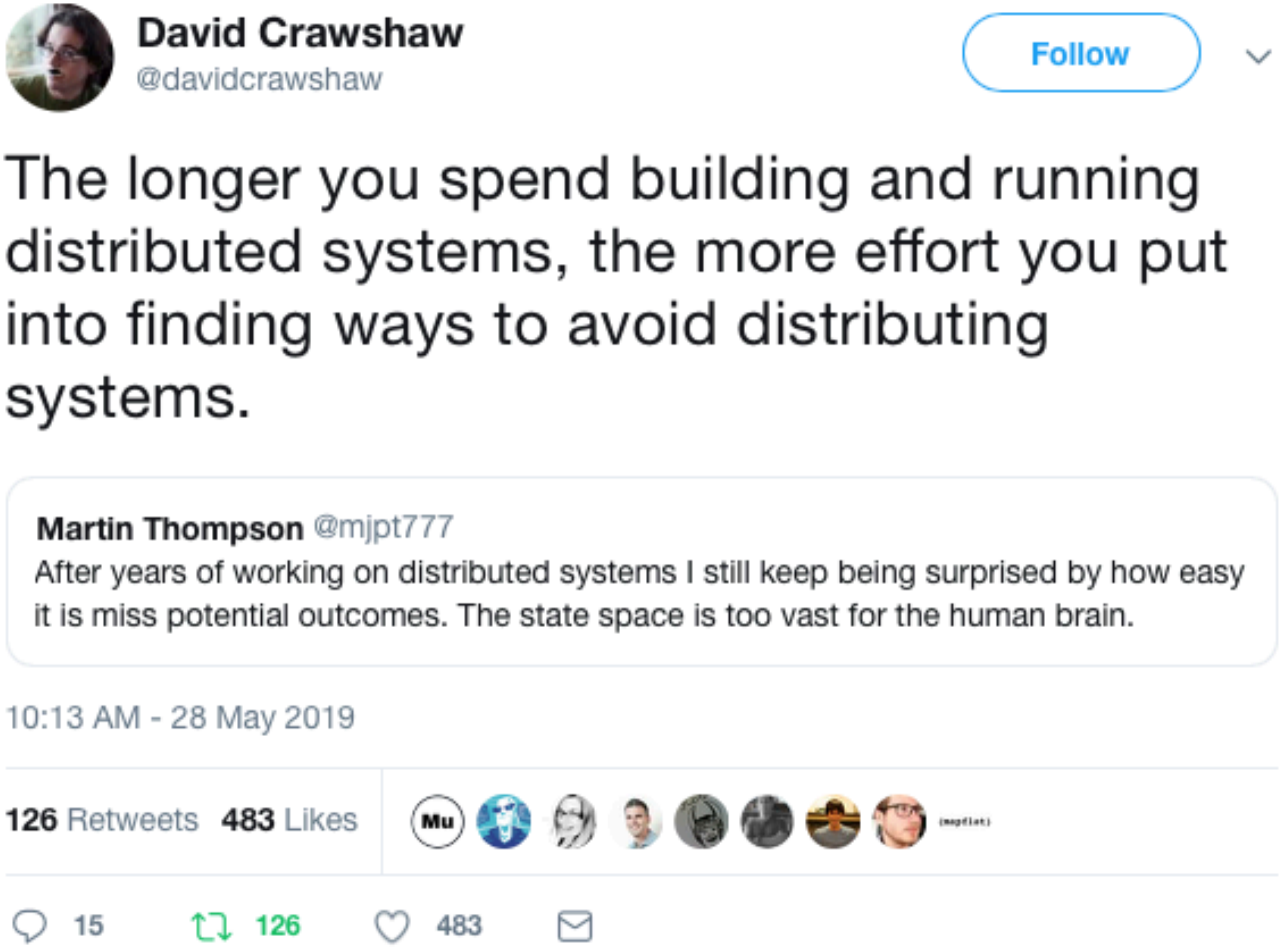
Rafting Trip

(AKA: Distributed Systems)

David Beazley
(@dabeaz)

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

Saw Recently...



David Crawshaw
@davidcrawshaw

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

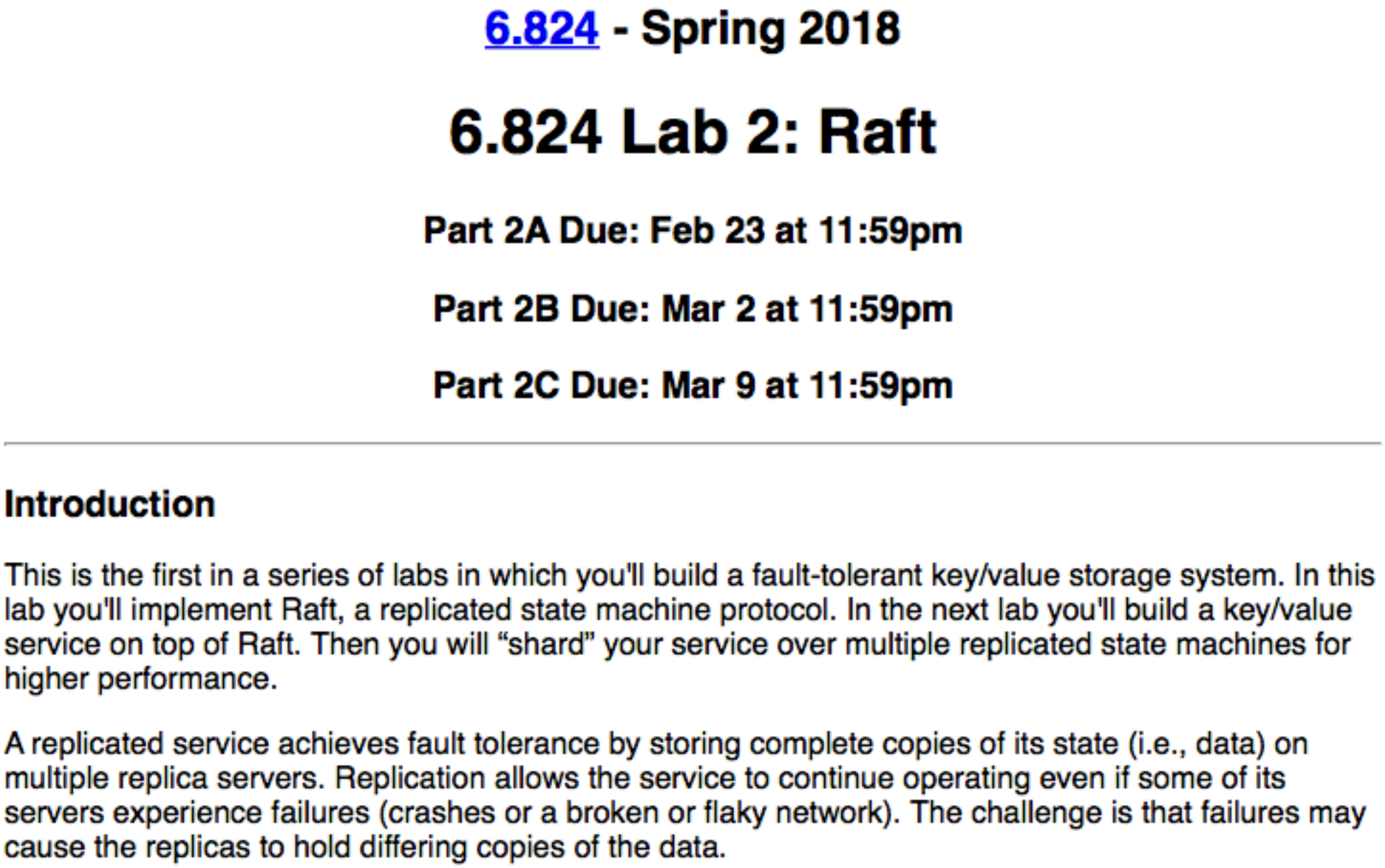
126 Retweets 483 Likes

15 126 483

Copyright (c) 2019, http://www.dabeaz.com

This Week

- We attempt to implement a project from MIT's 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

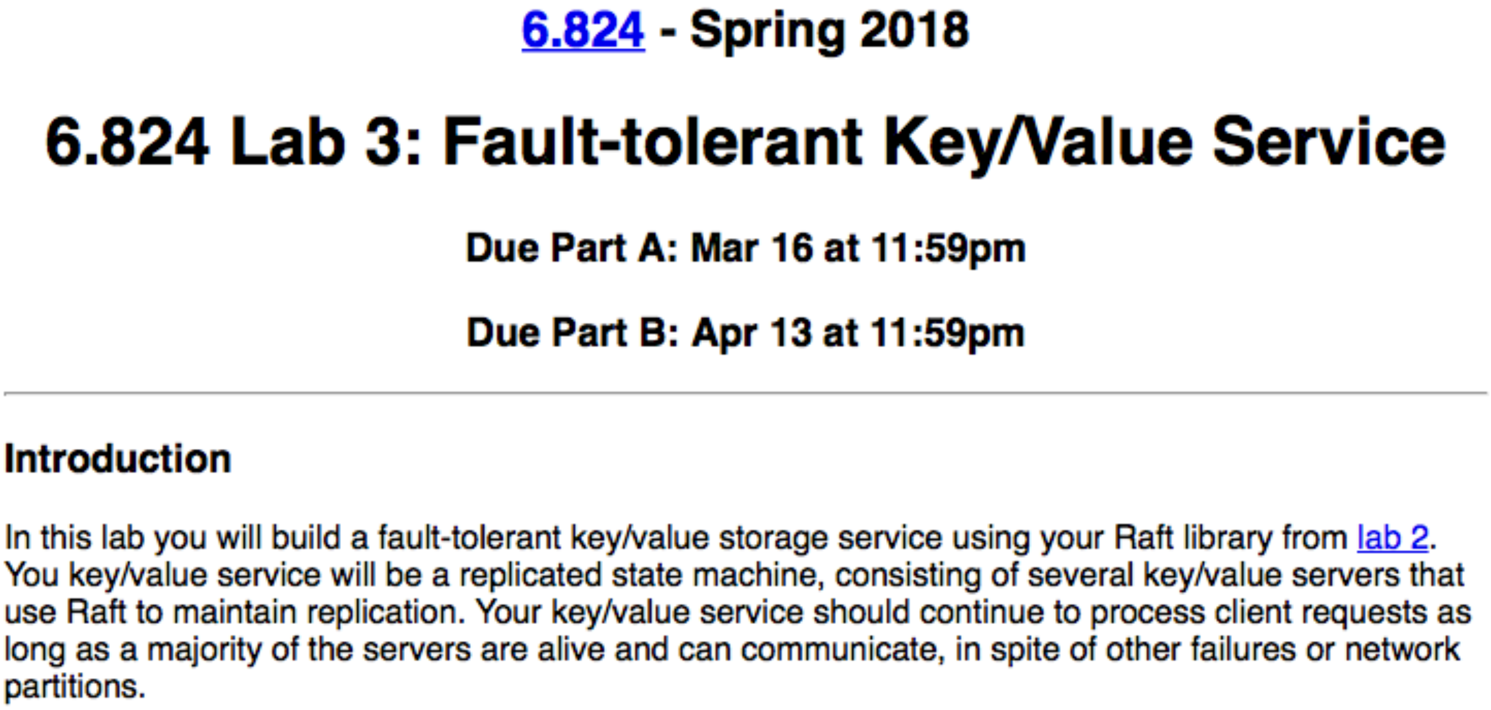
Introduction

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 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

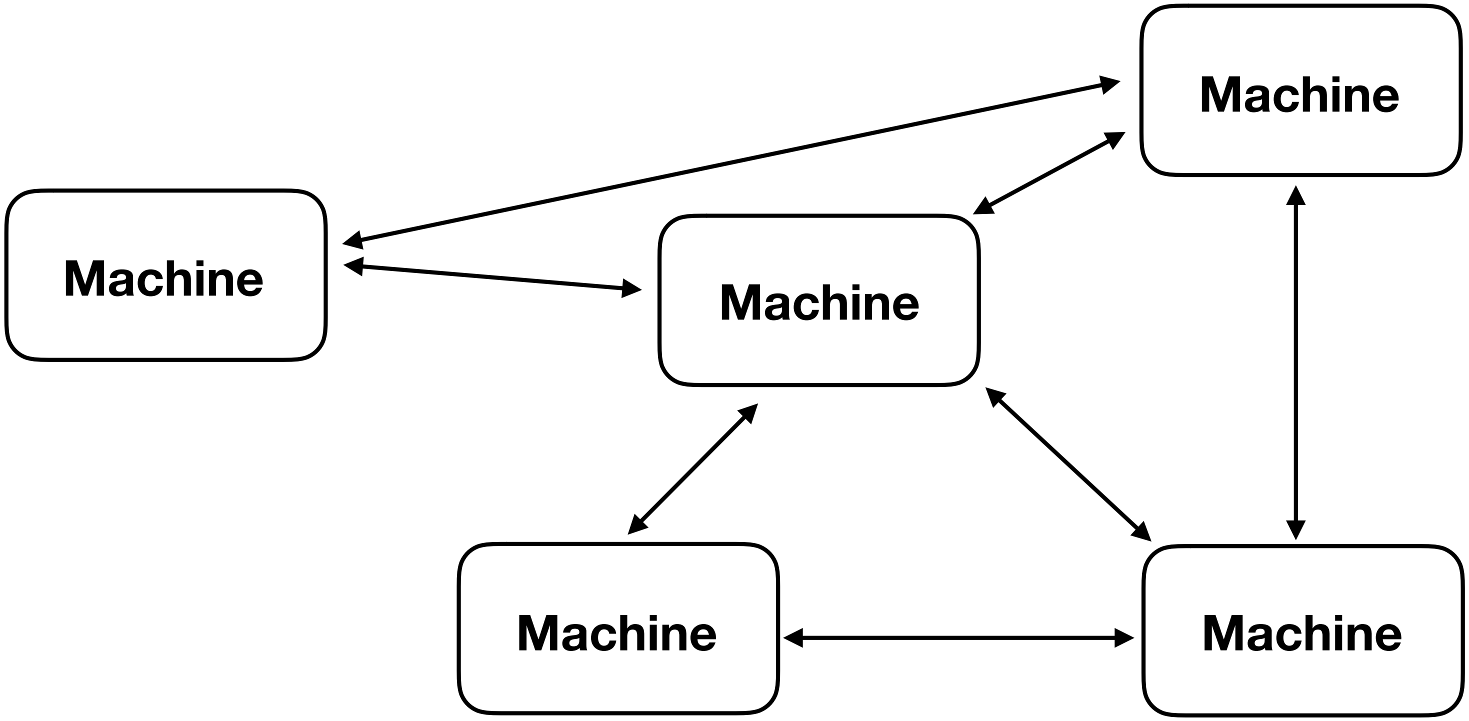
In this lab you will build a fault-tolerant key/value storage service using your Raft library from lab 2. You key/value service will be a replicated state machine, consisting of several key/value servers that use Raft to maintain replication. Your key/value service should continue to process client requests 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!

Copyright (c) 2019, http://www.dabeaz.com

High Level View

- Distributed Computing

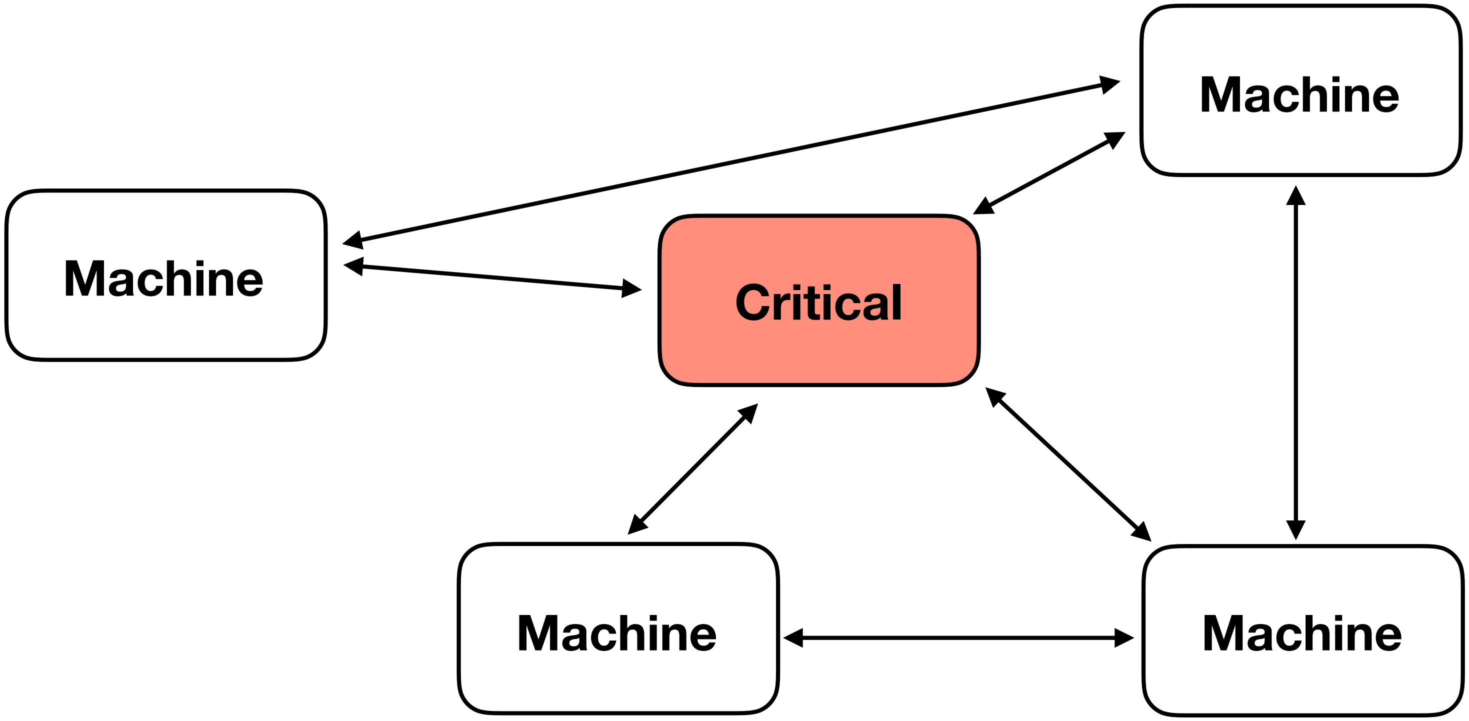


- Machines/services communicating over a network

Copyright (c) 2019, http://www.dabeaz.com

High Level View

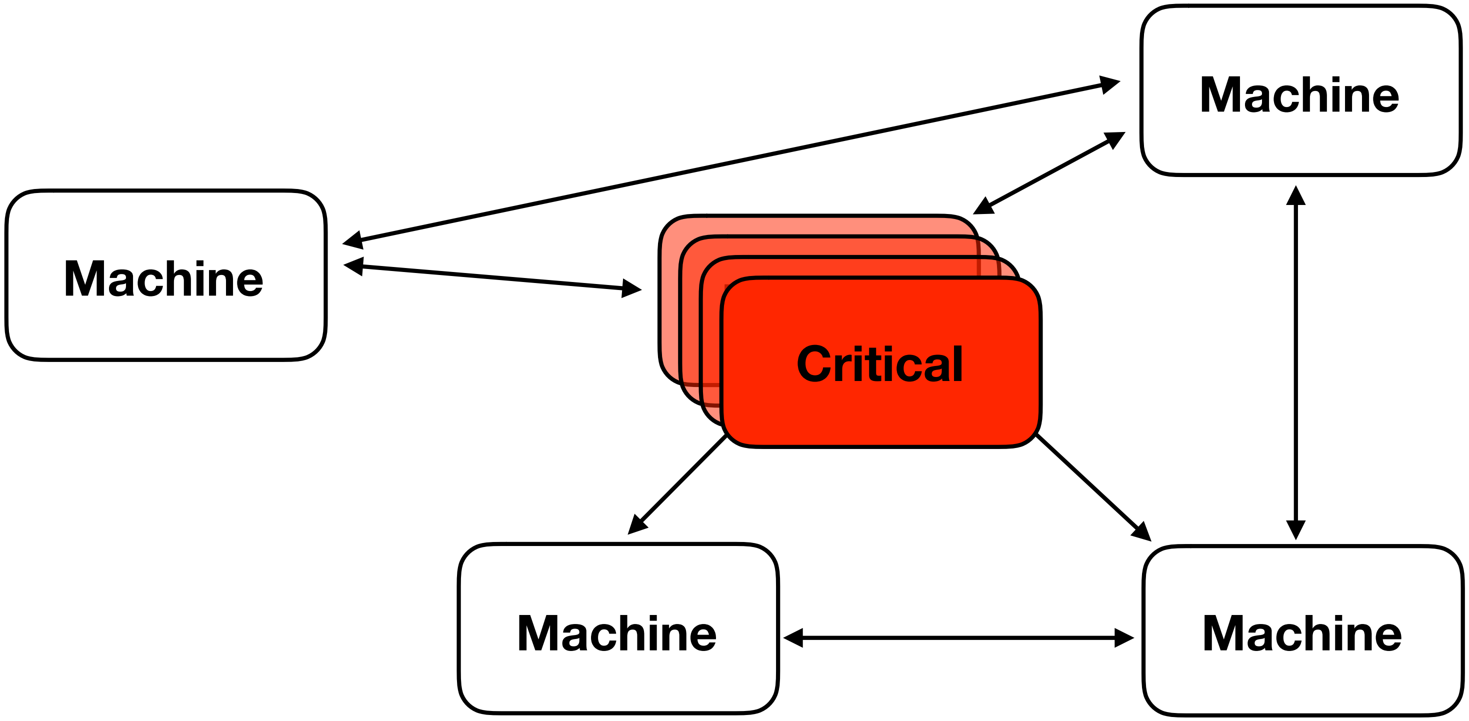
- Certain services are more critical than others



- Example: DNS, a database, etc.

High Level View

- Obvious Solution: Replication!



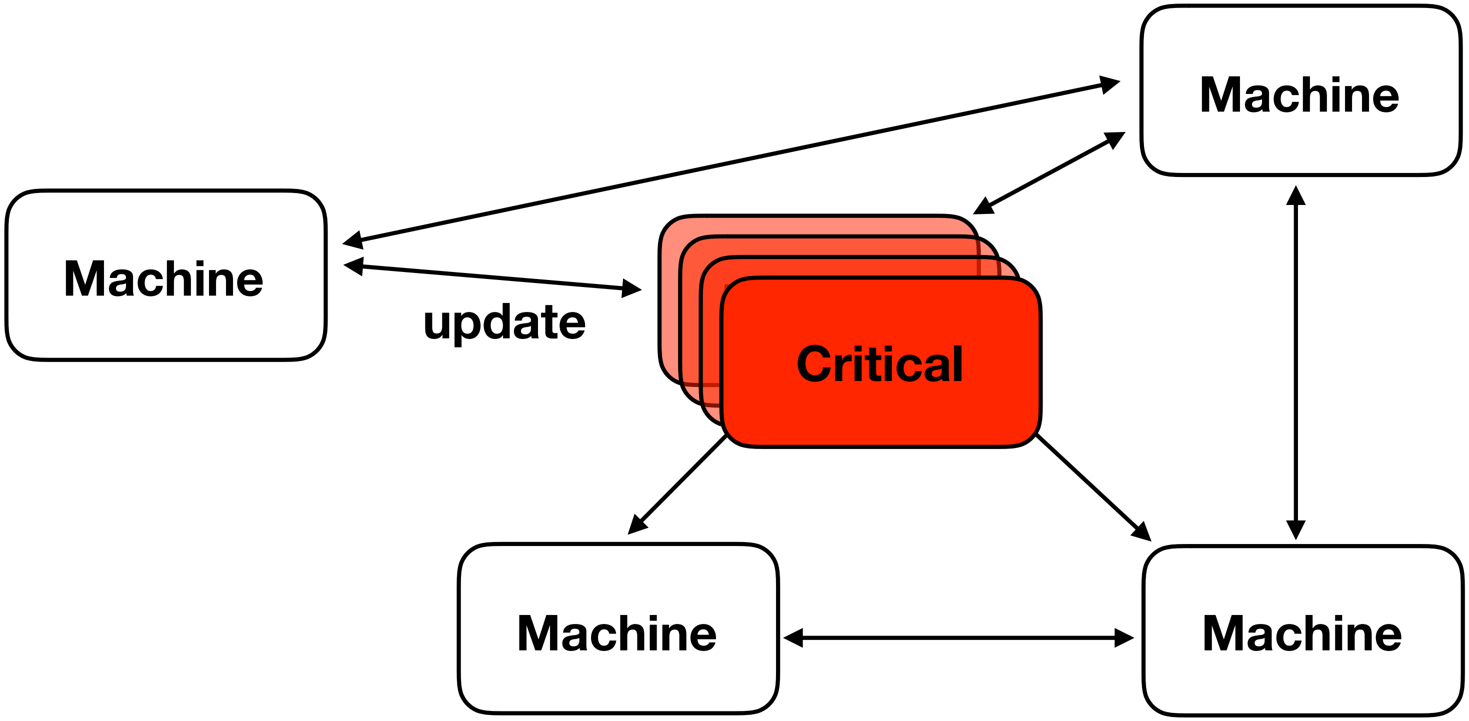
- Whew! Crisis averted via redundancy.

- It's "web scale"

Copyright (c) 2019, http://www.dabeaz.com

Major Problem

- Mutability/State Updates



- It is impossible for state to simultaneously update on all replicas at once (physics, timing, etc.)

- Also must account for all possible machine failures

Copyright (c) 2019, http://www.dabeaz.com

Solution: Consensus

- Replicated servers require a mechanism for agreeing on the "state" of the system.

- This is one of the central hard problem of distributed computing

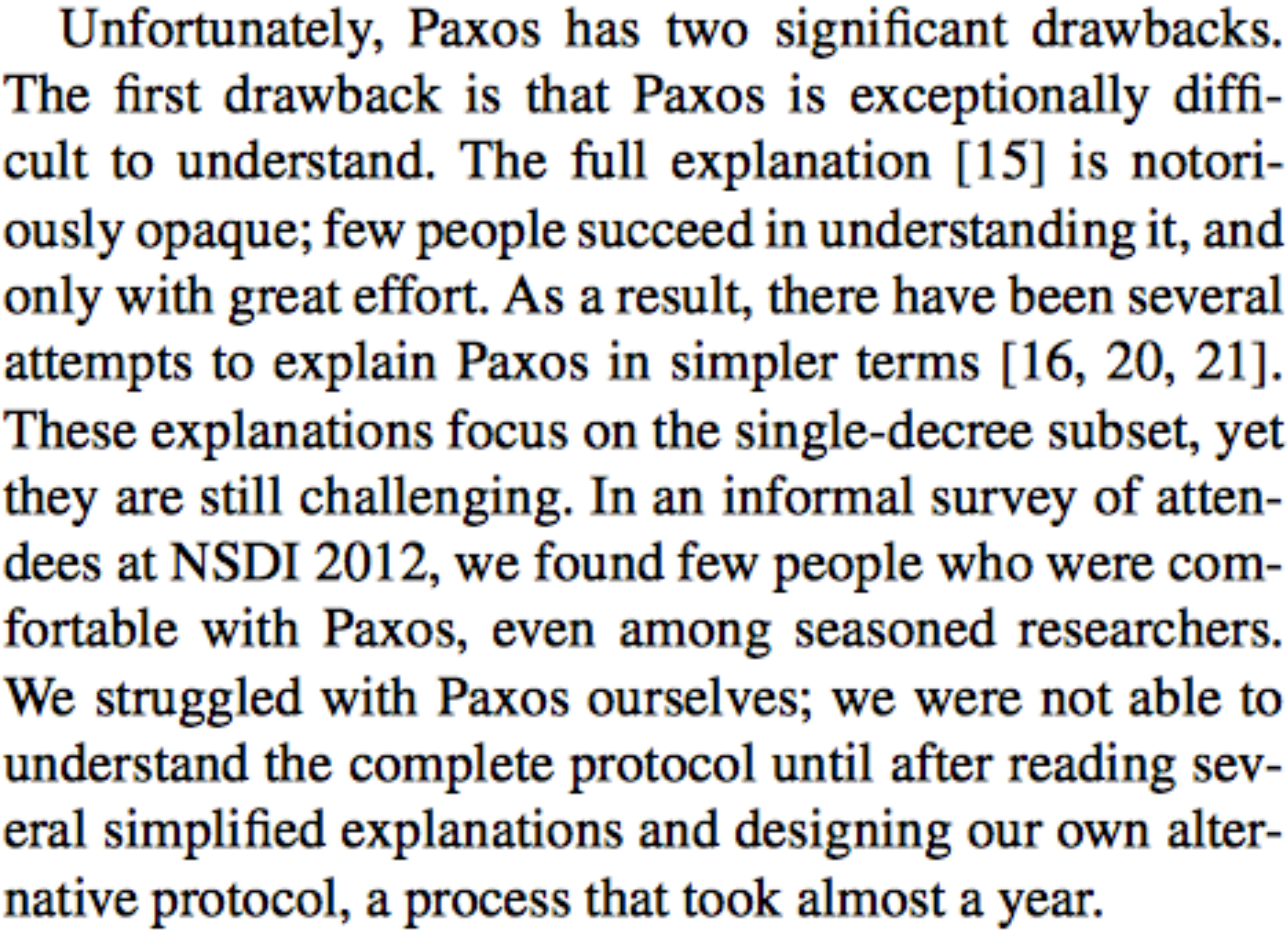
- Algorithms: Distributed Consensus

Copyright (c) 2019, http://www.dabeaz.com

Historical Background

- Consensus : Problem of maintaining consistent state 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
- Problem: Translating Paxos into an actual implementation is notoriously hard (mathematical, incomplete "details")

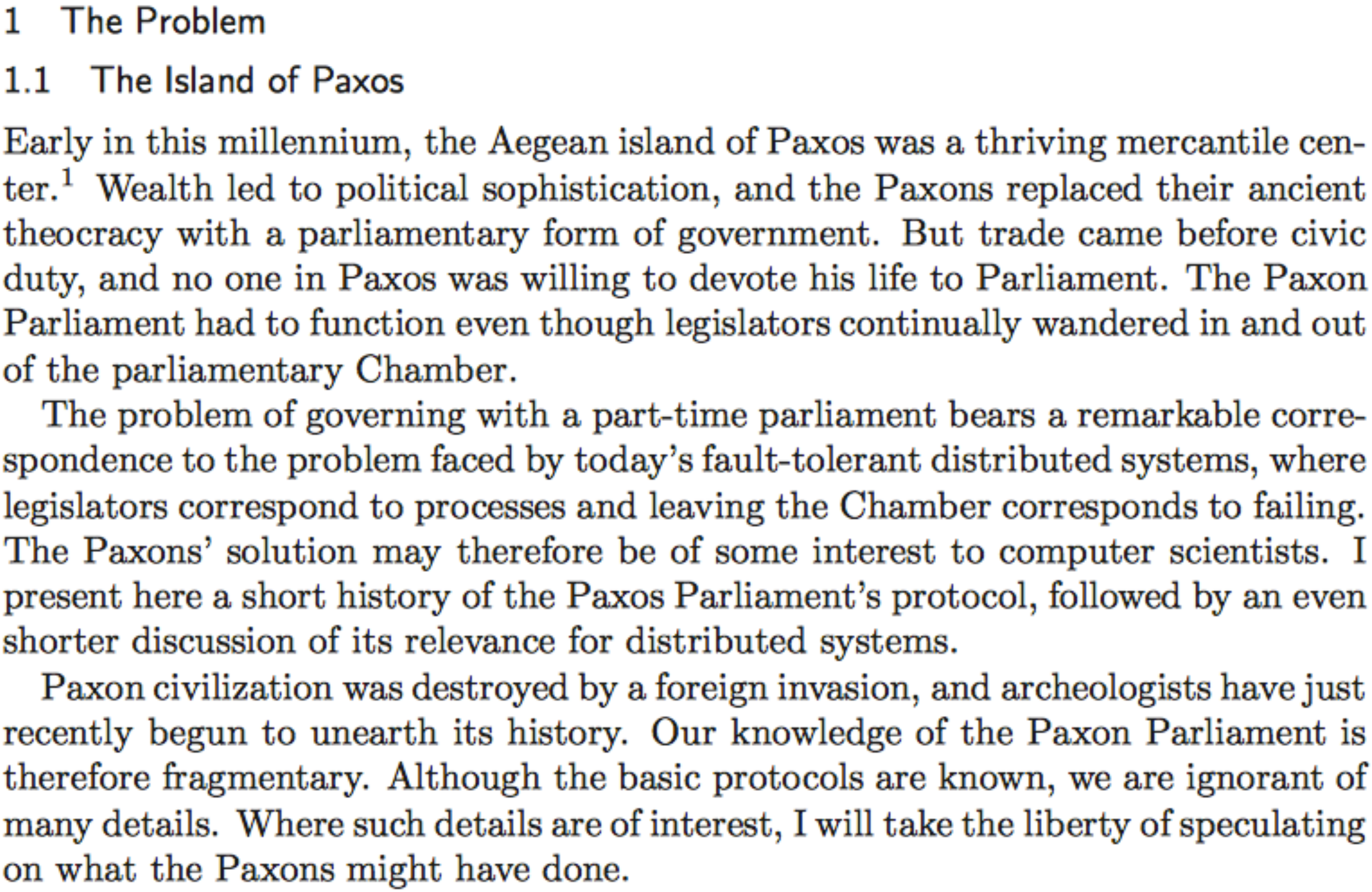
Copyright (c) 2019, http://www.dabeaz.com



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

Copyright (c) 2019, http://www.dabeaz.com



1 The Problem

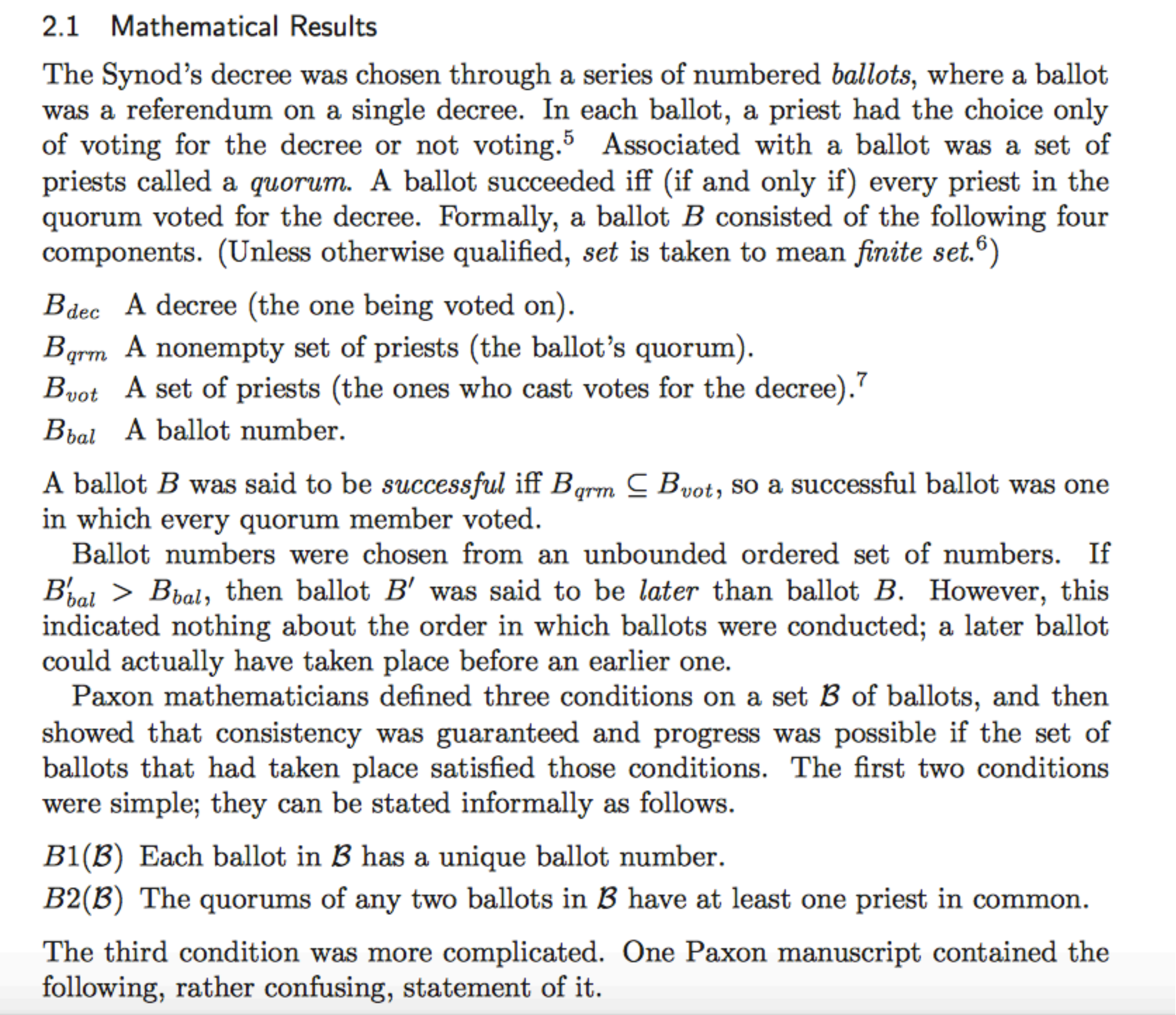
1.1 The Island of Paxos

Early in this millennium, the Aegean island of Paxos was a thriving mercantile center.[1] Wealth led to political sophistication, and the Paxons replaced their ancient theocracy with a parliamentary form of government. But trade came before civic duty, and no one in Paxos was willing to devote his life to Parliament. The Paxon Parliament had to function even though legislators continually wandered in and out of the parliamentary Chamber.

The problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by today's fault-tolerant distributed systems, where legislators correspond to processes and leaving the Chamber corresponds to failing. The Paxons' solution may therefore be of some interest to computer scientists. I present here a short history of the Paxos Parliament's protocol, followed by an even shorter discussion of its relevance for distributed systems.

Paxon civilization was destroyed by a foreign invasion, and archeologists have just recently begun to unearth its history. Our knowledge of the Paxon Parliament is therefore fragmentary. Although the basic protocols are known, we are ignorant of many details. Where such details are of interest, I will take the liberty of speculating on what the Paxons might have done.

Copyright (c) 2019, http://www.dabeaz.com



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.

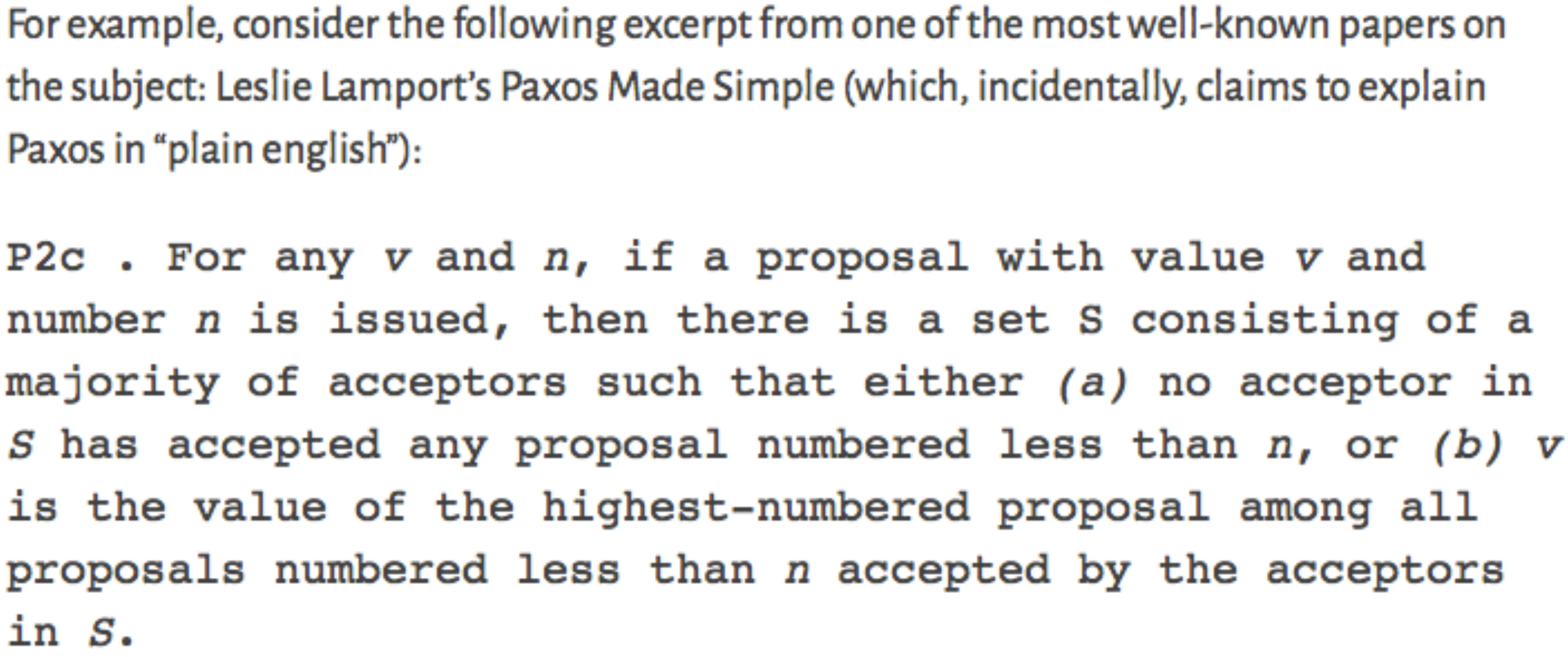
Paxon mathematicians defined three conditions on a set $\mathcal{B}$ of ballots, and then showed that consistency was guaranteed and progress was possible if the set of ballots that had taken place satisfied those conditions. The first two conditions were simple; they can be stated informally as follows.

$B1(\mathcal{B})$ 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.

Copyright (c) 2019, http://www.dabeaz.com



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*.

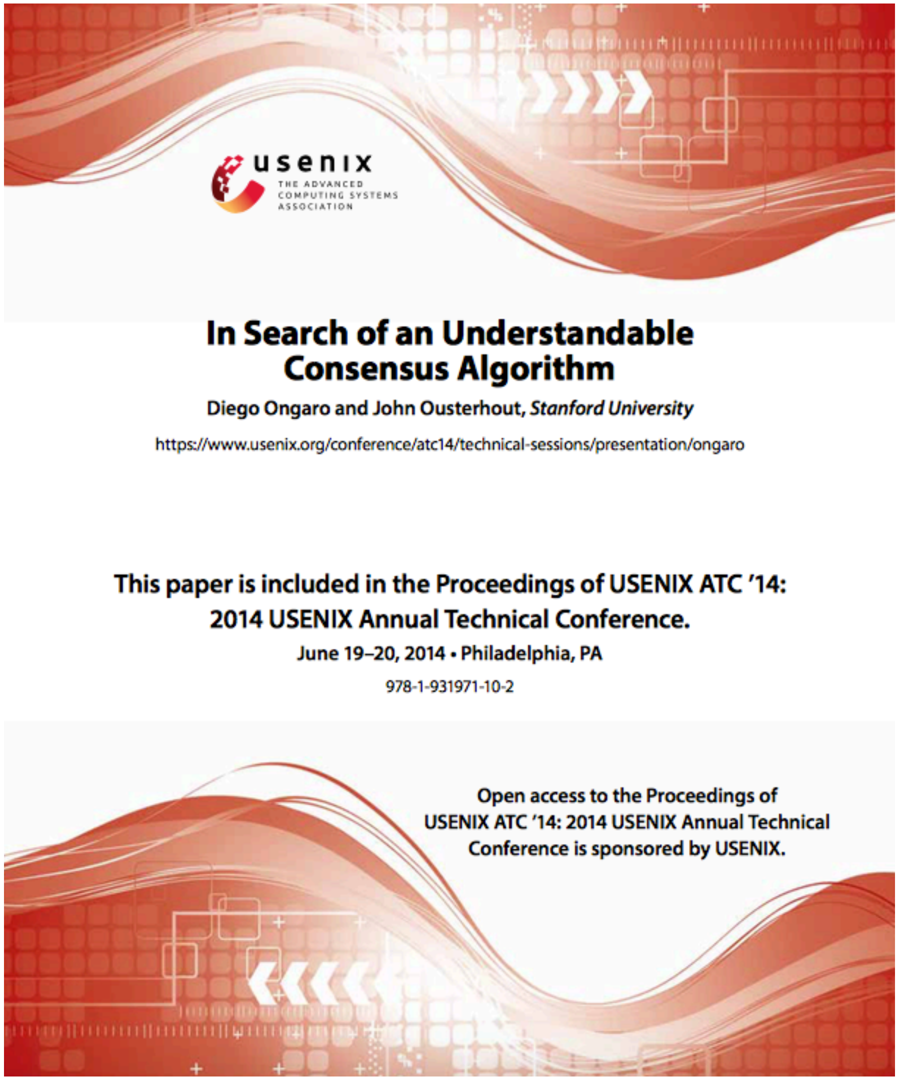
Copyright (c) 2019, http://www.dabeaz.com

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

- Mike Burrows

Copyright (c) 2019, http://www.dabeaz.com

Our Challenge



usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

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.

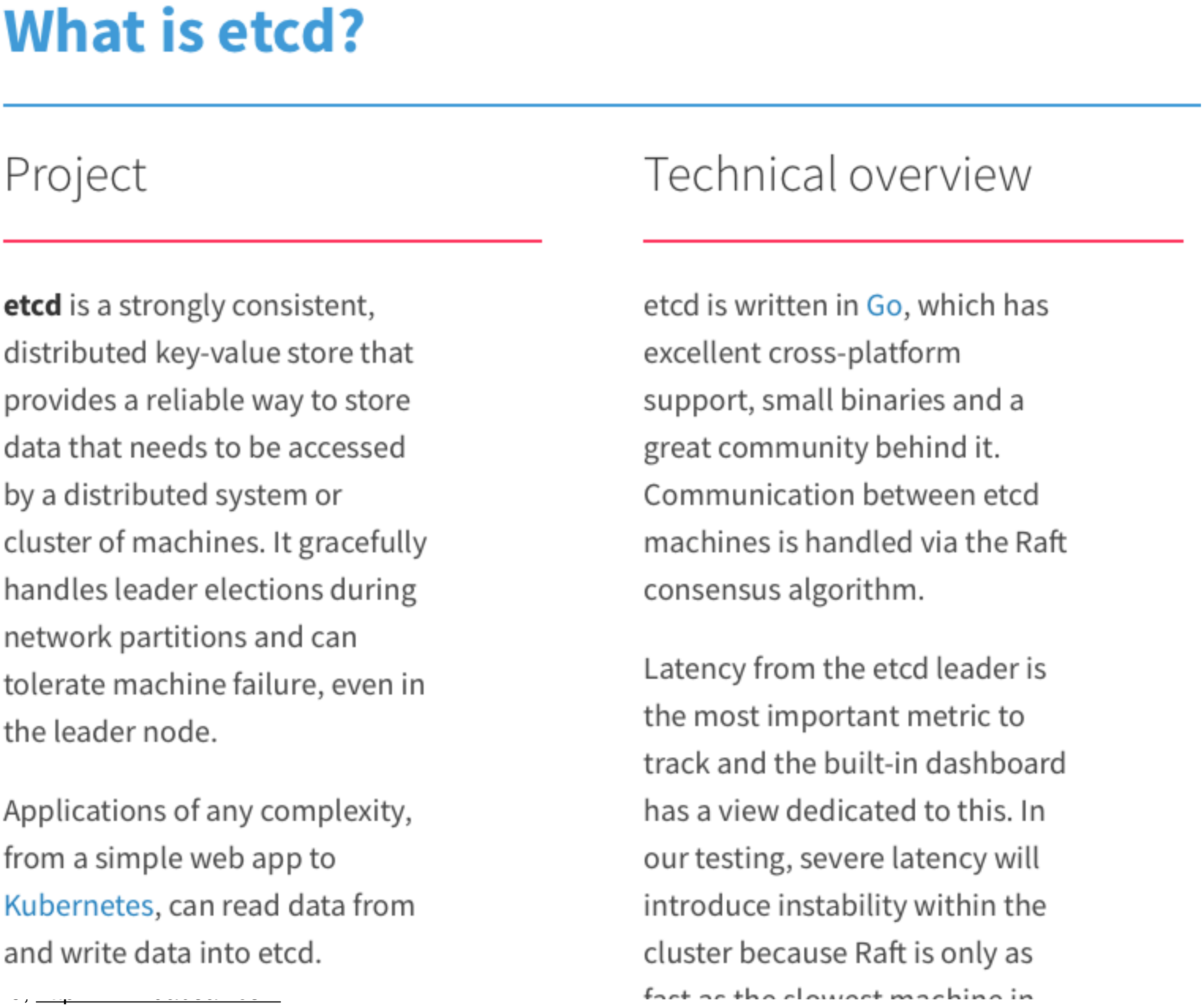
June 19–20, 2014 • Philadelphia, PA

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"

Real World Raft Example



What is etcd?

Project

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

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

Technical overview

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

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

Copyright (c)

Why This Topic?

- Real world: "Fault Tolerance" is 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

Copyright (c) 2019, http://www.dabeaz.com

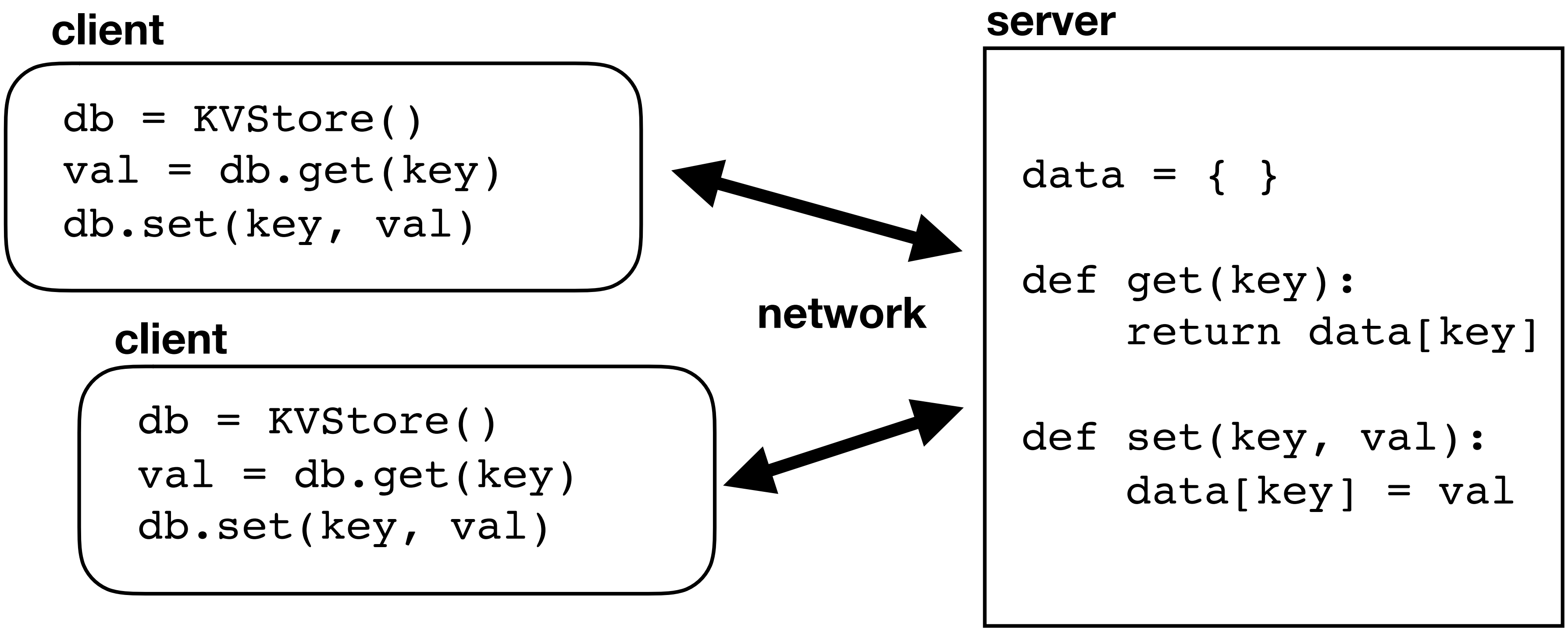
Core Topics

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

Copyright (c) 2019, http://www.dabeaz.com

The Project

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



client

```
db = KVStore()
val = db.get(key)
db.set(key, val)
```

client

```
db = KVStore()
val = db.get(key)
db.set(key, val)
```

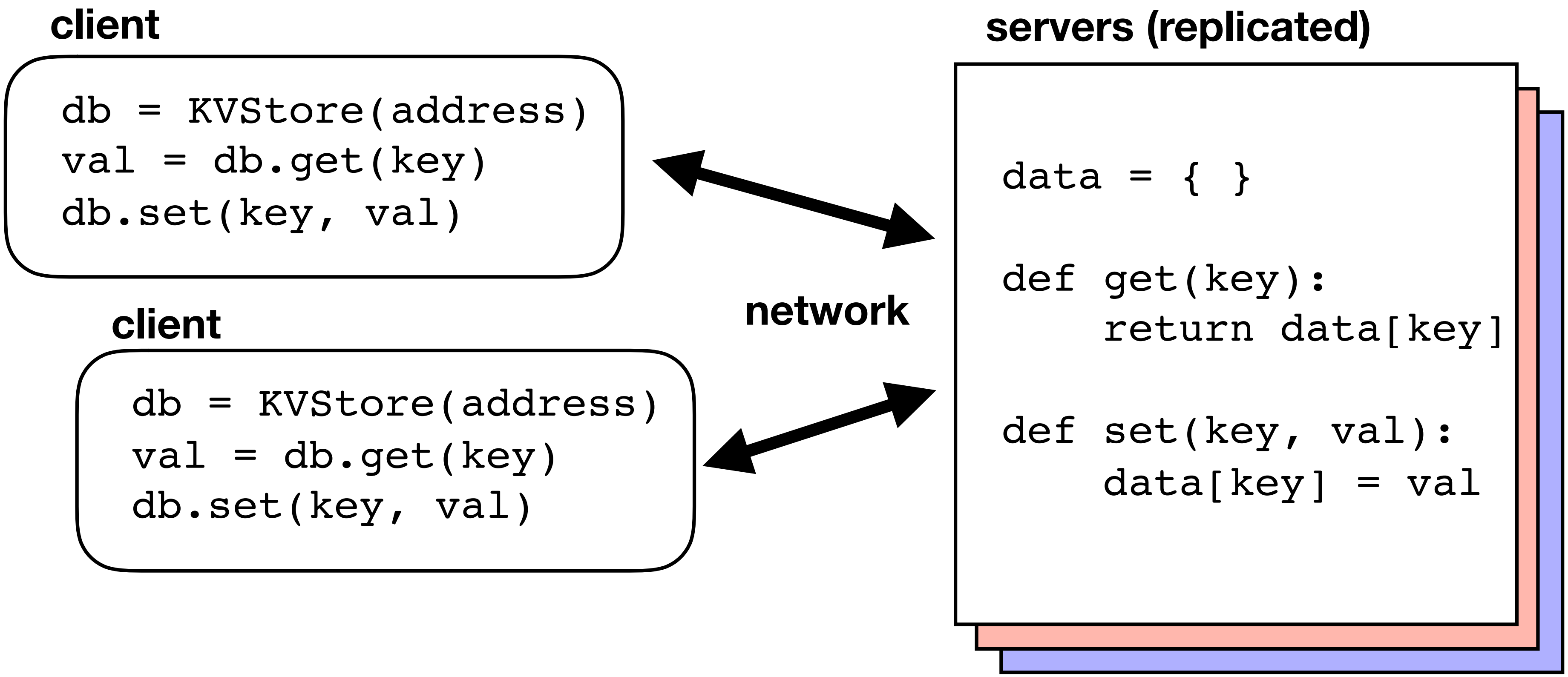
network

server

```
data = { }

def get(key):
    return data[key]

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

The Problem

- Fault-tolerance
- Always available, never lose data



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

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

```
data = { }

def get(key):
    return data[key]

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

Copyright (c) 2019, http://www.dabeaz.com

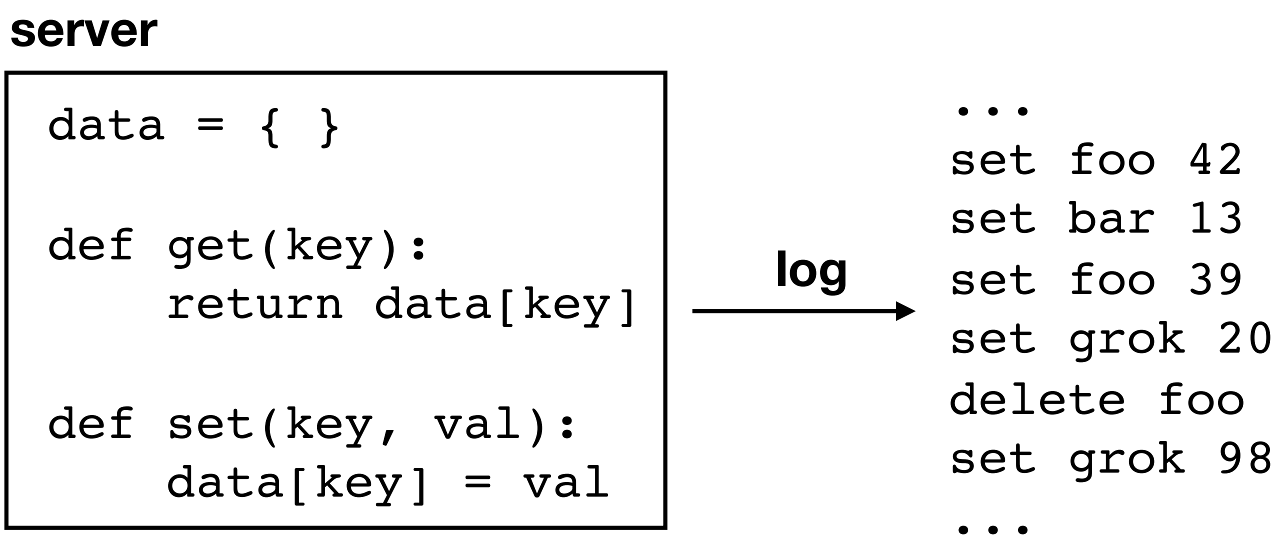
Raft

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

Copyright (c) 2019, http://www.dabeaz.com

Transaction Logs

- Servers record 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
...
```

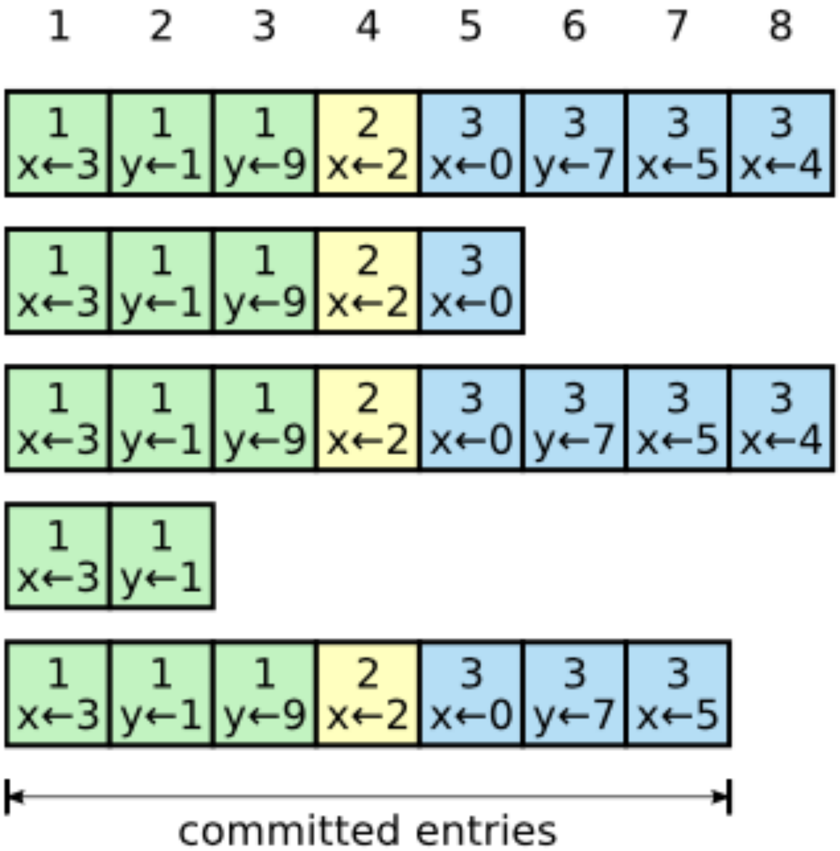
- Log keeps an ordered record of all state changes

- Crash recovery: Replay the log to restore state

Copyright (c) 2019, http://www.dabeaz.com

Replication

- The servers replicate the transaction log



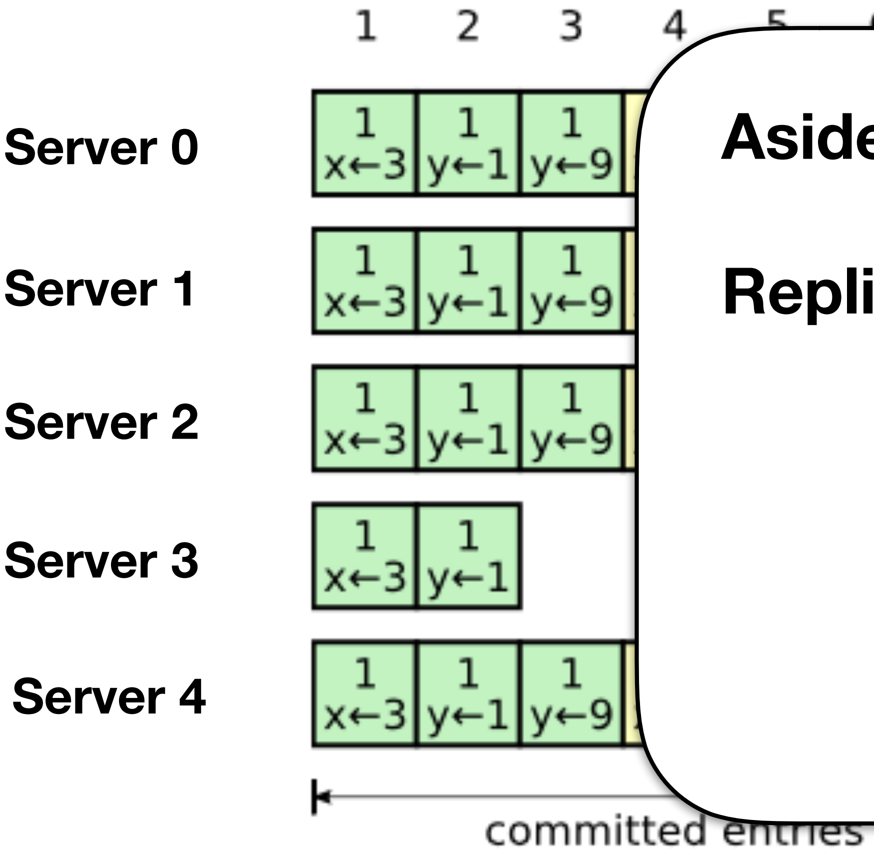
	1	2	3	4	5	6	7	8
Server 0	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4
Server 1	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0			
Server 2	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4
Server 3	1 x←3	1 y←1						
Server 4	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	

- Log contains committed and uncommitted entries

Copyright (c) 2019, http://www.dabeaz.com

Replication

- The servers replicate the transaction log



	1	2	3
Server 0	1 x←3	1 y←1	1 y←9
Server 1	1 x←3	1 y←1	1 y←9
Server 2	1 x←3	1 y←1	1 y←9
Server 3	1 x←3	1 y←1	
Server 4	1 x←3	1 y←1	1 y←9

Aside:

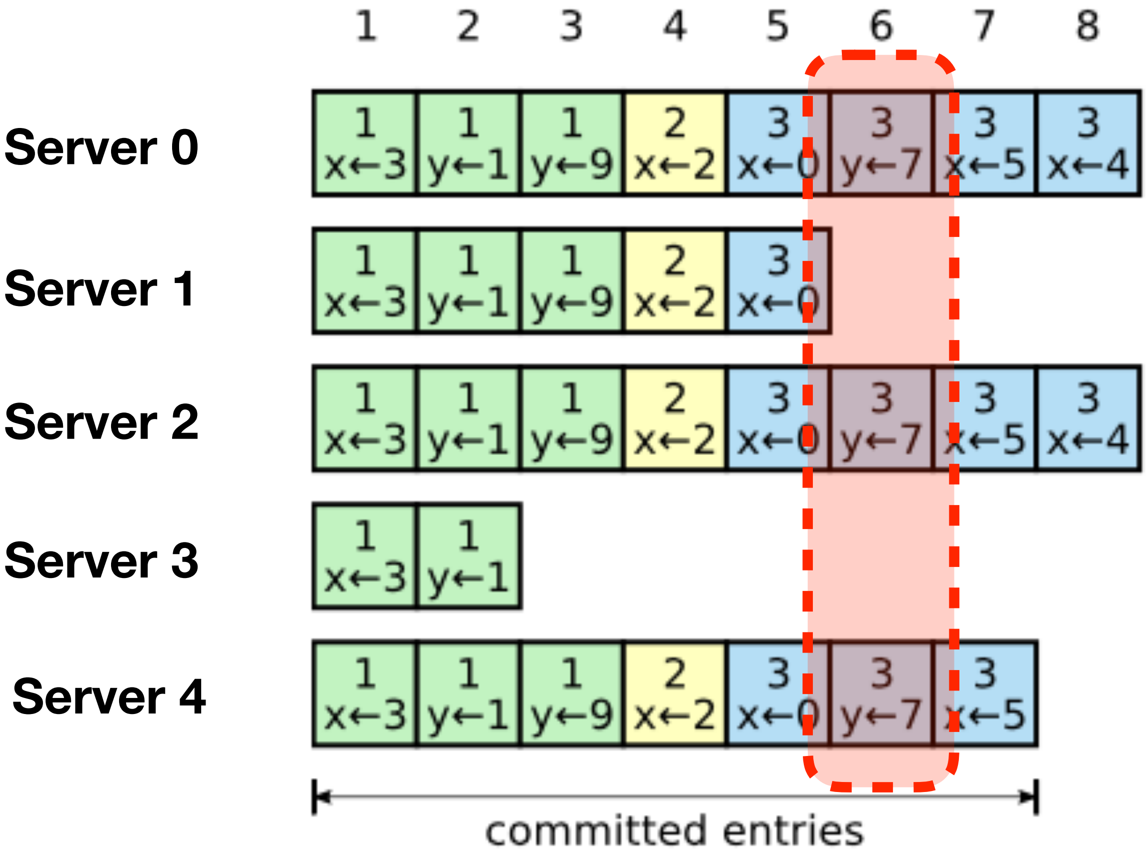
Replicated Logs == Raft

- Log contains committed and uncommitted entries

Copyright (c) 2019, http://www.dabeaz.com

Majority Rules

- Transactions are committed by consensus



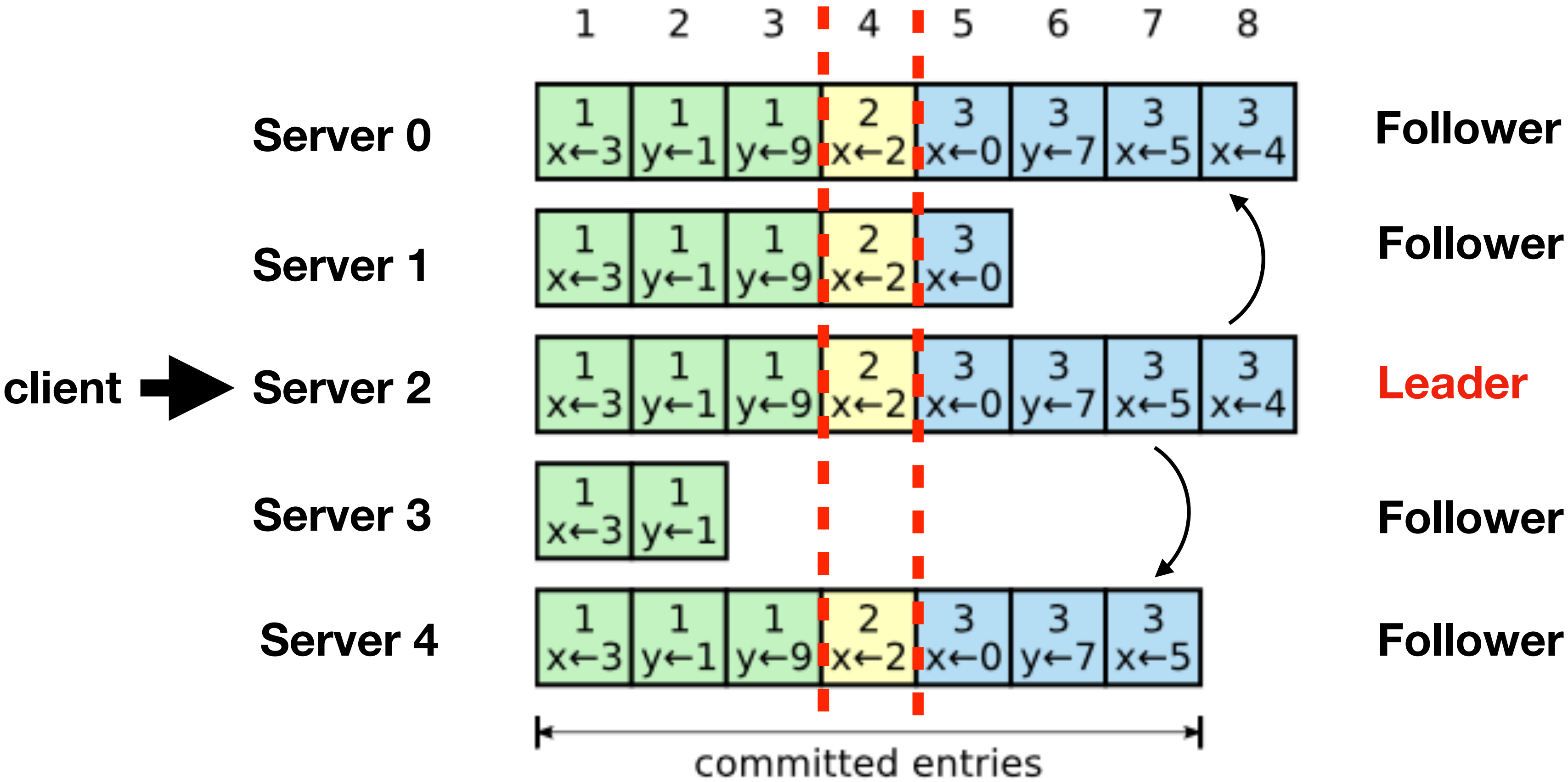
	1	2	3	4	5	6	7	8
Server 0	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4
Server 1	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0			
Server 2	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4
Server 3	1 x←3	1 y←1						
Server 4	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	

- Consensus means replication on a quorum

Copyright (c) 2019, http://www.dabeaz.com

There Can Be Only One

- All actions are coordinated by <u>one and only one</u> leader

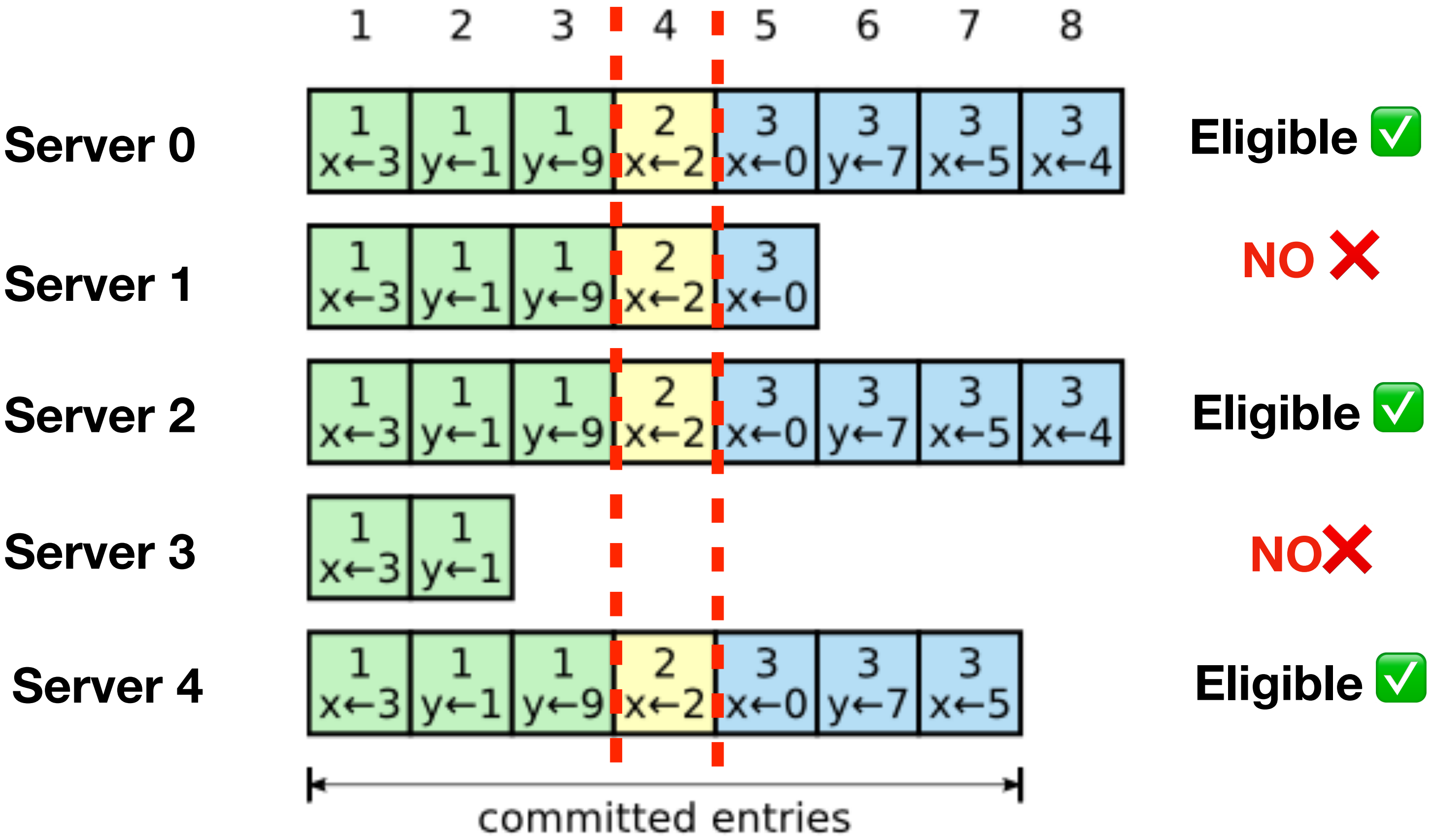


	1	2	3	4	5	6	7	8	
Server 0	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4	Follower
Server 1	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0				Follower
Server 2	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4	Leader
Server 3	1 x←3	1 y←1							Follower
Server 4	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5		Follower

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

Knowledge is Power

- Only followers with all committed entries can lead



	1	2	3	4	5	6	7	8	
Server 0	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4	**Eligible** ✅
Server 1	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0				**NO** ❌
Server 2	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4	**Eligible** ✅
Server 3	1 x←3	1 y←1							**NO**❌
Server 4	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5		**Eligible** ✅

🤔

- Any quorum will have at least one member with a full log

Copyright (c) 2019, http://www.dabeaz.com

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.

Copyright (c) 2019, http://www.dabeaz.com

The Plan

- I 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

Copyright (c) 2019, http://www.dabeaz.com

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. Better strategy: make it 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.

Copyright (c) 2019, http://www.dabeaz.com

How to Fail

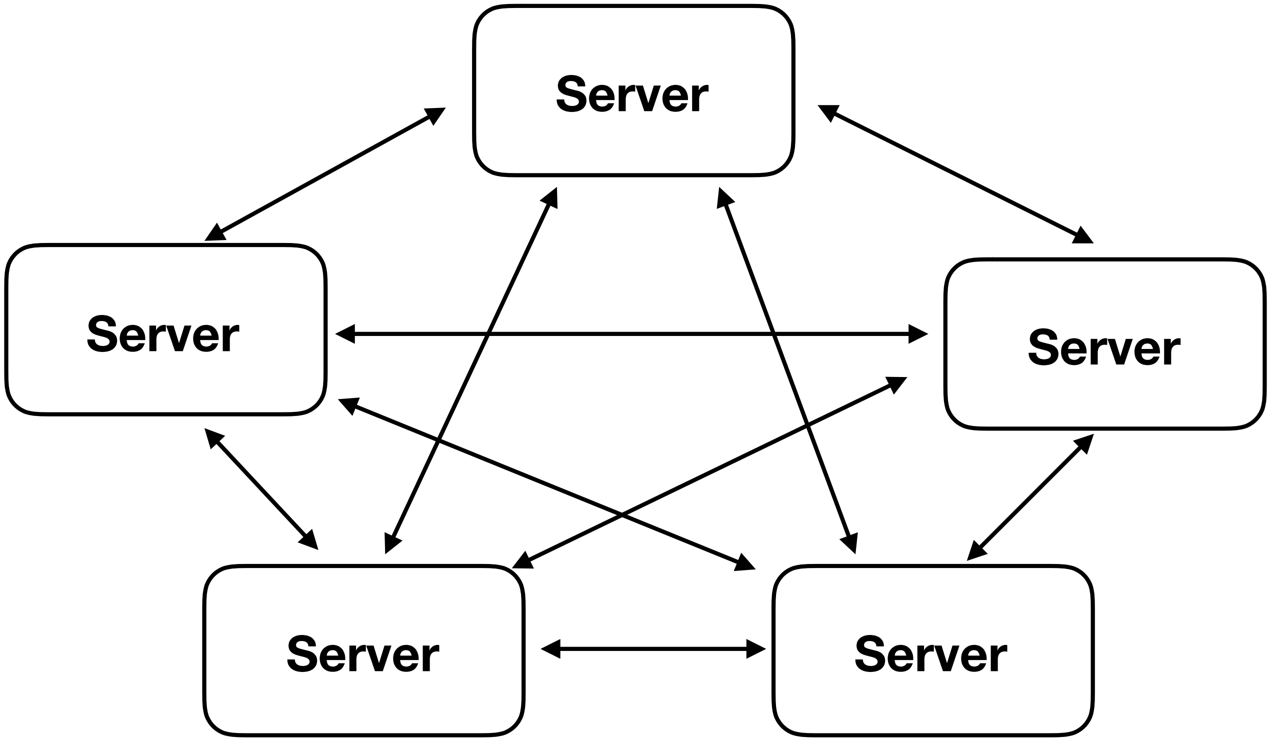
- **Detail Paralysis:** The Raft paper can be challenging to read. Don't get bogged down in tiny details or in ambiguity. Trust your intuition in devising a solution.
- **Silent Suffering:** Don't spend hours trying to track down some bug or being confused about part of the Raft paper/ specification. 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.
- **Food coma:** Pace yourself ;-).

Part 1

Technical Foundations

Raft and Networks

- Raft involves a cluster of identical servers

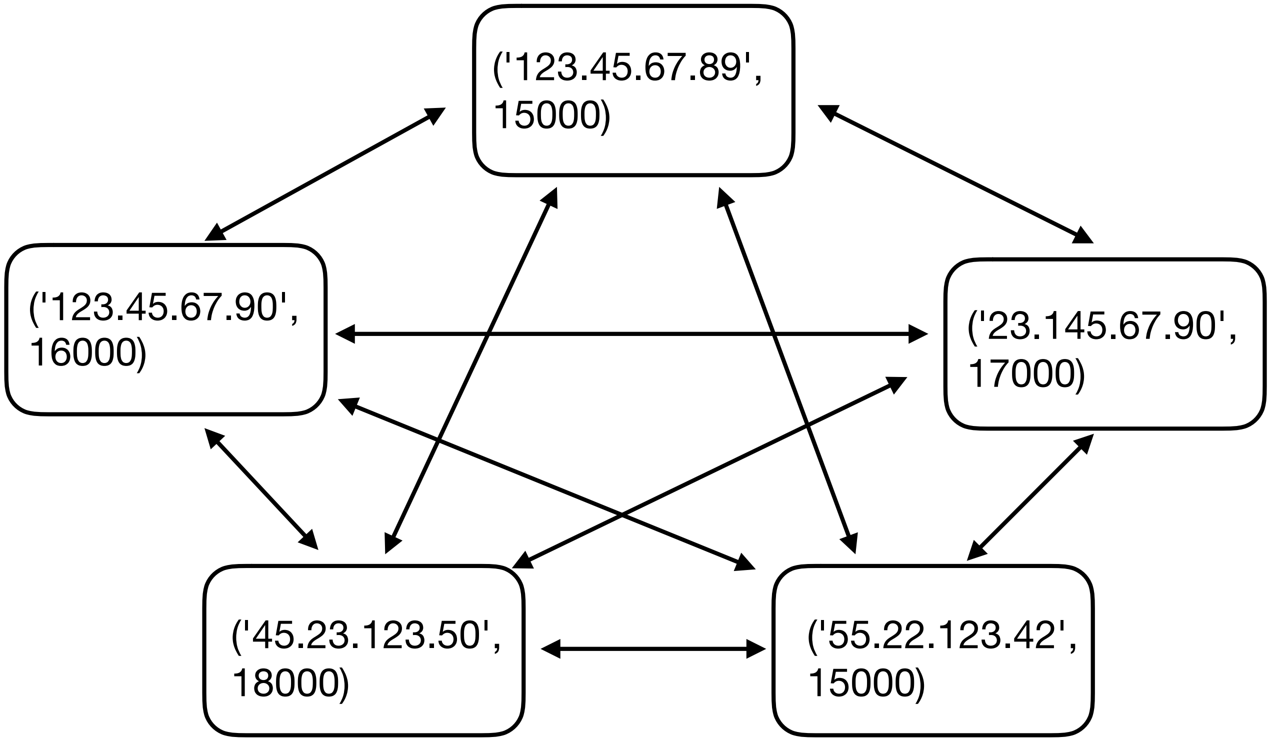


- They exchange messages over a network

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Network Addressing

- Servers are identified by (hostname, port)



- The addresses are part of the configuration

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Configuration

- You'll need to maintain a network config

```
# raftconfig.py

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

- Raft involves a cluster of machines that work together to maintain consensus.

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 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 work with partial data (might have to assemble into a final message)

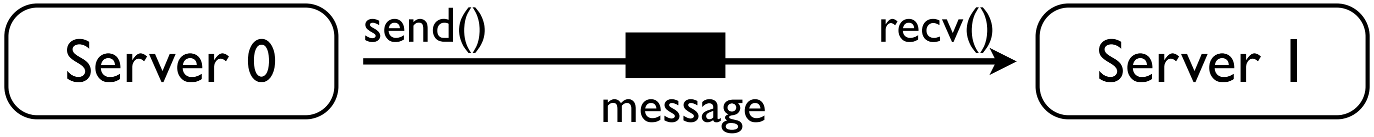
Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Exercise

- Implement a simple echo server
- Implement an echo client
- Main goal: Figure out how to run/debug

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Message Passing

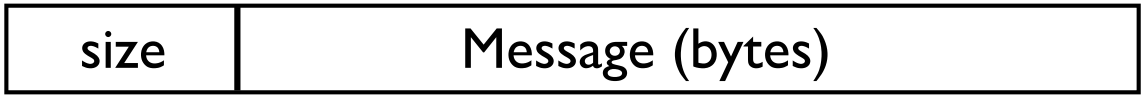


- Servers send and receive messages
- A message is a discrete packet of bytes
- Indivisible
- Sockets only send byte streams (not packets)

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

What is a Message?

- Usually a size-prefixed byte vector



size | Message (bytes)

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Payload Encoding

- Some options for Python
  - Pickle
  - JSON, XML, etc.
  - struct module (binary encoding)

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Exercise

- Create a higher level messaging interface

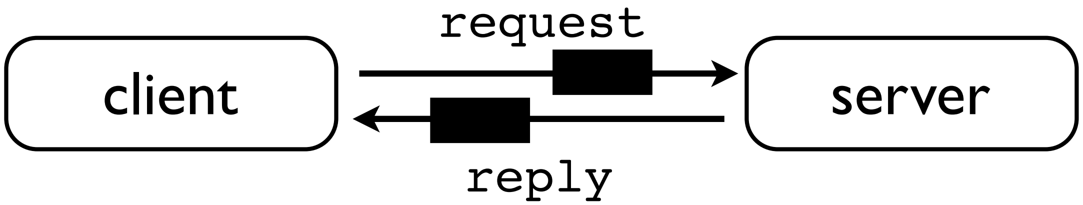
```
send_message(sock, msg)    # Send a message
msg = recv_message(sock)   # Receive a message
```

- It works with discrete messages
- It should wrap around existing sockets
- Test it by writing a simple echo service.
- KEEP. IT. SIMPLE.

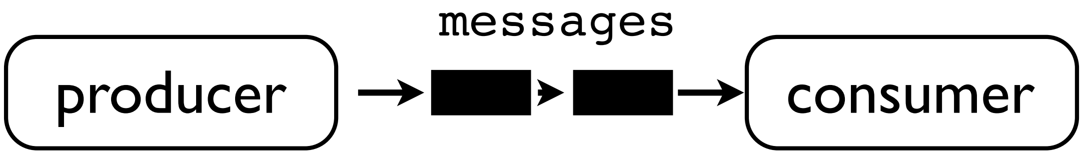
Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Messaging Patterns

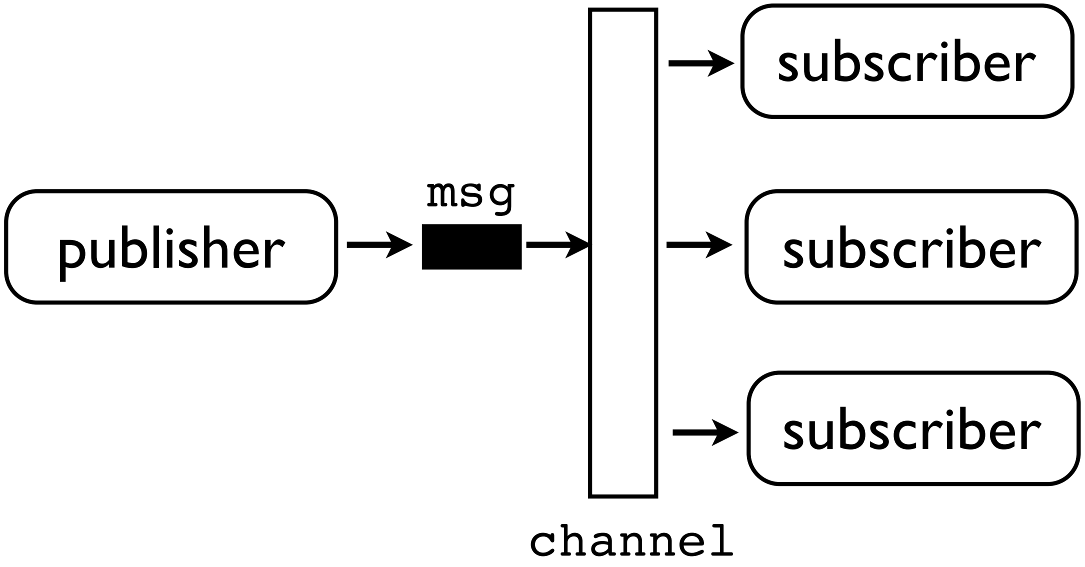
- Request/Reply



- Queue/FIFO



- Publish/Subscribe

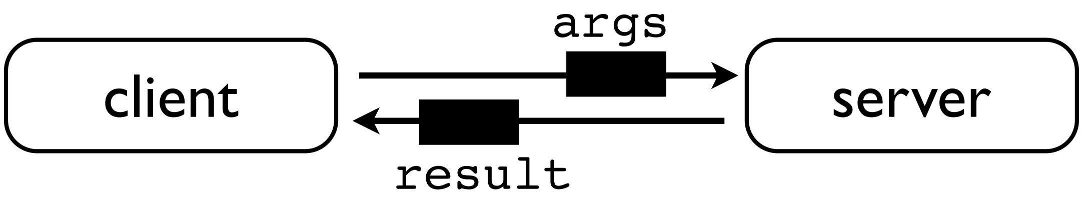


Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Remote Procedure Call

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

```
# server implementation
func():
    args = recv()
    ...
    send(result)
```



- Message passing, but the messages encode function arguments and results
- Client access via a proxy function

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Exercise

- Implement an RPC key-value store service

```
data = { }

def get(key):
    return data[key]

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

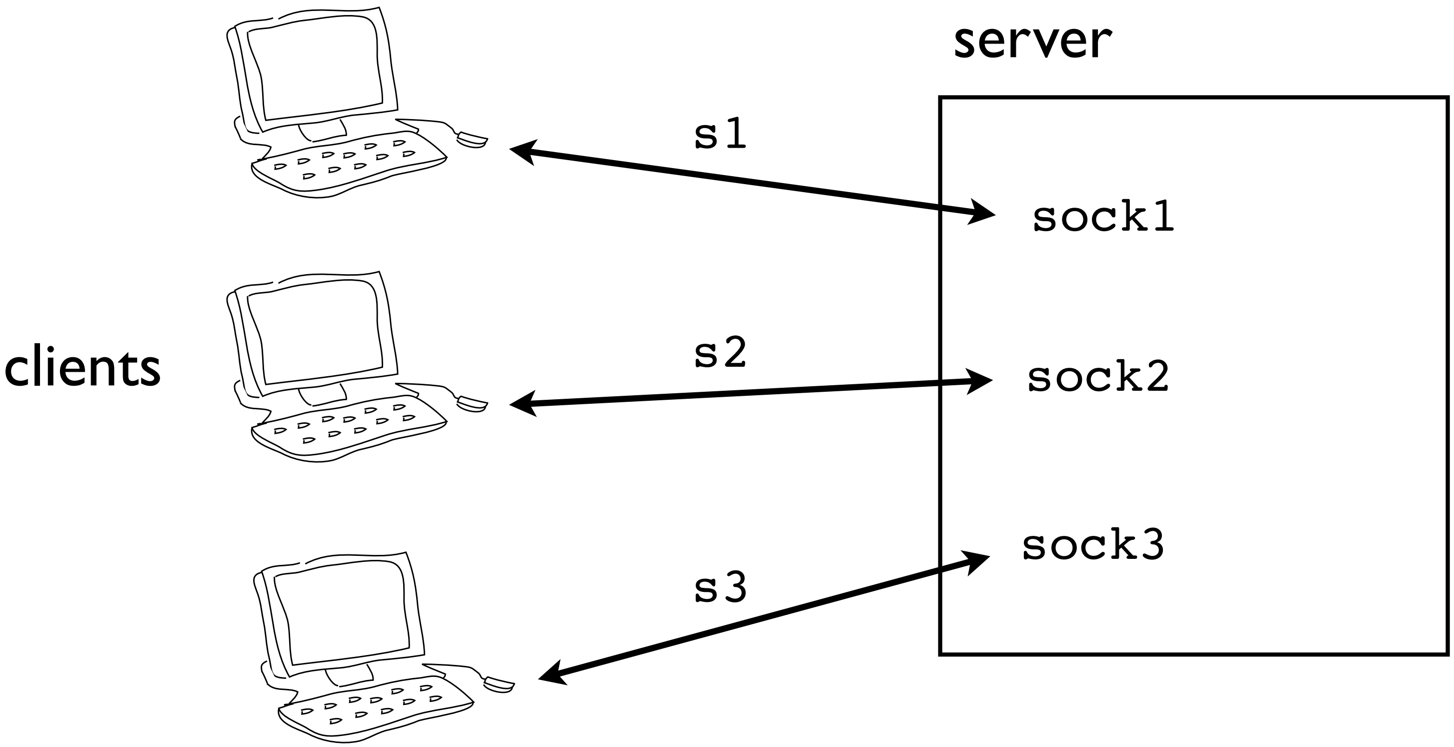
- Server should expose only those functions
- Create a client object that allows access

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

- KEEP. IT. SIMPLE.

Concurrency

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

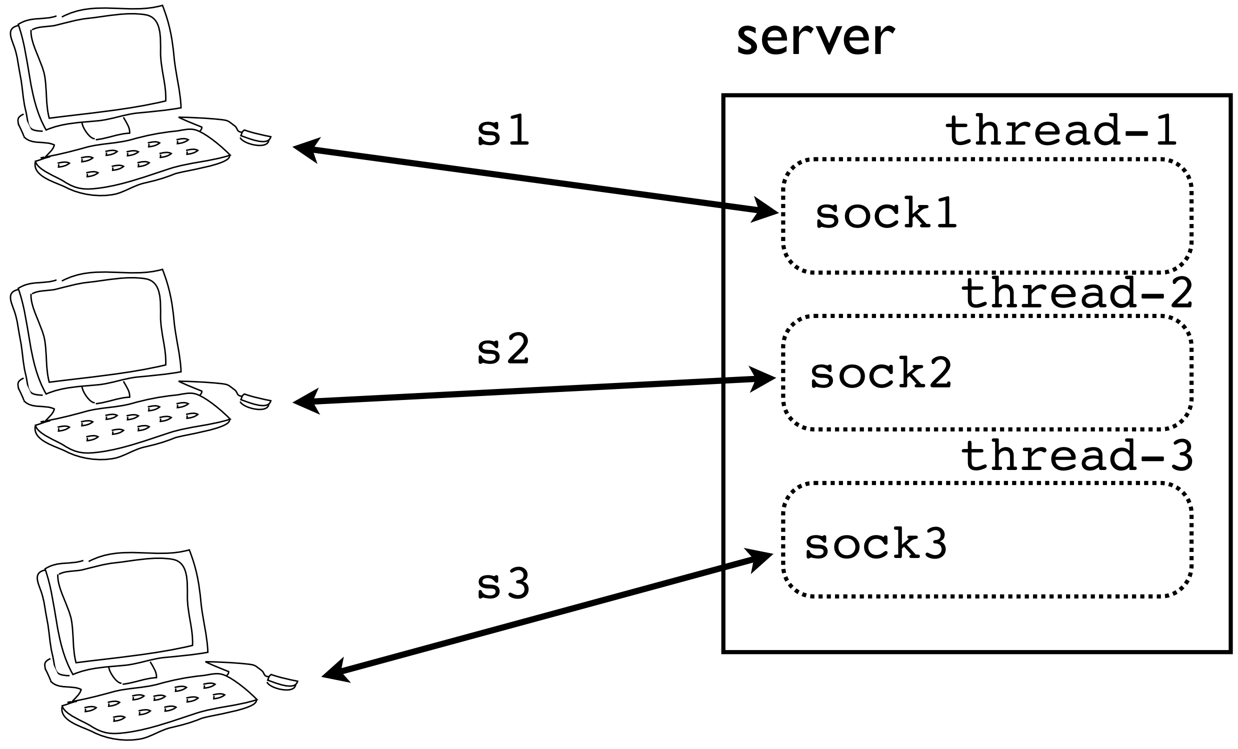


- How to coordinate execution on server?

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Multiple Clients

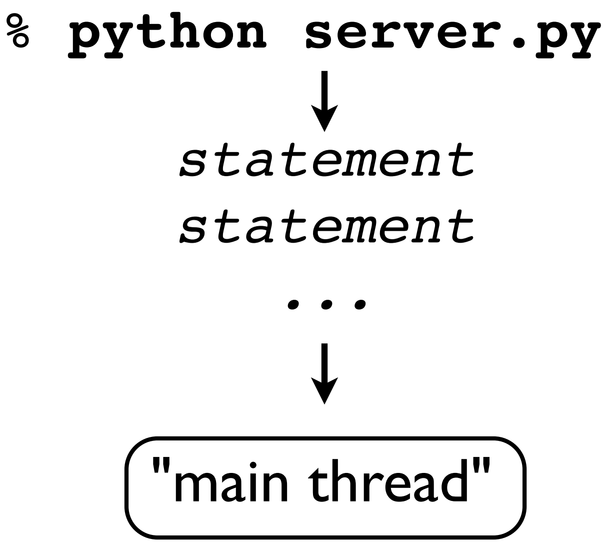
- One solution: handle each client in a thread



- Independent handling of each client

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Thread Basics



```
% python server.py
```

statement

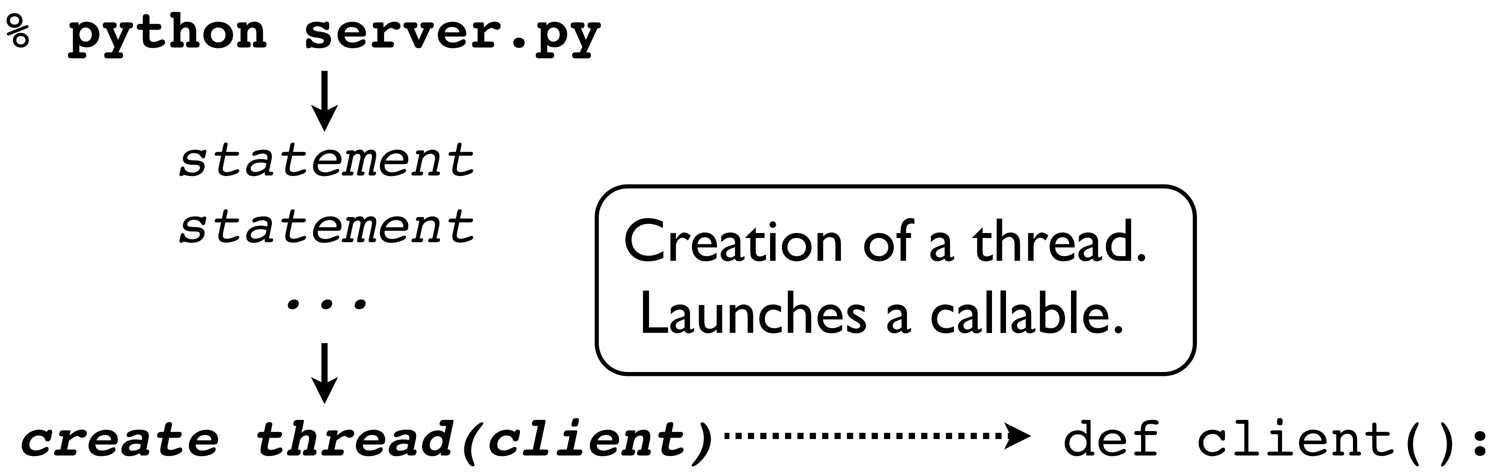
statement

...

"main thread"

Program launch.
Program starts
executing statements

Thread Basics

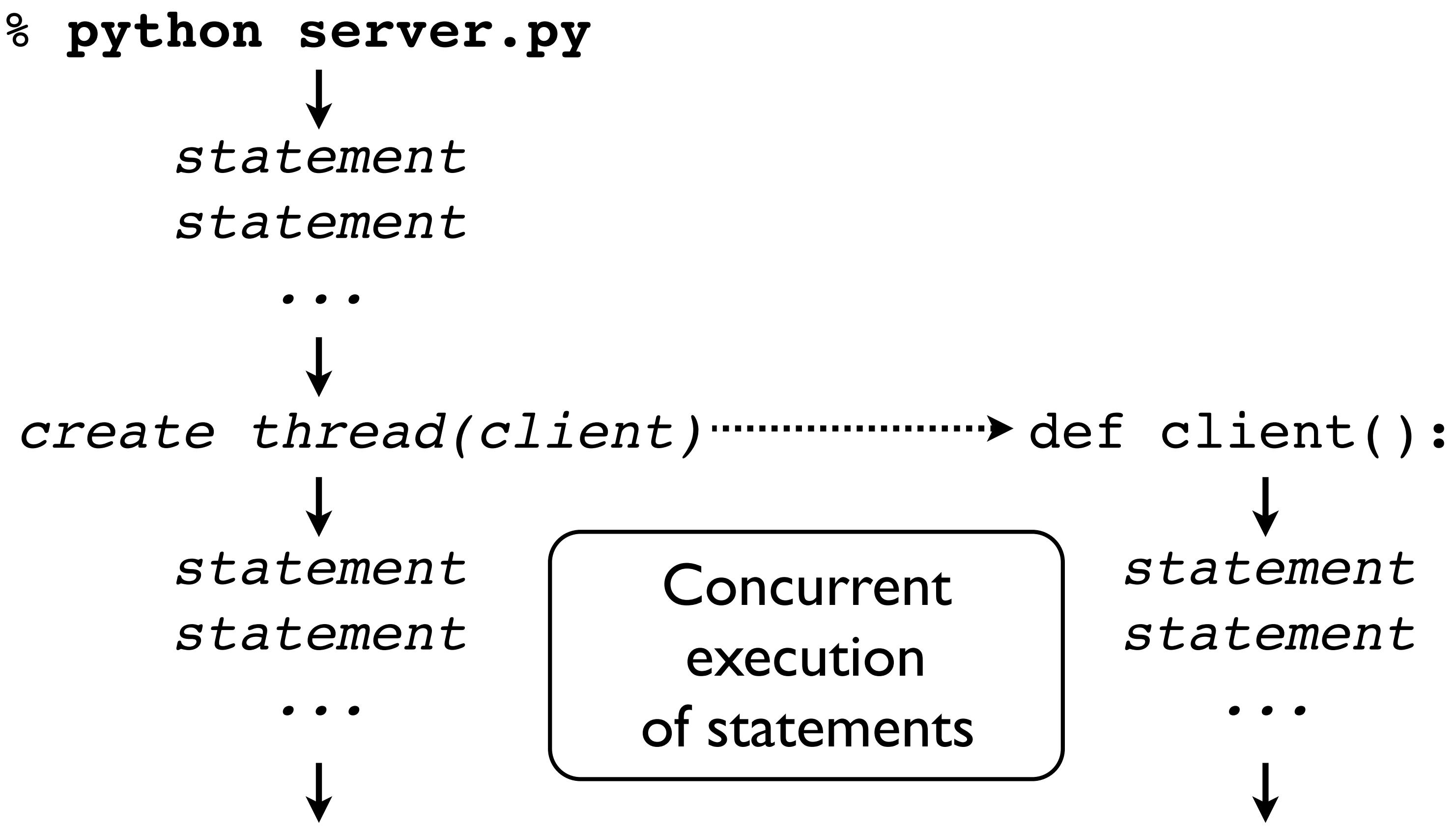


```
% python server.py
      statement
      statement
         ...
create thread(client) ········> def client():
```

Creation of a thread.
Launches a callable.

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Thread Basics



% **python server.py**

statement

statement

...

create thread(client)

def client():

statement

statement

...

Concurrent execution of statements

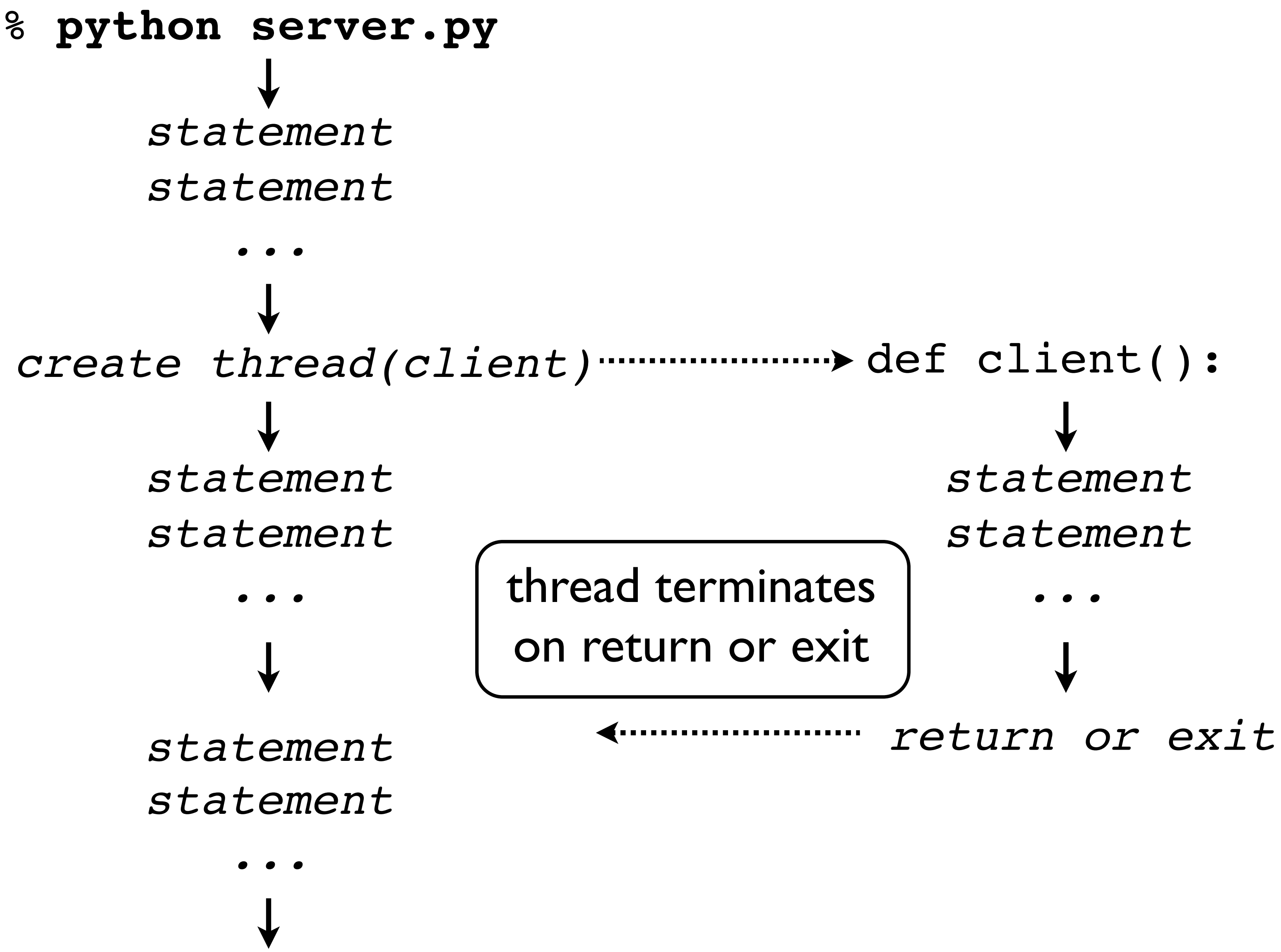
statement

statement

...

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

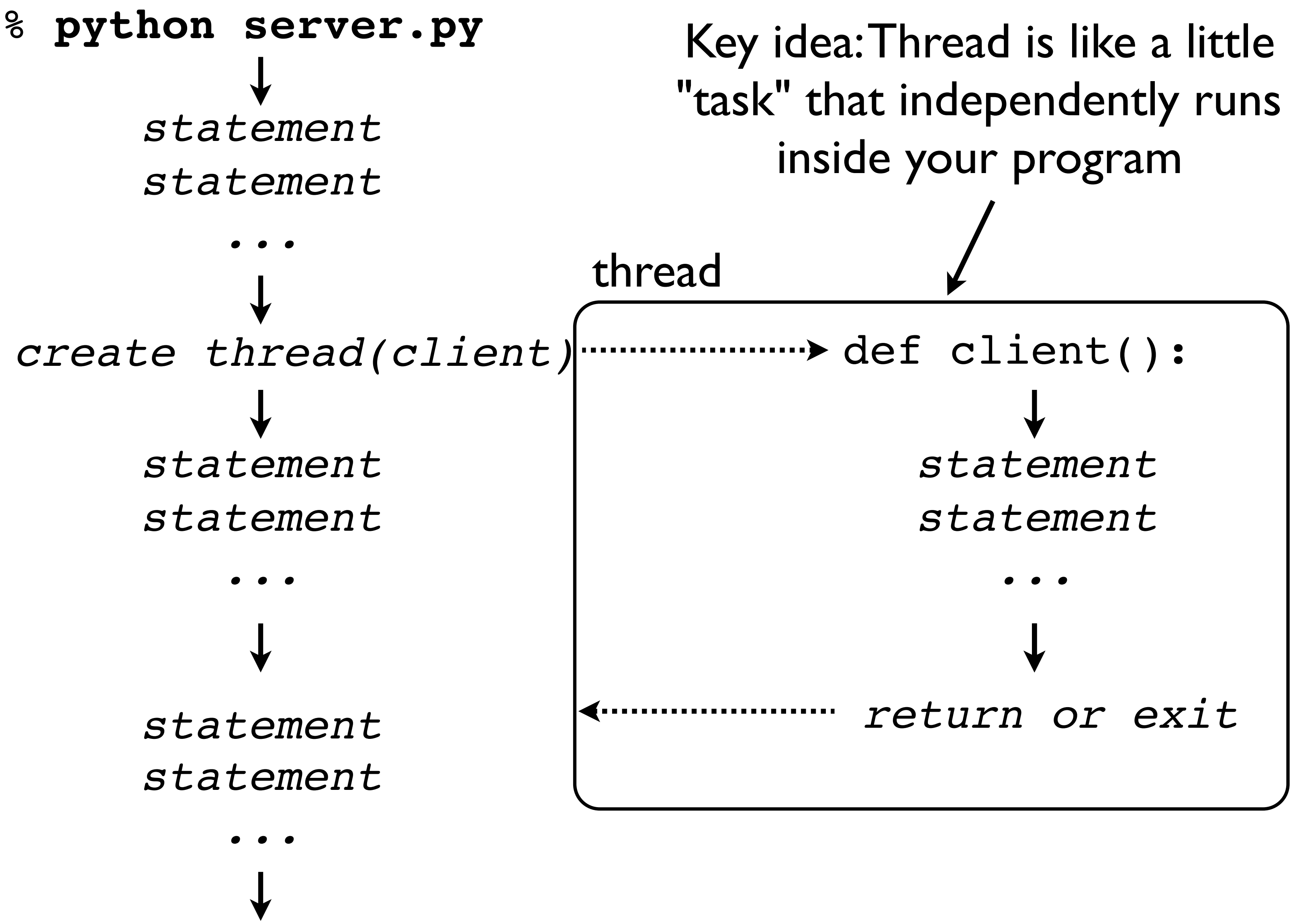
Thread Basics



```
% python server.py
      statement
      statement
      ...
create thread(client)        def client():
      statement                  statement
      statement                  statement
      ...                        ...
                                 return or exit
      statement
      statement
      ...
```

thread terminates on return or exit

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

```
def client():
```

statement
statement
...

return or exit

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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:
        ...
```

New thread on each connection

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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.

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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)

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Execution Order

- Consider shared state

```
data = { }
```

- One thread sets the value, another reads it

```
Thread-1                     Thread-2
--------                     --------
...                          ...
return data['x']             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                     Thread-2
--------                     --------
...                          ...
x = x + 1                    x = x - 1
...                          ...
```

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

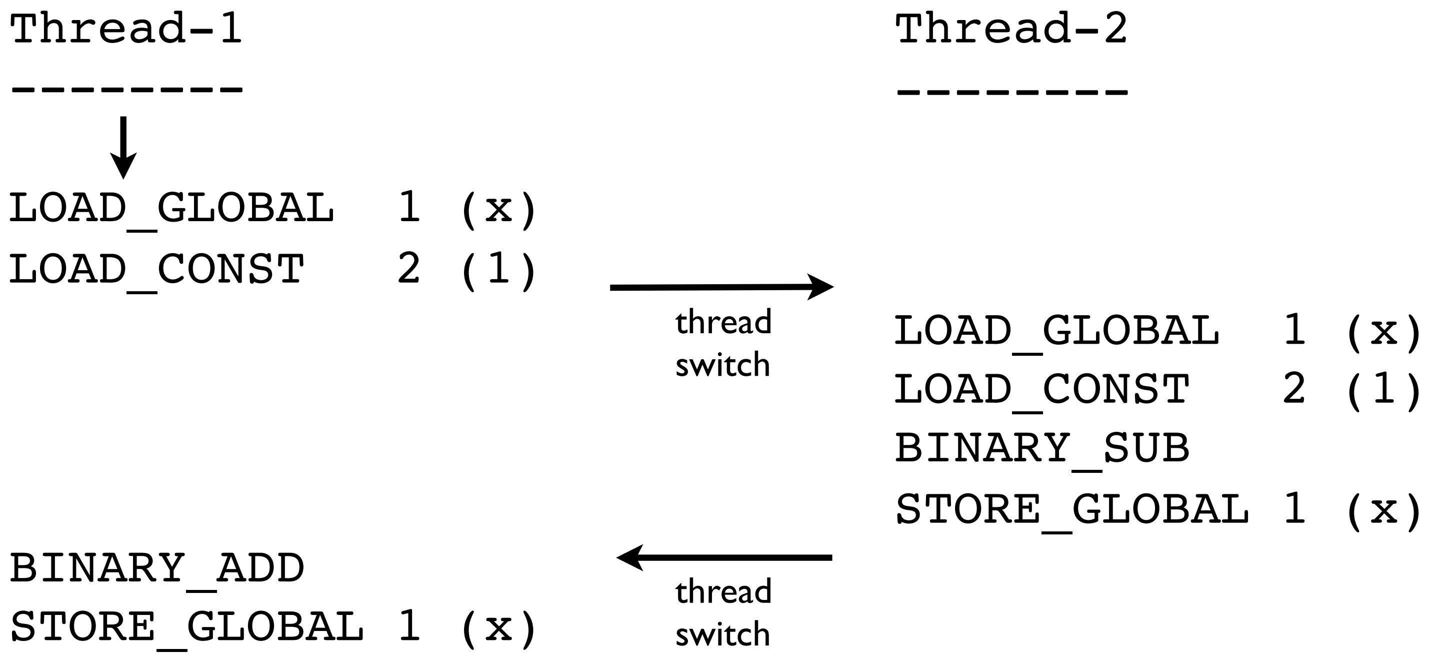
Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Concurrent Updates

- The two threads

```
Thread-1                     Thread-2
--------                     --------
...                          ...
x = x + 1                    x = x - 1
...                          ...
```

- Low level code execution

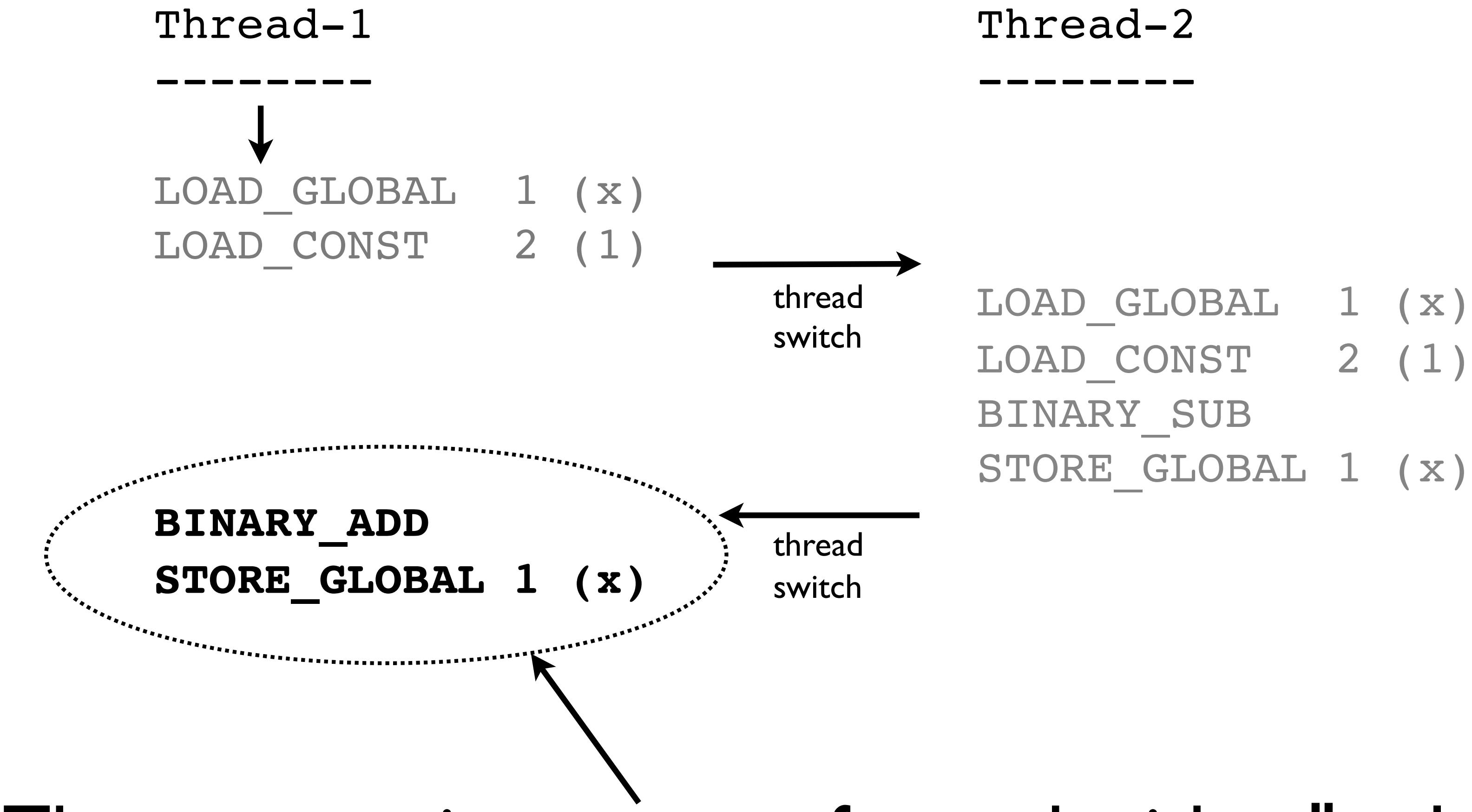


```
Thread-1                                Thread-2
--------                                --------
   |
   v
LOAD_GLOBAL  1 (x)
LOAD_CONST   2 (1)      ------->
                         thread         LOAD_GLOBAL  1 (x)
                         switch         LOAD_CONST   2 (1)
                                        BINARY_SUB
                                        STORE_GLOBAL 1 (x)
BINARY_ADD              <-------
STORE_GLOBAL 1 (x)       thread
                         switch
```

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Concurrent Updates

- Low level interpreter code



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

LOAD_GLOBAL  1 (x)
LOAD_CONST   2 (1)
                   thread
                   switch
                              LOAD_GLOBAL  1 (x)
                              LOAD_CONST   2 (1)
                              BINARY_SUB
                              STORE_GLOBAL 1 (x)
BINARY_ADD
STORE_GLOBAL 1 (x)
                   thread
                   switch
```

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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

Note: Raft is solving the problem of distributed shared state. It's well beyond your normal drunken bar fight.

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

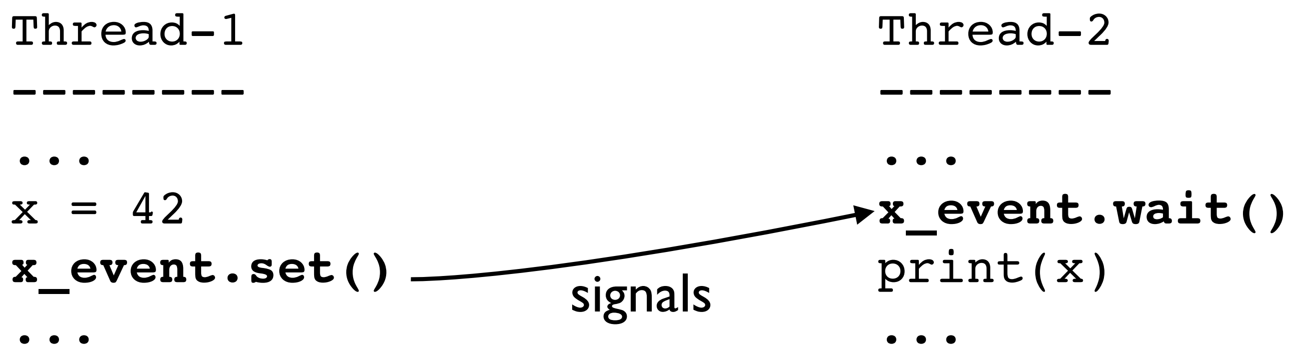
Thread Synchronization

- 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                     Thread-2
--------                     --------
...                          ...
x = 42                       x_event.wait()
x_event.set()                print(x)
...                          ...
```

- Caution : Events only have one-time use

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Exercise

- Implement a concurrent "Result"

```
class Result:
    def set_result(self, value):
        ...
        self.value = value
        ...
    def get_result(self):
        ...
        return self.value
```

- Allow this use:

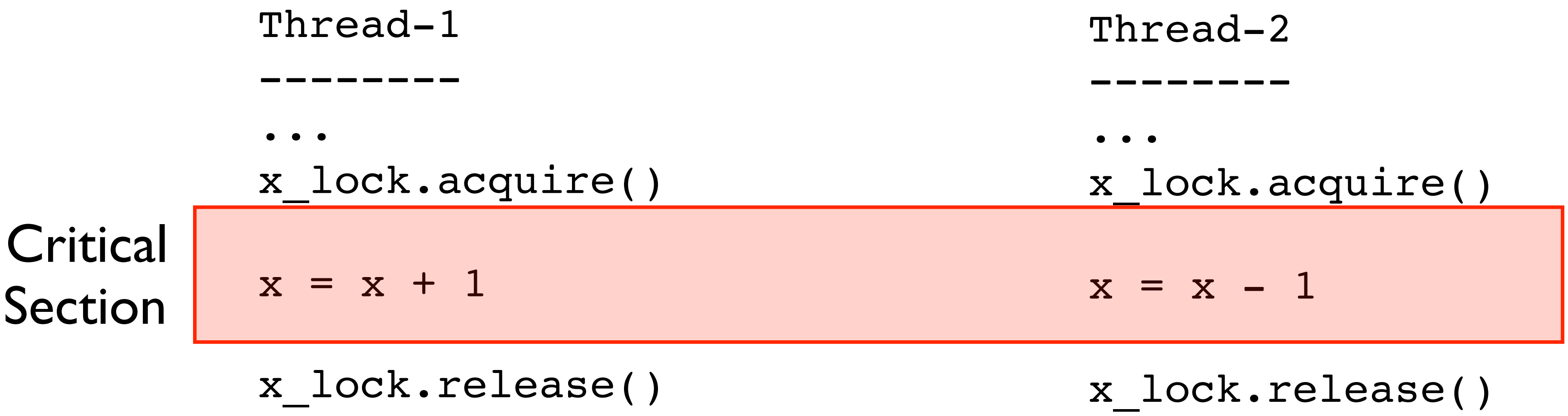
```
def func(x, y, result):
    time.sleep(10)
    result.set_result(x+y)

res = Result()
threading.Thread(target=func, args=(2,3,res)).start()
print("Answer:", res.get_result())
```

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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  x = x + 1                         x = x - 1
Section

          x_lock.release()                  x_lock.release()
```

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Exercise

- Modify the Key-Value server to only allow one transaction to occur at once
- Use mutex locks

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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()
```

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Exercise

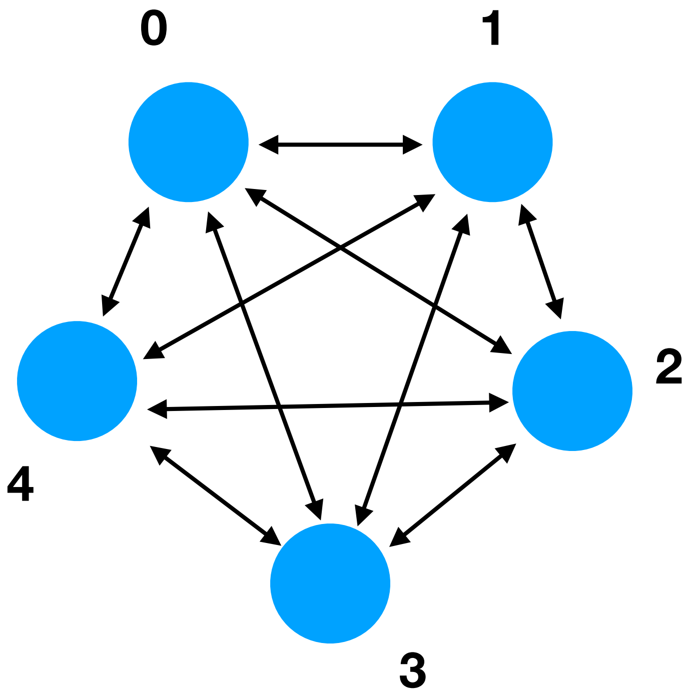
- Modify the Key-Value server to serialize all transactions using a queue instead of using locks
- Model it after the producer/consumer code
- Use the "Result" class created earlier.

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Raft Project

- Implement a messaging layer for a cluster
- Step 1: Addressing

```
# config.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).

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Raft Project

- Step 2: Implement a Networking Layer

```
# 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.recv()
```

- Have a mechanism where you can send a message to any server using its server number
- Have a way to receive a message (could be sent from anyone)

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Raft Project

- Important assumptions about Raft messaging
  - Message delivery is asynchronous (the sender does NOT wait for receiver to 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).

Part 2

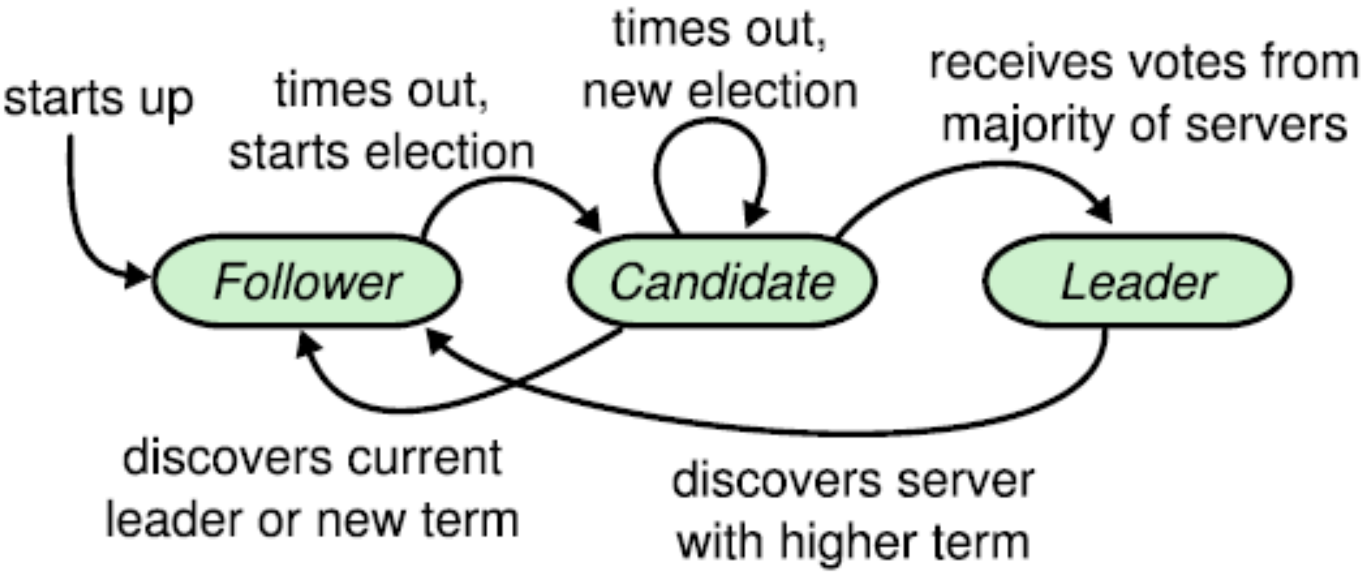
State Machines

(mathematical foundations)

Copyright (c) 2019, http://www.dabeaz.com

Raft Operational States

- Servers in Raft operate in different states


starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

- A major complexity concerns the transitions between the different states

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

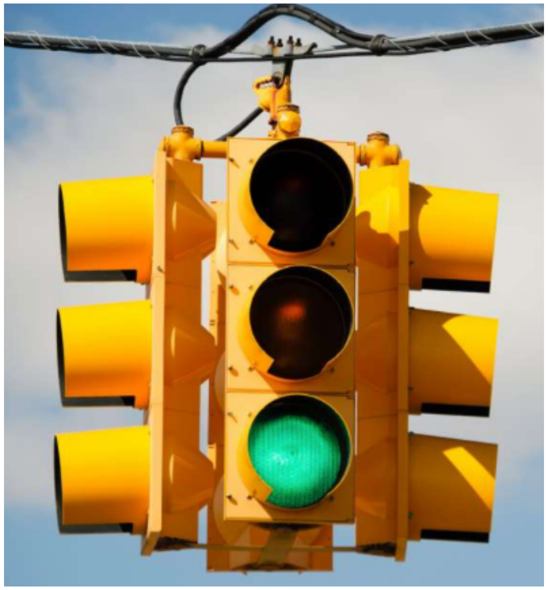
State Machines

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

A Simpler Example

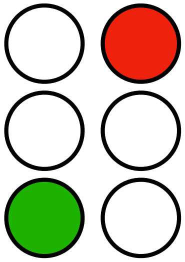
- A traffic light
- What are its operational states?

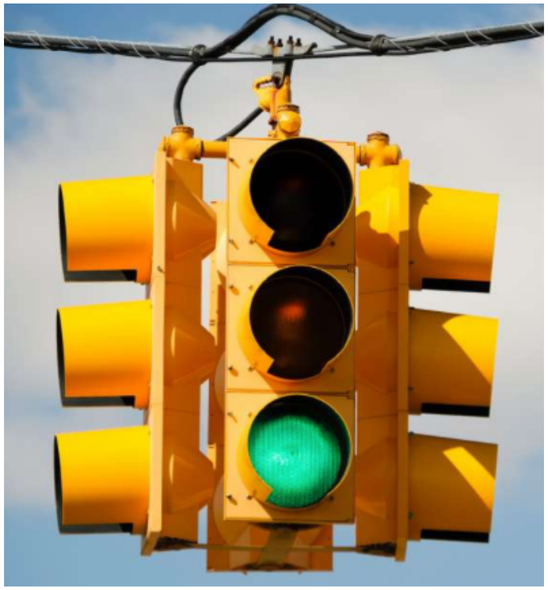


Copyright (C) 2019, David Beazley, http://www.dabeaz.com

A Simpler Example

- A traffic light
- What are its operational states?

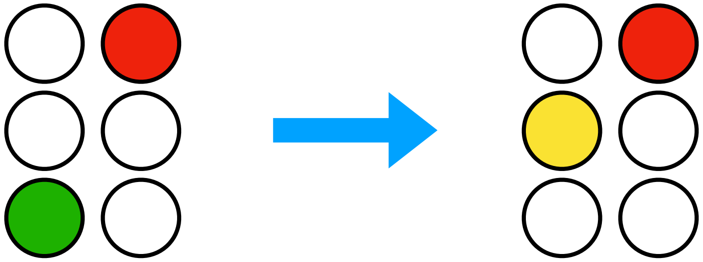


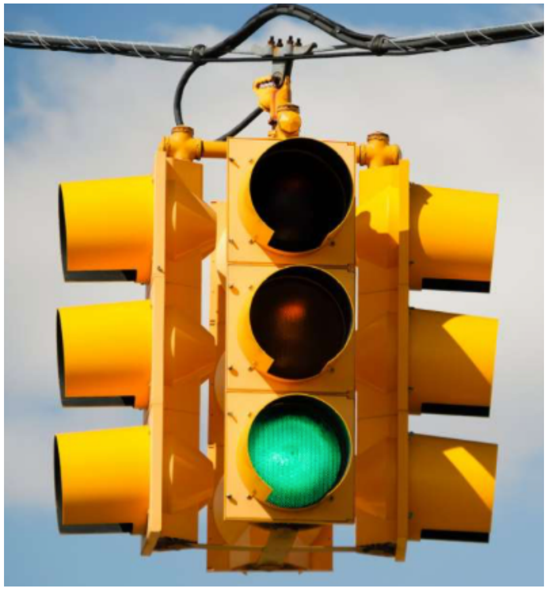


Copyright (C) 2019, David Beazley, http://www.dabeaz.com

A Simpler Example

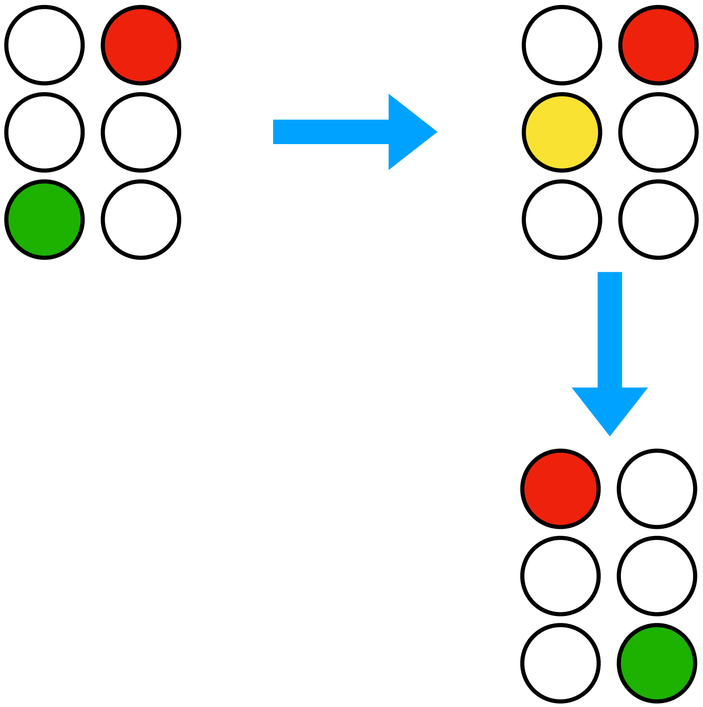
- A traffic light
- What are its operational states?

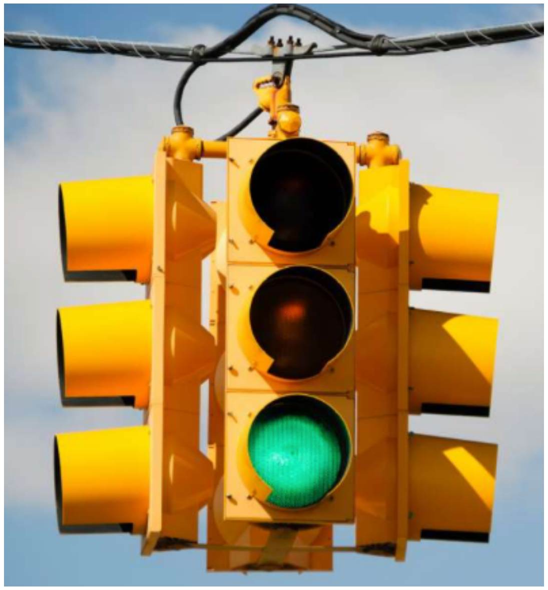




A Simpler Example

- A traffic light
- What are its operational states?

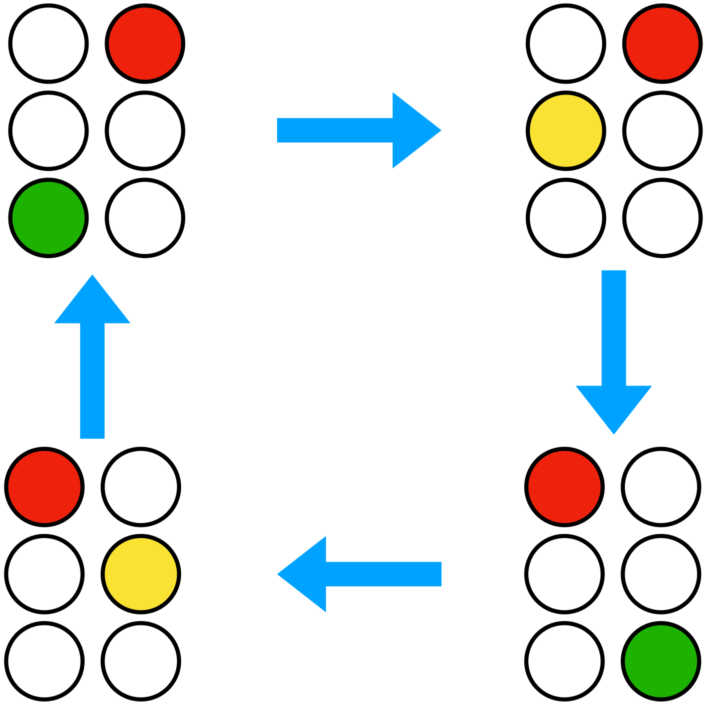


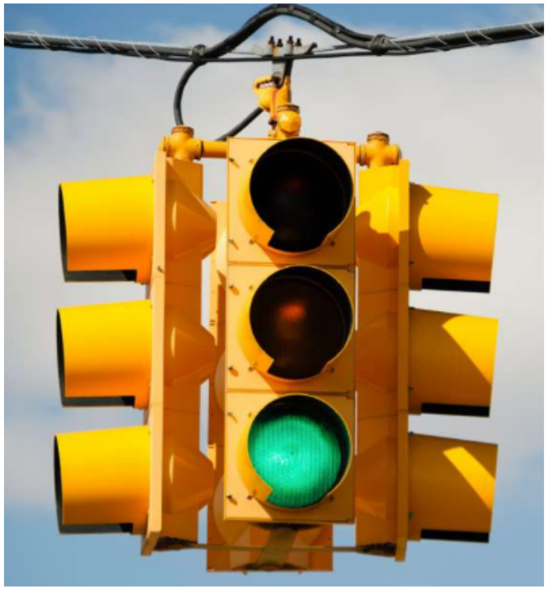


Copyright (C) 2019, David Beazley, http://www.dabeaz.com

A Simpler Example

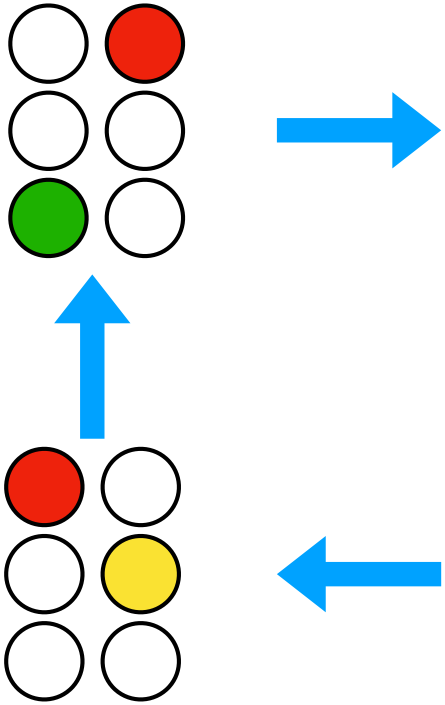
- A traffic light
- What are its operational states?

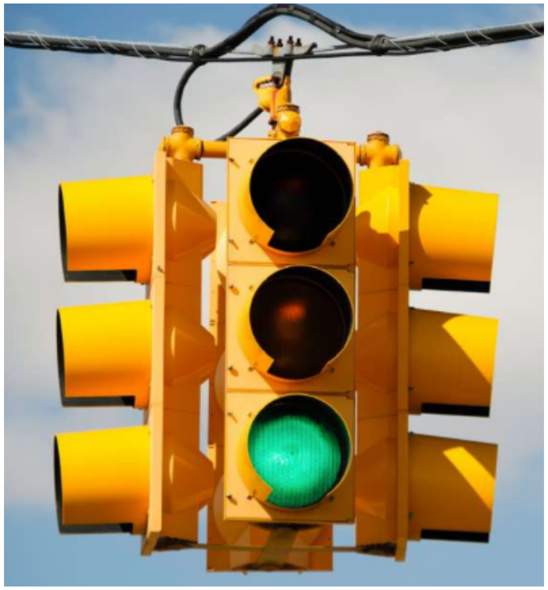




A Simpler Example

- A traffic light
- What are its operational states?



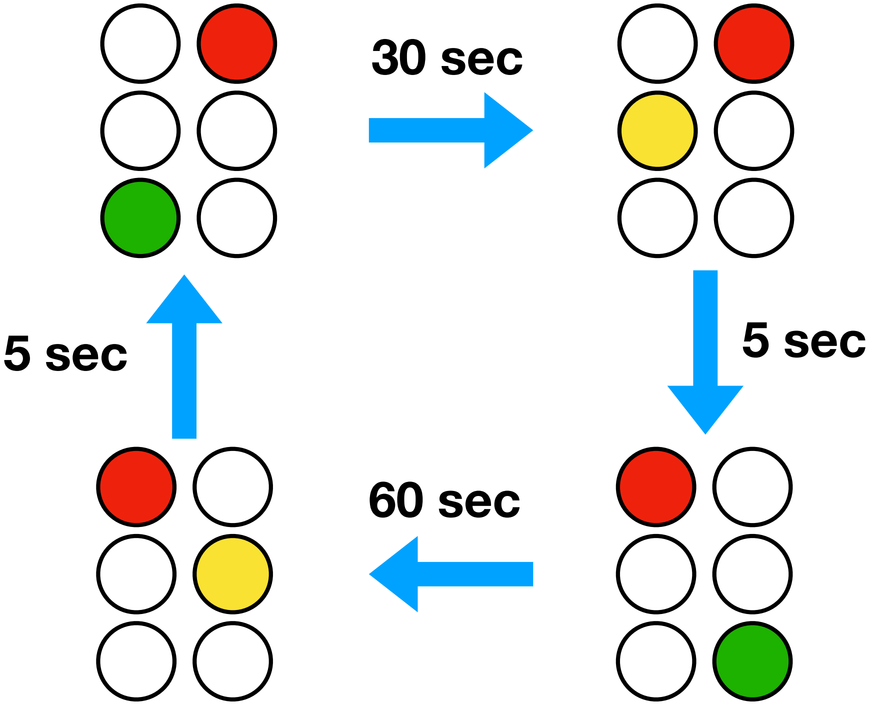


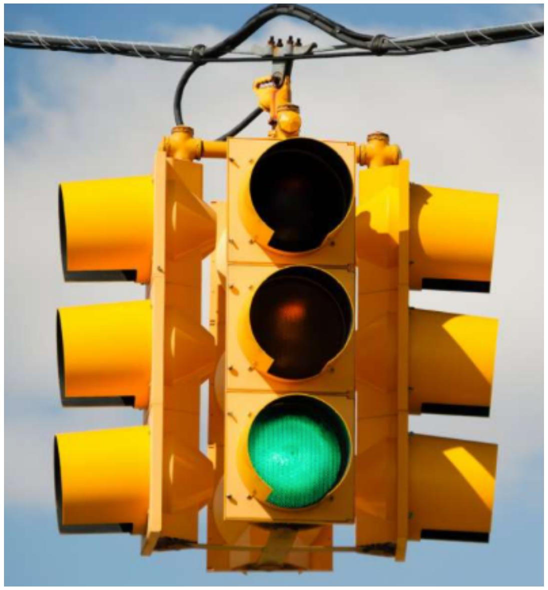
- What causes states to change?

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

A Simpler Example

- A traffic light
- What are its operational states?



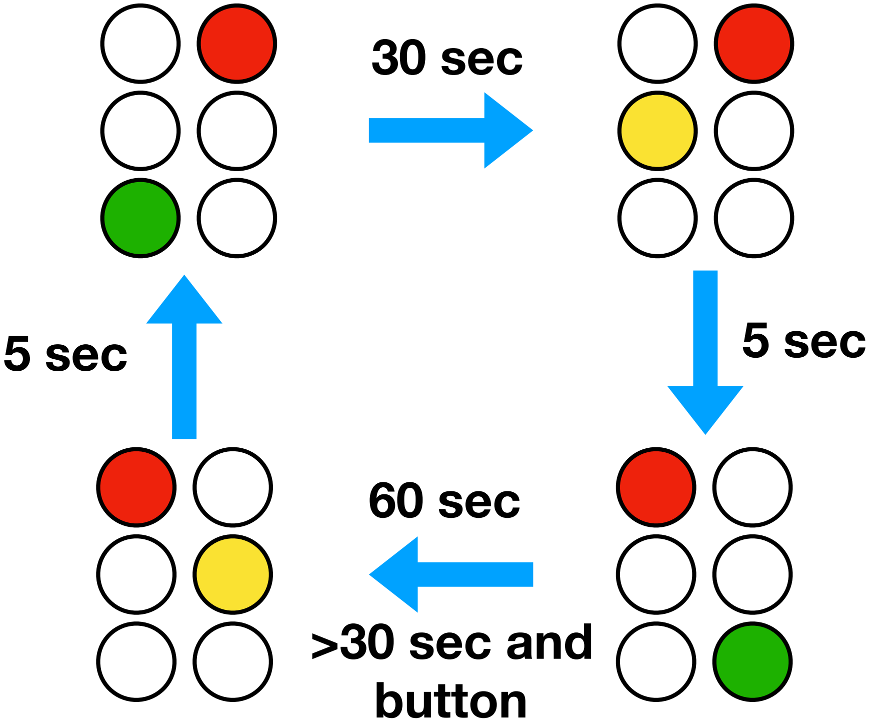


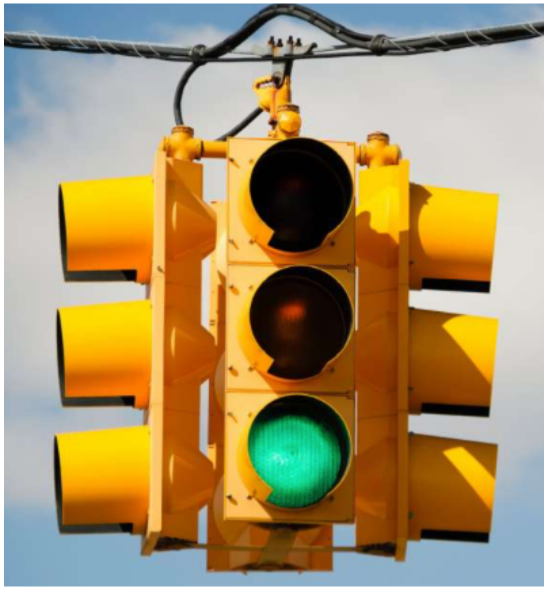
- What causes states to change? (events)

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

A Simpler Example

- A traffic light
- What are its operational states?



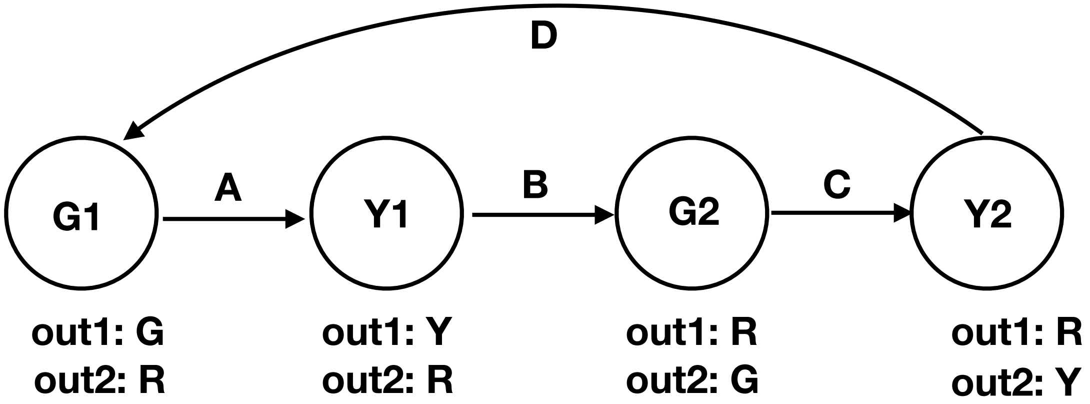


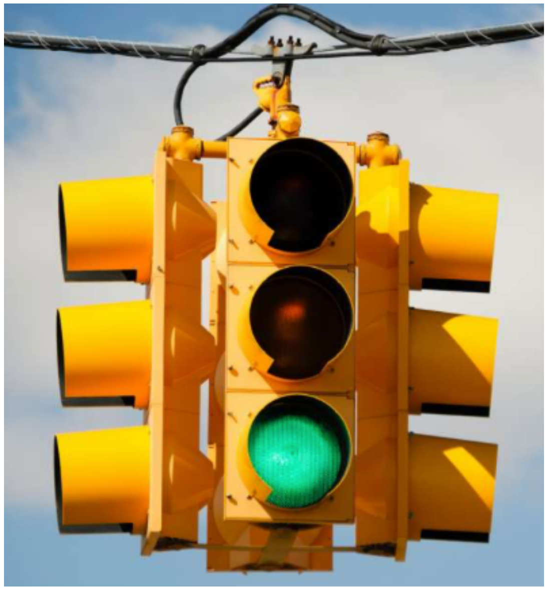
- What causes states to change? (events)

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Idea: Operational State

- Example: Traffic Light





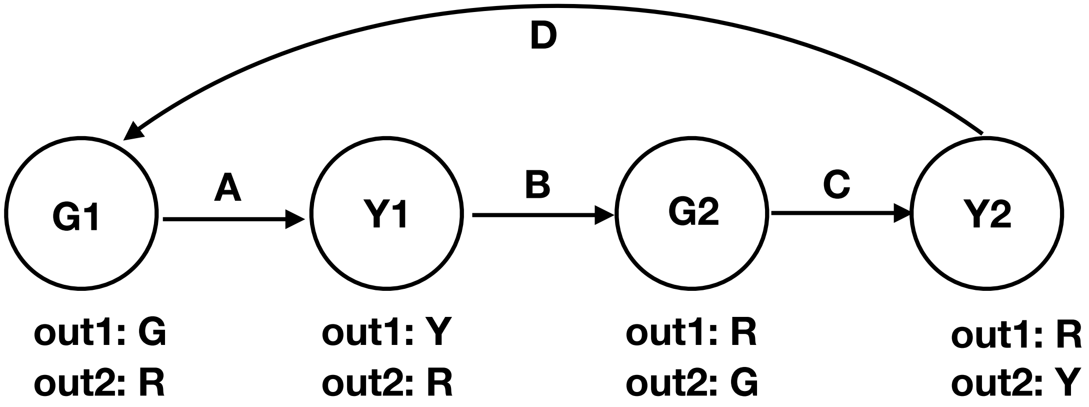
- States are named (circles)
- Arrows represent "events" (cause state change)

A: 30s timer
B: 5s timer
C: 60s timer or (walk_button and >= 30s timer)
D: 5s timer

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Exercise: Traffic Light

- Implement this state machine



D

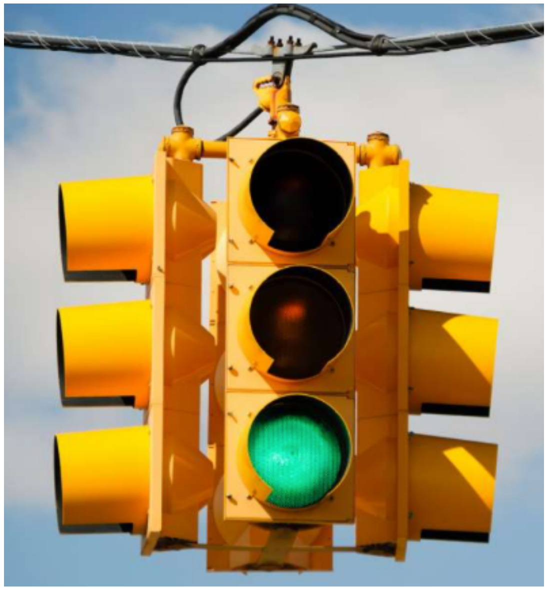
G1 A Y1 B G2 C Y2

out1: G
out2: R

out1: Y
out2: R

out1: R
out2: G

out1: R
out2: Y



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

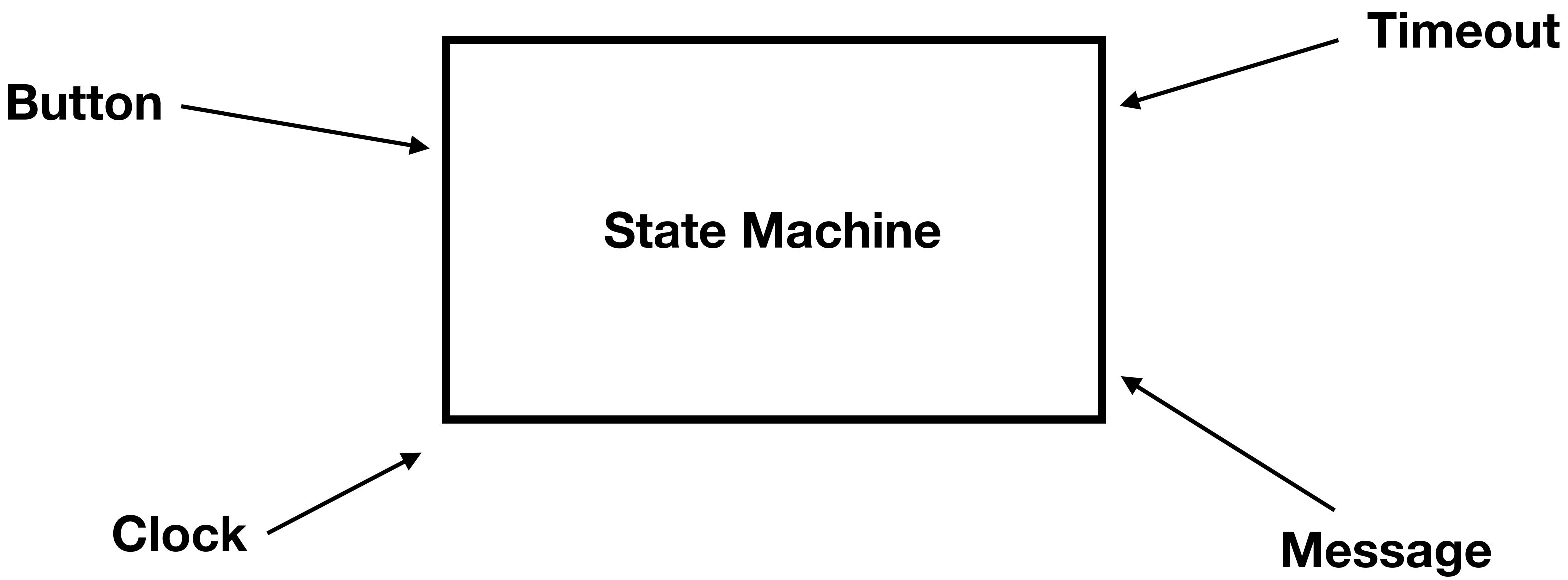
Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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?

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Issue: Time and Events



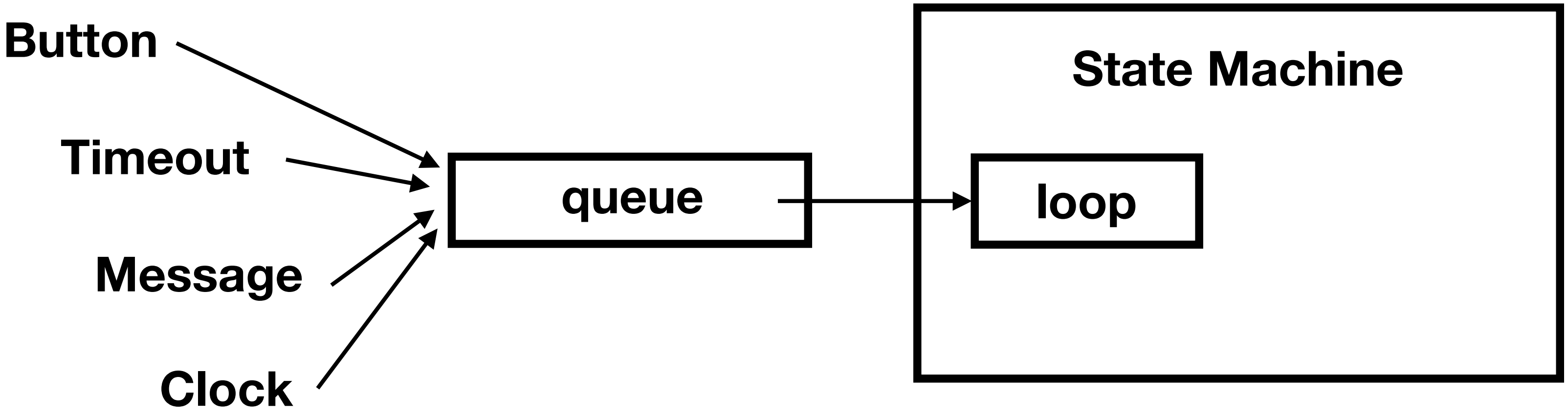
- One option: Callback functions/handlers

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

- Trigger the appropriate handler on event

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Event Serialization



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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

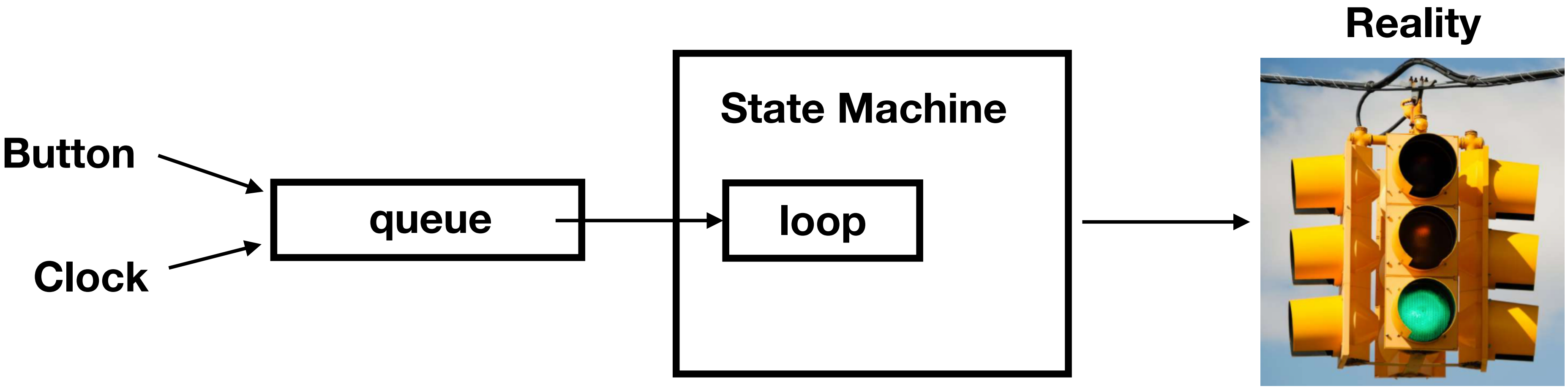
Exercise

- 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)

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Problem: Actions

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



Reality

State Machine

Button

Clock

queue

loop

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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 verify state machines

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Representing States

- A "state" refers to a collection of values

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

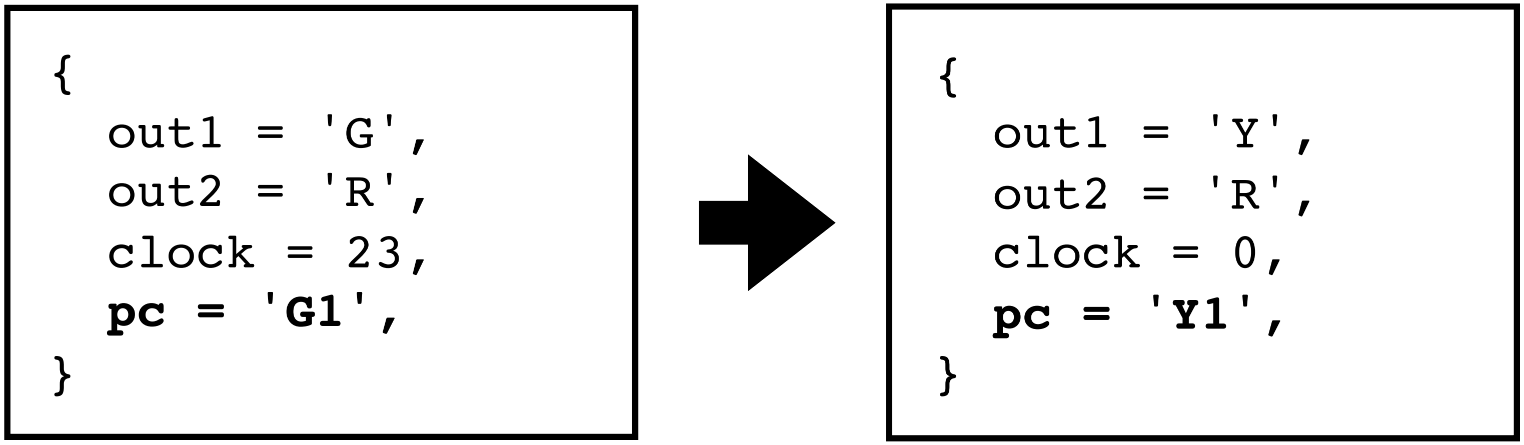
```
{
  out1 = 'Y',
  out2 = 'R',
  clock = 0,
}
```

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Control State

- There is control sequencing between states



```
{
  out1 = 'G',
  out2 = 'R',
  clock = 23,
  pc = 'G1',
}
```

```
{
  out1 = 'Y',
  out2 = 'R',
  clock = 0,
  pc = 'Y1',
}
```

- Control can be expressed by a "pc" variable
- Models a "program counter" on a CPU (the current instruction)

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

State Membership

- How to express state membership?

```
{
  out1 = 'G',
  out2 = 'R',
  clock = 23,
  pc = 'G1',
}
```

```
{
  out1 = 'Y',
  out2 = 'R',
  clock = 0,
  pc = 'Y1',
}
```

- Use a mathematical formula

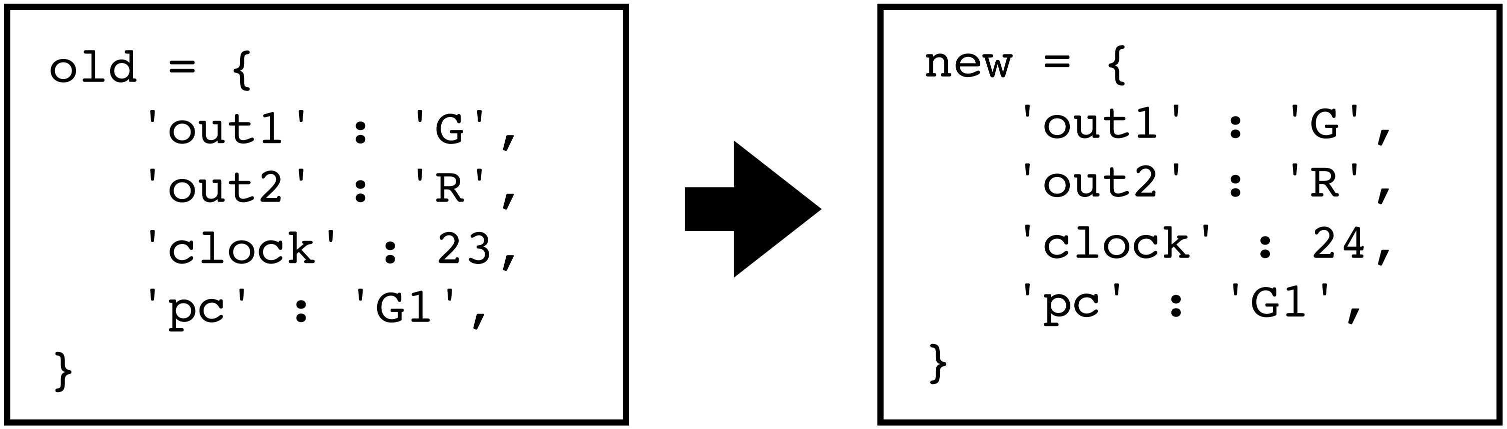
```
pc == 'G1' and clock < 30      pc == 'Y1' and clock < 5
```

- Thought: Do outputs determine membership?

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

State Changes

- A state change is an update to the values



```
old = {
    'out1' : 'G',
    'out2' : 'R',
    'clock' : 23,
    'pc' : 'G1',
}
```

```
new = {
    'out1' : 'G',
    'out2' : 'R',
    'clock' : 24,
    'pc' : 'G1',
}
```

- It's a function: dict -> dict

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

- Get all old values + updated values

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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.

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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!

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

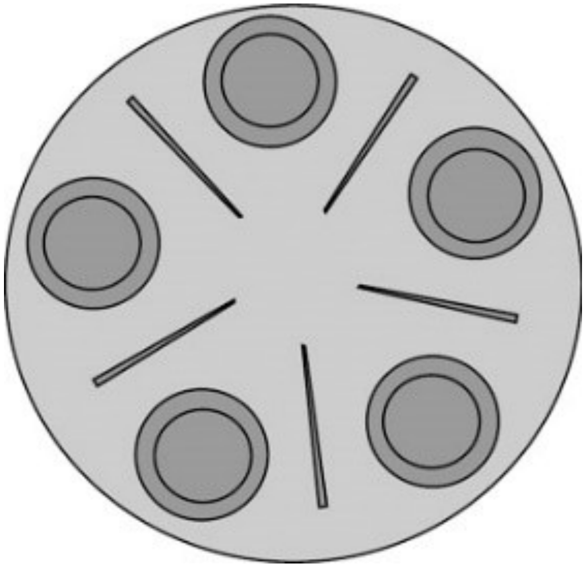
Group Exercise

- 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...

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Group Exercise

- Challenge: Can you simulate the dining philosophers problem?
  - 5 philosophers
  - 5 chopsticks
  - One stick at a time
  - Need two sticks to eat



Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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
- Slides that follow will be a "brief" intro

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

TLA+: Getting Started

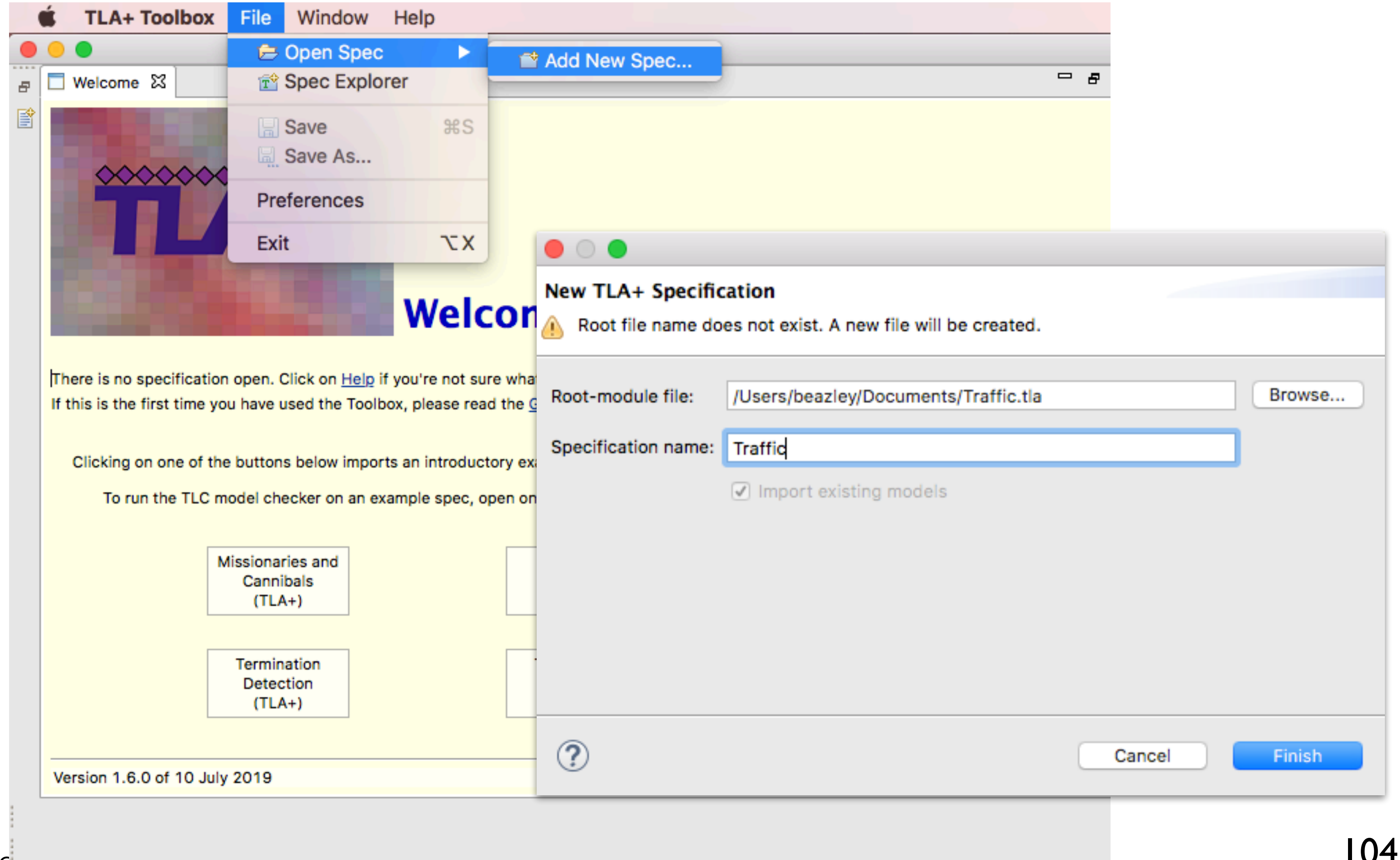
- Obtain the toolbox

  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.

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

TLA+: Creating a Spec



There is no specification open. Click on Help if you're not sure wha
If this is the first time you have used the Toolbox, please read the G

Clicking on one of the buttons below imports an introductory ex

To run the TLC model checker on an example spec, open on

Version 1.6.0 of 10 July 2019

New TLA+ Specification

⚠ Root file name does not exist. A new file will be created.

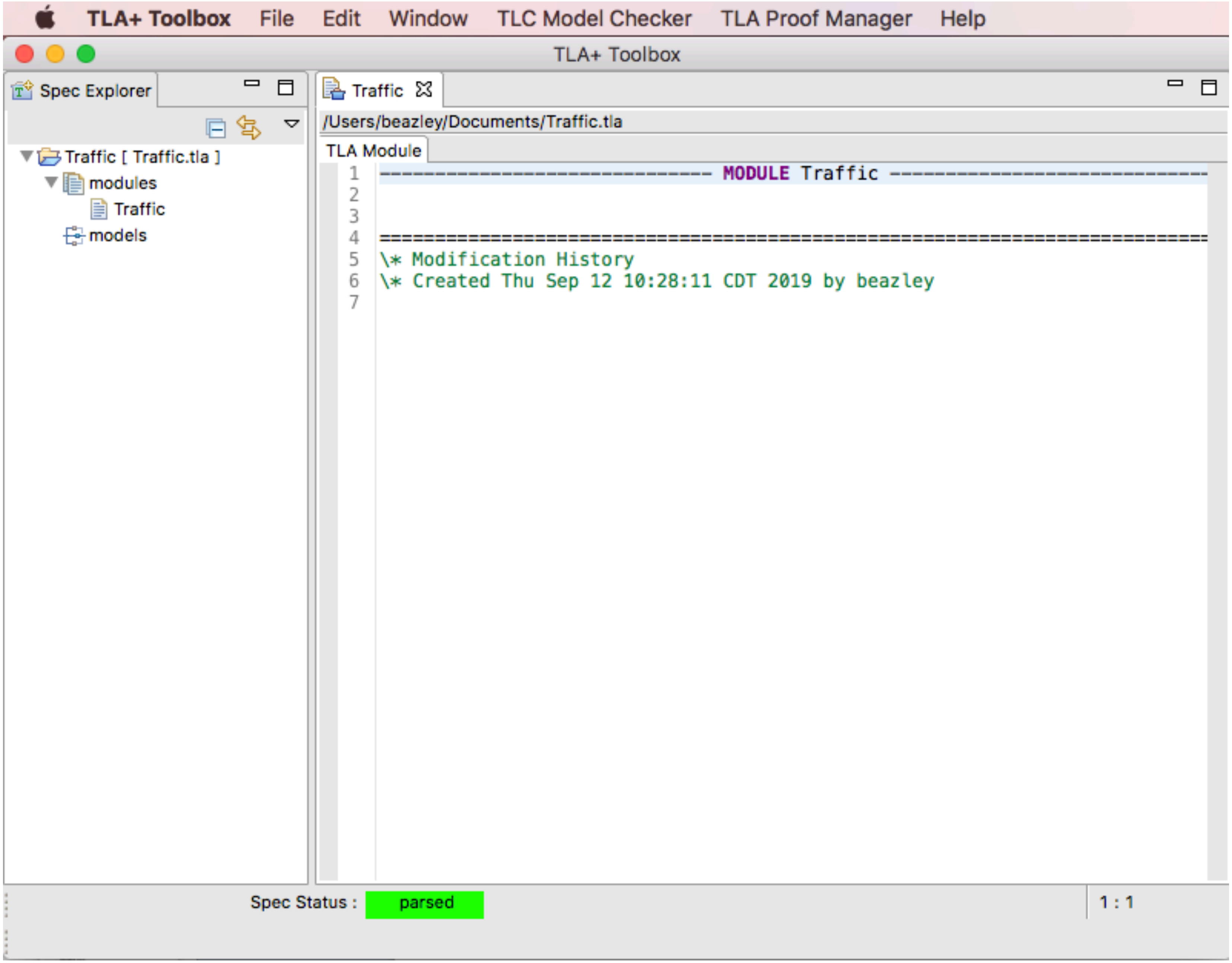
Root-module file: /Users/beazley/Documents/Traffic.tla

Specification name: Traffic

☑ Import existing models

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

TLA+: Starting Point



```
------------------------------ MODULE Traffic ------------------------------


=============================================================================
\* Modification History
\* Created Thu Sep 12 10:28:11 CDT 2019 by beazley
```

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

TLA+ Modules

- TLA+ defines modules

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


====================================================
```

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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
```

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Boolean Logic

- AND, OR, NOT operators

```
A /\ B        \* AND (∧)
A \/ B        \* OR (∨)
~A            \* NOT (¬)
```

- Grouping by indentation (implies parens)

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

- Same as

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

TLA+ State Variables

- State is held in designated variables

```
VARIABLES out1, out2, clock, pc
```

- Variables are initialized in a special definition

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

- This is the "start" state

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

TLA+ Next State

- Next state is expressed as a math formula

```
Next == \/ /\ pc = "G1"
           /\ clock < 30
           /\ clock' = clock + 1
           /\ UNCHANGED <<pc, out1, out2>>

        \/ /\ pc = "G1"
           /\ clock = 30
           /\ clock' = 0
           /\ out1' = "Y"
           /\ pc' = "Y1"
           /\ UNCHANGED <<out2>>

        ...
```

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

TLA+ Next State

- Next state is expressed as a math formula

```
Next == \/ /\ pc = "G1"
           /\ clock < 30
           /\ clock' = clock + 1
           /\ UNCHANGED <<pc, out1, out2>>

        \/ /\ pc = "G1"
           /\ clock = 30
           /\ clock' = 0
           /\ out1' = "Y"
           /\ pc' = "Y1"
           /\ UNCHANGED <<out2>>

        ...
```

- Each "state" is expressed as a group

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

TLA+ Next State

- Next state is expressed as a math formula

```
Next == \/ /\ pc = "G1"
           /\ clock < 30
           /\ clock' = clock + 1
           /\ UNCHANGED <<pc, out1, out2>>

        \/ /\ pc = "G1"
           /\ clock = 30
           /\ clock' = 0
           /\ out1' = "Y"
           /\ pc' = "Y1"
           /\ UNCHANGED <<out2>>

        ...
```

- Conditions to determine state membership

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

TLA+ Next State

- Next state is expressed as a math formula

```
Next == \/ /\ pc = "G1"
           /\ clock < 30
           /\ clock' = clock + 1
           /\ UNCHANGED <<pc, out1, out2>>

        \/ /\ pc = "G1"
           /\ clock = 30
           /\ clock' = 0
           /\ out1' = "Y"
           /\ pc' = "Y1"
           /\ UNCHANGED <<out2>>

        ...
```

- State changes written: *var*' = *expression*

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

TLA+ Next State

- Next state is expressed as a math formula

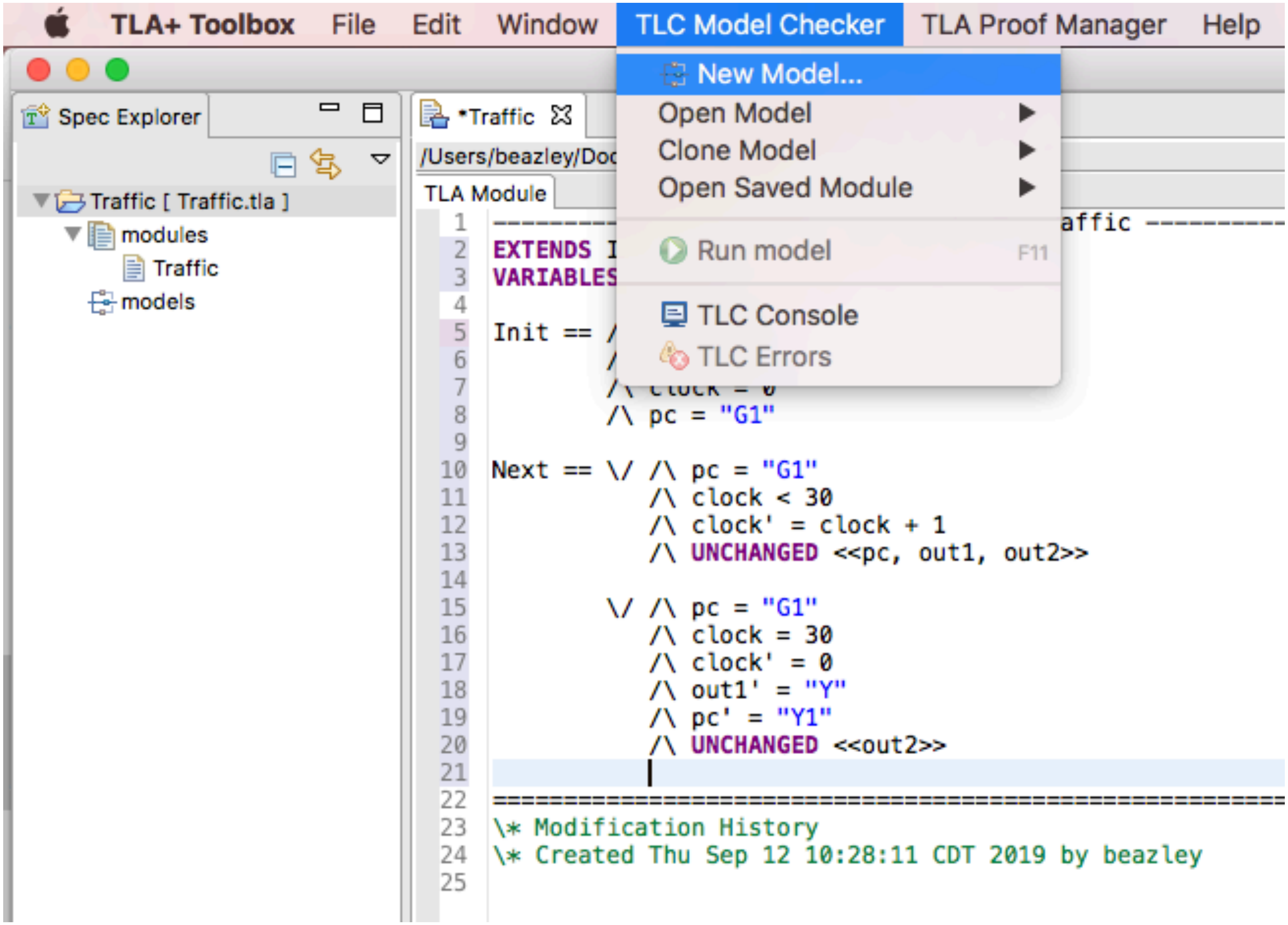
```
Next == \/ /\ pc = "G1"
           /\ clock < 30
           /\ clock' = clock + 1
           /\ UNCHANGED <<pc, out1, out2>>

        \/ /\ pc = "G1"
           /\ clock = 30
           /\ clock' = 0
           /\ out1' = "Y"
           /\ pc' = "Y1"
           /\ UNCHANGED <<out2>>

        ...
```

- ALL variables must be accounted for

- Use UNCHANGED << ... >>

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

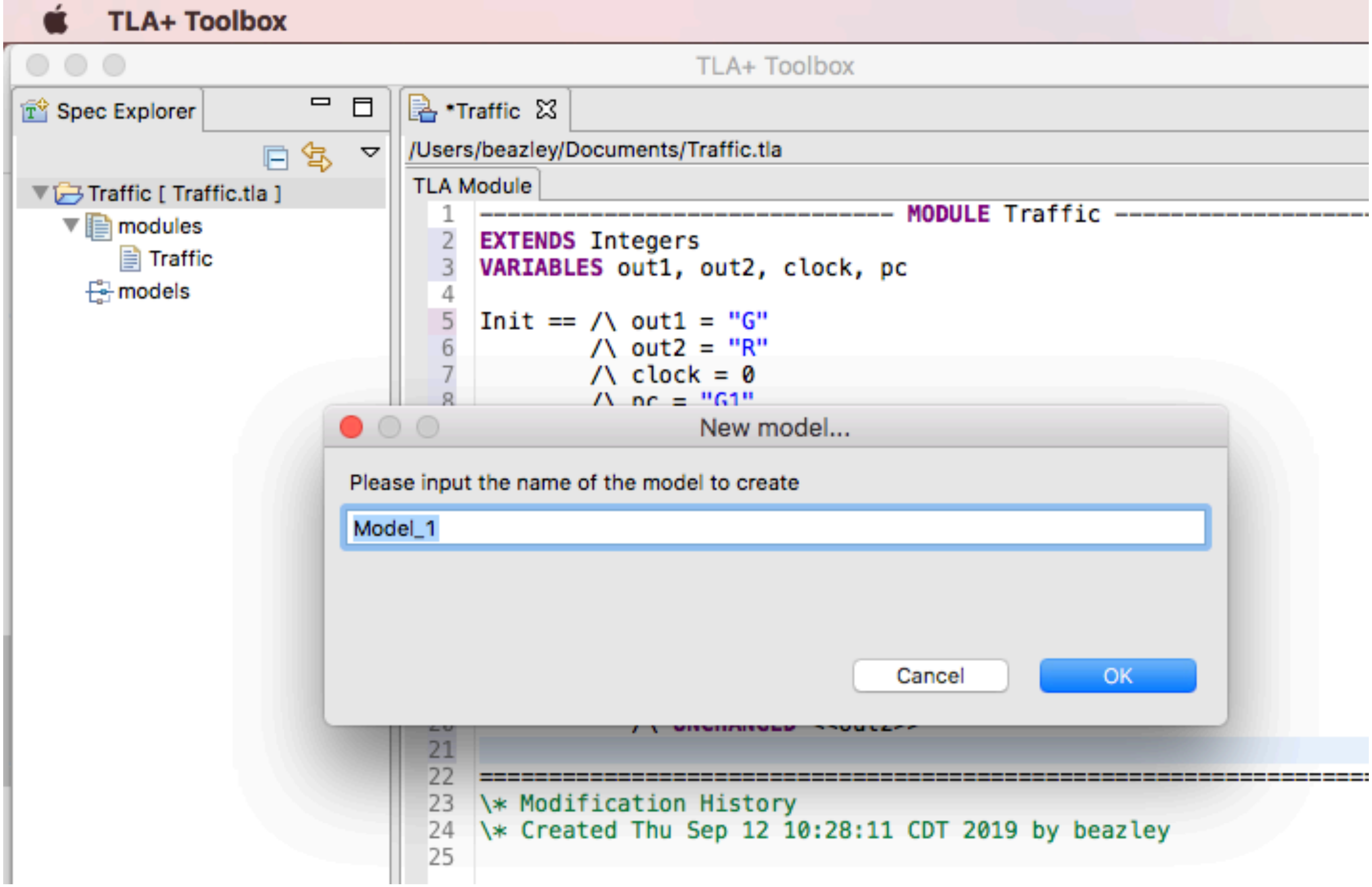
Model Checking



```
---------                                 affic ----------
EXTENDS I
VARIABLES

Init == /
        /
        /\ clock = 0
        /\ pc = "G1"

Next == \/ /\ pc = "G1"
           /\ clock < 30
           /\ clock' = clock + 1
           /\ UNCHANGED <<pc, out1, out2>>

        \/ /\ pc = "G1"
           /\ clock = 30
           /\ clock' = 0
           /\ out1' = "Y"
           /\ pc' = "Y1"
           /\ UNCHANGED <<out2>>

==========================================================
\* Modification History
\* Created Thu Sep 12 10:28:11 CDT 2019 by beazley
```

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Model Checking



```
------------------------------ MODULE Traffic -------------------
EXTENDS Integers
VARIABLES out1, out2, clock, pc

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

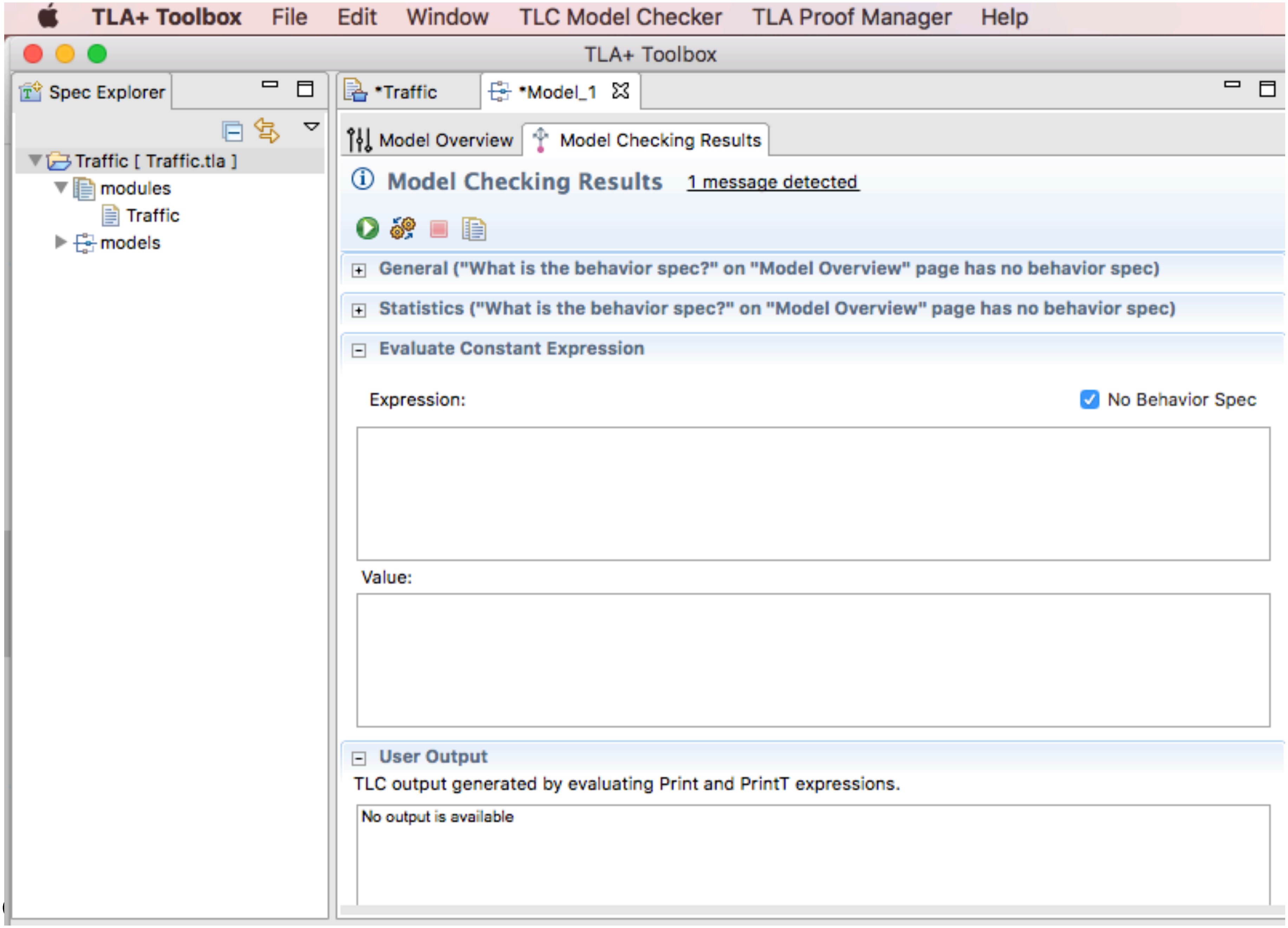
Please input the name of the model to create

Model_1

```
               /\ UNCHANGED <<out2>>

=================================================================
\* Modification History
\* Created Thu Sep 12 10:28:11 CDT 2019 by beazley

```

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Model Checking



Model Checking Results 1 message detected

General ("What is the behavior spec?" on "Model Overview" page has no behavior spec)

Statistics ("What is the behavior spec?" on "Model Overview" page has no behavior spec)

Evaluate Constant Expression

Expression: ☑ No Behavior Spec

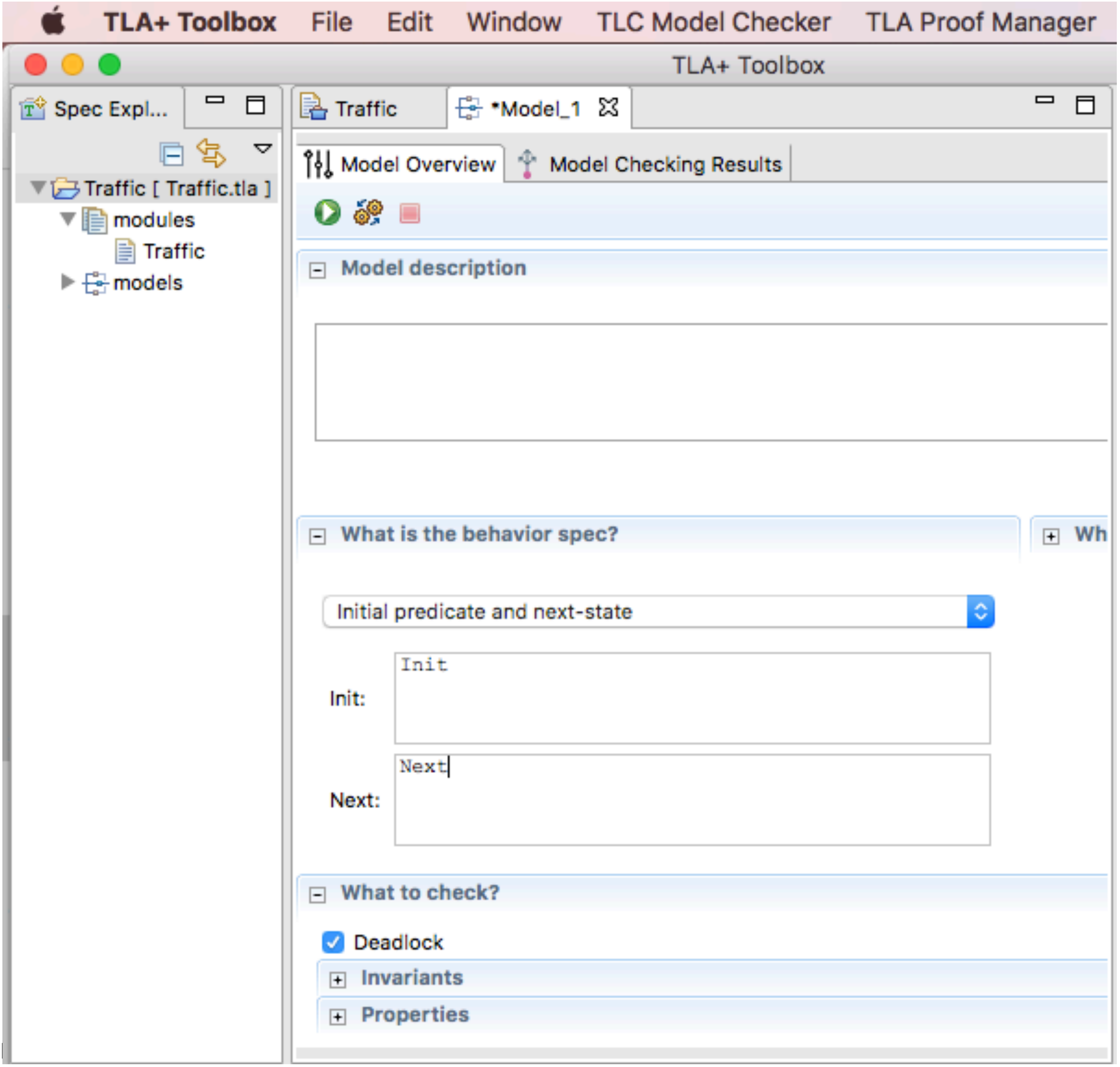
Value:

User Output

TLC output generated by evaluating Print and PrintT expressions.

No output is available

Model Checking



What is the behavior spec?

Initial predicate and next-state

Init: Init

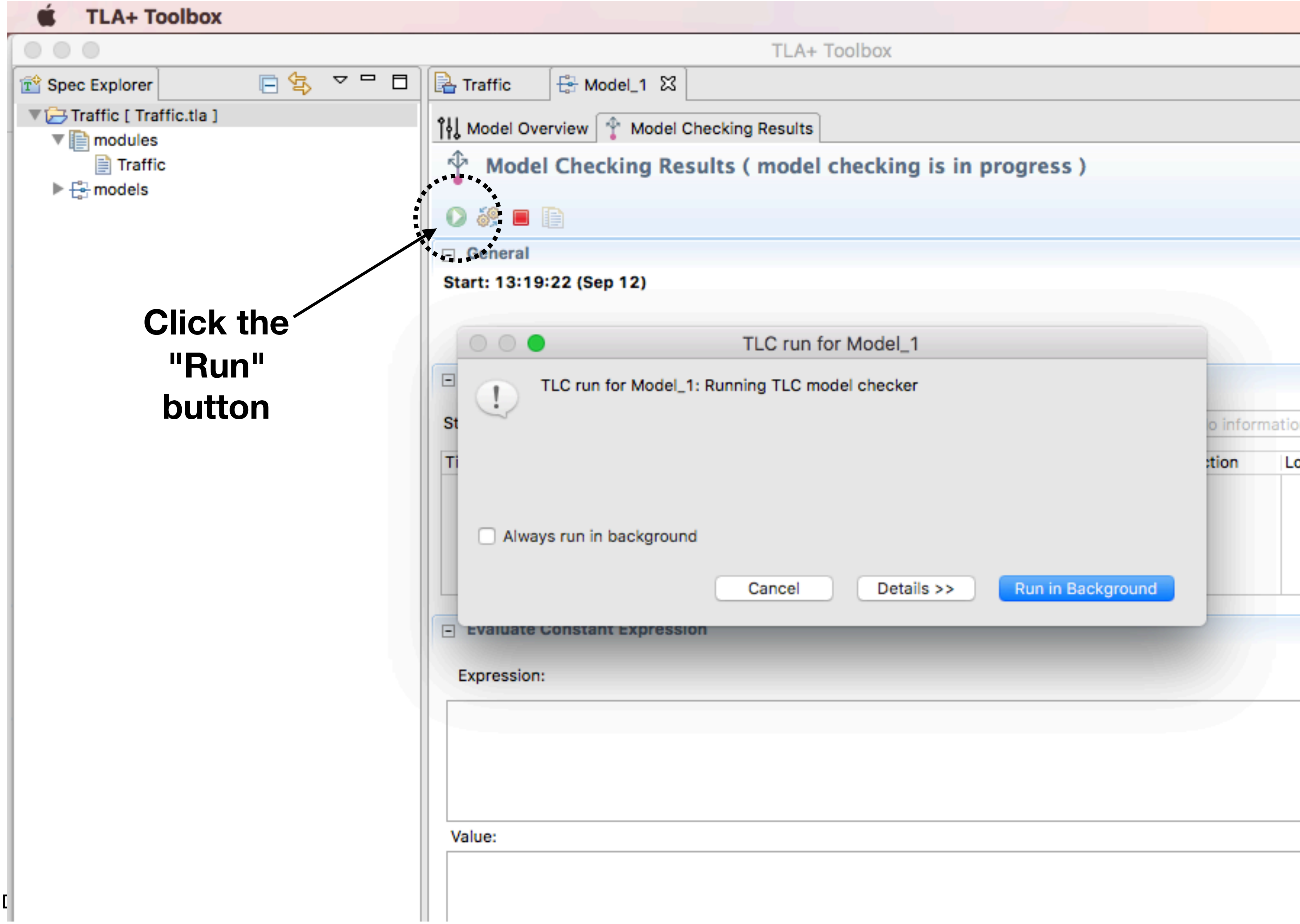
Next: Next

What to check?

☑ Deadlock

Set the Init and Next state definitions

Model Checking



Model Checking Results (model checking is in progress)

Start: 13:19:22 (Sep 12)

TLC run for Model_1

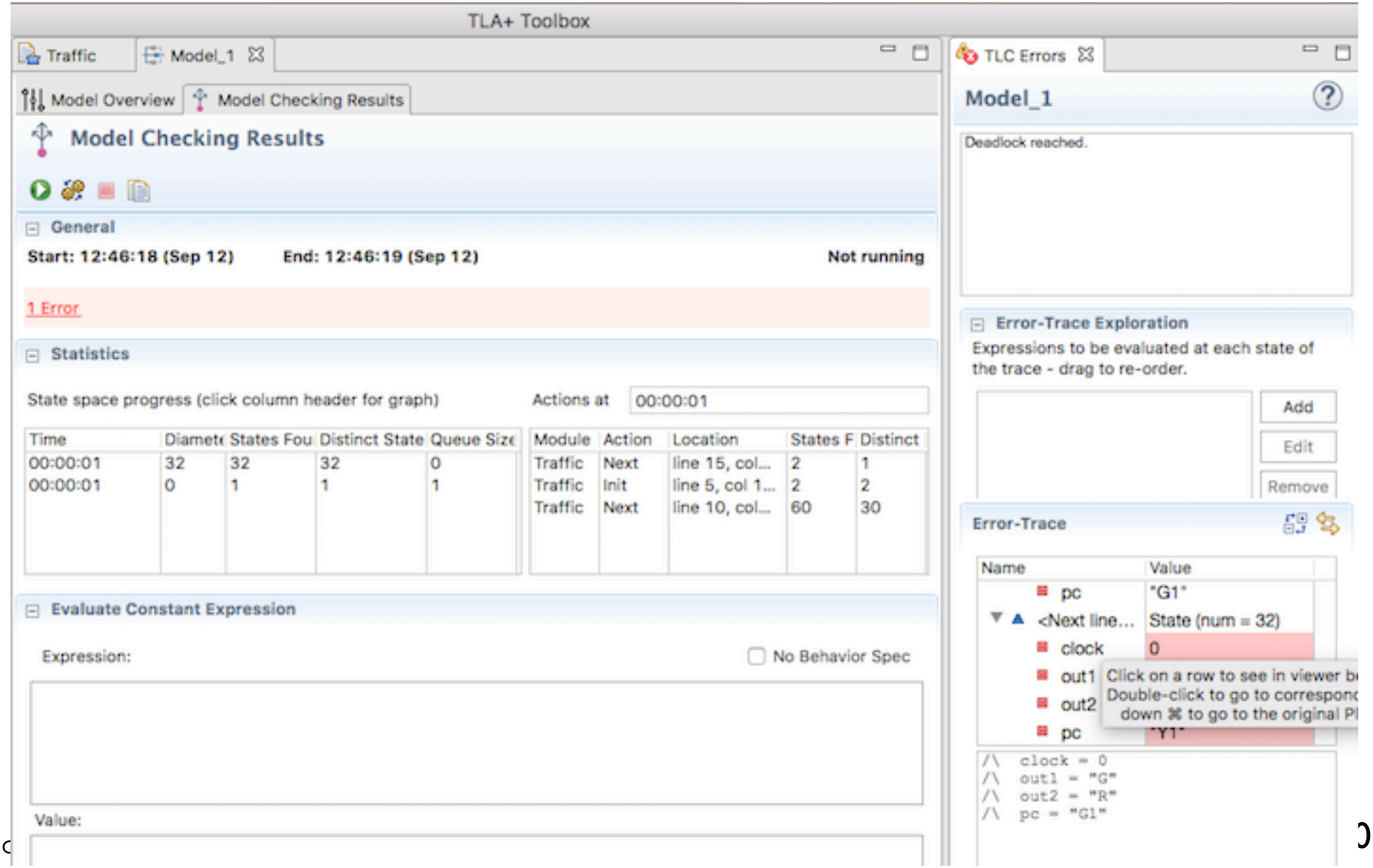
TLC run for Model_1: Running TLC model checker

□ Always run in background

Click the "Run" button

Copyright (C) 2019, [

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)

Time	Diamete	States Fou	Distinct State	Queue Size
00:00:01	32	32	32	0
00:00:01	0	1	1	1

Actions at 00:00:01

Module	Action	Location	States F	Distinct
Traffic	Next	line 15, col...	2	1
Traffic	Init	line 5, col 1...	2	2
Traffic	Next	line 10, col...	60	30

Evaluate Constant Expression

Expression: □ No Behavior Spec

Value:

Model_1

Deadlock reached.

Error-Trace Exploration

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

Error-Trace

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

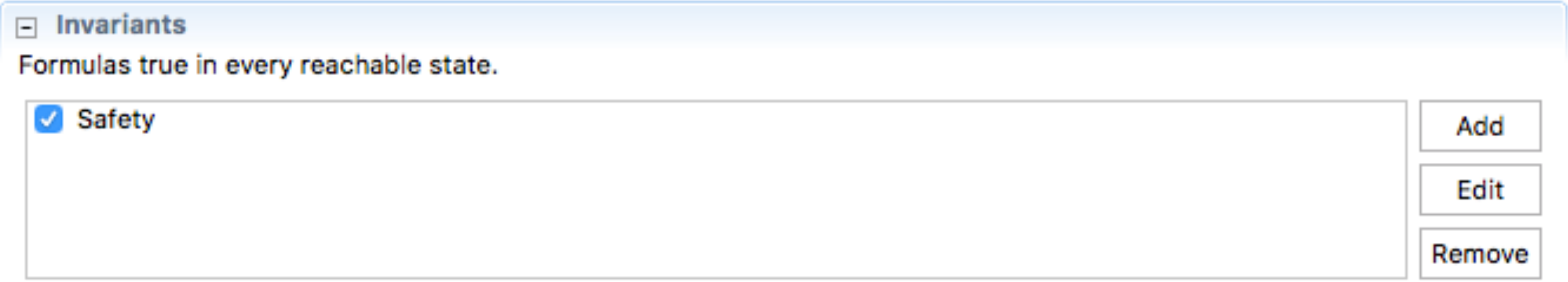
Click on a row to see in viewer b
Double-click to go to correspond
down ⌘ to go to the original Pl

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

Invariants

- TLA+ can also encode invariants
- Conditions that hold in all states
- Example: Both traffic lights can't be green at same time.

```
Safety == ~(out1 = "G" /\ out2 = "G")
```



⊟ Invariants

Formulas true in every reachable state.

☑ Safety

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Exercise

- Finish the traffic light spec
- Figure out how to simulate a button press

Sets

- TLA has sets and set operations

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

x \in values                  \* Membership test
\A x \in values: x > 0        \* ∀x ∈ values: x > 0
\E x \in values: x > 3        \* ∃x ∈ 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 }
```

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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"]
```

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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.

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Exercise

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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

```
Next == A \/ B \/ C \/ D
```

- TLA+ explores all possible branches

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Big Picture

- You might see a spec like this:

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

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

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

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?

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Part 3

A Few Design Thoughts

Copyright (c) 2019, http://www.dabeaz.com

The Big Problem

- Focus on the log, the log, the LOG.
- Make sure you understand the LOG.
- Not that log, THE LOG.
- Focus.... focus... focus... ON. THE. LOG.

The LOG

- Define THE LOG as a stand-alone object

```
class TheLog:
    def append_entries(self, index, prevTerm, entries):
        ...
    # Other methods (as applicable)
    def __len__(self):
        ...

    def __getitem__(self, index):
        ...
```

- Make sure you understand the semantics of the log and append_entries in particular.
- Write tests for THE LOG. Keep it working.

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

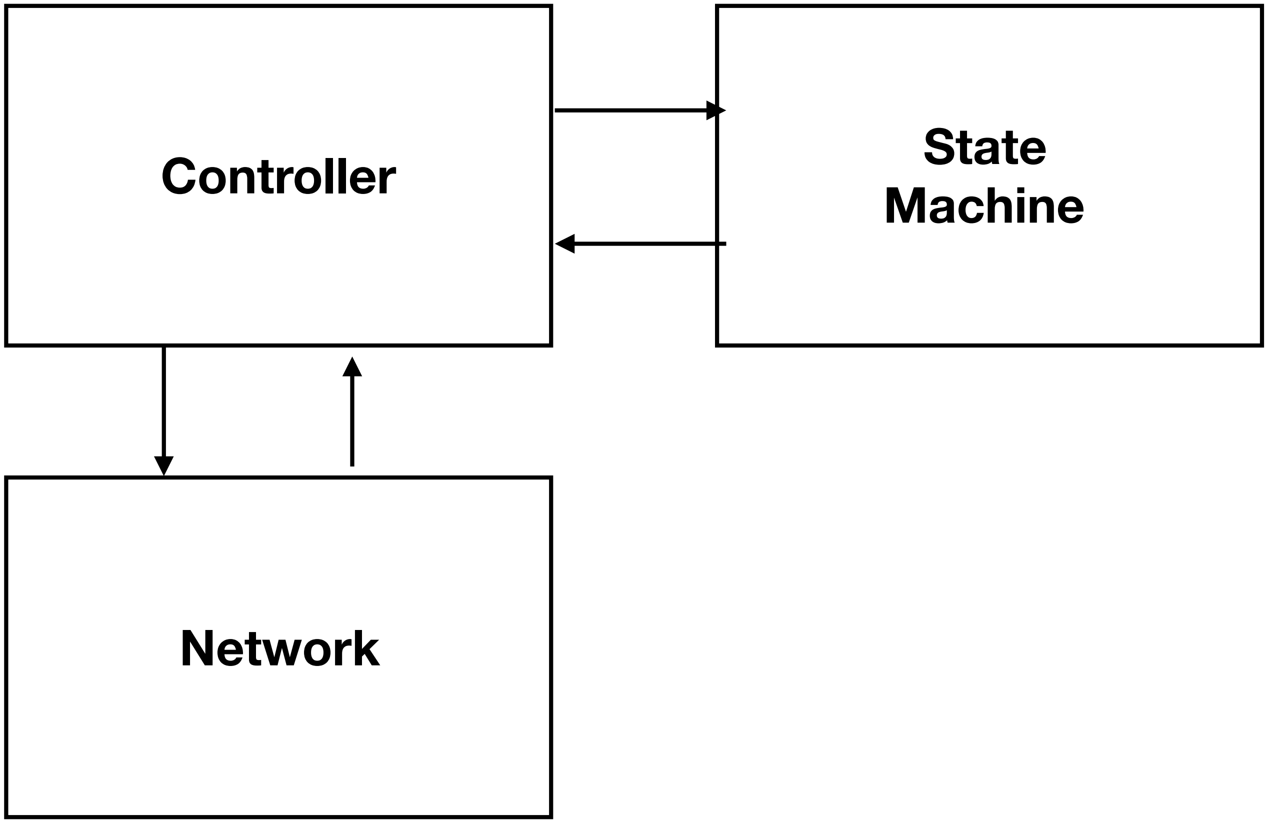
Managing Complexity

- There is a lot of inherent complexity in Raft
  - Multiple servers
  - Networking
  - Threads
  - State machines
- Testing/verifying it is very difficult

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Decoupling

- Work to decouple things into well-defined parts



Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Data Abstraction

- Program to interfaces, not implementation

```
class RaftNet:
    def recv(self):
        # Receive a message
        ...
    def send(self, dest, msg):
        # Send a message
        ...

class ThreadRaftNet(RaftNet):
     ...

class TCPRaftNet(RaftNet):
     ...
```

- Make it easy to redefine the implementation

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Prefer Composition

- Have components use other components

```
class Controller:
    def __init__(self, net: RaftNet):
        self.net = net

    def do_something(self):
        ...
        self.net.send(dest, msg)
        ...
```

- Focus on glue that holds components together
- Allow the parts to be changed
- Support Dependency Injection

Make it Debuggable

- Allow for debug-logging
- Take advantage of the REPL (allow for live-interaction with a running system)
- Controversial opinion: Making it debuggable might be MORE useful than focusing on exhaustive test coverage.   It is extremely difficult to write tests for all possible system states (part of motivation for TLA+).

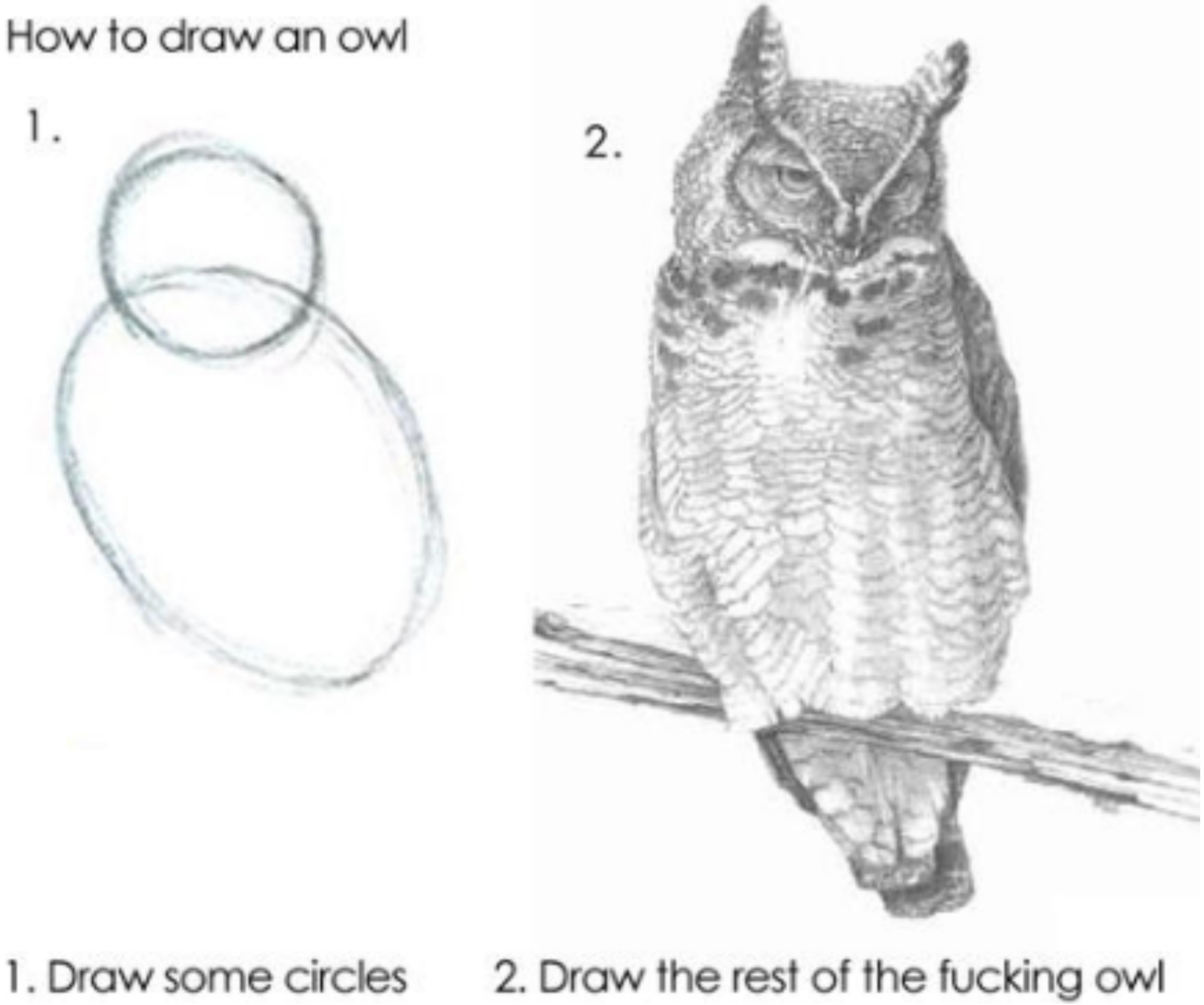
Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Let it Fail

- Don't be too defensive on error handling
- Errors should be loudly reported
- Handling of failure is an inherent part of Raft
- Might be easier to just restart a server than to program the server to gracefully recover from every possible fault

Copyright (C) 2019, David Beazley, http://www.dabeaz.com

Project: Implement Raft



Copyright (C) 2019, David Beazley, http://www.dabeaz.com