Distributed Systems

Spring Semester 2020

Lecture 24: Ray

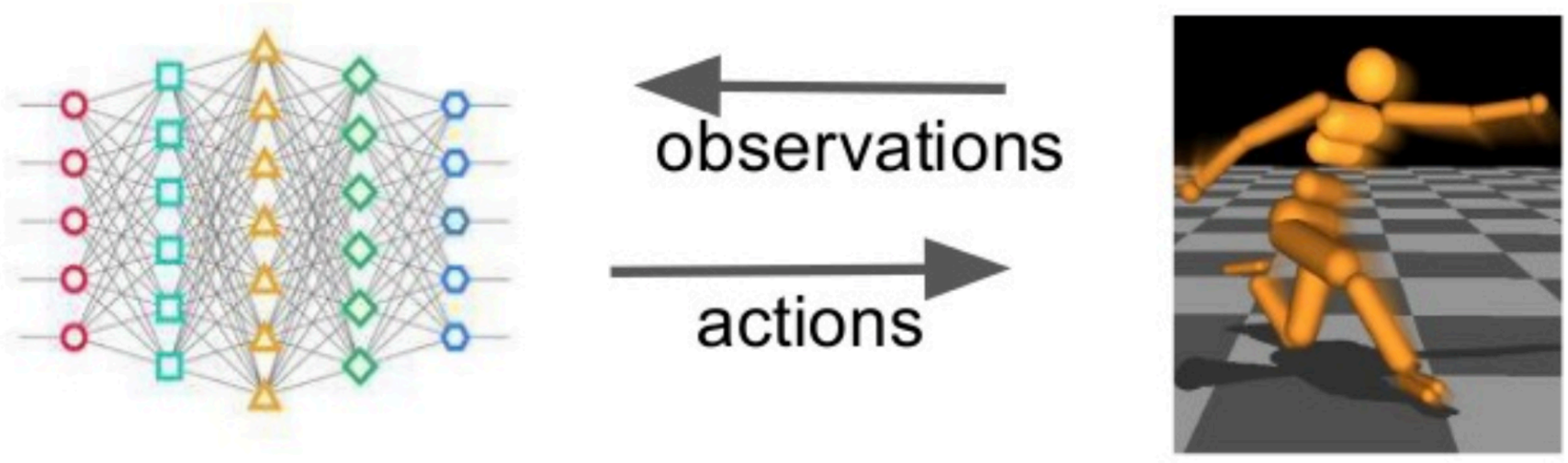
John Liagouris
liagos@bu.edu

Why this paper

- A general-purpose cluster computing framework
- Dynamic dataflow graphs
  - Needed for emerging AI applications
- Interesting programming model
  - Tasks — stateless functions
  - Actors — stateful computation
  - Futures — for communication between threads

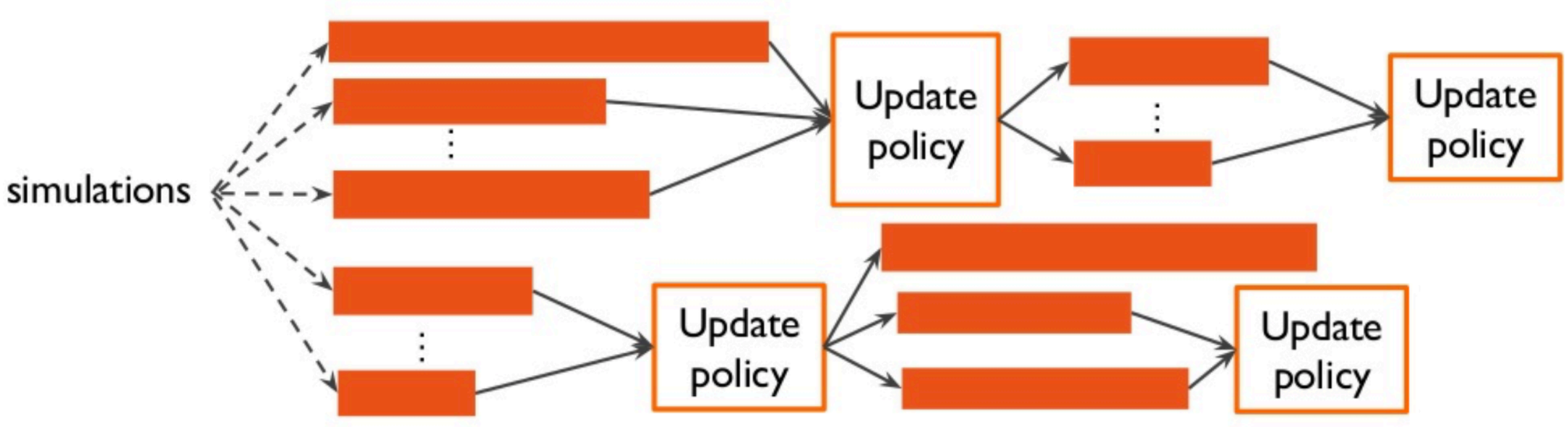
Reinforcement Learning

- Process inputs from many sensors in parallel and in real-time
- Execute large number of simulations — 100s of millions
- Result of simulations and/or interaction with the environment can change the next step

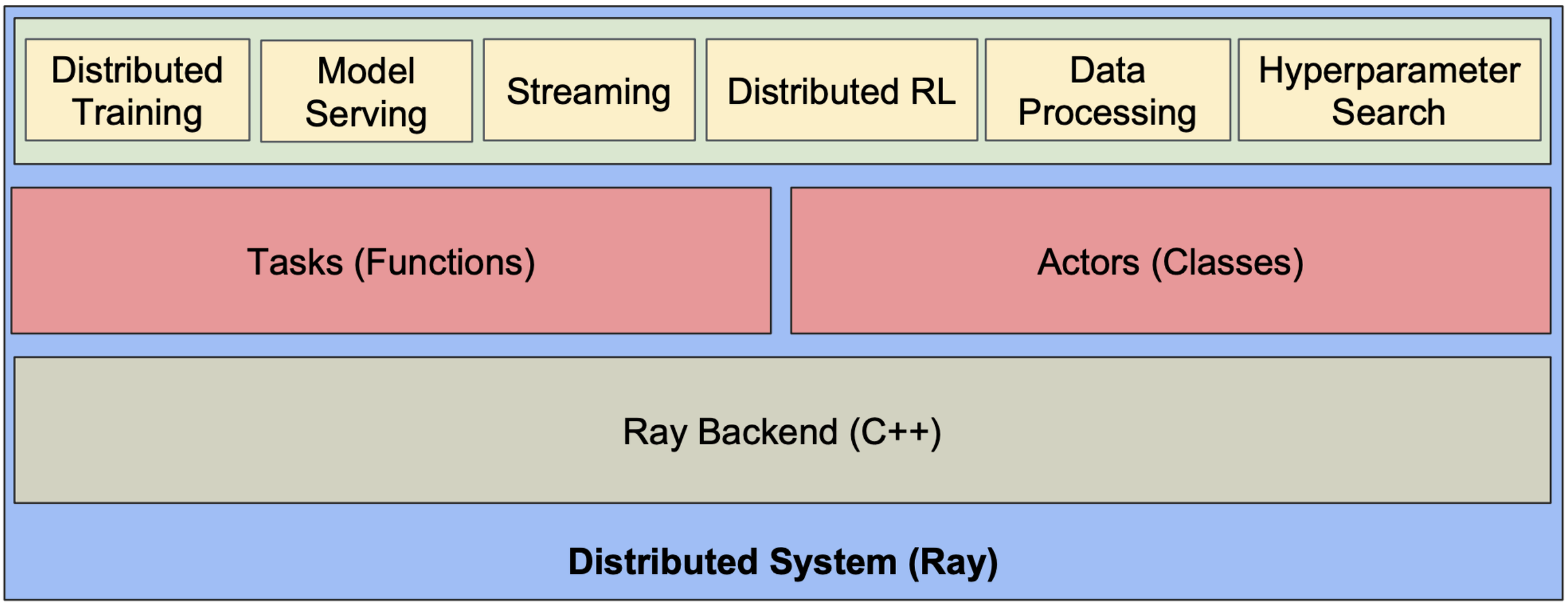


Reinforcement Learning

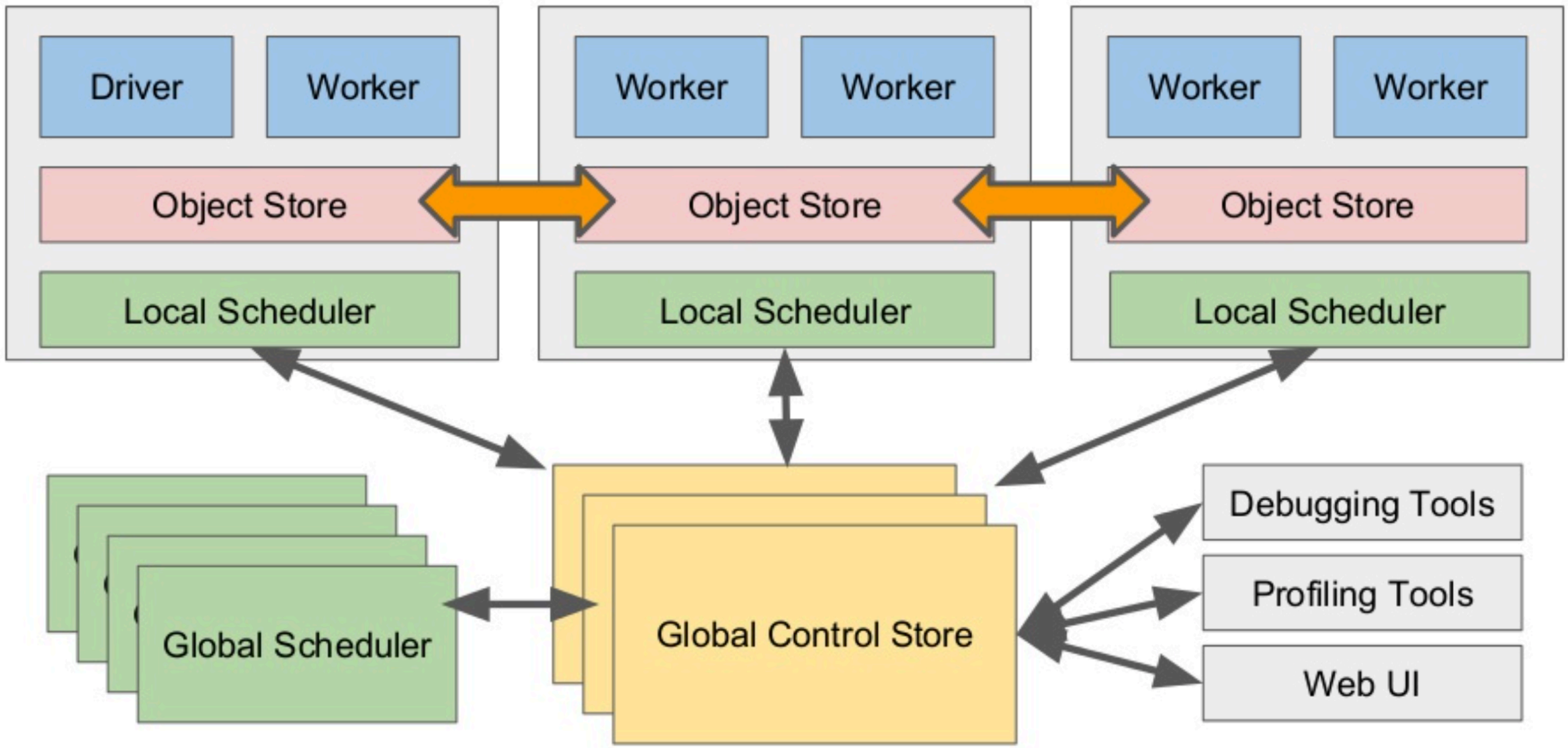
- Need to handle dynamic task graphs
  - Heterogenous durations
  - Heterogenous computations
- Scale millions of tasks per second



What is Ray?


Distributed Training
Model Serving
Streaming
Distributed RL
Data Processing
Hyperparameter Search
Tasks (Functions)
Actors (Classes)
Ray Backend (C++)
Distributed System (Ray)

Ray Architecture



Programming Model

- Tasks — stateless functions
- Actors — stateful objects
- Futures — for thread communication

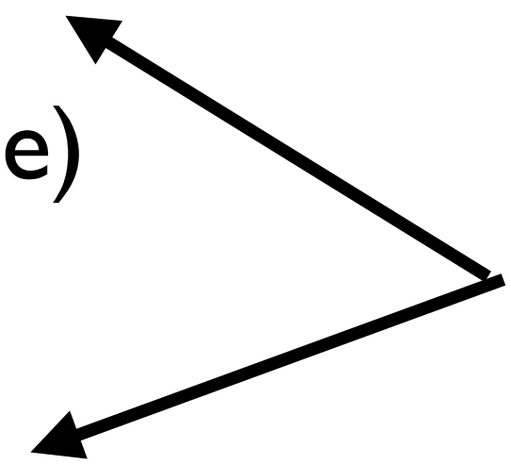
…in Python!

Tasks

```
def zeros(shape):
        return np.zeros(shape)


def dot(a, b):
    return np.dot(a, b)
```

Tasks

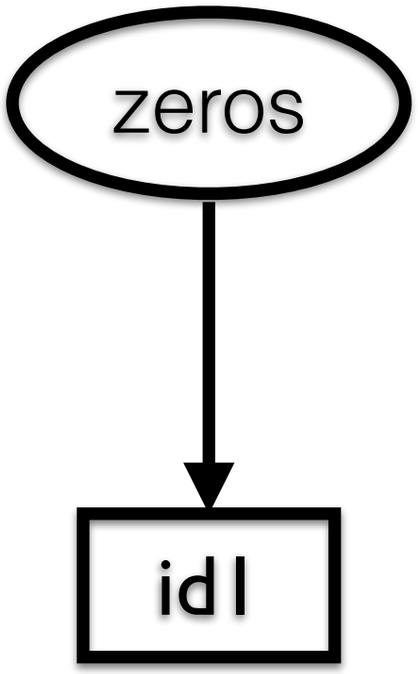
```
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
    return np.dot(a, b)
```



Can be also placed explicitly at a remote machines

Tasks

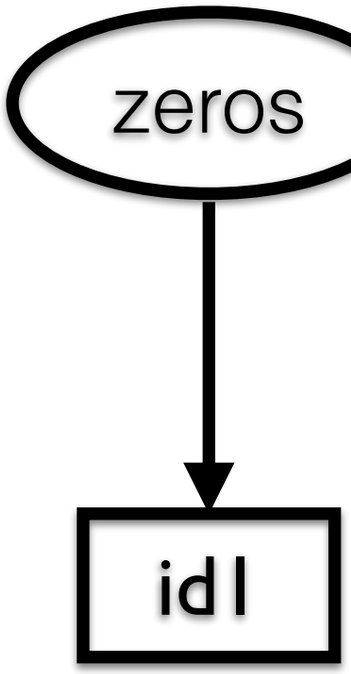
```
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
    return np.dot(a, b)

id1 = zeros.remote([5, 5])
```



Future: Encapsulates a value that is not available yet

Tasks

```
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
    return np.dot(a, b)

id1 = zeros.remote([5, 5])
```



Non-blocking call

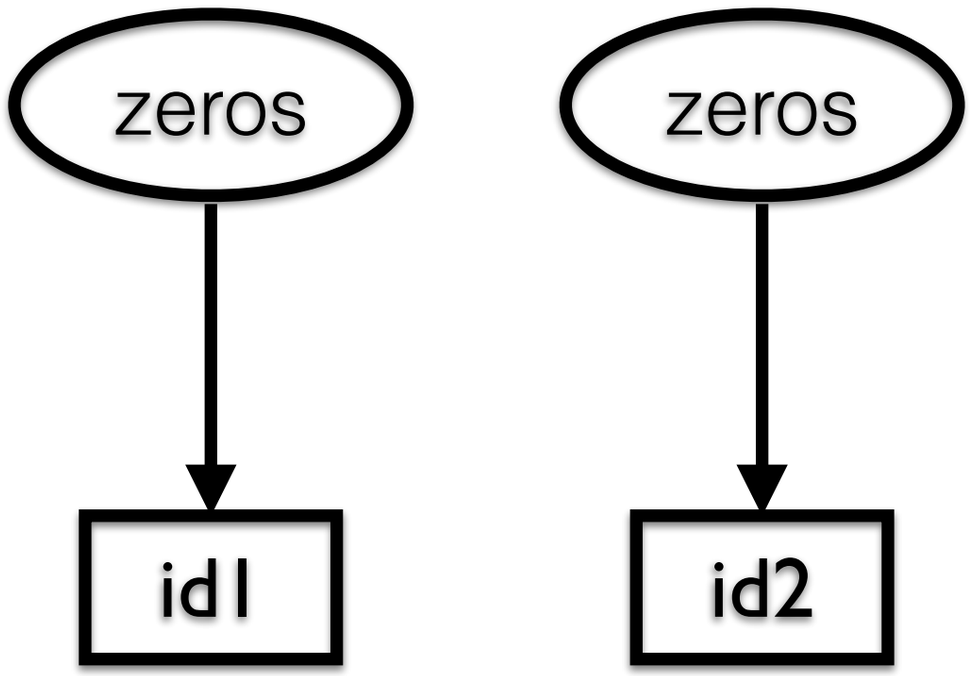
zeros() is executed by a different thread

Tasks

```
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
    return np.dot(a, b)
```

id1 = zeros.remote([5, 5])

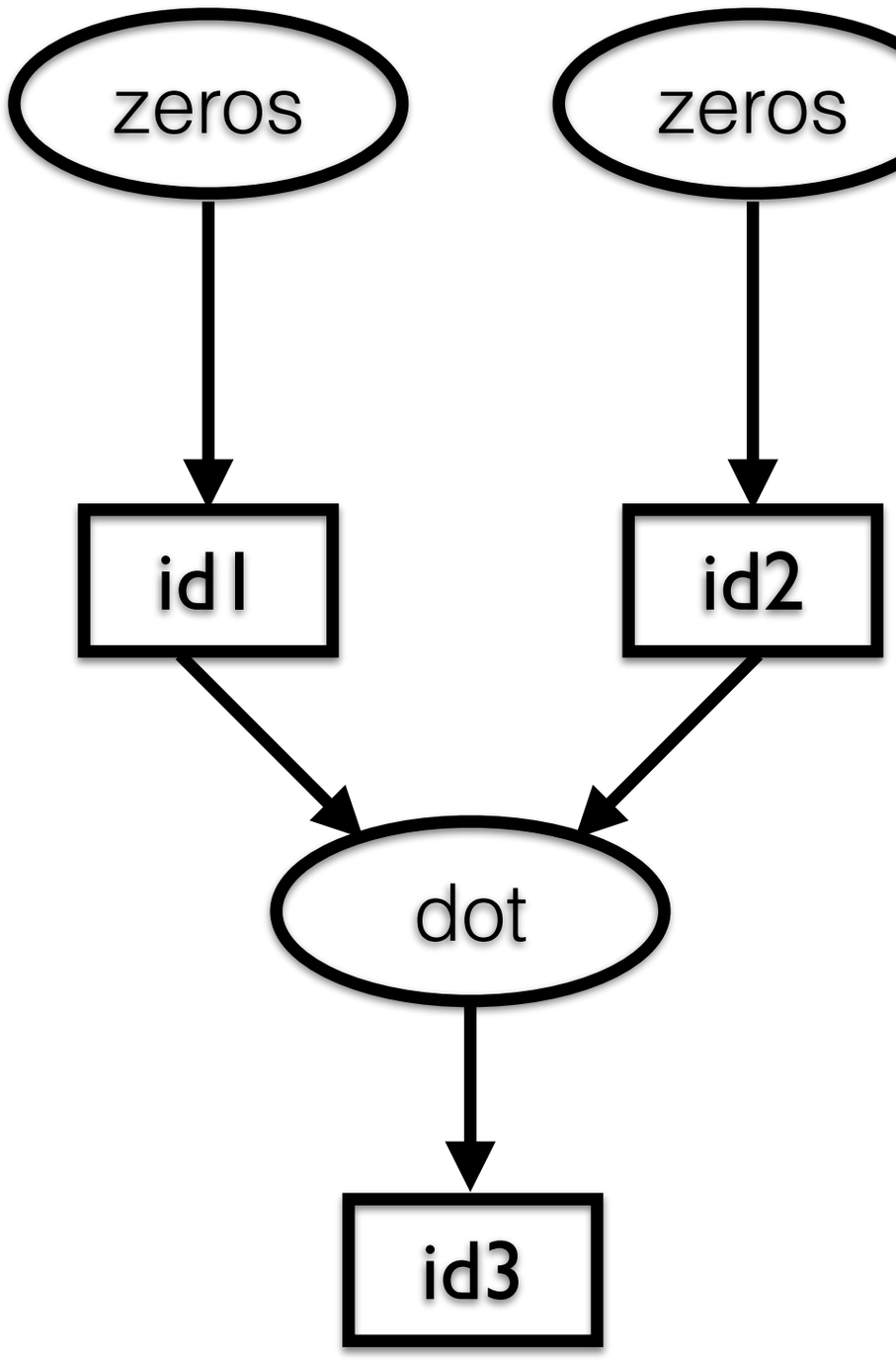
id2 = zeros.remote([5,5])



Tasks

```
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
    return np.dot(a, b)

id1 = zeros.remote([5, 5])

id2 = zeros.remote([5,5])

id3 = dot.remote(id1, id2)
```

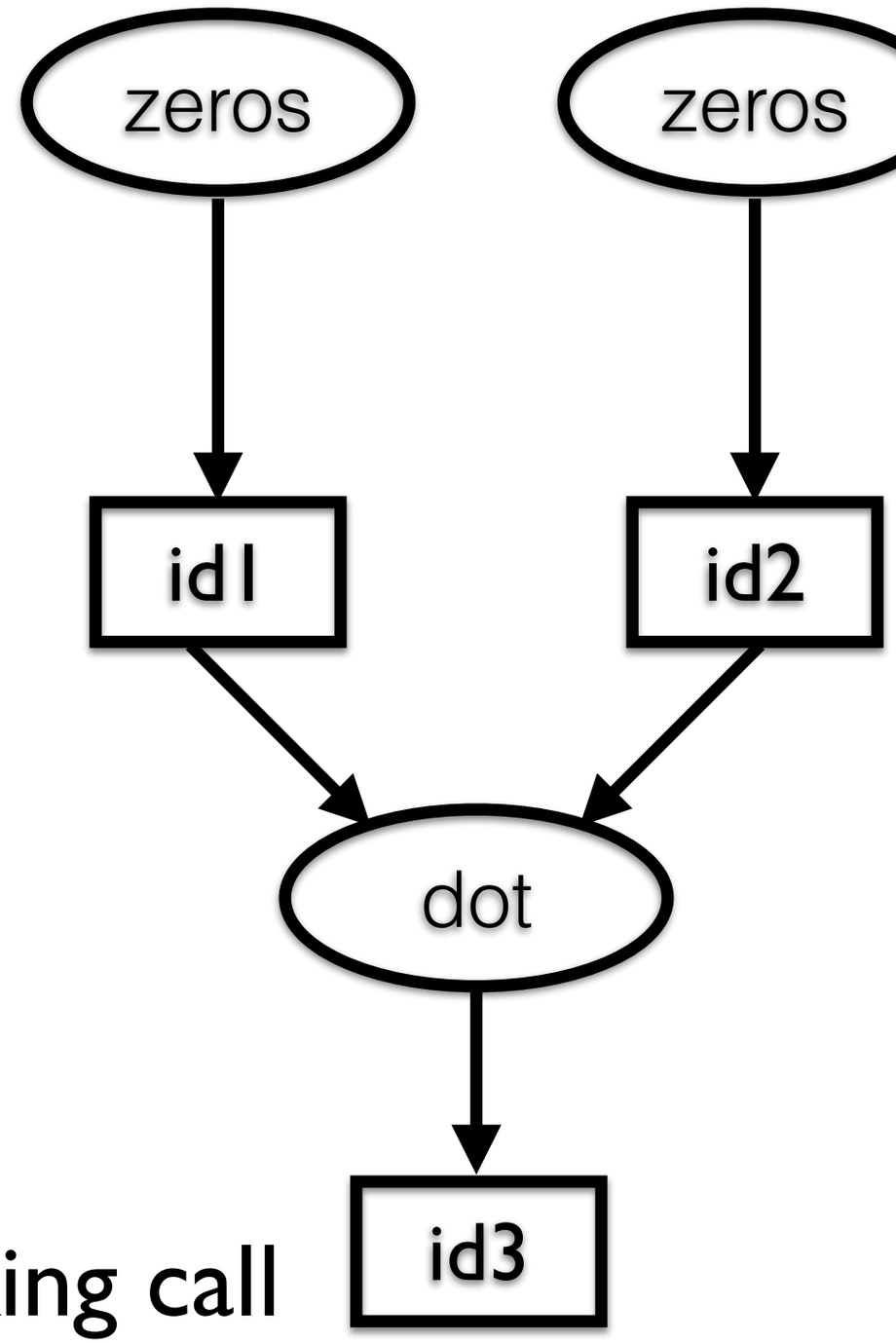
Futures can be chained!



Tasks

```
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
    return np.dot(a, b)

id1 = zeros.remote([5, 5])
id2 = zeros.remote([5,5])
id3 = dot.remote(id1, id2)
ray.get(id3)
```

Blocking call



Actors

```
class Counter(object):
    def __init__(self):
        self.value = 0

    def inc(self):
        self.value += 1
        return self.value
```

Actors

Can be also placed explicitly at a remote machine

```
@ray.remote(num_gpus=1)
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value
```

We can specify resources needed for the actor, e.g., 1 GPU

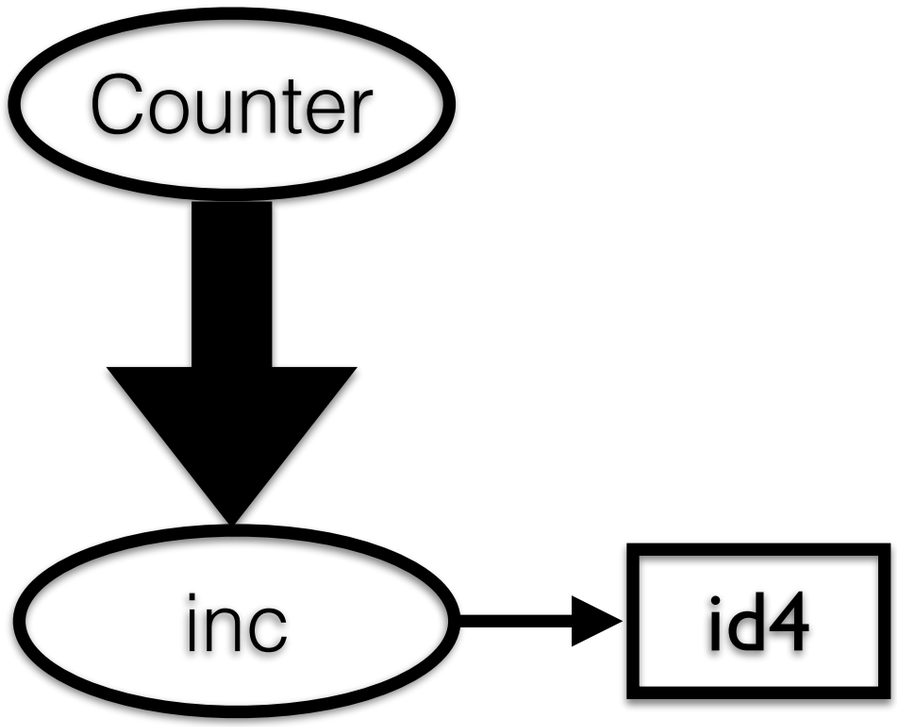
Actors

Counter

Returns a handle to the actor instance

```
@ray.remote(num_gpus=1)
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value
c = Counter.remote()
```

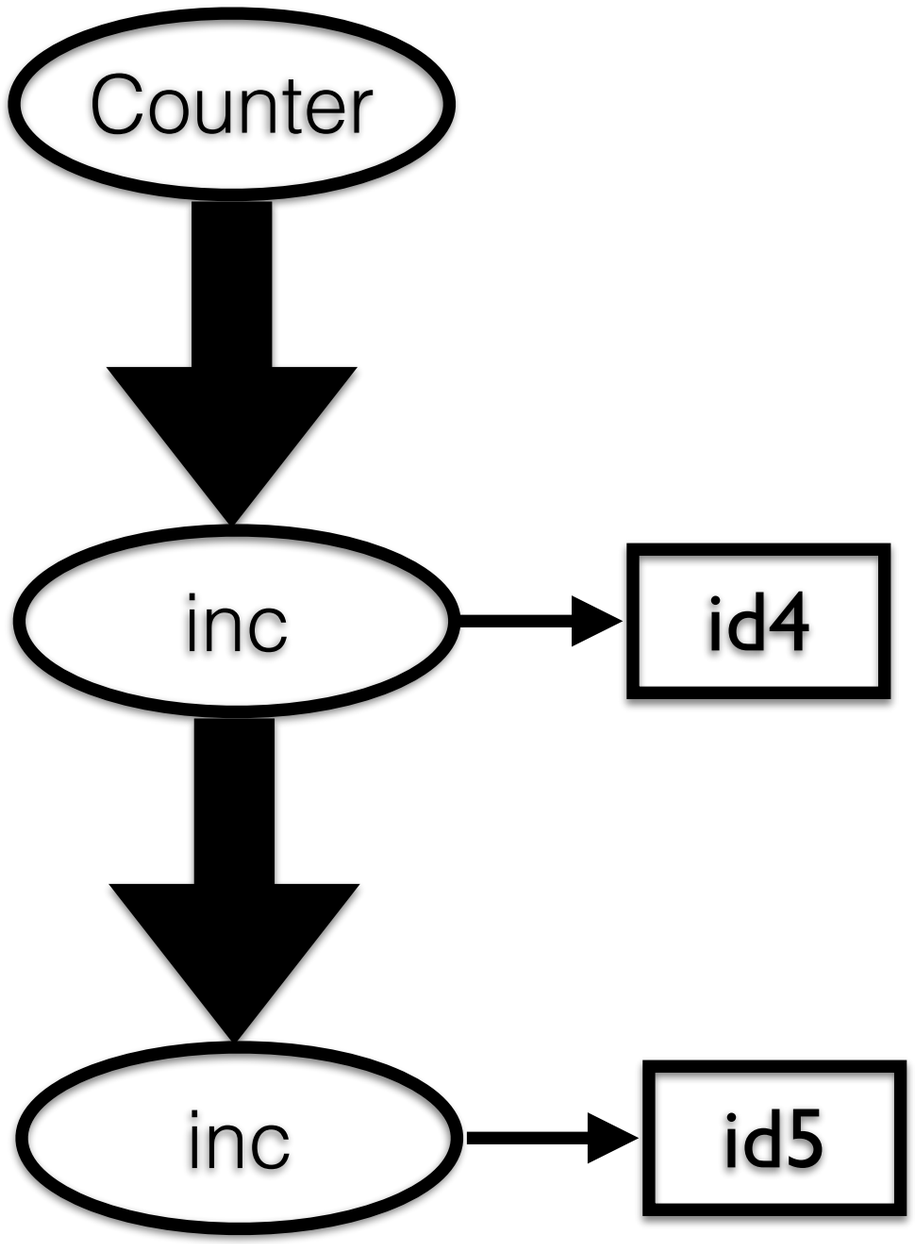
Actors




```
@ray.remote(num_gpus=1)
class Counter(object):
    def __init__(self):
        self.value = 0

    def inc(self):
        self.value += 1
        return self.value
c = Counter.remote()
id4 = c.inc.remote()
```

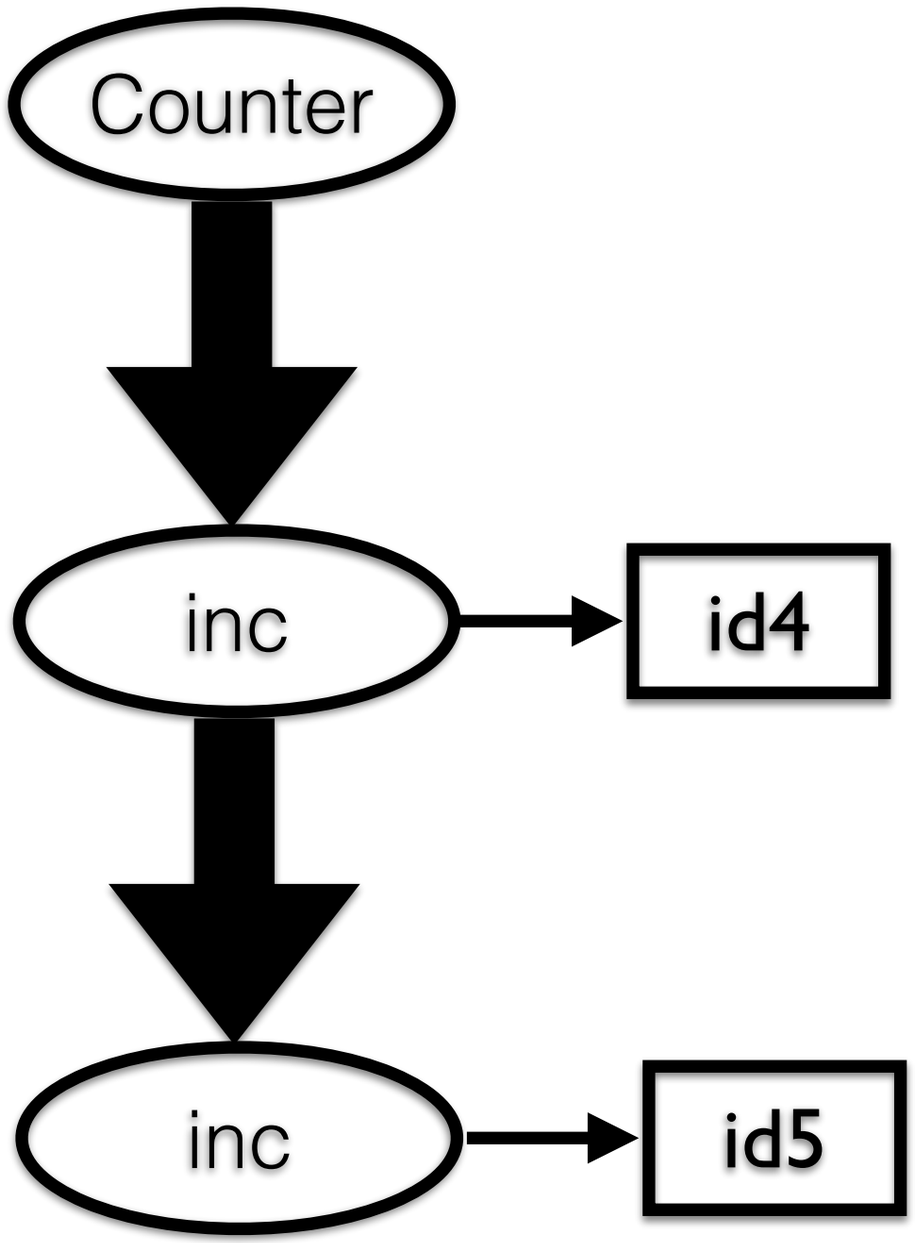
Stateful dependency

Actors




```
@ray.remote(num_gpus=1)
class Counter(object):
    def __init__(self):
        self.value = 0

    def inc(self):
        self.value += 1
        return self.value
c = Counter.remote()
id4 = c.inc.remote()
id5 = c.inc.remote()
```

Actors




```
@ray.remote(num_gpus=1)
class Counter(object):
    def __init__(self):
        self.value = 0

    def inc(self):
        self.value += 1
        return self.value
c = Counter.remote()
id4 = c.inc.remote()
id5 = c.inc.remote()
ray.get([id4, id5])
```

API

Name
$futures = \textbf{f.remote}(args)$
$objects = \textbf{ray.get}(futures)$
$ready\_futures = \textbf{ray.wait}(futures, k, timeout)$
$actor = \textbf{Class.remote}(args)$ $futures = actor.\textbf{method.remote}(args)$

How would you implement Raft in Ray?

```
class Raft(object):
    def __init__(self):
        self.peers = N
        ...
    def start_election(self):
        ...
    def request_votes(self):
        ...
```

How would you implement Raft in Ray?

```
def start_election(self):
    ...
  majority = len(self.peers)/2+1
  votes = []
for p in self.peers:
   v = p.request_votes.remote()
   votes.append(v)
ready, _ = ray.wait(votes, majority, election_timeout)
```

```
class Raft(object):
    def __init__(self):
        self.peers = N
        ...
    def start_election(self):
        ...
    def request_votes(self):
```

References

- Ray: https://ray.io
  - Documentation
  - Code
  - Tutorials
  - Use cases
- Anyscale: https://anyscale.com