RAY

Ray: A **distributed framework** for emerging {AI} applications [PDF] usenix.org

P Moritz, R Nishihara, S Wang, A Tumanov… - … USENIX symposium on …, 2018 - usenix.org

… propose **Ray**, a general-purpose cluster-computing **framework** … , **Ray** implements a unified interface that can express both task-parallel and actorbased computations. Tasks enable **Ray** …

☆ Save 99 Cite Cited by 1432 Related articles All 23 versions »

R244 - Chris Tomy - 2024/10/23

REFRESHER ON RL

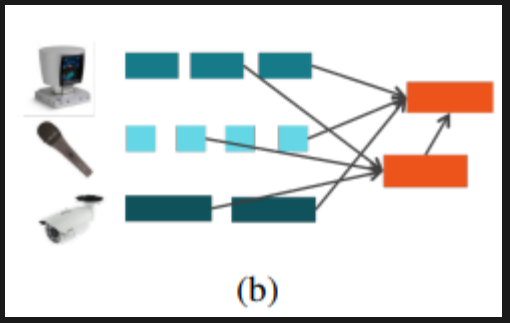
Goal: learn a policy $\pi : S \rightarrow A$

How?

- Generate trajectories based on a policy $[(s_1, r_1), \ldots, (s_n, r_n)]$
- Update the policy

PROPERTIES OF RL TRAINING

```
def rollout(policy, env):
  trajectory = []
  state = env.init()
  while env.running():
    # policy computation requires GPU
    action = policy.compute(state)
    # task parallel with different computations:
    state, reward = environment.step(action)
    trajectory.append(state, reward)
  # trajectory length varies (hence varied duration)
  return trajectory
```



i.e. heterogeneous across:

- Functionality: `environment.step(action)`
- Duration: `len(trajectory)`
- Resource types: `policy.compute(state)`

BSP IS INSUFFICIENT

Bulk Synchronous Parallel model generally expects:

- Same computations
- Same duration to complete

SO, WHAT IS RAY?

A compute cluster scheduler and task-parallel programming API.

RAY PROGRAMMING MODEL

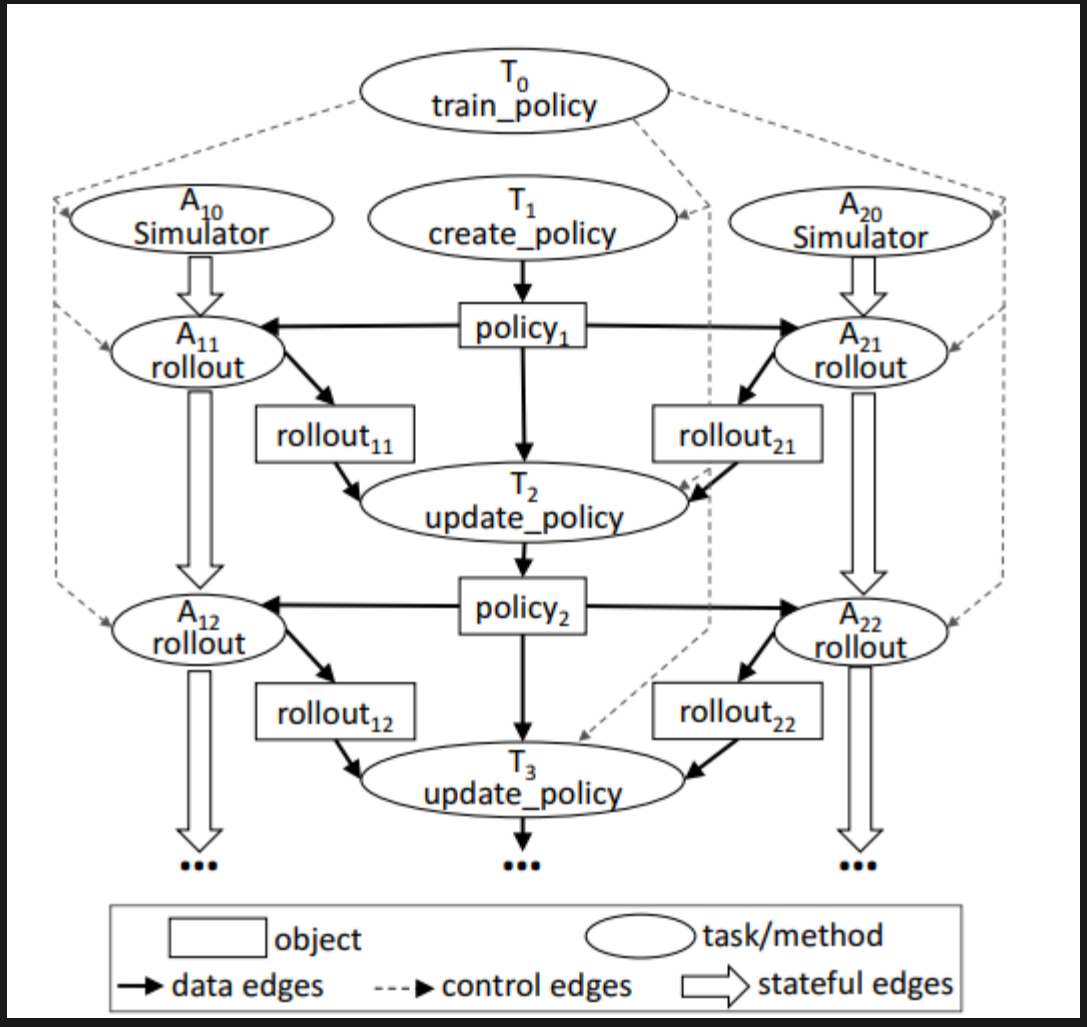
Basic task parallelism (futures)

```
@ray.remote
def example_task(n):
  # stateless and side-effect free function
  time.sleep(n)
  return n
# non-blocking call:
results = [example_task.remote(i) for i in range(4)]
# blocking call:
output = ray.get(results)
```

RAY'S ANSWER TO HETEROGENEITY

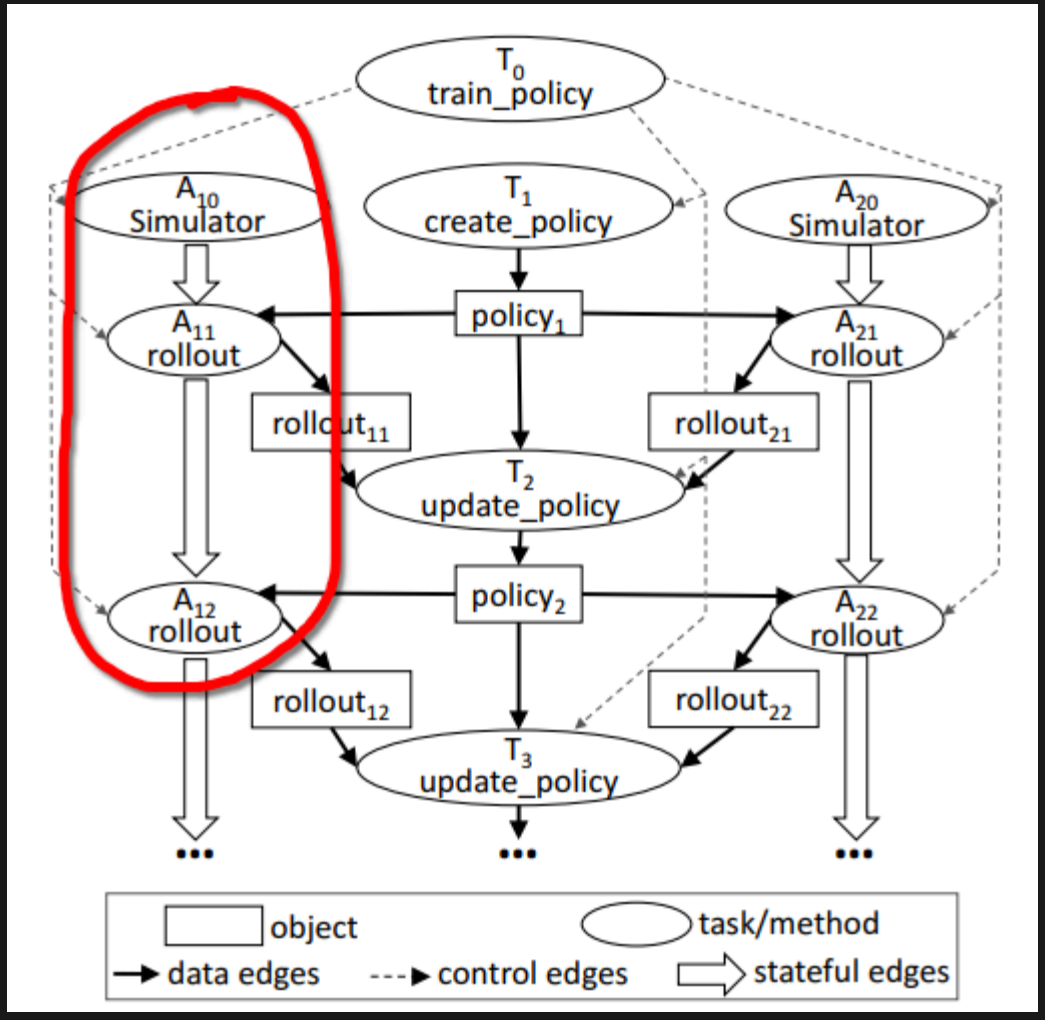
- Nested remote functions (avoiding driver bottleneck)
- `ray.wait` - for duration variance
- `@ray.remote(num_gpus=n)` - for different resource types

RL APPLICATION



NOVEL CONTRIBUTION

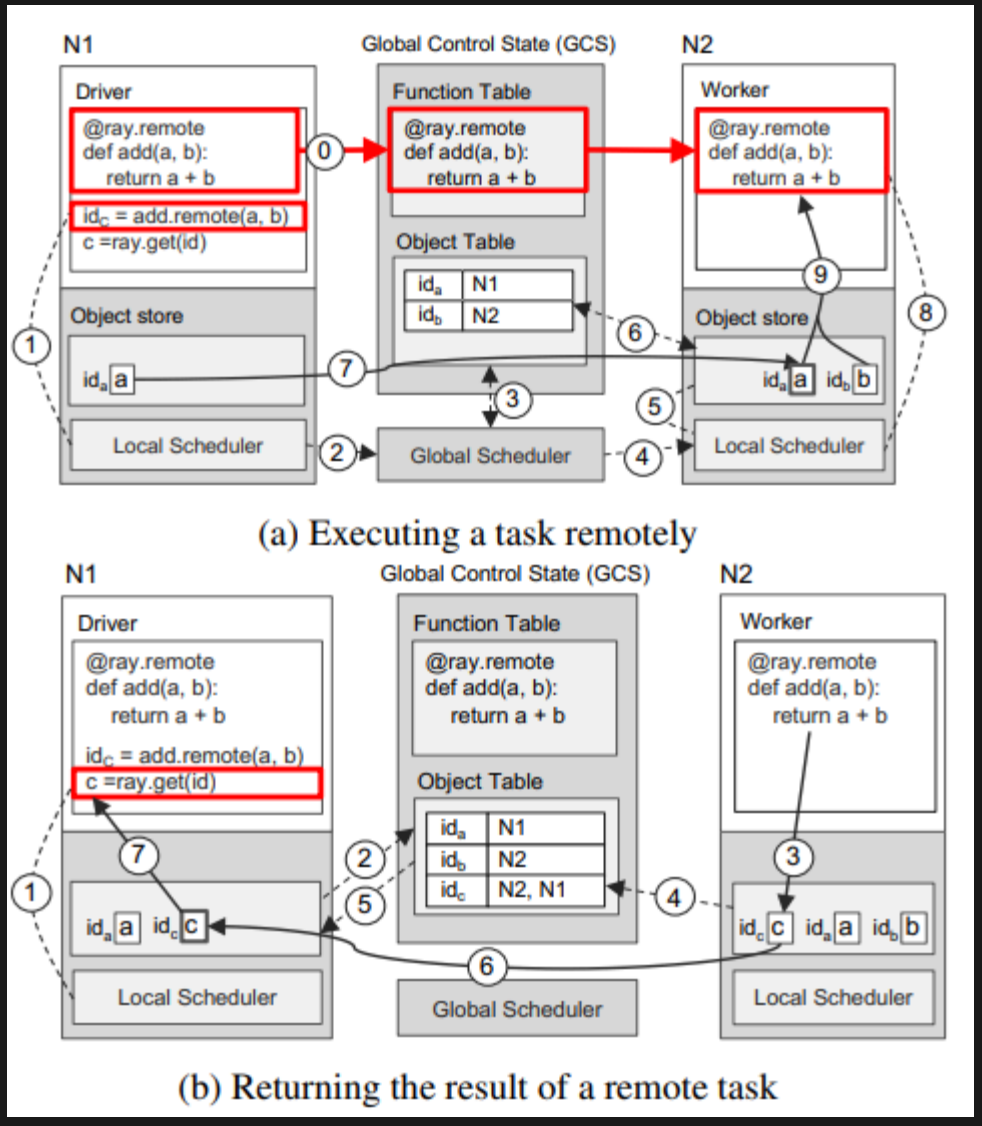
Actor model on top of task-parallelism



Actor model

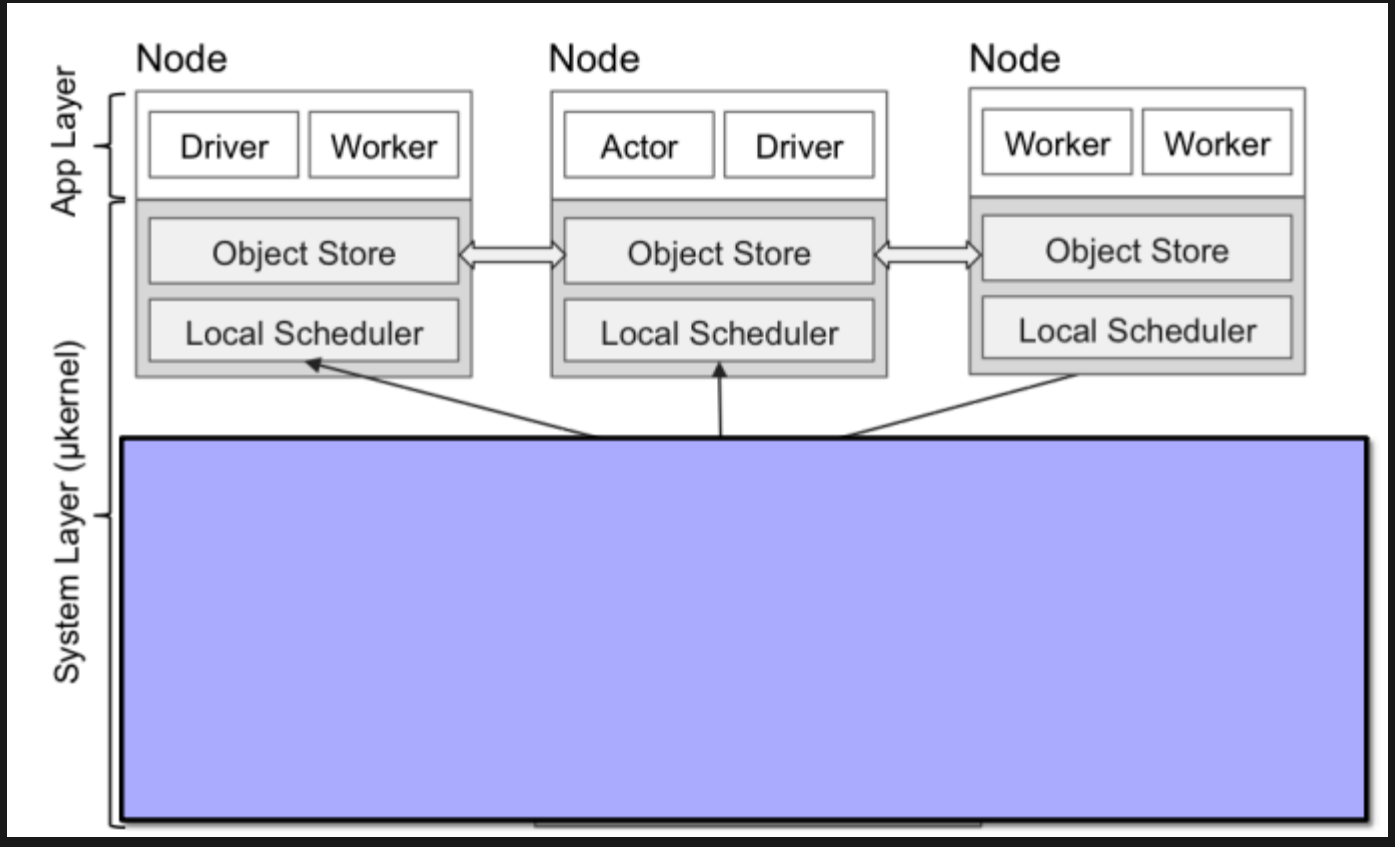
```
@ray.remote
class Button:
    # stateful class
    def __init__(self):
        self.presses = 0
    def press(self):
        self.presses += 1
        return self.presses

...

for _ in range(10):
    button.press().remote()
```

SYSTEM WALKTHROUGH



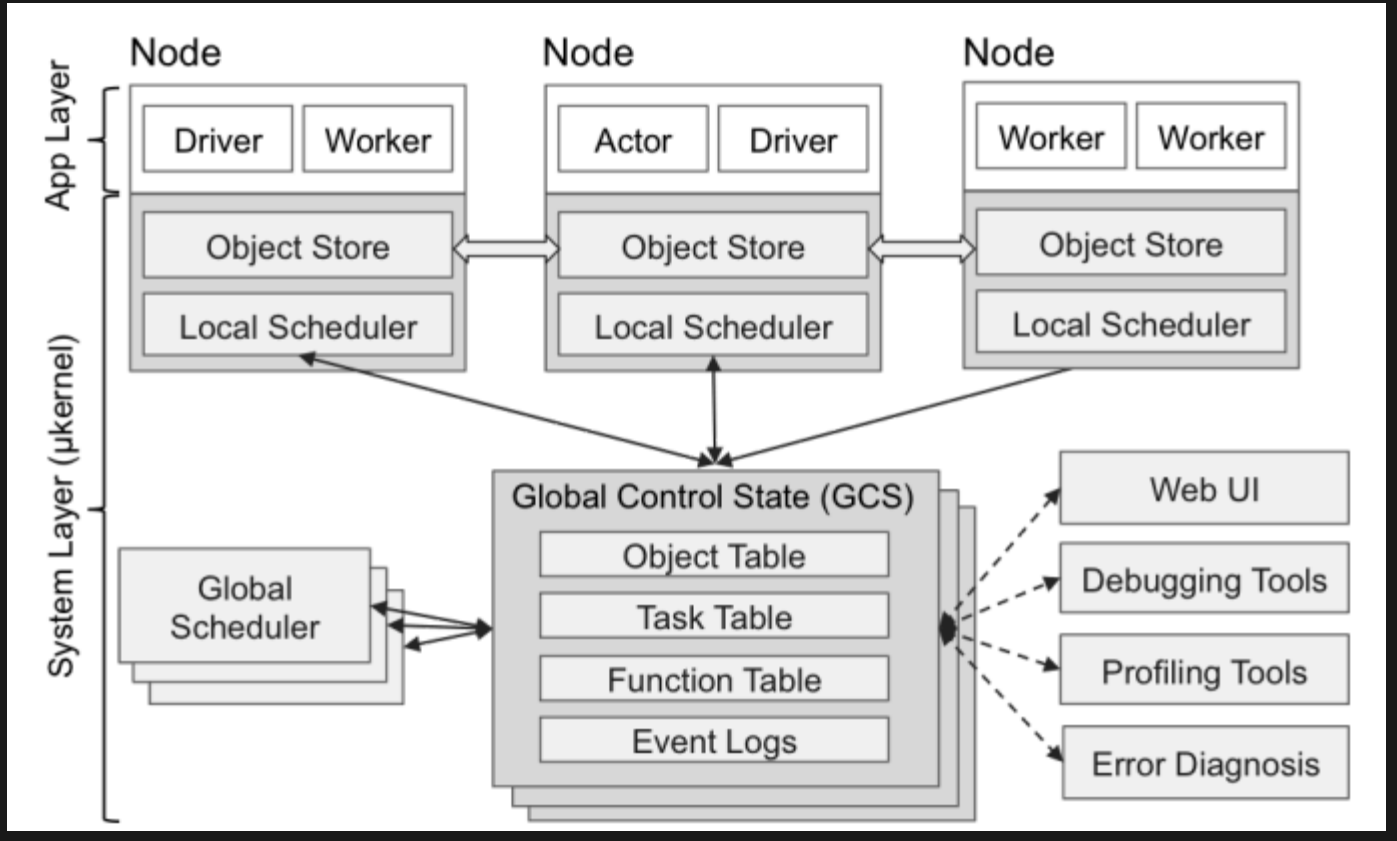
COMPONENTS

- Driver - main process (user program)
- Worker - stateless process
- Actor - stateful process



SYSTEM/BACK-END

- Global Control Store (GCS)
- Bottom-up distributed scheduler
- In-Memory Distributed Object Store



IMPLEMENTATION DETAILS

- Distributed scheduler
  - Both single-threaded processes
  - Local scheduler heartbeats to global scheduler with load info
- Object store
  - Objects accessed through shared memory
  - Fast serial/deserialization with Apache Arrow
- GCS
  - Uses Redis as a key-value store
  - Sharded and replicated

SCALABLE

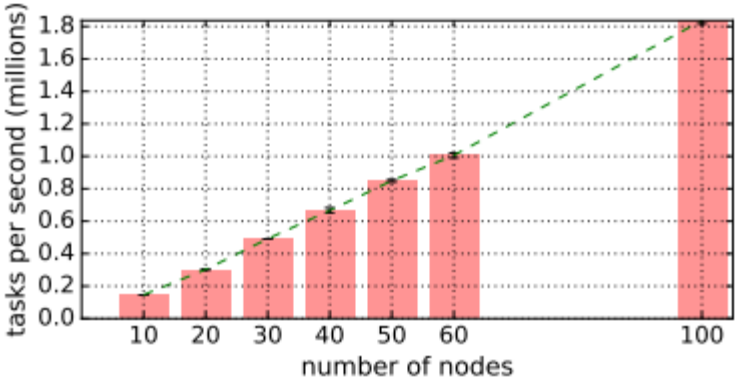


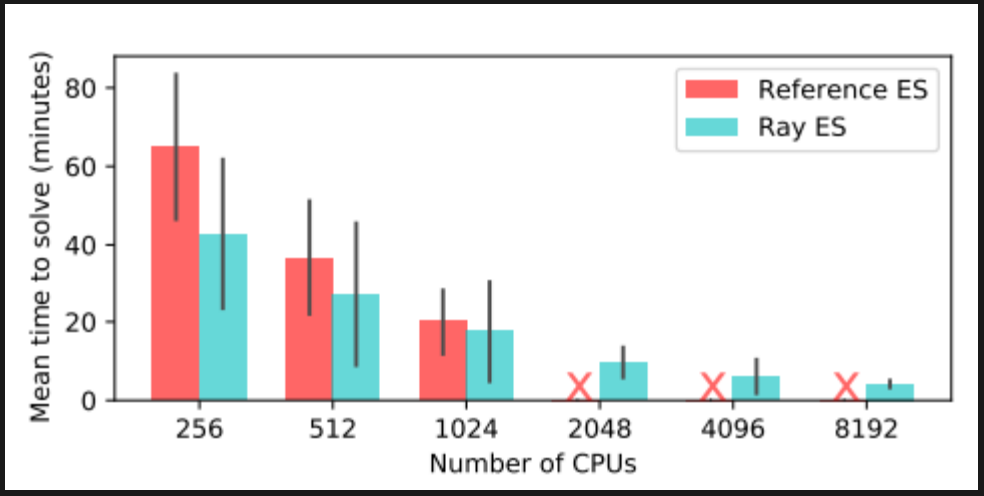
Figure 7: End-to-end scalability of the system is achieved in a linear fashion, leveraging the GCS and bottom-up distributed scheduler. Ray reaches 1 million tasks per second throughput with 60 m4.16xlarge nodes and processes 100 million tasks in under a minute. We omit $x \in \{70, 80, 90\}$ due to cost.

(However, ambiguity on task.)

RESULTS ON RL

EFFECTIVENESS ON EVOLUTION STRATEGIES

Since "hill-climbing" is scalable.



https://github.com/openai/gym/wiki/Humanoid-V1

GENERAL CRITIQUES

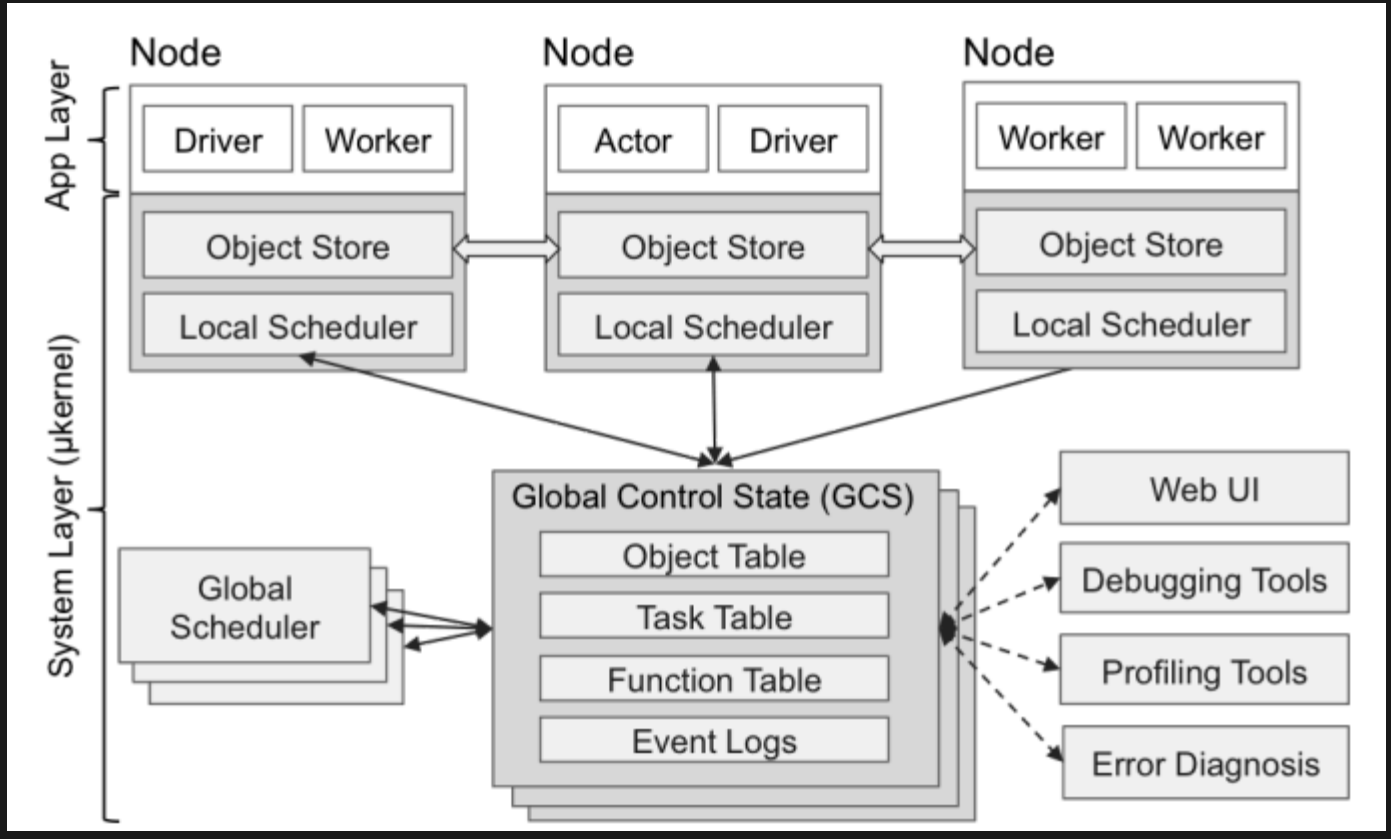
"SOFT" REALTIME

Time budget (ms)	30	20	10	5	3	2	1
% actions dropped	0	0	0	0	0.4	40	65
Stable walk?	Yes	Yes	Yes	Yes	Yes	No	No

Table 3: Low-latency robot simulation results

Ray introduces overhead

GLOBAL STATE LIMITATIONS?



NO EVAL AGAINST BSP