

# Actor-Based Distributed Systems

Marek Konieczny

[marekko@agh.edu.pl](mailto:marekko@agh.edu.pl),

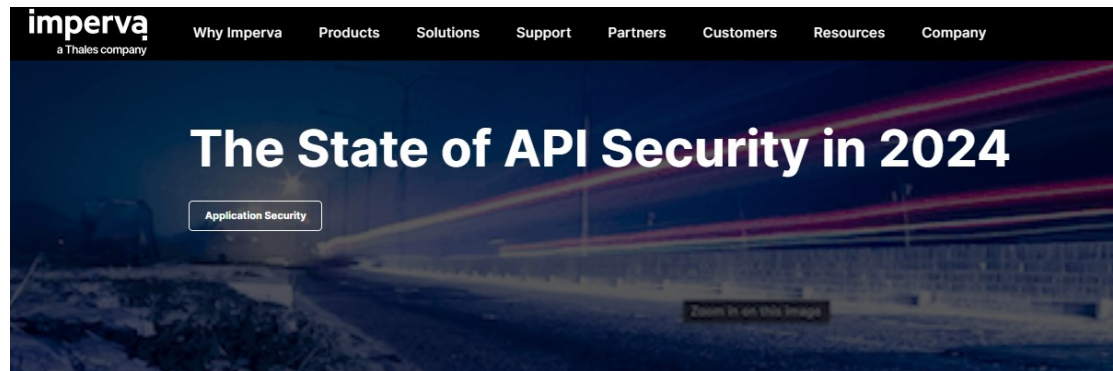
Room 4.43, Spring 2024



AKADEMIA GÓRNICZO-HUTNICZA  
IM. STANISŁAWA STASZICA  
W KRAKOWIE

# Notes on REST API

- The most common security issues:
  - Shadow API
  - Deprecated API
  - Unauthorized API



## **API calls make up a massive 71% of all web traffic**

Widespread API usage is expanding the attack surface for bad actors. These threats make it critical for organizations to understand the risks and complexities of APIs and the importance of API Security.

# Class Logistics

- Laboratory classes during week 25.3-5.4
  - There is no meeting on Wednesday 3.4
- We will use desktops from Computer Networks Laboratory
- Preliminary plan homework assignment
  - Elastic schedule on week 25-29.4

# Class Logistics

- Laboratory grading
  - Showing up => 1 pt
  - UPEL task upload => 1 pt
  - Hard work on lab meeting => 2 pt
    - You need to show me logs of your tasks execution on Ray Dashboard
  - Extra activity, labs improvement => 1 pt

# Class Logistics

- During our meeting we will use Ray Framework
- Install all required components
  - *pip install -U "ray[default]"*
  - <https://docs.ray.io/en/latest/ray-overview/installation.html>
- Important *pip install ray* is not enough !!!

# Actors

- Mathematical model of concurrent computation proposed by Carl Hewitt in 1973

Session 8 Formalisms for  
Artificial Intelligence

## A Universal Modular ACTOR Formalism for Artificial Intelligence

Carl Hewitt

Peter Bishop

Richard Steiger

### Abstract

This paper proposes a modular ACTOR architecture and definitional method for artificial intelligence that is conceptually based on a single kind of object: actors [or, if you will, virtual processors, activation frames, or streams]. The formalism makes no presuppositions about the representation of primitive data structures and control structures. Such structures can be programmed, micro-coded, or hard wired in a uniform modular fashion. In fact it is impossible to determine whether a given object is "really" represented as a list, a vector, a hash table, a function, or a process. The architecture will efficiently run the coming generation of PLANNER-like artificial intelligence languages including those requiring a high degree of parallelism. The efficiency is gained without loss of programming generality because it only makes certain actors more efficient; it does not change their behavioral characteristics. The architecture is general with respect to control structure and does not have or need goto, interrupt, or semaphore primitives. The formalism achieves the goals that the disallowed constructs are intended to achieve by other more structured methods.

PLANNER Progress

# Actors

- The actor is an object that encapsulates state and behavior
- Originated from
  - Scheme, Smalltalk, Petri Nets, Channels ...
  - More details on theory in attached document
- More recent
  - *Scheme* made them concrete
  - *Erlang* made them useful
  - *Akka* made them cool
  - *Ray* makes them easy

# Actors

- Primitives for concurrency/parallelism
- Entities having a message queue and associated behavior → **And isolated state!!**
- Can exchange messages between each-other
- When an actor receives a message it can:
  - Send a finite number of messages to other actors
  - Create a finite number of new actors
  - Modify its internal behavior on receiving messages



# Actors

- Messages between actors are always sent asynchronously
- No requirement on order of message arrival
- Queuing and dequeuing of messages in an actor mailbox are atomic operations, **no race conditions anymore**
- Fire and Forget approach

# Origins

- Originated on UC Berkeley, Ion Stoica:
  - Spark, Messos, Ray
  - Managed by Anyscale

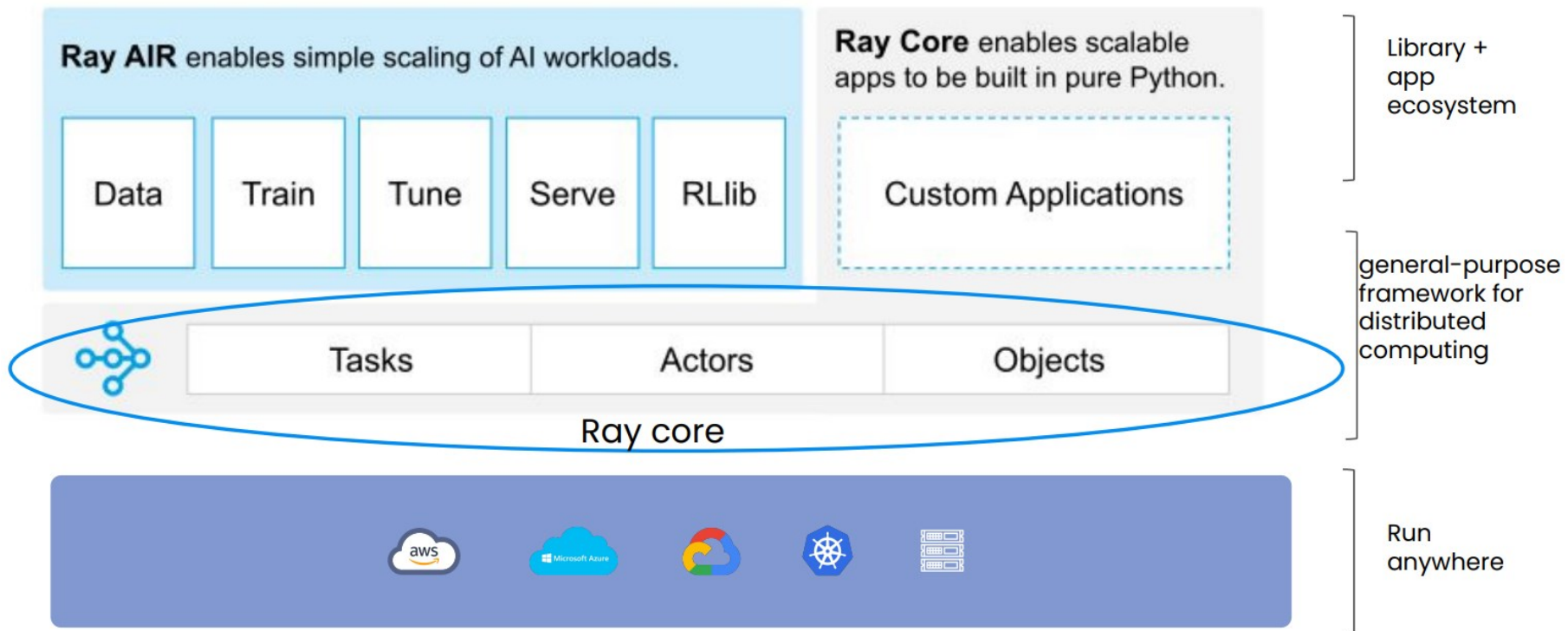
The next generation of AI infrastructure is being built and scaled on Ray



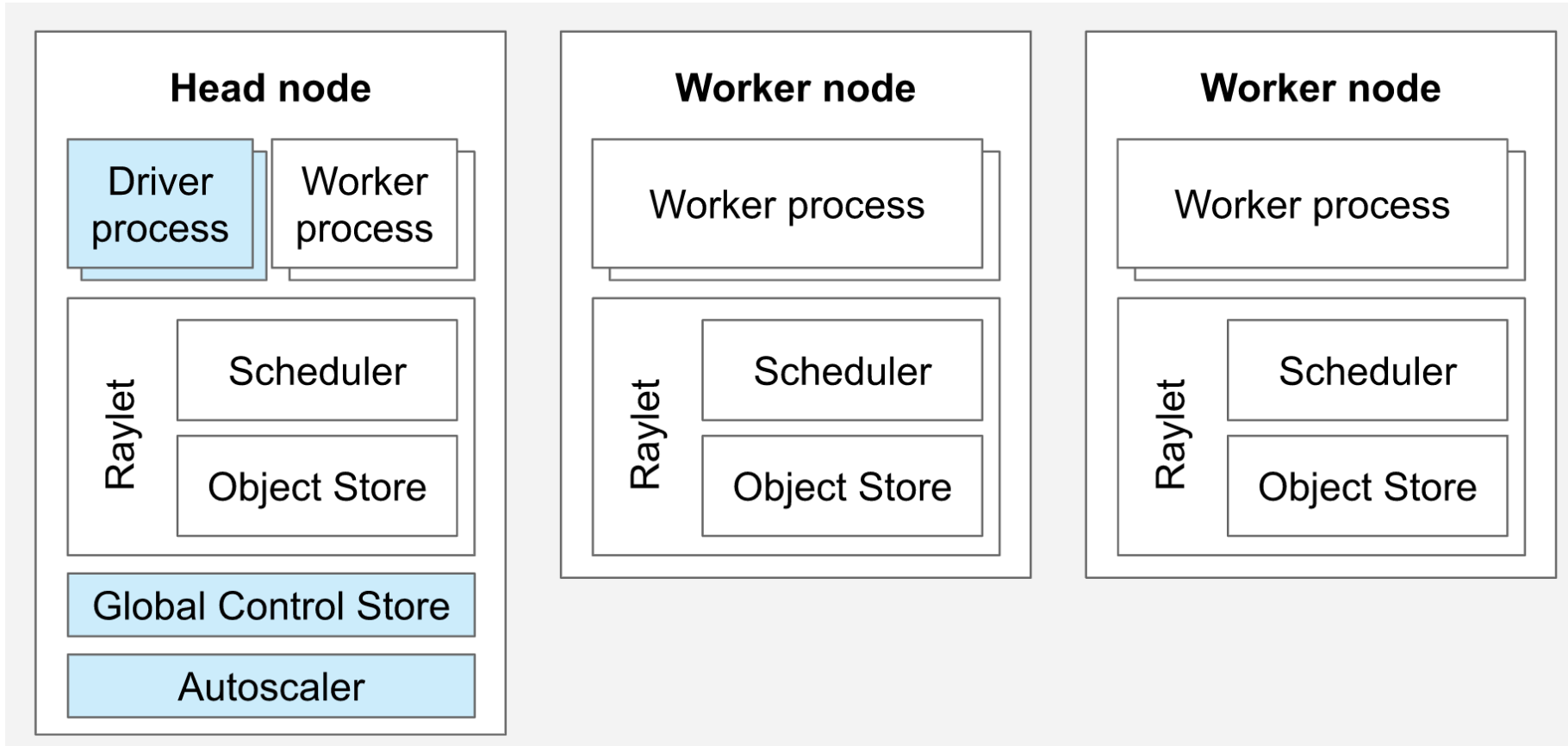
# The FAST Compute Model

- **Futures:** references to objects
- **Actors:** remote class instances (objects)
- **Shared in-memory distributed object store**
- **Tasks:** remote functions

# What is Ray ?

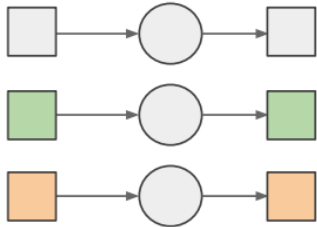


# What is Ray ?



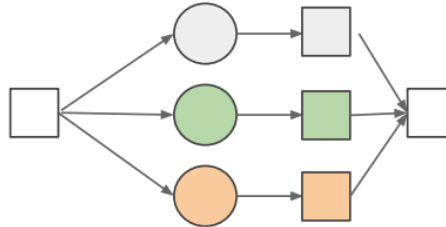
# Scaling patterns

**Batch Training / Inference**



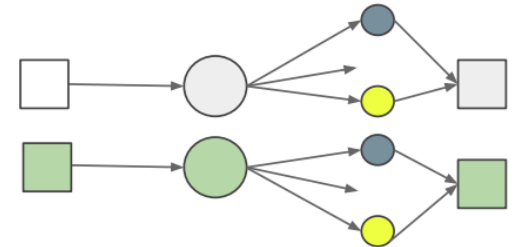
**Different data / Same function**

**AutoML**

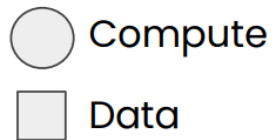


**Same data / Different function**

**Batch Tuning**



**Different data / Same function**



# What is Ray ?

Python → Ray APIs



```
def f(x):  
    # do something with x:  
    y= ...  
    return y
```

Task



```
@ray.remote  
def f(x):  
    # do something with x:  
    y= ...  
    return y
```

Distributed



...



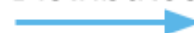
```
class Cls():  
    def __init__(self,  
x):  
    def f(self, a):  
        ...  
    def g(self, a):  
        ...
```

Actor

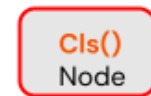


```
@ray.remote  
class Cls():  
    def  
__init__(self, x):  
    def f(self, a):  
        ...  
    def g(self, a):  
        ...
```

Distributed



...



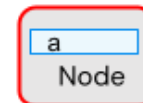
```
import numpy as np  
a= np.arange(1, 10e6)  
b = a * 2
```

Distributed  
immutable  
object

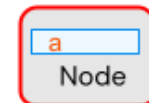


```
import numpy as np  
a = np.arange(1, 10e6)  
obj_a = ray.put(a)  
b = ray.get(obj_a) * 2
```

Distributed



...

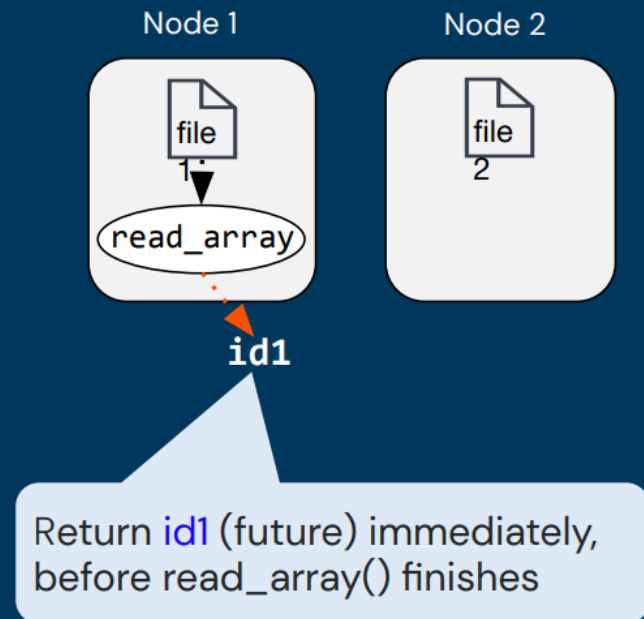


# Task API

```
@ray.remote
def read_array(file):
    # read ndarray "a"
    # from "file"
    return a

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

id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```



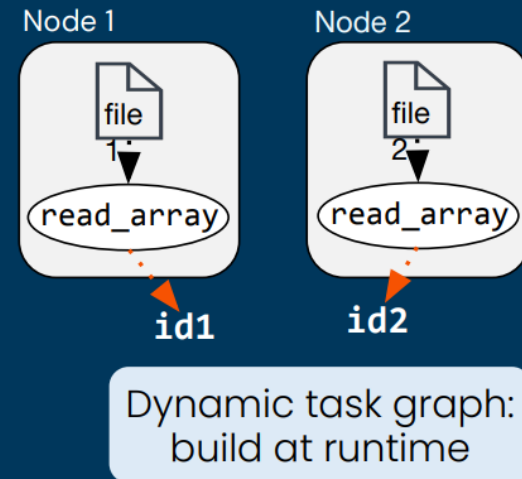


# Task API

```
@ray.remote
def read_array(file):
    # read ndarray "a"
    # from "file"
    return a

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

id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```



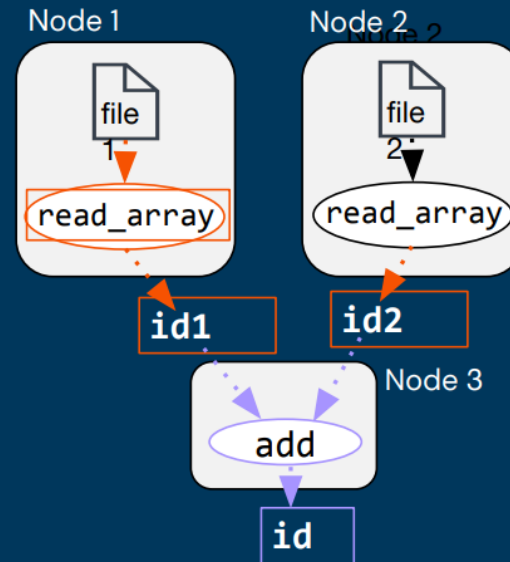
# Task API

```
@ray.remote
def read_array(file):
    # read ndarray "a"
    # from "file"
    return a
```

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

```
id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```

`ray.get()` block until  
result available

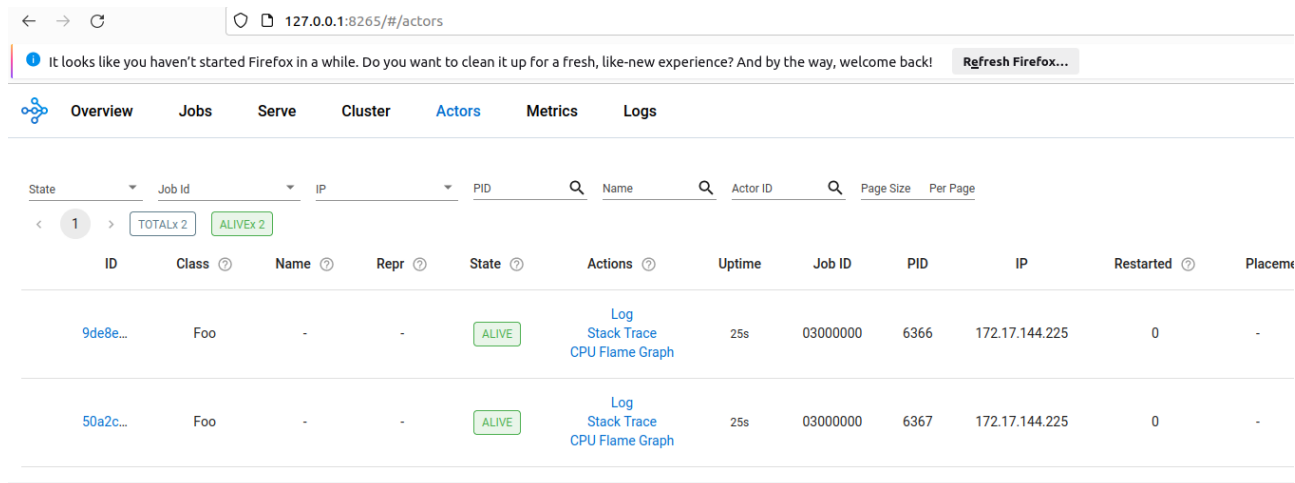


# Task 0

- *Use Ubuntu Image in Network Lab*
- Start a node
  - *ray start --head --port=6379*
- Check the dashboard on :
  - <http://127.0.0.1:8265/>
- Copy notebook to folder and start jupyter service
  - *jupyter notebook*

# Task 1

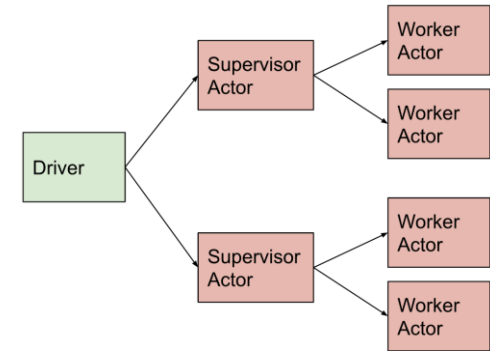
- Open and fill all the gaps in attached notebook
  - All important aspects of ray framework are covered
  - At the end there is main pi computation task
- Your dashboard should look like:



The screenshot shows the Ray dashboard interface. At the top, there's a navigation bar with tabs: Overview, Jobs, Serve, Cluster, Actors (selected), Metrics, and Logs. Below the navigation bar, there's a table of actors. The table has columns: ID, Class, Name, Repr, State, Actions, Uptime, Job ID, PID, IP, Restarted, and Placement. Two actors are listed, both in the 'ALIVE' state.

ID	Class	Name	Repr	State	Actions	Uptime	Job ID	PID	IP	Restarted	Placement
9de8e...	Foo	-	-	ALIVE	<a href="#">Log</a> <a href="#">Stack Trace</a> <a href="#">CPU Flame Graph</a>	25s	03000000	6366	172.17.144.225	0	-
50a2c...	Foo	-	-	ALIVE	<a href="#">Log</a> <a href="#">Stack Trace</a> <a href="#">CPU Flame Graph</a>	25s	03000000	6367	172.17.144.225	0	-

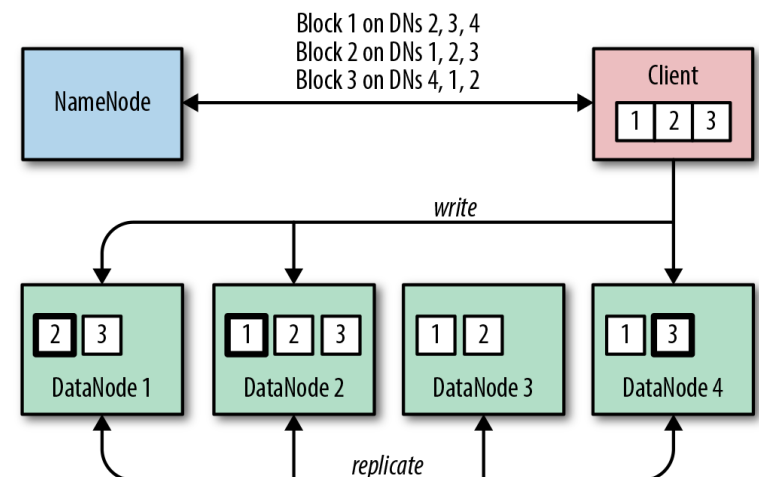
# Task 2



- Implement a distributed artifacts storage system:
  - there is one name node and x storage node (all actors)
  - execute it in a separate script in the system terminal/console
  - name node distributes artifacts to storage nodes in chunks and keeps track of it
  - there are few copies of chunks to improve high availability
  - randomly, data storage nodes can fail; the name node keeps track of all chunks and manages the copies
  - provide a list operation of the most important data

# Task 2

- User can perform the following operations from the interactive shell
  - upload - by providing an artifact name and content as a string
  - update the artifact content already present in the storage
  - user can delete artifacts present in the storage
  - user can get the artifacts from the storage
  - user can list the status of the cluster



# Task 2

- Verification tests:
  - upload an artifact, list the status
  - update an artifact, list the status
  - get an artifact, list the status
  - delete an artifact, list the status

# Homework

- Tier 1 (4 pts)
  - Prepare and present all exercises from labs from Task 1 => 4 pts
- Tier 2 (6-8 pts)
  - Prepare implementation of the Task 2 from lab
  - Present demo solution and test scenarios => 7 pts
  - Add some additional advanced functionality, select anything from items not discussed on labs (<https://docs.ray.io/en/latest/ray-core/actors.html>) => 8 pts
  - Add distribution of environment (e.g. execute it on cluster based on K8s, docker ...) => 9 pts



# Homework

- Tier 3 (9 pts)
  - Prepare sample project with other Ray modules
  - Select 3 different modules
  - Prepare tests form your solution
  - Present how you move it from regular implementation to demo on ray
  - Execute them on top of distributed environment  
(+ 1 pt = 10 total)

