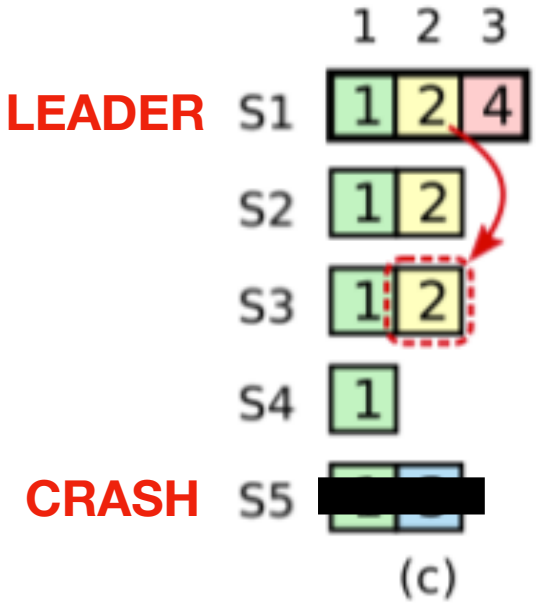
"Figure 8"

- 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

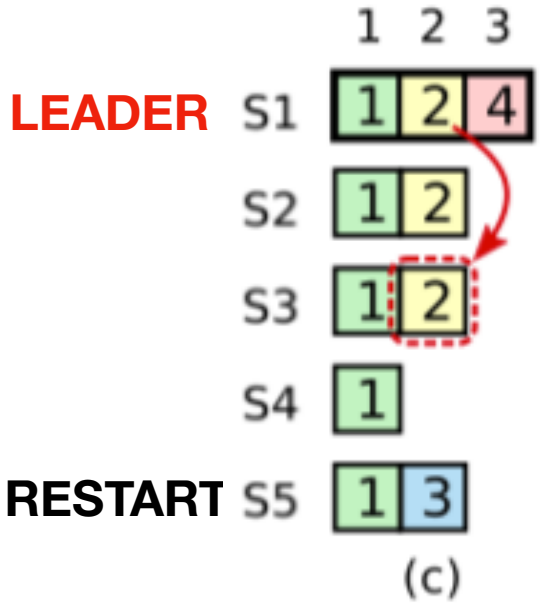


Observe: Entry in Index 2 replicated on majority

- 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

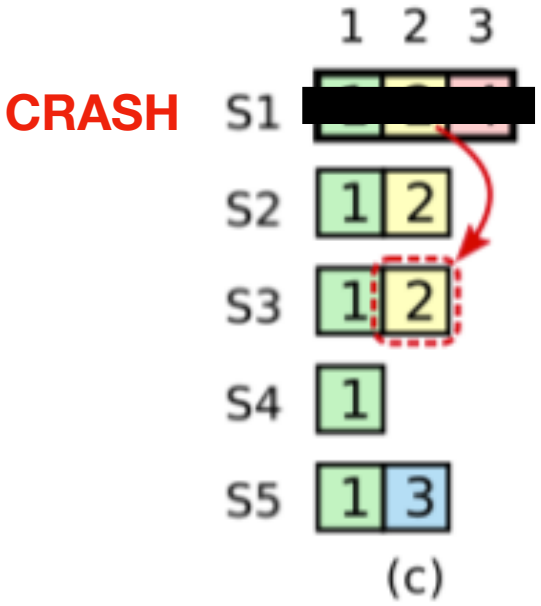


Observe: Entry in Index 2 replicated on majority

- 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

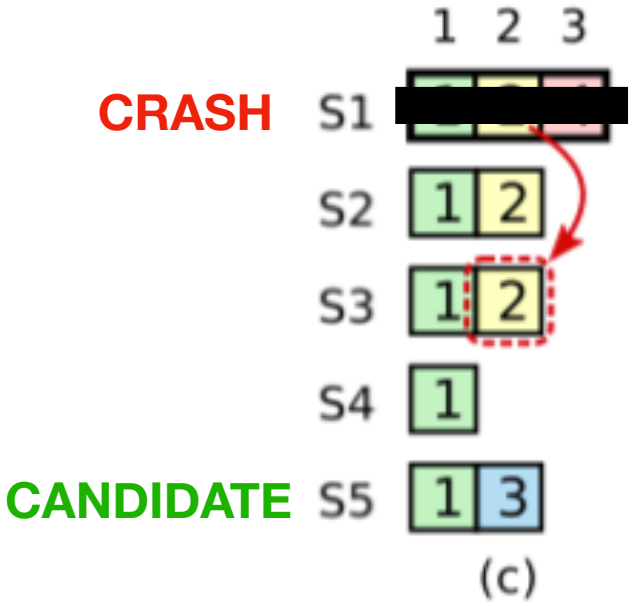


Observe: Entry in Index 2 replicated on majority

- 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

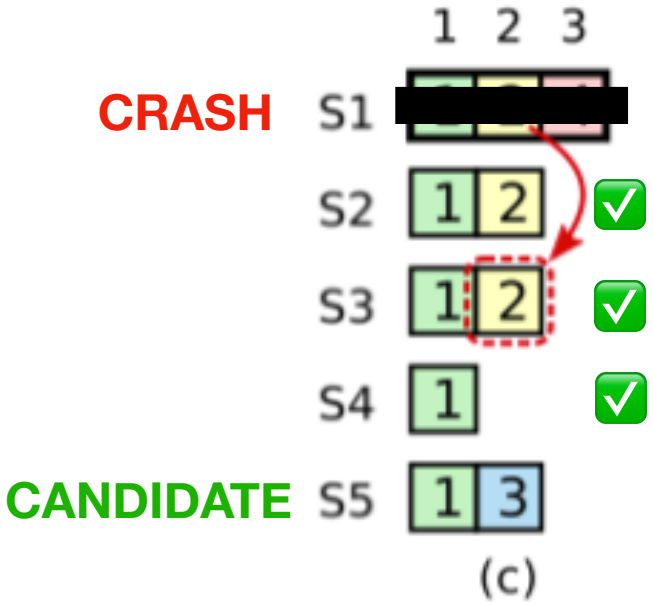


Observe: Entry in Index 2 replicated on majority

- 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

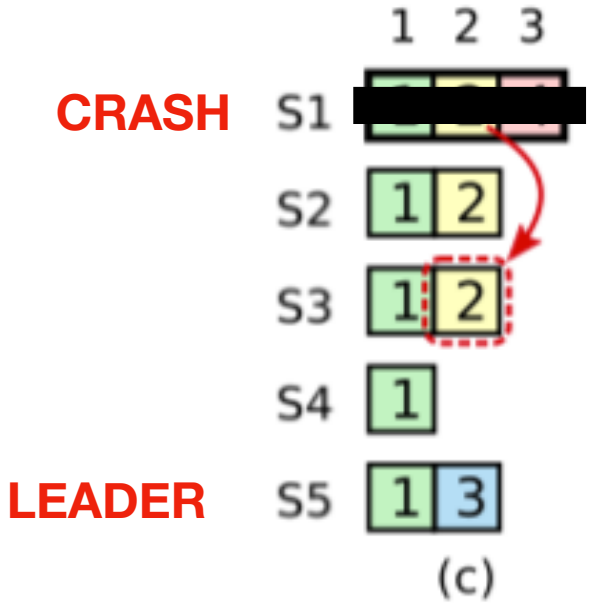


Observe: Entry in Index 2 replicated on majority

- 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

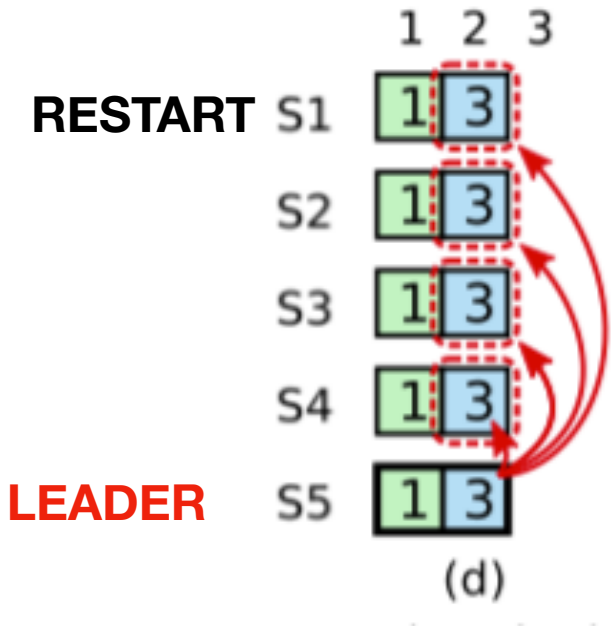


Observe: Entry in Index 2 replicated on majority

- 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

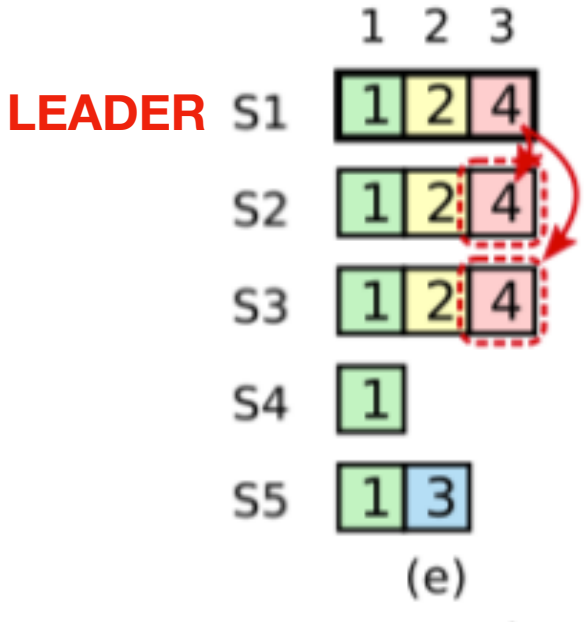


New leader overwrites prior entry (even though replicated on majority)

- 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

- Implement Raft leader election
- Again, using a model/simulation
- Focus on testing/debugging

Part 9

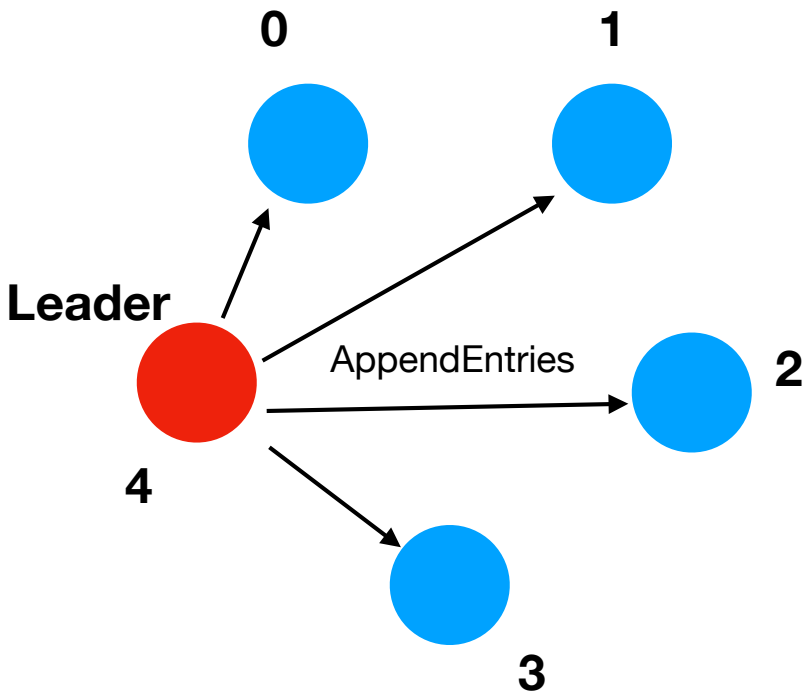
Timing

Heartbeats and Timeouts

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

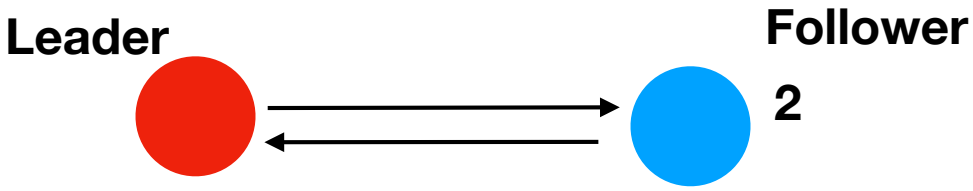
Heartbeats

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



Heartbeat Timing

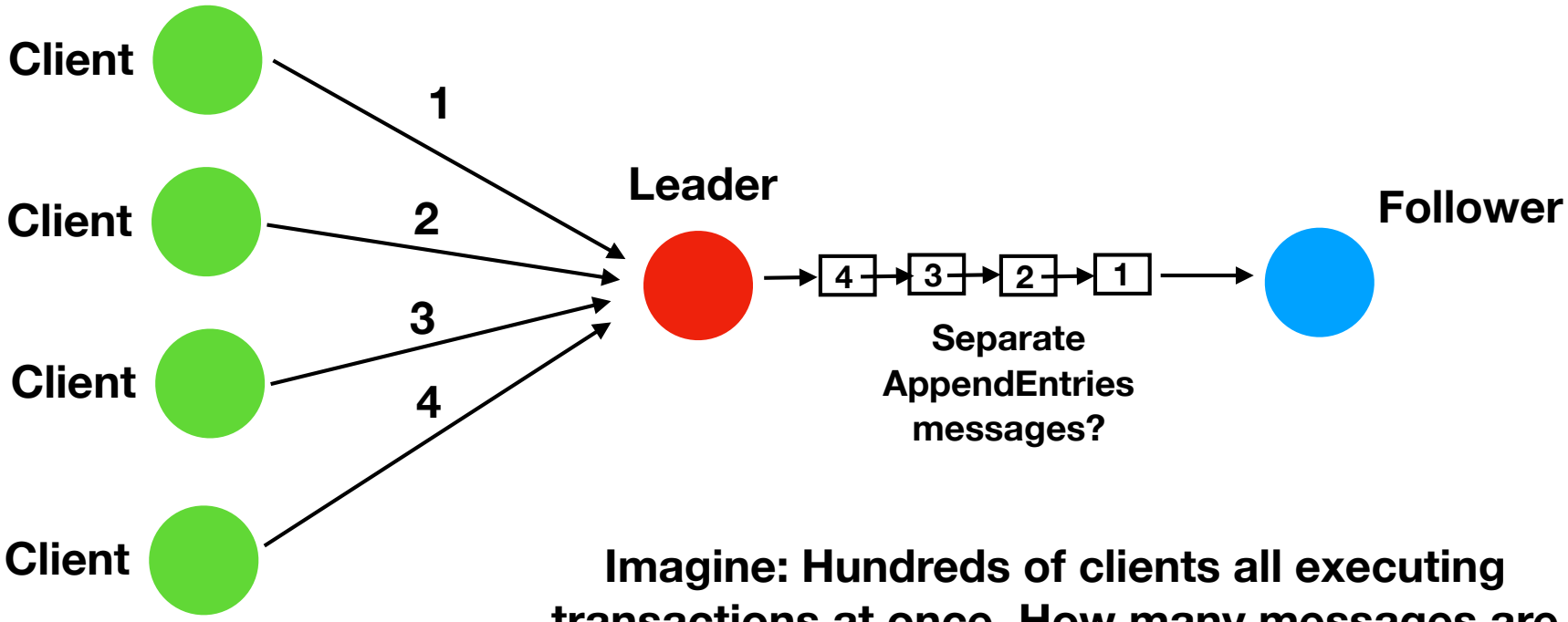
- Timing based on round-trip "ping time"



- 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

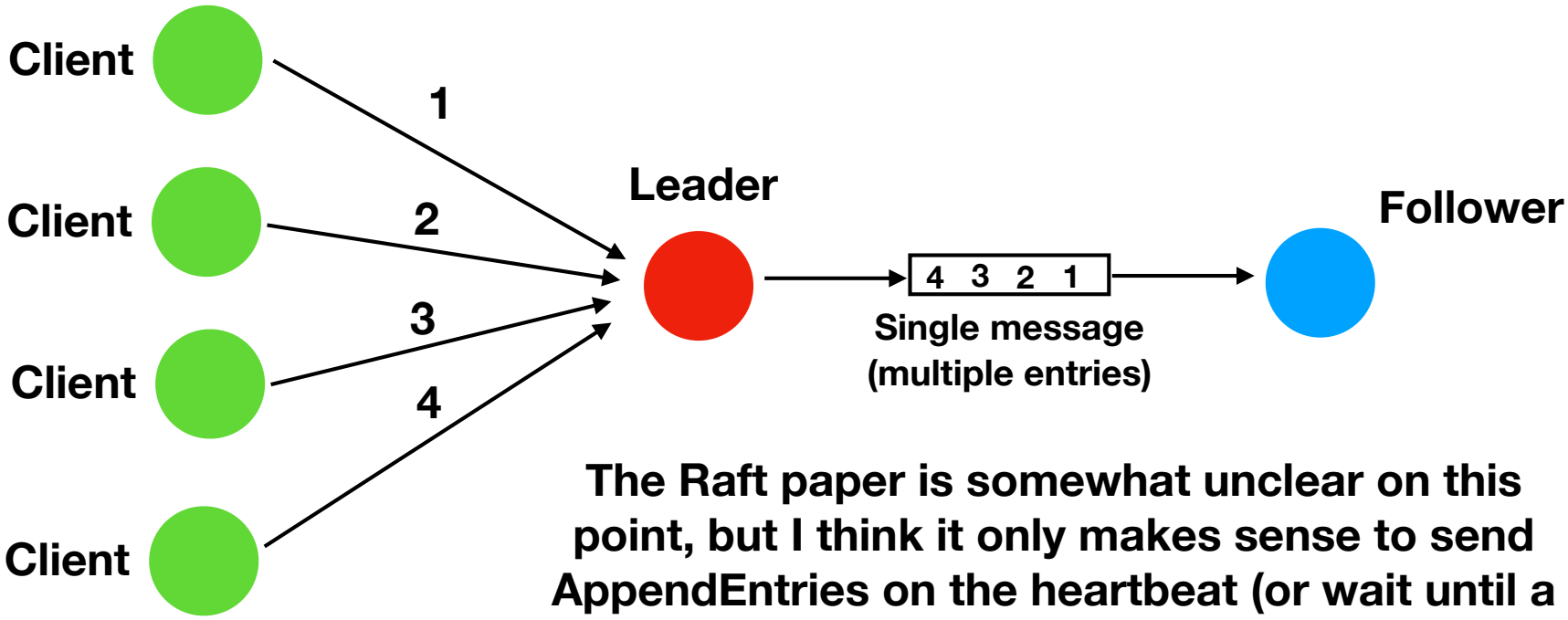
- Q: Does the leader immediately send AppendEntries for every client transaction?
- Scenario:



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:



The Raft paper is somewhat unclear on this point, but I think it only makes sense to send AppendEntries on the heartbeat (or wait until a response has been received).

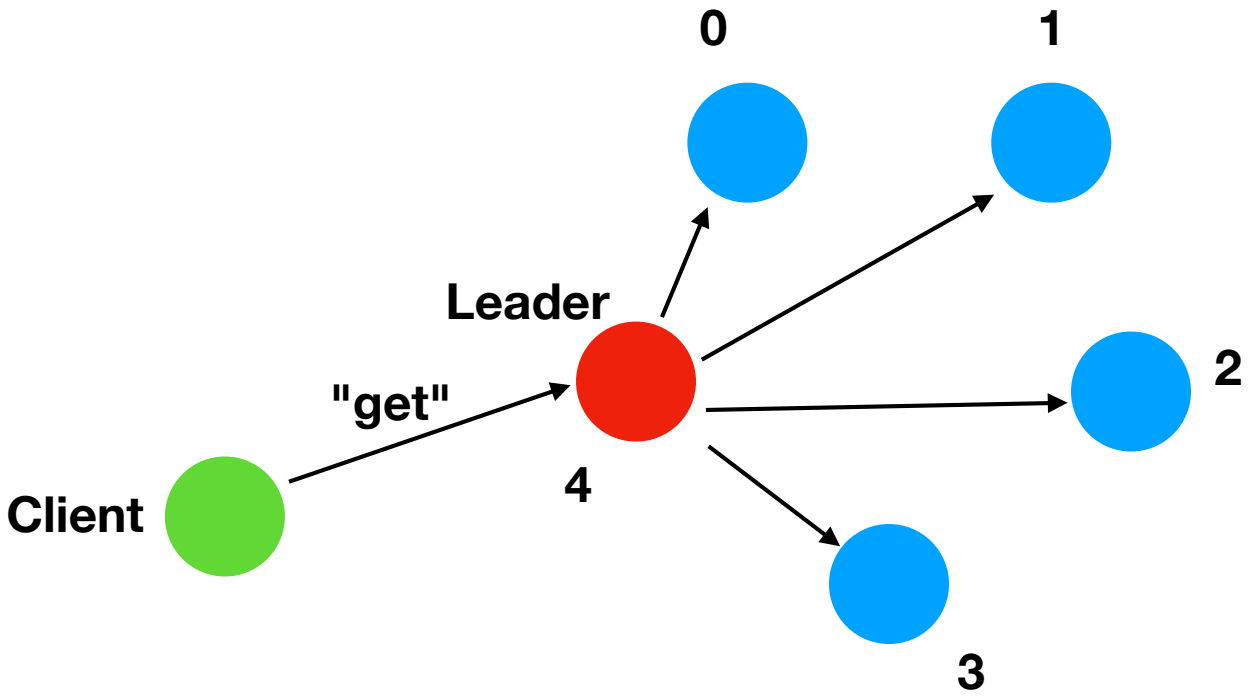
Discussion

- 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

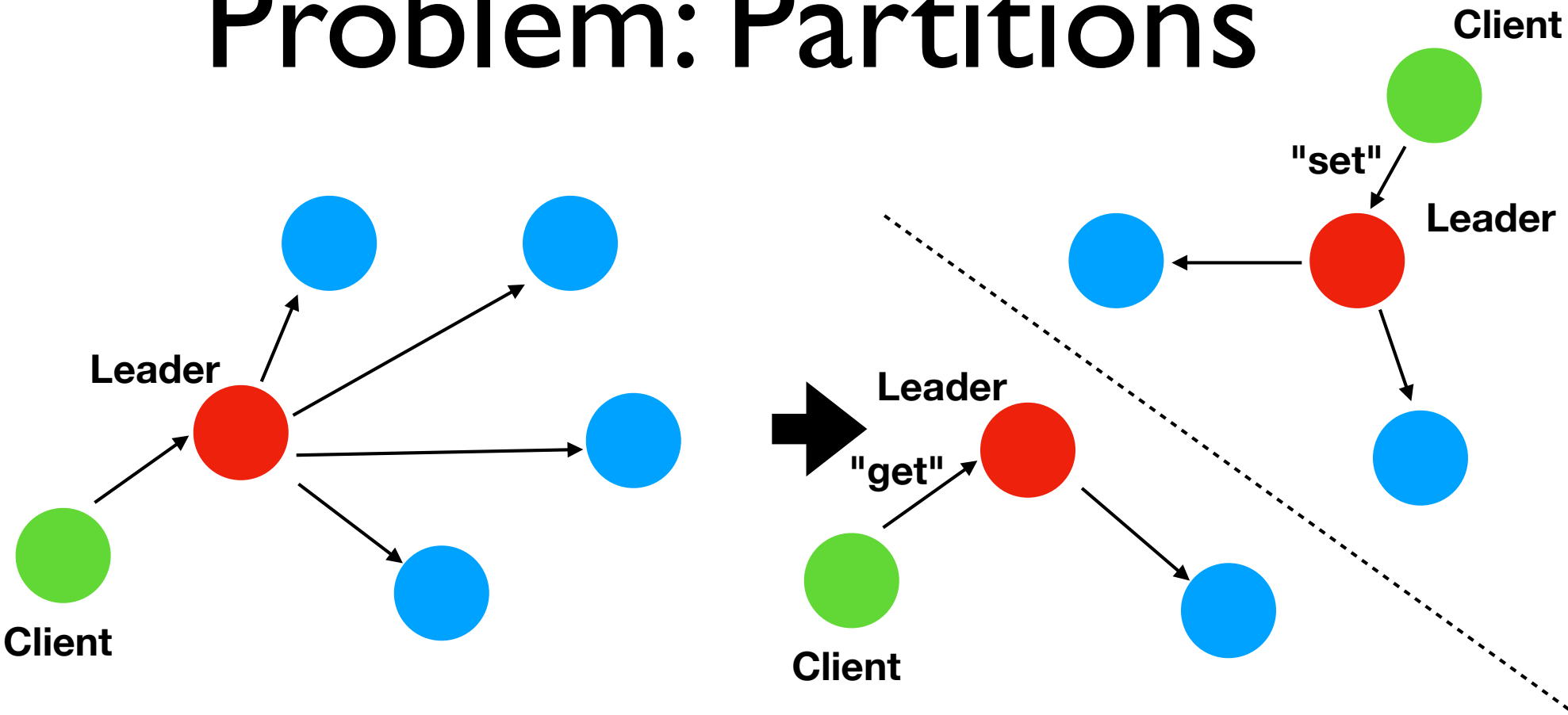
- Is the heartbeat a global timer that applies to all followers at once?
- Or does the leader maintain a different heartbeat timer for each follower?
- The Raft paper leaves this unanswered. We just know that each follower should regularly receive AppendEntries.

The Trouble with Reads



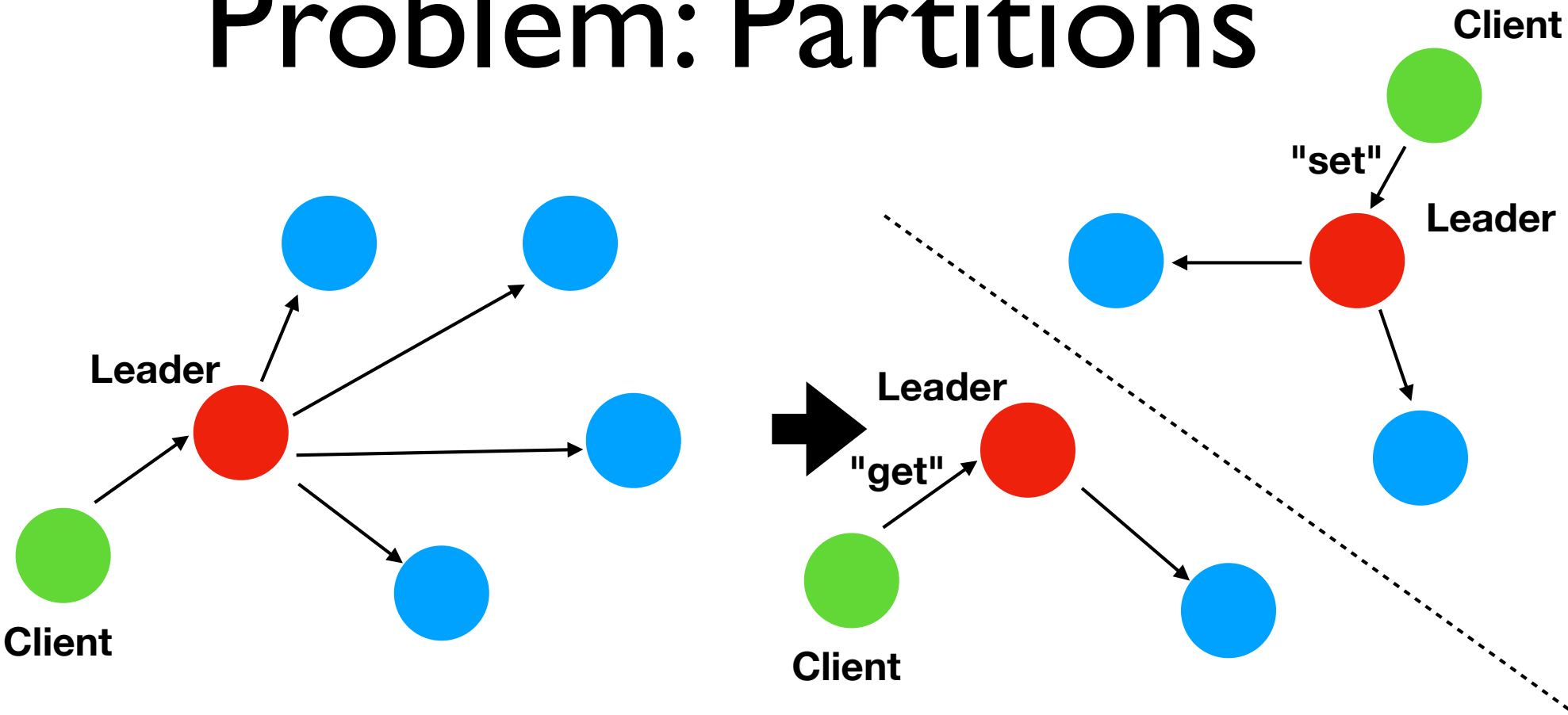
- Question: Are "reads" a transaction in the log?
- They don't modify state. Is it critical?

Problem: Partitions



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

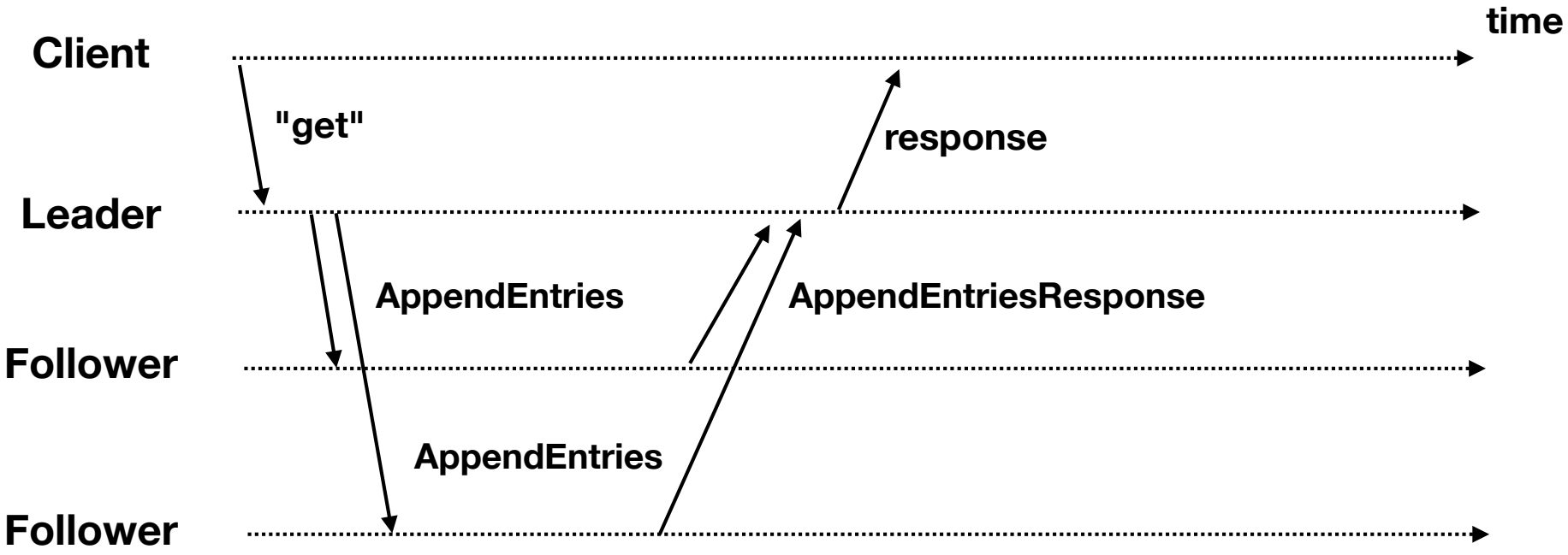
Problem: Partitions



- 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



Discussion

- Is the leader allowed to respond to ANY client request until it knows (for certain) that it's actually the leader?
- Simply having a "leader" flag doesn't seem like it's enough.
- Would need to know that leader is in active communication with a consensus of followers.

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

Election Timeouts

- Servers that haven't heard from a leader in awhile call for an election (become candidate)
- This is a randomized timeout--typically at least 10-20x longer than the leader heartbeat
- Randomized to prevent all servers from calling an election at the same time.

Project 9

- Figure out how to add "timing" elements to the model/simulation
- Implement a mechanism for establishing leadership
- Reminder: We're still focused on testing/ debugging.
- No threads, network, etc.

Part 10

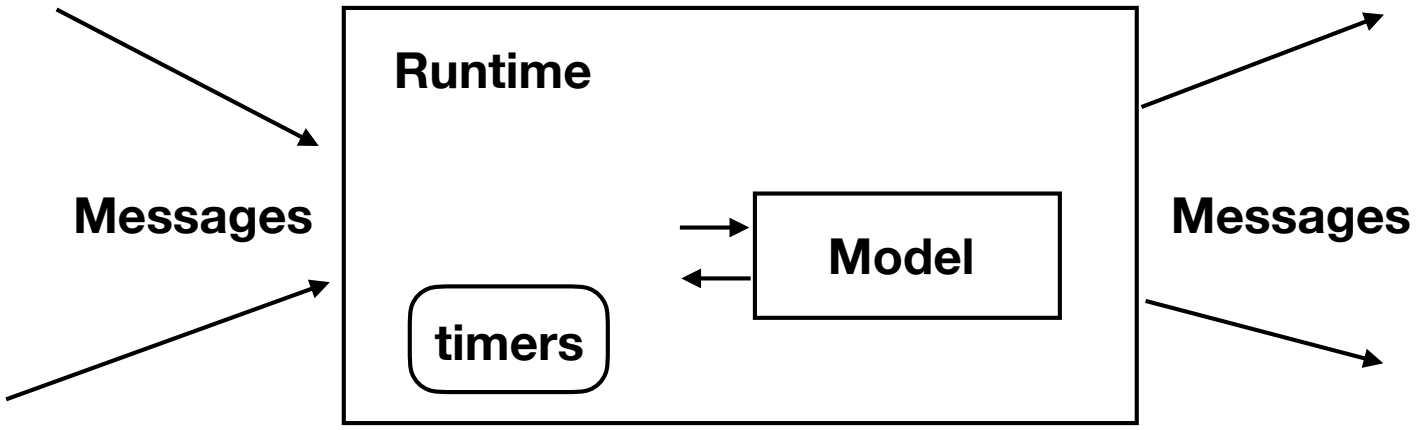
The System

System Implementation

- At some point, we have to turn Raft into an operational "system" of some sort
- Not a model, not a simulation.
- Question: How?!?

One Strategy: Embedding

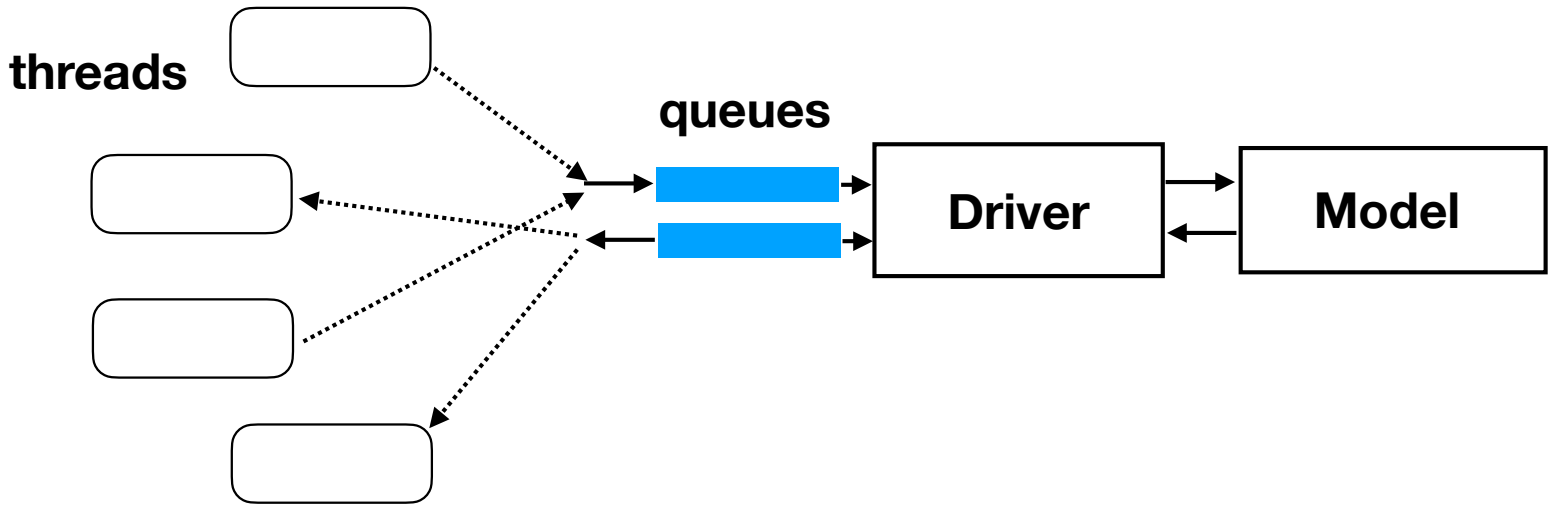
- Embed the model into a surrounding runtime environment



- If you were able to test the model, then you should be able to drive it for real.

Use Queues

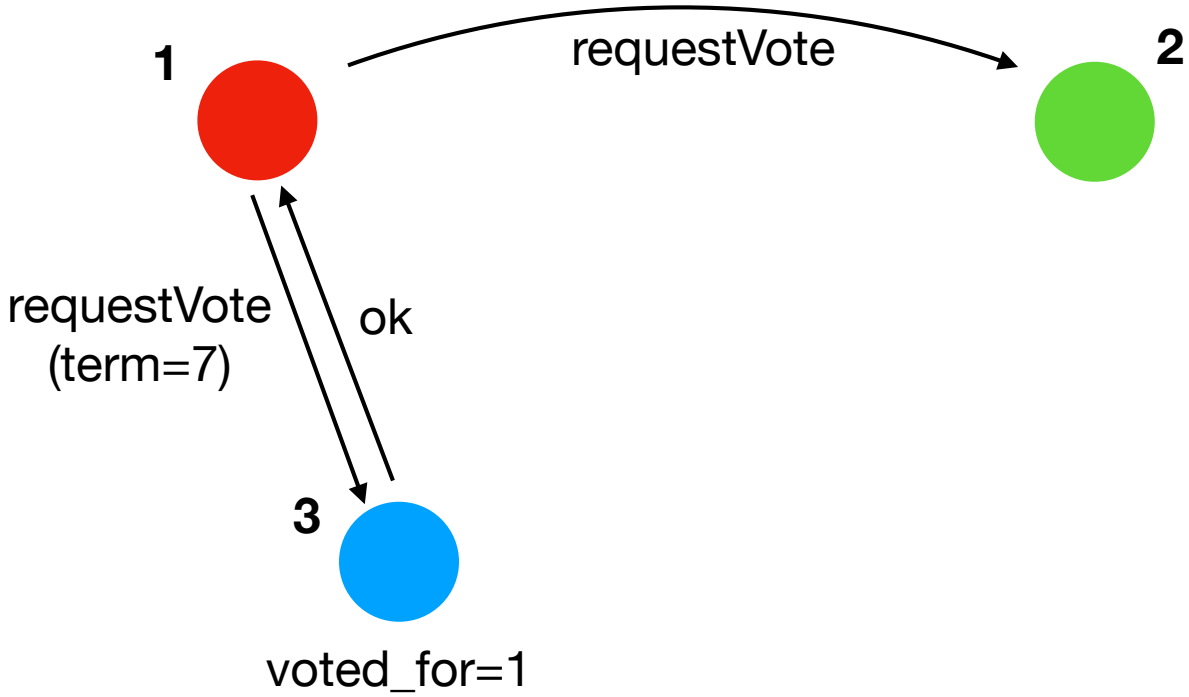
- One approach, implement a single-threaded driver using queues



- Make all concurrency sit on the "outside"

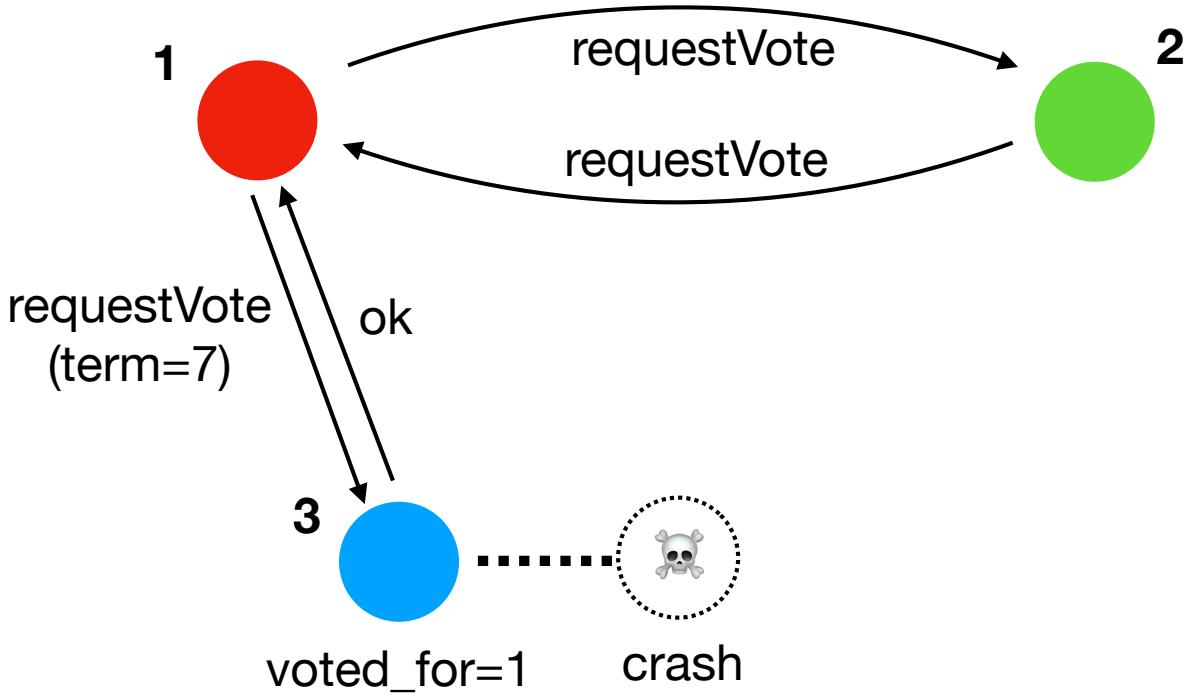
Non-volatile Storage

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



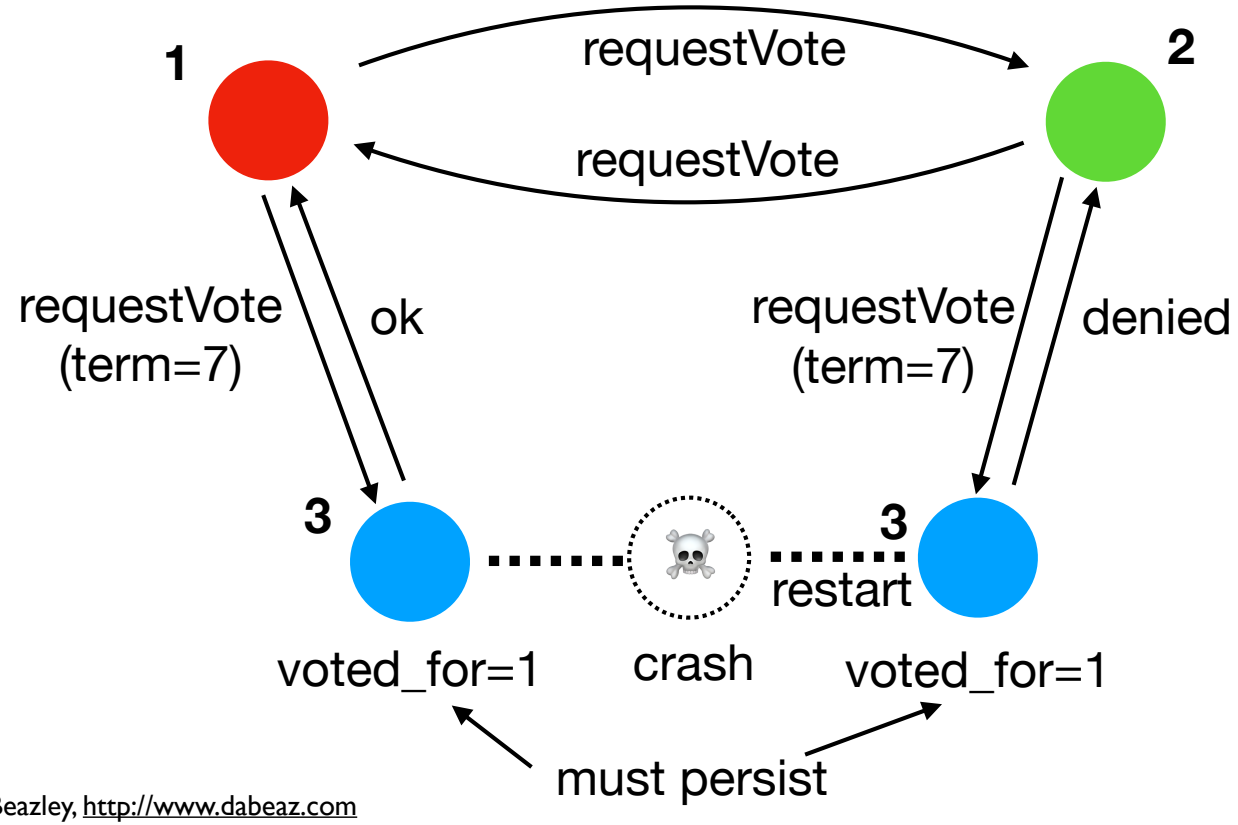
Non-volatile Storage

- 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!

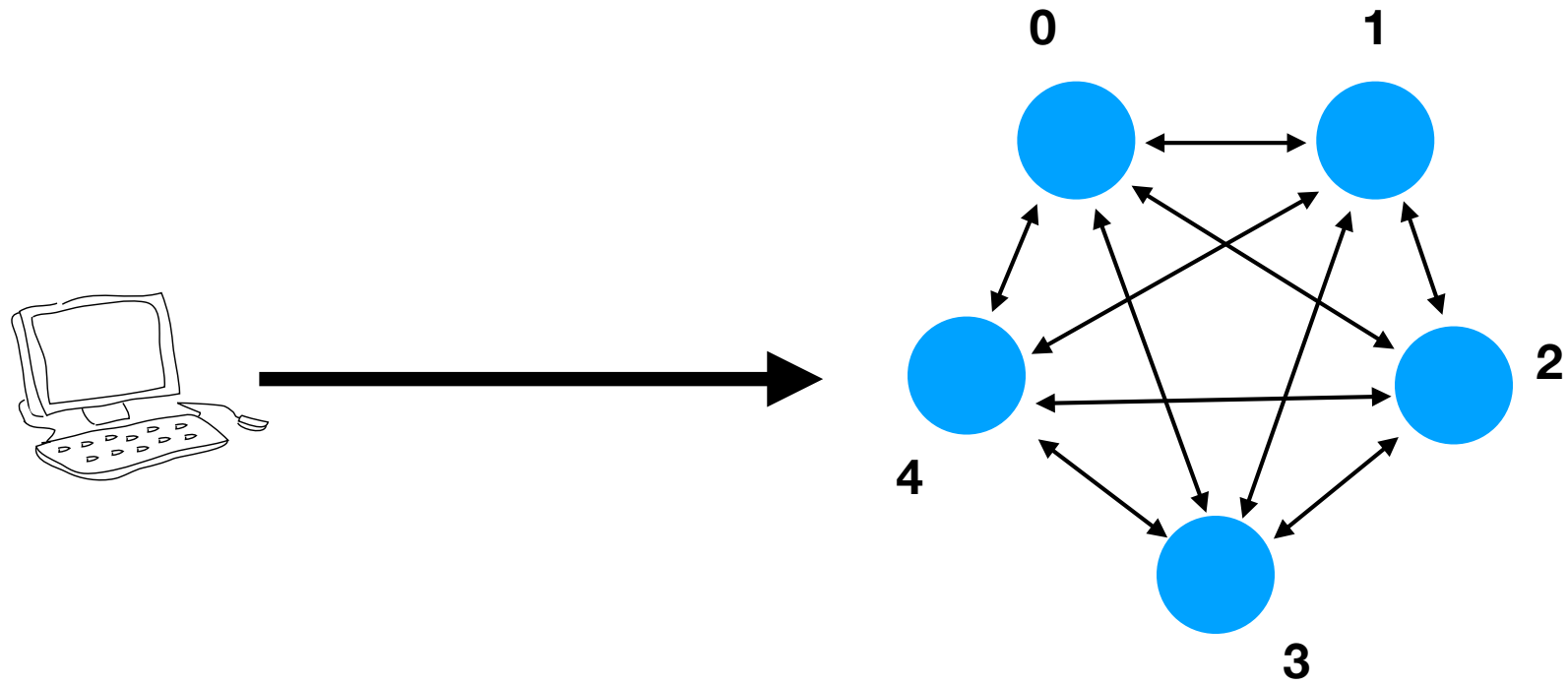


Idea on NV Storage

- Server could maintain a "private" append-only (disk) log of all transactions and state changes
- Would include AppendEntries operations, but also changes to certain variables such as the current term, voting, etc.
- Emphasize: This is NOT the Raft log.
- On crash/restart, could replay the log to restore the server to last known operational state.

Client-Connection

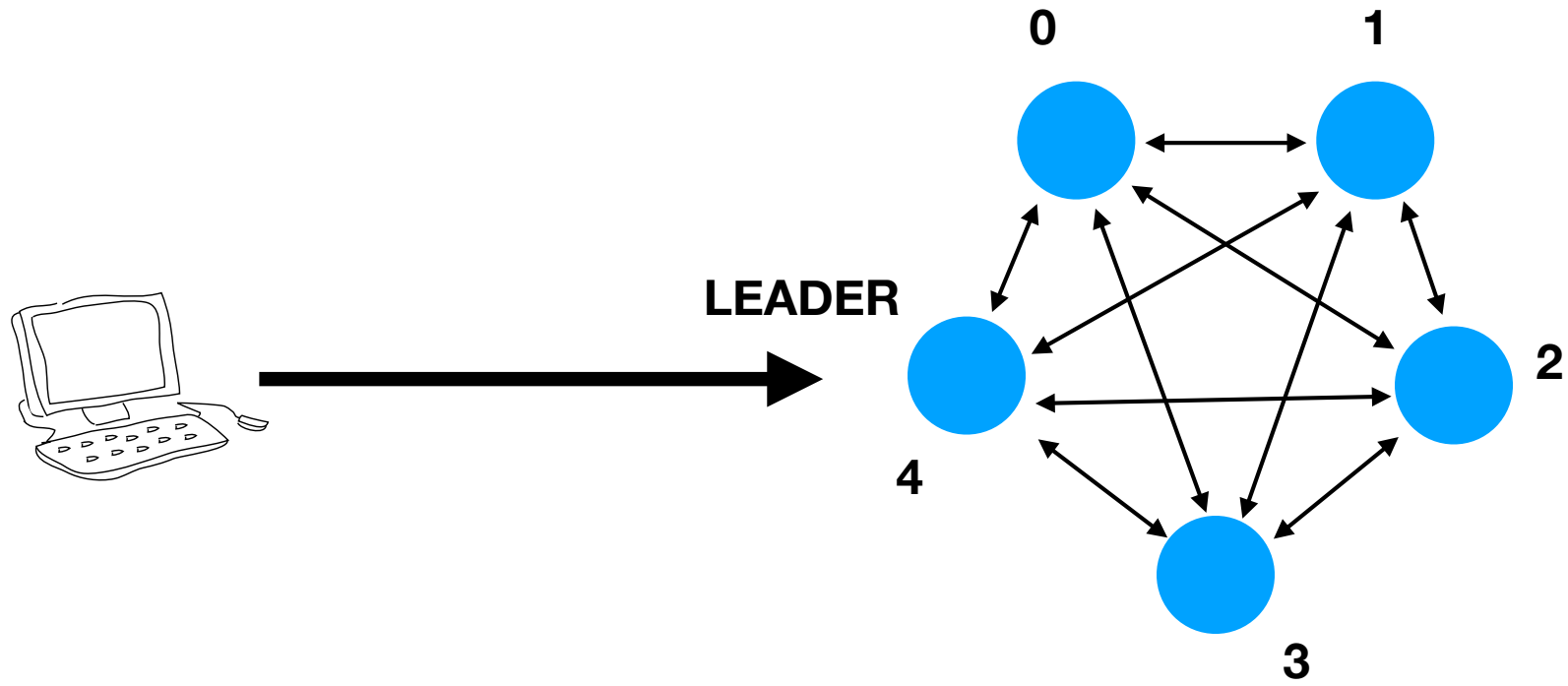
- At some point, a client must talk to Raft



- For example, to implement a key-value store

Strong Leader

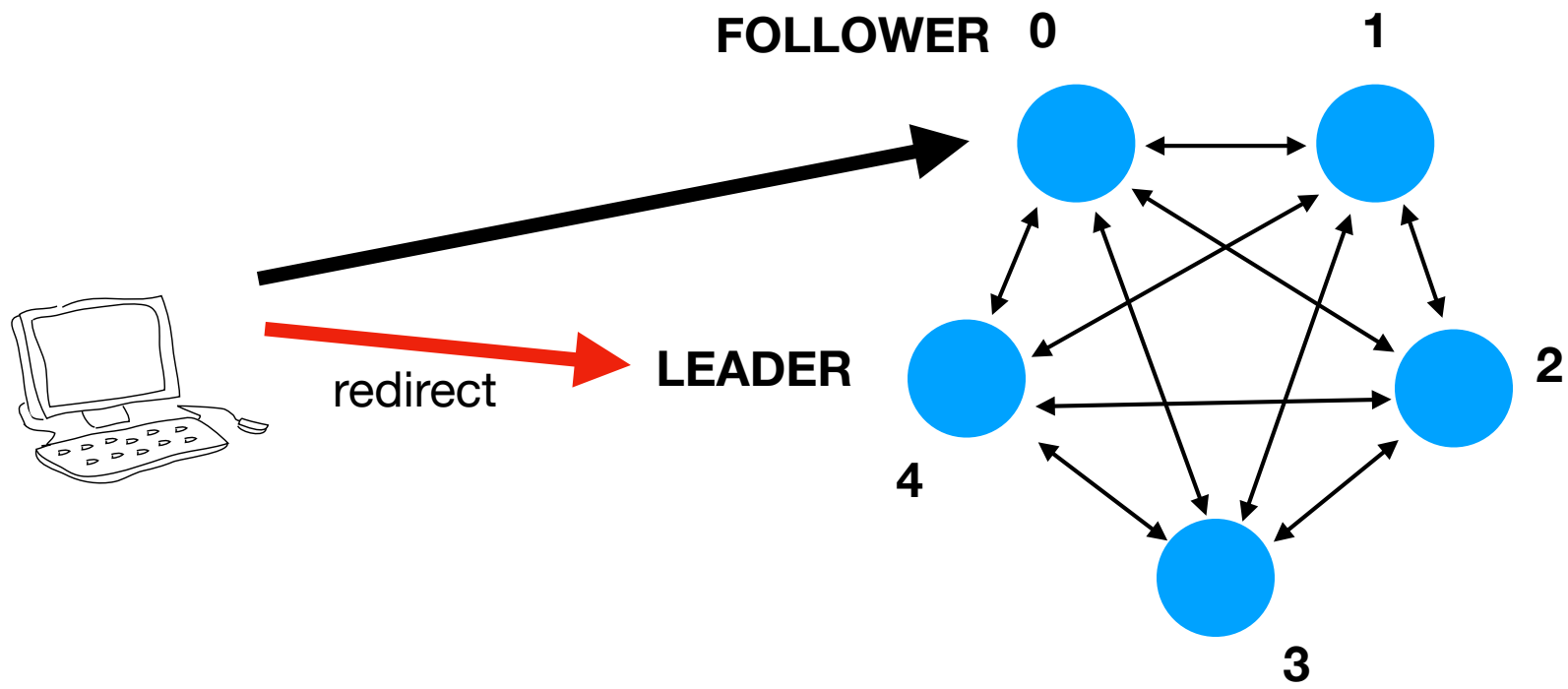
- Clients only talk to the leader



- 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

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 10

- Build a runtime environment for Raft
- Test it on a cluster of independent processes
- Implement a fault-tolerant Key-Value store
- And finish everything else!