Dask Distributed

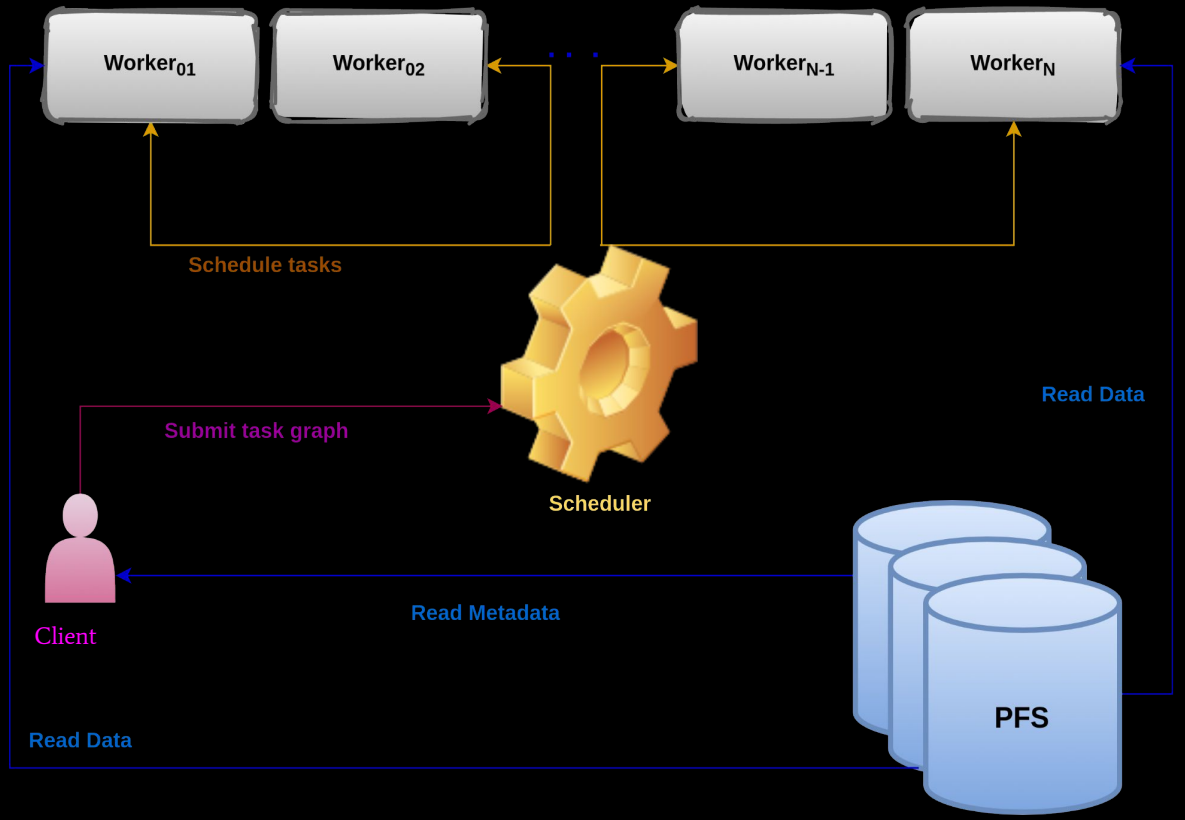
Amal Gueroudji

Outline

- Terminology and Internals
  - Client
  - Server
  - Task & Task-graph
  - Internal Classes
  - Scheduling Algorithms
- Low Level API
  - Delayed Decorator
  - Futures
- Data Model and High Level APIs
  - Dask-array & blocked Algorithms
- Actors (Class running in a worker)
- Elasticity in Dask (Adding new workers)

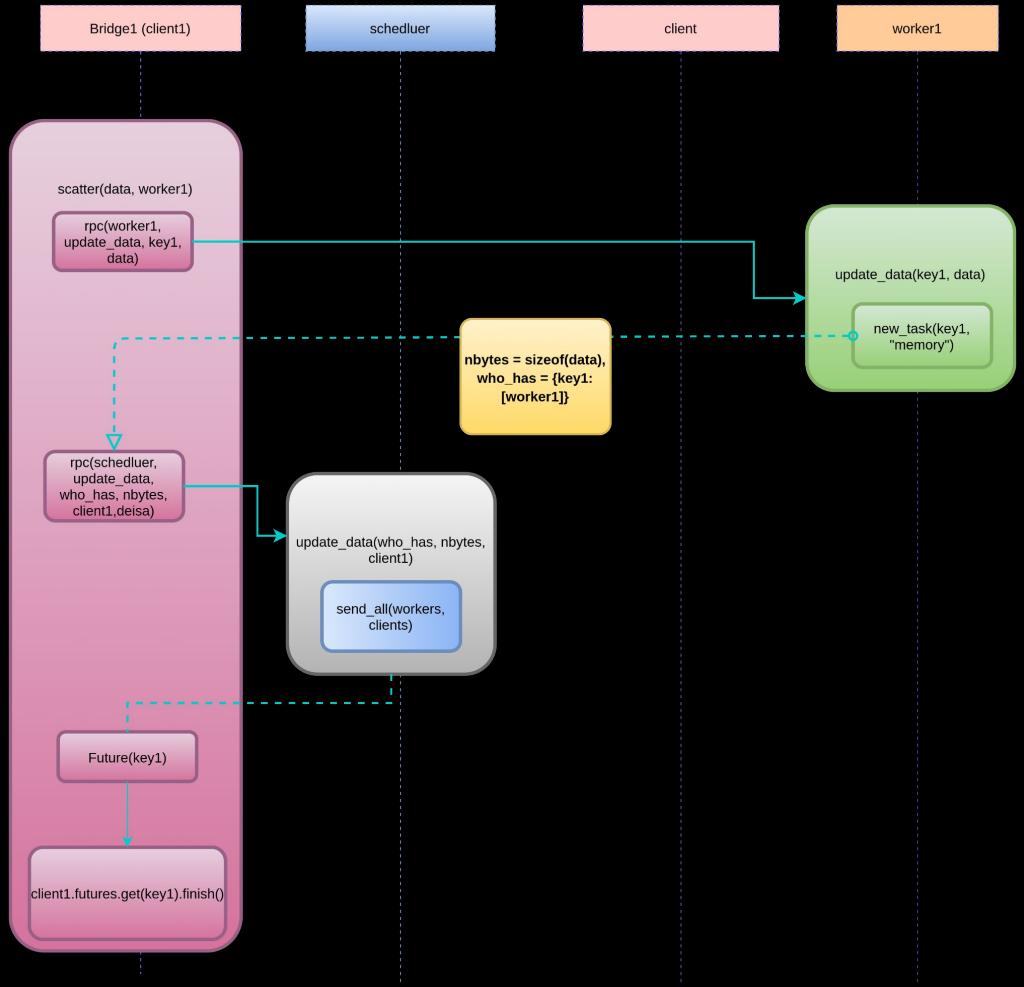
High Level Architecture

- Client, Scheduler, Worker
- Internally : Client and Server classes
- The Scheduler and the worker are Servers
- They added a Worker-Client functionality where a Worker can instentiate a client and submit tasks



Communications

- Every thing goes through RPCs
- RPC calls are hidden from the user point of view
- They generated are generated/ called by higher level APIs



Client: Task-Graph Creation

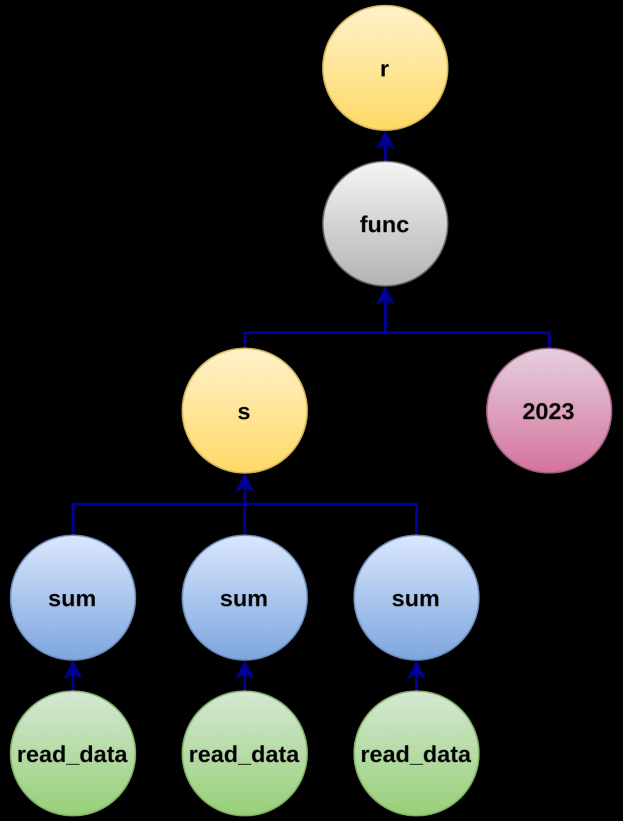
```
# Dask Distributed Analytics
 import dask.array as da
 from distributed import Client
 import h5py

 # Create a Client
 client = Client(123.456.7.89)

 # Read Dask array from dataset
 file = h5py.File('data.hdf5')
 arrays = da.from_array(file[“data”])
 s = arrays.sum()

 @dask.delayed
 def func(a, b):
     return a*b

 r = func(s, 2023)
 result = Client.compute(r).result()
```

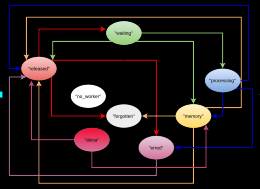


Tasks Submission Example

```
# Client Code
# …
z = client.compute(add, x, y)
```

```
# generates this RPC that is
sent to the scheduler
{'op': 'update-graph',
 'tasks': {'z': (add, x, y)},
 'keys': ['z']}
```

```
# The handler in the
scheduler is “update_graph”
scheduler.update_graph(tas
ks=msg['tasks'],
keys=msg['keys'])
```

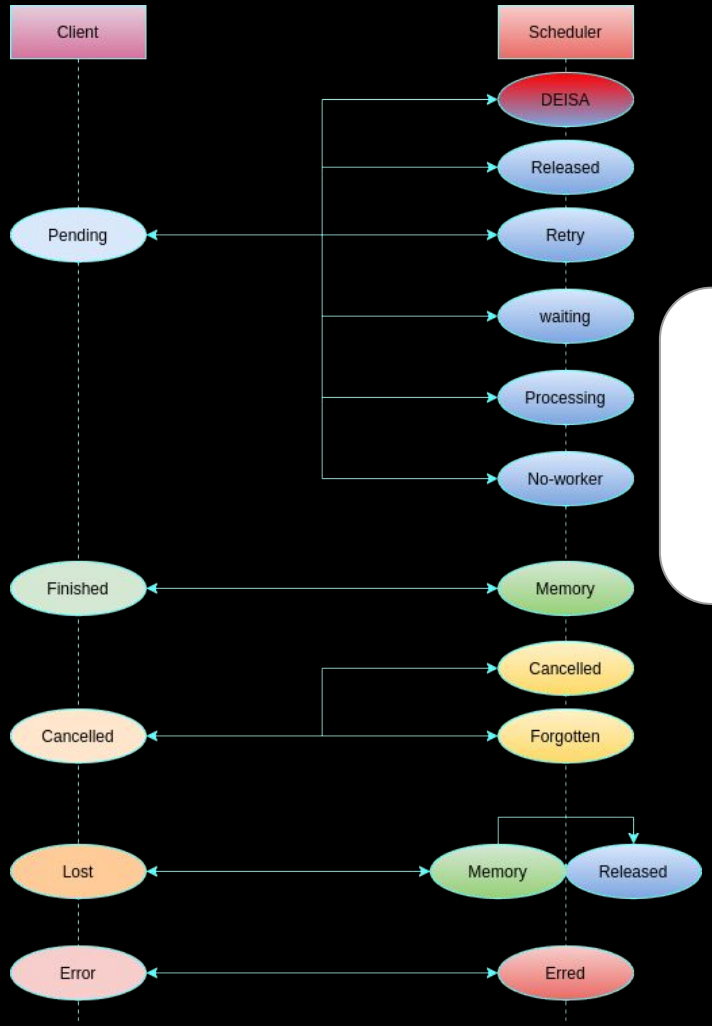


Scheduling

```
# Generates an RPC to a selected worker
{'op': 'compute',
 'function': execute_task,
 'args': ((add, 'x', 'y'),),
 'who_has': {'x': {(worker_host, port)},
             'y': {(worker_host, port), (worker_host, port)}},
 'key': 'z'}
```

Futures and Equivalent Tasks States

- When a function is submitted a future is created in the client side.
- The function is described as a task in the scheduler side.
- Both of them has there states



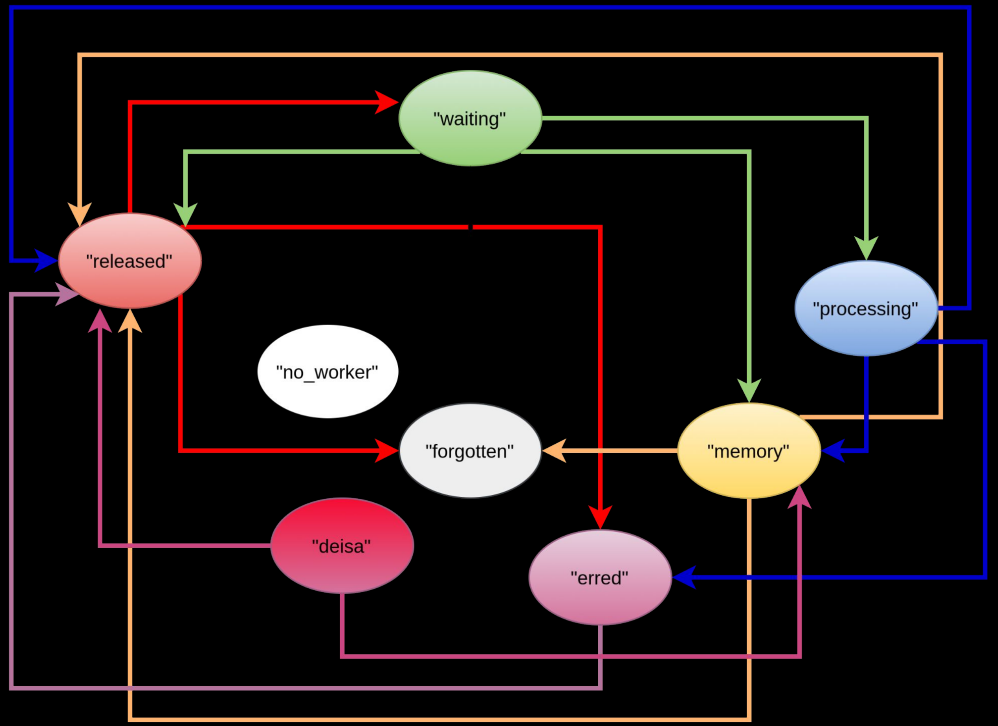
The workers have almost the same task states as the schduler

Task States Transitions in the Scheduler

- Transitions are triggered by stimuli coming from workers or clients

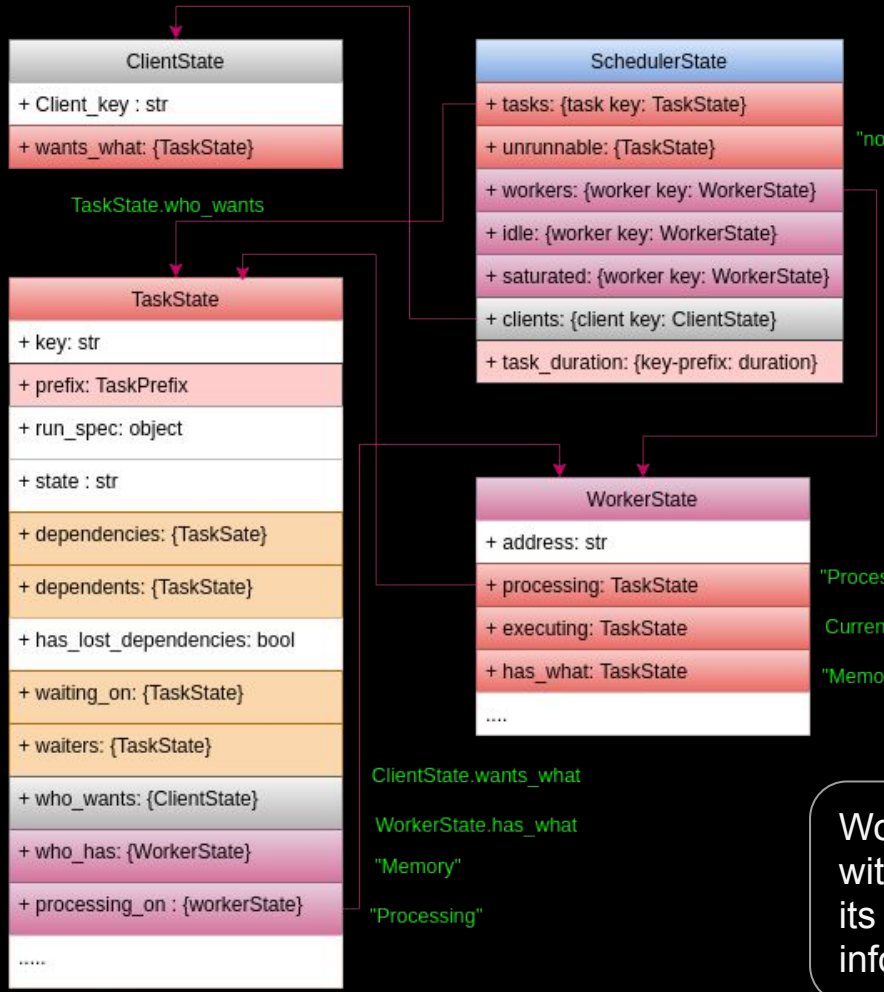
- Each transition has a corresponding “handler” in the scheduler for expl: (processing_to_memory)

- Each transition method updates the internal state classes



Scheduler Internal Classes

TaskState keeps trace of the task's current state



SchedulerState: keeps information about all what happens in the cluster.

WorkerState is associated with a worker and keeps all its current internal state information

Other Concepts: Actors

- Tasks are stateless so we define an actor as a statefull entity in Dask.
- An actor is a pointer to a user-defined-object living on a remote worker.
- Anyone with that actor can call method on that remote object.

Other Concepts: Pure Data Tasks

- Pure data sent to the Cluster is considered as a task (or its output)
- A way to push data from the client memory to the workers memory without going through the scheduler.

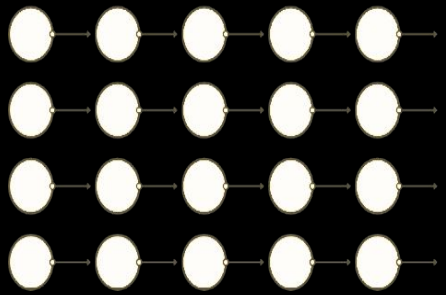
Takeaway Further Ideas

- Support for higher level annotations (Task, Service ...) instead of keeping it low level RPCs.
- Support of SSG-equivalent functionality for tasks or services.
- Go further in dynamicity to support dynamique scheduling of services in available resources.
- Consider a Higher lever class as a service provider, that can support and manage adding removing mini-services and scheduling them in resources.
- Provide an API to submit a pipeline (similar to a graph of tasks) at once.
- All the state classes can provide rich logs about the scheduling (3rd project).
- Thinking about DSL on top of Mochi.

Takeaway: Task Scheduling

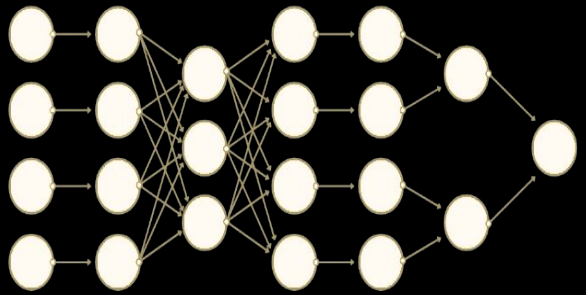
Embarrassingly Parallel

Hadoop/Spark/Dask/Airflow/Prefect



MapReduce

Hadoop/Spark/Dask



Full Task Scheduling

Dask/Airflow/Prefect