# Ray Core Internals

*DS 5110/CS 5501: Big Data Systems*

*Spring 2024*

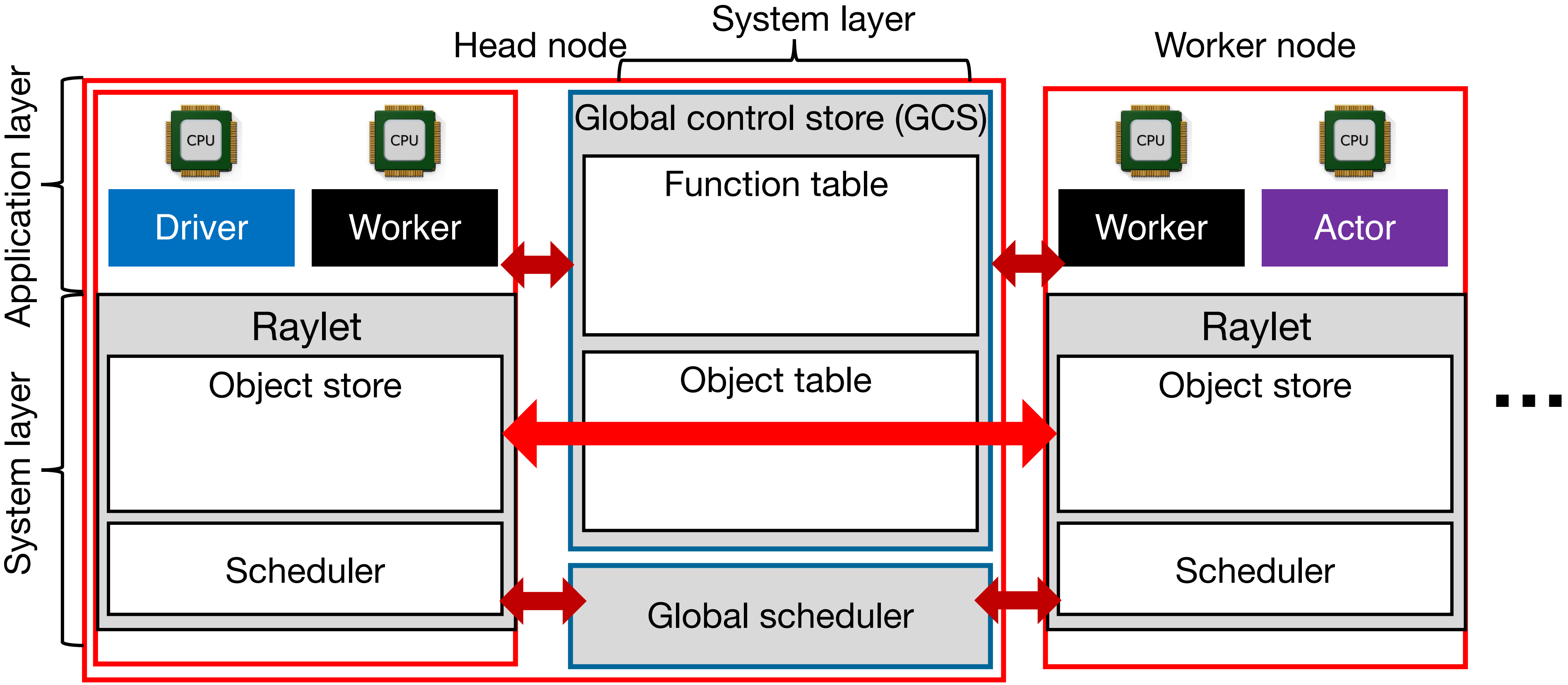
Lecture 6b

Yue Cheng

# Learning objectives

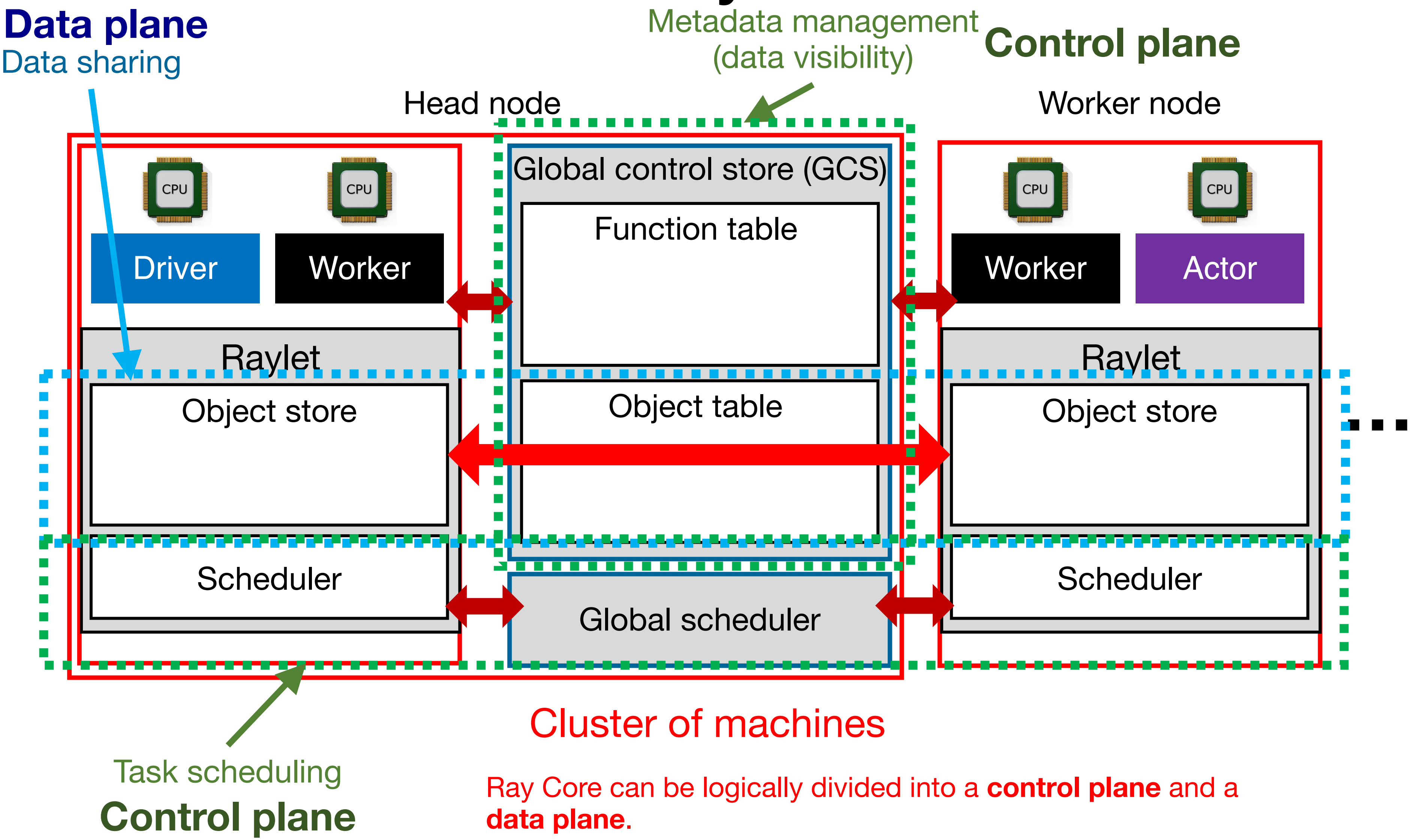
- Understand how Ray tasks and actors are managed under the hood
- Know the concept of a control plane and a data plane

# Under the hood: Ray Core architecture



Cluster of machines

# Under the hood: Ray Core architecture



Cluster of machines

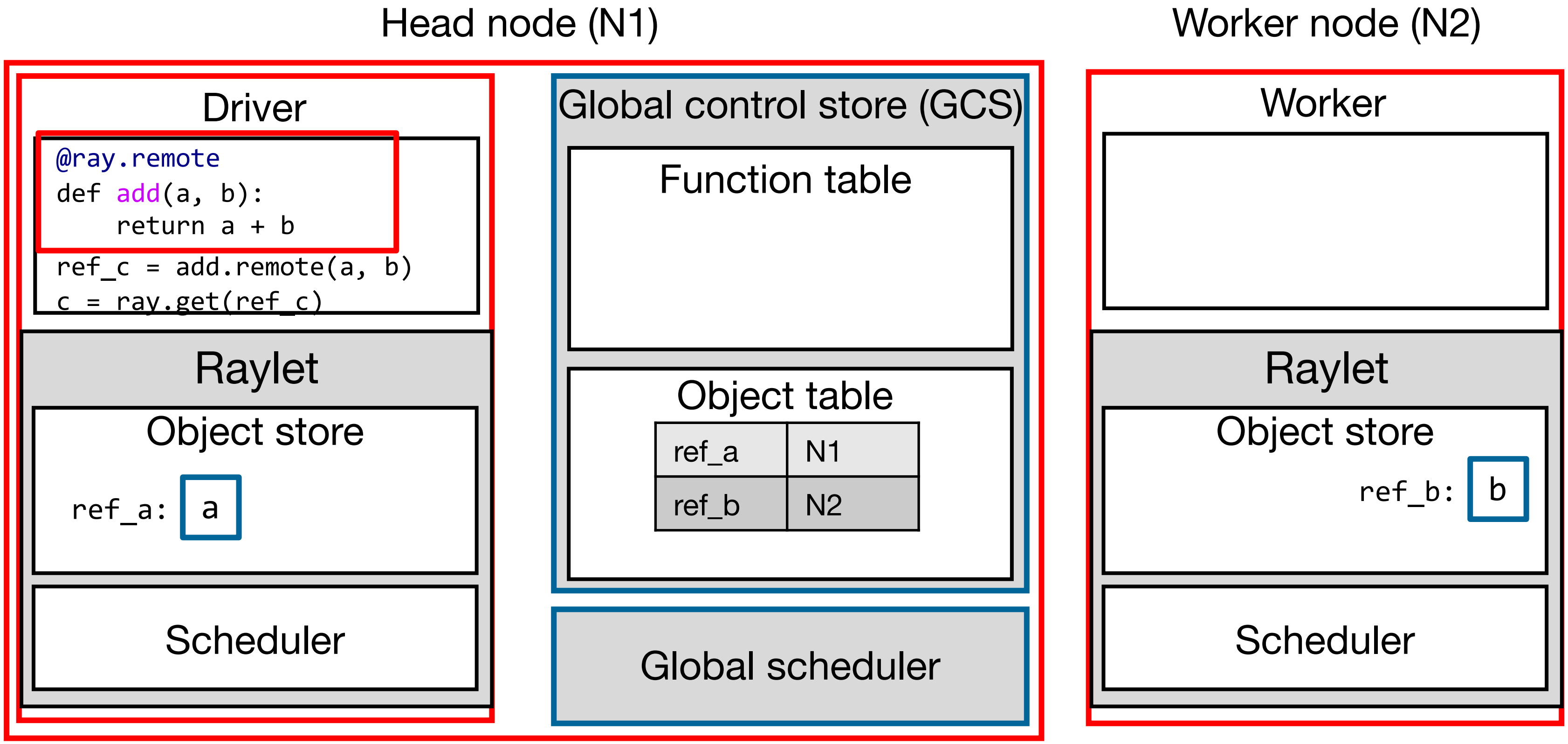
Ray Core can be logically divided into a **control plane** and a **data plane**.

# Life cycle of a remote task

# Executing a task remotely

# Executing a task remotely

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

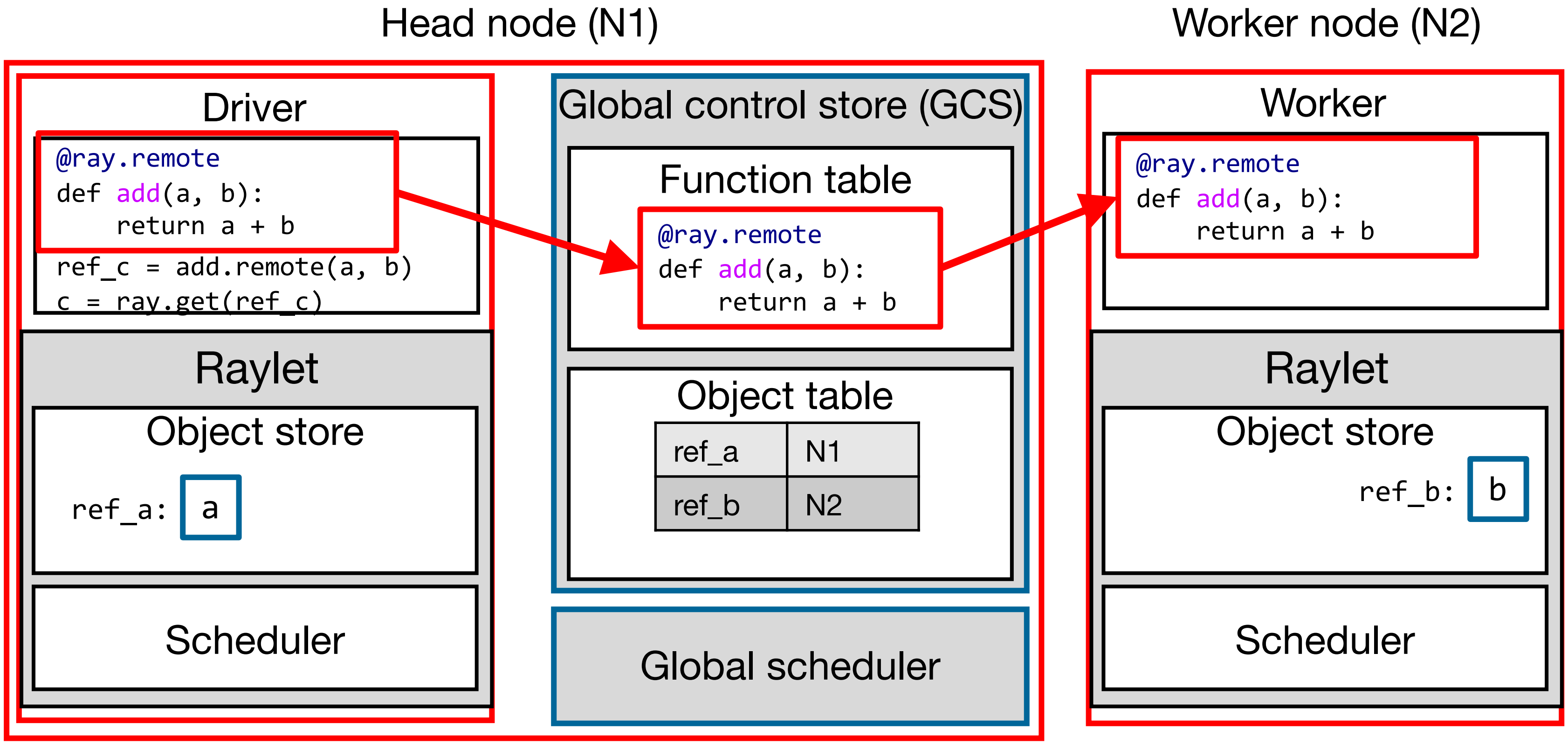
| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |

Cluster of machines

Now, the remote task function add is initialized…

# Executing a task remotely

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

```
@ray.remote
def add(a, b):
    return a + b
```

```
@ray.remote
def add(a, b):
    return a + b
```

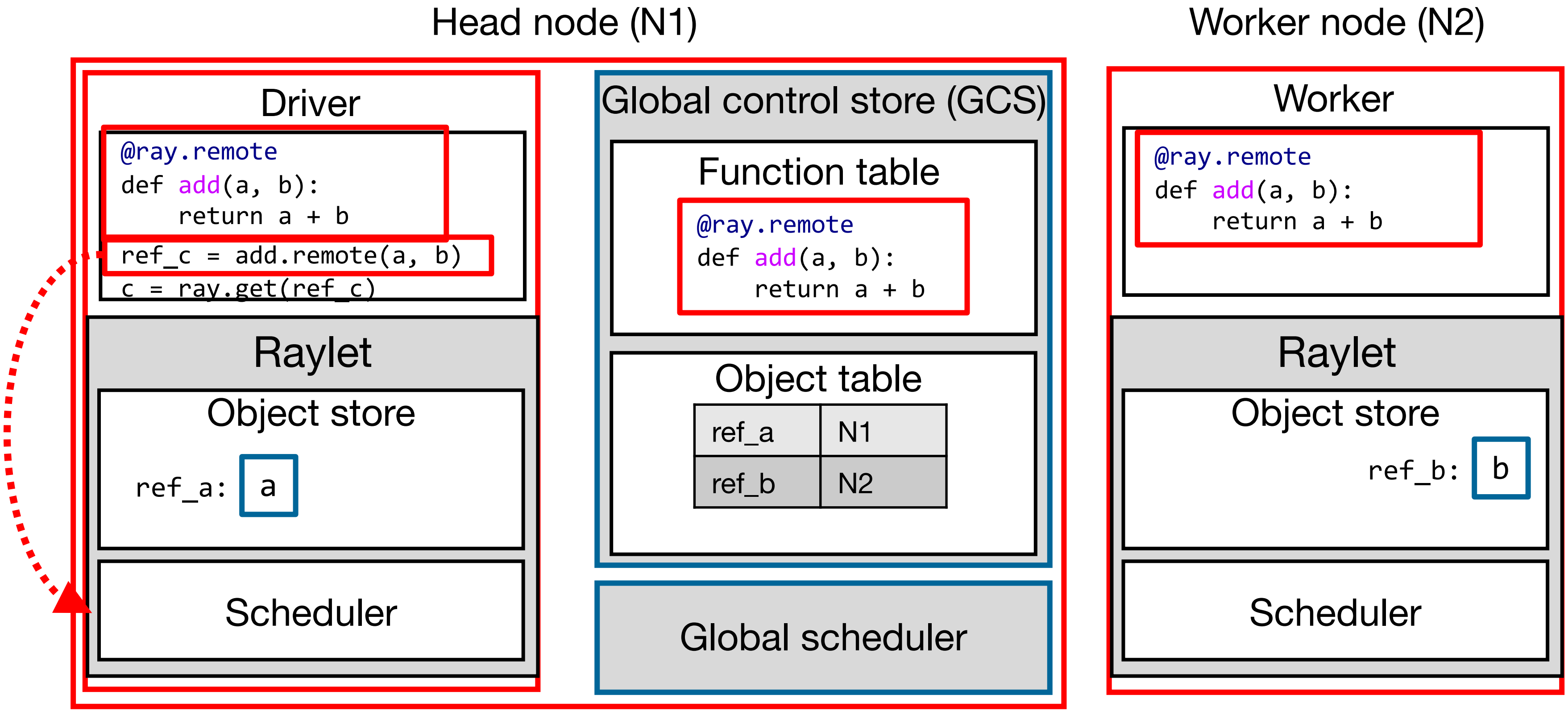
| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |

Cluster of machines

Step 0: Ray automatically registers each initialized remote function and distributes it to every worker in the cluster.

# Executing a task remotely

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

```
ref_a: a
```

```
@ray.remote
def add(a, b):
    return a + b
```

| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |

```
@ray.remote
def add(a, b):
    return a + b
```

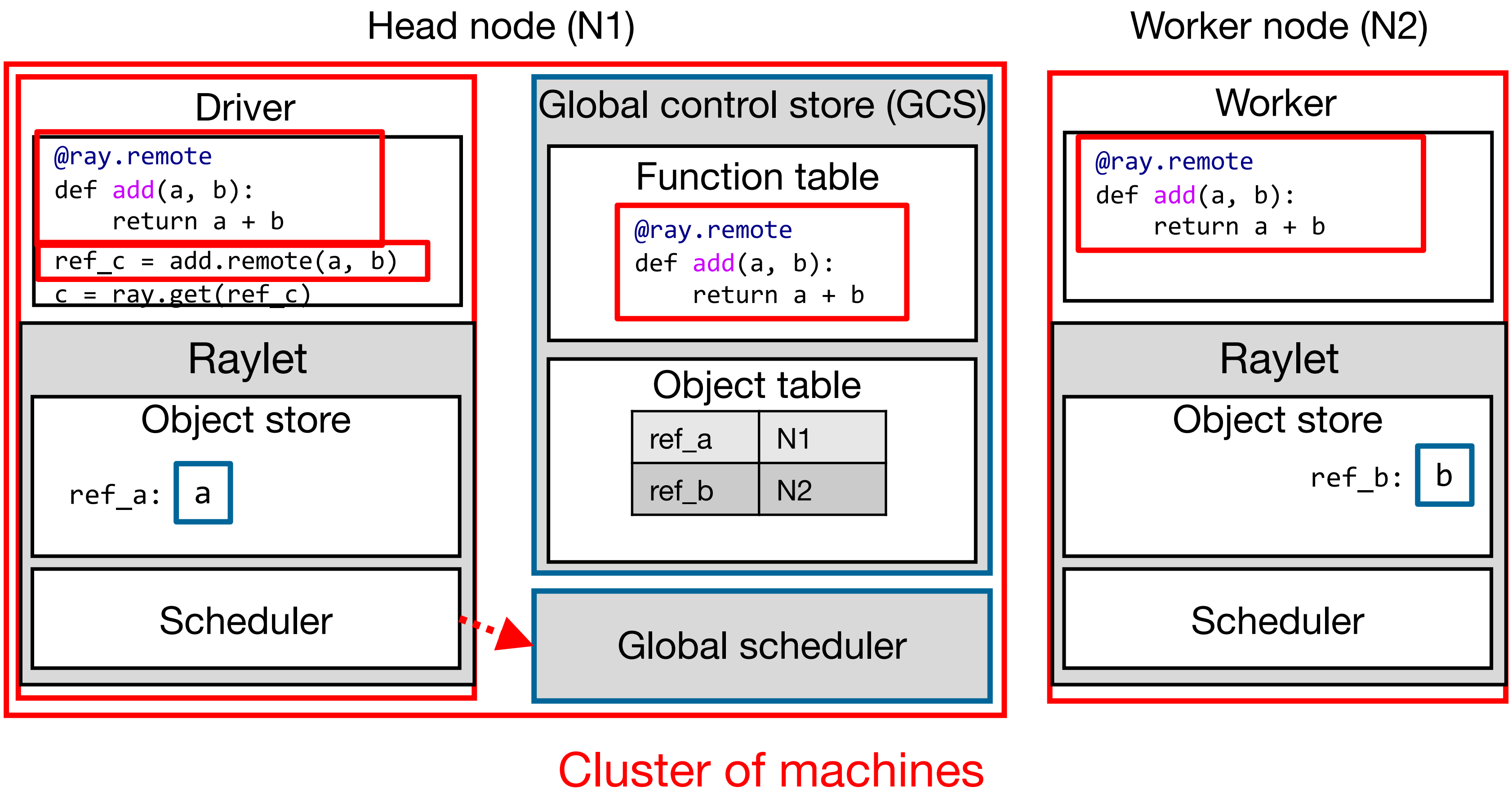
```
ref_b: b
```

Cluster of machines

Step 1: Driver contacts N1's local scheduler to find out the ownership of object b (which node holds b).

# Executing a task remotely

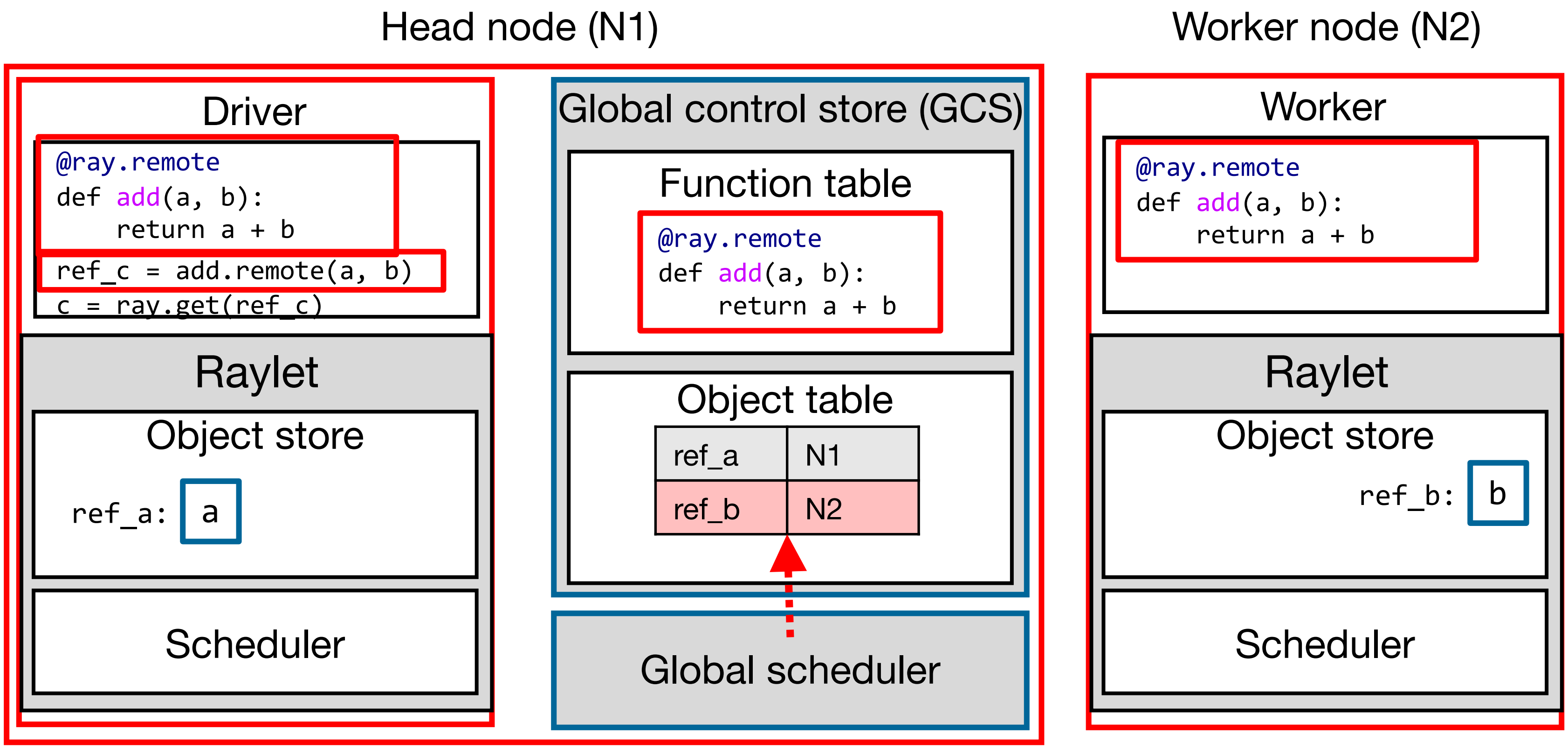
```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



Step 2: N1’s local scheduler (located on N1) contacts global scheduler.

# Executing a task remotely

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

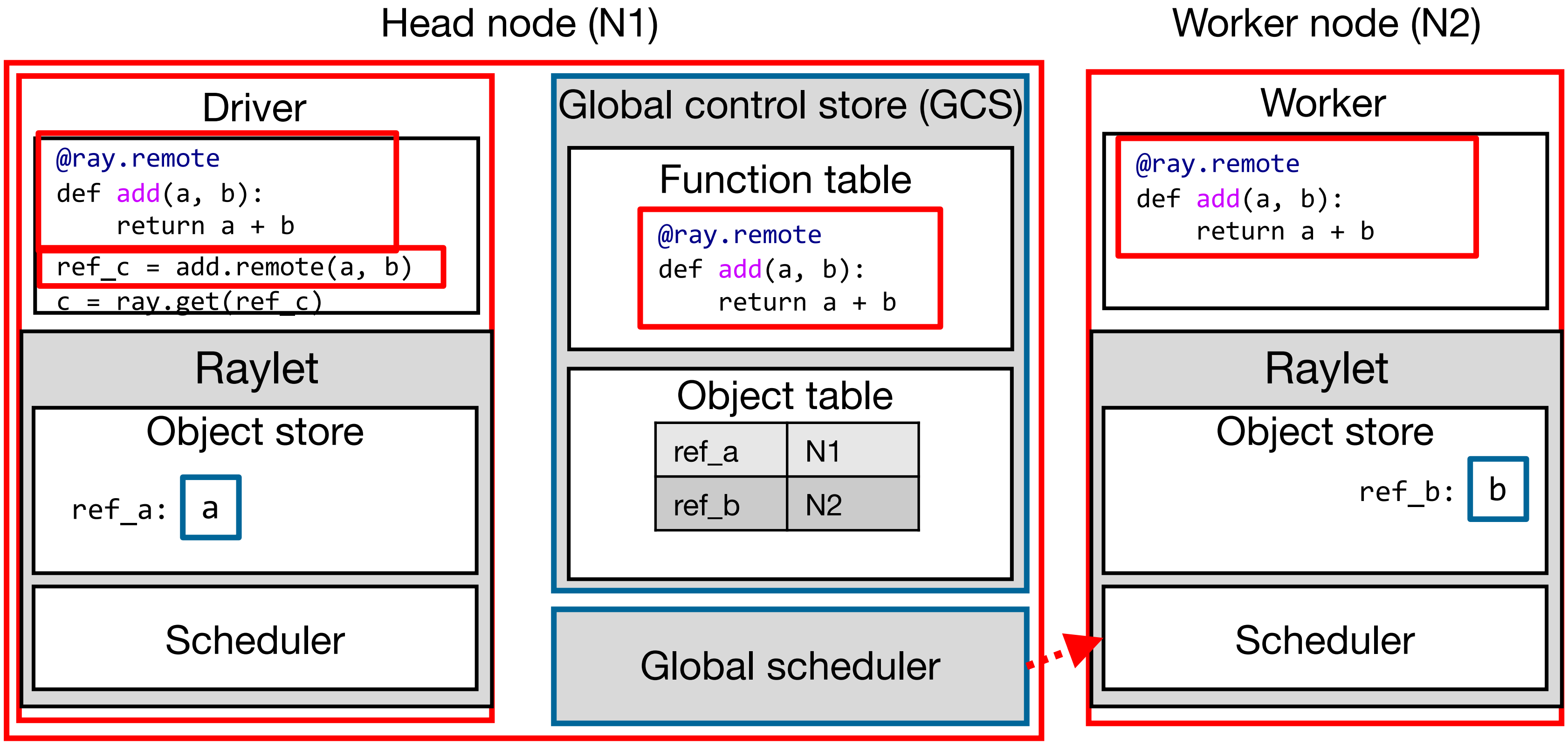


Cluster of machines

Step 3: Global scheduler performs an object table lookup and finds N2 holds b.

# Executing a task remotely

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

```
@ray.remote
def add(a, b):
    return a + b
```

| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |

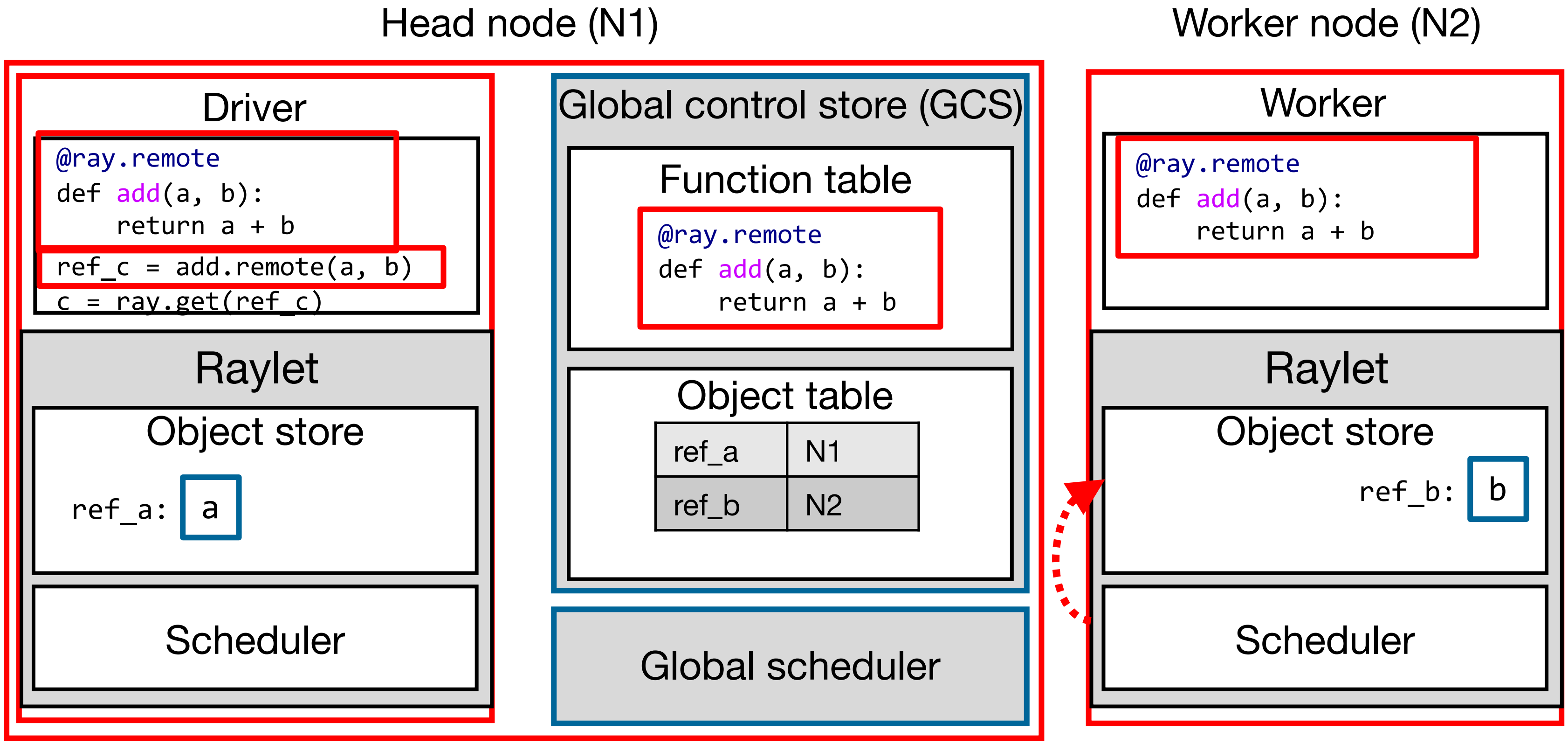
```
@ray.remote
def add(a, b):
    return a + b
```

Cluster of machines

Step 4: Global scheduler does some thinking (scheduling decision making) and decides to schedule the task on N2.

# Executing a task remotely

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



Head node (N1)

Driver

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

Raylet

Object store

ref_a: a

Scheduler

Global control store (GCS)

Function table

```
@ray.remote
def add(a, b):
    return a + b
```

Object table

| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |

Global scheduler

Worker node (N2)

Worker

```
@ray.remote
def add(a, b):
    return a + b
```

Raylet

Object store

ref_b: b

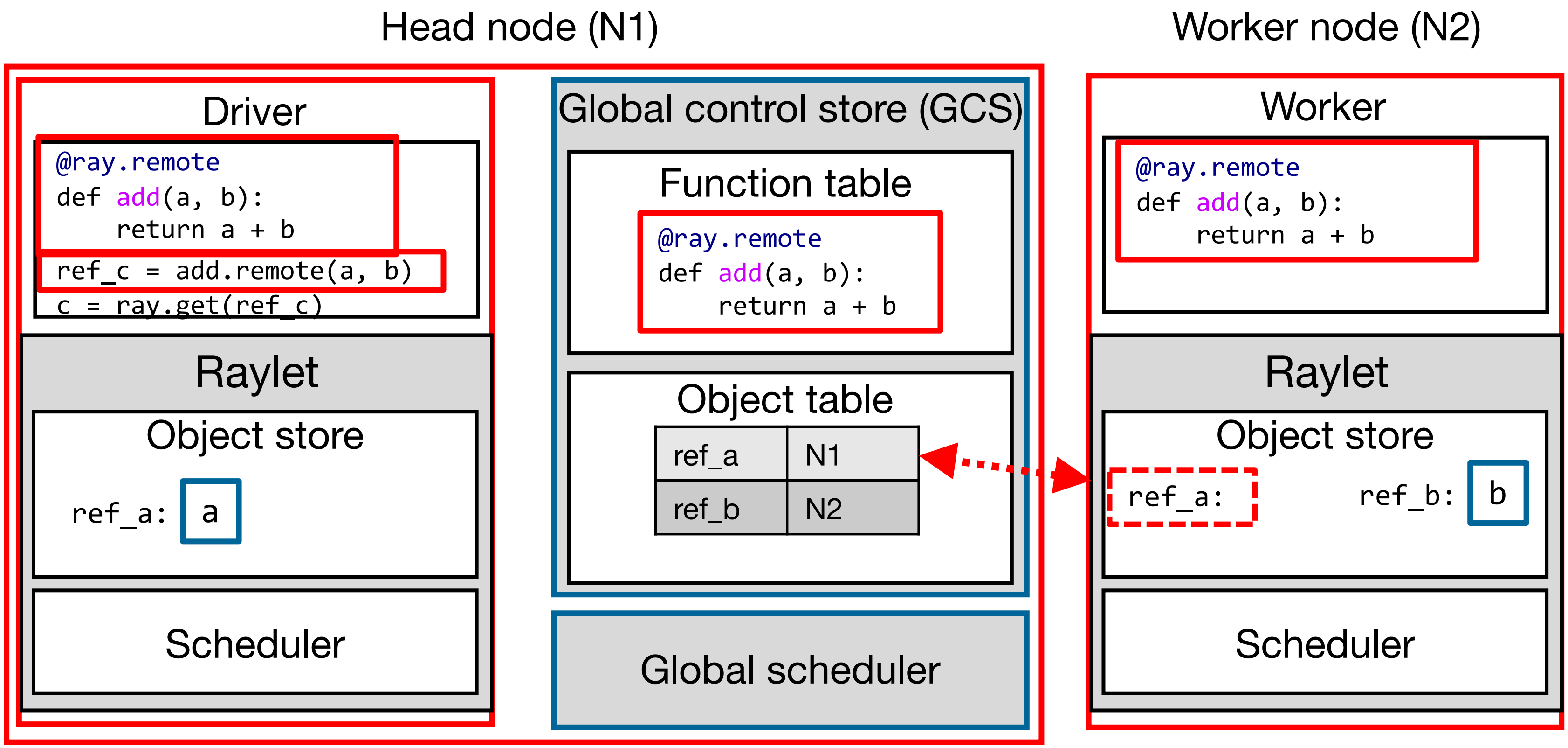
Scheduler

Cluster of machines

Step 5: N2's local scheduler checks whether the local object store contains `add(a,b)`'s arguments.

# Executing a task remotely

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

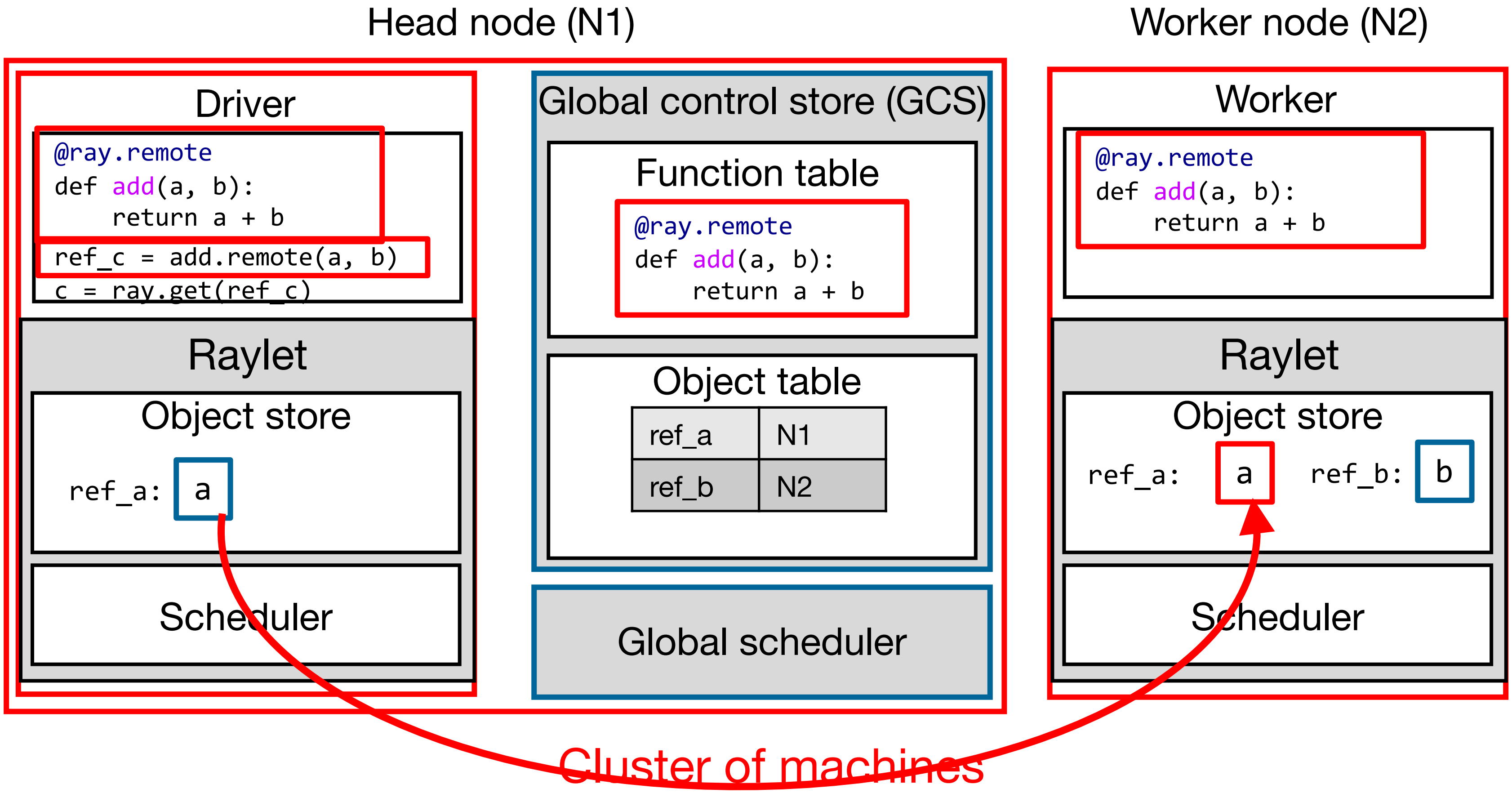


Cluster of machines

Step 6: N2 looks up a’s location in the GCS.

# Executing a task remotely

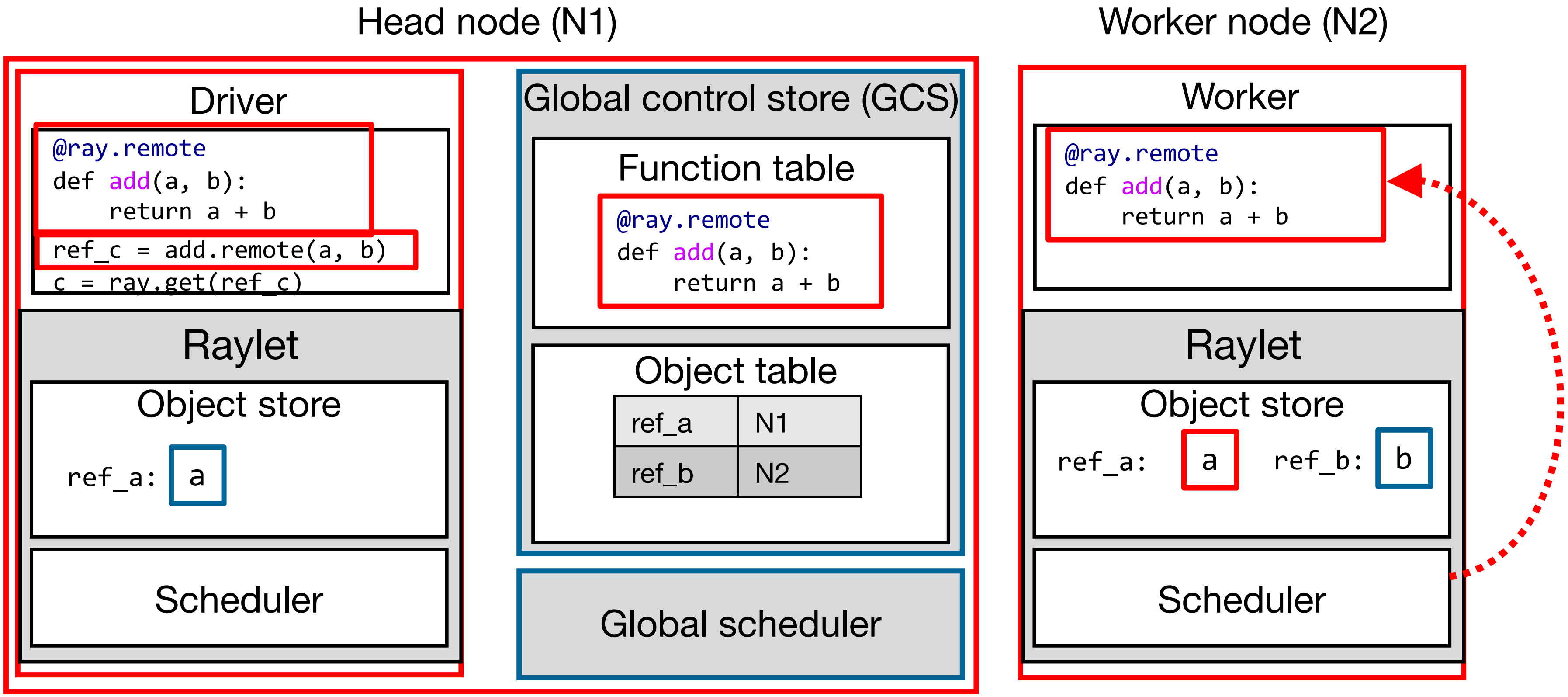
```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



Step 7: Learning that N1 holds a, N2 fetches object a from N1's object store and replicates it locally.

# Executing a task remotely

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

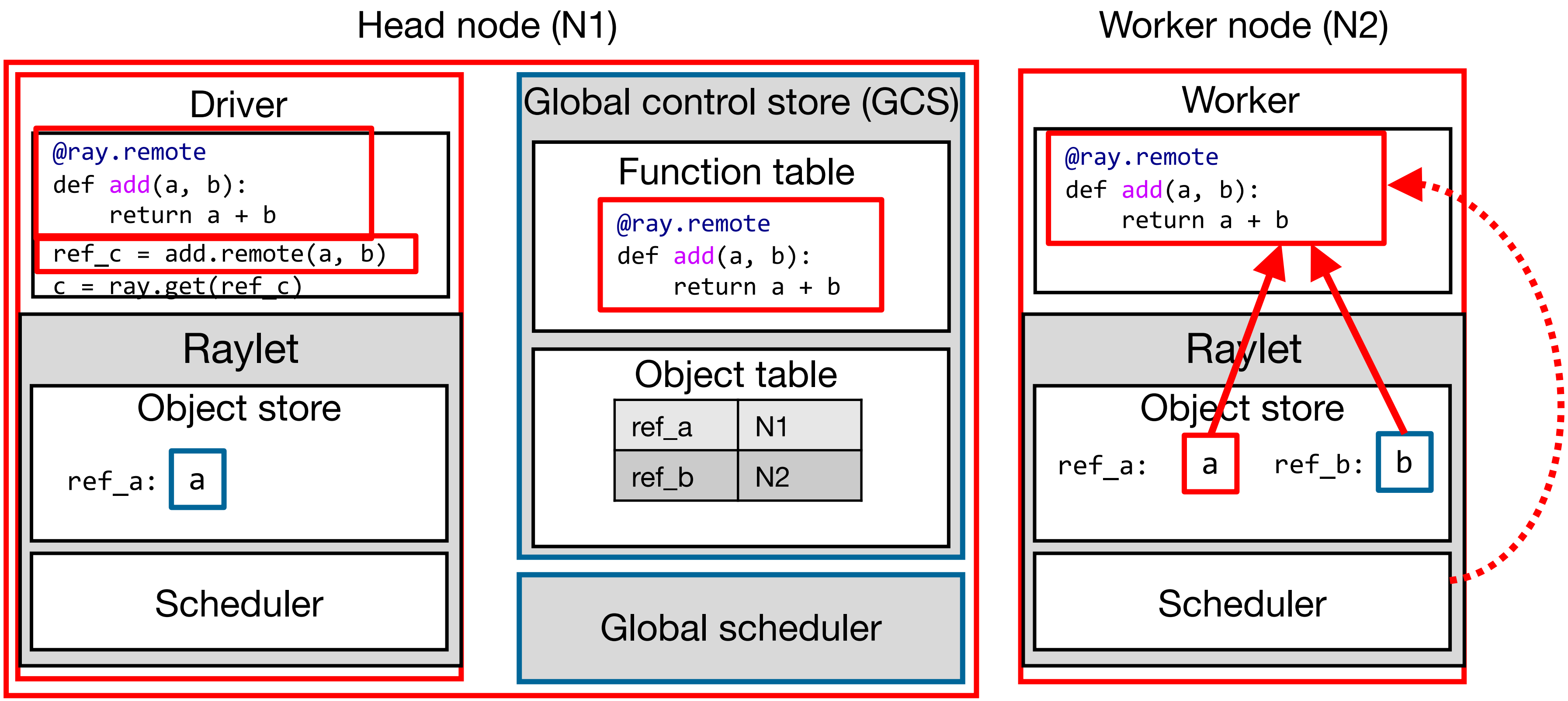
| ref_a | N1 |
|---|---|
| ref_b | N2 |

Cluster of machines

Step 8: N2’s local scheduler invokes the task function add() at N2’s local worker.

# Executing a task remotely

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |

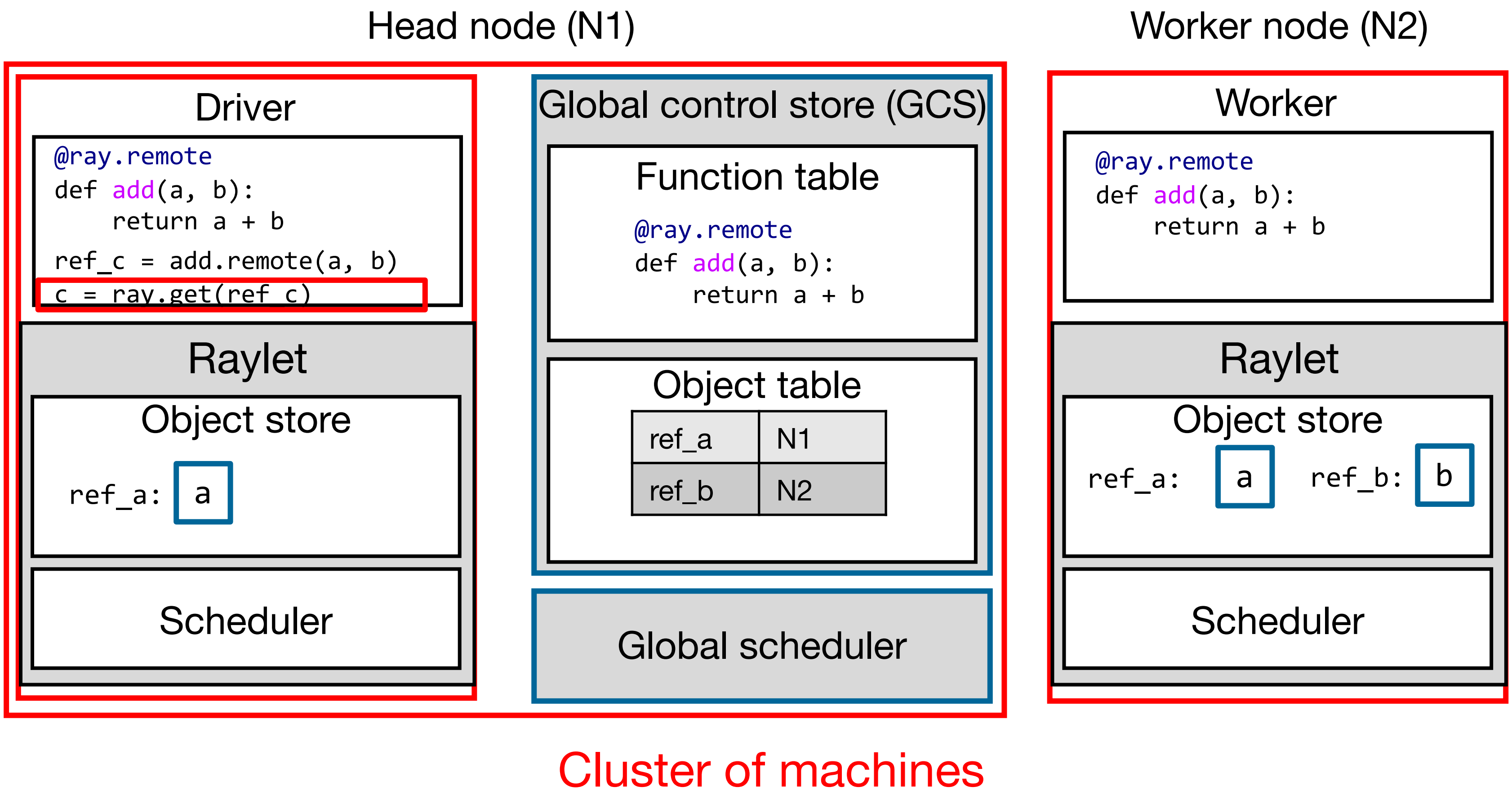
Cluster of machines

Step 9: N2's worker process executes the function code by accessing locally stored object a and b.

# Getting the result of a remote task w/ `ray.get()`

# Getting the result of a remote task w/ `ray.get()`

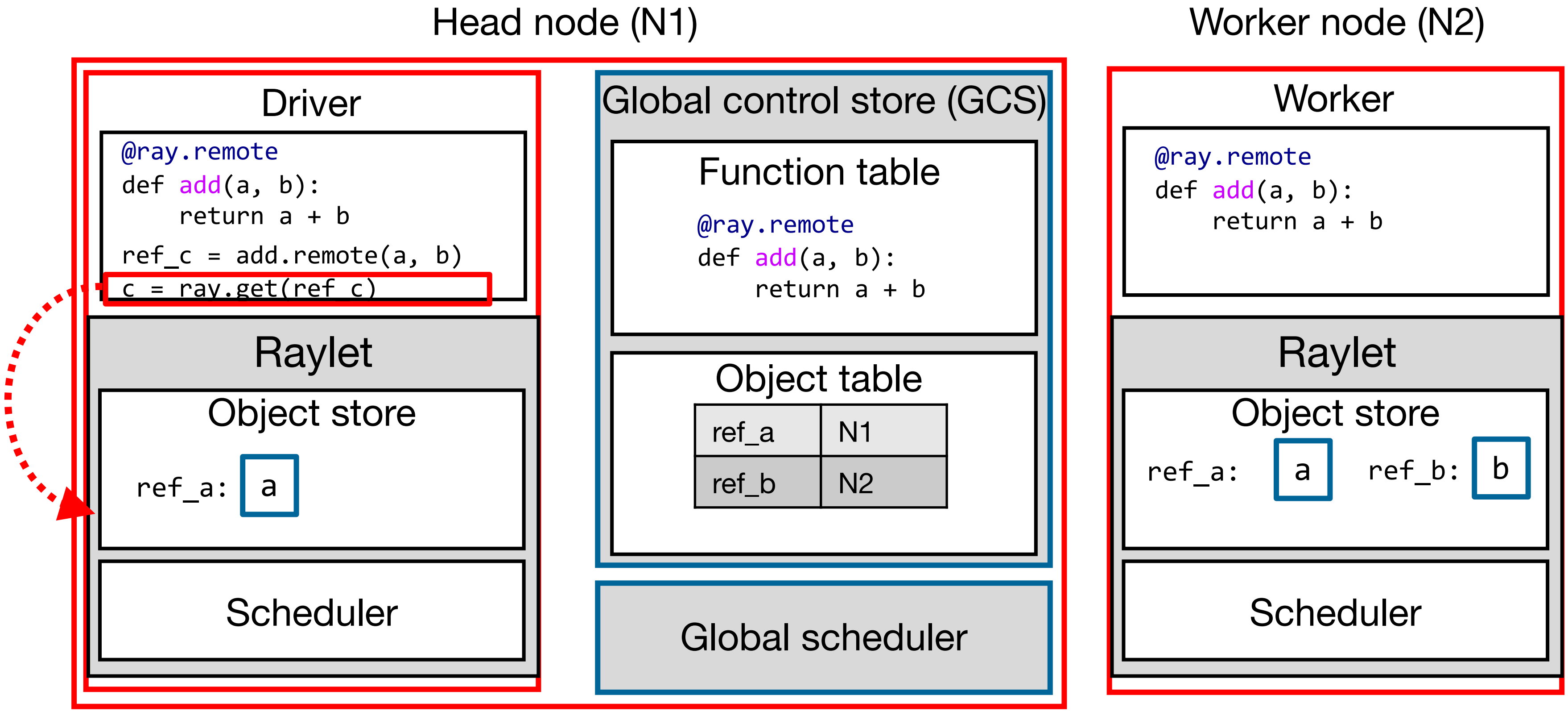
```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



Now, executing `ray.get(c)`…

# Getting the result of a remote task w/ `ray.get()`

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



Driver:

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

Function table:

```
@ray.remote
def add(a, b):
    return a + b
```

Object table:

| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |

Worker:

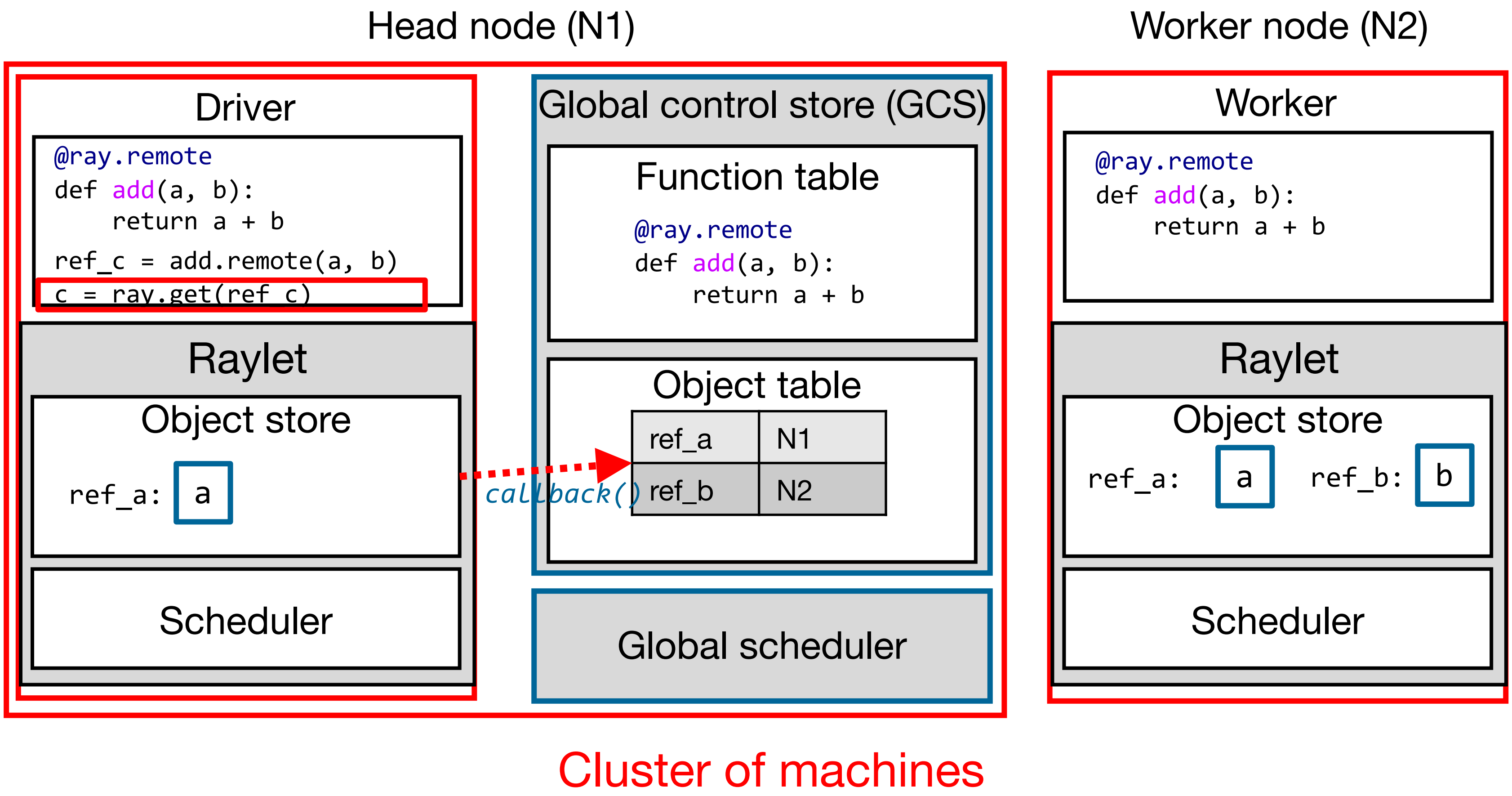
```
@ray.remote
def add(a, b):
    return a + b
```

Cluster of machines

Step 1: Driver checks local object store for object c using the future ref of c.

# Getting the result of a remote task w/ `ray.get()`

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

```
@ray.remote
def add(a, b):
    return a + b
```

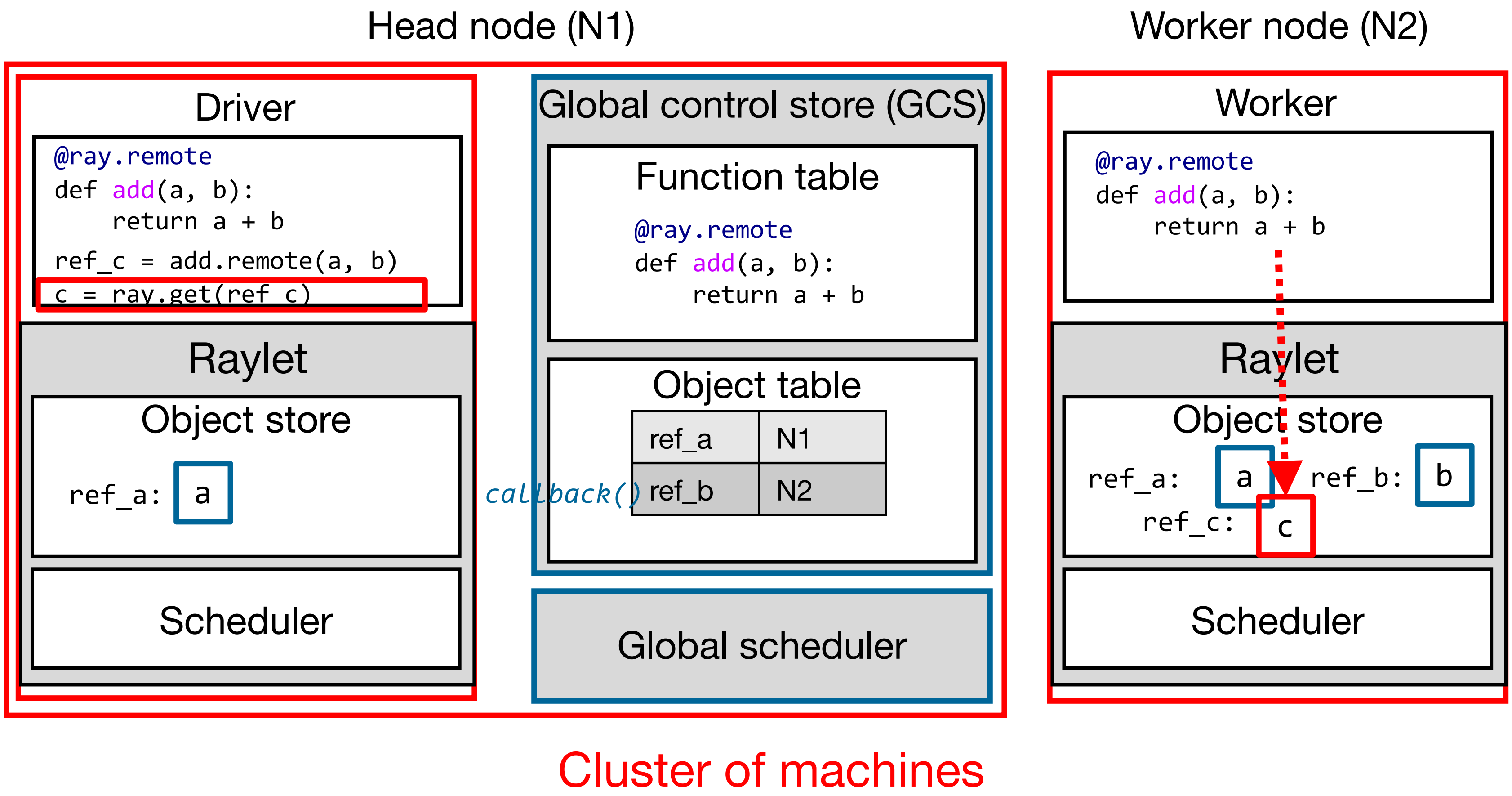
```
@ray.remote
def add(a, b):
    return a + b
```

| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |

Step 2: N1's local object store looks up c's location in GCS. GCS does not have an entry for c yet. Therefore, N1 registers a callback with GCS' object table to be triggered when c's entry is created.

# Getting the result of a remote task w/ `ray.get()`

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



Driver:

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

Function table:

```
@ray.remote
def add(a, b):
    return a + b
```

Worker:

```
@ray.remote
def add(a, b):
    return a + b
```

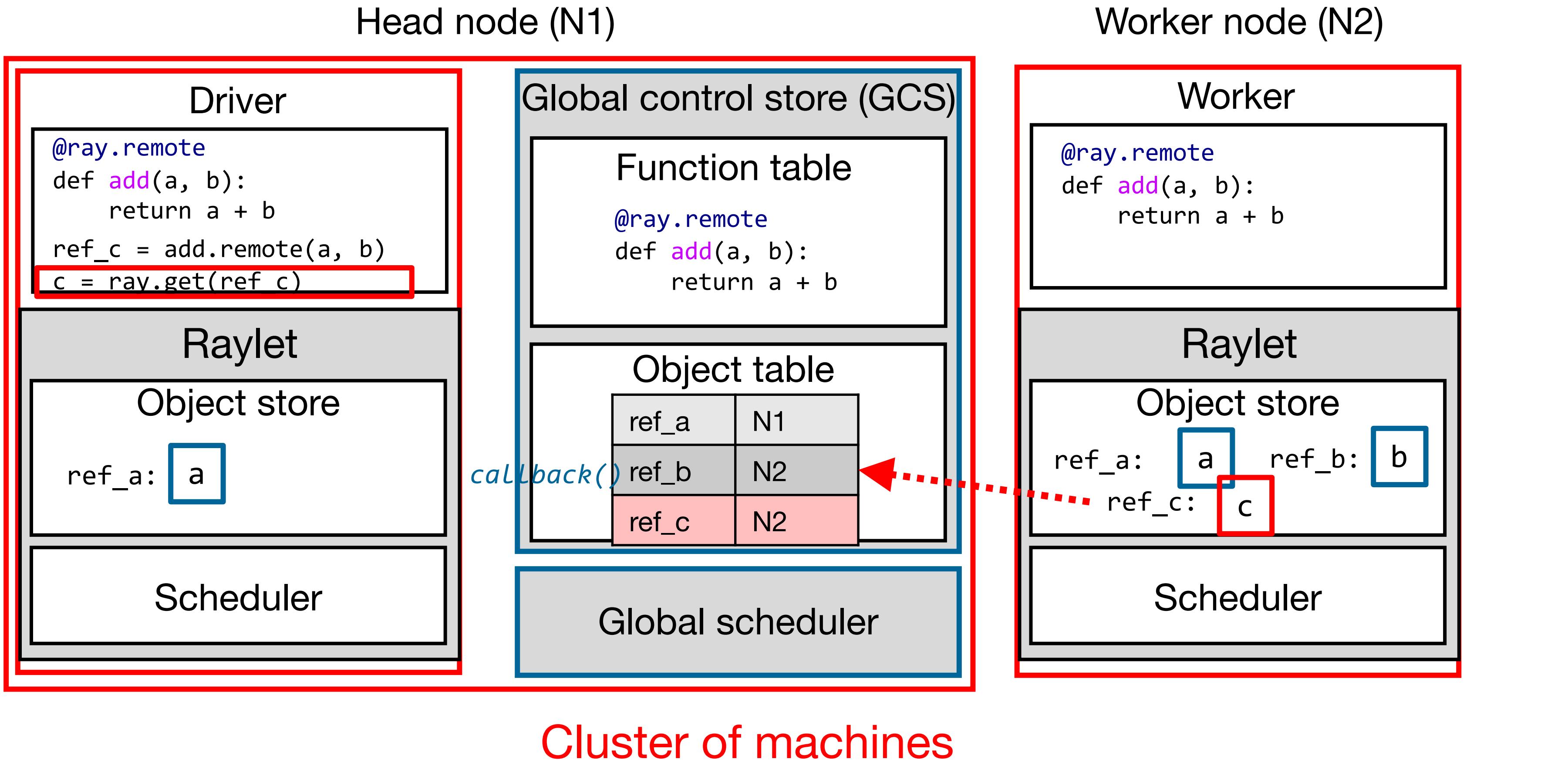
Object table:

| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |

Step 3: N2’s worker completes the execution of `add()` and stores the result c to the local object store.

# Getting the result of a remote task w/ `ray.get()`

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

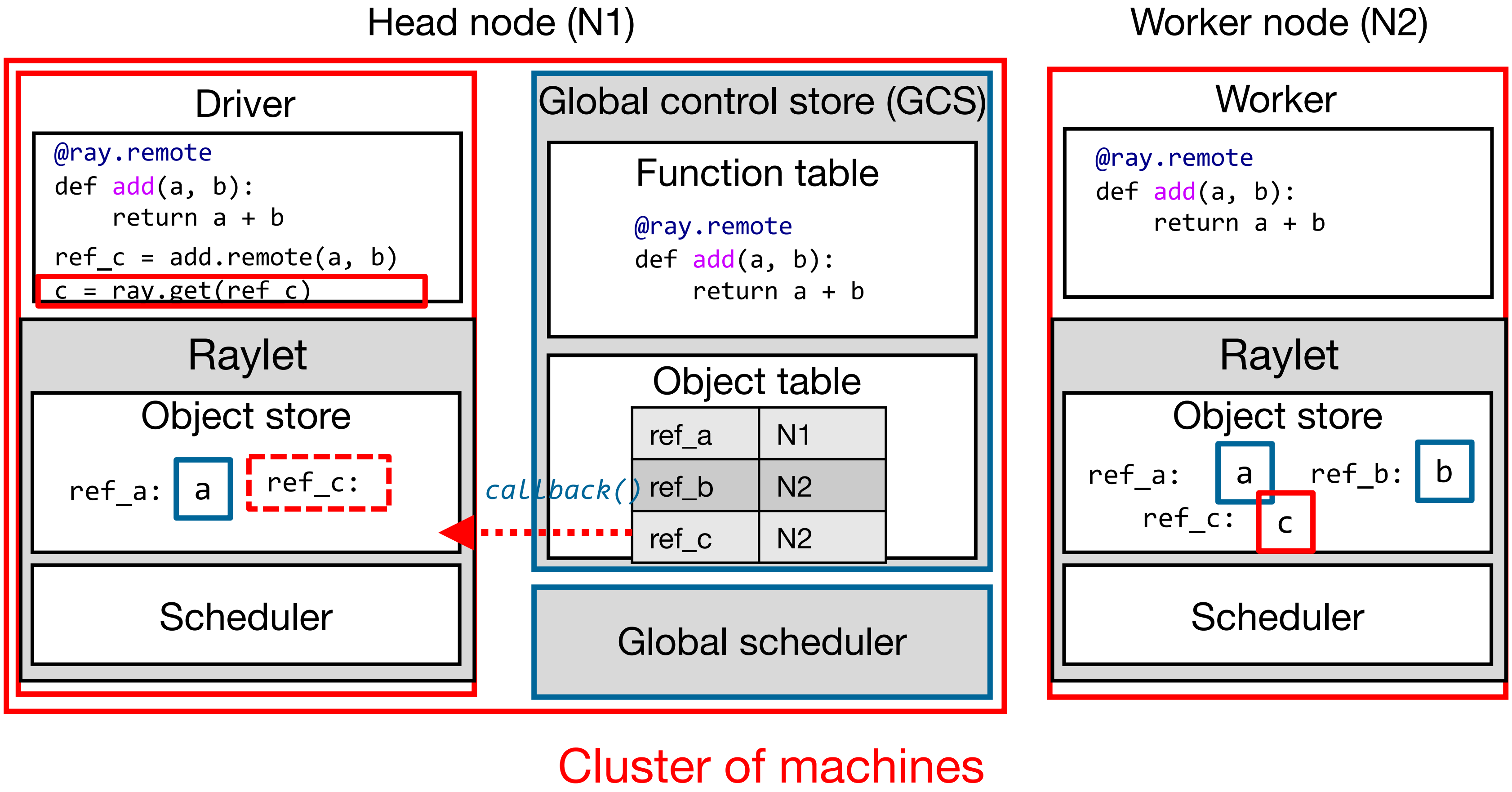
```
@ray.remote
def add(a, b):
    return a + b
```

| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |
| ref_c | N2 |

Step 4: N2's local object store in turn adds c's entry to GCS.

# Getting the result of a remote task w/ `ray.get()`

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

```
@ray.remote
def add(a, b):
    return a + b
```

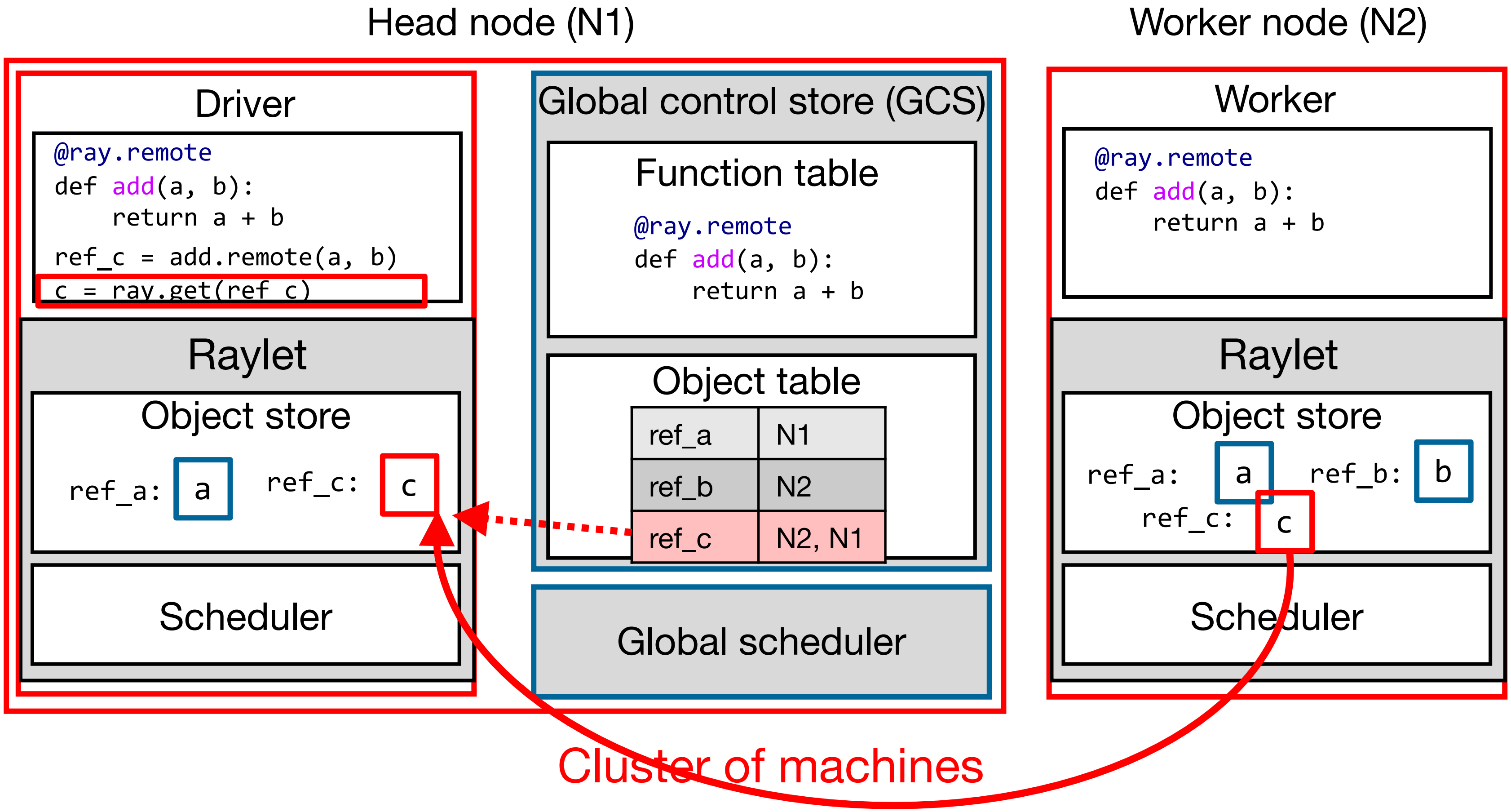
| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |
| ref_c | N2 |

```
@ray.remote
def add(a, b):
    return a + b
```

Step 5: GCS triggers the previously registered callback to N1's object store with c's entry.

# Getting the result of a remote task w/ `ray.get()`

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```



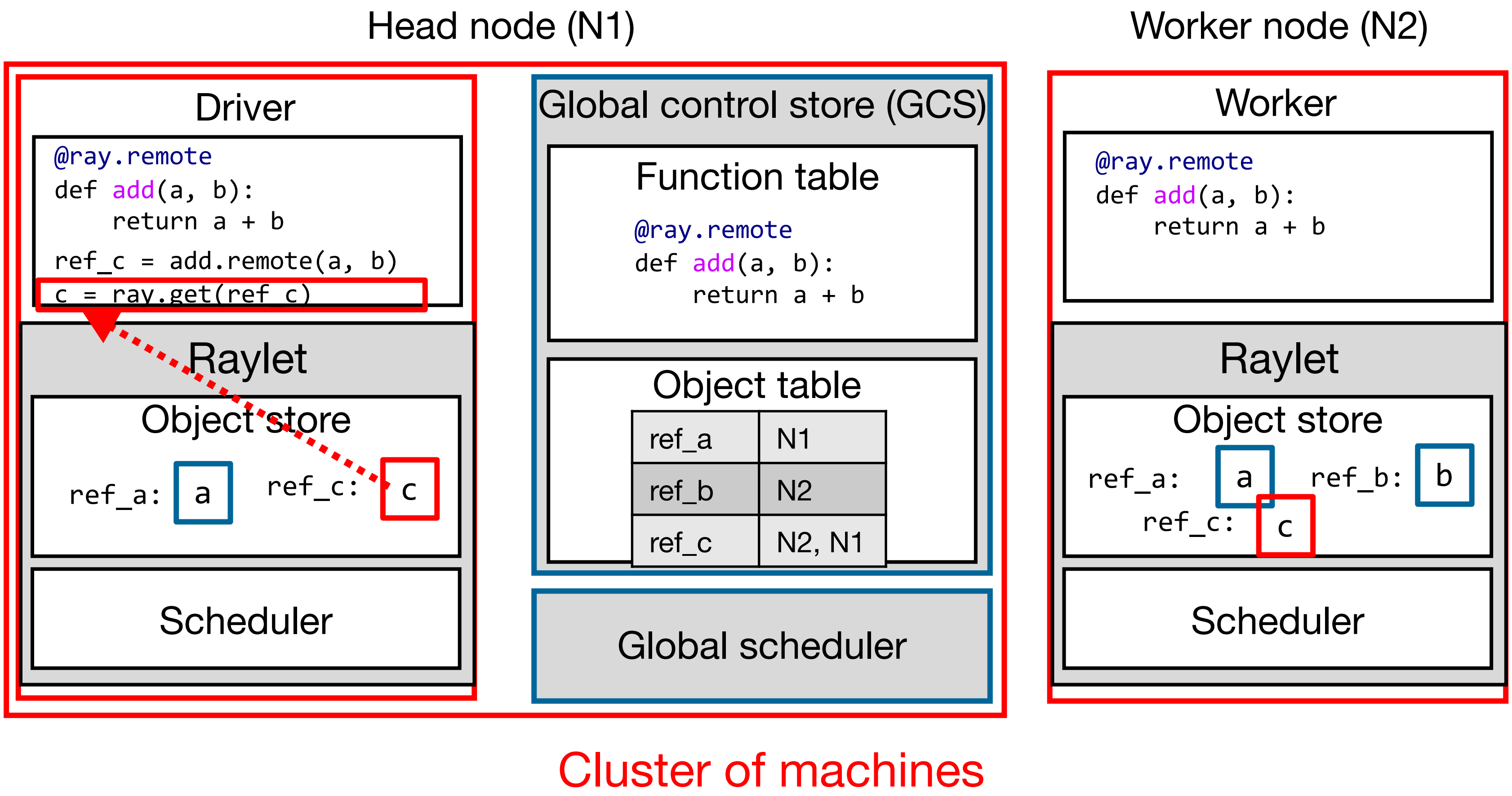
```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

| | |
|---|---|
| ref_a | N1 |
| ref_b | N2 |
| ref_c | N2, N1 |

Step 6: N1 fetches c from N2’s object store and replicates it in N1’s local object store.

# Getting the result of a remote task w/ `ray.get()`

```
@ray.remote
def add(a, b):
    return a + b
ref_c = add.remote(a, b)
c = ray.get(ref_c)
```

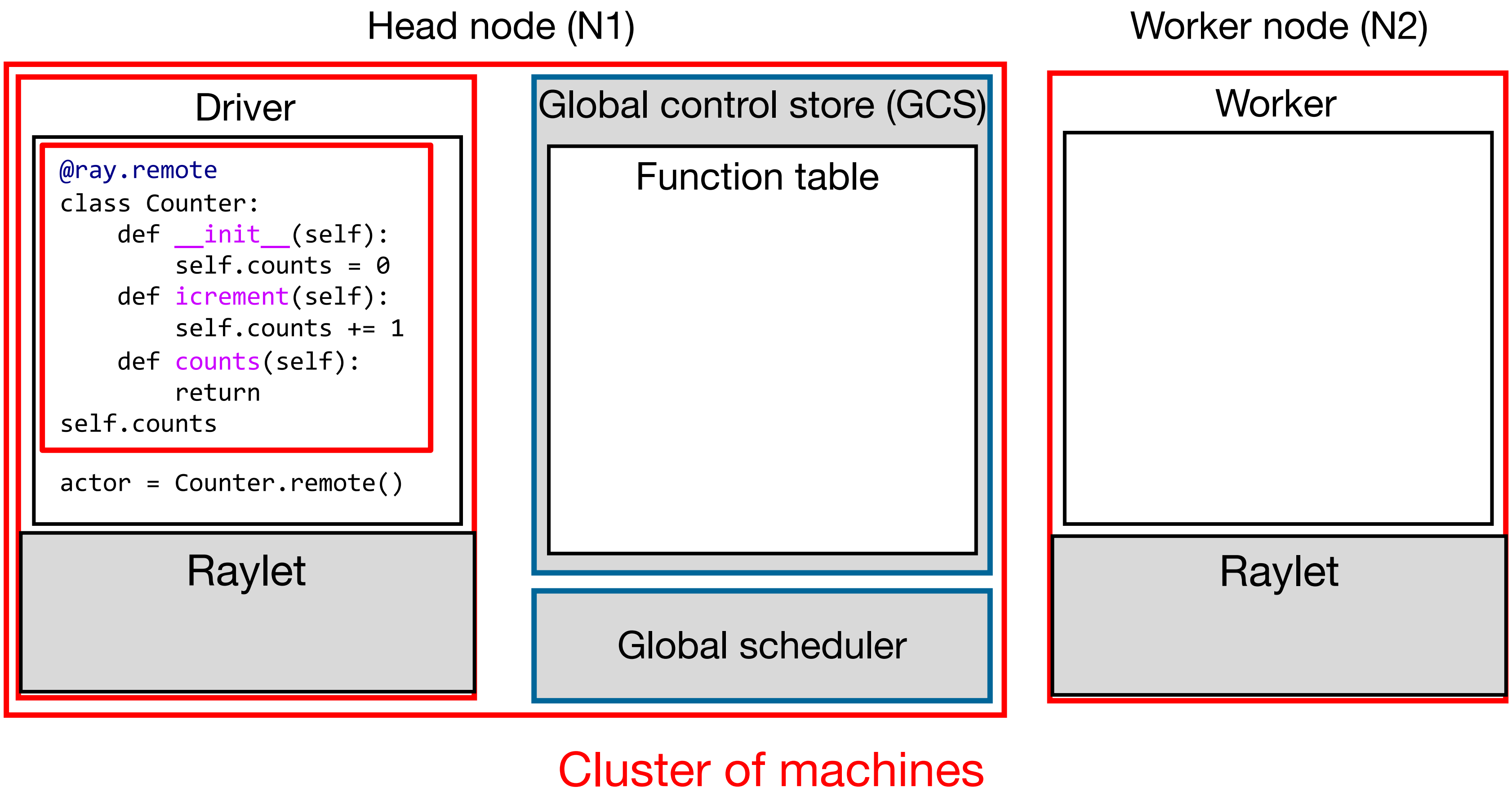


Step 7: N1's object store returns c to `ray.get()`.

# Actor management

# Actor creation

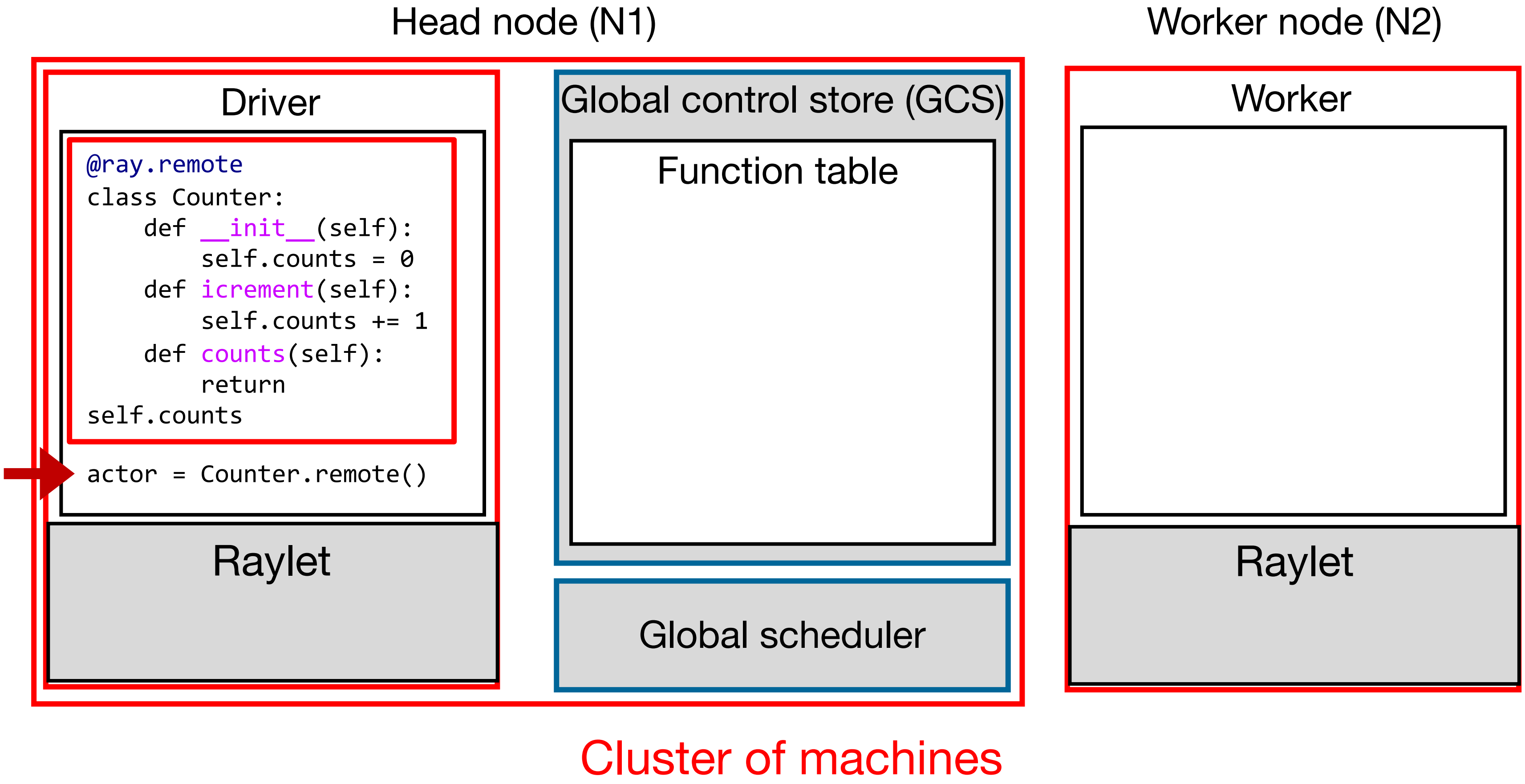
```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return self.counts
```



```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return
self.counts

actor = Counter.remote()
```

# Actor creation

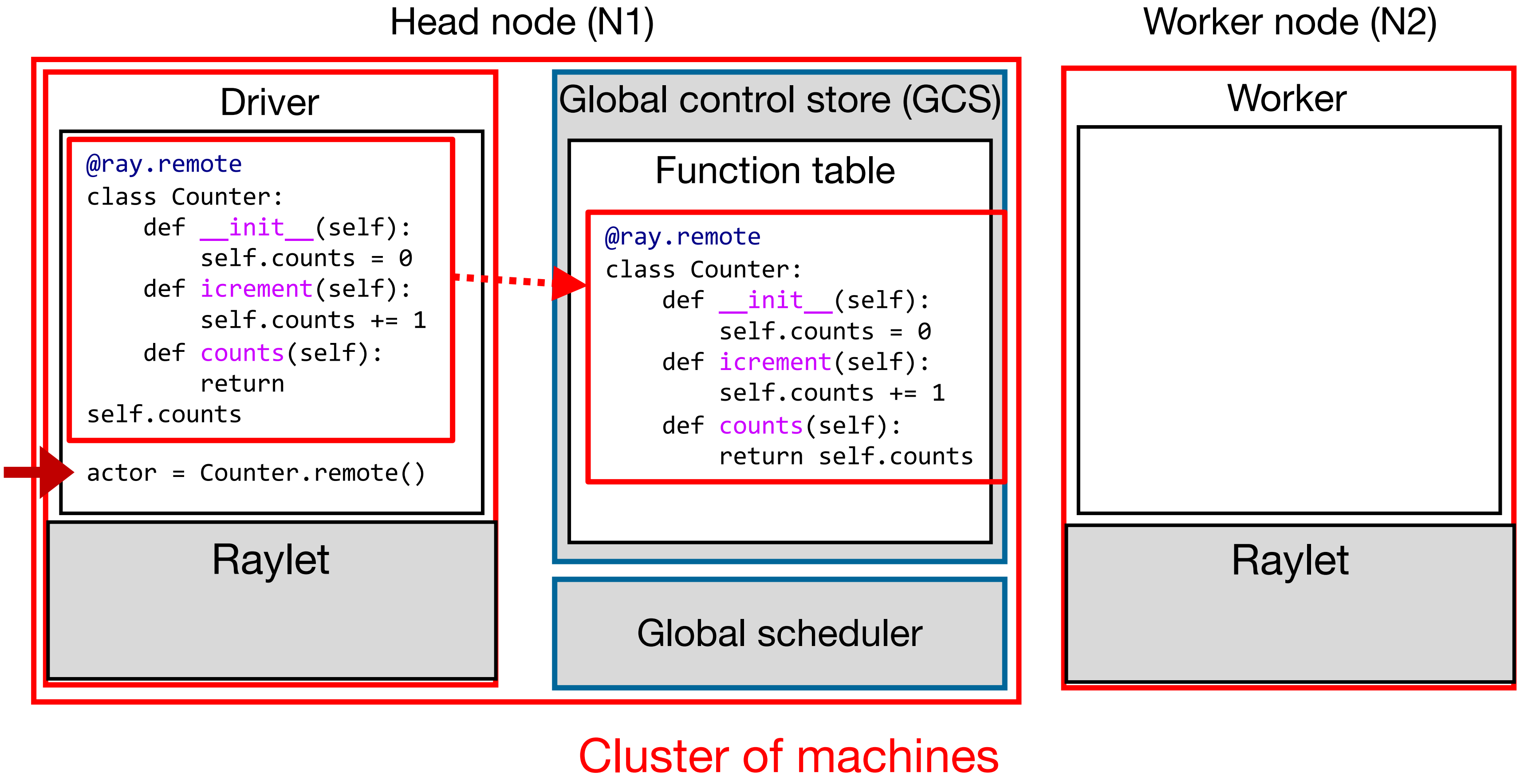
```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return self.counts
```



```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return
self.counts

actor = Counter.remote()
```

# Actor creation

```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return self.counts
```



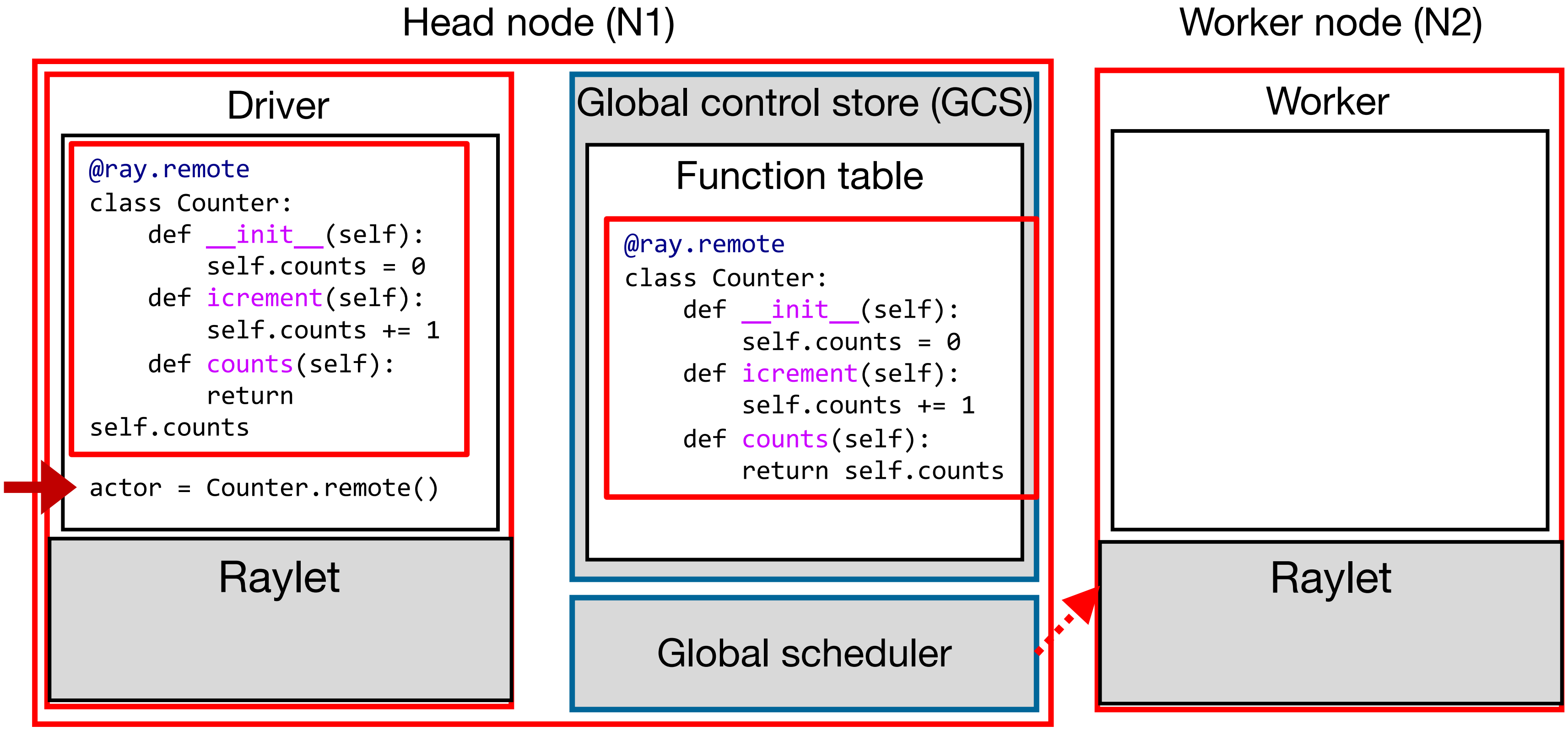
```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return
self.counts

actor = Counter.remote()
```

```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return self.counts
```

Step 1: Driver registers the actor with GCS.

# Actor creation

```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return self.counts
```



```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return
self.counts

actor = Counter.remote()
```

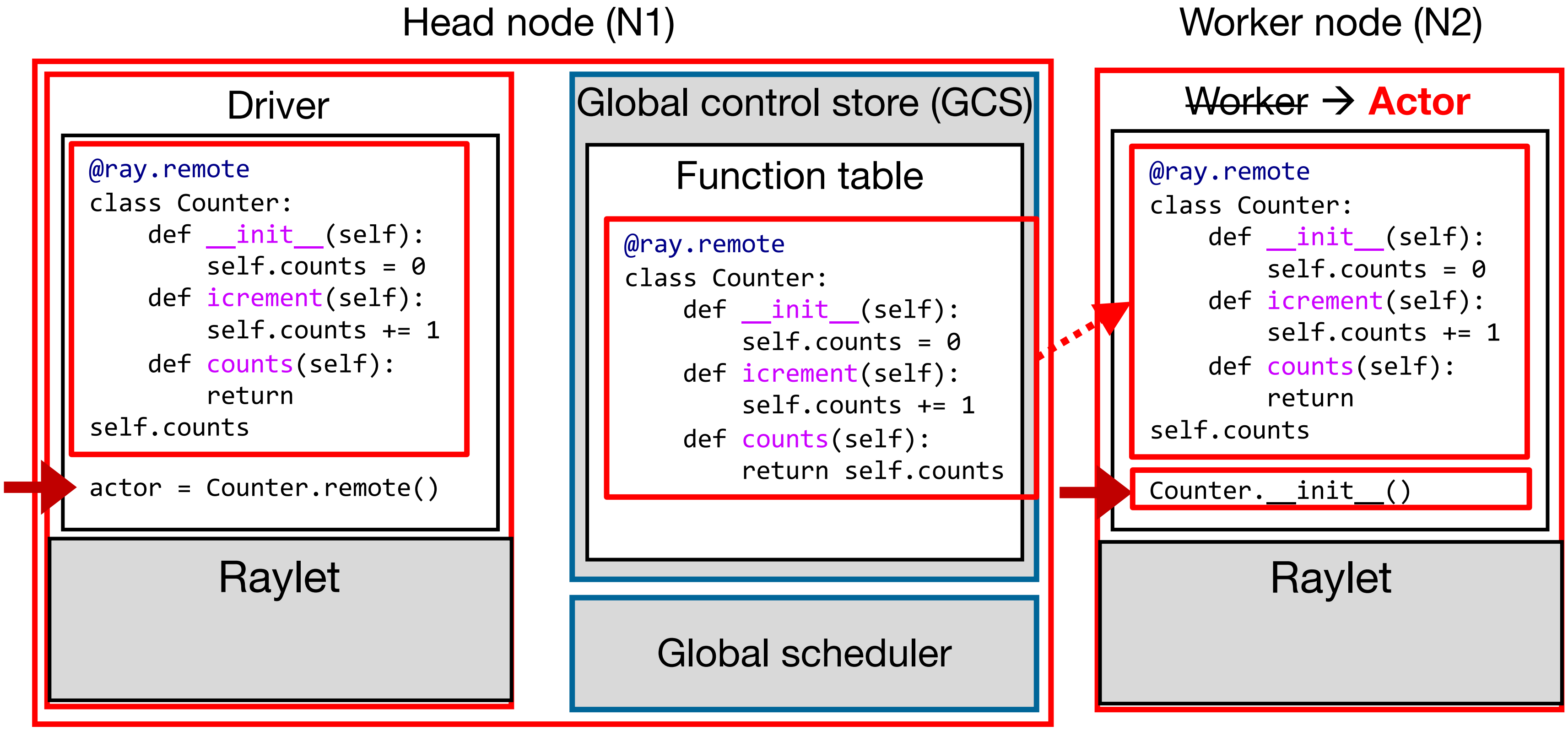
```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return self.counts
```

Cluster of machines

Step 2: Global scheduler selects a worker's raylet (N2), enqueue the actor creation request, and waits for the raylet to grant a resource lease.

# Actor creation

```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return self.counts
```



Head node (N1)

Worker node (N2)

Driver

```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return
self.counts

actor = Counter.remote()
```

Raylet

Global control store (GCS)

Function table

```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return self.counts
```

Global scheduler

~~Worker~~ → Actor

```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return
self.counts

Counter.__init__()
```

Raylet

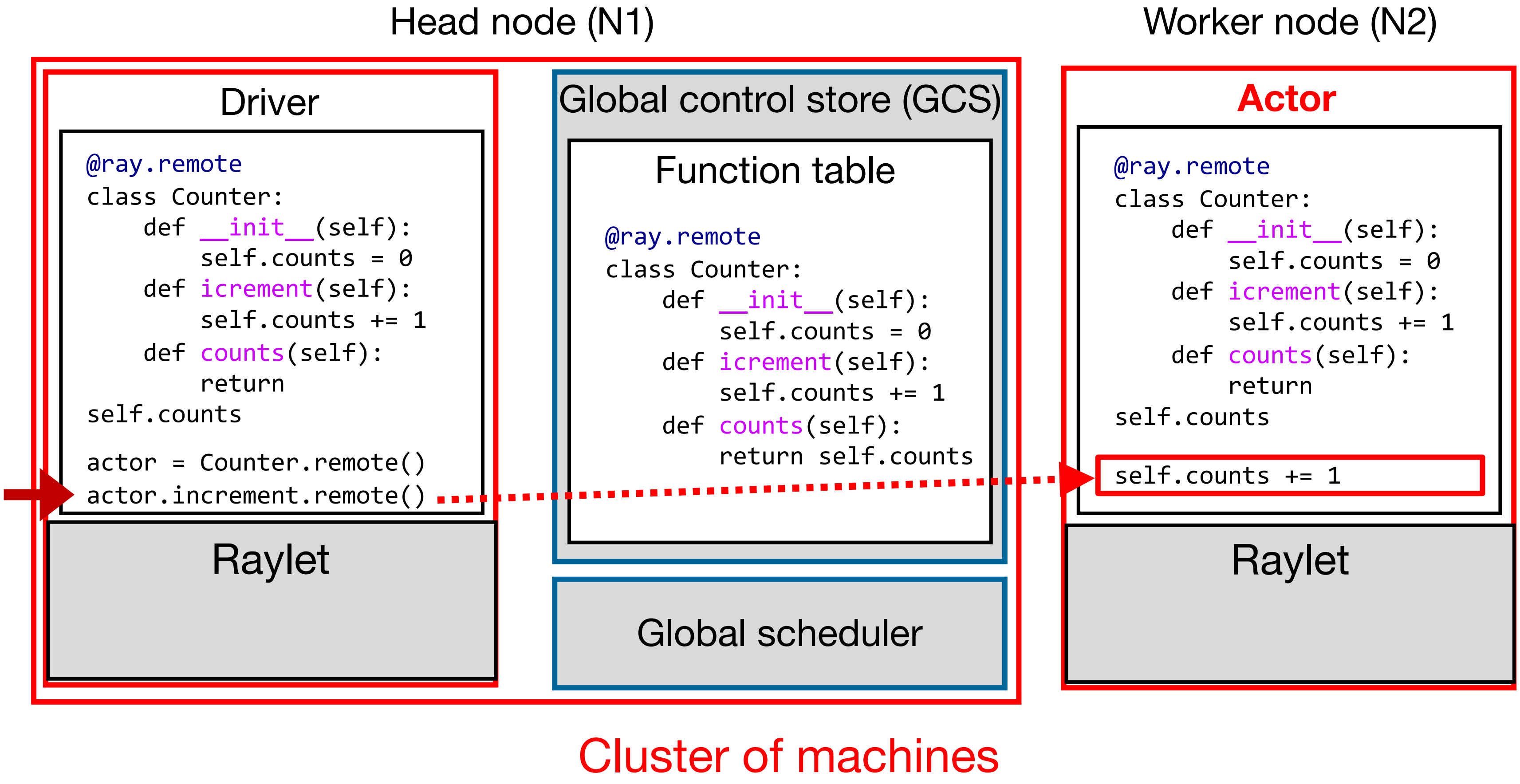
Cluster of machines

Step 3: Once resource is granted on N2, GCS schedules the actor creation task on N2. N2 now is effectively an Actor.

# Actor task execution

# Actor task execution

`actor.increment.remote()`



```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return
self.counts

actor = Counter.remote()
actor.increment.remote()
```

```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return self.counts
```

```
@ray.remote
class Counter:
    def __init__(self):
        self.counts = 0
    def icrement(self):
        self.counts += 1
    def counts(self):
        return
self.counts

self.counts += 1
```

Actor tasks are sent via remote function calls to the Actor process (N2).

# Quiz 5 and Demo …

# Ray Core API summary

## Tasks

```
futures = f.remote(args)
```

Execute function f remotely. `f.remote()` can take objects or futures as inputs and returns one or more futures. This is **non-blocking**.

## Actors

```
actor = Class.remote(args)
futures = actor.method.remote(args)
```

Instantiate class `Class` as a remote actor and return a handle to it.
Call a method on the remote actor and return one or more futures.
Both are **non-blocking**.

```
objects = ray.get(futures)
```

Return the values associated with one or more futures. **This is blocking**.

```
ready_futures = ray.wait(futures, k, timeout)
```

Return the futures whose corresponding tasks have completed as soon as either k have completed or the timeout expires.