

Rafting Trip

(AKA: Distributed Systems)

<http://www.dabeaz.com/raft.zip>

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 graduate distributed systems class (6.824)

6.824 - Spring 2018

6.824 Lab 2: Raft

Part 2A Due: Feb 23 at 11:59pm

Part 2B Due: Mar 2 at 11:59pm

Part 2C Due: Mar 9 at 11:59pm

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 graduate distributed systems class (6.824)

6.824 - Spring 2018

6.824 Lab 3: Fault-tolerant Key/Value Service

Due Part A: Mar 16 at 11:59pm

Due Part B: Apr 13 at 11:59pm

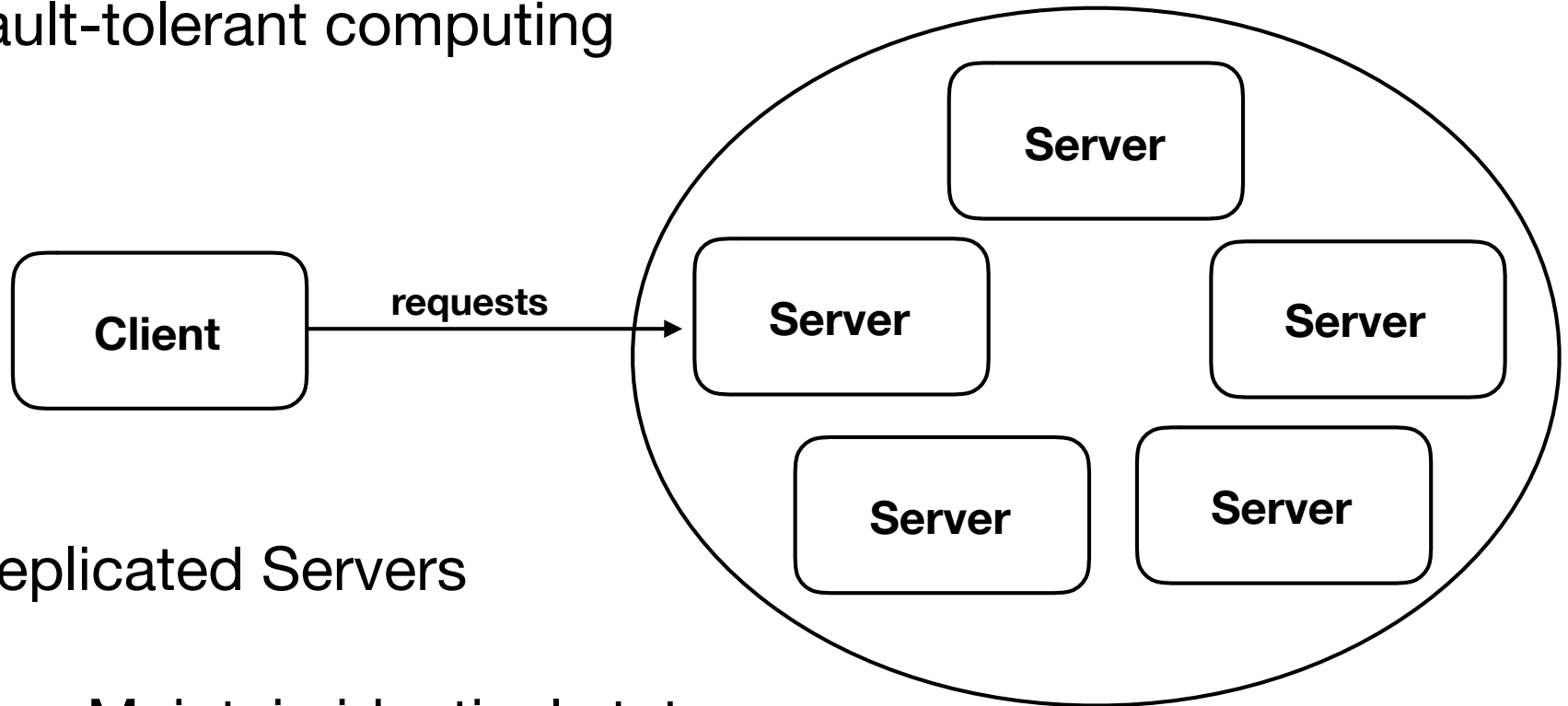
Introduction

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 to maintain 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

- Fault-tolerant computing



- Replicated Servers
 - Maintain identical state
 - Operate via majority consensus
 - Tolerant to the failure of any minority subset

Historical Background

- Consensus : A central problem in distributed systems about maintaining reliability in the presence of failures.
- Most well known algorithm: Paxos (Leslie Lamport).
 - First published (1989), First Journal Article (1998, submitted 1990)
 - Notable for having a formal proof of correctness
- 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

For example, consider the following excerpt from one of the most well-known papers on the subject: Leslie Lamport's Paxos Made Simple (which, incidentally, claims to explain Paxos in "plain english"):

P2c . For any v and n , if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either (a) no acceptor in S has accepted any proposal numbered less than n , or (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S .

"Every consensus protocol out there or every fully distributed consensus protocol is either Paxos or Paxos with cruft or broken"

- Mike Burrows

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



Open access to the Proceedings of
USENIX ATC '14: 2014 USENIX Annual Technical
Conference is sponsored by USENIX.

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

Why This Topic?

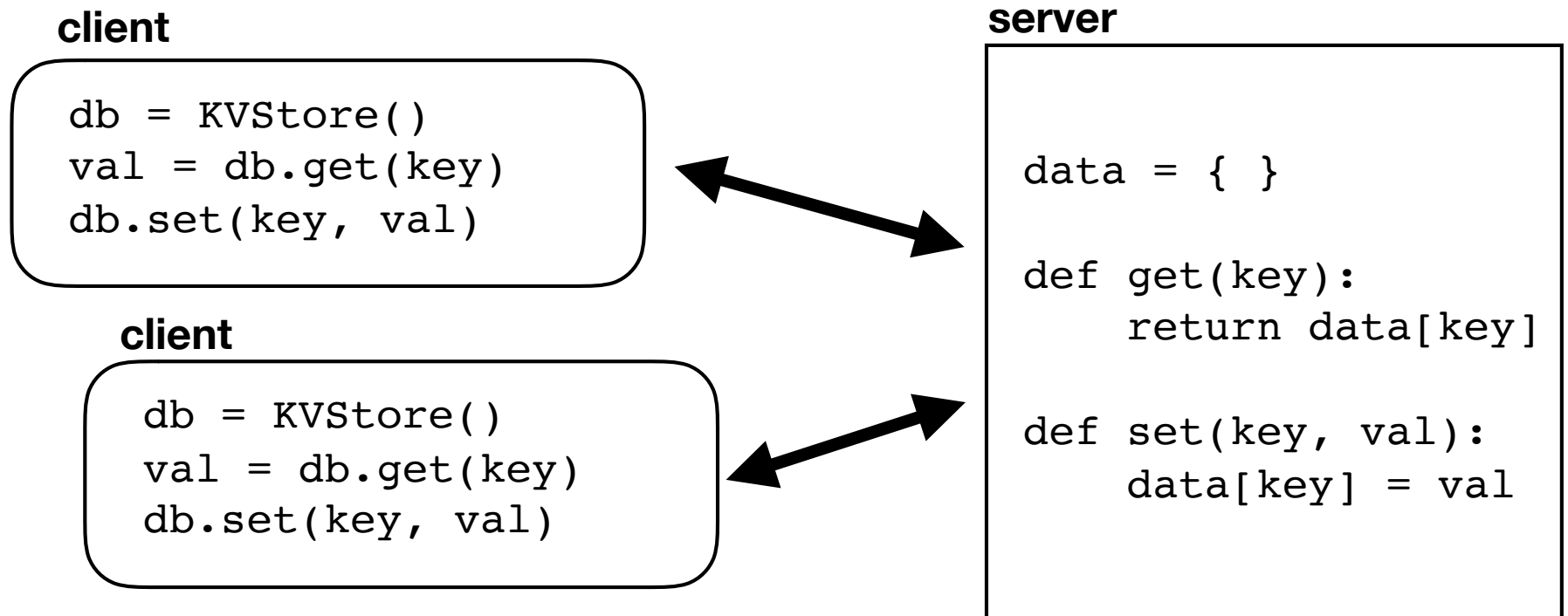
- Real world: "Fault Tolerance" sounds good
- Self-contained: It's a "small problem"
- Non-trivial. Many moving parts. Not an "echo server."
- Challenging: concurrency, networks, testing, etc.
- Solving it transcends the details of just this algorithm

Core Topics

- Messaging and networks
- Threads and processes
- State machines
- Software architecture/OO
- Formal specification/testing

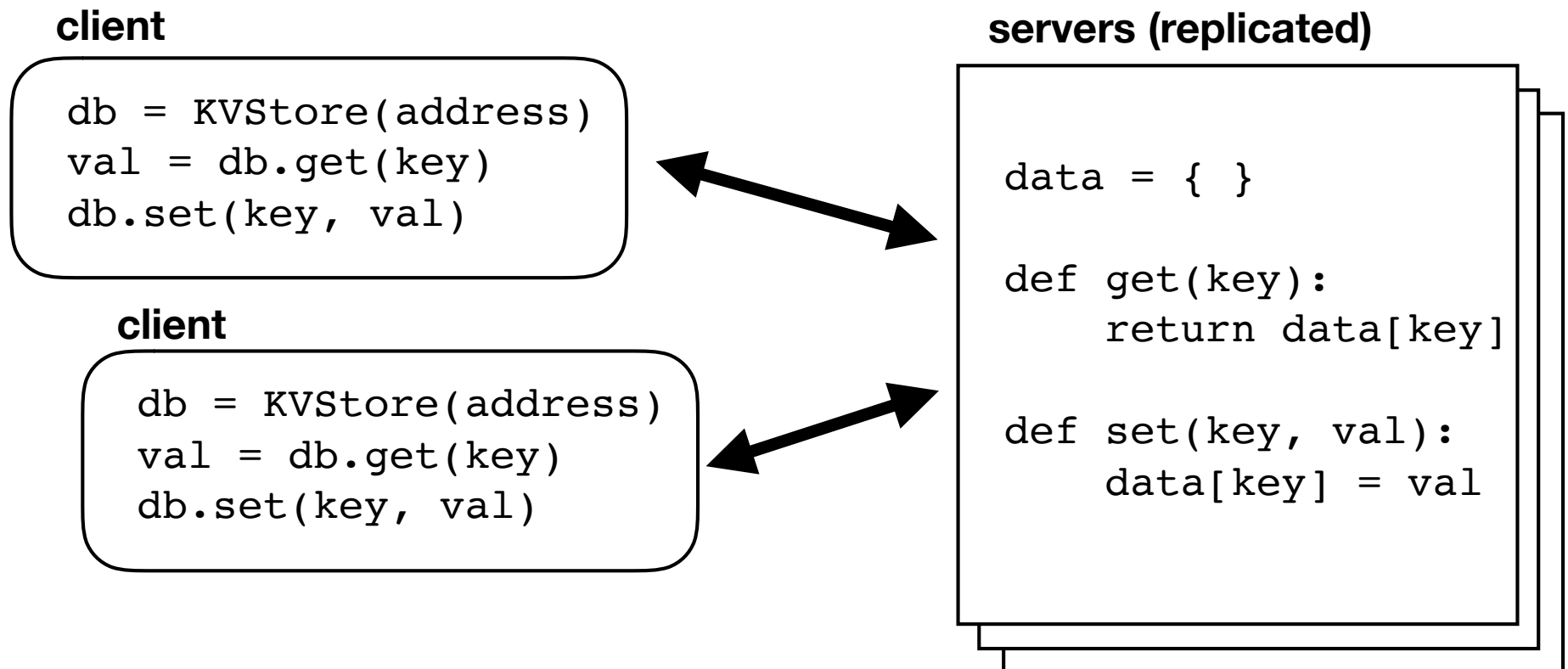
The Project

- We're going to build a distributed key/value store
- In a nutshell: A networked dictionary



The Problem

- Fault-tolerance
- Always available, can never lose data

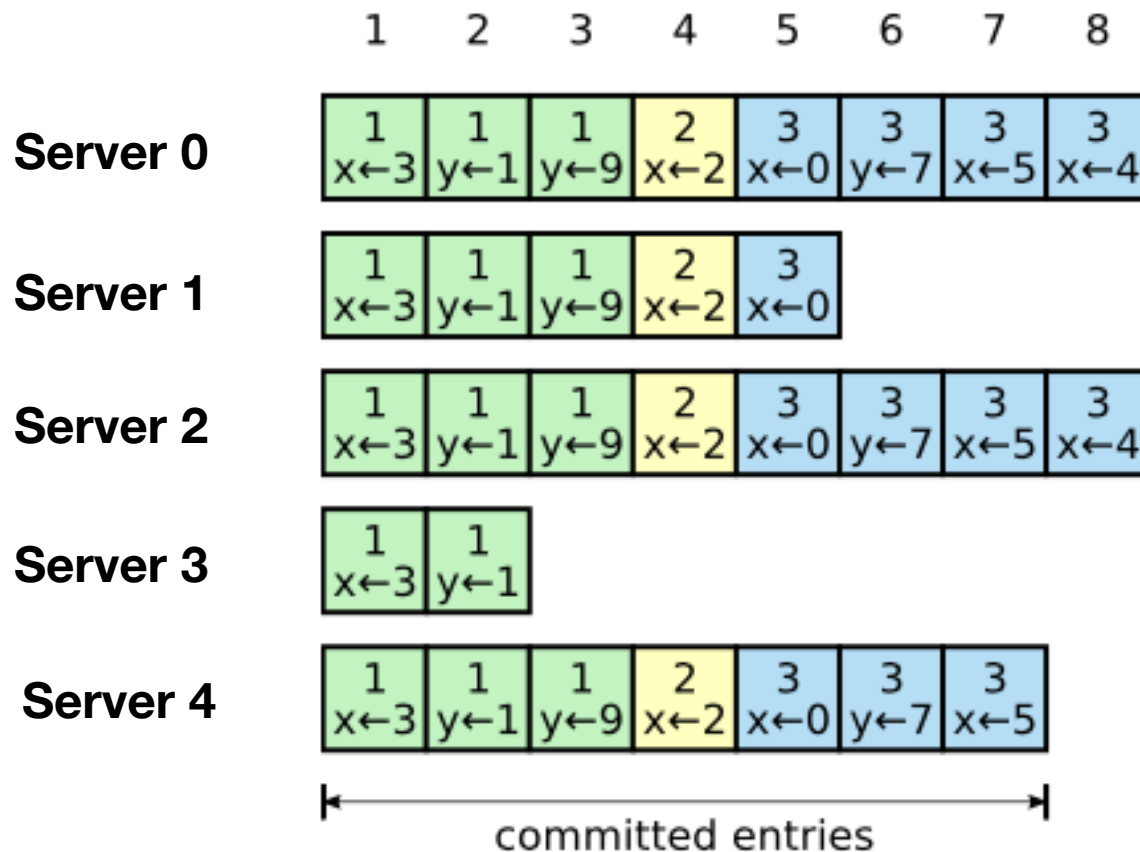


Raft

- Raft is an algorithm that solves this problem
- I will attempt to explain how in 3 slides
- There are a few core principles

Replicated Logs

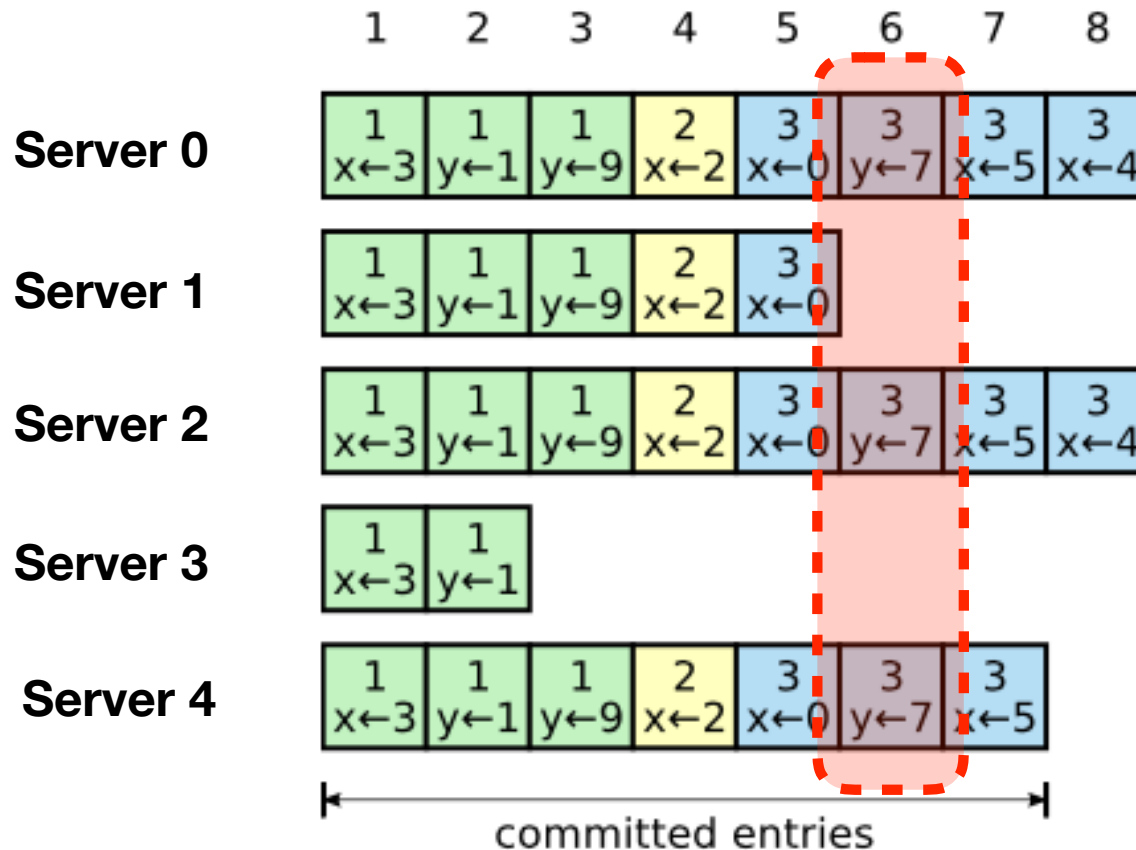
- The servers maintain a replicated transaction log



- Log contains committed and uncommitted entries

Majority Rules

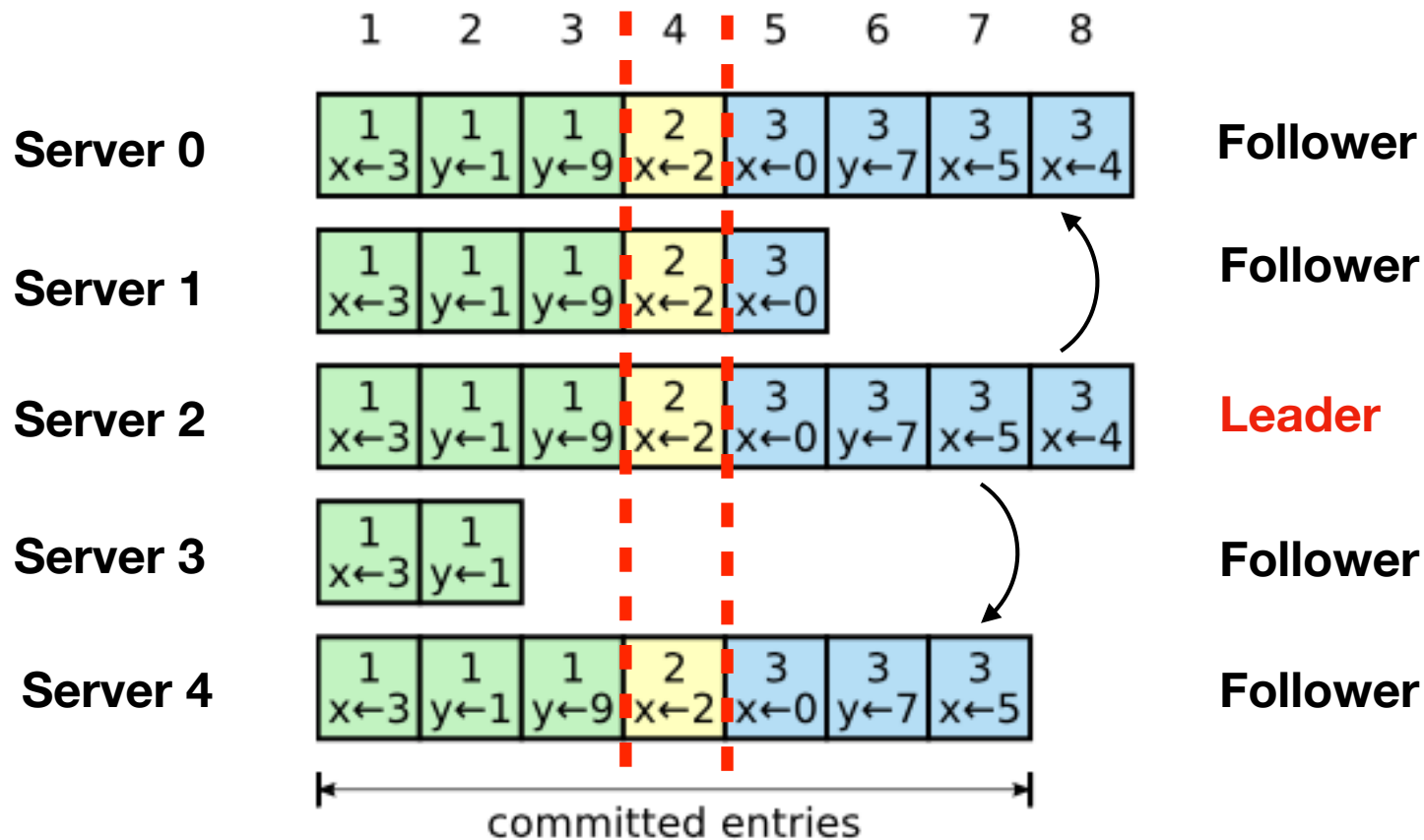
- Log entries are committed by consensus



- If committed, then known to be replicated on a majority

There Can Be Only One

- All actions are coordinated by one and only one leader



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

Complications

- 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.

The Plan

- I will cover some foundational topics
 - Networks, concurrency, state machines
 - There are a few exercises
- There will be a lot of open-coding (work on Raft)
- Key to success: TAKE. IT. SLOW.
 - Read/study the problem.
 - Think about testing/validation more than coding

Part 1

Foundations

Message Passing



- Machines send and receive messages
- Primary issues
 - What is a message?
 - How is it transported?

What is a Message?

- Usually a size-prefixed byte vector



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

Message 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))
```

Message Transport

- Receiving data on a socket

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

- Sending data on a socket

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

```
# Alternative
sock.sendall(data)
```

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

Exercise

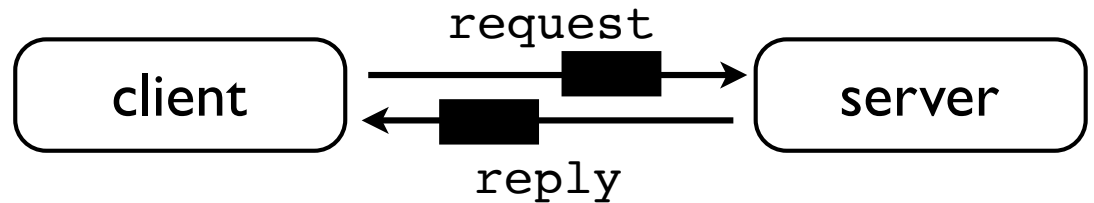
- Create a higher level message channel class

```
ch = Channel(sock)
ch.send(msg)           # Send a message
ch.recv()              # Receive a message
```

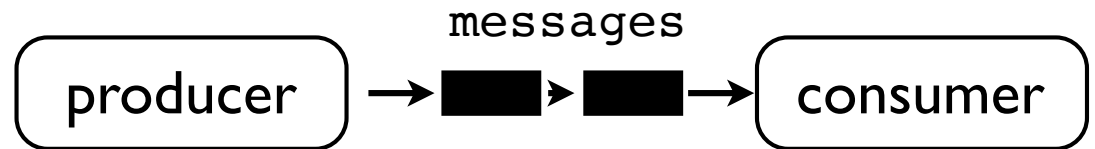
- Have it wrap around an existing socket
- Give it send()/recv() that operate on messages
- Test it by writing a simple echo service.
- KEEP.IT.SIMPLE.

Messaging Patterns

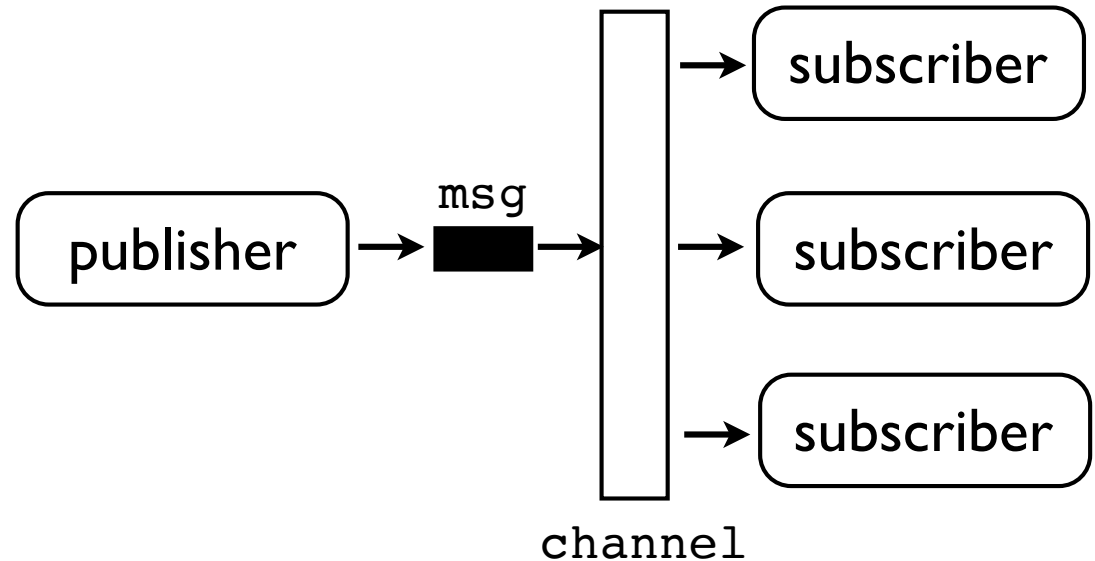
- Request/Reply



- Queue



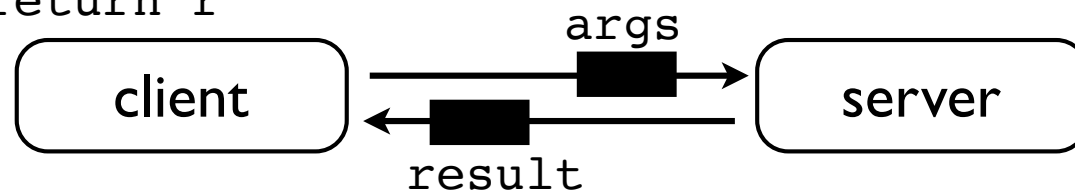
- Publish/Subscribe



Remote Procedure Call

```
# proxy  
func(args):  
    send(args)  
    r = recv()  
    return r
```

```
# implementation  
func(args):  
    ...  
    return result
```



- Message passing, but function args/result
- Client accesses via proxy function
- A layer on top of request/reply messages

Exercise

- Implement an RPC key-value store service

```
data = { }
```

```
def get(key):  
    return data[key]
```

```
def set(key, value):  
    data[key] = value
```

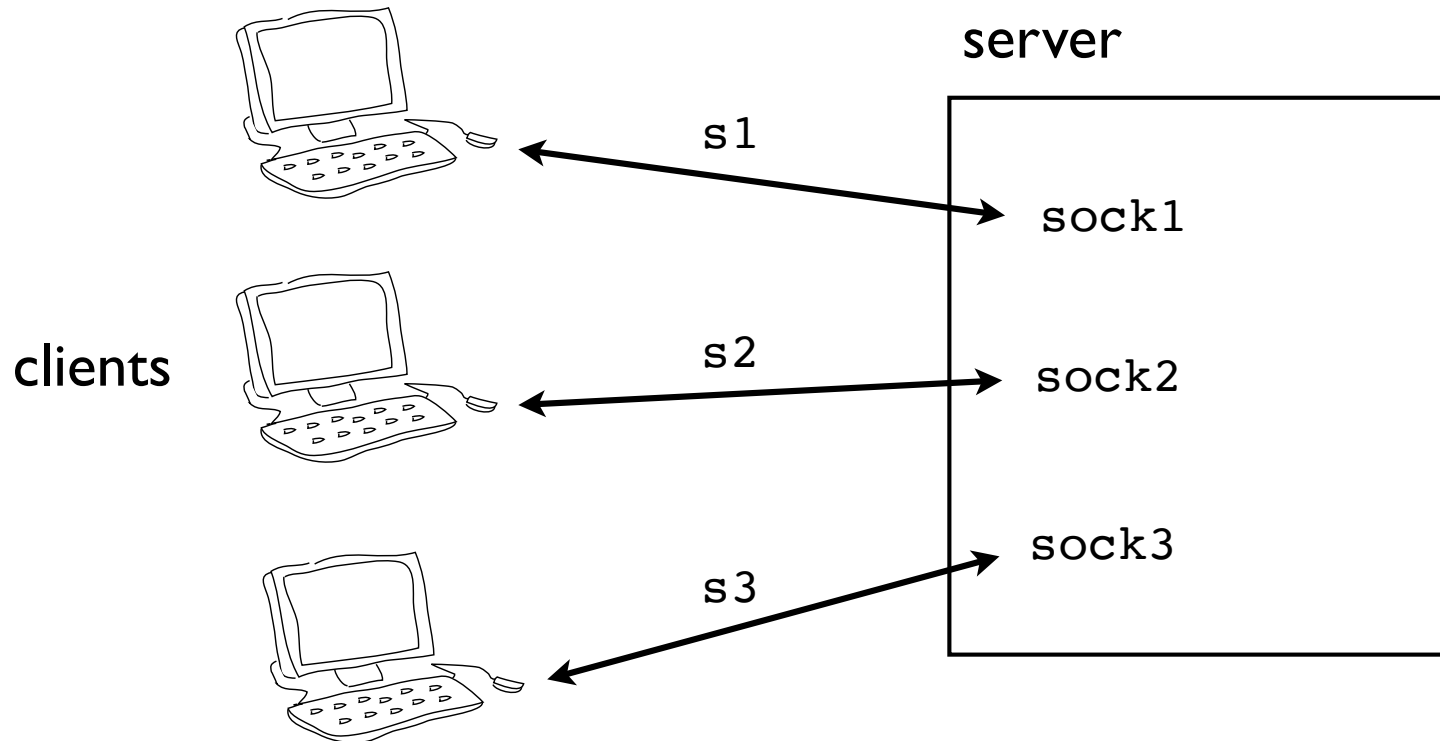
- Server should expose only those functions
- Create a client object that allows access

```
db = KVStore(address)  
db.set(key, value)  
value = db.get(key)
```

- **KEEP. IT. SIMPLE.**

Concurrency

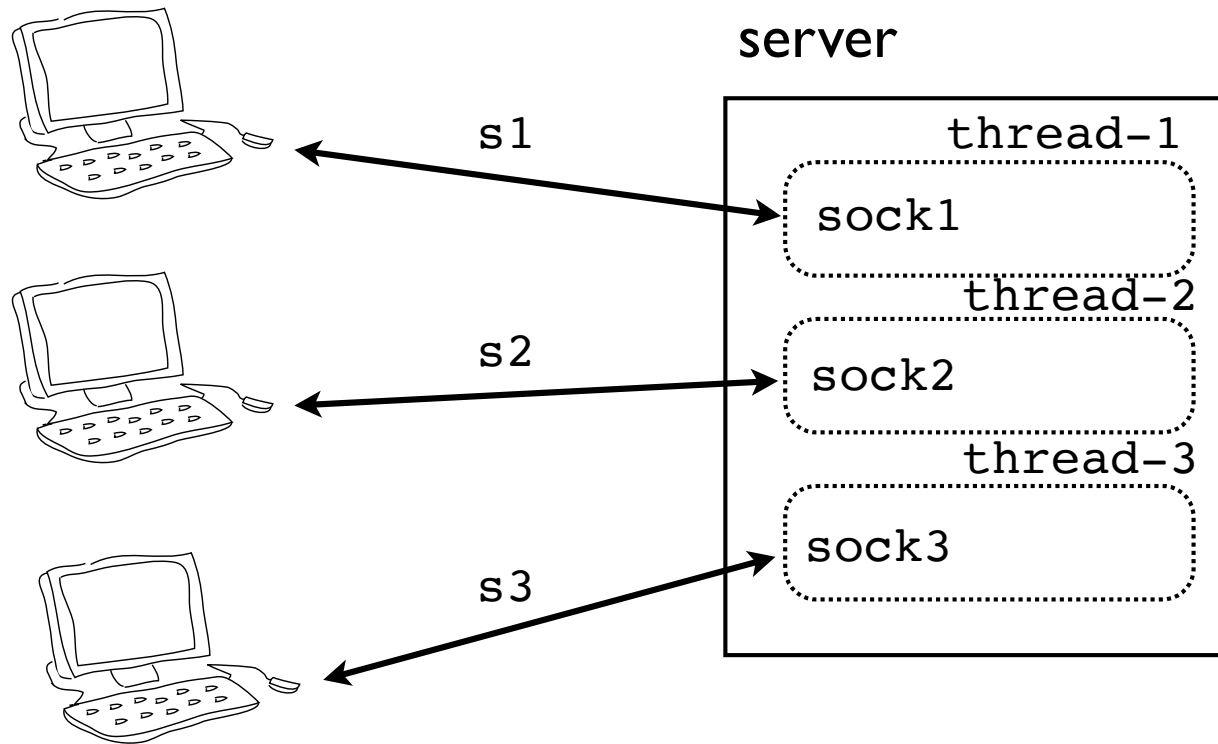
- In a distributed world, servers want to interact with multiple clients at once



- How to coordinate execution on server?

Multiple Clients

- A solution: handle each client in a thread



- Independent handling of each client

Thread Basics

% **python server.py**

↓
statement
statement
...

↓
"main thread"

Program launch.
Program starts
executing statements

Thread Basics

% python server.py



statement

statement

...



create thread(client) ➔ **def client():**

Creation of a thread.
Launches a callable.

Thread Basics

`% python server.py`

↓
statement
statement

...



create thread(client) → `def client():`

↓
statement
statement

...



Concurrent
execution
of statements

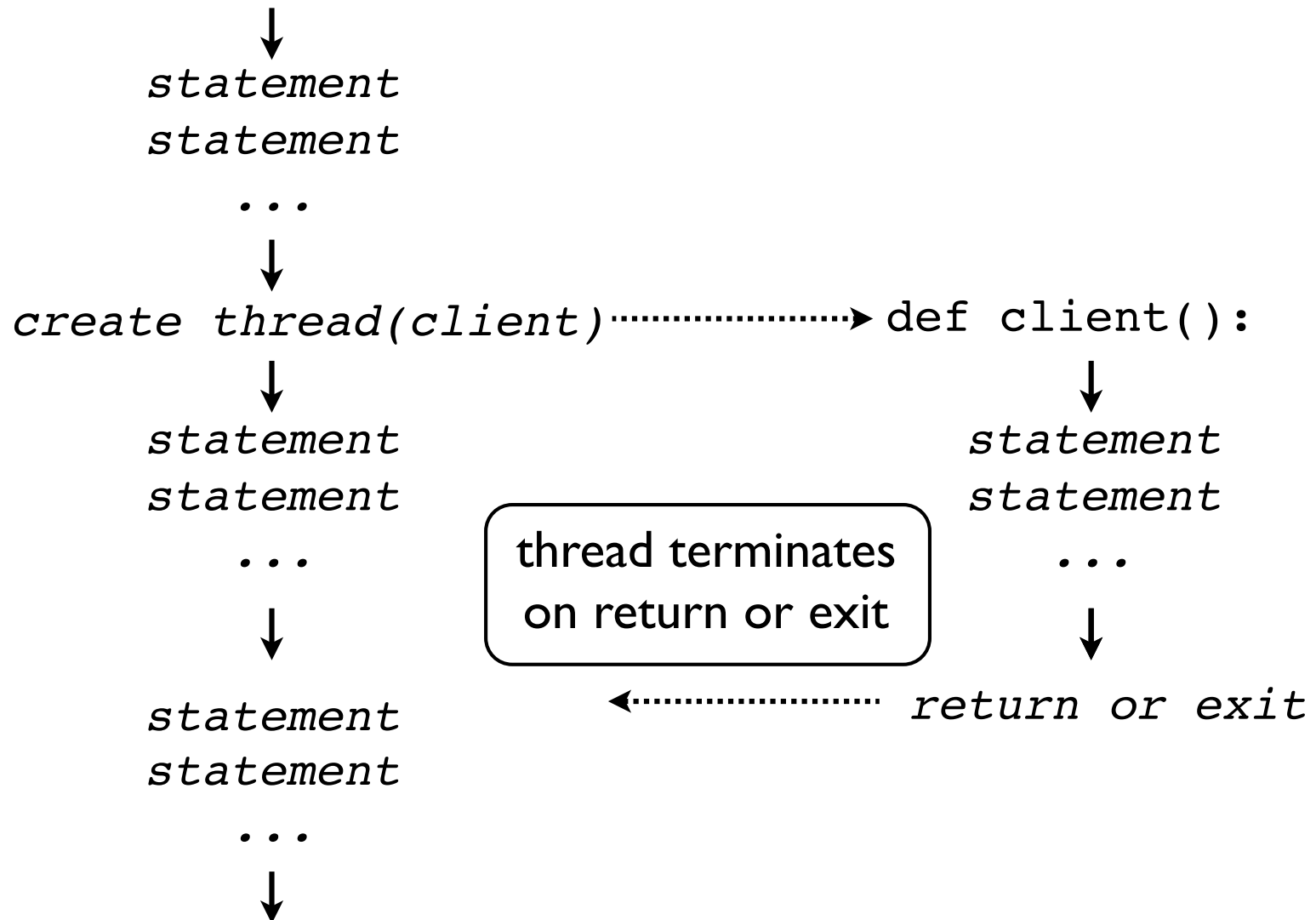
↓
statement
statement

...



Thread Basics

`% python server.py`



Thread Basics

% **python server.py**

↓
statement
statement

...



create thread(client)

↓
statement
statement

...



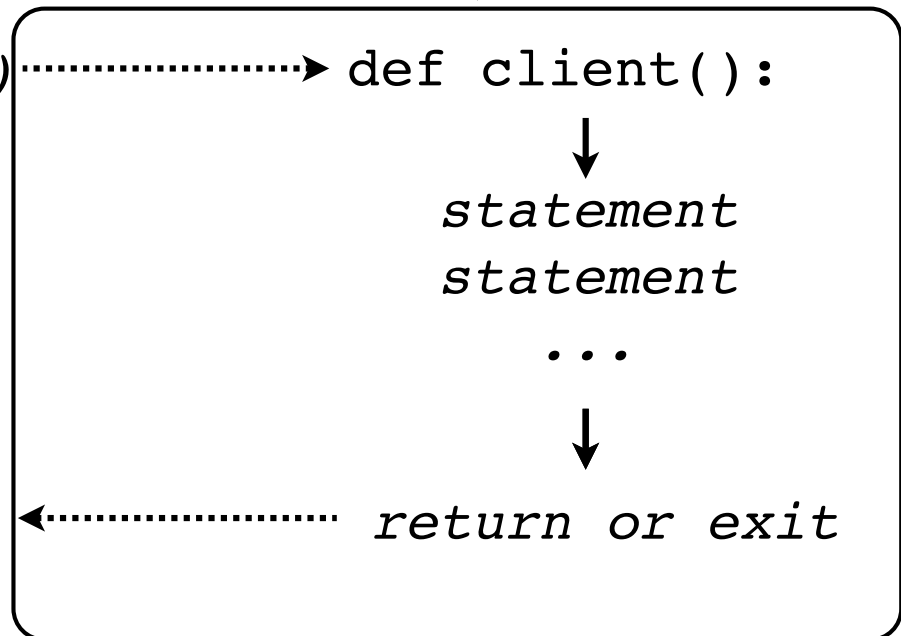
statement
statement

...



Key idea: Thread is like a little "task" that independently runs inside your program

thread



threading Module

- How to launch threads in a server

```
from socket import socket, AF_INET, SOCK_STREAM
import threading

def server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(1)
    while True:
        client, addr = sock.accept()
        t = threading.Thread(target=handle_client,
                              args=(client, addr))
        t.start()

def handle_client(client_sock, addr):
    print("Connection from:", addr)
    with client_sock:
        ...
```

Exercise

- Make your key-value store server support concurrent client connections
- Allow clients to have persistent connections (each client can keep their connection open)
- Handle each client in a server thread
- KEEP. IT. SIMPLE.

Shared Memory

- Threads share data

```
data = {}                # A global variable
```

```
def get(key):
```

```
    ...
```

```
    return data[key]
```

```
    ...
```

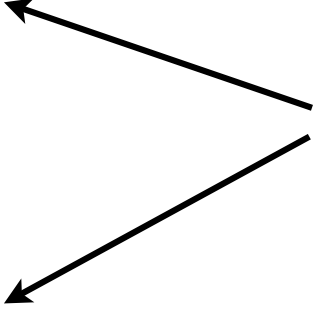
```
def set(key, val):
```

```
    ...
```

```
    data[key] = val
```

```
    ...
```

These operations both
manipulate the global
variable "data"



- If multiple threads, it's possible that both operations are executed simultaneously

Nondeterminism

- Thread execution is non-deterministic
- Operations that take several steps might be interrupted mid-stream (non-atomic)
- Concurrent access to shared data structures becomes non-deterministic (which is a really good way to have your head explode)

Execution Order

- Consider shared state

```
data = { }
```

- One thread sets the value, another reads it

```
Thread-1
```

```
-----
```

```
...
```

```
return data[ 'x' ]
```

```
...
```

```
Thread-2
```

```
-----
```

```
...
```

```
data[ 'x' ] = value
```

```
...
```

- Problem : Which thread runs first?
- Answer : It could be either one...

Concurrent Updates

- Consider a shared value

$x = 0$

- What if there are concurrent updates?

Thread-1

...

$x = x + 1$

...

Thread-2

...

$x = x - 1$

...

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

Concurrent Updates

- The two threads

Thread-1

...

$x = x + 1$

...

Thread-2

...

$x = x - 1$

...

- Low level code execution

Thread-1



LOAD_GLOBAL 1 (x)

LOAD_CONST 2 (1)

BINARY_ADD

STORE_GLOBAL 1 (x)

Thread-2

→
thread
switch

LOAD_GLOBAL 1 (x)

LOAD_CONST 2 (1)

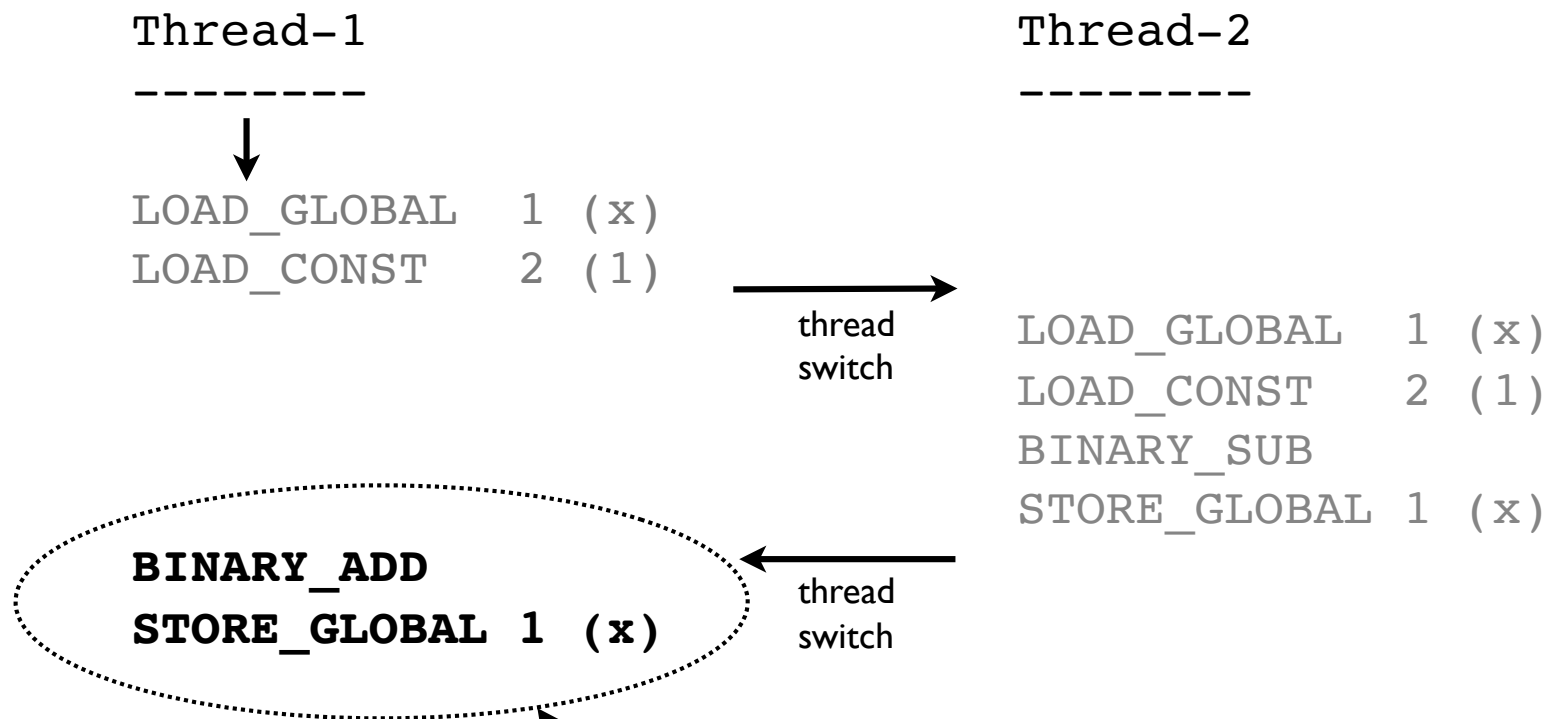
BINARY_SUB

STORE_GLOBAL 1 (x)

←
thread
switch

Concurrent Updates

- Low level interpreter code



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

If there's one lesson we've learned from 30+ years of concurrent programming, it is: just don't share state. It's like two drunkards trying to share a beer. It doesn't matter if they're good buddies. Sooner or later, they're going to get into a fight. And the more drunkards you add to the table, the more they fight each other over the beer. The tragic majority of MT applications look like drunken bar fights.

-ZeroMQ Manual

Thread Synchronization

- Execution can be coordinated through some classic synchronization primitives
 - Events
 - Mutexes
- There are others (e.g., Semaphores)

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

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)

Python Note

- Prefer the use of context managers

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

```
Thread-1
```

```
-----
```

```
...
```

```
with x_lock:
    x = x + 1
```

```
...
```

- If using another language, check for the proper idiom on using a lock

Exercise

- Modify the Key-Value server to write a transaction log that records all data updates

```
data = { }
```

```
def set(key, value):  
    data[key] = value  
    write_log('set', key, value)
```

- On server startup, replay the transaction log
- Server should be able to get back to last state

Thinking....

- Fault tolerance study... how can this code fail?

```
data = { }
```

```
def get(key):  
    return data[key]
```

```
def set(key, value):  
    data[key] = value  
    write_log('set', key, value)
```

- What bad things could happen on the server?

Exercise

- Modify the server to enforce these rules:
 - Unlimited concurrent gets
 - No concurrent get/set (getters must block if a set is in progress)
 - No concurrent sets (one setter at a time)
- Note: This is reader/writer locking
- KEEP. IT. SIMPLE.

Project

- Finish coding a "reliable" single-machine key-value server
 - All updates must be logged
 - Restartable (if killed, the log is replayed to populate the key-value store on restart)
 - No concurrent get/set (or sets).
- KEEP. IT. SIMPLE.

Extended Project

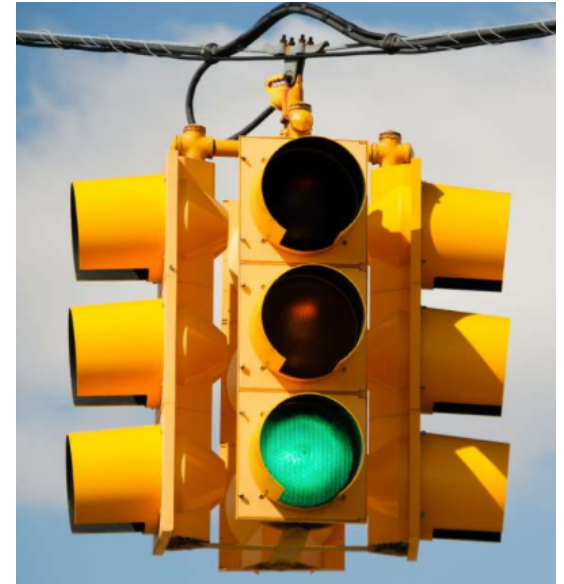
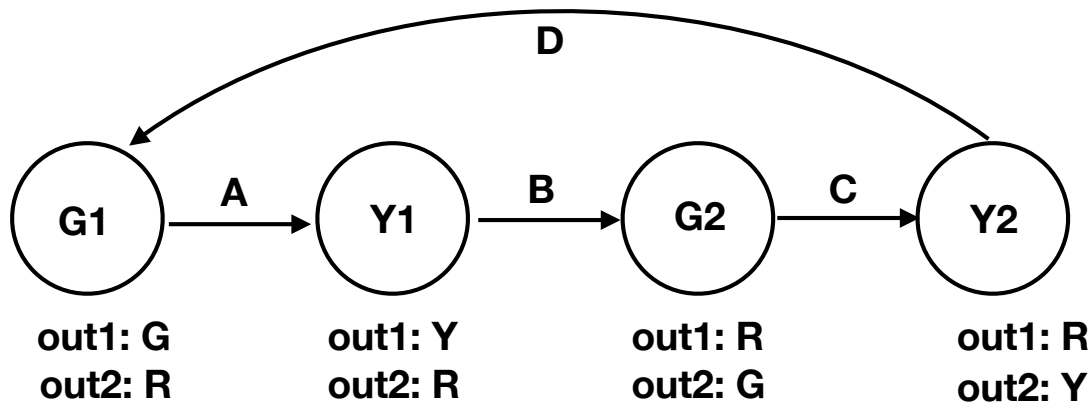
- Work on message passing constructs for Raft
- Read pg. 4 of the Raft paper.
- Think about data structures for representing messages and the encoding
- Think about how messages might be sent through the earlier Channel class.

Part 2

State Machines

Idea: Operational State

- Example: Traffic Light



- Circles represent "state"
- Arrows represent "events" (cause state change)

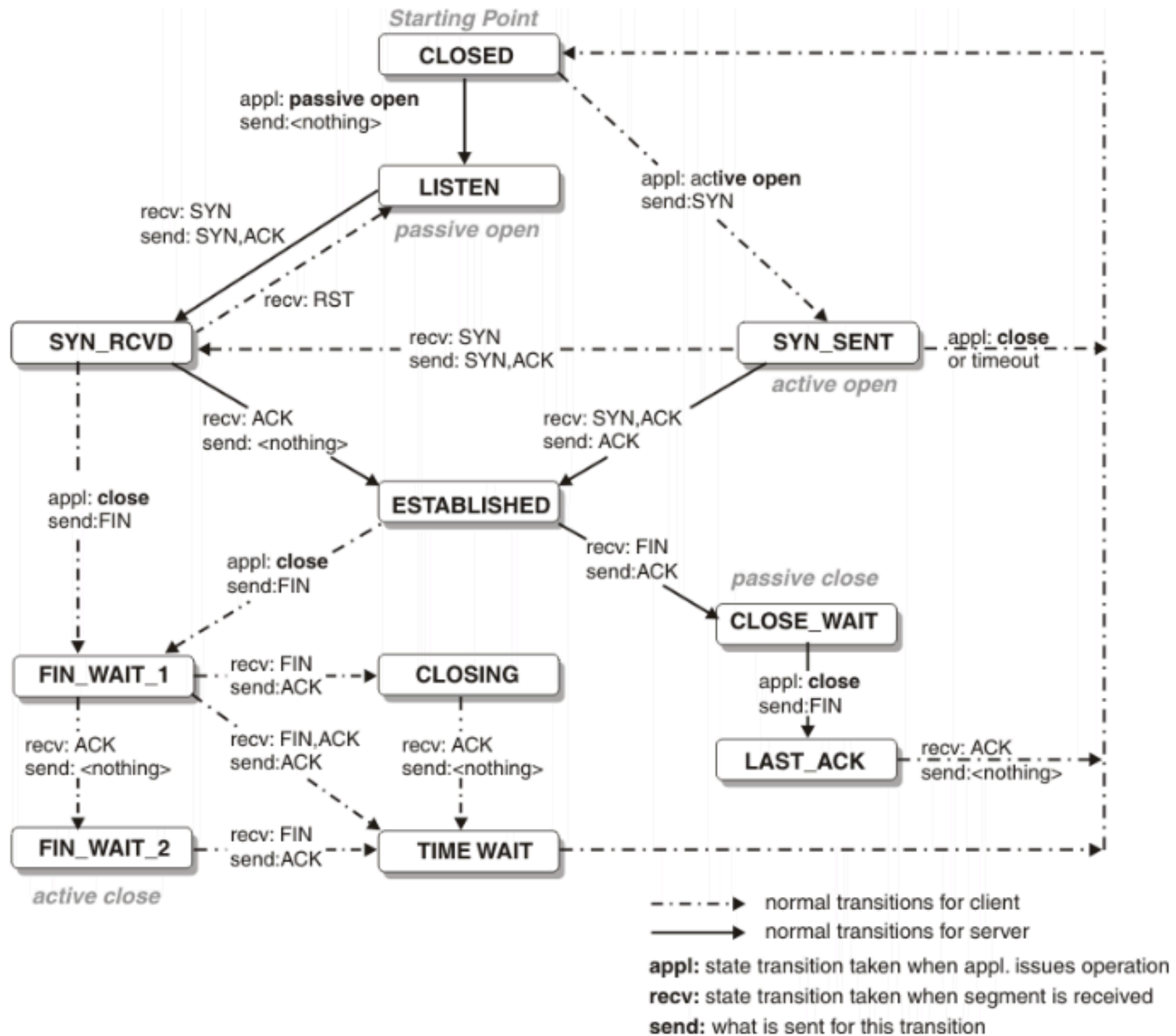
A: 30s timer

B: 5s timer

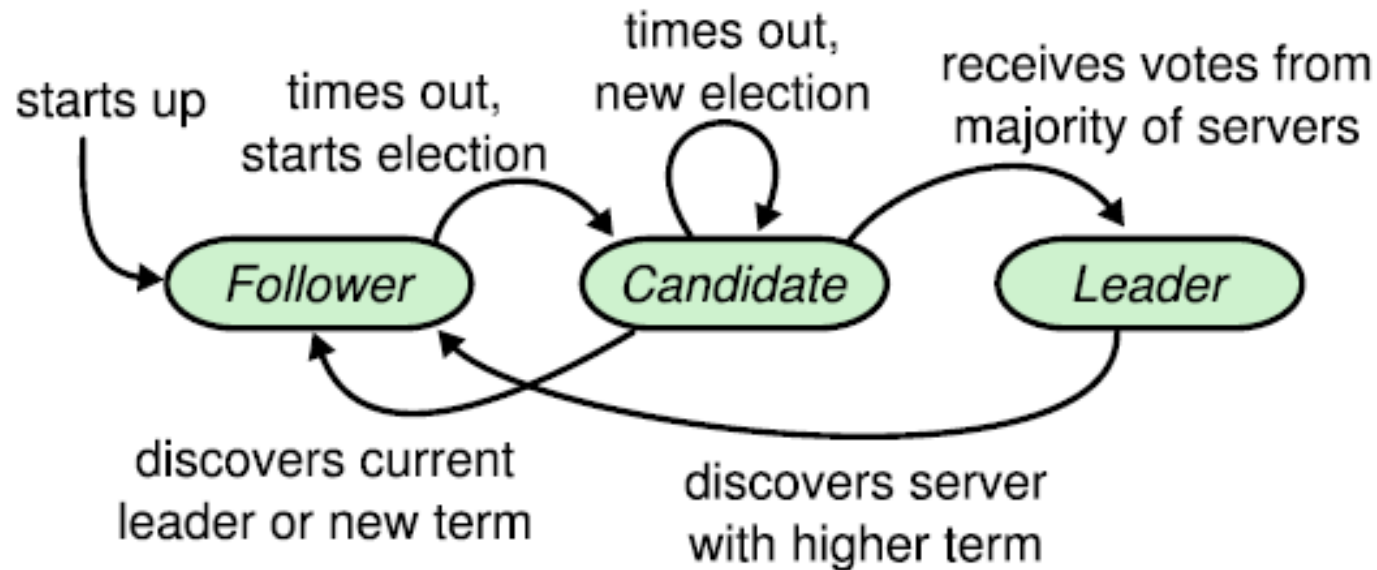
C: 60s timer or (walk_button and 30s timer)

D: 5s timer

Figure 1. TCP state transition diagram

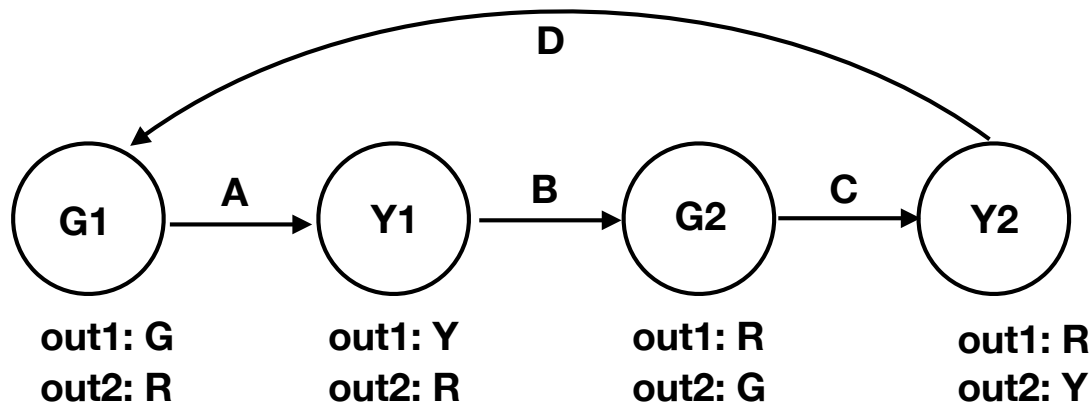


Raft Operational States



Exercise: Traffic Light

- Implement this state machine



A: 30 seconds

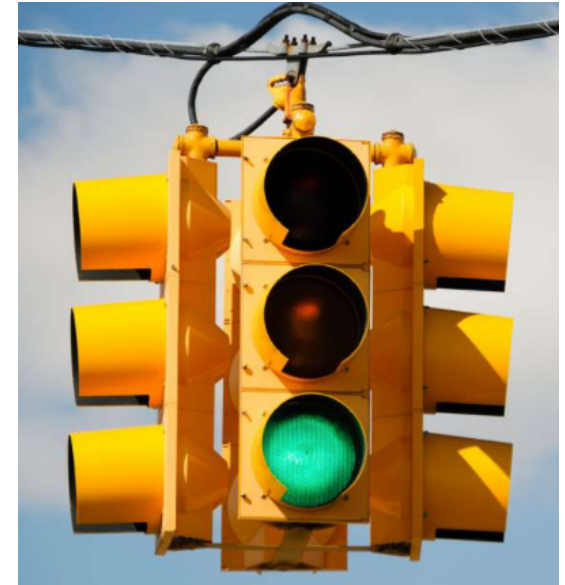
B: 5 seconds

C: 60 seconds or (>30 seconds and walk_button)

D: 5 seconds

Note: Pressing walk button causes signal change if G2 state has been displayed for more than 30 seconds.

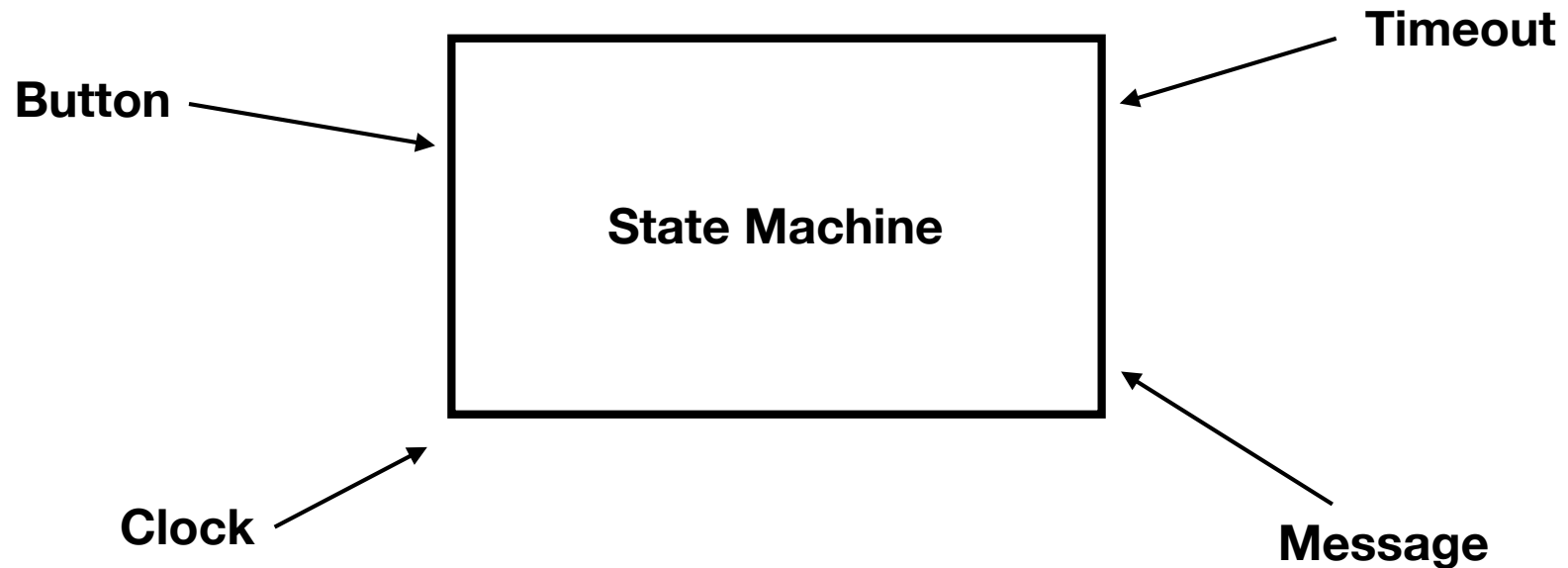
- Use any technique you know



Issue: Time and Events

- State machines operate in response to events
 - Timers
 - Timeouts (lack of events)
 - Buttons
- Usually asynchronous and concurrent
 - What is software architecture for it?

Issue: Time and Events

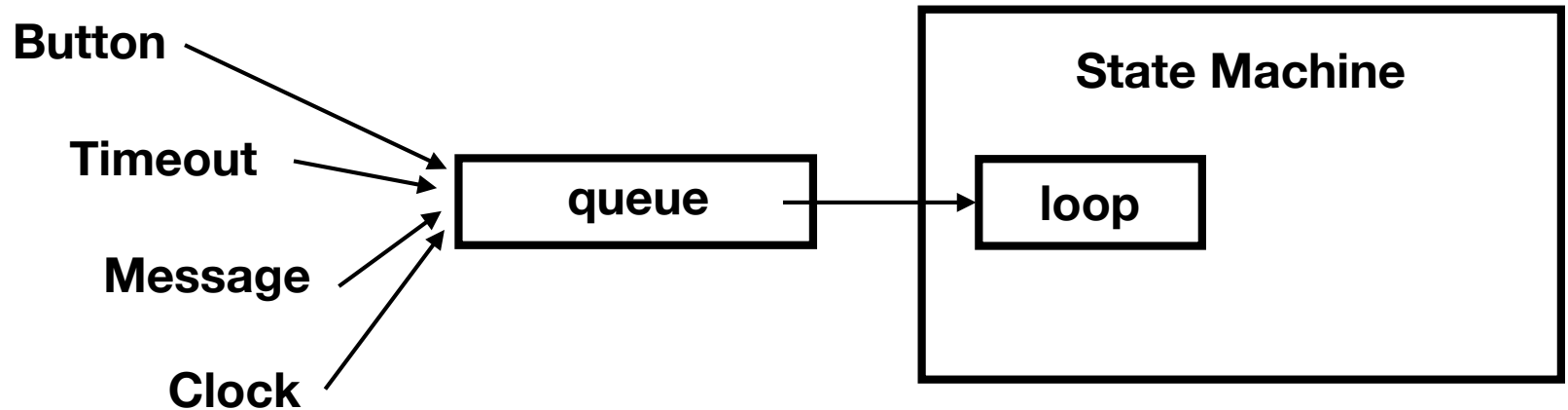


- One option: Callback functions/handlers

```
machine.handle_button()  
machine.handle_clock()  
machine.handle_message(msg)  
machine.handle_timeout()
```

- Trigger the appropriate handler on event

Event Serialization



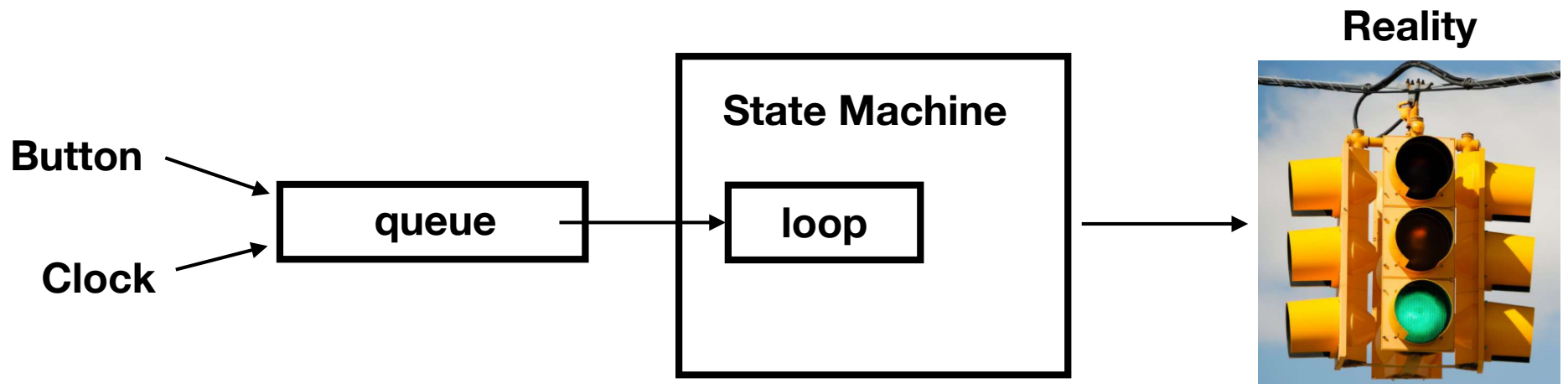
- Concurrent event handling often tricky
- Can serialize events onto a queue
- Process one at a time with an event loop

Exercise

- Keep working on traffic light code
- Use a queue to serialize external events
- Try to isolate the machine from runtime details
- Implement some kind of controller that can run the machine in real-time
- Read button-press from keyword (Return/Enter)

Problem: Actions

- How to translate the state of a state machine into concrete actions?



- Need to observe or subscribe to state changes
- Must execute code in response

Exercise

- Modify traffic light machine to allow observers or actions to be registered with it
- Write code that monitors the light and stores the current light state in the key-value server you created earlier!
- Because, why not?
- (Thought: hope nothing "bad" happens to server)

Formalizing State Machines

- How to specify state
- How to specify state changes
- How to implement state machines
- How to test state machines
- How to prove state machines

Representing Values

- Variables

```
out1 = "G"  
clock = 23
```

- Data structures (tuples, sets, records, etc.)

```
lights = ( "G", "R" )
```

- These represent storage (storing a value)

Representing States

- A "state" refers to a collection of values

```
{  
  out1 = 'G',  
  out2 = 'R',  
  clock = 23,  
}
```

```
{  
  out1 = 'R',  
  out2 = 'G',  
  clock = 15,  
}
```

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

State Membership

```
{  
  out1 = 'G',  
  out2 = 'R',  
  clock = 23,  
}
```

```
{  
  out1 = 'R',  
  out2 = 'G',  
  clock = 15,  
}
```

- State membership is expressible via relations

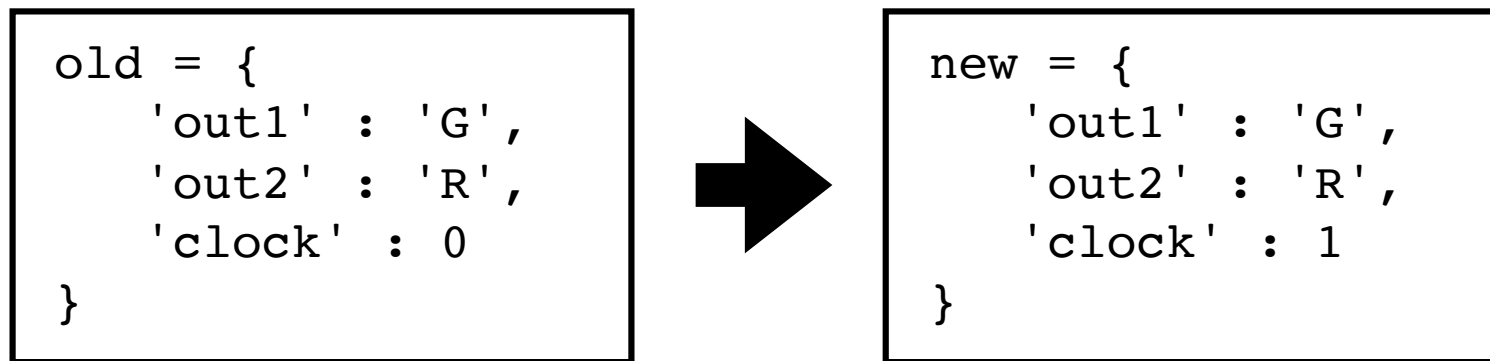
G1: out1 == 'G' and
out2 == 'R' and
clock < 30

G2: out1 == 'R' and
out2 == 'G' and
clock < 60

- Should be unambiguous (probably)

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

External Events

- When do state machines actually run?
- In response to events!
 - Clock tick
 - Button press
- When do these happen? We don't know!
- What we do know: what happens afterwards.

Safety of State Machines

- There may be invariant conditions

```
assert not (s['out1'] in {'G', 'Y'} and  
            s['out1'] == s['out2'])
```

- Deadlock (when there's no next state)
- Stuttering (state unchanged on event)
- Question: How can you test these things?
- Short answer: Have to test every possible combination of states and events!

Group Exercise

- Challenge: Can you define a more mathematical foundation for state machines?
- Based on logic, functions, and basic ideas.
- We're going to code this together (with guidance)
- Hold on...

TLA+

- A tool for modeling/verifying state machines
- It is based on a 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

TLA+ Definitions

- Values are defined 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
```

TLA+ Variables

- State is held in designated variables

```
VARIABLE out1  
VARIABLE out2  
VARIABLE clock
```

```
vars == <<out1, out2, clock>>
```

- Variables are initialized in a special definition

```
Init == /\ out1 = "G"  
        /\ out2 = "R"  
        /\ clock = 0
```


Sets

- There are 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 }
```

Tuples and Structures

- Tuples (1-indexed)

```
t == <<"a", "b", "c">>
t[1] \* -> "a"
```

- A structure

```
lights = [out1 |-> "G", out2 |-> "R"]
```

- Can access via (.)

```
lights.out1 \* -> "G"
lights.out2 \* -> "R"
```

- Updates (creates a new structure)

```
[ t EXCEPT ![2]="x" ] \* <<"a", "x", "c">>
[ lights EXCEPT !.out1="Y" ] \* [out1|->"Y", out2|->"R"]
```

Boolean Logic

- AND, OR, NOT operators

T /\ F	* AND (\wedge)
T \/ F	* OR (\vee)
~T	* NOT (\neg)

- Grouping by indentation (implies parens)

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

- Same as

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

State Membership

- Expressed as relations

```
G1 == /\ out1 = "G"  
      /\ out2 = "R"  
      /\ clock < 30
```

```
G2 == /\ out1 = "R"  
      /\ out2 = "G"  
      /\ clock < 60
```

```
Y1 == /\ out1 = "Y"  
      /\ out2 = "R"  
      /\ clock < 5
```

- Think booleans (True/False check for a given state)

State Changes

- Variable changes: $\text{var}' = \text{var}$

```
G1 == /\ out1 = "G"  
      /\ out2 = "R"  
      /\ \/ clock < 30 /\ clock' = clock + 1  
                        /\ UNCHANGED <<out1, out2>>  
      \/ clock = 30 /\ clock' = 0  
                        /\ out1' = "Y"  
                        /\ UNCHANGED <<out2>>
```

- Ticked (') variables represent value in next state
- All variables must be explicitly accounted for in a state change

Working with Multiples

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

```
\* Five traffic lights  
nlights == 1..5
```

```
out1 == [ n \in nlights |-> "G" ] \* <<"G","G","G","G","G">>  
out2 == [ n \in nlights |-> "R" ] \* <<"R","R","R","R","R">>  
clock == [ n \in nlights |-> 0 ] \* <<0,0,0,0,0>>
```

- May see states expressed as operators

```
G1(i) == /\ out1[i] = "G"  
         /\ out2[i] = "R"  
         /\ \/ clock[i] < 30  
           /\ clock' = [clock EXCEPT ![i]=clock[i] + 1]  
           /\ UNCHANGED <<out1, out2>>
```

Events

- 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
- A next state relation lists possibilities

$\text{Next} == G1 \ \backslash / \ Y1 \ \backslash / \ G2 \ \backslash / \ Y2$

- TLA+ explores all possible branches

Events

- You might see a spec like this:

```
Next == \ / ButtonPress
        \ / Timeout
        \ / MessageReceived
```

- You ask: "Which one happens?"
- Answer: "Yes"
- They all happen. TLA+ is a simulator that models the entire state space.
- A TLA+ spec is NOT a runtime implementation.

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?

Project: Implement Raft

How to draw an owl

1.



1. Draw some circles

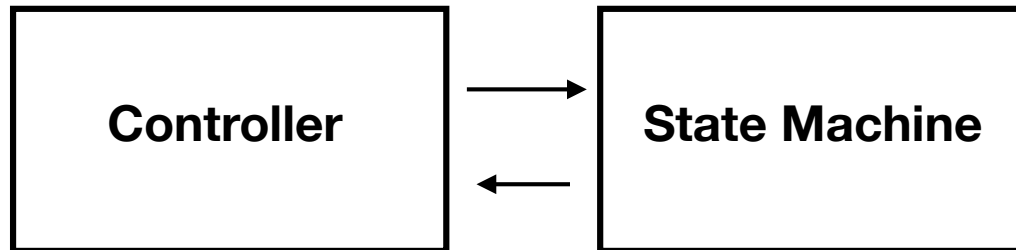
2.



2. Draw the rest of the fucking owl

Suggestions

- Focus your effort on the Raft state machine
- Work on a design that strongly decouples components



- Anything related to time, networks, threads, etc should go in the controller
- Think heavily about testing. TAKE. IT. SLOW.

Tip: Logging

- Raft maintains its own transaction log
 - Must be persistent
 - Must have random access by numeric index
- Sensible options
 - sqlite (database)
 - Store entries as unique files (1, 2, 3, etc.)
- It is okay to use modules such as pickle, json

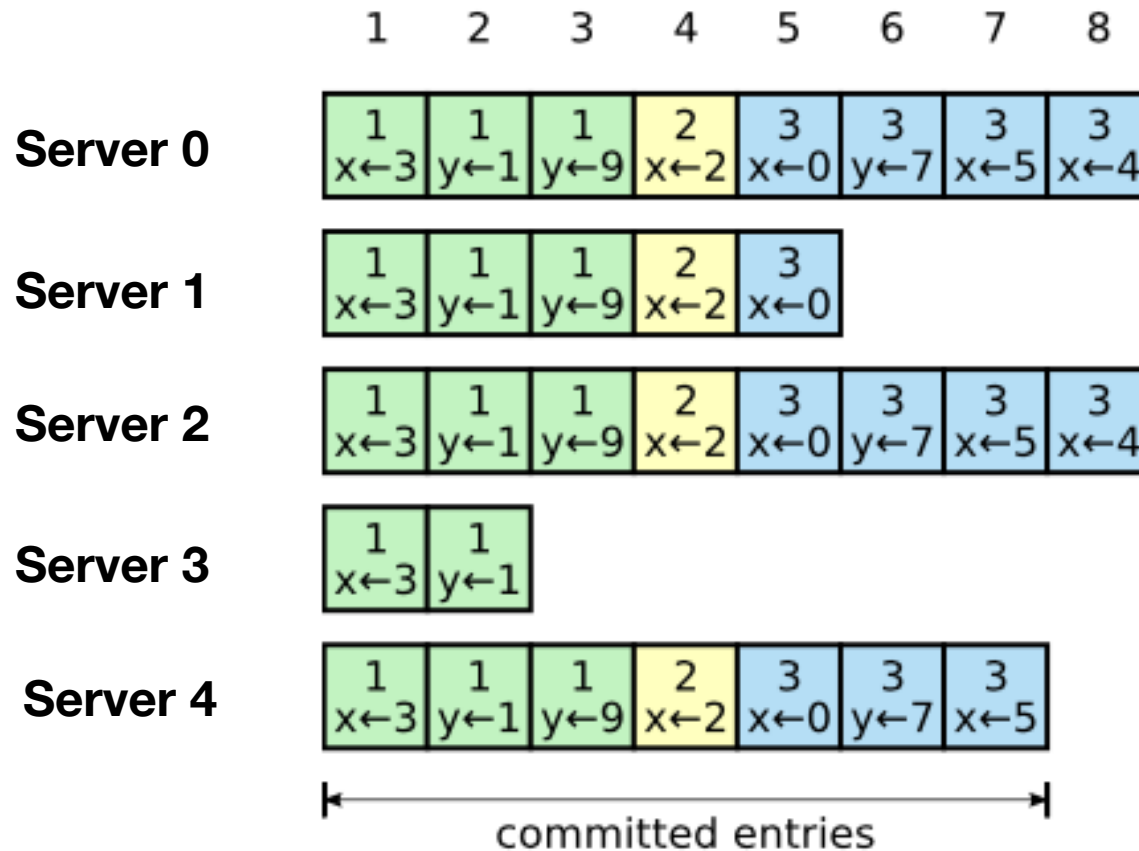
Tip: Messaging

- The servers send messages to each other
- Assume that all messages are one way (do NOT assume a reliable request/reply cycle)
- Plan for message loss (and errors)
- Don't over-design messaging. For example, if an error occurs, just drop the message and ignore.

Tip: Elections

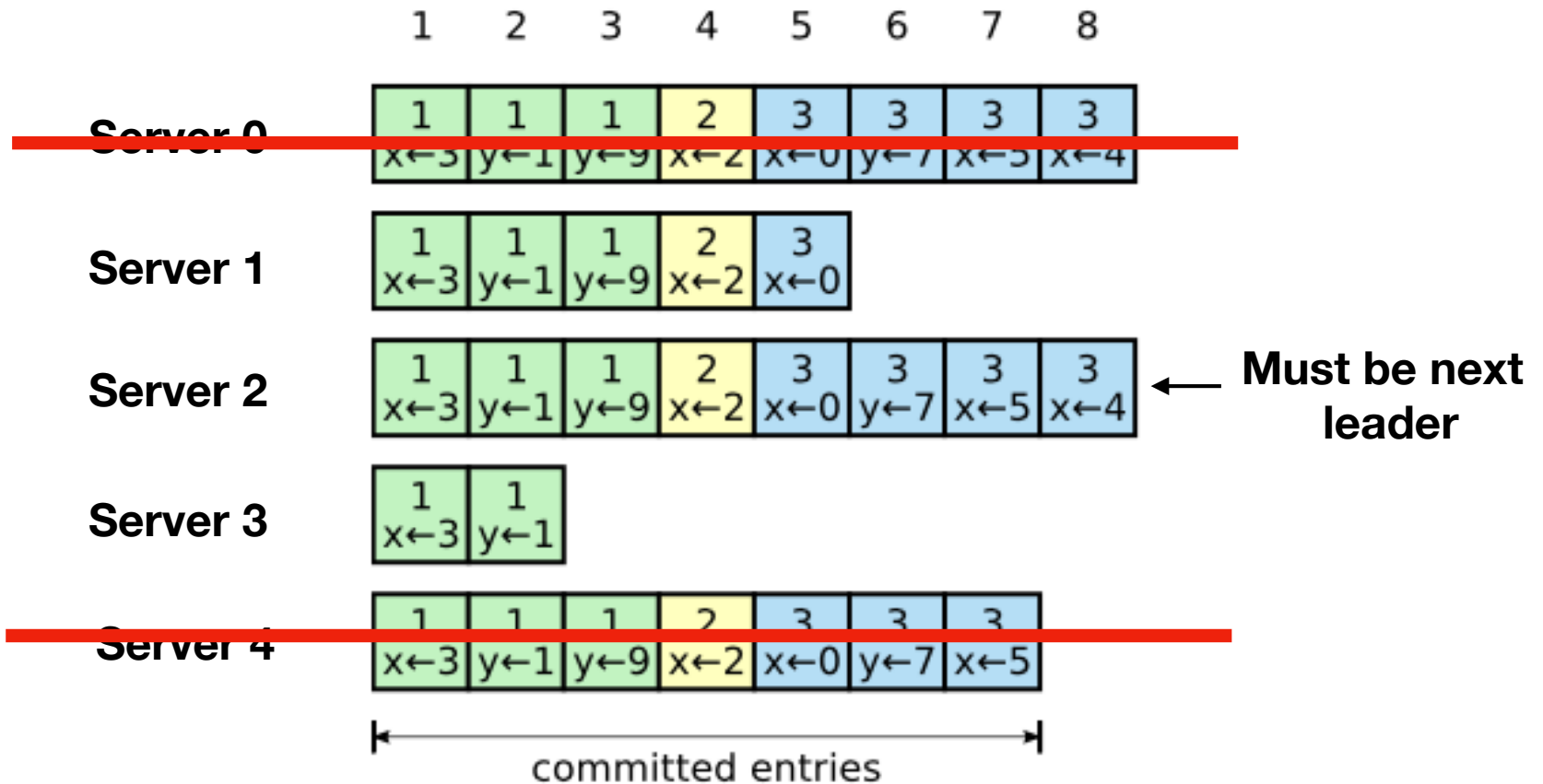
- A major part of Raft concerns the protocol for electing a new leader
- The leader must always have the most up-to-date version of the committed entry log
- This is subtle, but a critical part of the algorithm

Tip: Elections



- Kill any two servers.... pick two
- Which one is the next leader?

Tip: Elections



- Kill any two servers.... pick two
- Which one is the next leader?

Tip: Configuration

- A Raft cluster is typically small (3-7 machines)
- Assume that all participants know the configuration of the entire cluster

```
servers = [  
    ('hostname1', port1),  
    ('hostname2', port2),  
    ('hostname3', port3),  
    ('hostname4', port4),  
    ('hostname5', port5)  
]
```

- Hardcode it in a file someplace
- Can make it more "enterprise grade" later

Tip: Client Connections

- Eventually, clients must connect with the Raft server (e.g., the key-value store proxy)
- Assume clients also know the configuration
- Communication is only allowed with leader
- Use negative-acks to direct clients to the leader (if client connects to a follower, follower should respond with an error indicating "not leader").

Tip: TAKE. IT. SLOW

- Focus more on top-down design and thinking about the problem than low-level implementation details (i.e., sockets)
- Debugging Raft is extremely challenging
- Testing Rocks. Debugging Sucks.