

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.

This Week

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

6.824 - Spring 2020

6.824 Lab 3: Fault-tolerant Key/Value Service

Due Part A: Mar 13 23:59

Due Part B: Apr 10 23:59

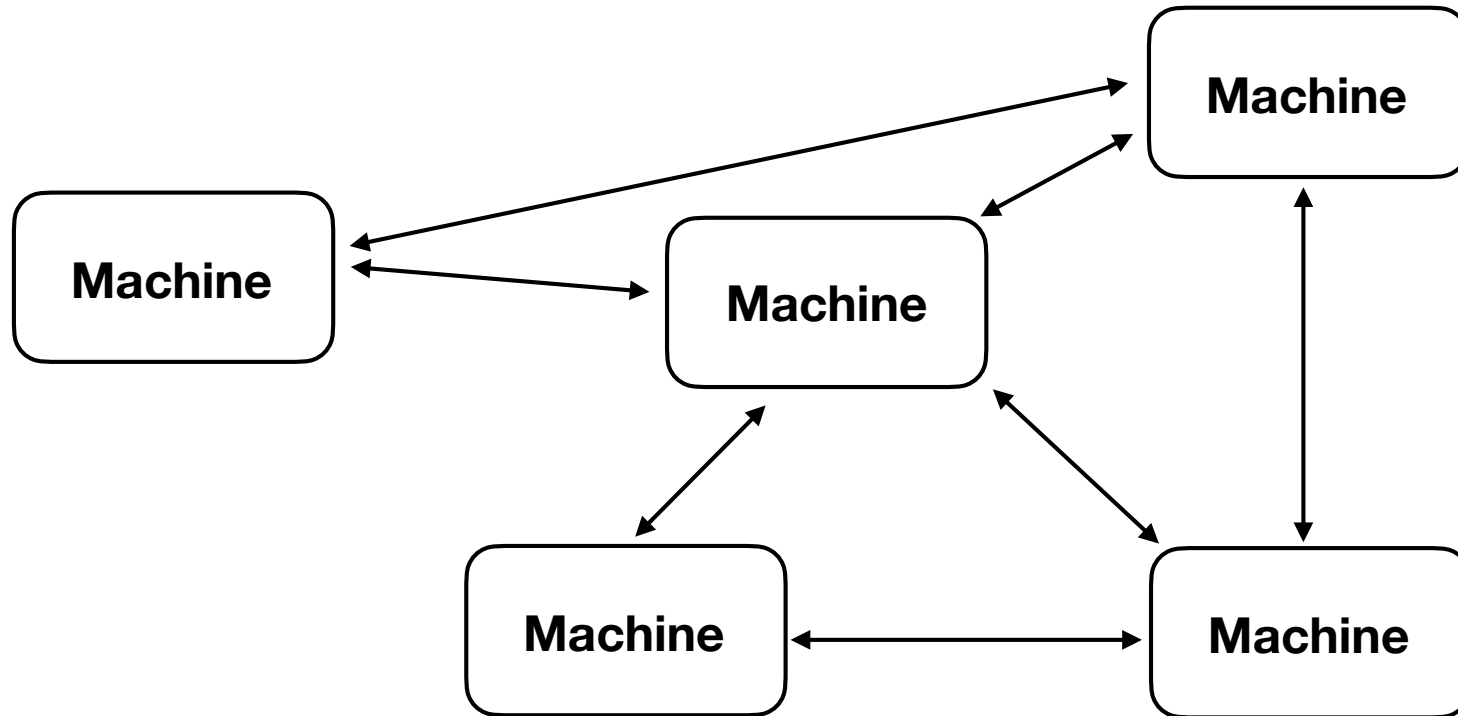
Introduction

In this lab you will build a fault-tolerant key/value storage service using your Raft library from [lab 2](#). Your key/value service will be a replicated state machine, consisting of several key/value servers that use Raft for replication. Your key/value service should continue to process client requests as long as a majority of the servers are alive and can communicate, in spite of other failures or network partitions.

- Note: It's an 8-week project for them. Not for you!

High Level View

- Distributed Computing



- Machines/services communicating over a network

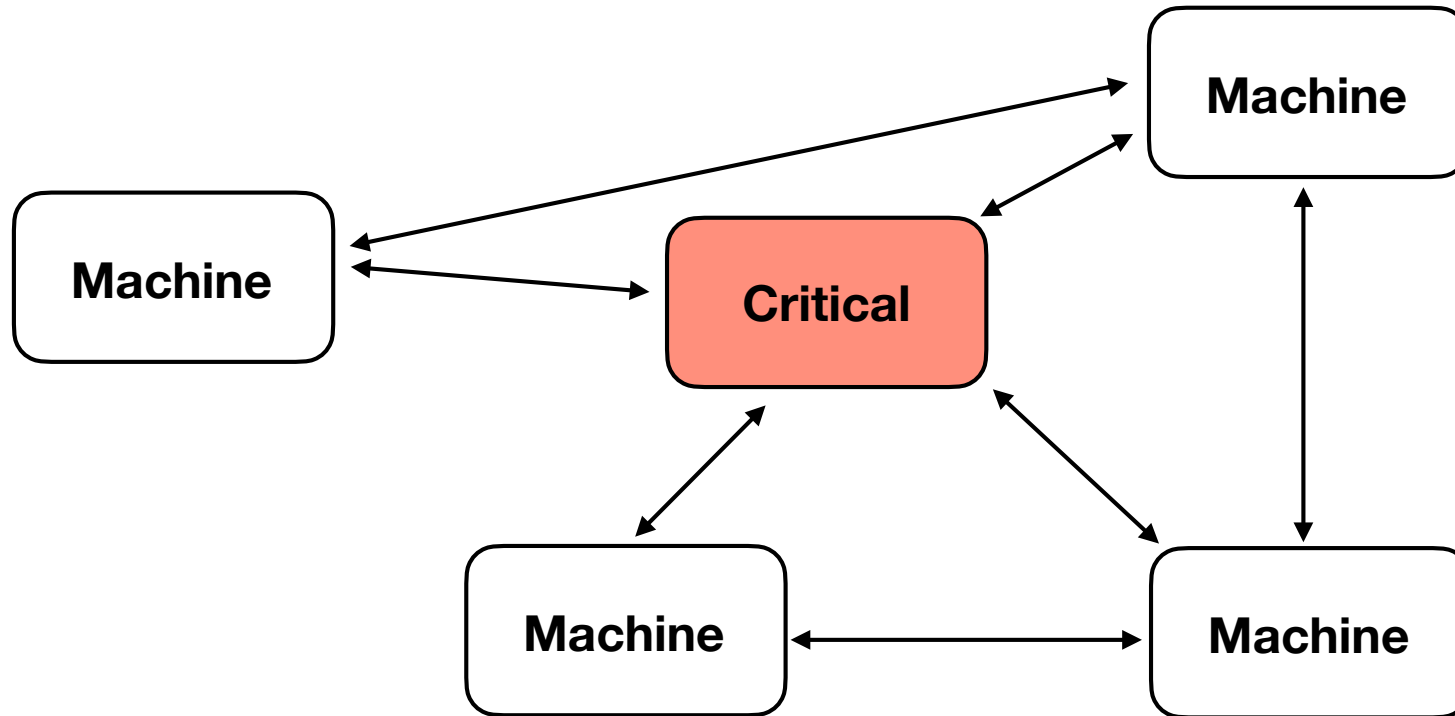
Motivations

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



Our Focus: Reliability

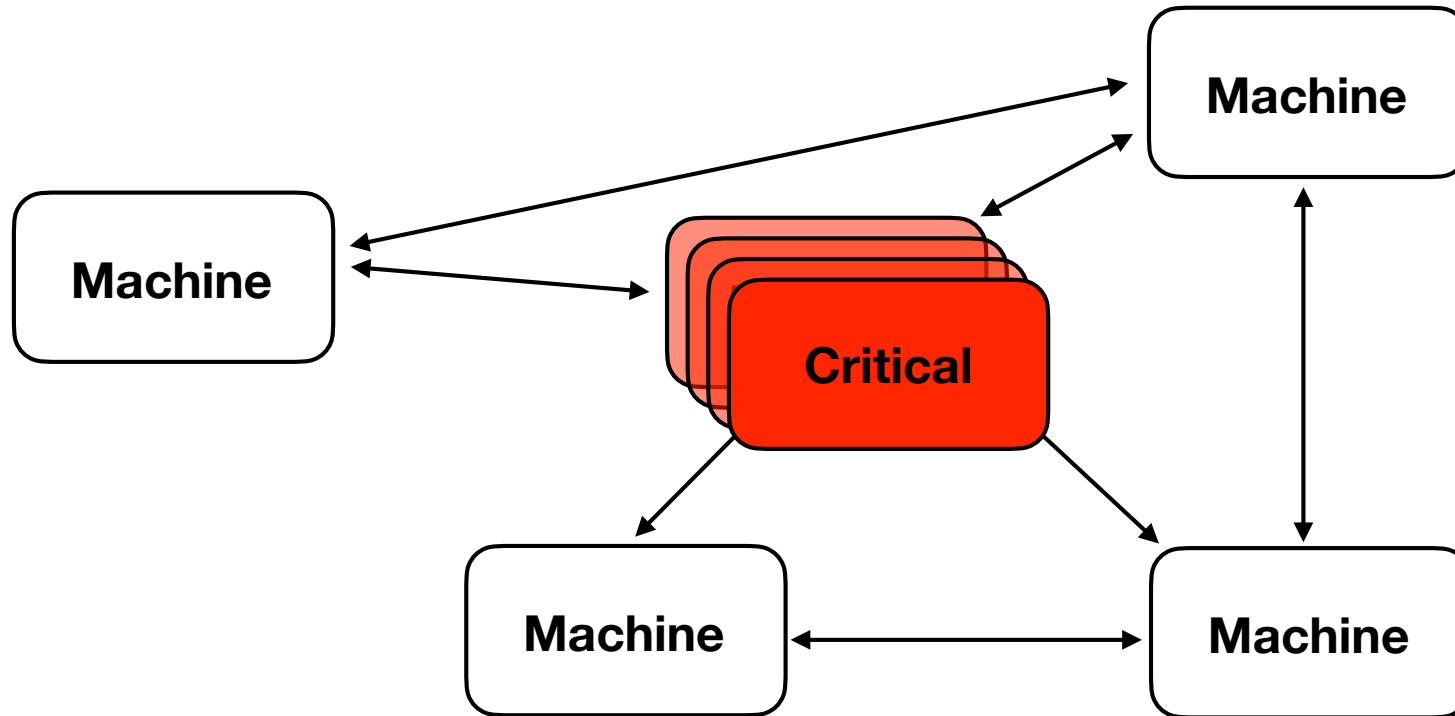
- Certain services are more critical than others



- Example: DNS, a database, etc.

Our Focus: Reliability

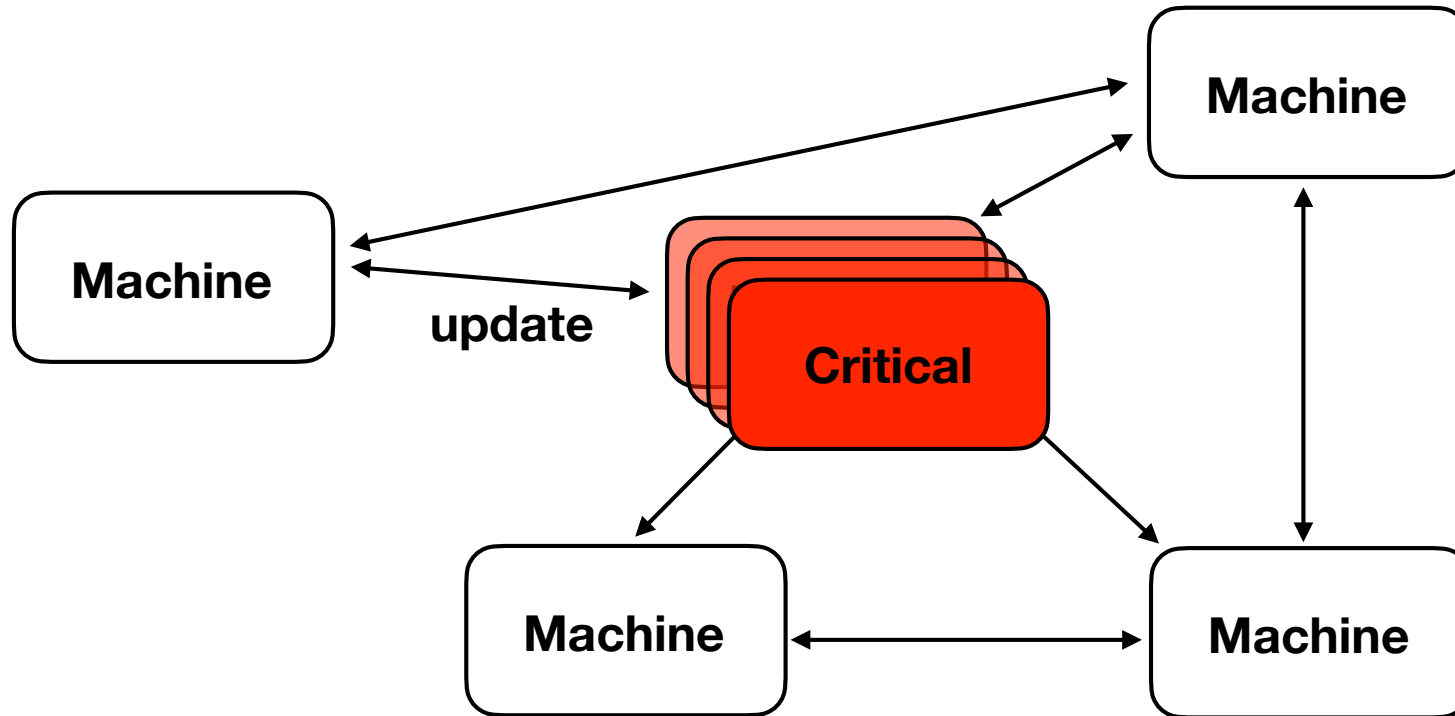
- Obvious Solution: Replication!



- Whew! Crisis averted via redundancy.

Major Problem

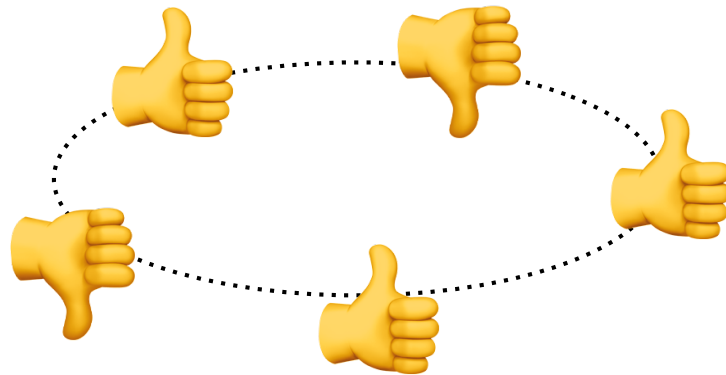
- Mutable state



- It is impossible for state to simultaneously update on all replicas at once (laws of physics, time, etc.)
- So, how do you deal with that?

Solution: Consensus

- Replicated servers require a mechanism for agreeing on the "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, incomplete "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**

2.1 Mathematical Results

The Synod's decree was chosen through a series of numbered *ballots*, where a ballot was a referendum on a single decree. In each ballot, a priest had the choice only of voting for the decree or not voting.⁵ Associated with a ballot was a set of priests called a *quorum*. A ballot succeeded iff (if and only if) every priest in the quorum voted for the decree. Formally, a ballot B consisted of the following four components. (Unless otherwise qualified, *set* is taken to mean *finite set*.⁶)

B_{dec} A decree (the one being voted on).

B_{qrm} A nonempty set of priests (the ballot's quorum).

B_{vot} A set of priests (the ones who cast votes for the decree).⁷

B_{bal} A ballot number.

A ballot B was said to be *successful* iff $B_{qrm} \subseteq B_{vot}$, so a successful ballot was one in which every quorum member voted.

Ballot numbers were chosen from an unbounded ordered set of numbers. If $B'_{bal} > B_{bal}$, then ballot B' was said to be *later* than ballot B . However, this indicated nothing about the order in which ballots were conducted; a later ballot could actually have taken place before an earlier one.

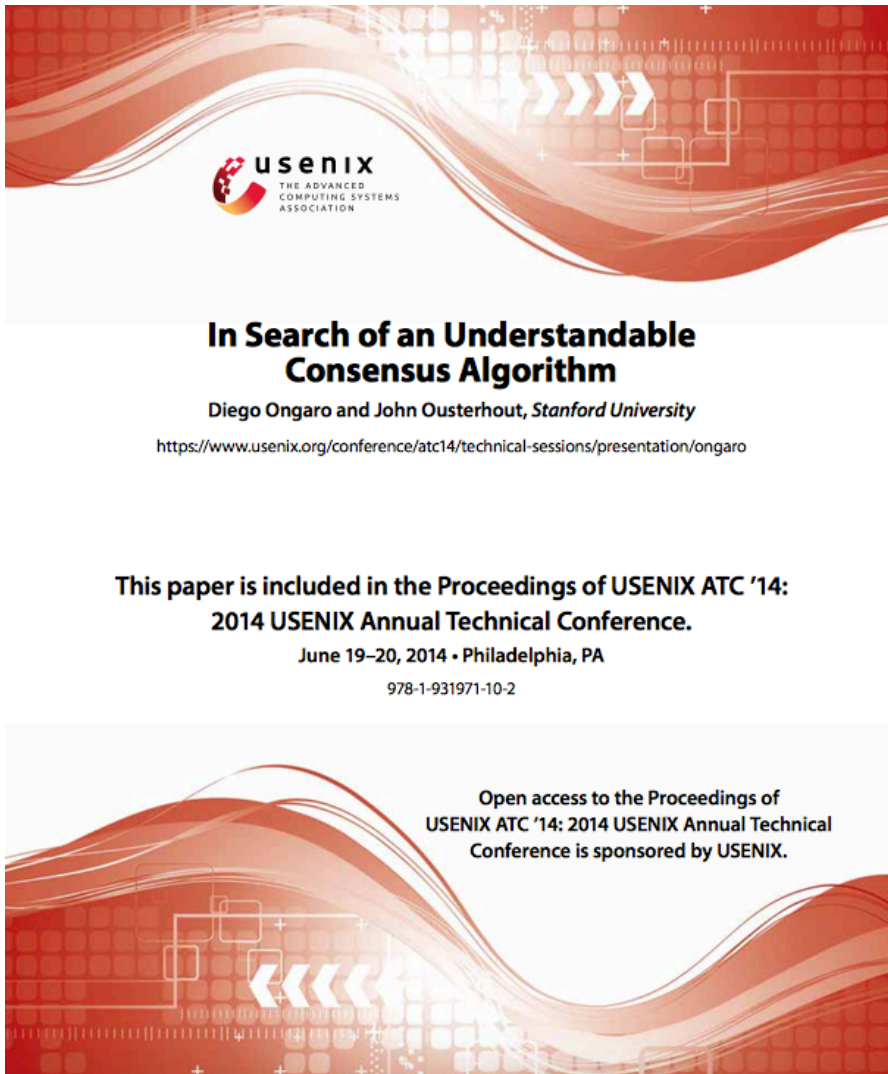
Paxon mathematicians defined three conditions on a set \mathcal{B} of ballots, and then showed that consistency was guaranteed and progress was possible if the set of ballots that had taken place satisfied those conditions. The first two conditions were simple; they can be stated informally as follows.

$B1(\mathcal{B})$ Each ballot in \mathcal{B} has a unique ballot number.

$B2(\mathcal{B})$ The quorums of any two ballots in \mathcal{B} have at least one priest in common.

The third condition was more complicated. One Paxon manuscript contained the following, rather confusing, statement of it.

Our Challenge



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

What is etcd?

Why Raft?

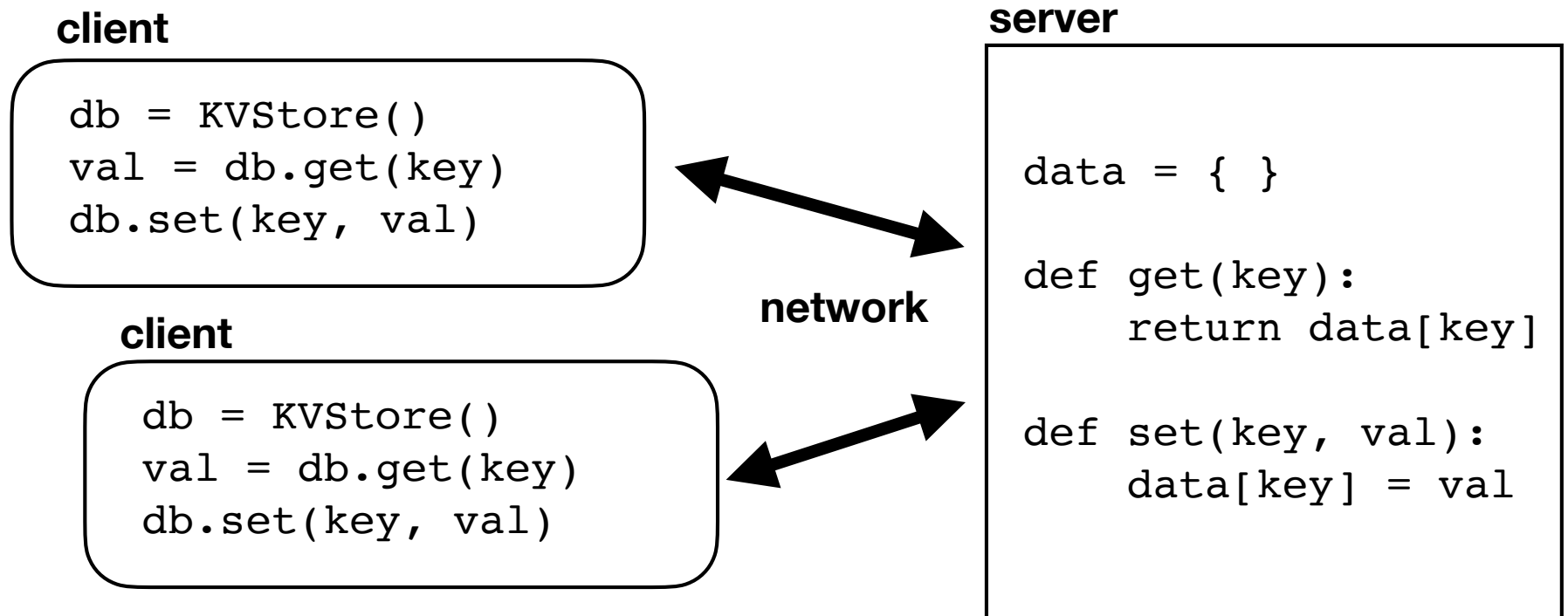
- There are many facets of distributed computing. However, distributed consensus might be the hardest.
- Fault tolerance: Cost of failure is high.
- Non-trivial: Many moving parts. Not an "echo server."
- Challenging: concurrency, networks, testing, etc.
- Solving it involves problems that transcend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

Core Topics

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

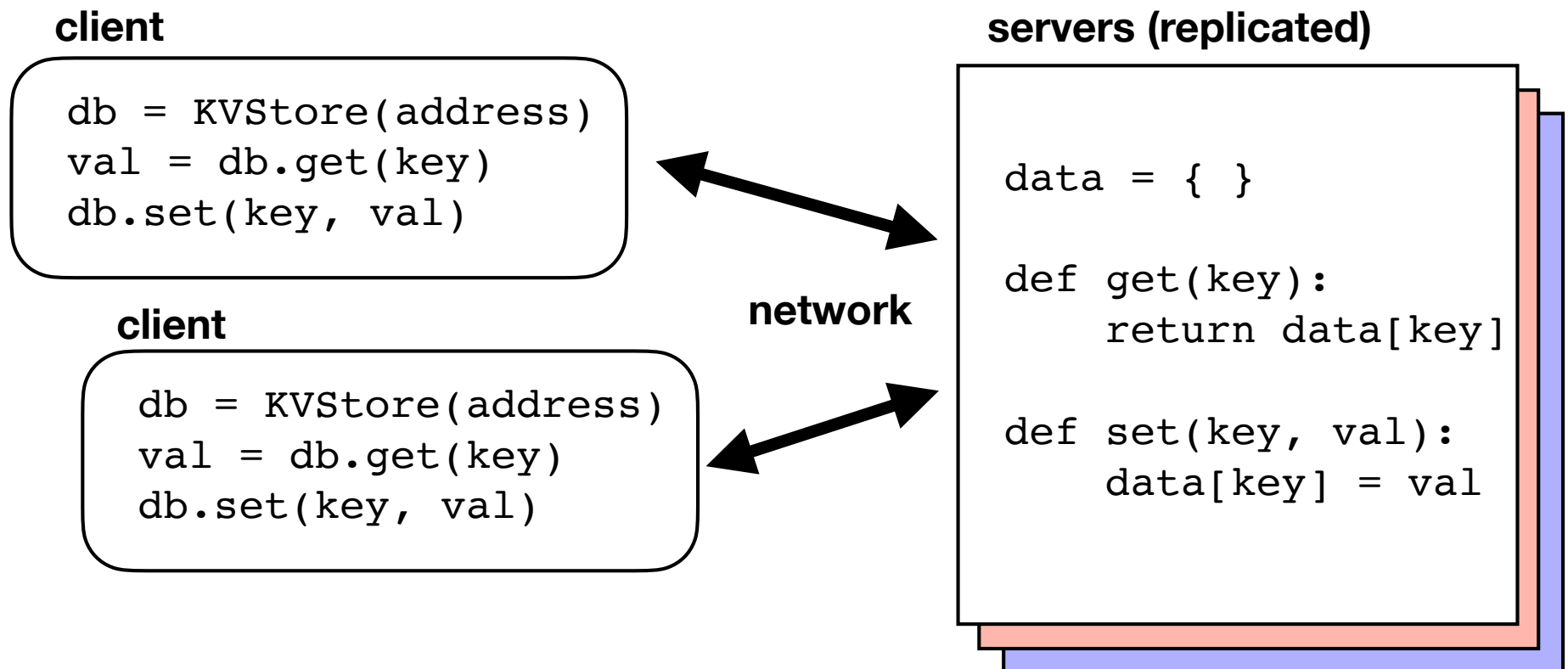
The Project

- 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



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

server

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

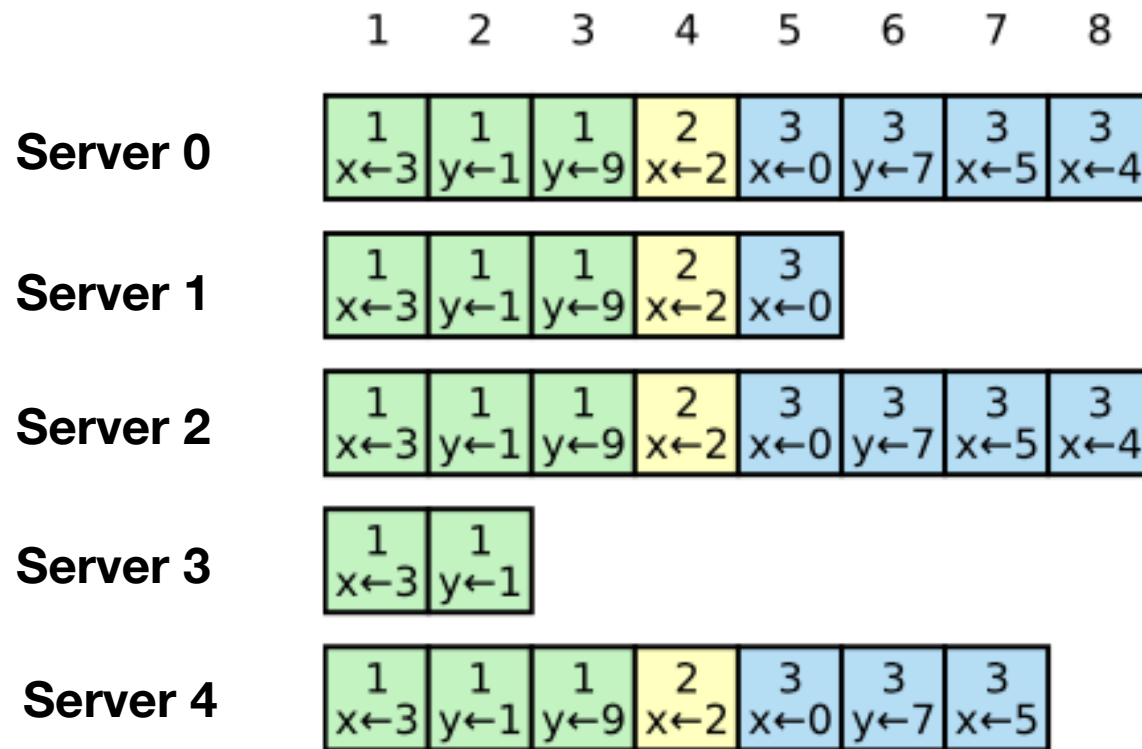
log

```
...  
set foo 42  
set bar 13  
set foo 39  
set grok 20  
delete foo  
set grok 98  
...
```

- Log keeps an ordered record of all state changes
- If you replay the log, you get back to the current state.

Replication

- Fault tolerance achieved by replicating the transaction log



Replication

- Fault tolerance achieved by replicating the transaction log

	1	2	3	4	5	6	7	8
Server 0	1 $x \leftarrow 3$	1 $y \leftarrow 1$	1 $y \leftarrow 9$					
Server 1	1 $x \leftarrow 3$	1 $y \leftarrow 1$	1 $y \leftarrow 9$					
Server 2	1 $x \leftarrow 3$	1 $y \leftarrow 1$	1 $y \leftarrow 9$					
Server 3	1 $x \leftarrow 3$	1 $y \leftarrow 1$						
Server 4	1 $x \leftarrow 3$	1 $y \leftarrow 1$	1 $y \leftarrow 9$					

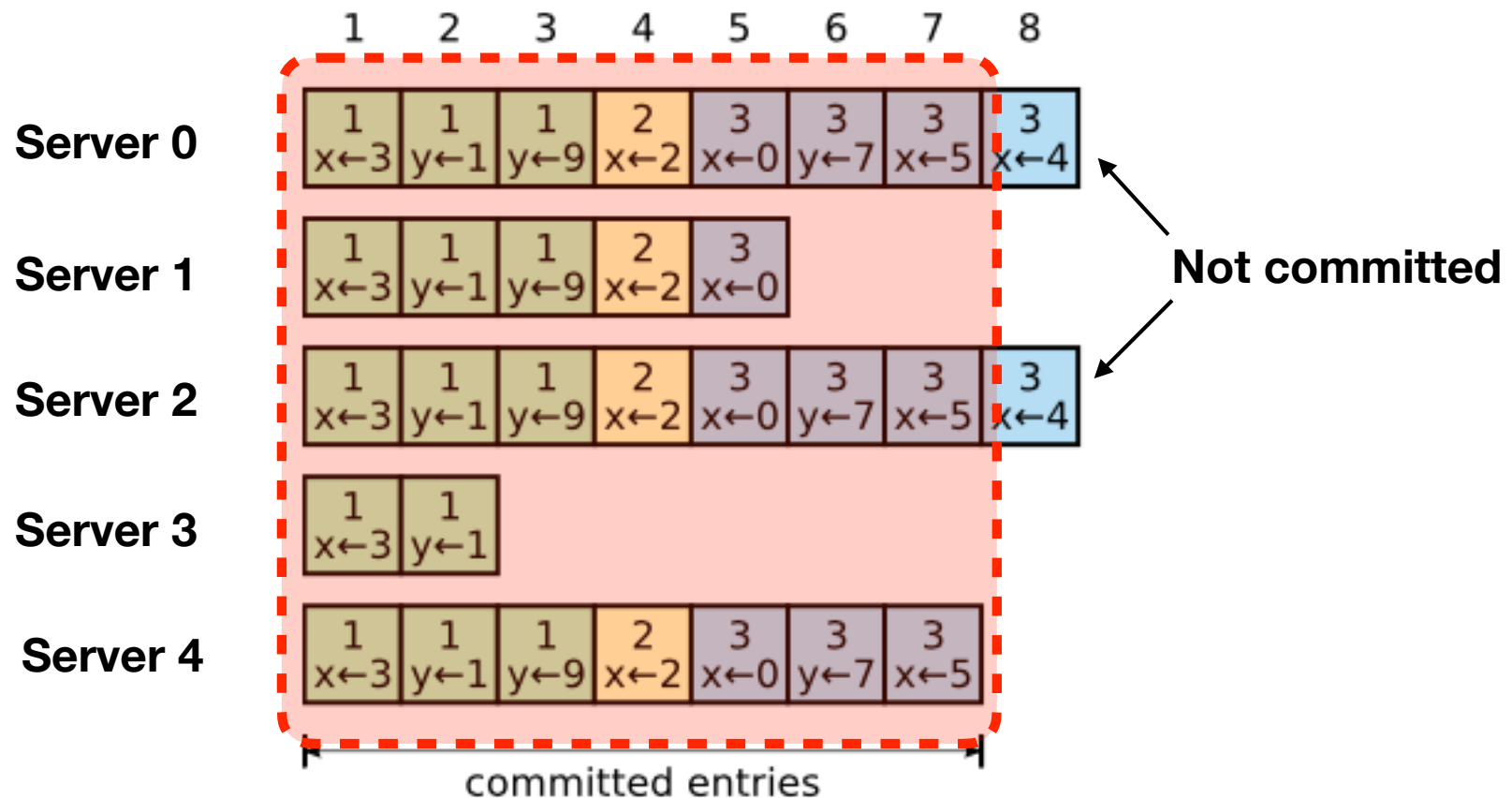
Aside:

Replicated Logs == Raft



Majority Rules

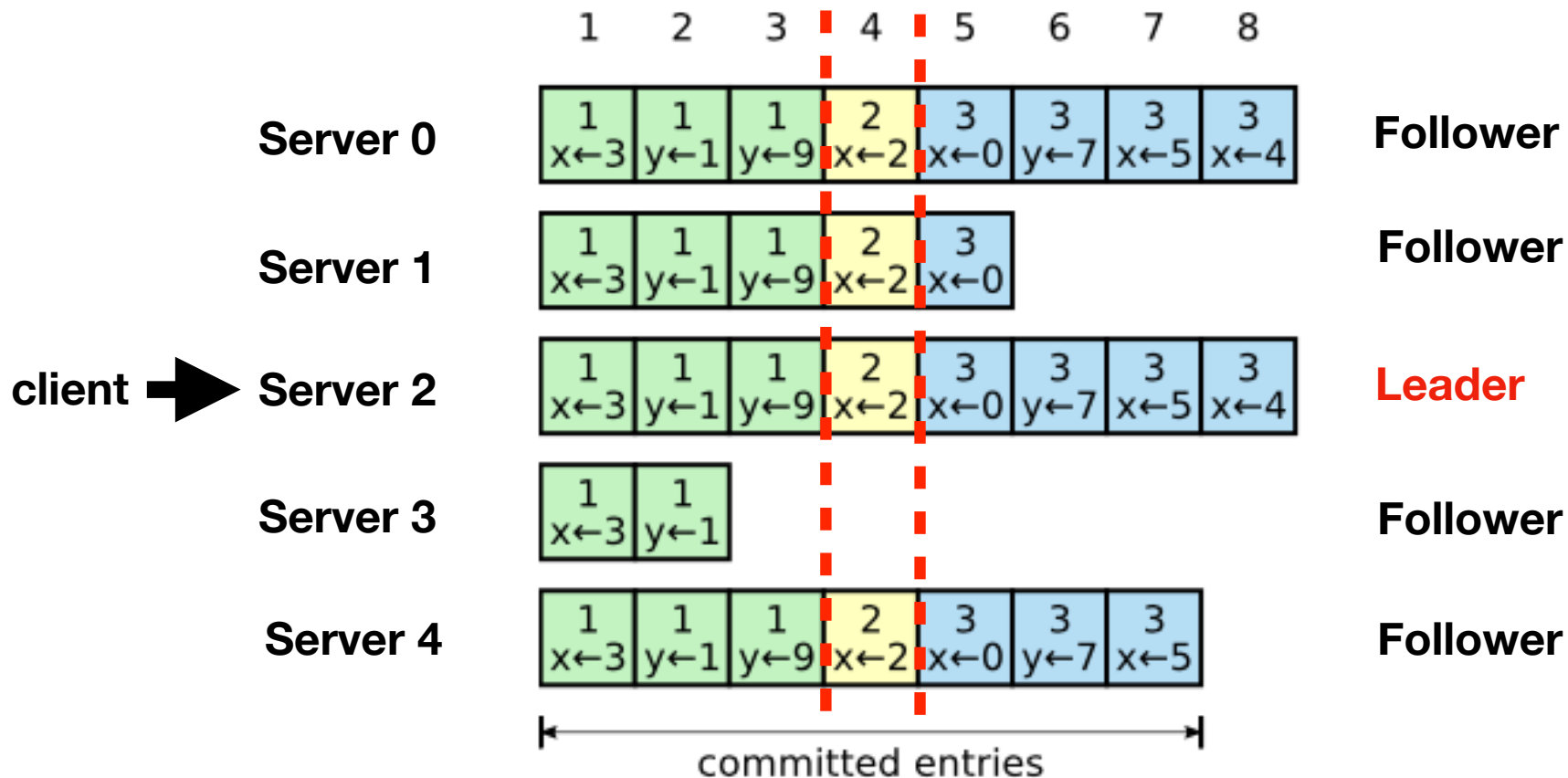
- Transactions are "committed" by consensus



- Consensus means replication on a quorum of machines

There Can Be Only One

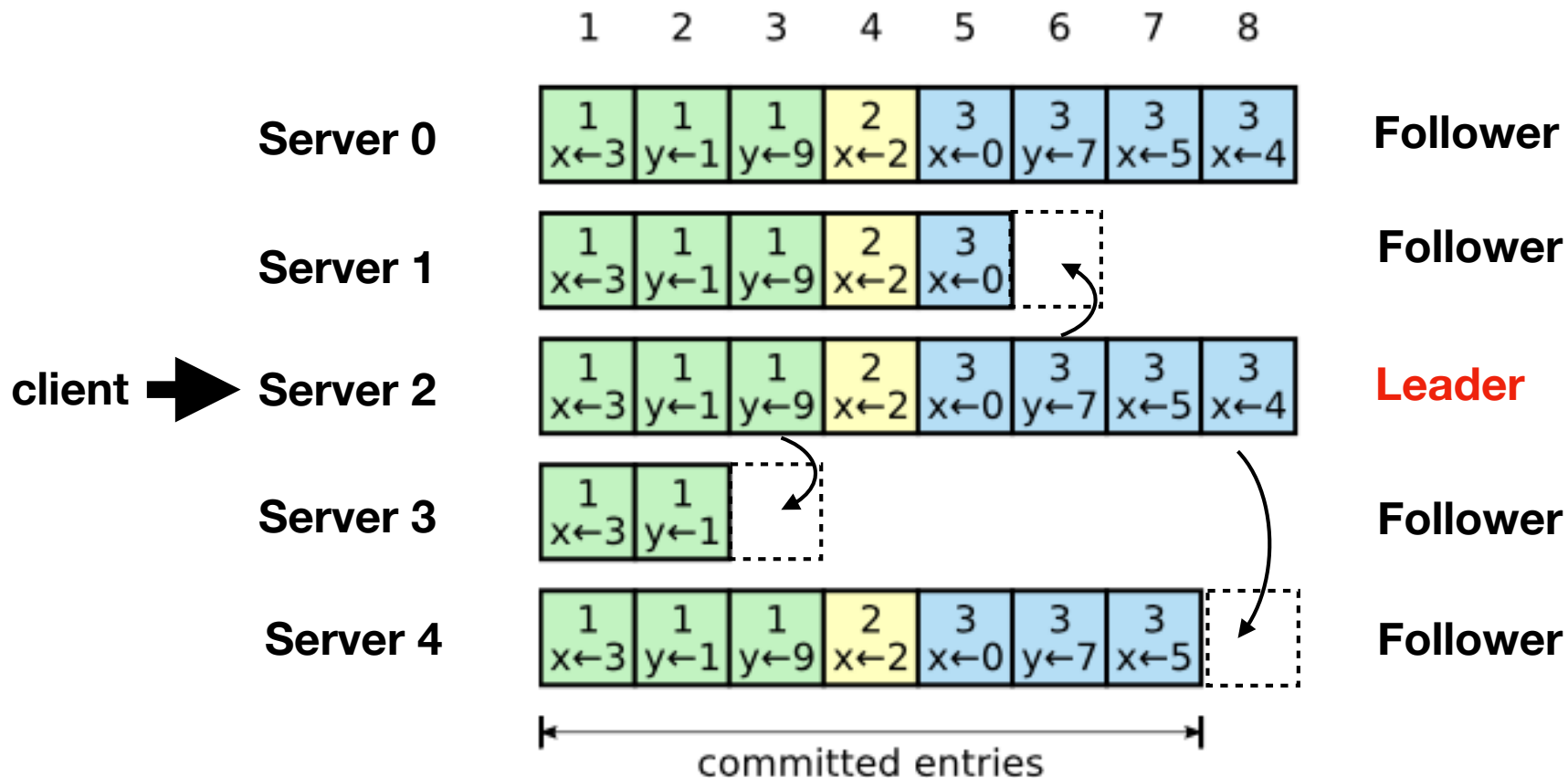
- All actions are coordinated by one and only one leader



- The leader might change over time (divided into terms)

Followers Update

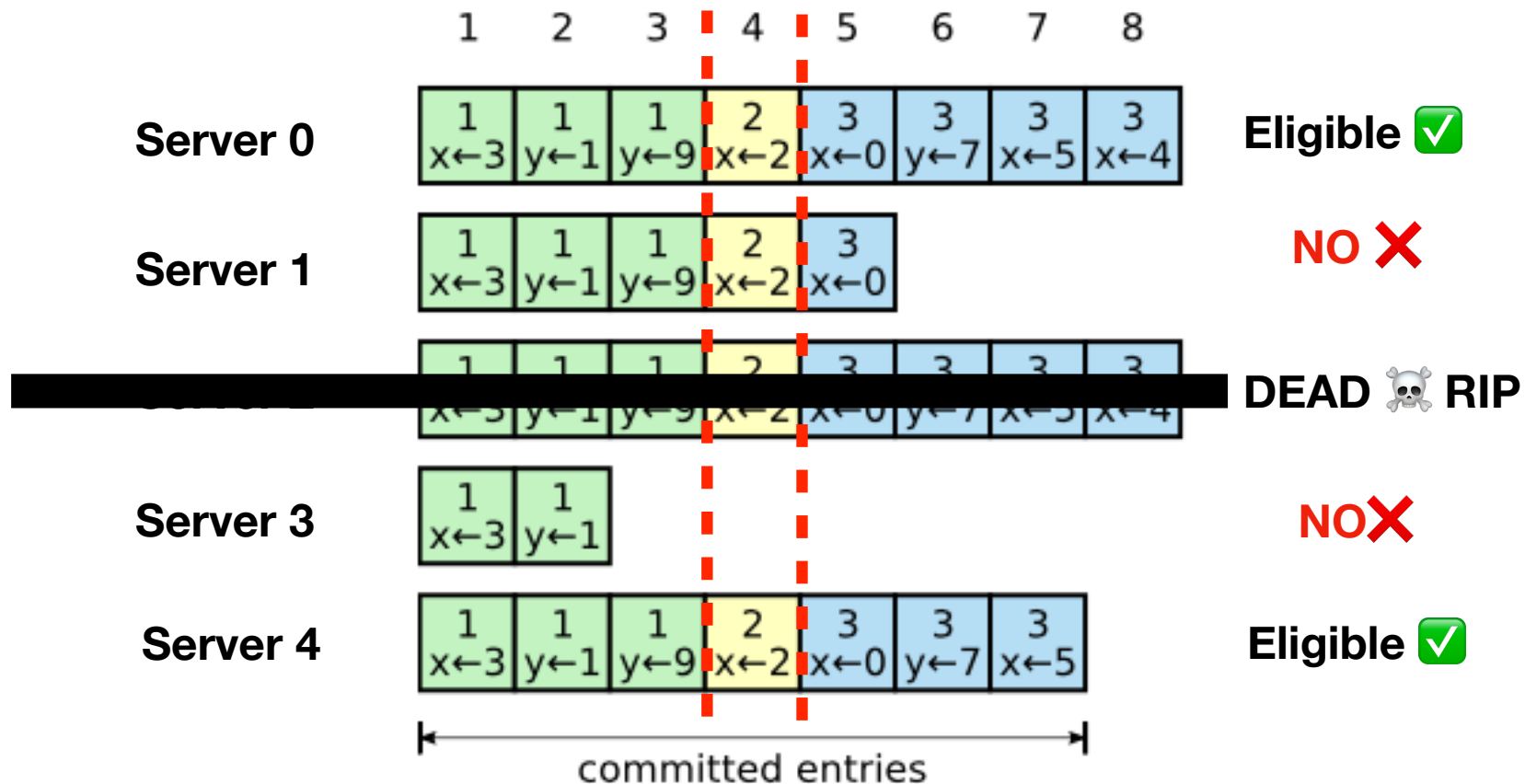
- The leader works to bring all followers up to date.



- But, there is a risk the leader could die at any time

Knowledge is Power

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



- Any quorum will have 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 cover some foundational topics
 - Technical: Sockets, networks, messages, etc.
 - Mathematical: State machines, formal models, etc.
- 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

How to Fail

- **Testing:** Testing is important, but it's easy to spend too much time testing the wrong thing. Most of the difficulty in Raft is in the integration of the parts and making them work together. It is very hard to test as a whole. Better strategy: make each part testable, but also focus on monitoring and debugging.
- **Analysis Paralysis:** Spending too much time thinking about software architecture, OO design patterns, and the "right way" to do things in the face of uncertainty. Don't overthink the problem. Pick a strategy and go with it. Plan for refactoring. Keep. It. Simple.

How to Fail

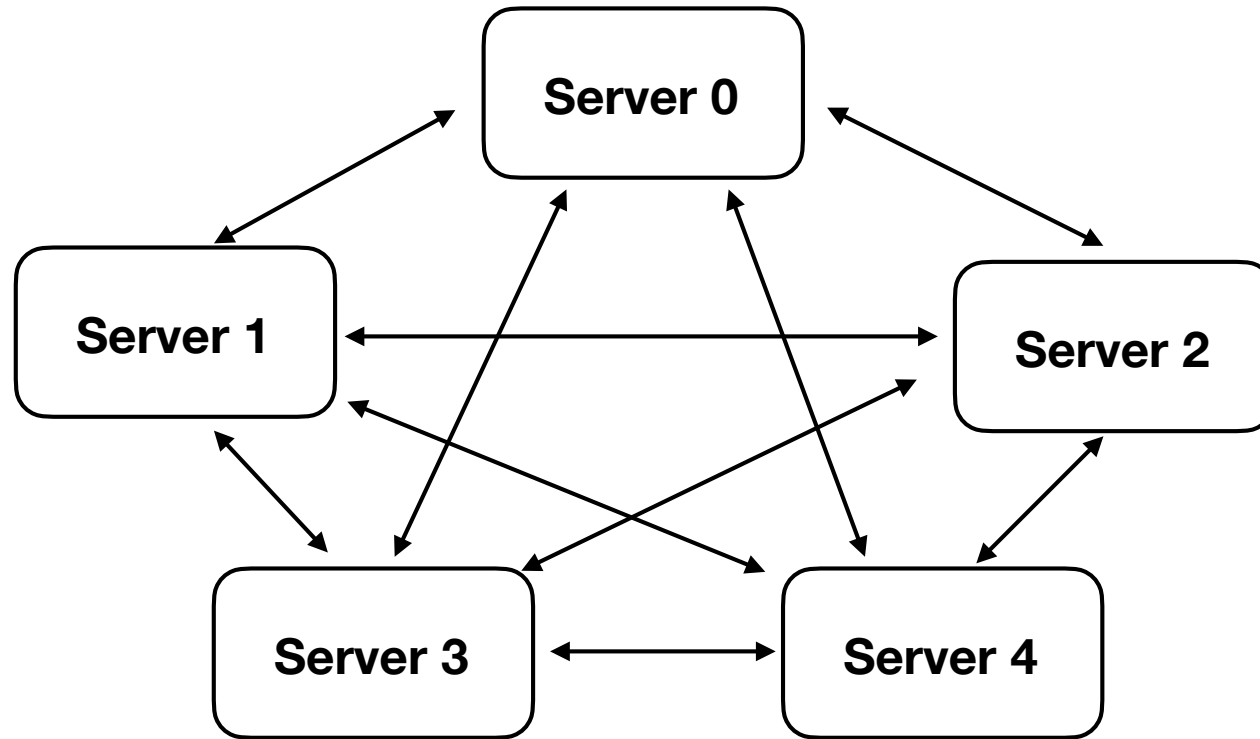
- **Detail Paralysis:** The Raft paper can be challenging to read. Don't get too bogged down in tiny details at first. Trust your intuition in devising a solution. I am going to try and guide you through the hairy parts.
- **Silent Suffering:** Don't spend hours trying to track down some bug or being confused about part of the Raft paper/specification. Bring attention to it so we can discuss as a group (others are likely having similar issues).
- **Distractions:** Try to avoid working on real work.

Part 1

Messaging

Raft and Networks

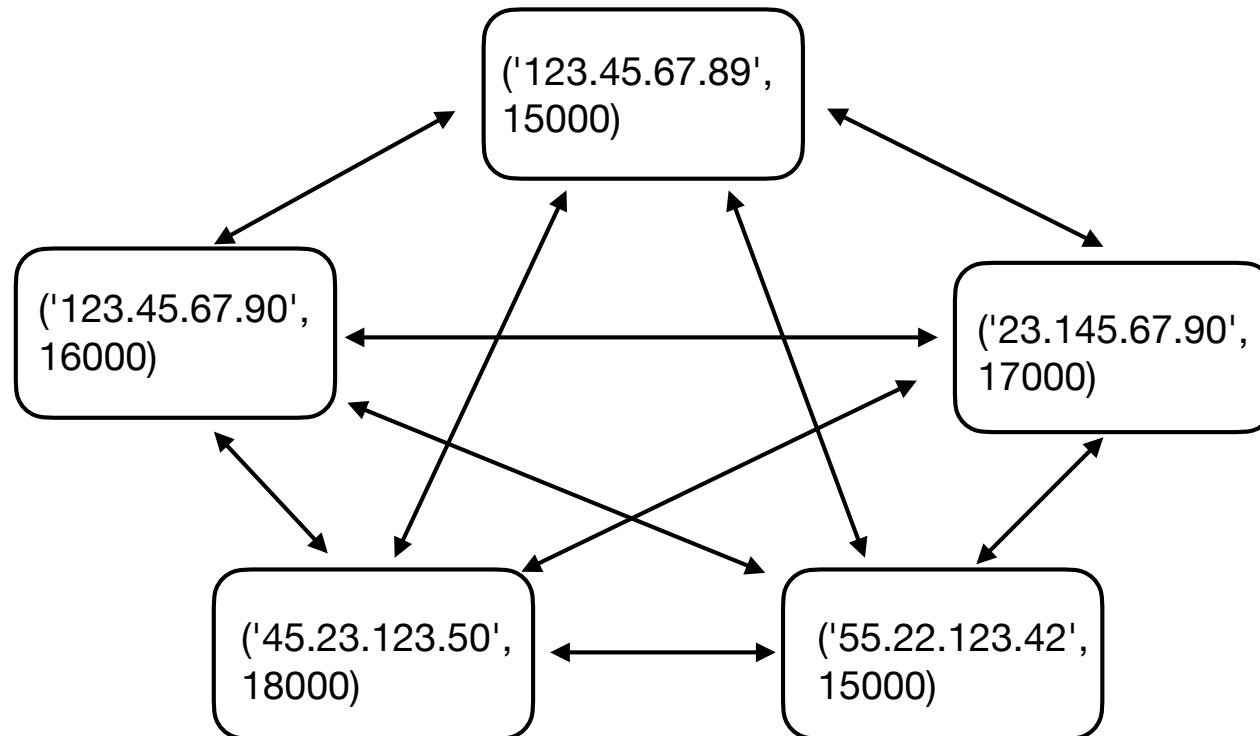
- Raft involves a cluster of identical servers



- They exchange messages over a network

Network Addressing

- Servers are identified by (hostname, port)



- The addresses are part of the configuration

Data Transport

- 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
    ...
```

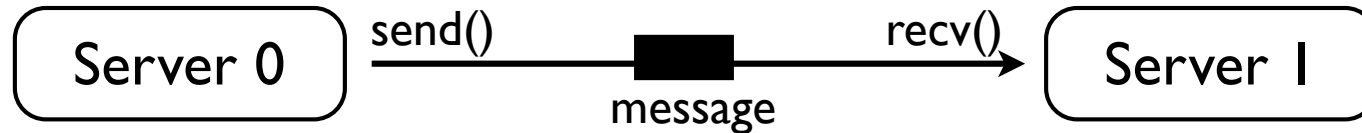
- Sending raw data on a socket

```
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 Passing



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

Message Encoding

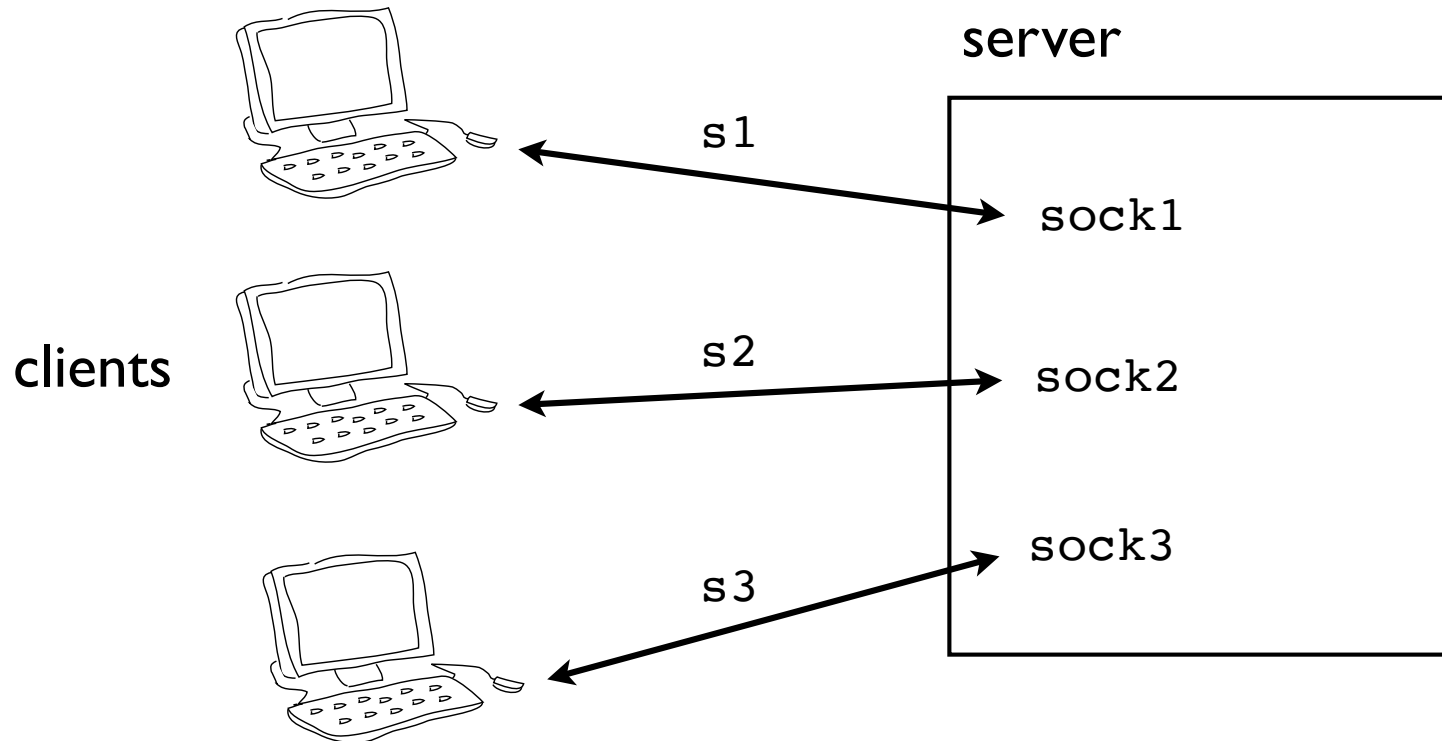
- Common: size-prefixed byte vector



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

Concurrency

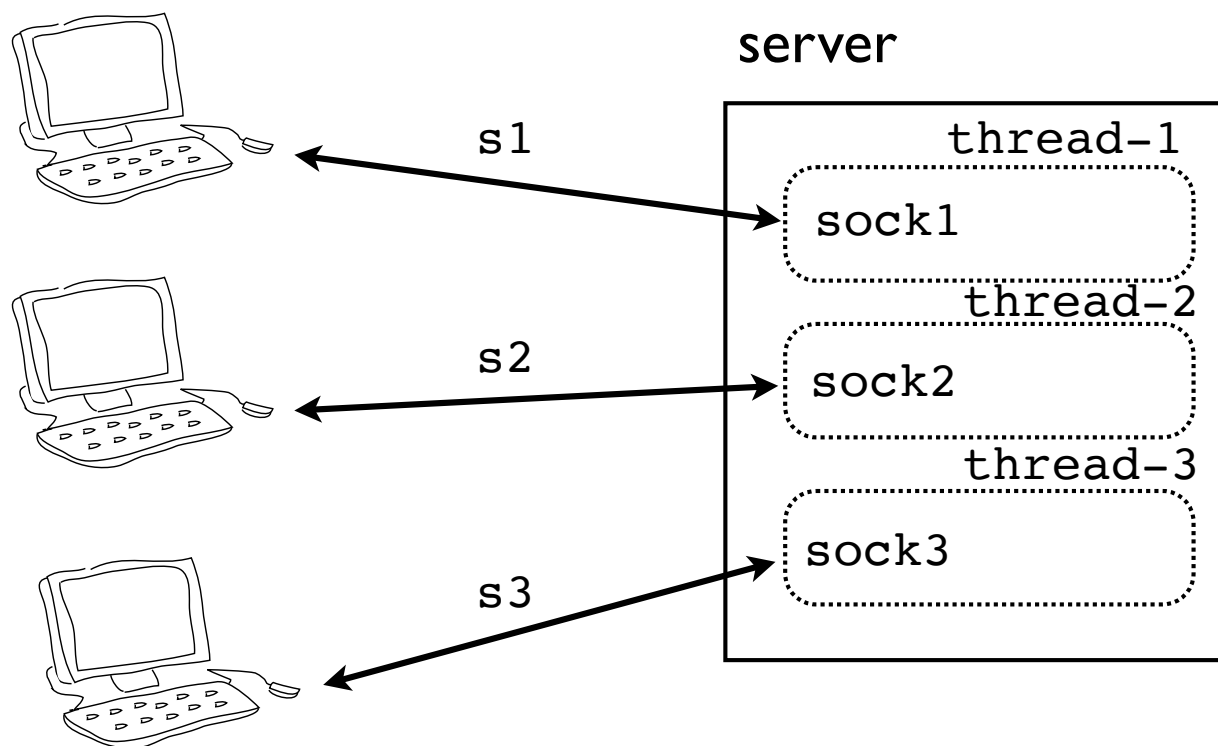
- In a distributed world, servers often need to interact with multiple connections at once



- How to coordinate execution?

Multiple Clients

- Solution: handle each connection in a thread



- Independent handling of each client

Thread Basics

server.py



statement

statement

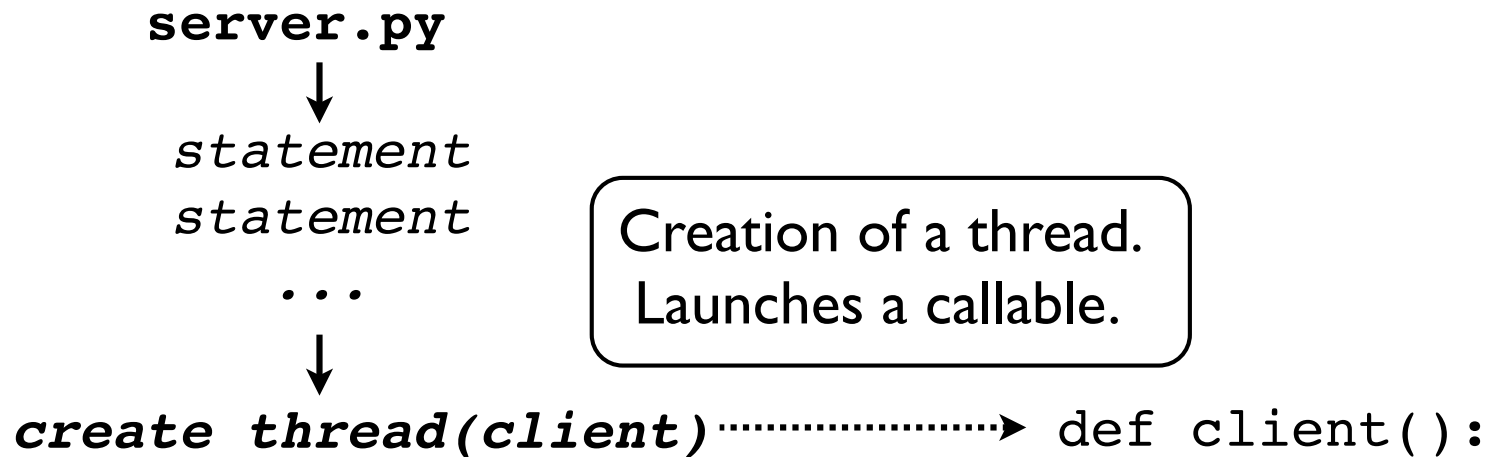
...



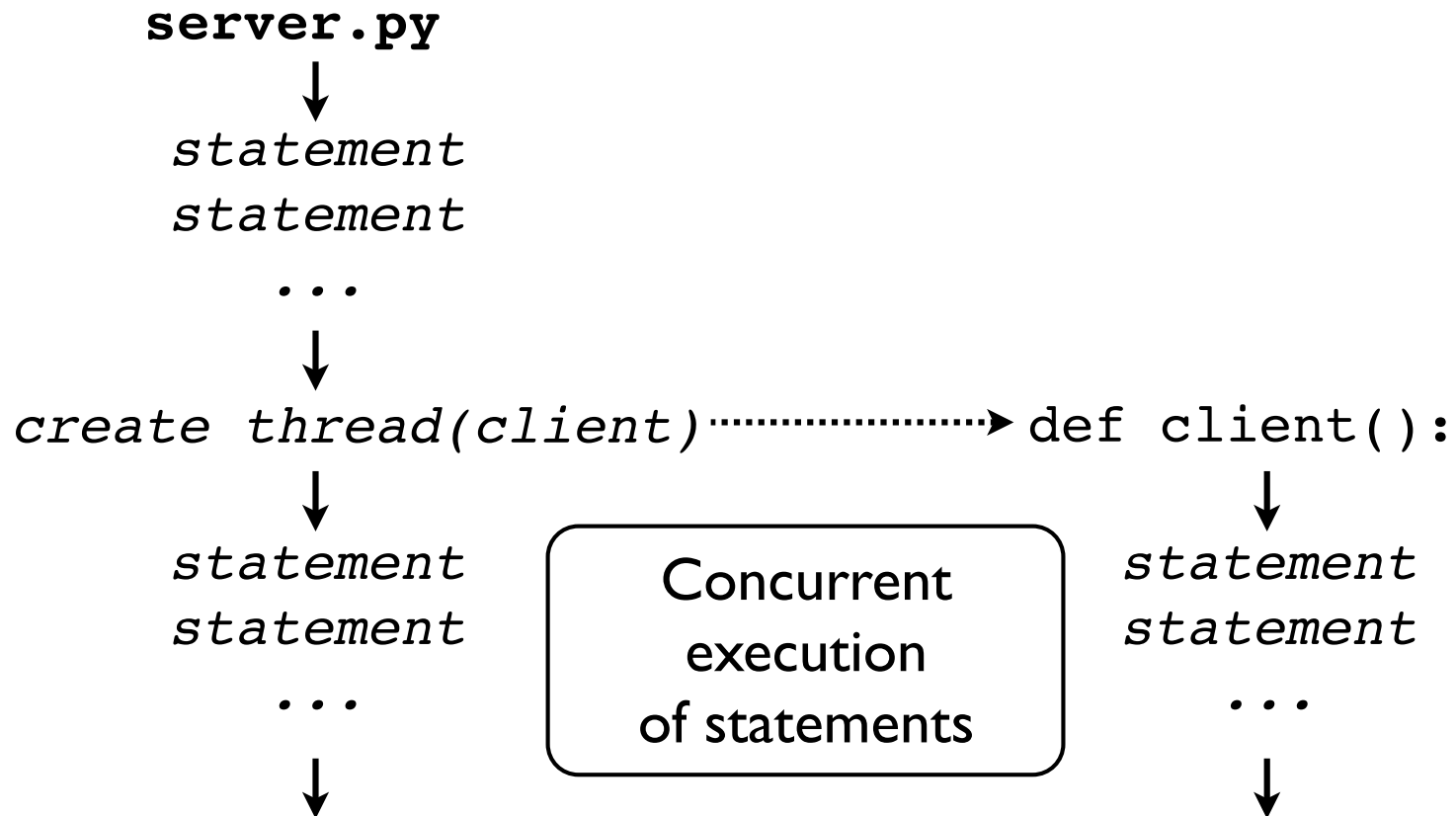
"main thread"

Program launch.
Program starts
executing statements

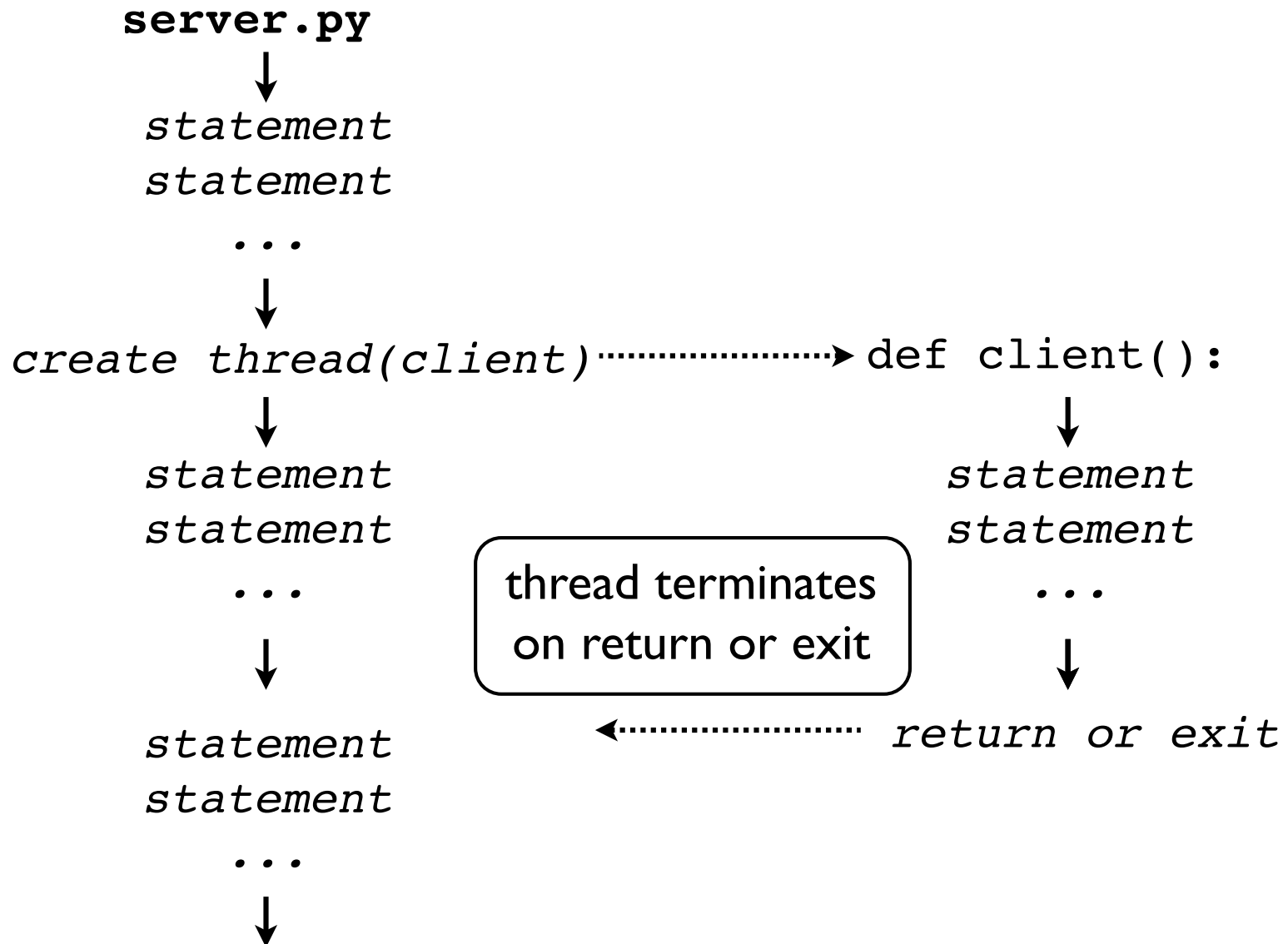
Thread Basics



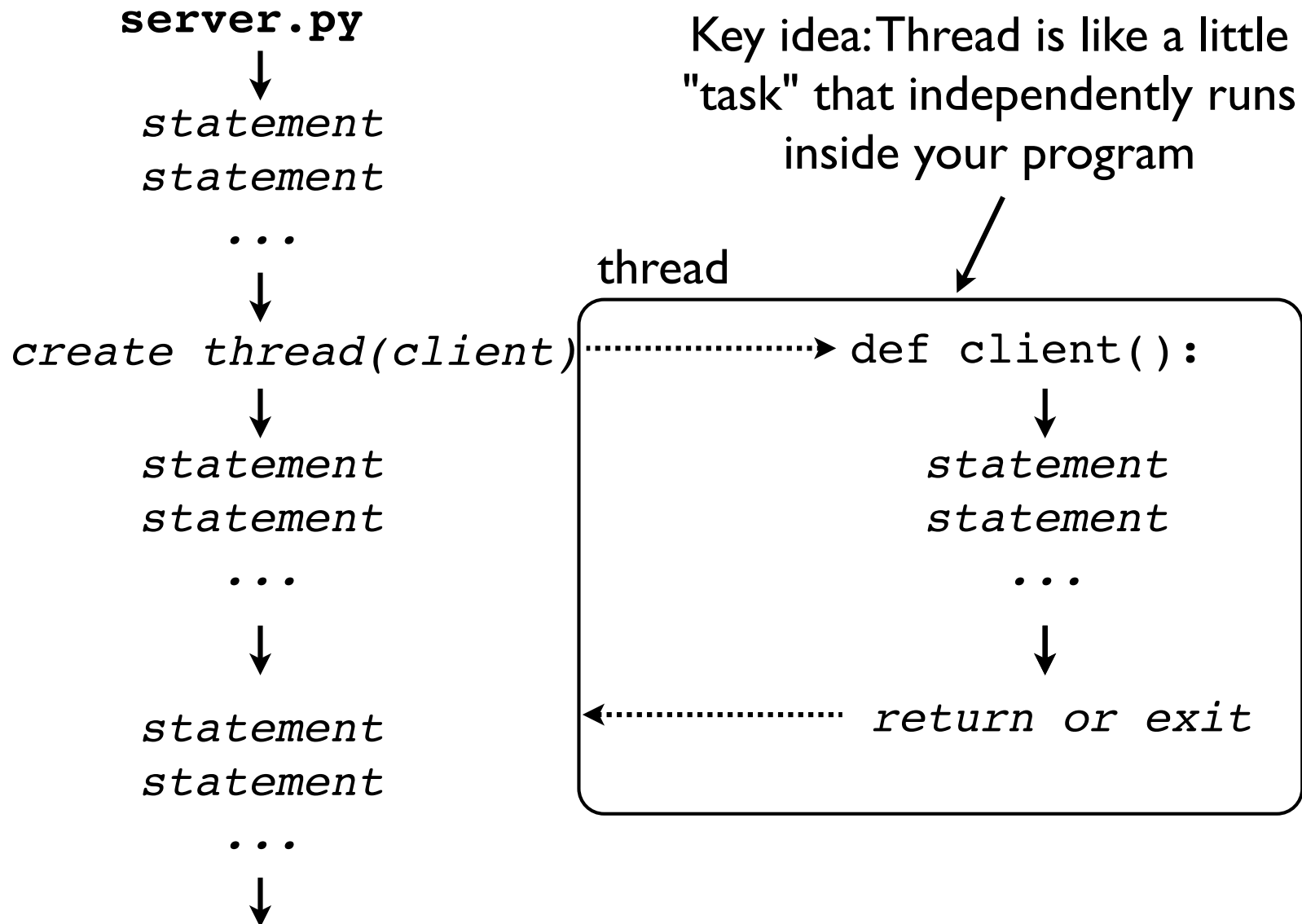
Thread Basics



Thread Basics



Thread Basics



Python Threading

- How to launch a thread

```
# 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()
    ...
```

- Once launched, thread runs independently

Commentary

- Programming with threads is difficult.
- They share memory and run independently
- It can be quite difficult to properly coordinate thread execution (often involves locks)

Concurrent Updates

- Consider a shared value

```
data = { 'x': 0 }
```

- What if there are concurrent updates?

Thread-1

...

```
data[ 'x' ]=data[ 'x' ]+1
```

...

Thread-2

...

```
data[ 'x' ]=data[ 'x' ]-1
```

...

- Here, it's possible that the resulting value will be corrupted due to thread scheduling

Concurrent Updates

- The two threads

Thread-1

...

data['x']=data['x']+1

...

Thread-2

...

data['x']=data['x']-1

...

- Low level code execution

Thread-1



t1 = LOAD data['x']

t2 = LOAD 1

t3 = t1 + t2

STORE t3, data['x']

Thread-2



thread
switch

t1 = LOAD data['x']

t2 = LOAD 1

t3 = t1 - t2

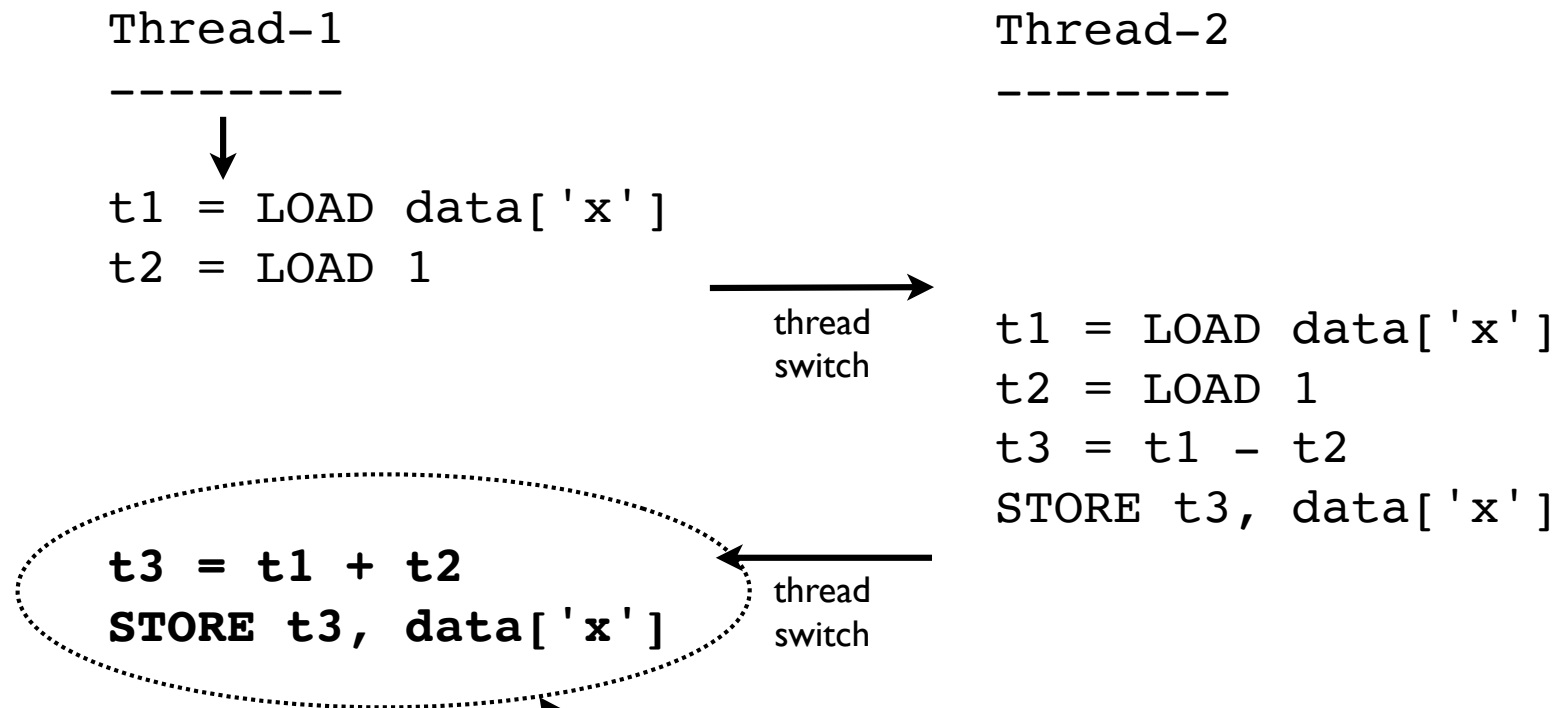
STORE t3, data['x']



thread
switch

Concurrent Updates

- Low level code execution



These operations get performed with a "stale" value of x. The computation in Thread-2 is lost.

Mutex Locks

- How to safely update shared data

```
x = 0
x_lock = threading.Lock()
```

	Thread-1	Thread-2
	-----	-----

	x_lock.acquire()	x_lock.acquire()
Critical Section	x = x + 1	x = x - 1
	x_lock.release()	x_lock.release()

- Only one thread can execute in critical section at a time (lock gives exclusive access)

Events

- How to make a thread wait for something

```
x = 0
x_event = threading.Event()
```

Thread-1

...

x = 42

x_event.set()

...

Thread-2

...

x_event.wait()

print(x)

...

signals

- Caution : Events only have one-time use

Prefer Queues

- Instead of shared-state, it is often easier to program threads using queues
- Idea: Have threads send "messages" to each other through a shared queue structure

Thread Queues

```
from queue import Queue

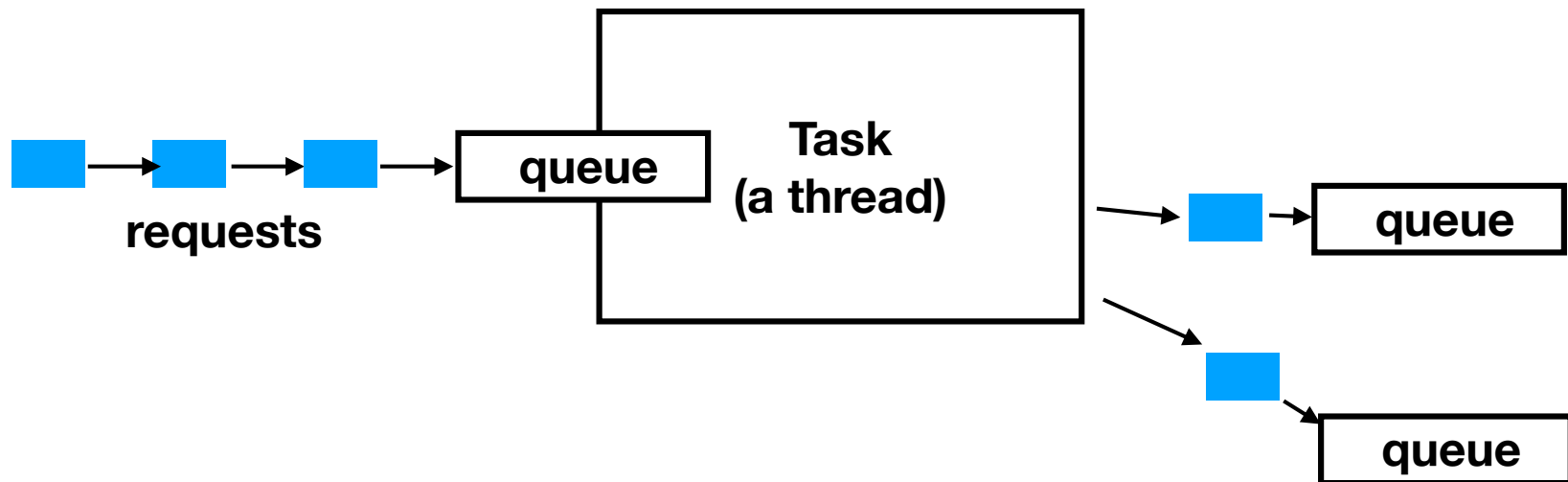
def producer(q):
    for i in range(10):
        q.put(i)
        time.sleep(1)
    q.put(None)

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 Note

- Almost everything in the Raft project can be structured around "messaging" via queues
- Doing this will likely simplify a lot things



Project I

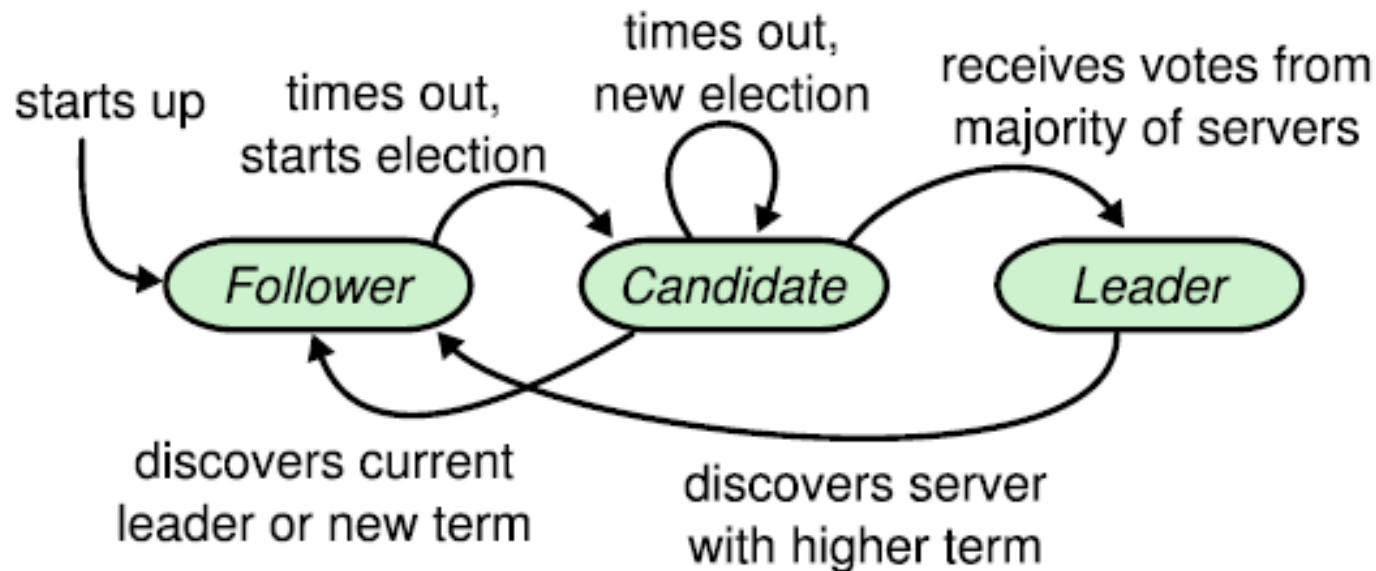
- Implement a low-level message transport layer
- Test it by implementing a key-value store
- Modify to support multiple clients if you can
- **KEEP.IT.SIMPLE.**

Part 2

State Machines

Raft Operational States

- Servers in Raft operate in different states



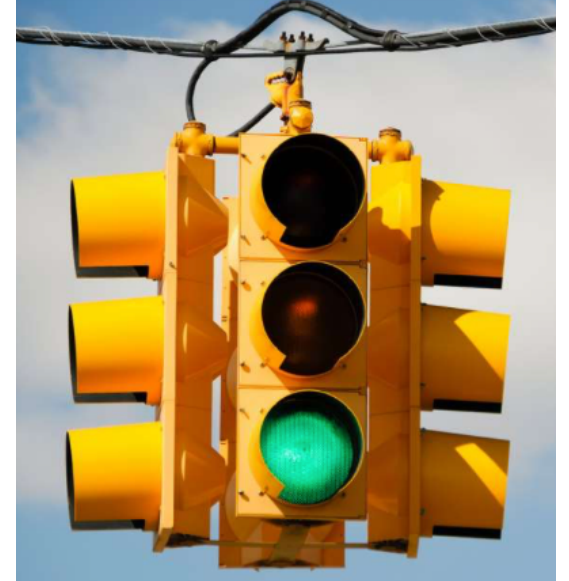
- A major complexity concerns the transitions between the different states

State Machines

- To better handle Raft, will focus on working through a simpler state example first
- Goal is to work out some mechanics of implementing and thinking about state machines

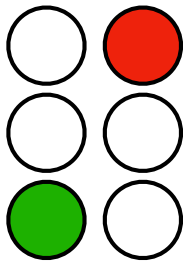
A Simple Example

- A traffic light
- What are its operational states?



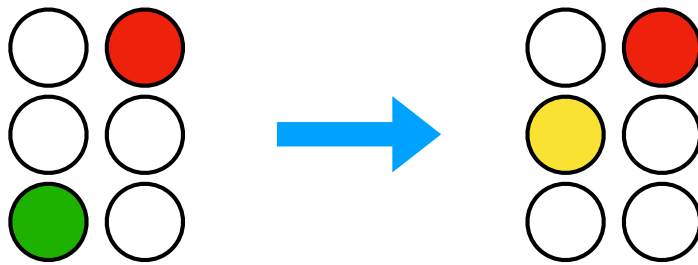
A Simple Example

- A traffic light
- What are its operational states?



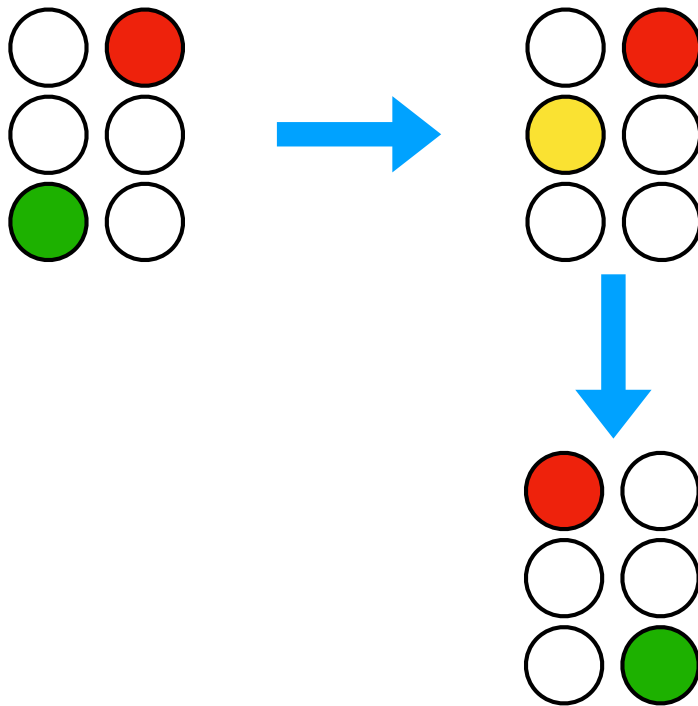
A Simple Example

- A traffic light
- What are its operational states?



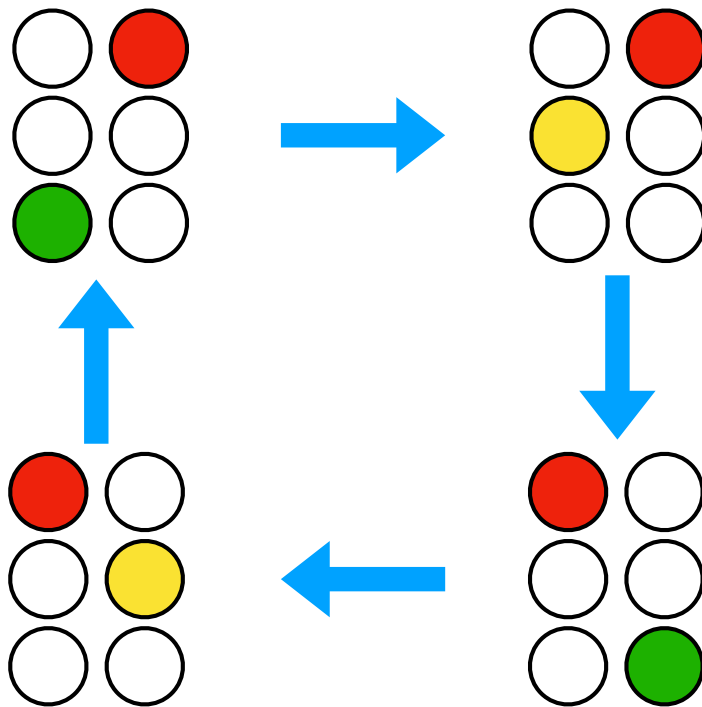
A Simple Example

- A traffic light
- What are its operational states?



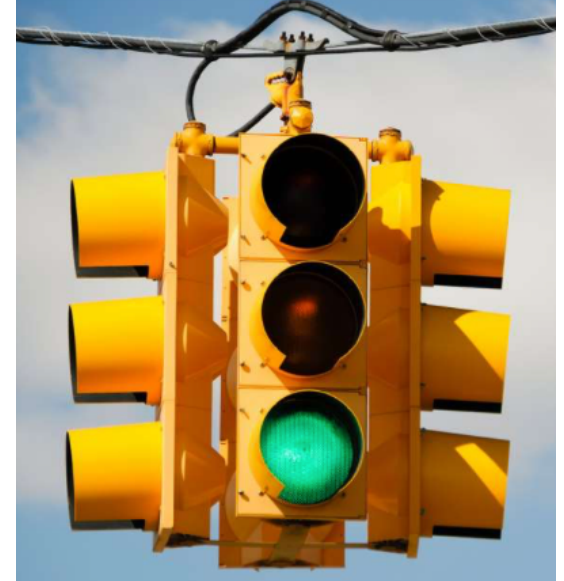
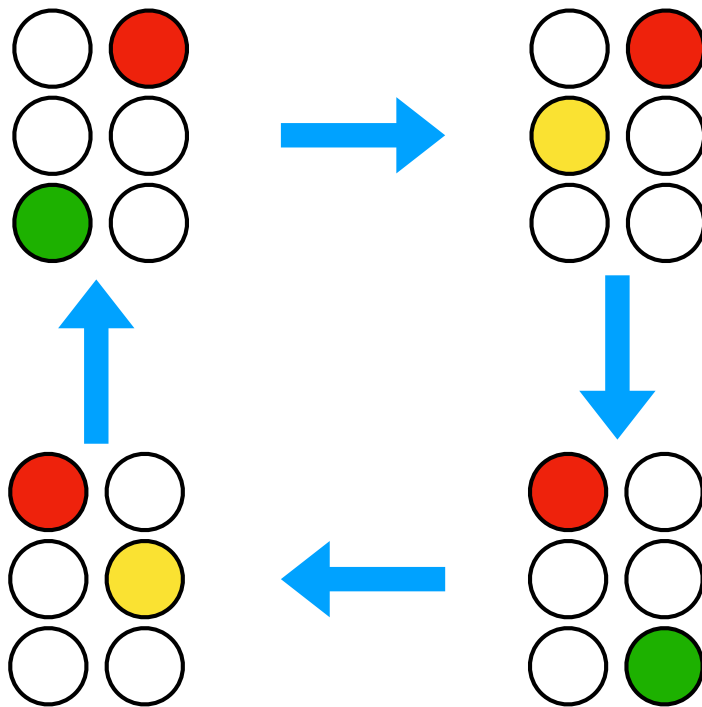
A Simple Example

- A traffic light
- What are its operational states?



A Simple Example

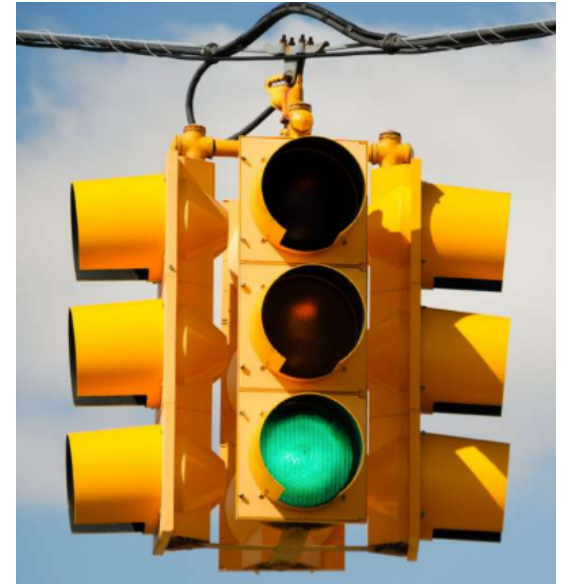
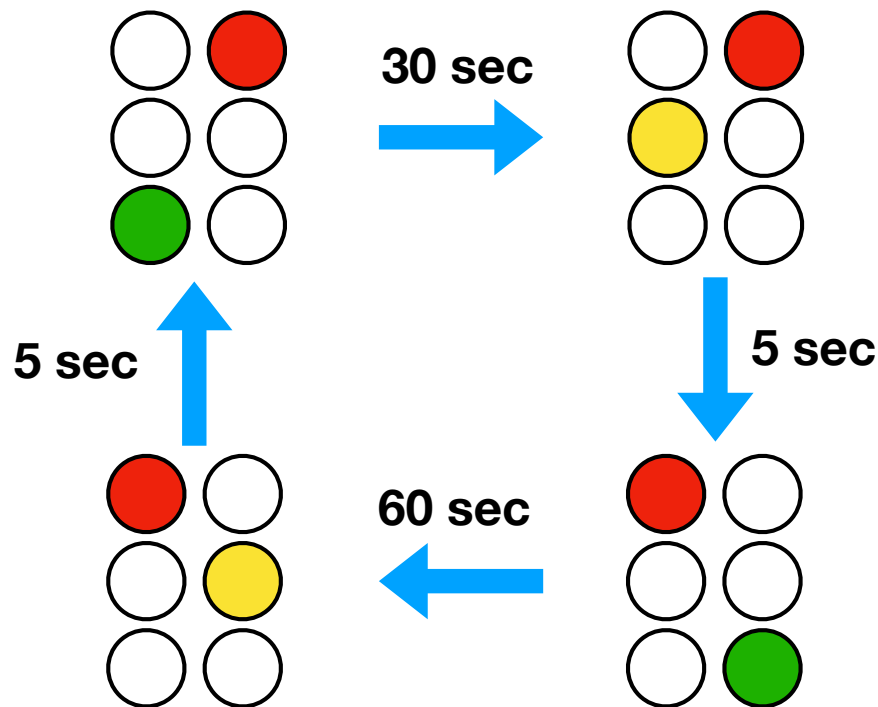
- A traffic light
- What are its operational states?



- What causes states to change?

A Simple Example

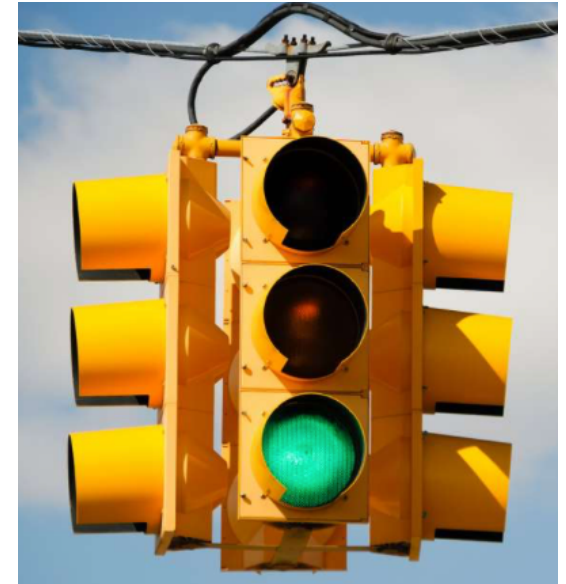
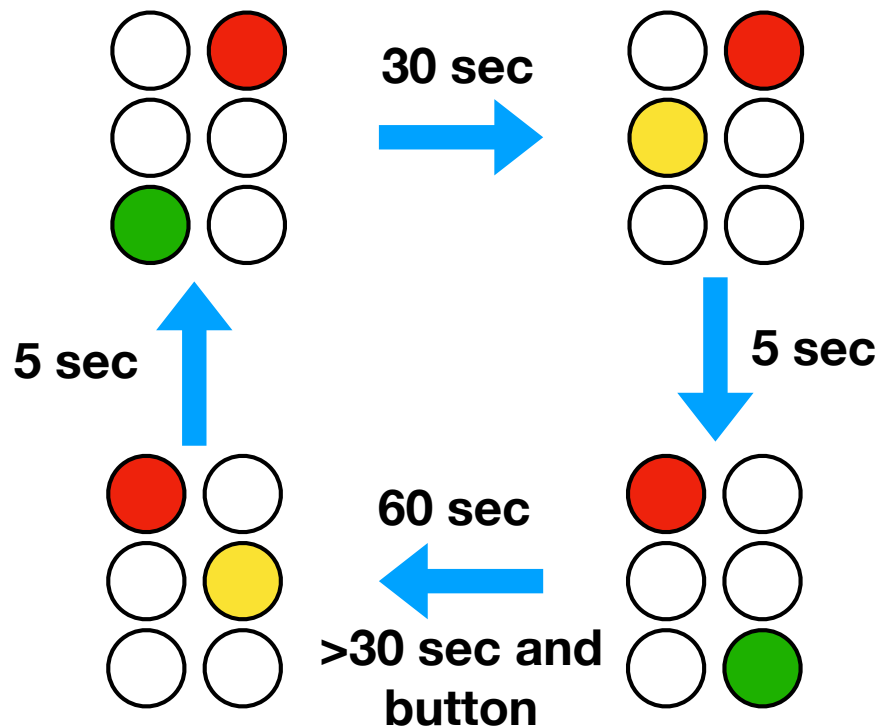
- A traffic light
- What are its operational states?



- What causes states to change? (timer events)

A Simple Example

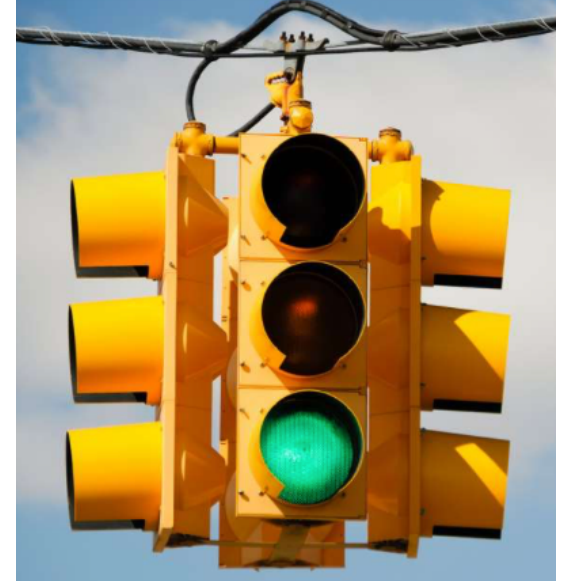
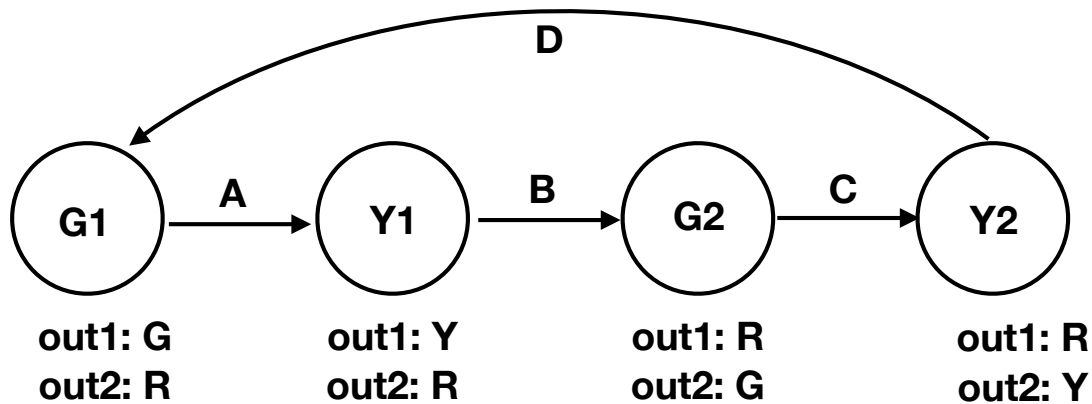
- A traffic light
- What are its operational states?



- What causes states to change? (events)

State Abstraction

- Draw a diagram, label it



- States are named (circles)
- Arrows represent "events" (cause state change)

A: 30s timer

B: 5s timer

C: 60s timer or (walk_button and \geq 30s timer)

D: 5s timer

Possible Implementation

- Object with event methods

```
class TrafficLightMachine:
    def __init__(self):
        self.state = "G1"
        self.clock = 0
        self.button_pressed = False

    def handle_clock(self):
        self.clock += 1
        ...

    def handle_button(self):
        self.button_pressed = True
        ...
```

- Plus logic for changing states (not shown)

Use of a Controller

```
class TrafficLightMachine:
    def __init__(self, control):
        self.control = control
        self.state = "G1"
        ...
    def handle_clock(self):
        ...
        if self.state == 'G1':
            self.control.set_out1("G")
            self.control.set_out2("R")

class TrafficLightControl:
    def __init__(self):
        self.machine = TrafficLightMachine(self)
        ...
```

Thoughts

- You may want to separate the state machine logic from the runtime environment
- Significantly easier to test
- Easier to repurpose
- Easier to refactor the runtime (threads vs. async)

Project 2

- Write software for a traffic light (see Exercise)
- Not directly Raft related, but the same techniques will prove useful

Part 3

Validating State Machines

Thought

- Testing state machines is hard.
- Can state machines be formalized in some way?
- Can they be proven?
- One approach: Simulation

Representing States

- A "state" refers to a collection of values

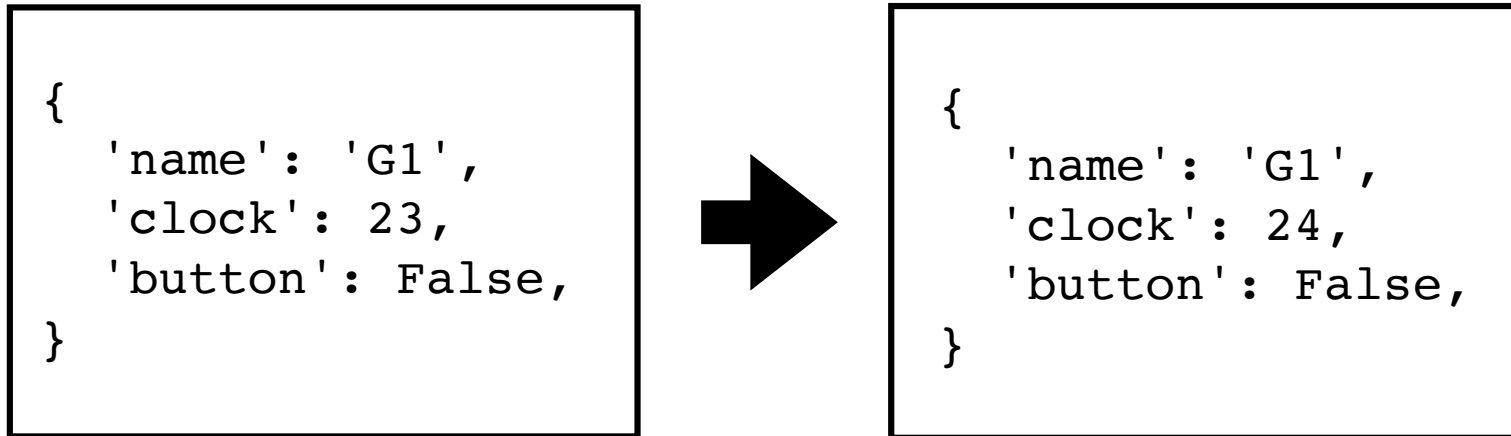
```
{  
  'name': 'G1',  
  'clock': 23,  
  'button': False,  
}
```

```
{  
  'name': 'Y1',  
  'clock': 2,  
  'button': True,  
}
```

- All variables collectively as a whole
- Mental model: Values in a dictionary

Sequencing

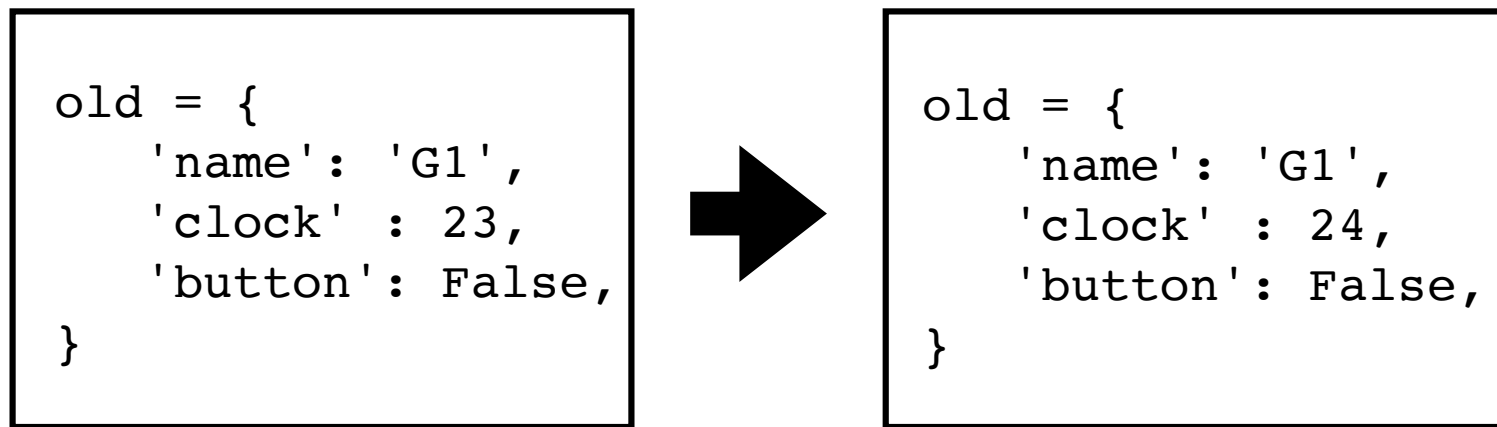
- State machines execute via transitions



- Again, it is driven by "events"

State Changes

- A state change is an update to the values



- It's a function: dict -> dict

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

- Get all old values + updated values

Expressing a State Machine

- Can implement a "next state" function

```
def next_state(state, event):
    if (state['name'] == 'G1'
        and event == 'clock'
        and state['clock'] < 30):
        return dict(state, clock=state['clock'] + 1)
    if (state['name'] == 'G1'
        and event == 'clock'
        and state['clock'] == 30):
        return dict(state, name='Y1', clock=0)
    if (state['name'] == 'Y1'
        and event == 'clock'
        and state['clock'] < 5):
        return dict(state, clock=state['clock'] + 1)
    ...
```

Expressing a State Machine

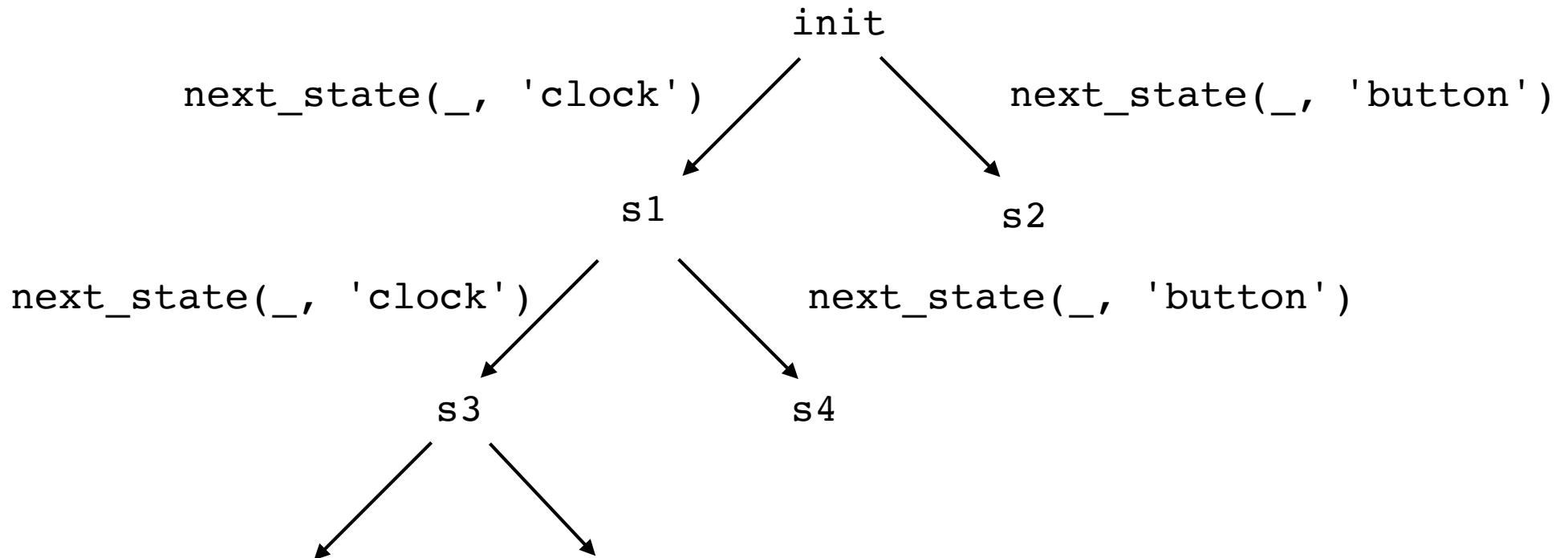
- Example:

```
>>> init = { 'name':'G1', 'clock':0, 'button': False }
>>> next_state(initial, 'clock')
{ 'name': 'G1', 'clock': 1, 'button': False}
>>> next_state(_, clock)
{ 'name': 'G1', 'clock': 2, 'button': False}
>>> next_state(_, clock)
{ 'name': 'G1', '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).
- Also: being able to run a simulation might make testing easier since it's decoupled from the runtime environment (i.e., sockets, threads)

Project 3

- Can you run a state machine simulation of the traffic light?
- A program that explores every possible configuration of the state machine and verifies certain invariants or behaviors

TLA+

- 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
- Slides that follow will be a "brief" intro

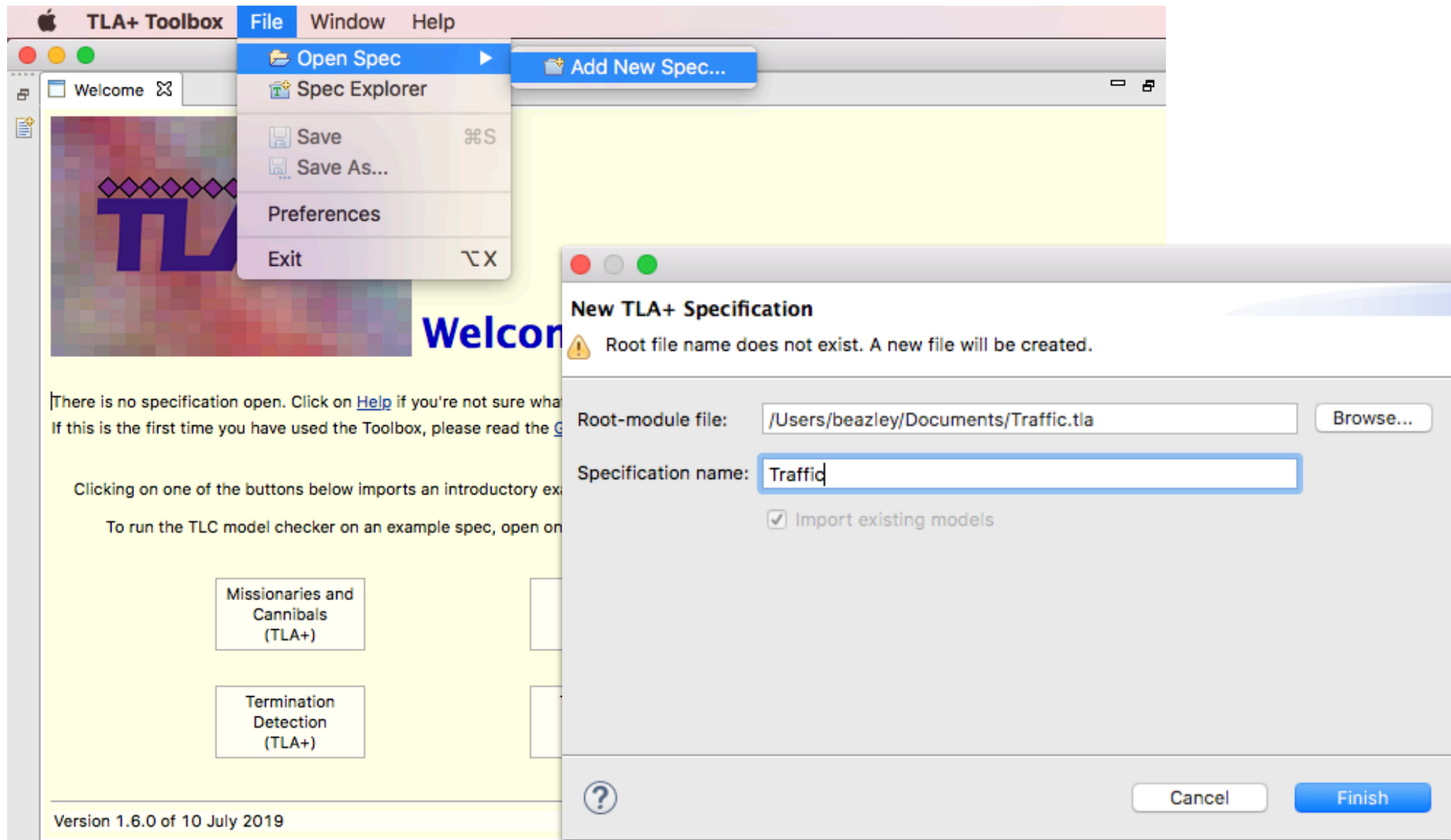
TLA+: Getting Started

- Obtain the toolbox

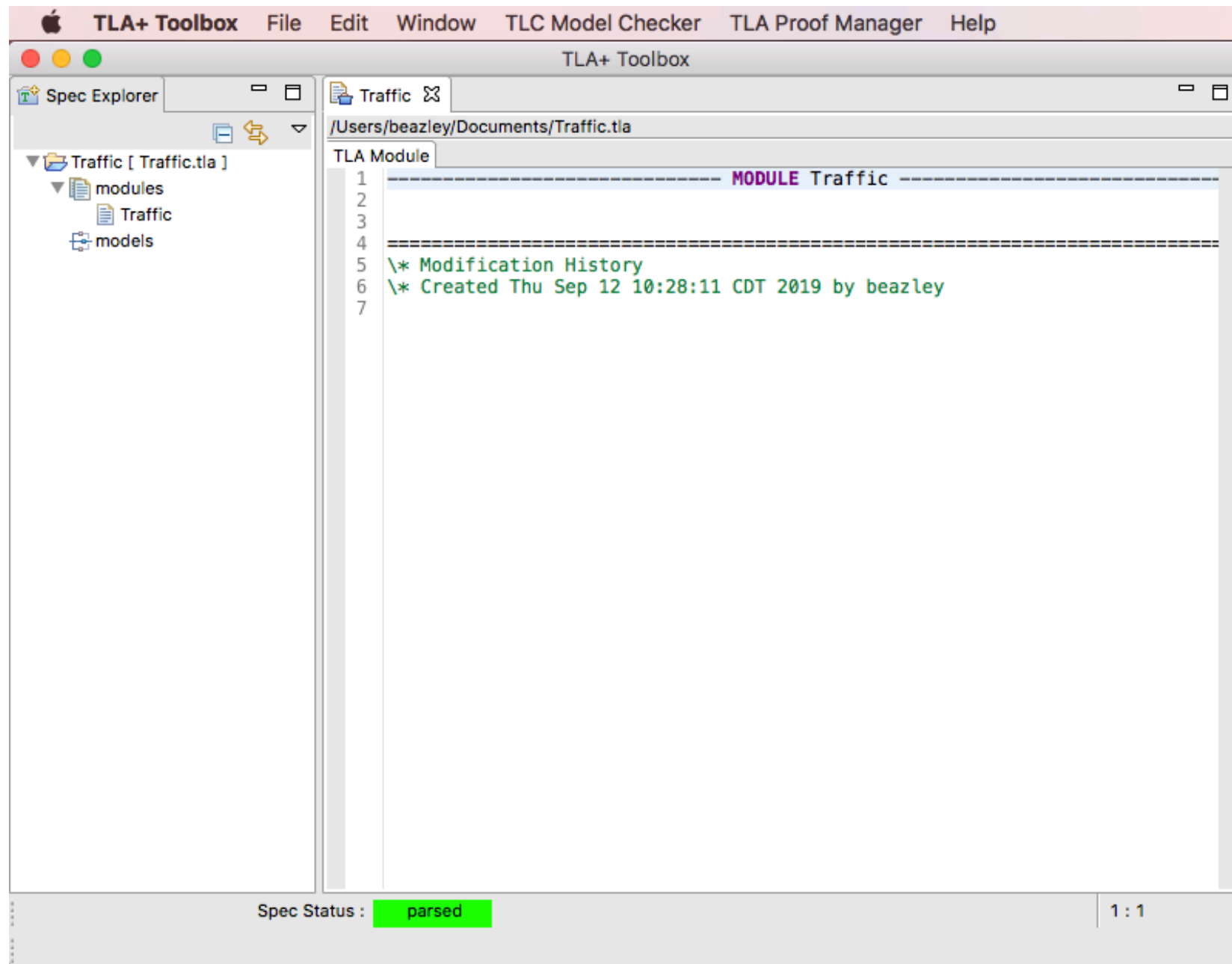
<https://lamport.azurewebsites.net/tla/toolbox.html>

- Minimally requires a Java runtime
- An IDE for writing/checking specifications
- The toolbox is NOT a programming language. It is state machine simulator. It is going to feel very wonky and weird at first.

TLA+: Creating a Spec



TLA+: Starting Point



TLA+ Modules

- TLA+ defines modules

```
----- MODULE Traffic -----  
EXTENDS Integers  
...  
  
=====
```

- EXTENDS is like an import
- In this case, adds support for integer ops

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 name, clock, button
```

- Variables are initialized in a special definition

```
Init == /\ name = "G"  
        /\ clock = 0  
        /\ button = FALSE
```

- This is the "start" state

TLA+ Next State

- Next state is expressed as a math formula

```
Next == \/  
        /\ name = "G1"  
        /\ clock < 30  
        /\ clock' = clock + 1  
        /\ UNCHANGED <<name, button>>
```

```
\/  
        /\ name = "G1"  
        /\ clock = 30  
        /\ clock' = 0  
        /\ name' = "Y1"  
        /\ UNCHANGED <<button>>
```

...

- Compare to the next_state function

TLA+ Next State

- Next state is expressed as a math formula

```
Next == \/  
  /\ name = "G1"  
  /\ clock < 30  
  /\ clock' = clock + 1  
  /\ UNCHANGED <<name, button>>
```

```
\/  
  /\ name = "G1"  
  /\ clock = 30  
  /\ clock' = 0  
  /\ name' = "Y1"  
  /\ UNCHANGED <<button>>
```

...

- First part determines state "membership"

TLA+ Next State

- Next state is expressed as a math formula

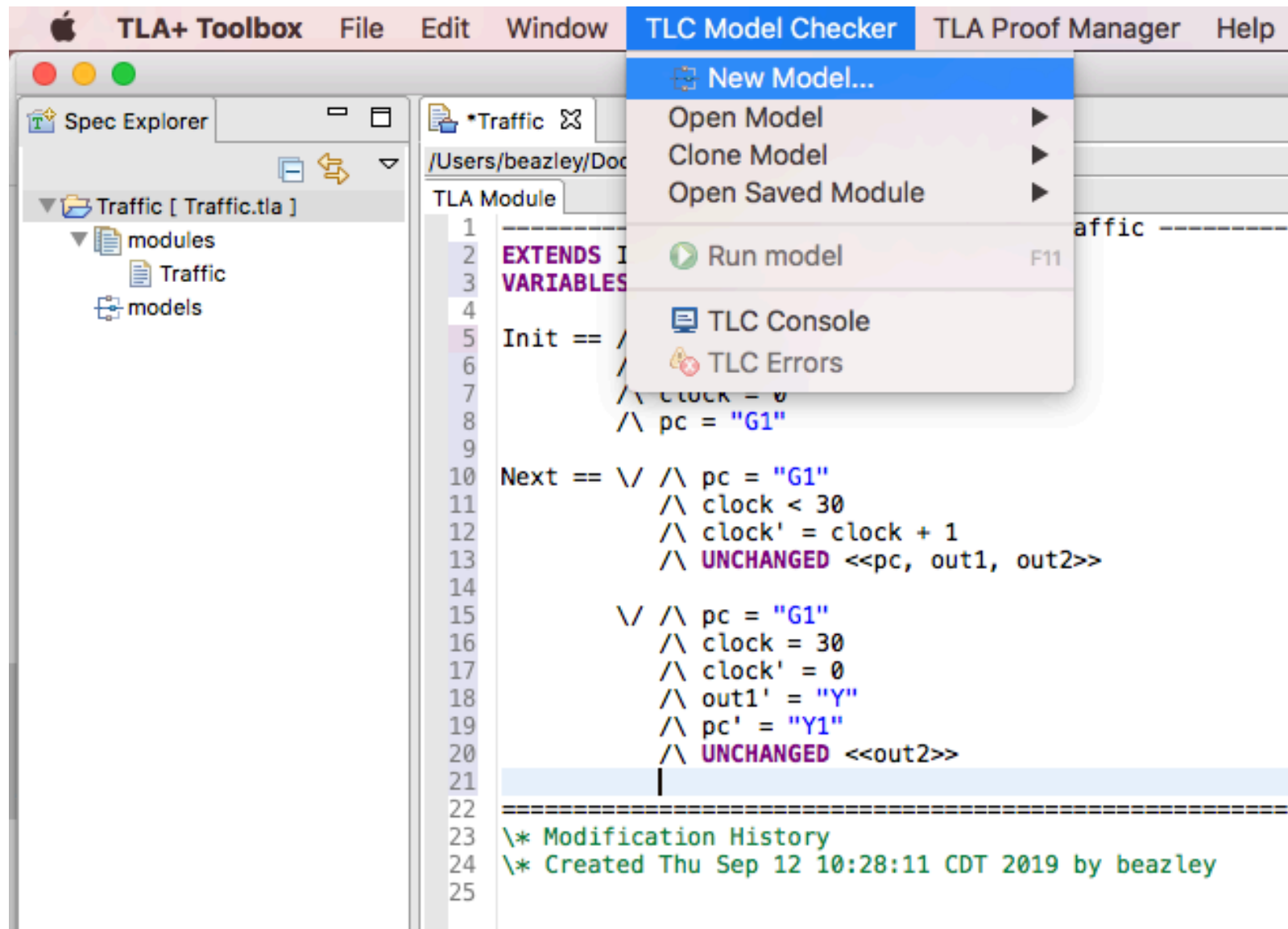
```
Next == \/  
  /\ name = "G1"  
  /\ clock < 30  
  /\ clock' = clock + 1  
  /\ UNCHANGED <<name, button>>
```

```
\/  
  /\ name = "G1"  
  /\ clock = 30  
  /\ clock' = 0  
  /\ name' = "Y1"  
  /\ UNCHANGED <<out2>>
```

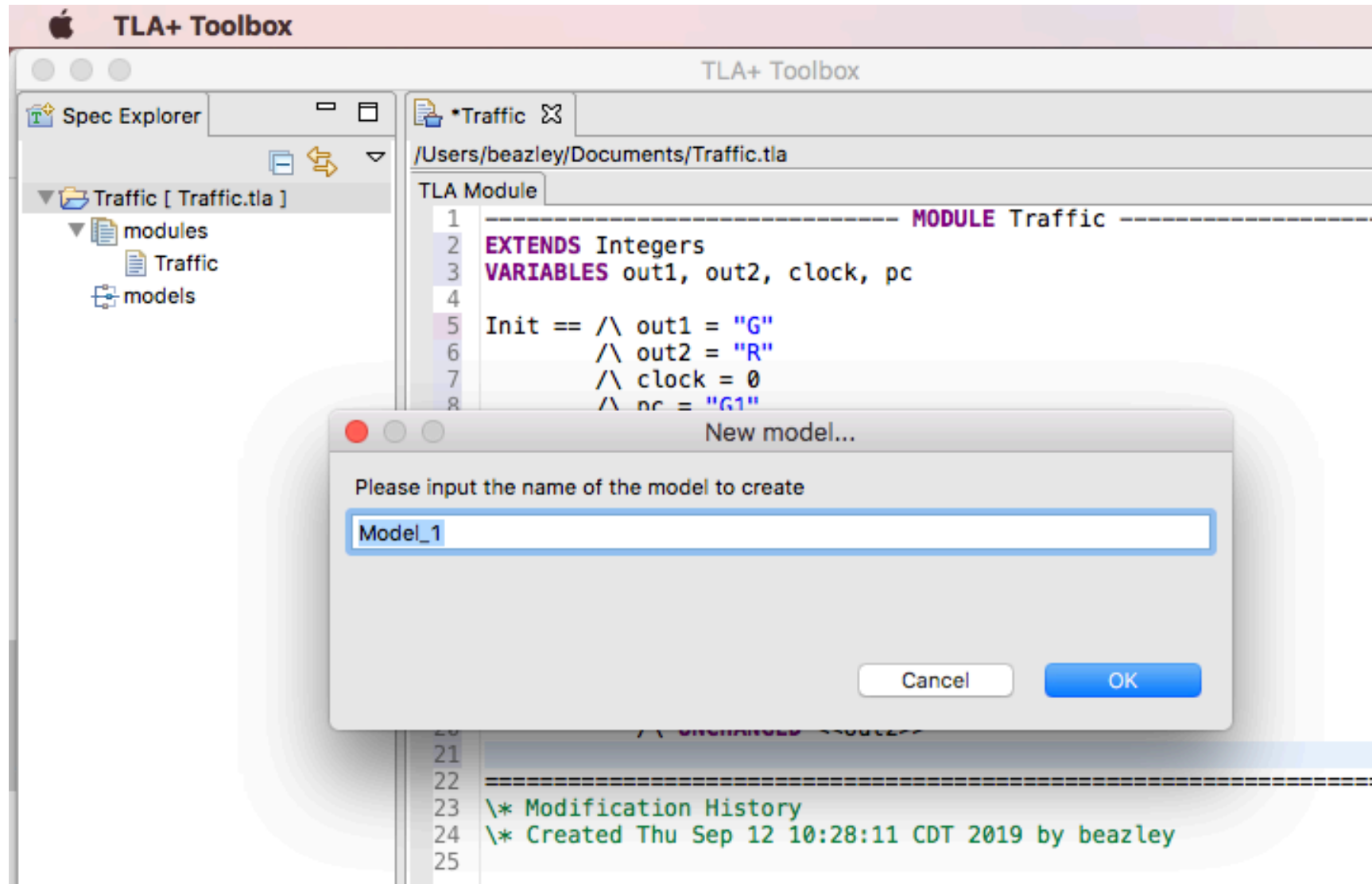
...

- State changes written: $var' = expression$

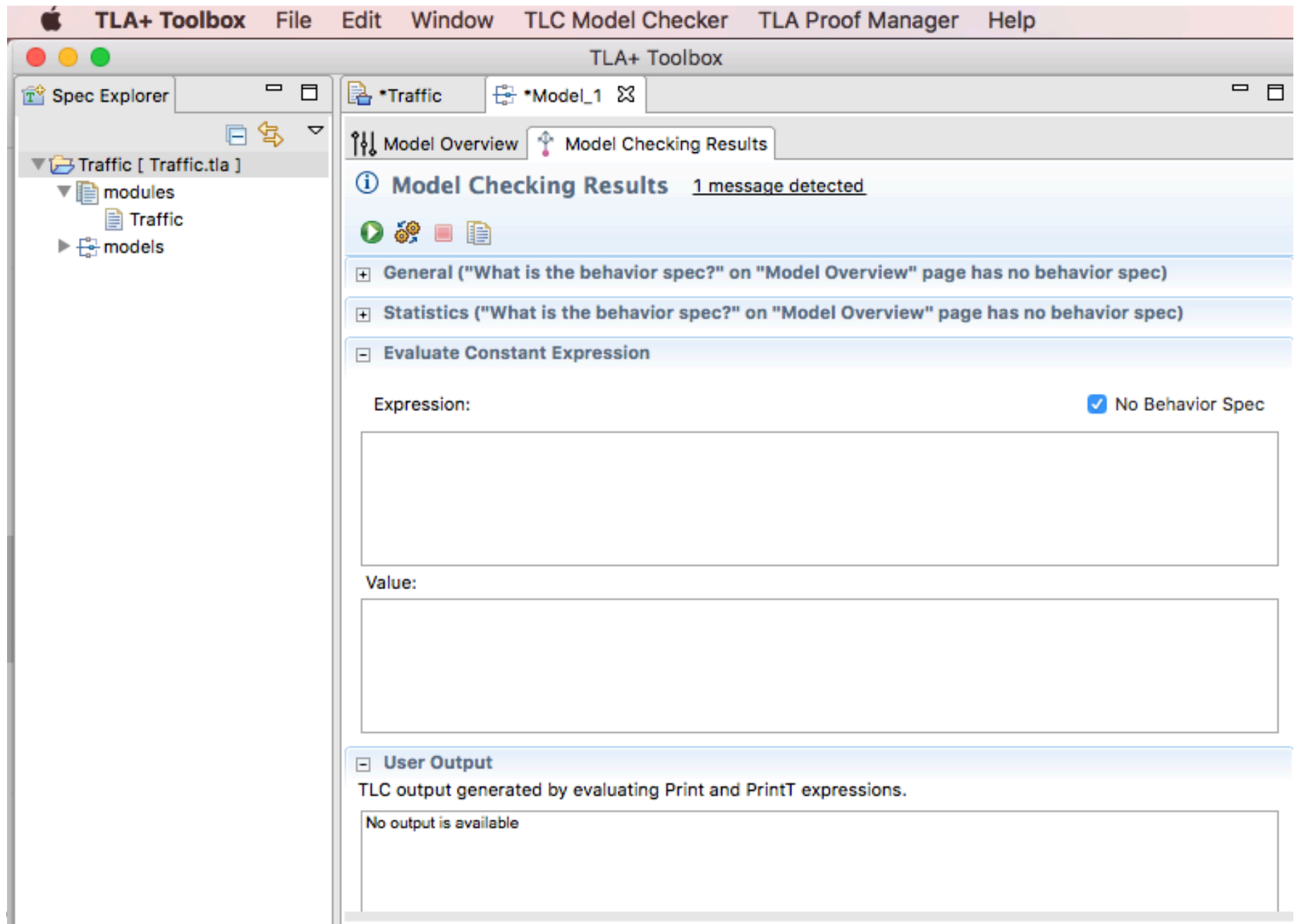
Model Checking



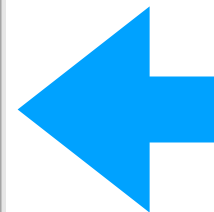
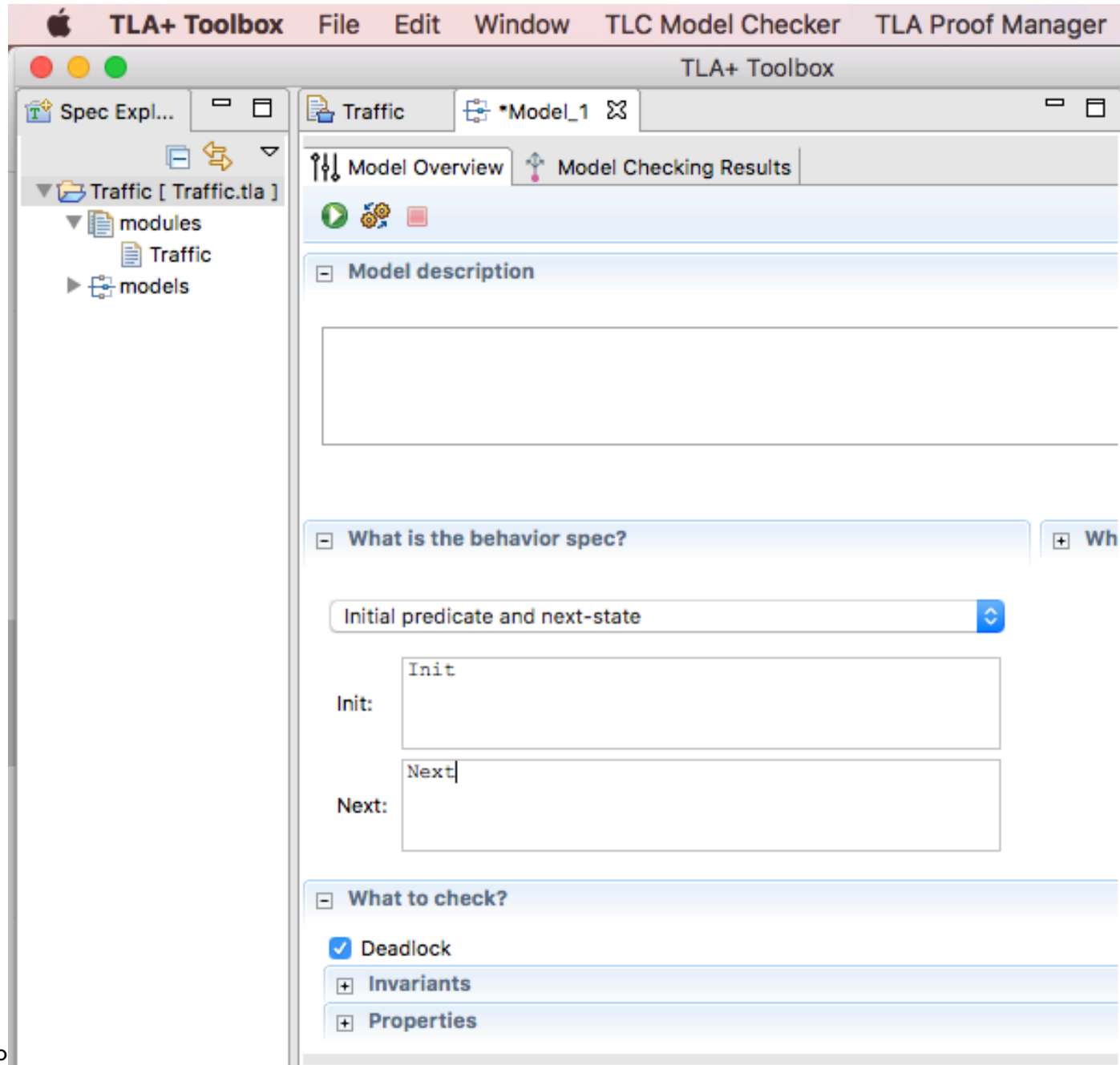
Model Checking



Model Checking

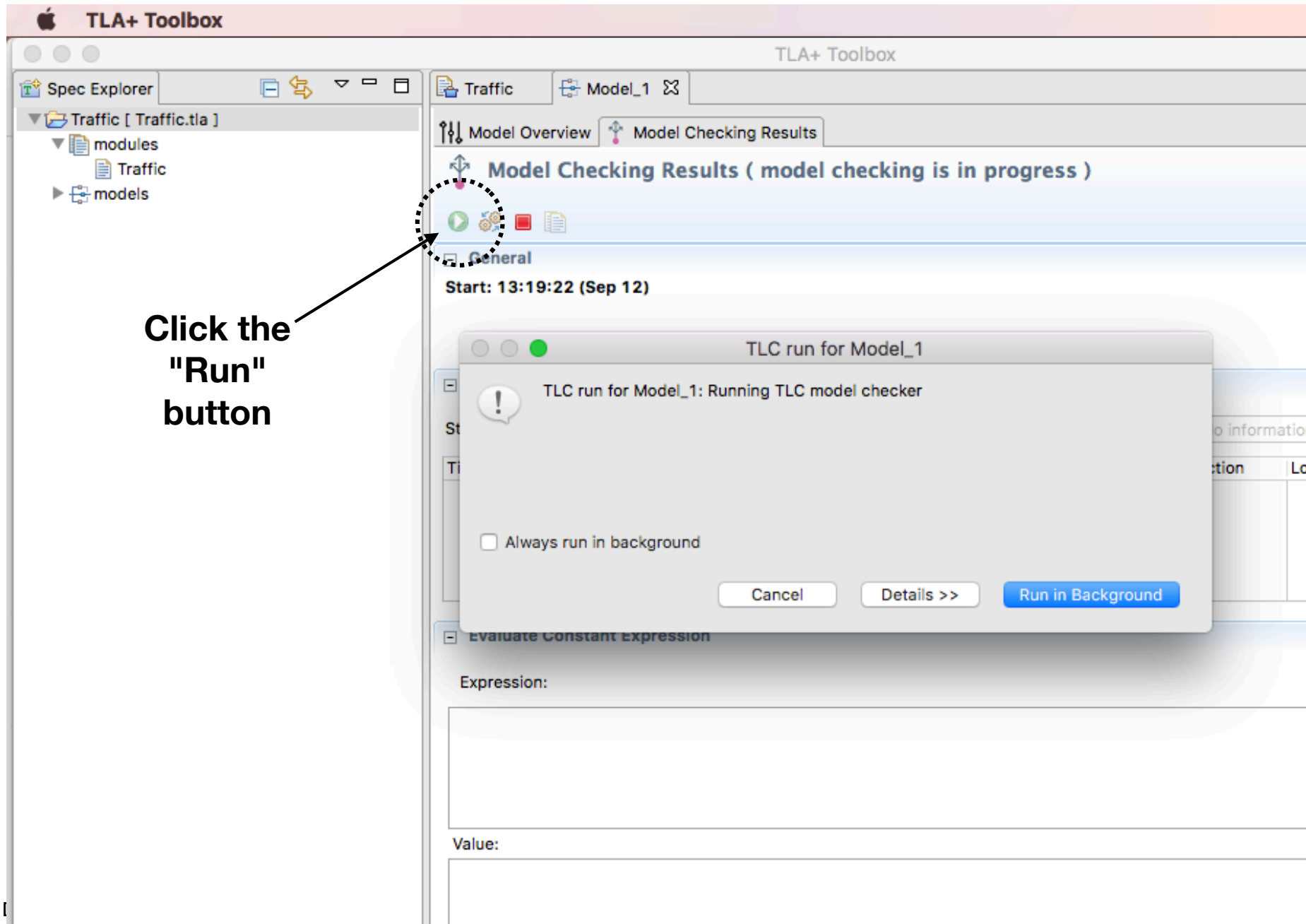


Model Checking



**Set the Init and
Next state
definitions**

Model Checking



Model Checking (Results)

TLA+ Toolbox

Traffic Model_1

Model Overview Model Checking Results

Model Checking Results

General

Start: 12:46:18 (Sep 12) End: 12:46:19 (Sep 12) Not running

1 Error

Statistics

State space progress (click column header for graph) Actions at 00:00:01

Time	Diameter	States Found	Distinct States	Queue Size	Module	Action	Location	States Found	Distinct
00:00:01	32	32	32	0	Traffic	Next	line 15, col...	2	1
00:00:01	0	1	1	1	Traffic	Init	line 5, col 1...	2	2
					Traffic	Next	line 10, col...	60	30

Evaluate Constant Expression

Expression: ☐ No Behavior Spec

Value:

TLC Errors

Model_1

Deadlock reached.

Error-Trace Exploration

Expressions to be evaluated at each state of the trace - drag to re-order.

Add Edit Remove

Error-Trace

Name	Value
pc	"G1"
<Next line...	State (num = 32)
clock	0
out1	
out2	
pc	"Y1"

Click on a row to see in viewer by Double-click to go to corresponding down % to go to the original P

```
/\ clock = 0
/\ out1 = "G"
/\ out2 = "R"
/\ pc = "G1"
```

Sets

- TLA has sets and set operations

```
values =={ 1, 4, 2}
```

```
x \in values          \* Membership test
\A x \in values: x > 0 \*  $\forall x \in \text{values}: x > 0$ 
\E x \in values: x > 3 \*  $\exists x \in \text{values}: x > 3$ 
```

```
CHOOSE x \in values: \A y \in values: x <= y
```

- Range, map, and filter

```
nums == 1..n          \* Range. { 1, 2, 3, ... n }
```

```
{ 10*x : x \in nums } \* { 10, 20, 30, ... }
{ x \in nums: x > 3 } \* { 4, 5, ... n }
```

Working with Multiples

- A specification might express the idea of working with multiple things (e.g., servers)

```
\* Dining Philosophers  
VARIABLES sticks, pc
```

```
Phils == 1..5
```

```
Init == /\ sticks = [ n \in Phils |-> 0 ]  
        /\ pc = [ n \in Phils |-> "hungry" ]
```

```
/* sticks = <<0, 0, 0, 0, 0>>
```

```
/* pc = <<"hungry", "hungry", "hungry", "hungry", "hungry">>
```

- Similar to a list comprehension

Working with Multiples

- State definitions can be parameterized

```
LeftStick(i) == (i % 5) + 1
RightStick(i) == ((i + 1) % 5) + 1
```

```
Hungry(i) == /\ pc[i] = "hungry"
              /\ sticks[LeftStick(i)] = 0
              /\ pc' = [pc EXCEPT ![i] = "grab1"]
              /\ sticks' = [sticks EXCEPT ![LeftStick(i)] = i]
```

```
Grab1(i) == /\ pc[i] = "grab1"
              /\ sticks[RightStick(i)] = 0
              /\ pc' = [pc EXCEPT ![i] = "eat"]
              /\ sticks' = [sticks EXCEPT ![RightStick(i)] = i]
```

```
Eat(i) == ...
```

```
Philosopher(i) == Hungry(i) \/ Grab1(i) \/ Eat(i)
```

Working with Multiples

- Simulating Concurrent Operation

```
Phils == 1..5
```

```
Init == /\ sticks = [n \in Phils |-> 0]  
        /\ pc = [n \in Phils |-> "hungry" ]
```

```
...  
Philosopher(i) == Hungry(i) \/ Grab1(i) \/ Eat(i)
```

```
Next == \E i \in Phils: Philosopher(i)
```

- It looks wild, but it's saying that the next state is defined by any philosopher that can do something.

Group Exercise

- Write TLA+ spec for Dining Philosophers
- See that it detects deadlock
- Fix to avoid deadlock

Big Picture

- TLA+ is NOT an implementation language
- There is no "runtime" in which you make a working state machine or process events
- The events are implicit in the model
- The next state relation lists possibilities

$\text{Next} == A \ \backslash / \ B \ \backslash / \ C \ \backslash / \ D$

- TLA+ explores all possible branches

Exercise

- Look at Raft state machine description in paper
- Take a look at formal Raft TLA+ spec

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

- Can you make any sense of it?

Part 4

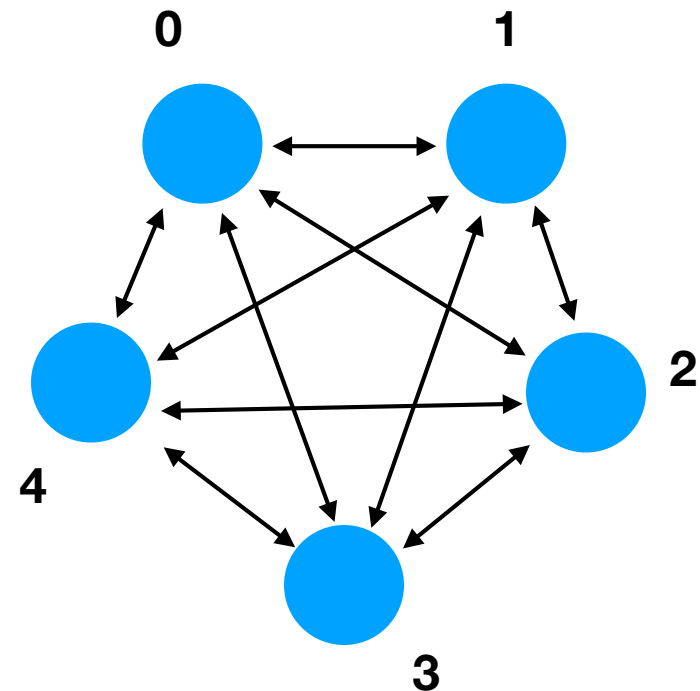
The Raft Network

Raft Networking

- It is a cluster of interconnected servers
- Addressing (need config)

```
# raftconfig.py
```

```
server_nodes = {  
    0: ('localhost', 15000),  
    1: ('localhost', 15001),  
    2: ('localhost', 15002),  
    3: ('localhost', 15003),  
    4: ('localhost', 15004)  
}
```



- Number each server (0-4). Map server numbers to actual network addresses (configuration).

Raft Networking

- Think about a "high level" interface

```
# Create the Raft network
net = RaftNetwork(0)    # Number is my identity

# Send a message to other servers
net.send(1, b"hello from 0")
net.send(2, b"hello from 0")

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

- Have a mechanism where you can send a message to any server using its server number
- Receive a message from anyone

Raft Networking

- Important assumptions about 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 everywhere
- Think about testing

Project 4

- Work on message passing constructs for Raft
- Think about data structures for representing messages and the encoding
- Think about testing/debugging mechanisms

Part 5

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 object
- 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: It only needs to record state changes

Example: Shared Lock

Operations

```
lock.acquire()  
lock.release()  
lock.acquire()  
lock.release()  
...
```

Transaction Log

```
'acquire'  
'release'  
'acquire'  
'release'
```

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: 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
```

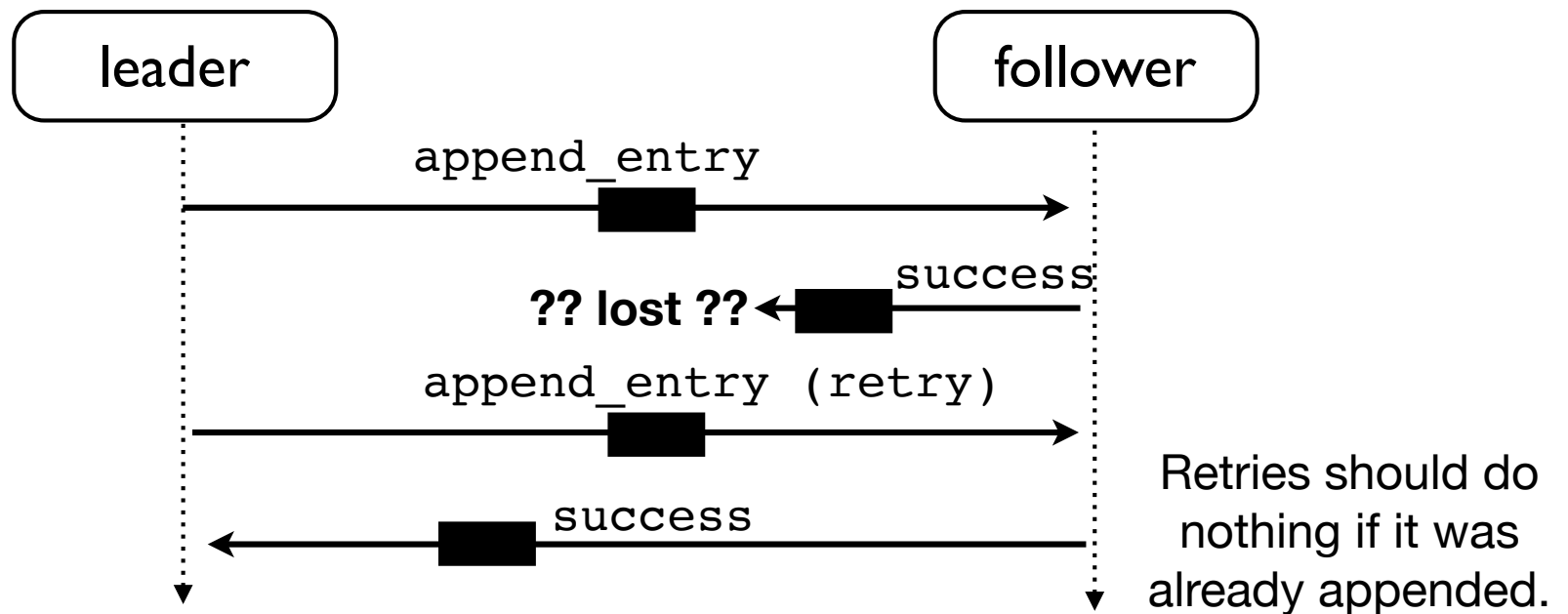
- The issue: What if a server dies due to catastrophic system failure (power-loss, hardware fault, kernel crash, etc..)
- This is more of a deployment issue. How robust does it have to be?

Problem: Idempotence

- Appends must be "idempotent"

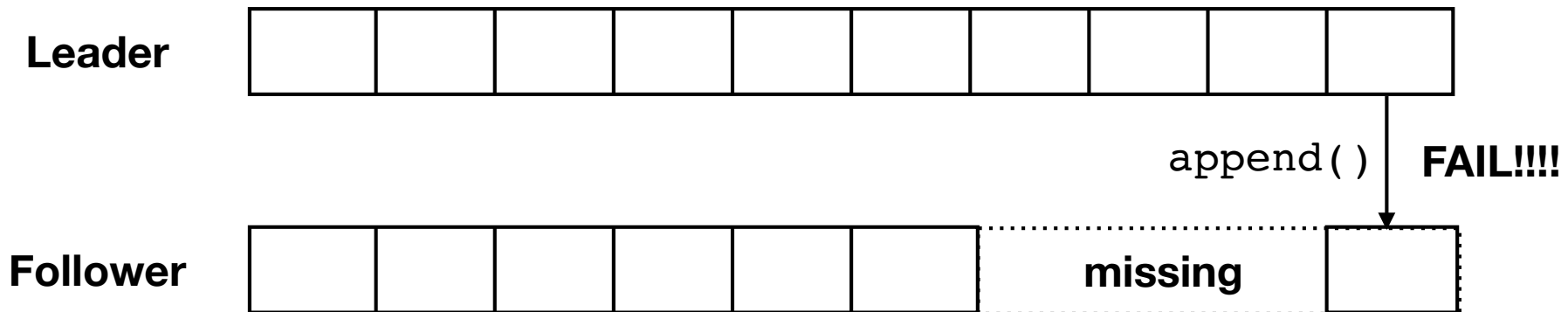
```
append_entry(log, entry)
append_entry(log, entry) # Does nothing (already appended)
```

- The issue: Network faults/retries



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.

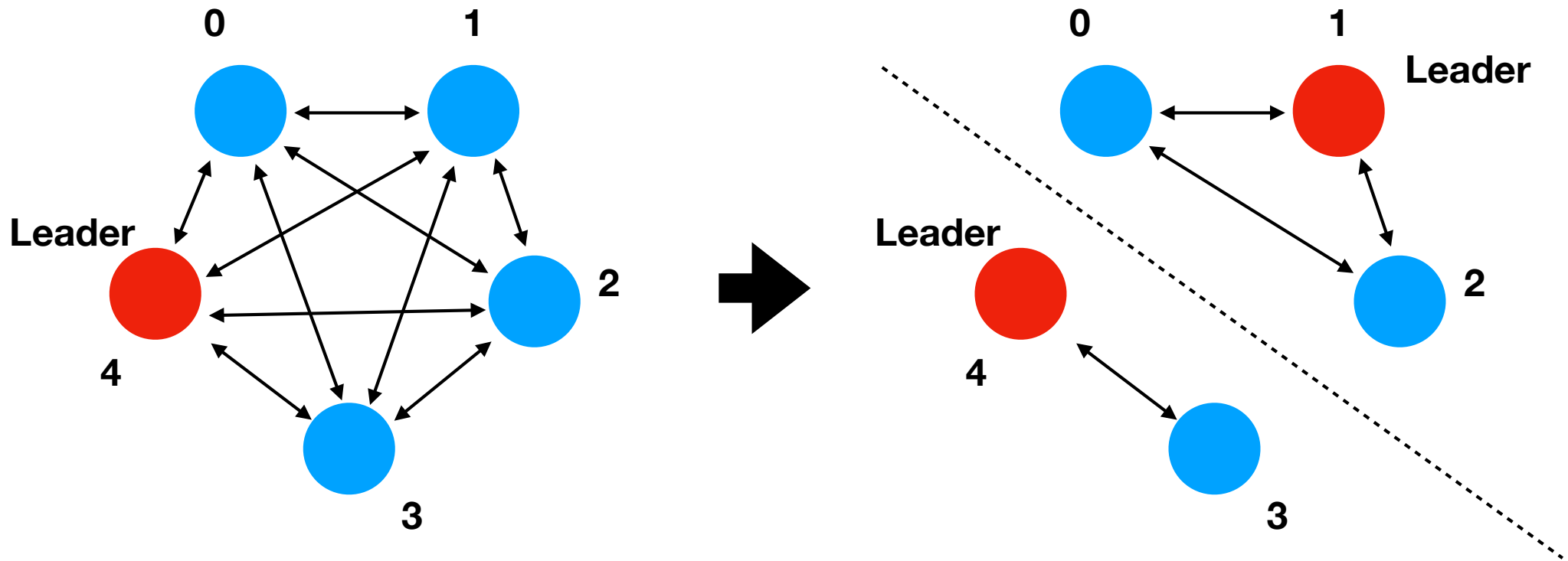
Log Matching Property I

- If entries in different logs have the same index, they must be the exact same entry

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

- Critically important: Logs in Raft are supposed to be identical across all servers. The same entries are in the same log position. Always.
- The algorithm works to enforce this.

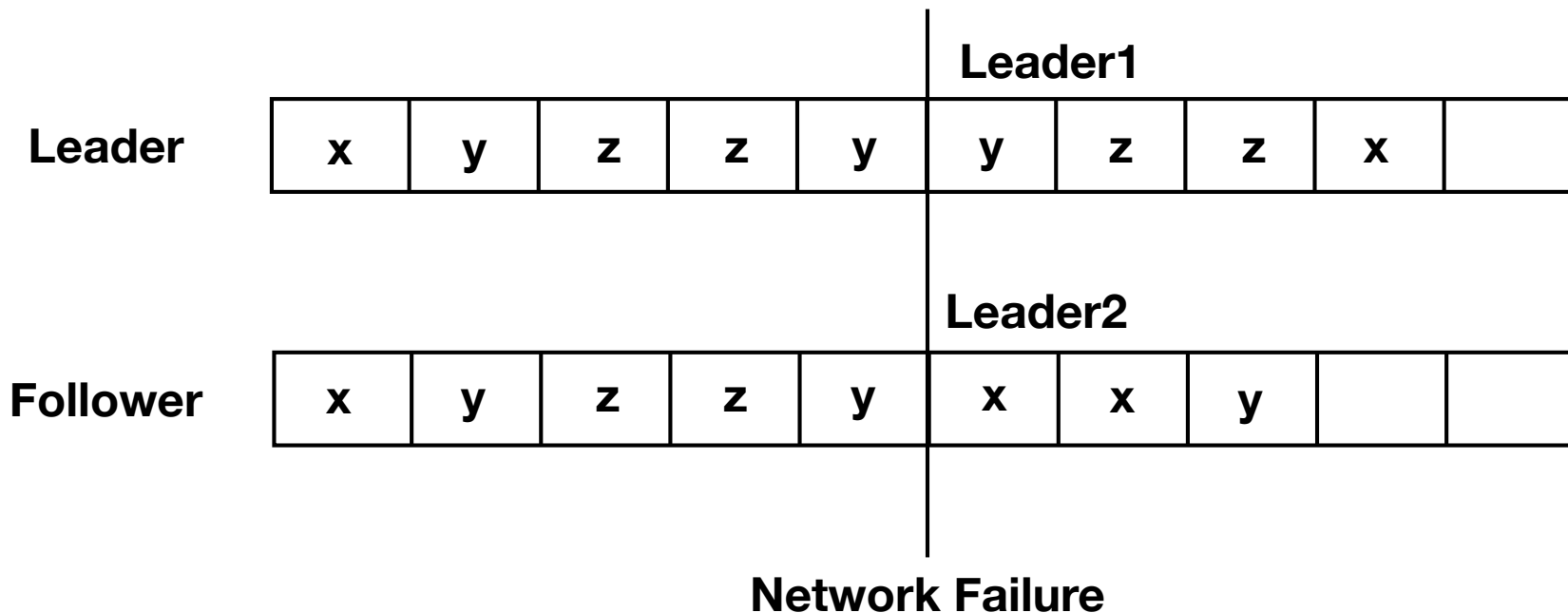
Problem: Partitions



- Could experience a network partition that results in two "leaders."
- Former leader doesn't know it's cut off.

Problem: Partitions

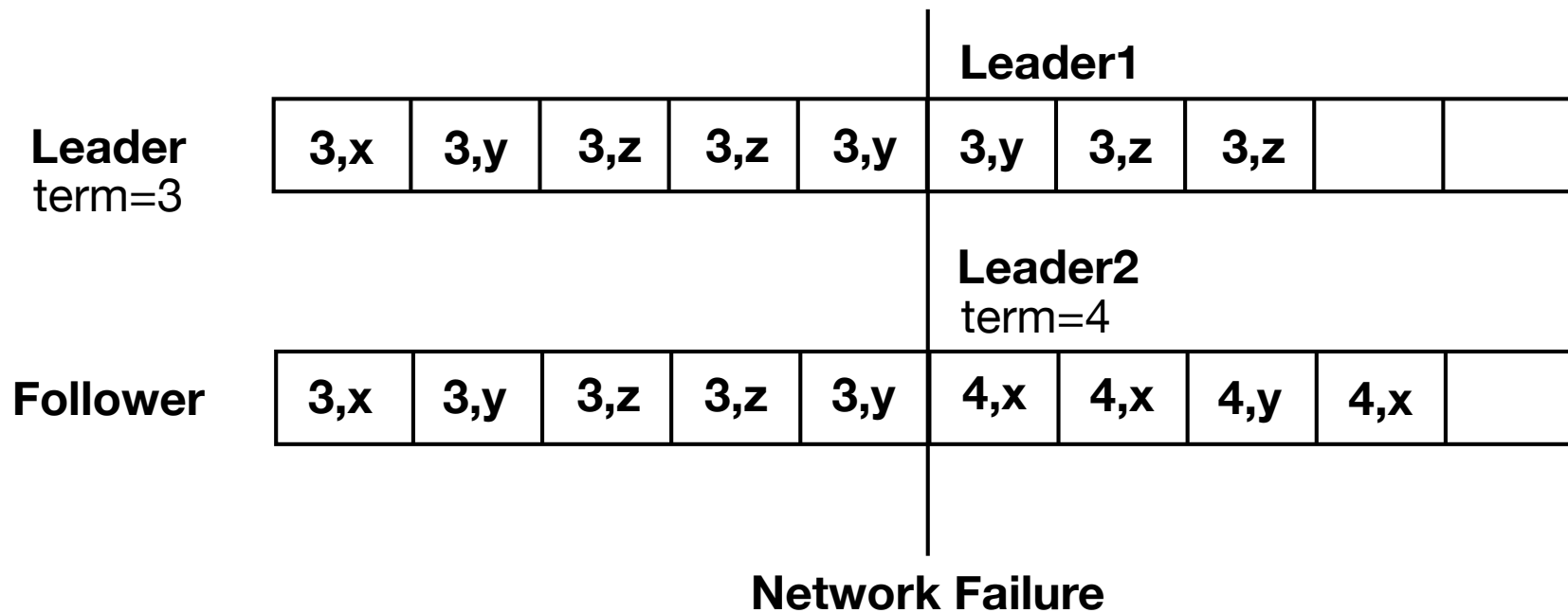
- Having two "leaders" causes a log divergence



- 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
- Incremented on every leader change
- The term is also recorded in the log



Log Matching Property 2

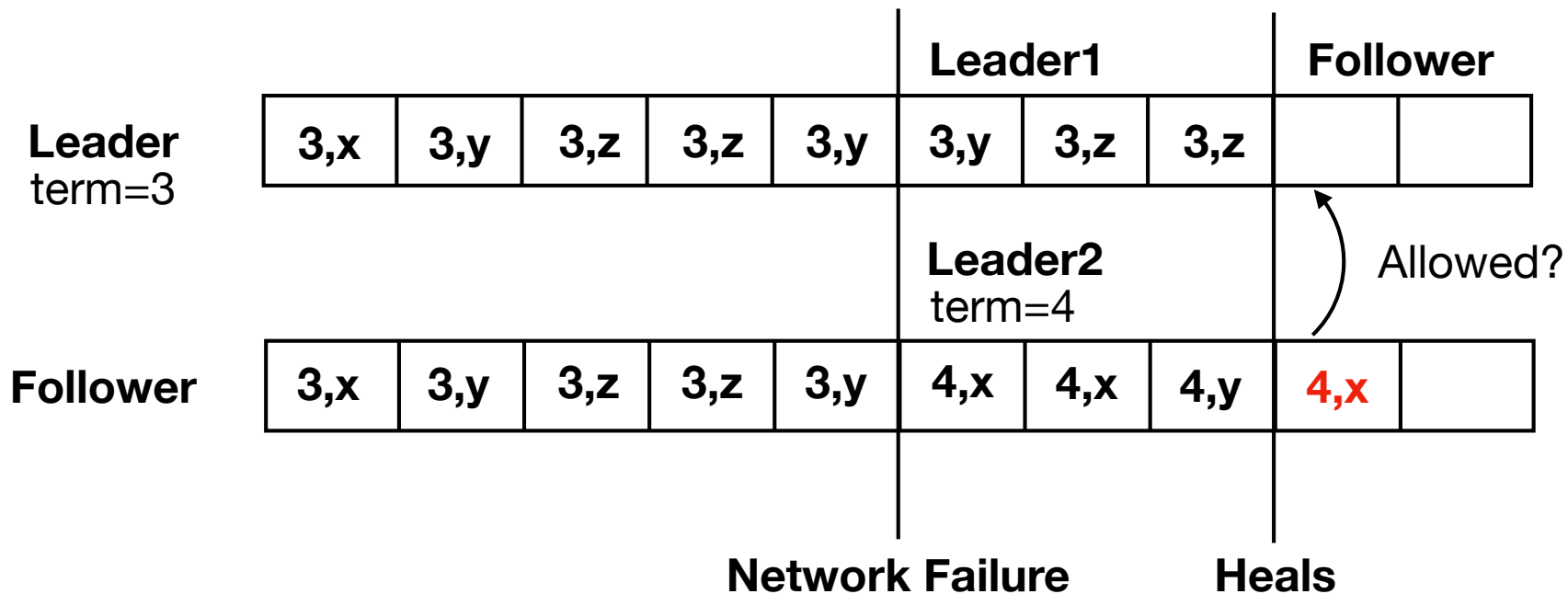
- If entries in different logs have the same index and term, then the logs must be identical in all preceding entries

```
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

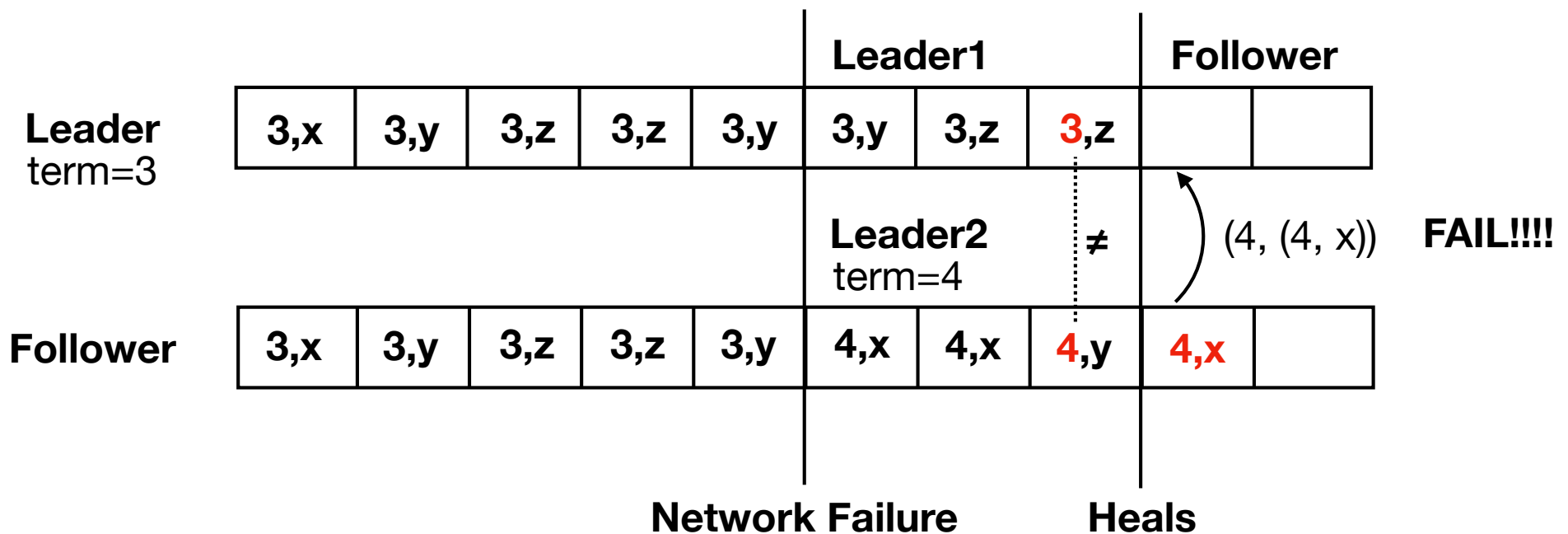
- The log is never allowed to have gaps, but consider this tricky scenario after healing



- Can the new leader just append onto old leader?

Log Matching

- Appends additionally require the term number of the preceding log entry



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

Log Implementation

- Log Appends are encapsulated into a single operation called "AppendEntries"
- It is basically a list append with conditions
 - Persistent storage (non-volatile)
 - Idempotent (repeated attempts ok)
 - No gaps/holes
 - Must enforce the log matching properties

Project 5

- 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.

Part 6

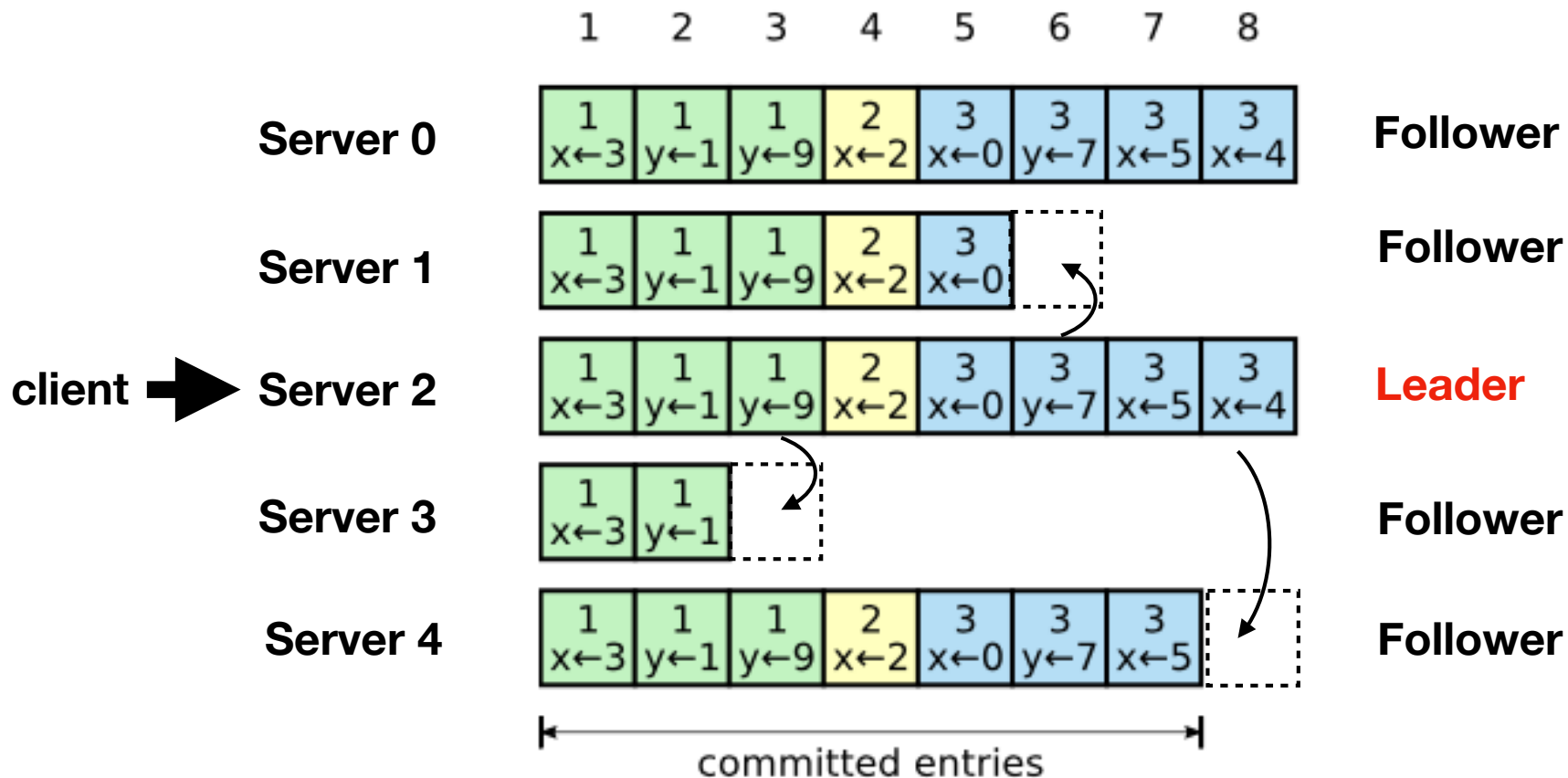
Log Replication

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
- And also to establish consensus

Followers Update

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



- It does this by sending messages (AppendEntries)

Comments

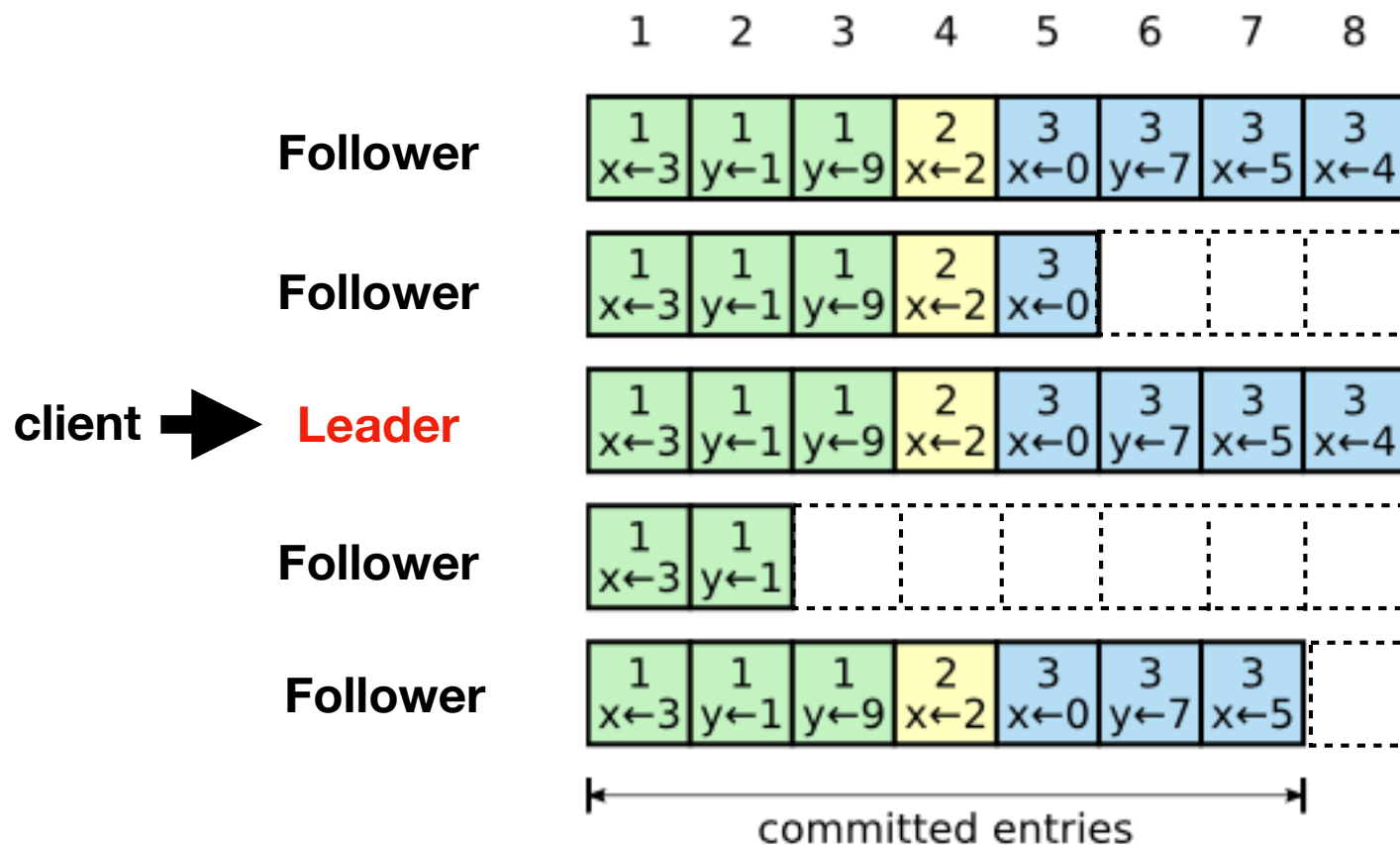
- Followers are passive.
- They receive messages from the leader and respond to the leader. NEVER with each other.
- If there is no leader, a new leader is elected from one of the followers (later).

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

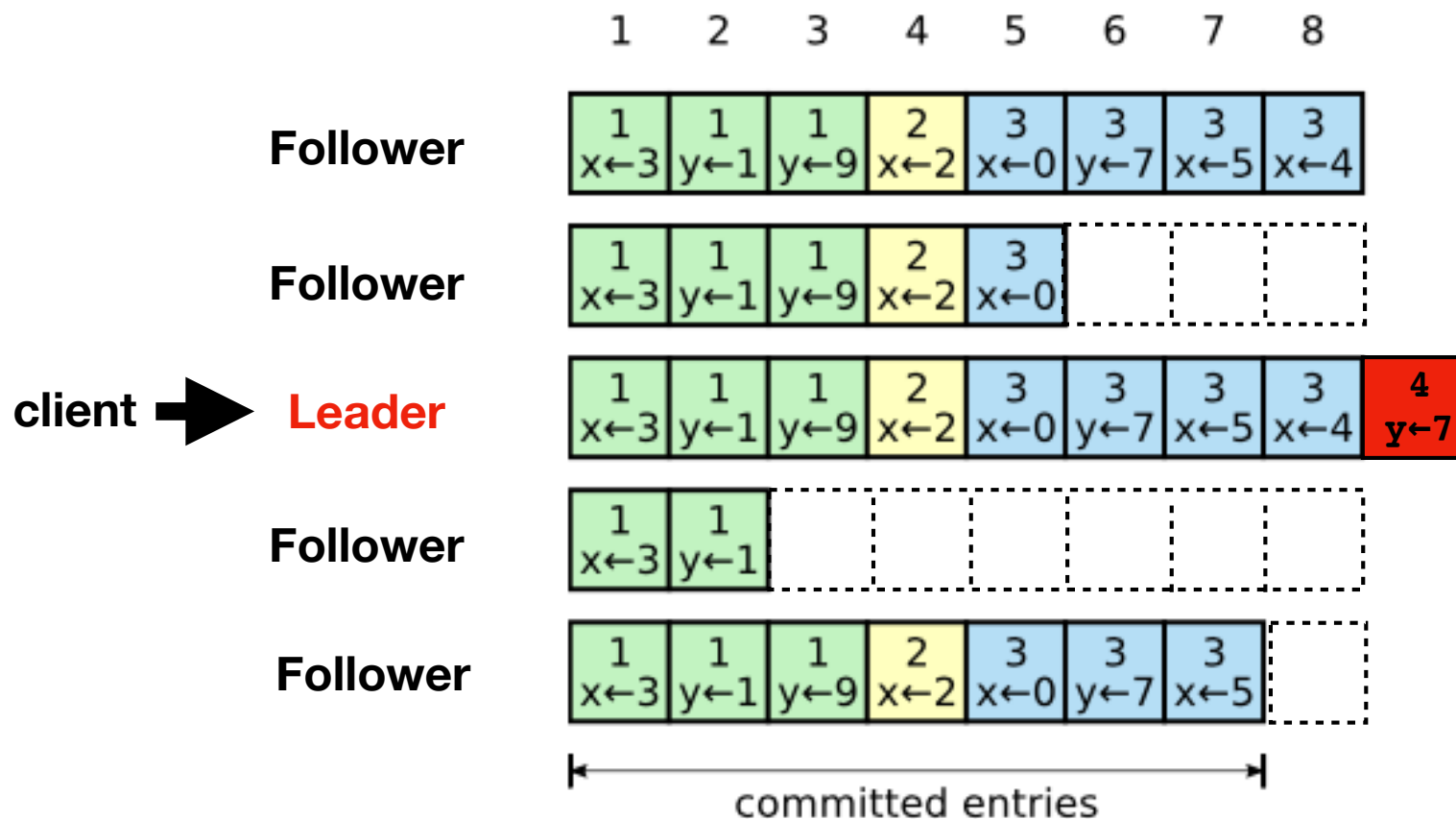
- Leaders start by assuming that all followers have the entire log (same information as leader)



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

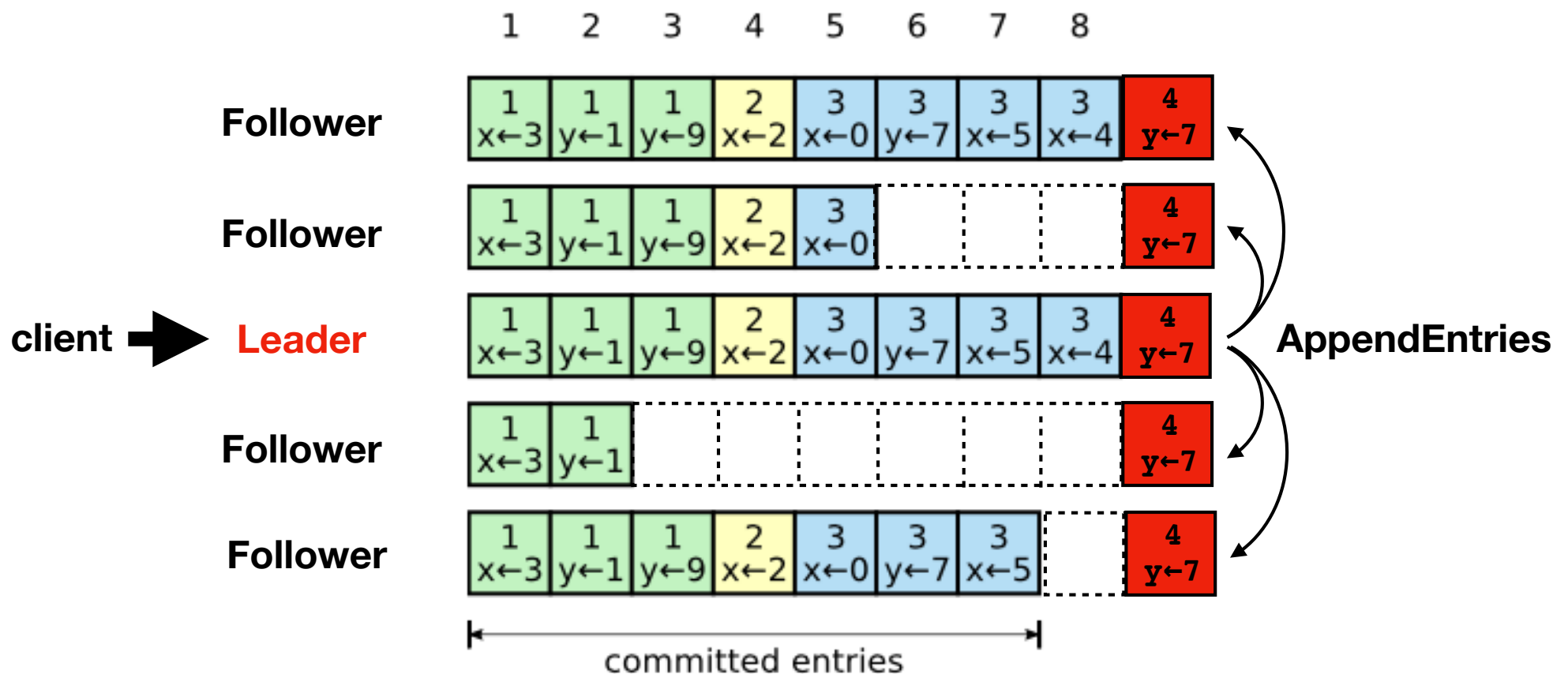
New Appends

- New entries go on end of leader log



Append Replication

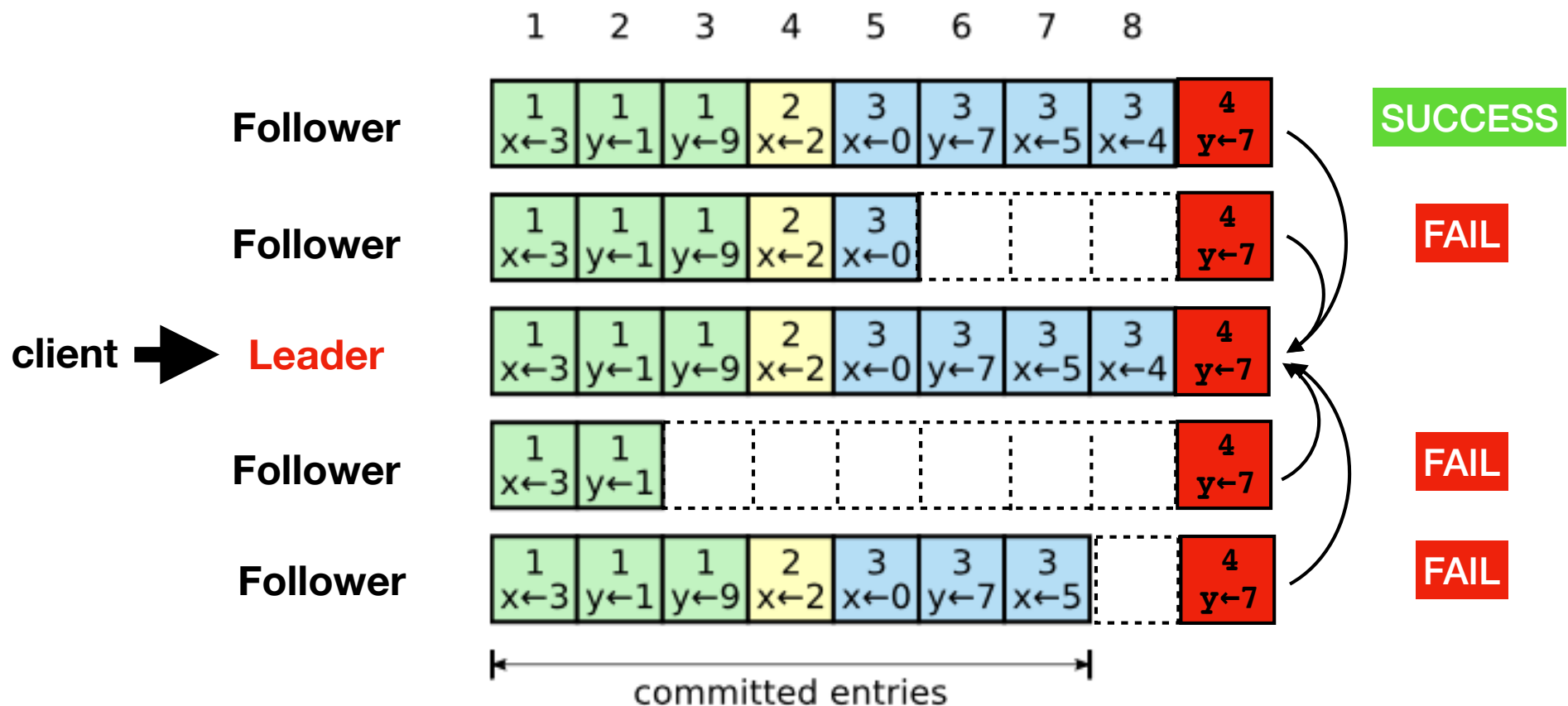
- Leader replicates to the followers



- This is a network message

Response Handling

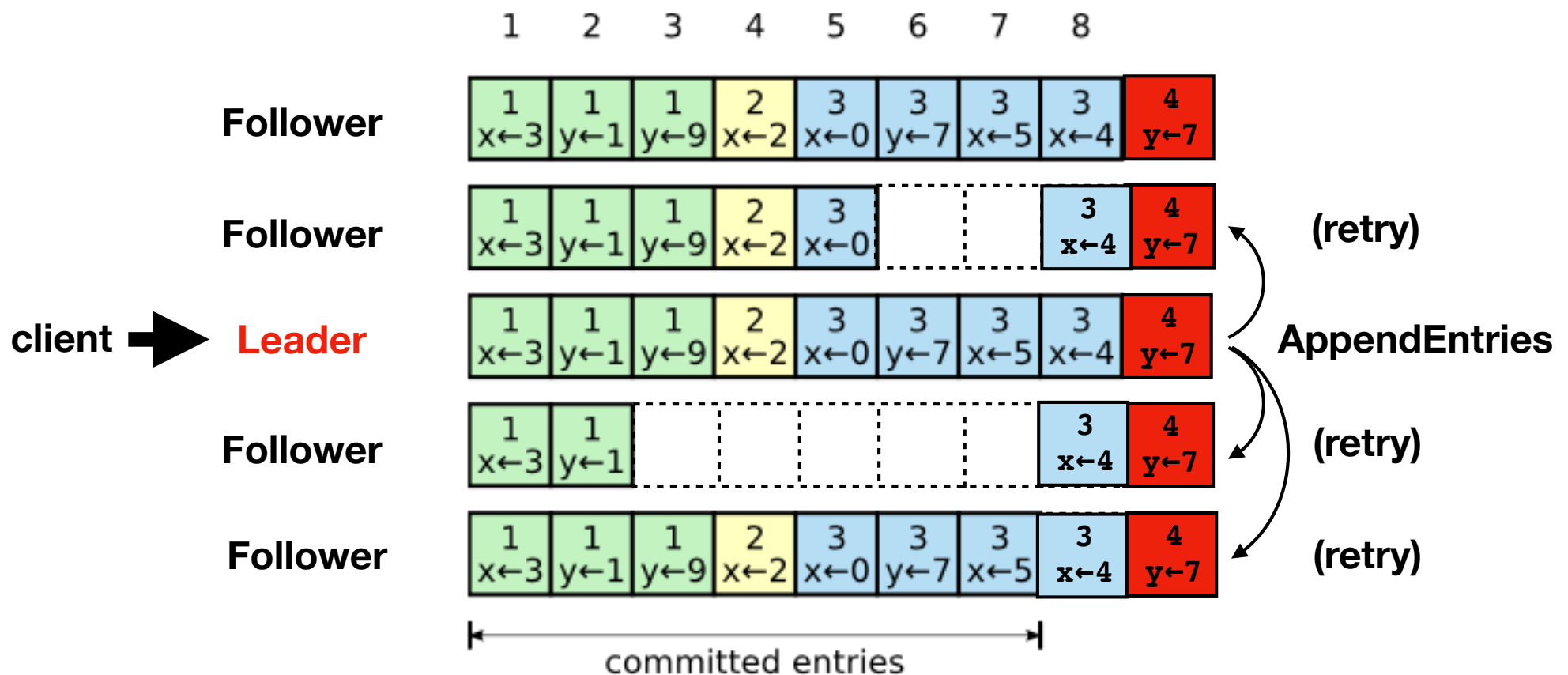
- AppendEntries will succeed or fail (no gaps)



- This is reported back (`AppendEntriesResponse`)

Backtracking

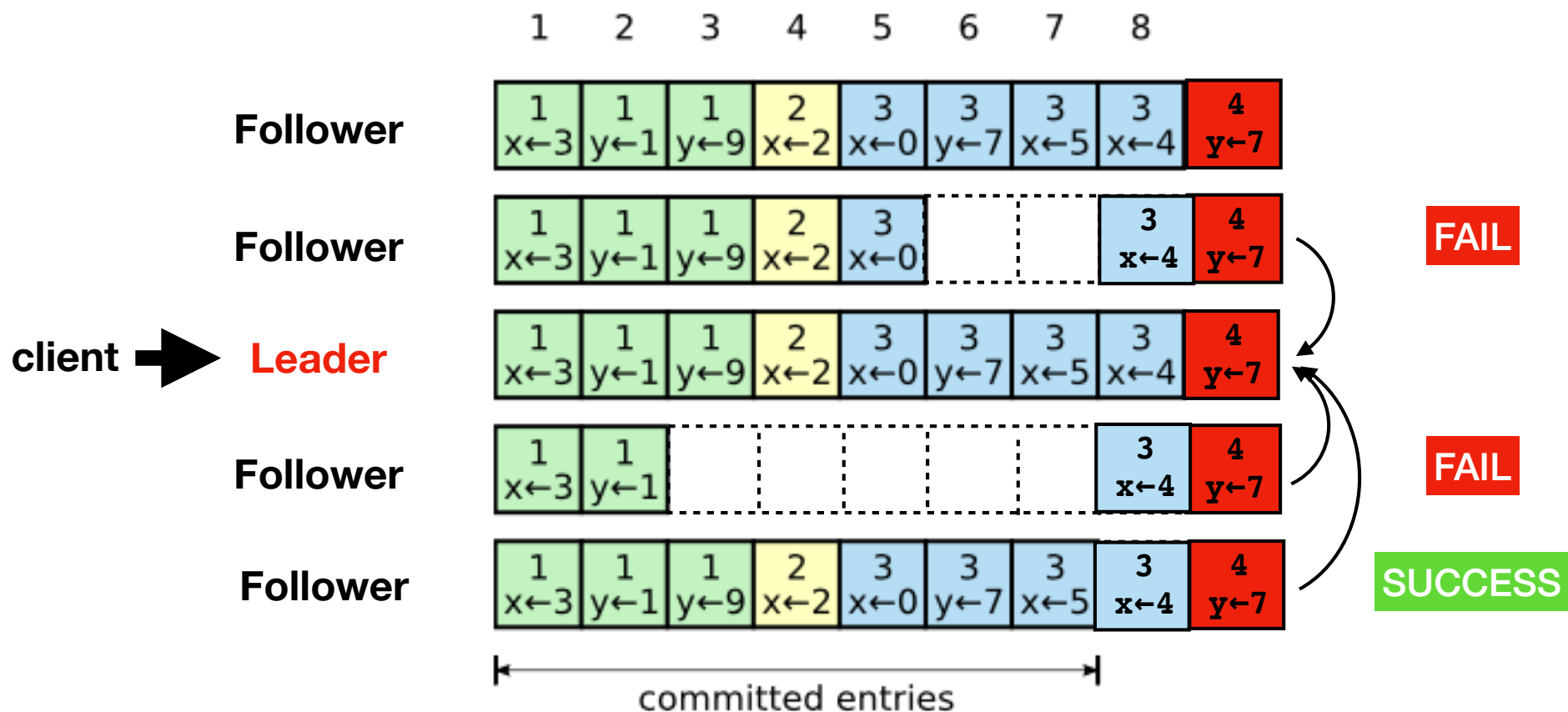
- Leader retries failures with earlier log entries



- Again, followers report back with success

Backtracking

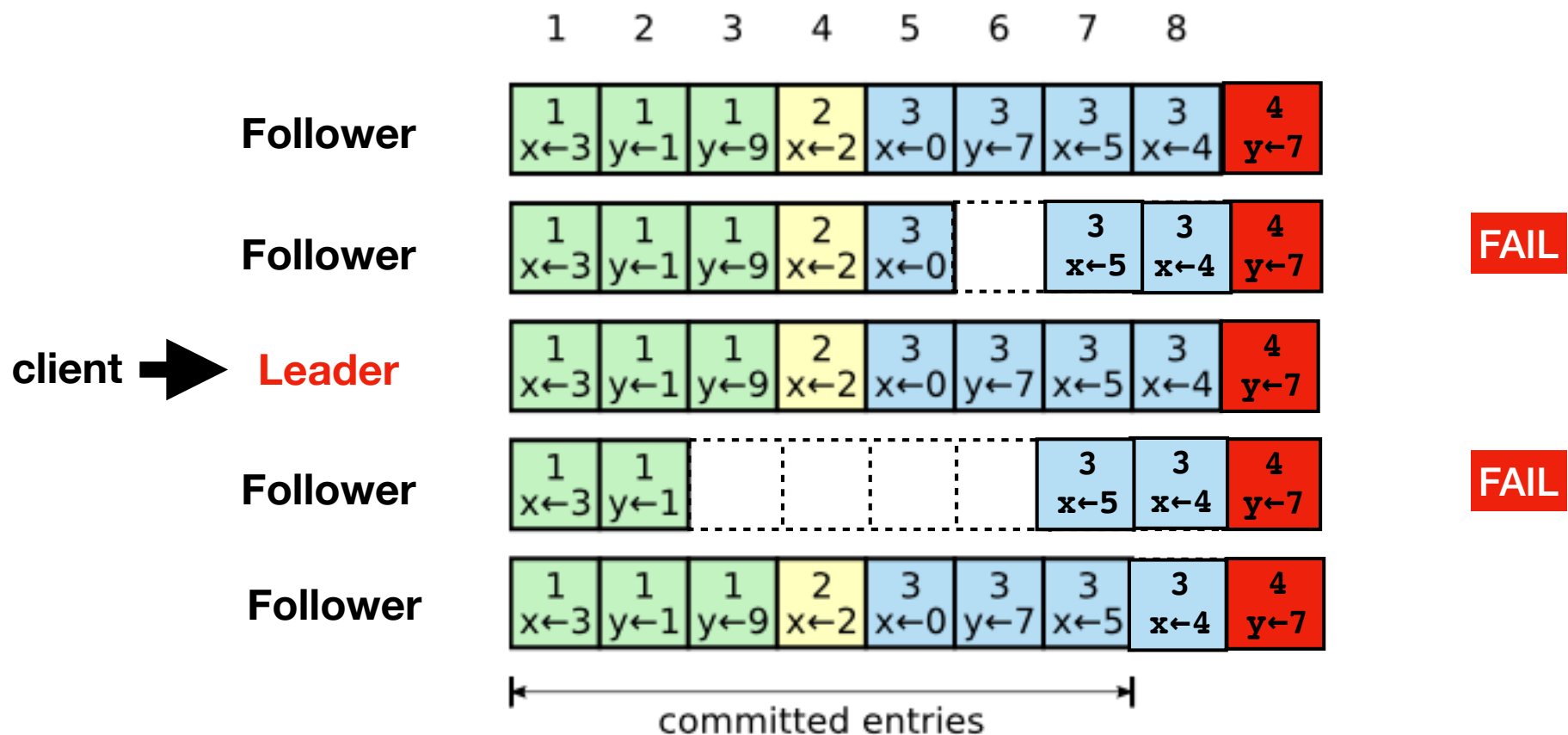
- This process repeats itself



- Leader keeps working backwards until success

Backtracking

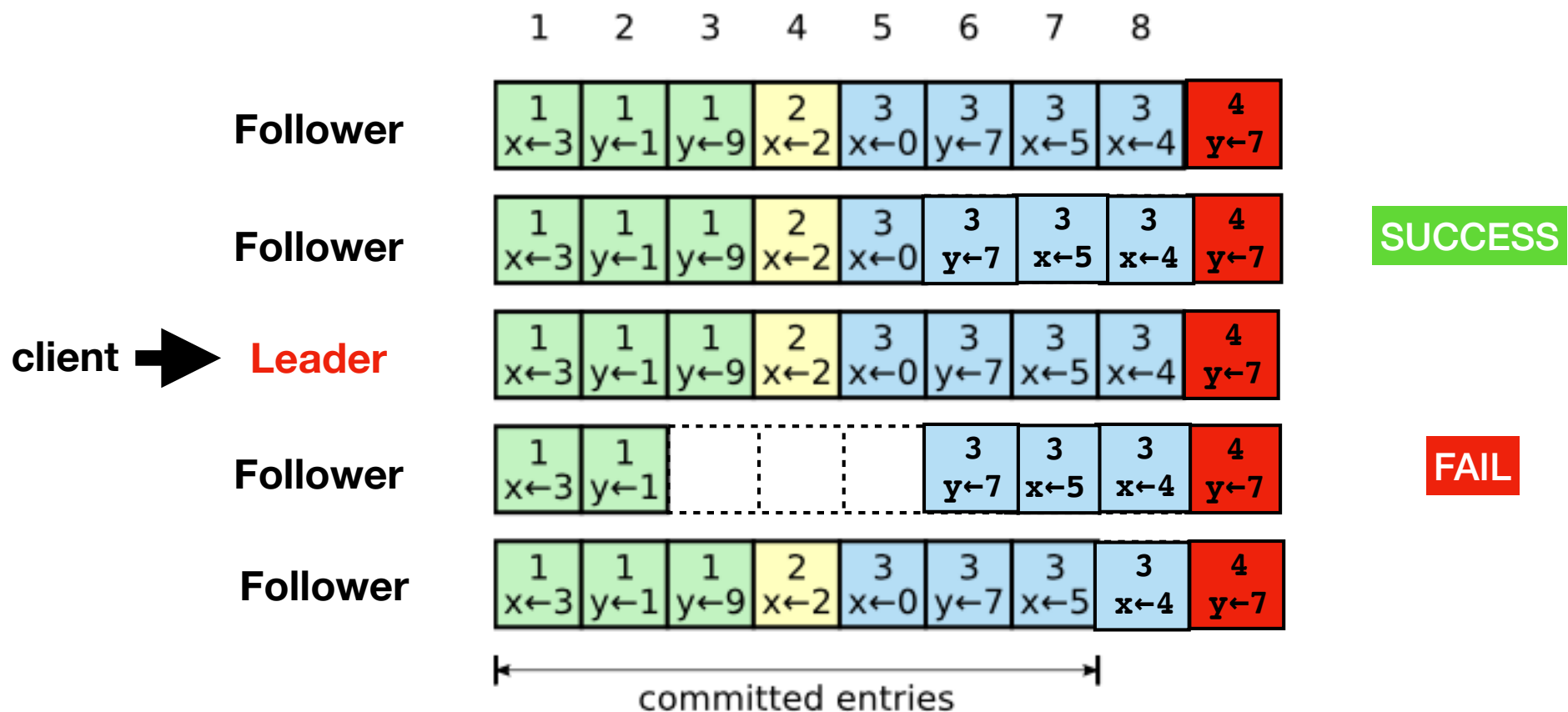
- This process repeats itself



- Leader keeps working backwards until success

Backtracking

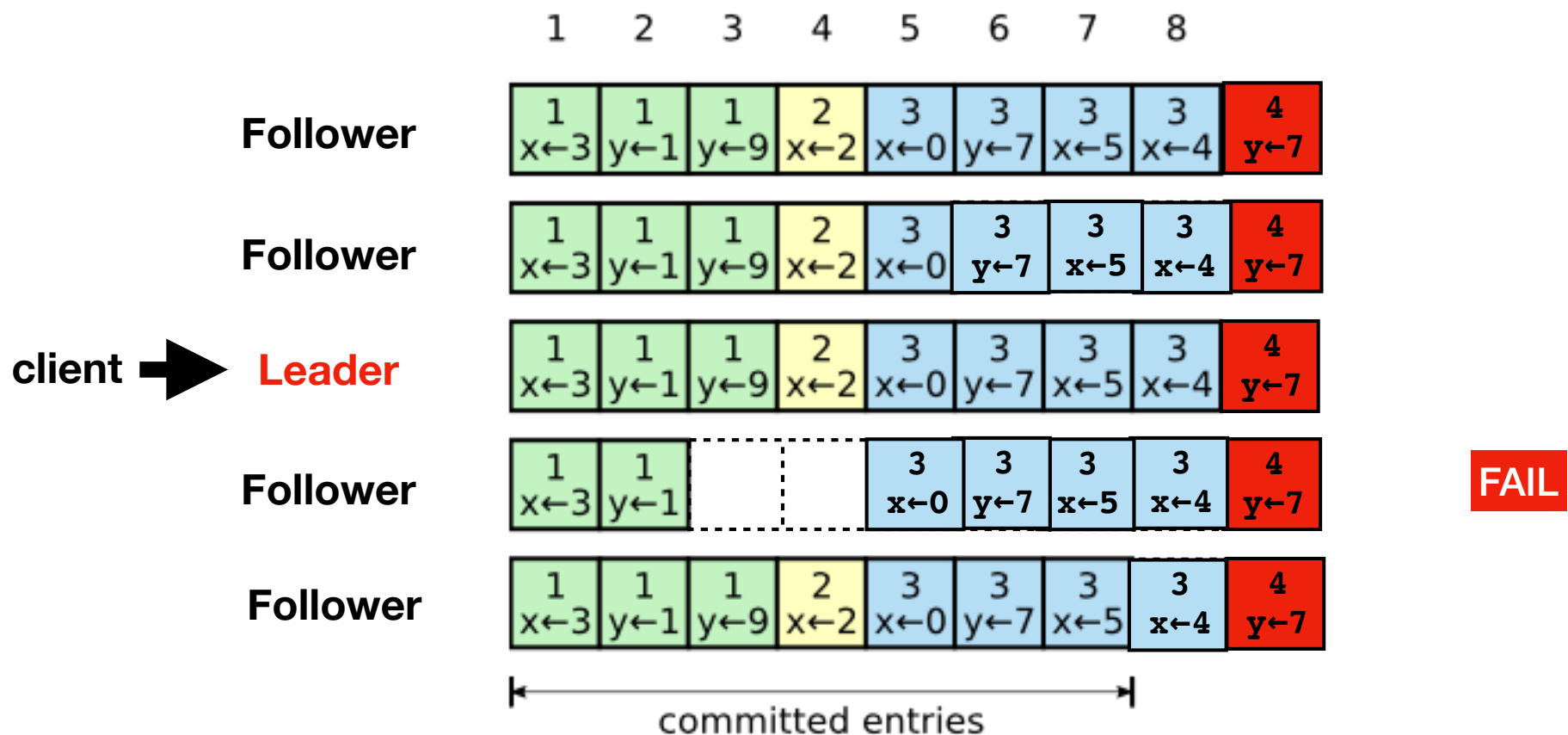
- This process repeats itself



- Leader keeps working backwards until success

Backtracking

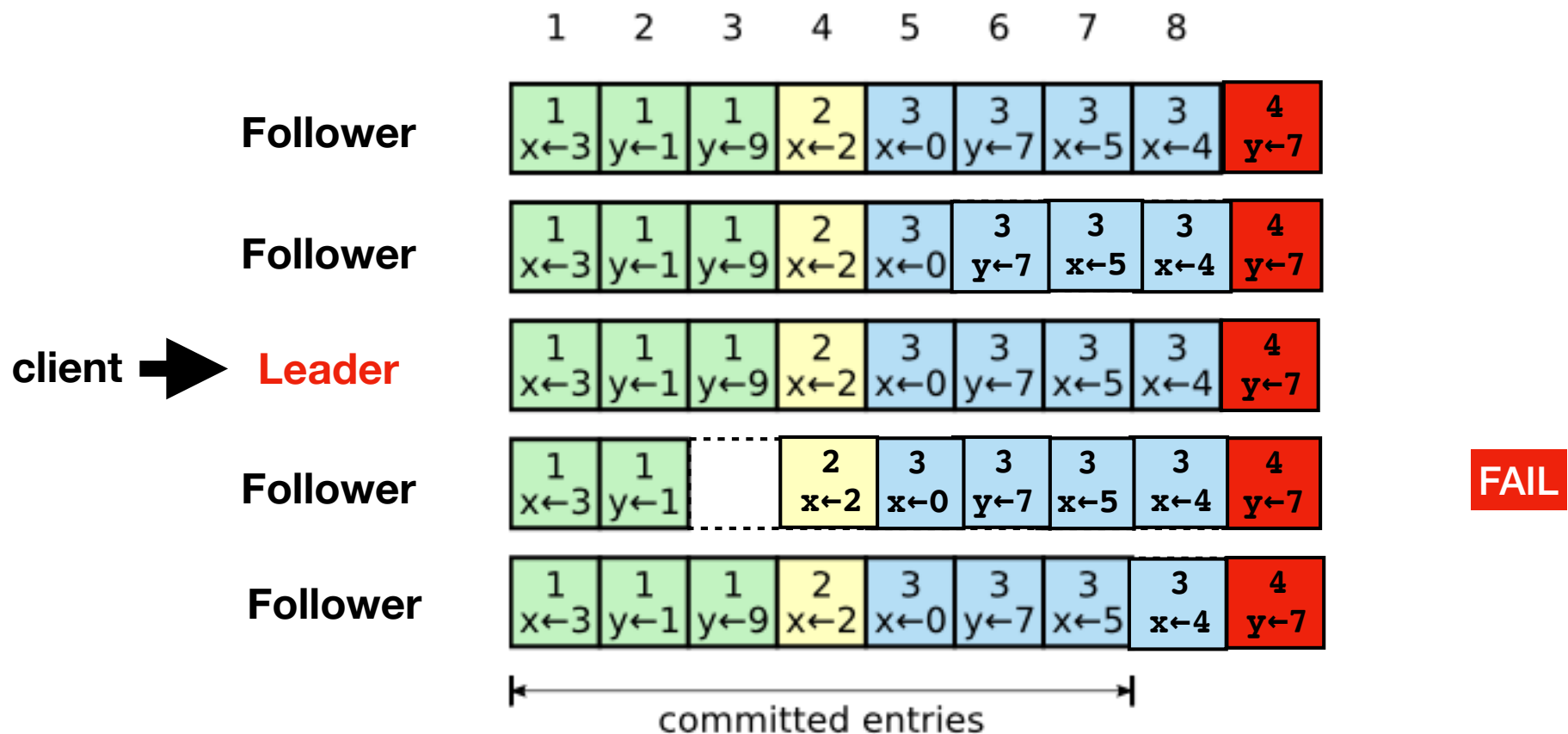
- This process repeats itself



- Leader keeps working backwards until success

Backtracking

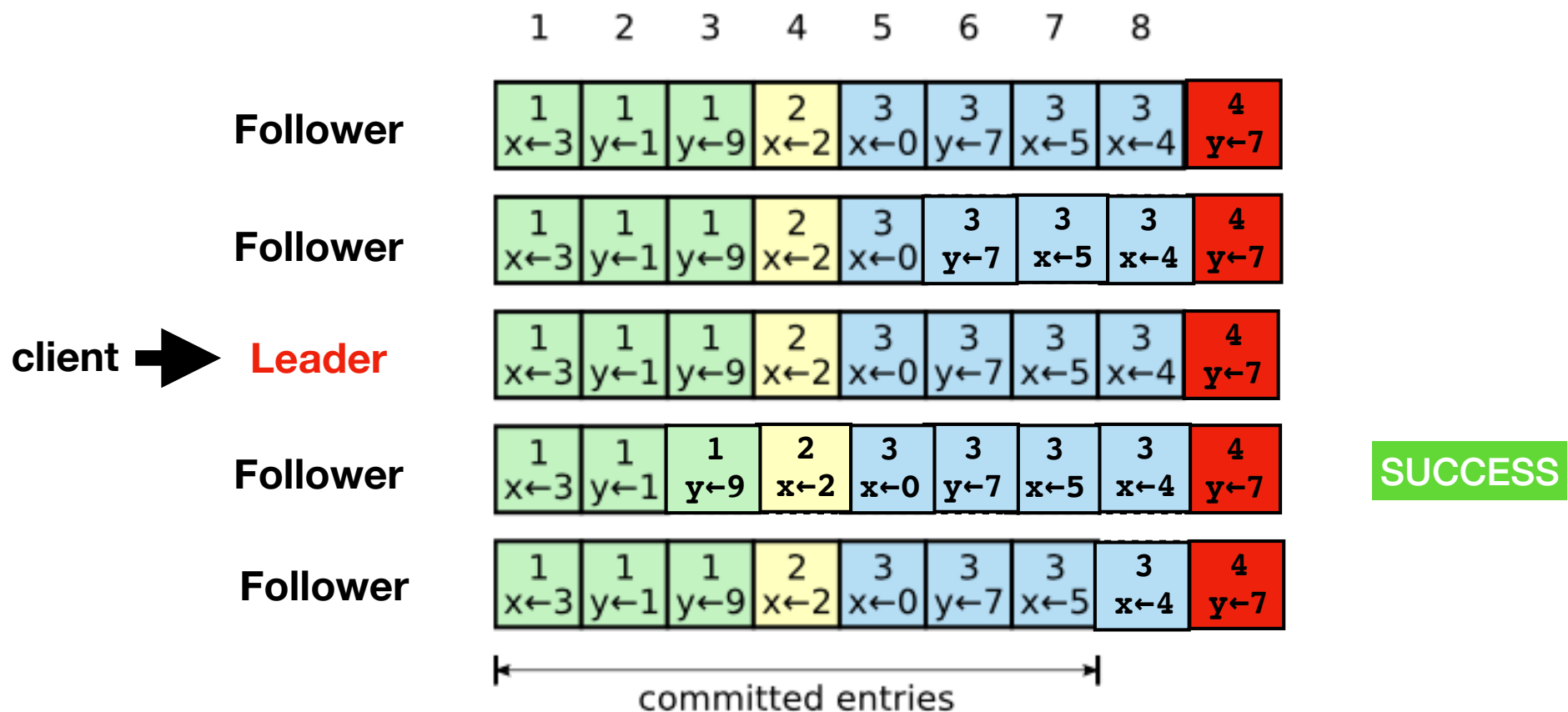
- This process repeats itself



- Leader keeps working backwards until success

Backtracking

- This process repeats itself



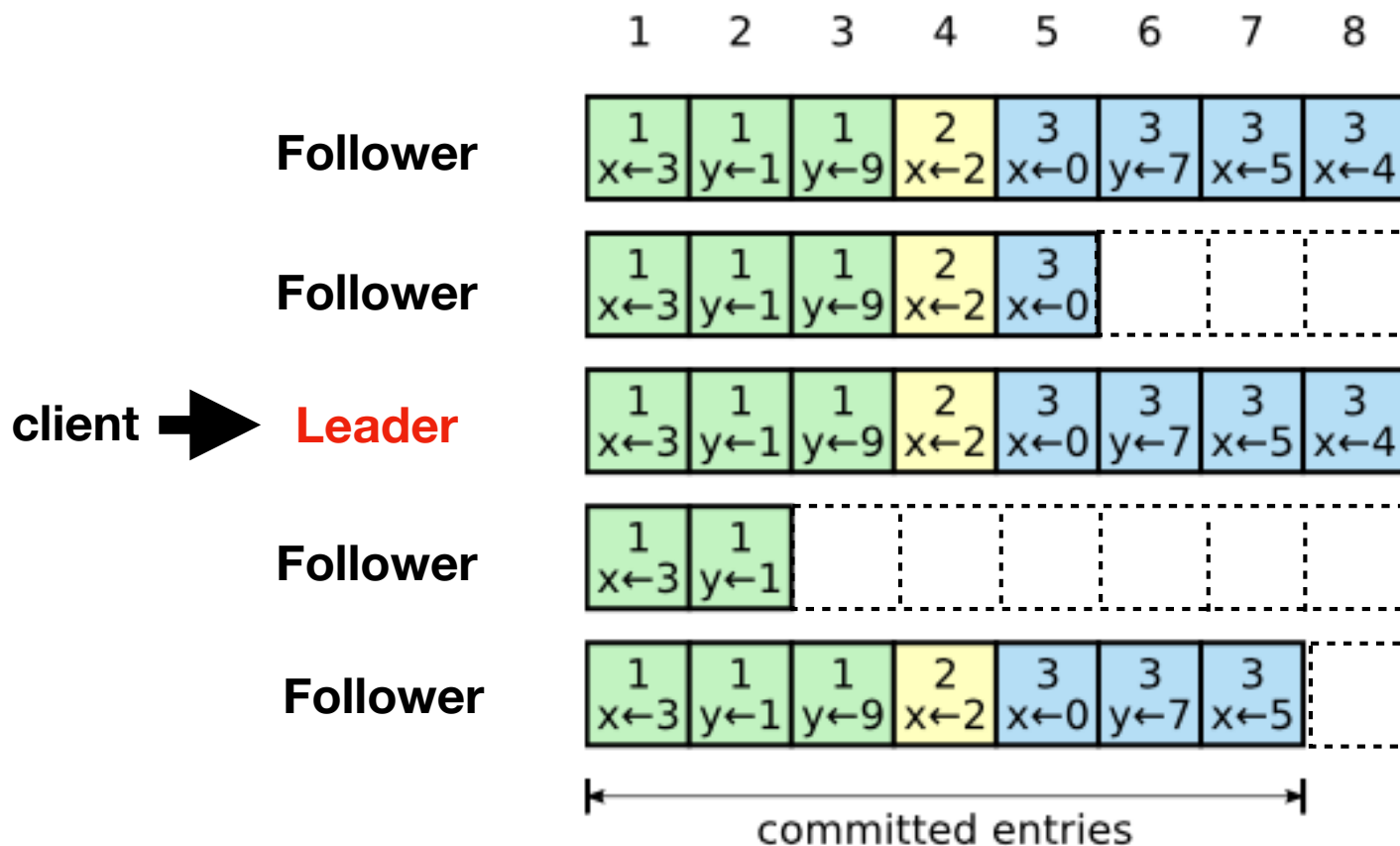
- Leader keeps working backwards until success

Commentary

- There is some book-keeping
- The leader needs to track the state of each follower independently
- But, follower states are 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)

Consensus

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



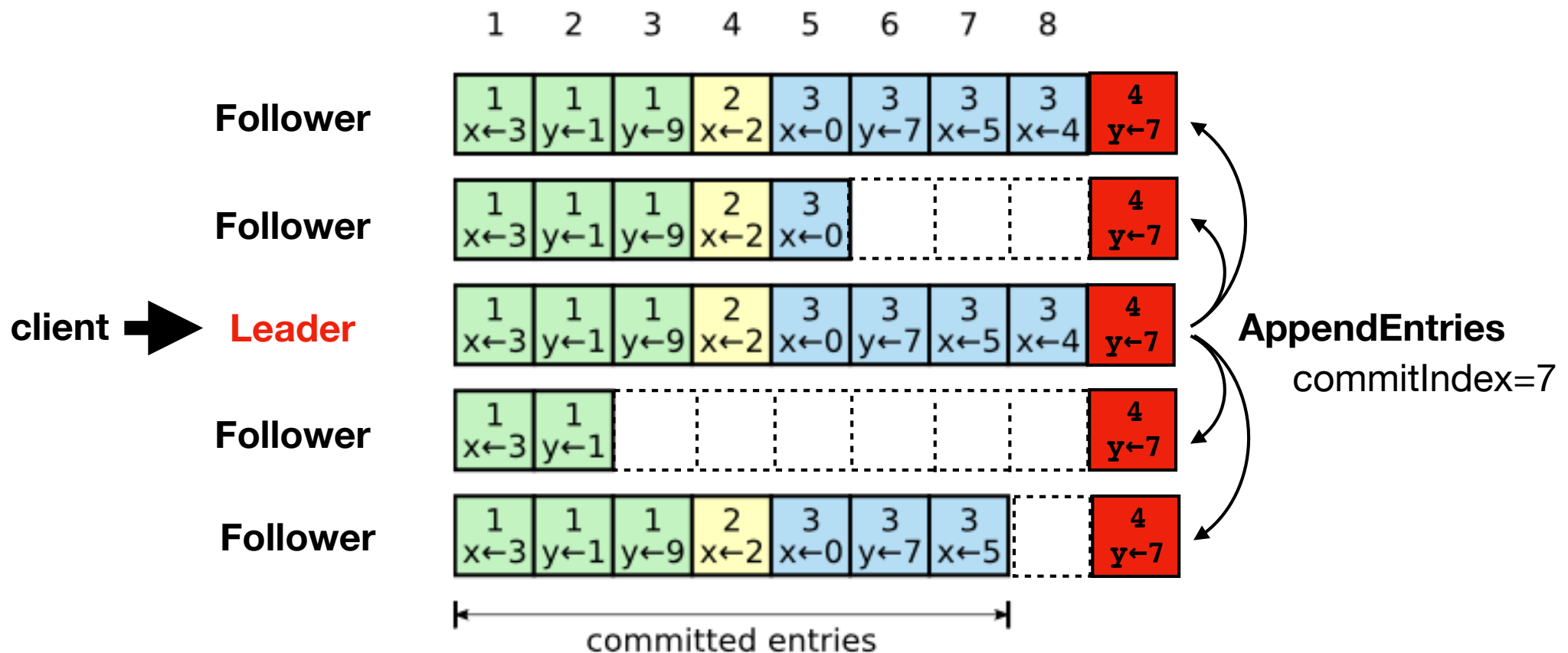
- When consensus is reached -- entries committed

Problem

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

Consensus

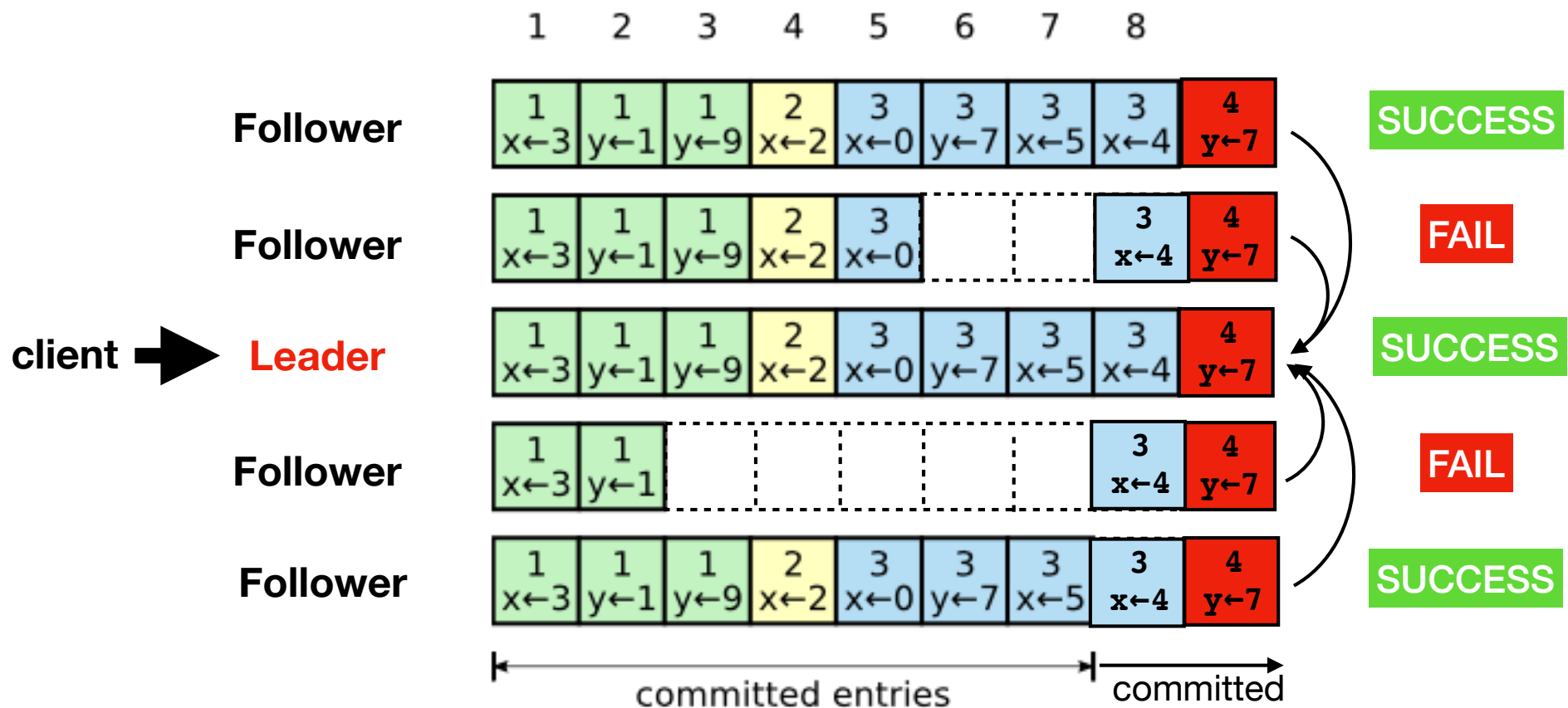
- The leader commit index is always sent



- Followers watch the index and apply to their logs

Consensus

- The leader updates the 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 you can "apply the log" to the state machine
- Basically, it means that each server can "do the thing" the client actually requested

Client

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

**There are some other
complicated issues here,
but ignore for now**

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
```

Project 6

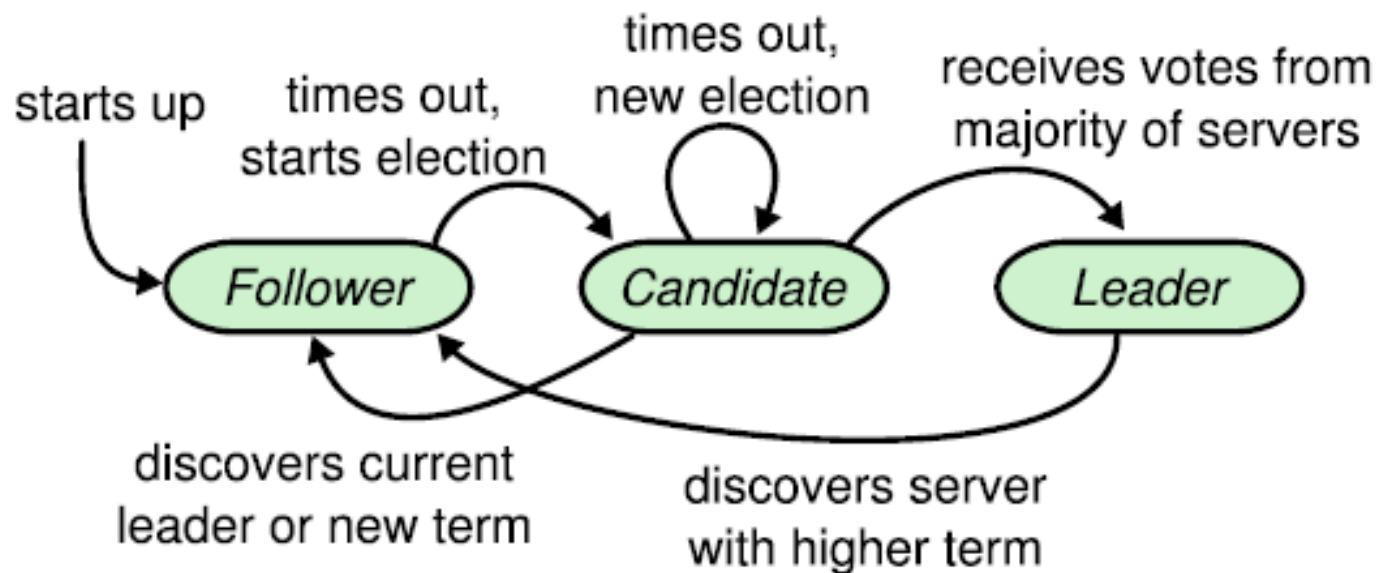
- Implement log replication via `AppendEntries`
- Designate one server as leader (manually)
- Implement a method to add an entry to its log
- Have the leader replicate to the followers
- Have the leader track consensus status

Part 7

Leader Election

Raft Operational States

- Servers in Raft operate in different states



- A major complexity concerns the transitions between the different states

The States

- 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 always 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

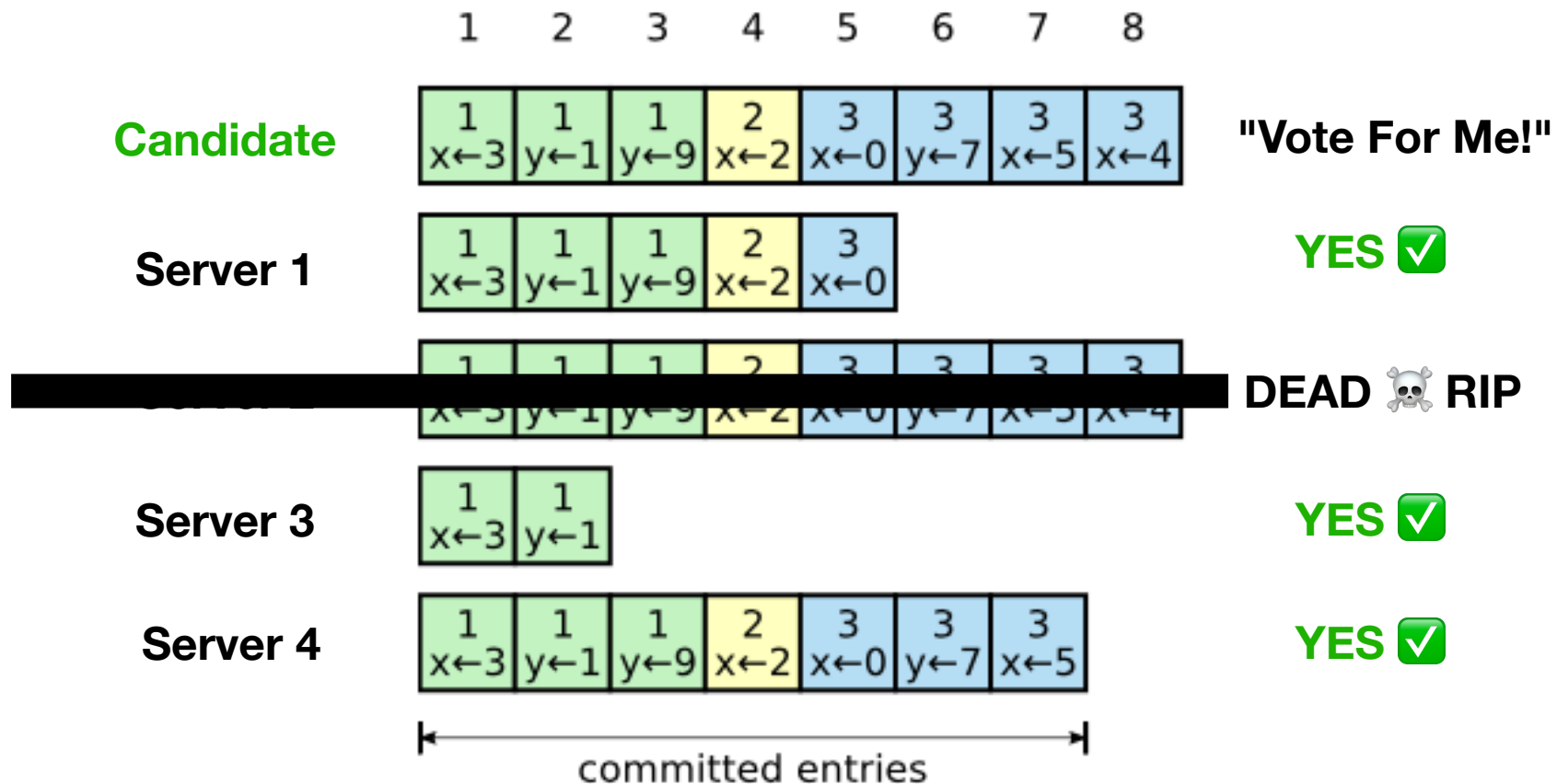
- A server may only vote for one candidate in a given term (subsequent requests denied)
- This is for dealing with split votes
- It's possible that two followers could promote to candidate at the same time. They'd have the same term number, but might not be able to get a quorum.

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.

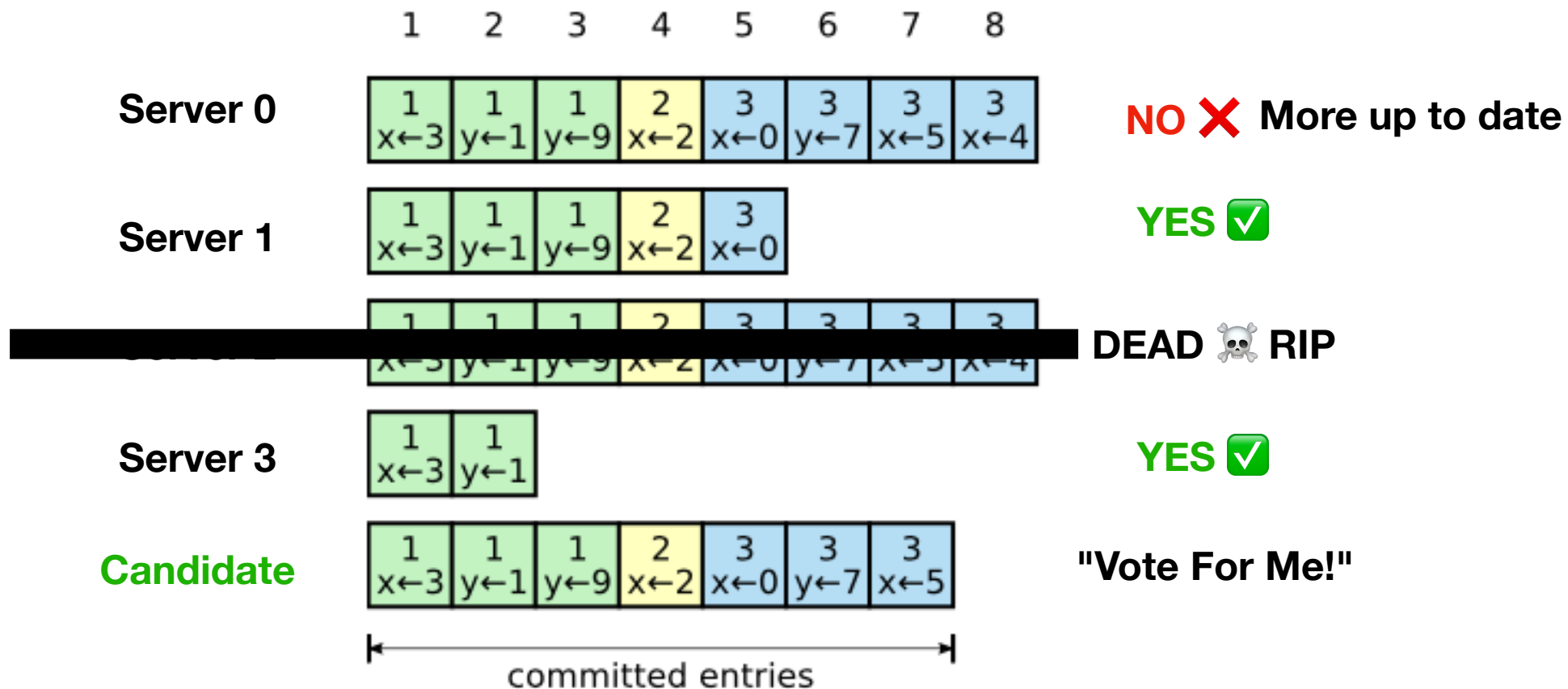
Voting Example

- 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

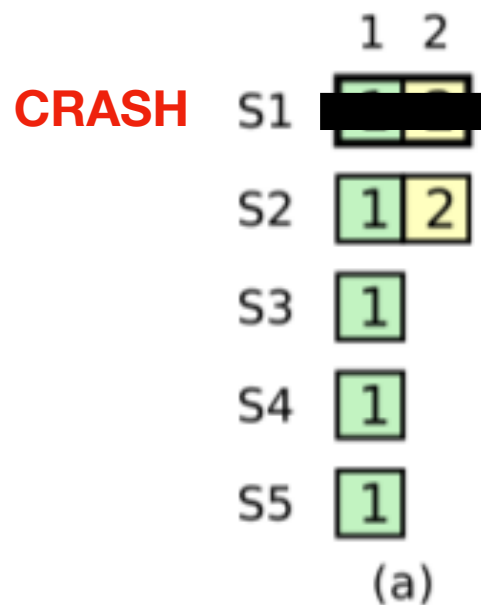
"Figure 8"

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



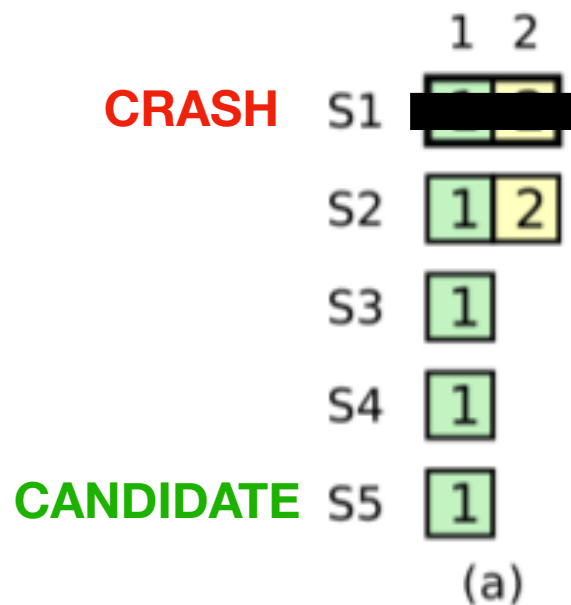
"Figure 8"

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



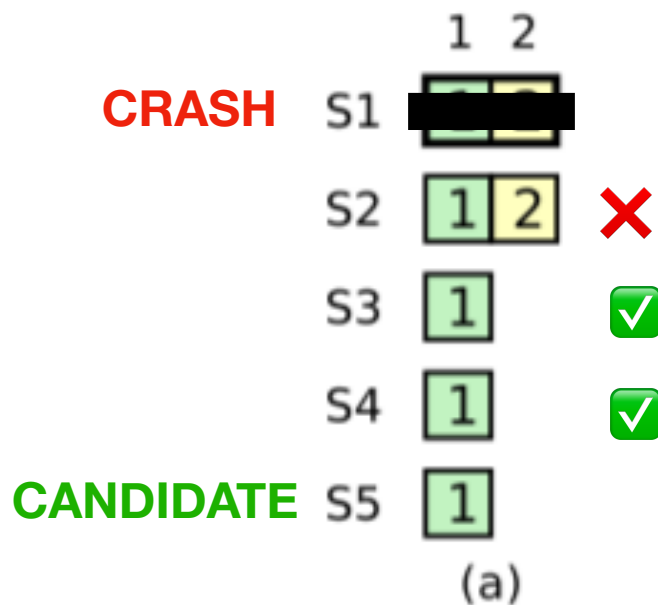
"Figure 8"

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



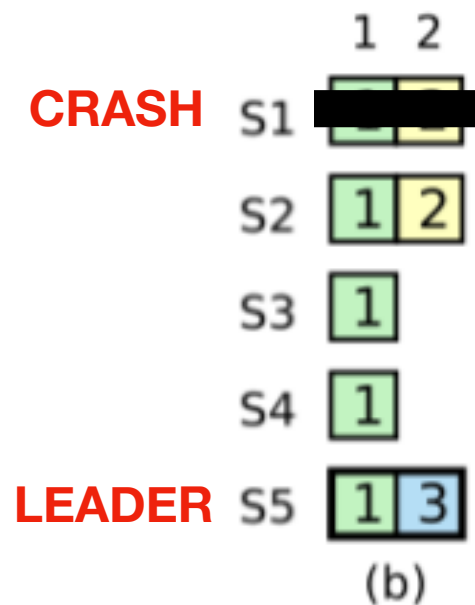
"Figure 8"

- 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



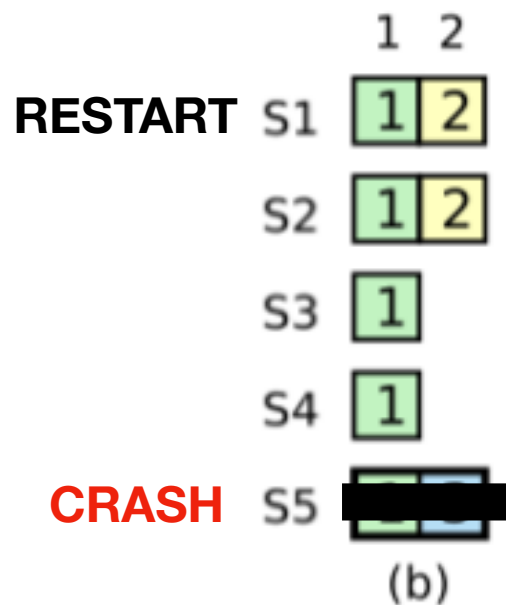
"Figure 8"

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



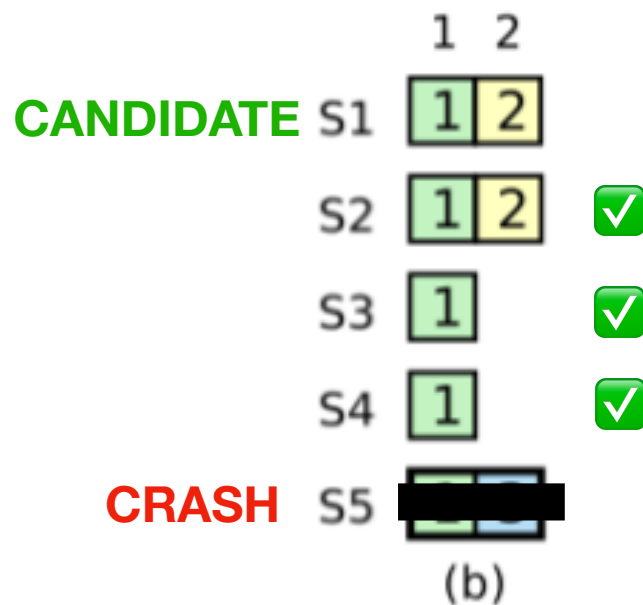
"Figure 8"

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



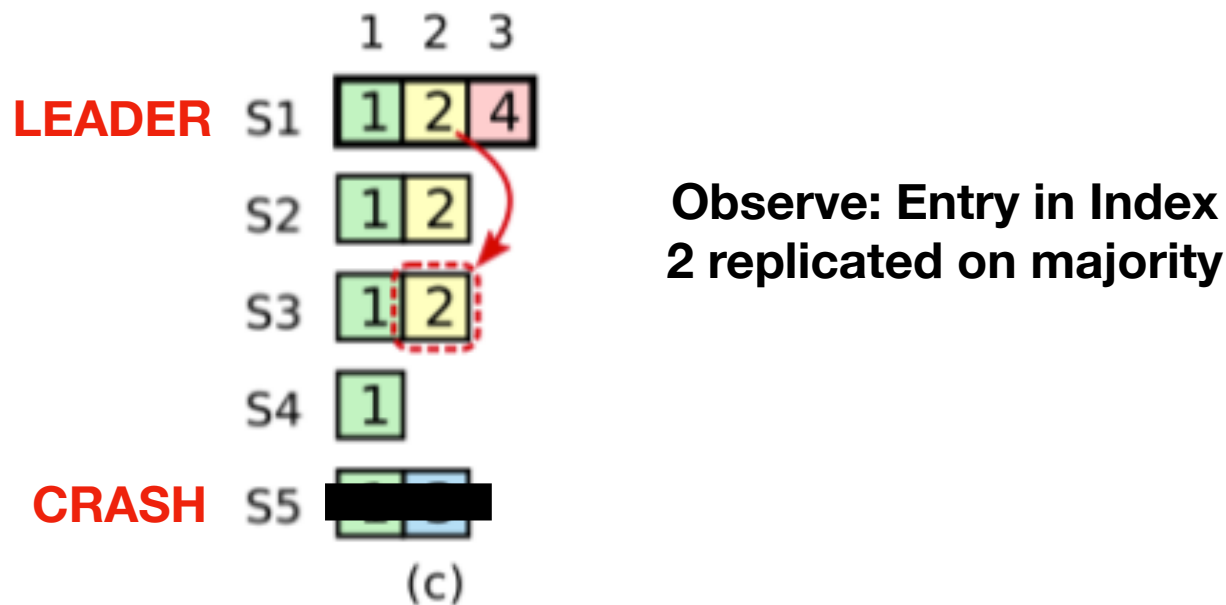
"Figure 8"

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



"Figure 8"

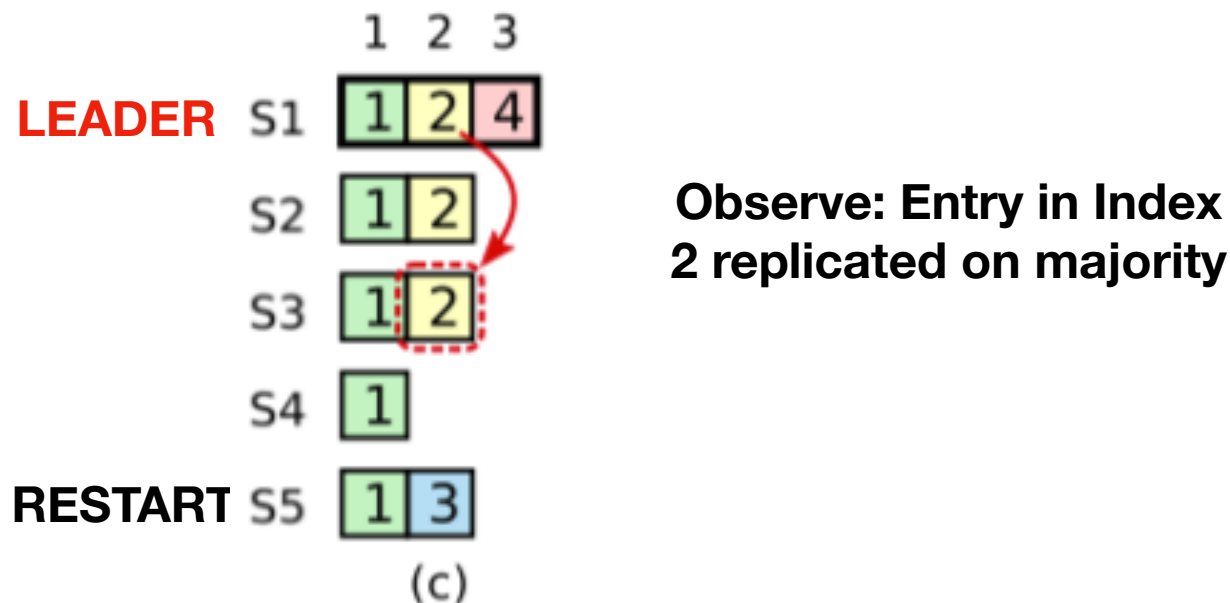
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



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

"Figure 8"

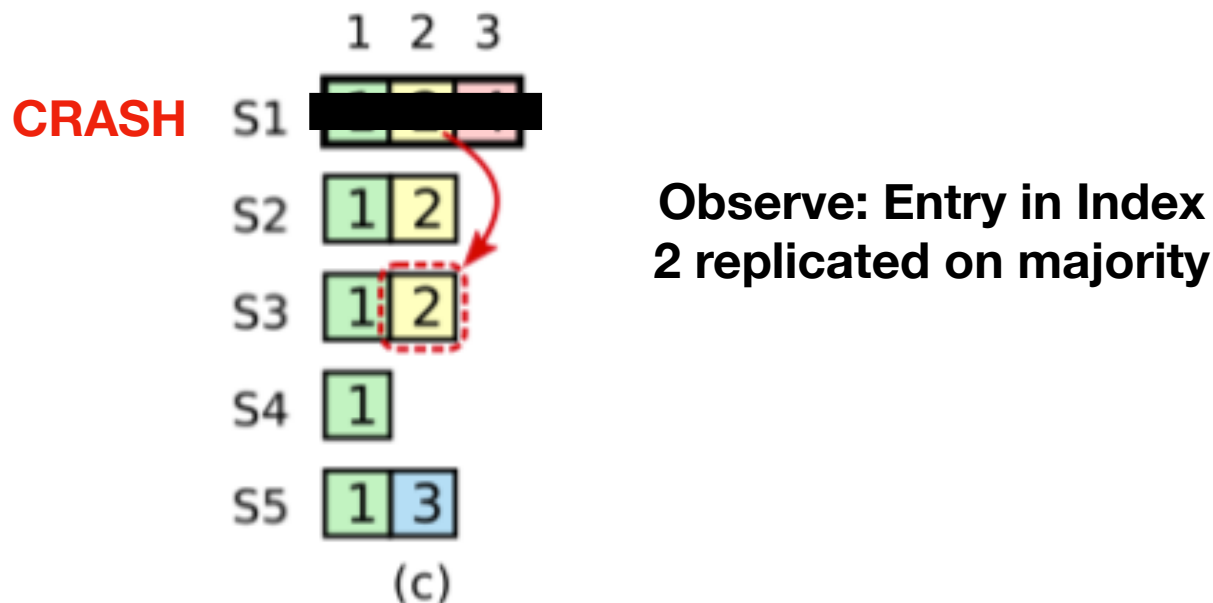
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



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

"Figure 8"

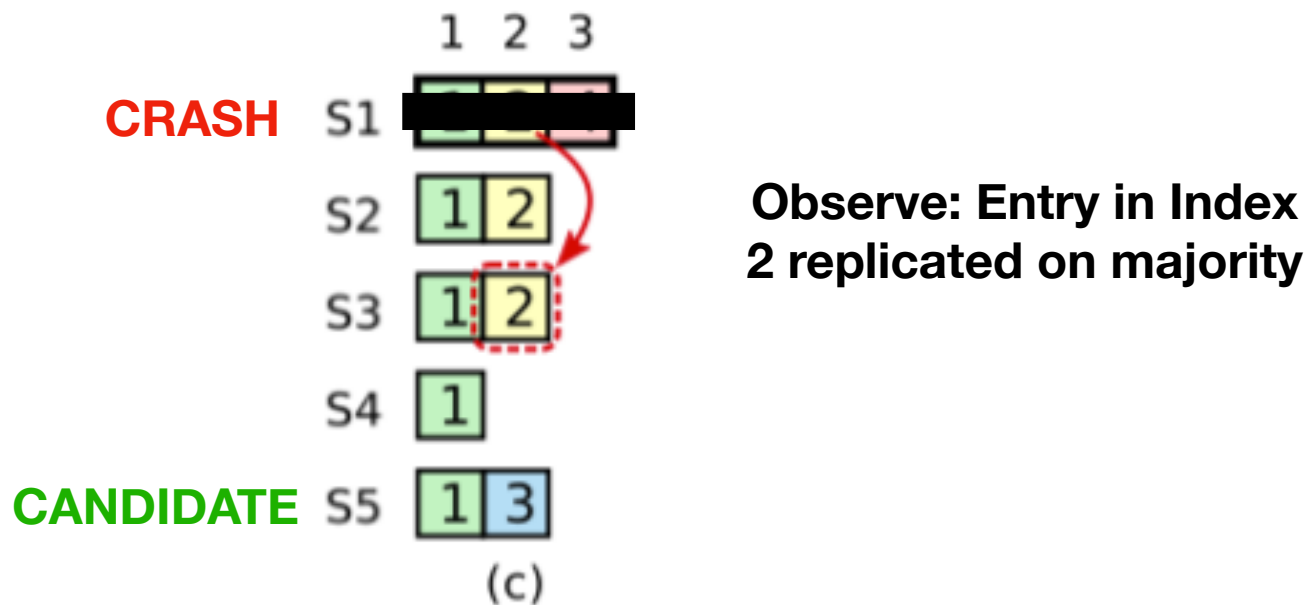
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



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

"Figure 8"

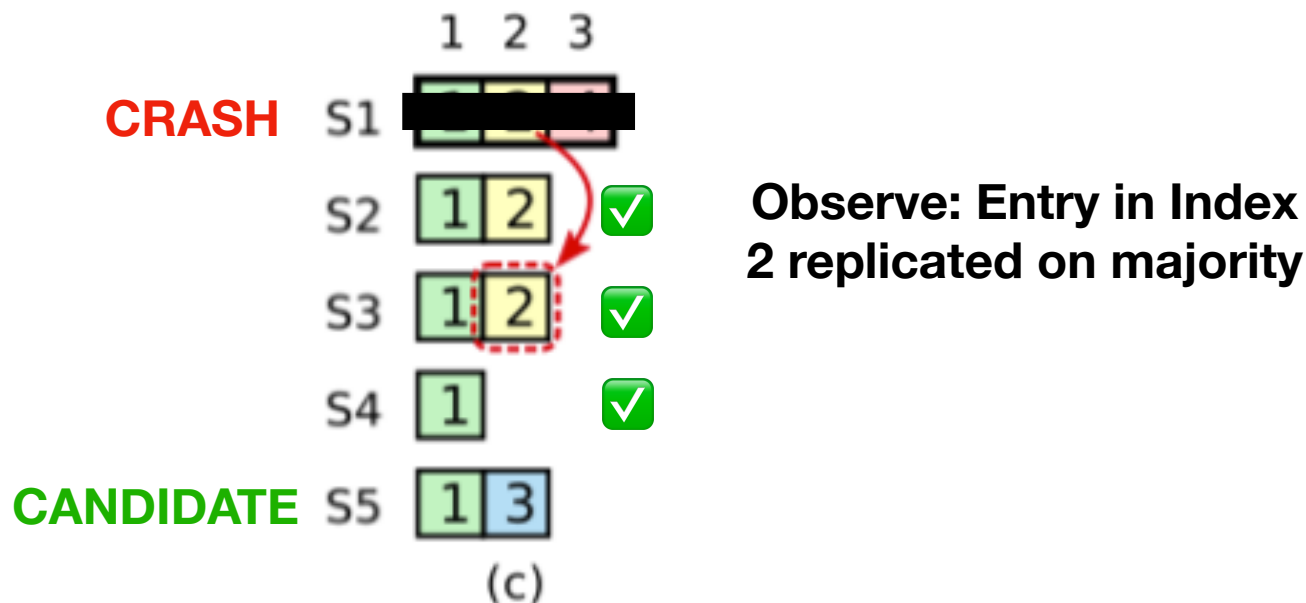
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



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

"Figure 8"

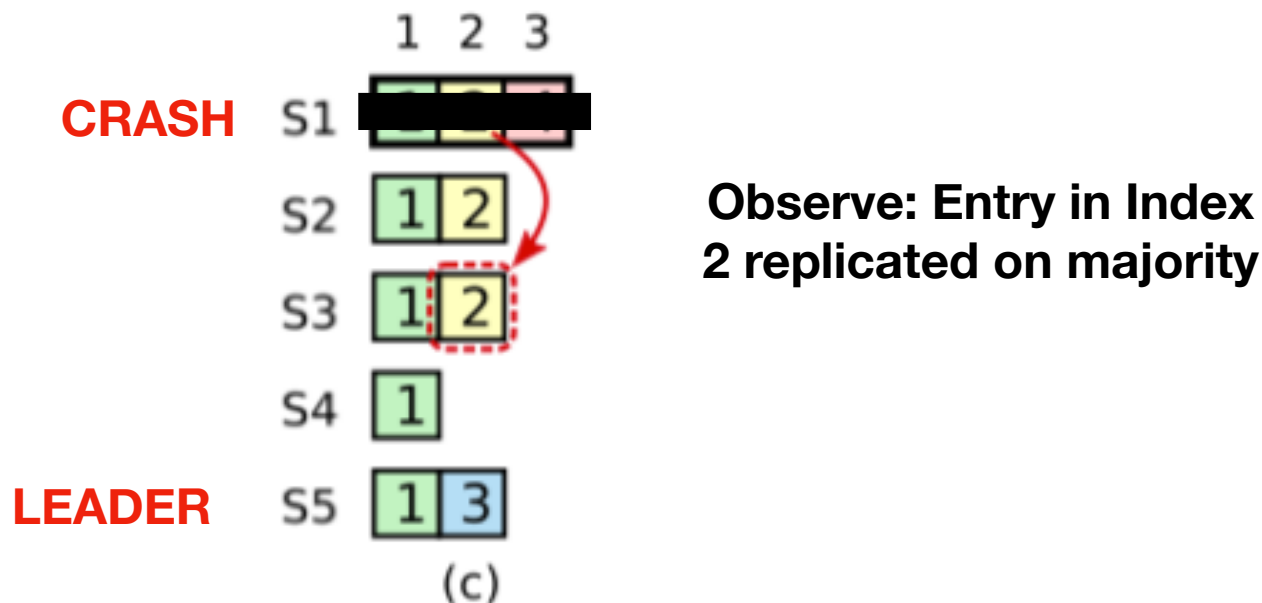
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



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

"Figure 8"

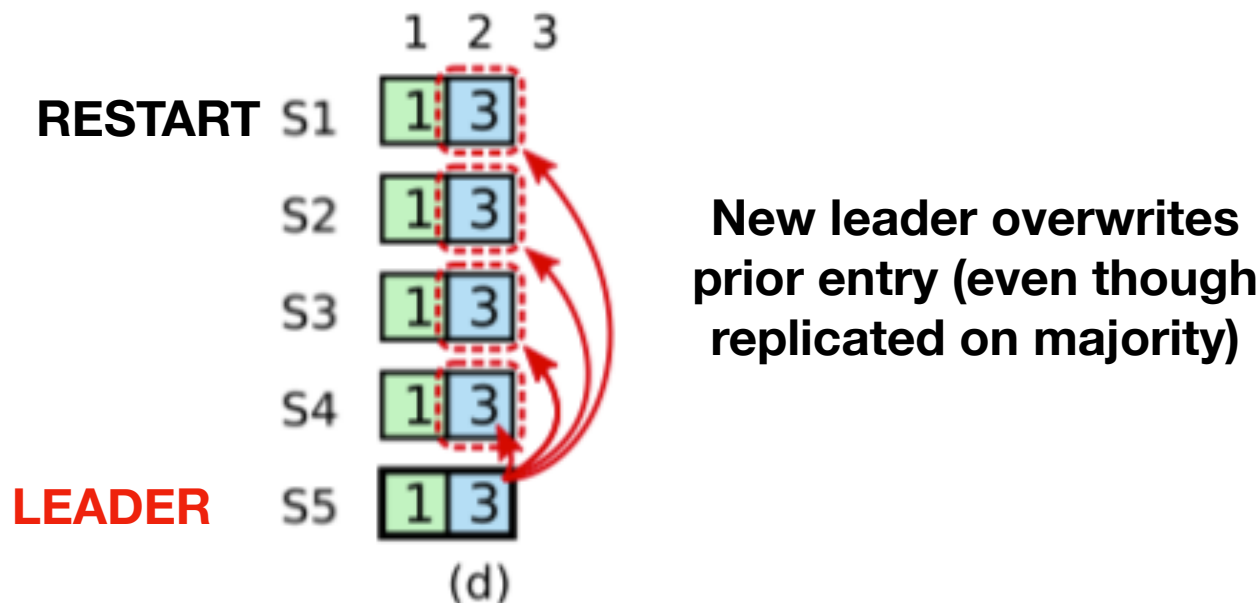
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



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

"Figure 8"

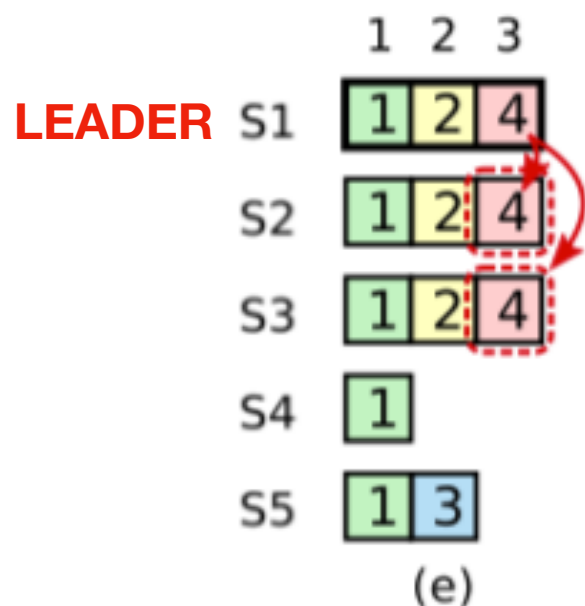
- A newly elected leader can never commit entries from the previous leader--even if replicated on majority



- 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 7

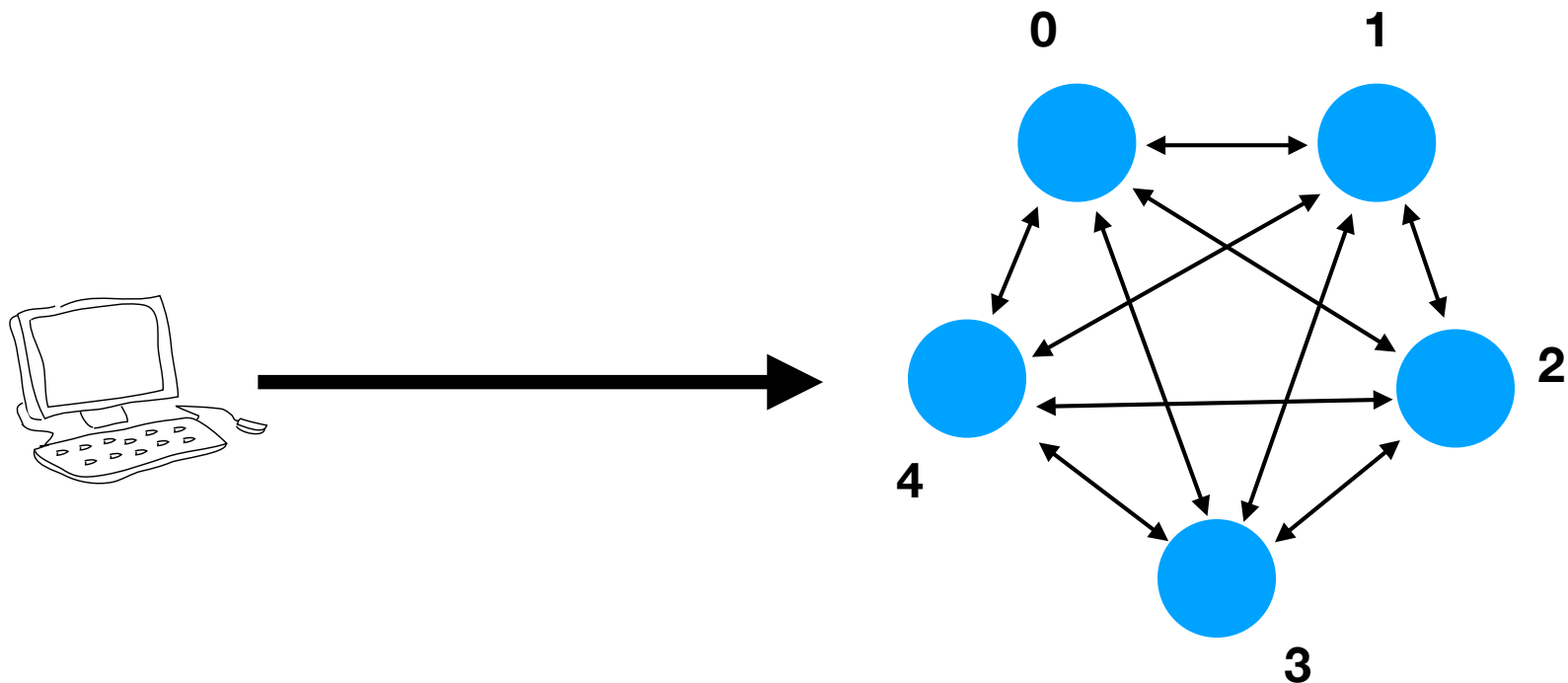
- Implement the Raft state machine for leader/candidate/follower selection
- Try to do it in a manner that can be tested/verified in some sane way

Part 8

Client-Raft Interaction

Client-Connection

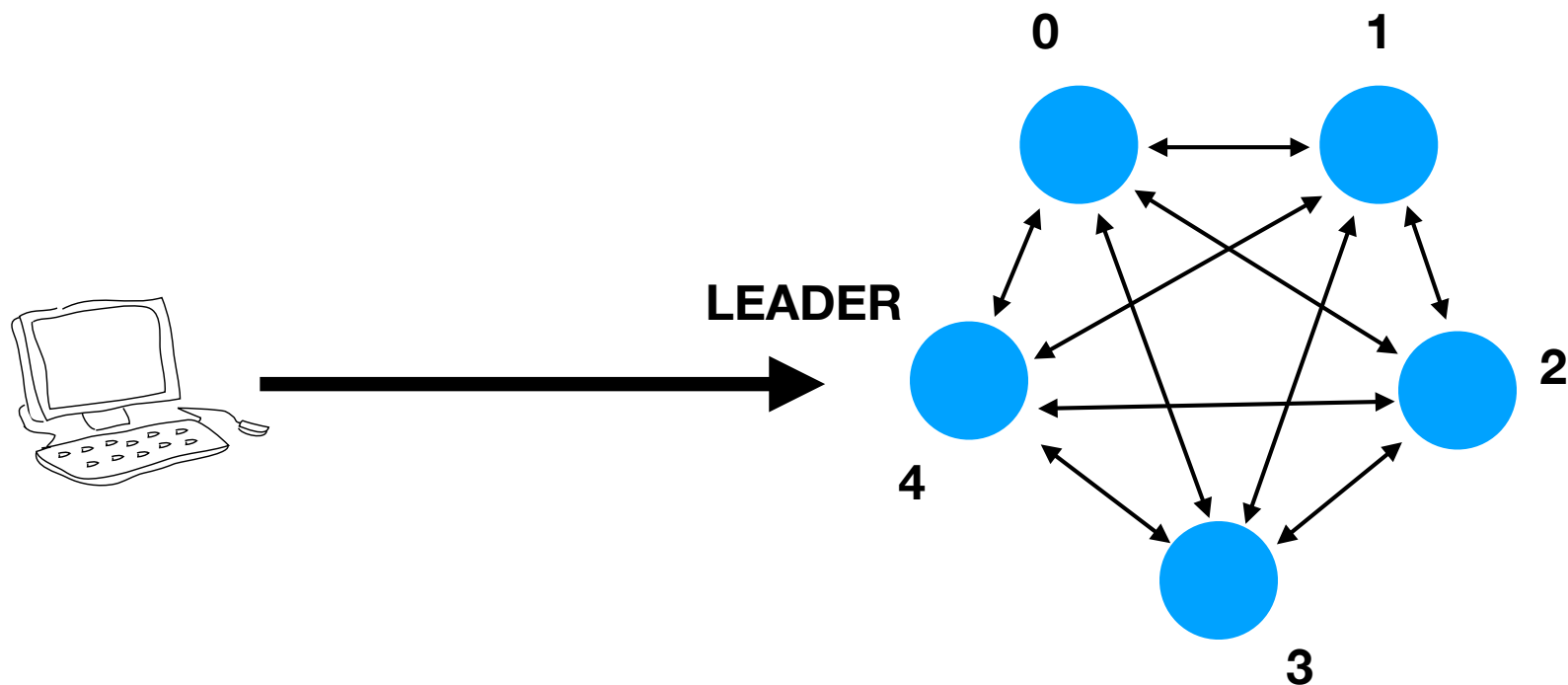
- At some point, a client must talk to Raft



- For example, to implement a key-value store

Strong Leader

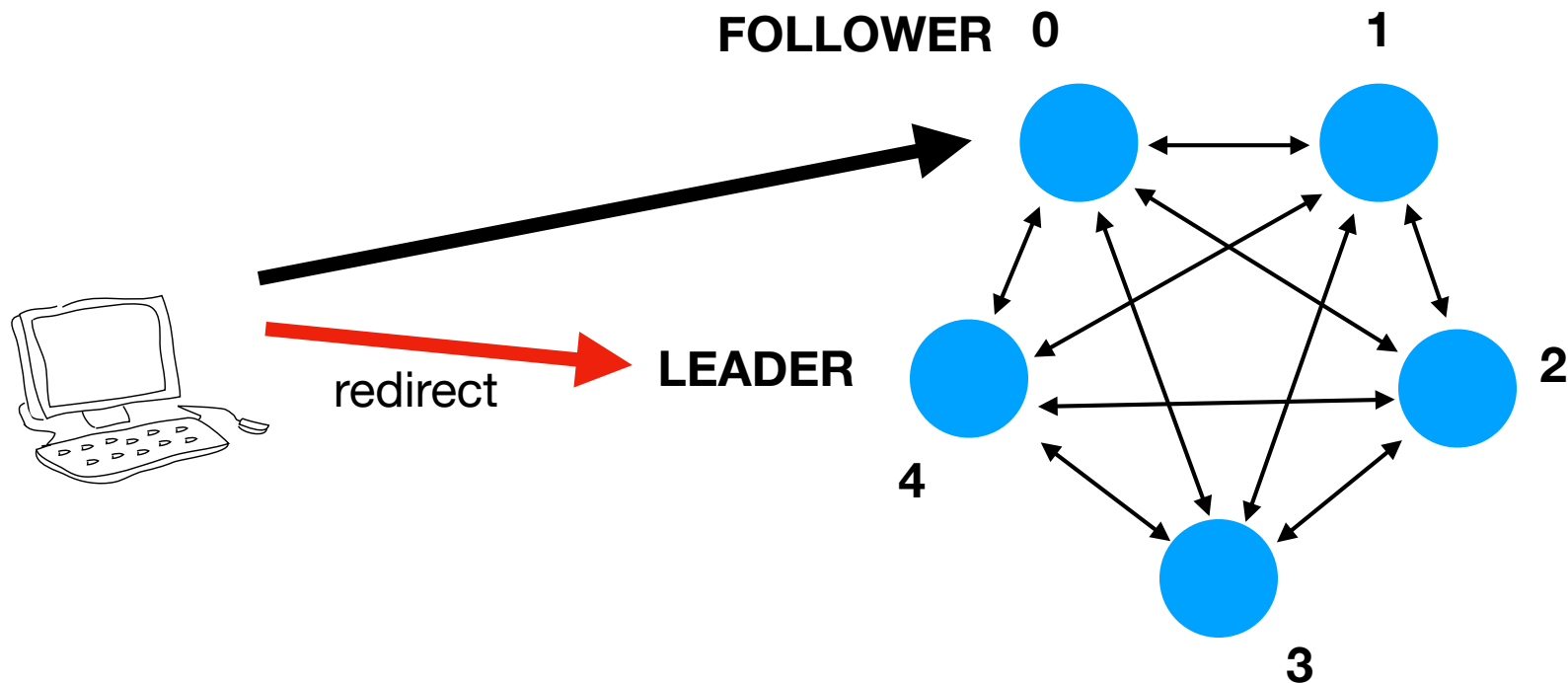
- Clients only talk to the leader



- 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

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 executed 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 requests, don't re-execute requests if already executed.

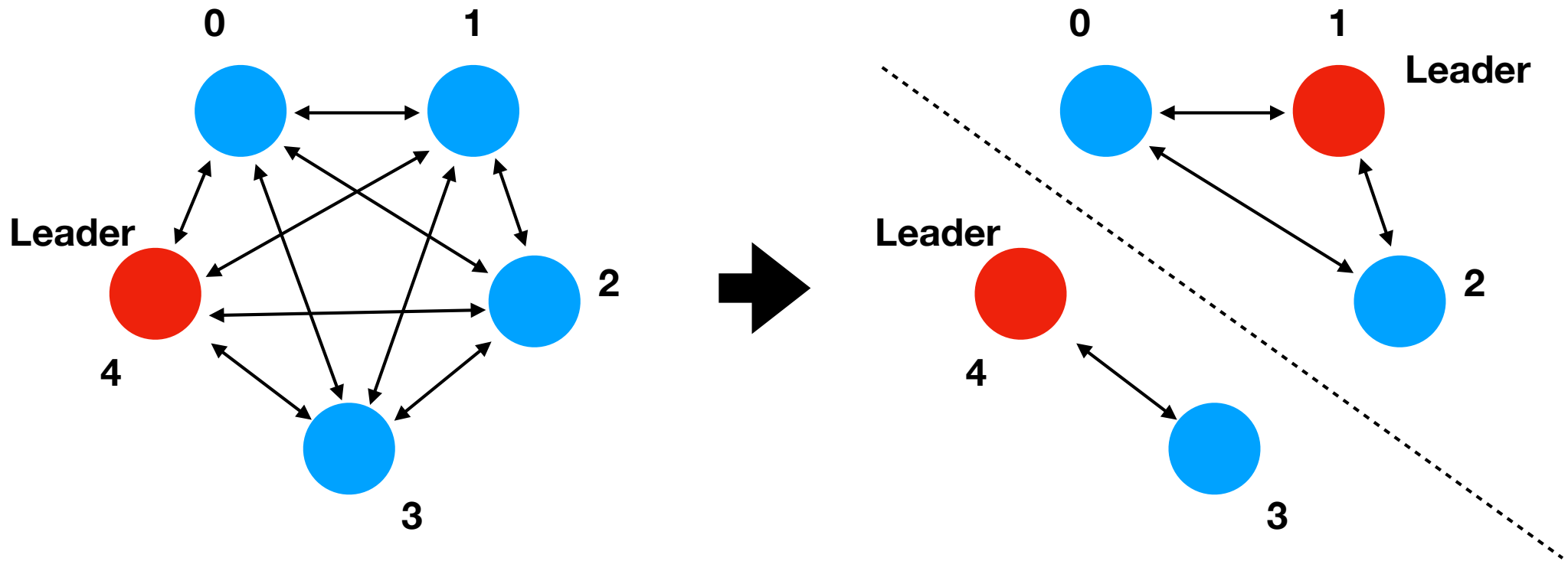
Read Requests

- Read requests still involve the leader, but don't involve new log entries
- However, this poses new challenges
- Raft wants linearized semantics

```
kv.set('foo', 42)  
...  
assert kv.get('foo') == 42
```

- `get()` after `set()` should return same value

Problem: Partitions



- Could experience a network partition that results in two leaders.
- Former leader doesn't know it's cut off.

Solution

- Leader is not allowed to respond to a read request until it has exchanged a heartbeat message with a majority of the cluster after it has received the read-request

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)
- Requires interaction with followers

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
- Subtlety: A leader can never commit entries from a previous leader before it commits an entry from its own term (see section 5.4.2)

Project 8

- Read section 8 of the Raft paper.
- Implement Raft client interaction
- Implement a fault-tolerant Key-Value store