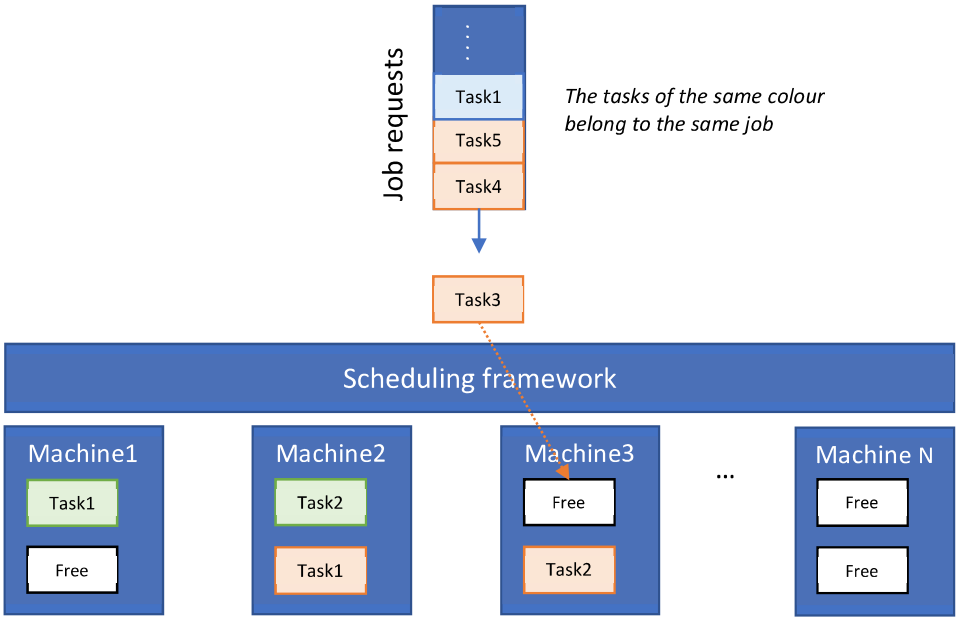
YACS: Yet Another Centralized Scheduler

Overview

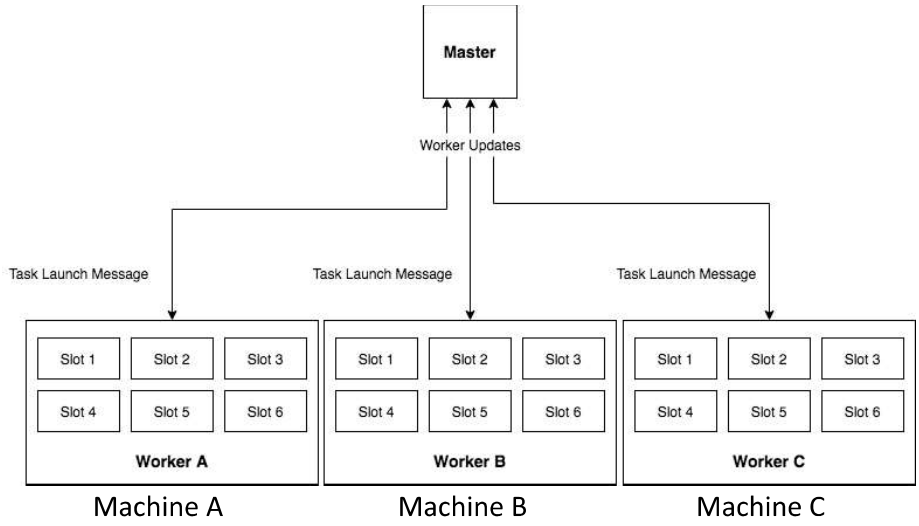
Big data workloads consist of multiple jobs from different applications. These workloads are too large to run on a single machine. Therefore, they are run on clusters of interconnected machines. A scheduling framework is used to manage and allocate the resources of the cluster (CPUs, memory, disk, network bandwidth, etc.) to the different jobs in the workload.



A job is made of one or more *tasks*. In the figure above, tasks of the same colour belong to the same job. The scheduling framework receives job requests and launches the tasks in the jobs on machines in the cluster. “Launching” a task means the scheduling framework allocates resources to the task on a specific machine. To do so, the scheduler must find a machine with available resources. In the figure, Task 3 of the orange job is assigned to the free resources on Machine 3. Once the task is allocated resources on a machine, it executes its code. As and when a task finishes execution, the scheduling framework is informed, and the resources are freed. The framework can thereafter assign these freed resources to other tasks.

YACS:

The final project is focused on YACS, a **centralized scheduling framework**. The framework consists of **one Master,** which runs on a dedicated machine and manages the resources of the rest of the machines in the cluster. The other machines in the cluster have **one Worker process** running on each of them **.** The Master process makes scheduling decisions while the Worker processes execute the tasks and inform the Master when a task completes its execution.



The Master listens for job requests and dispatches the tasks in the jobs to machines based on a *scheduling algorithm*. Each machine is partitioned into equal-sized resource encapsulations (E.g. 1 CPU core, 4GB of memory, 1TB of disk space) called **slots**. The number of slots in each machine is fixed. Each slot has enough resources to execute one task at a time. However, since the machines may have different capacities (in terms of amount of total memory, total number of cores, etc.), the number of slots may differ from machine to machine. Therefore, at any point of time, the maximum number of tasks a specific machine can run is equal to the number of slots in the machine. The Master is informed of the number of machines and the number of slots in each machine with the help of a config file.

The Worker processes listen for Task Launch messages from the Master. On receiving a Launch message, the Worker adds the task to the execution pool of the machine it runs on. The execution pool consists of all currently running tasks in the machine.

Simulating the working of the framework:

You will be simulating the functioning of YACS. In the simulated framework that you create, you will have **1 Master process and 3 Worker processes**. You will be running all the processes run on the same PC, however, they must behave as though they are on separate dedicated machines. That is, your *simulated cluster* will consist of a machine running the Master and 3 other machines, each running a Worker process. These simulated machines have nothing to do with the PC on which you will be running the processes. The number of slots in each simulated machine is as per the config passed to the Master process.

The Master process must execute as though it manages a cluster with real machines and real slots. However, in your implementation, *slots* are only an abstraction. For instance, the value of available slots for a simulated machine is only a number. This number is decremented when a slot is said to have been allocated to a task, and incremented when a slot is said to have been freed on a task's completion.

The Worker process listens for task launch message from the Master. When it receives a task launch message, it adds the task to its execution pool. The Worker process must simulate the running of the tasks in the execution pool. It does so by decrementing the *remaining_duration* value of each task, every second, until it reaches 0. Once the *remaining_duration* of a task reaches 0, the Worker removes the task from its execution pool and reports to the Master that the task has completed its execution.

The framework will have to respect the map-reduce dependency in the jobs. Therefore, when a map task completes execution, the Master will have to check if it satisfies the dependencies of any reduce tasks and whether the reduce tasks can now be launched. A job is said to have completed execution only when all the tasks in the job have finished executing. Refer to the flowcharts below for more clarity.

All jobs have 2 stages only. The first stage consists of map tasks and the second stage consists of reduce tasks. The reduce tasks in a job can only execute after all the map tasks in the job have finished executing. There is no ordering within reduce tasks, or within map tasks. All map tasks can run in parallel, and all reduce tasks can run in parallel.

The three scheduling algorithms you will be using to pick a machine for a task are: *random, round-robin* and *least-loaded.*

You will be using sockets for all communications between the Master and Workers. As mentioned earlier, each Worker's machine will be configured with a fixed number of slots. A single slot can execute only one task at a time. When a task finishes executing, the Worker has to communicate this event to the Master. The Master needs to keep track of the number of slots available in each machine.

The Master and the Workers need to maintain a log of important events.

Part 1: Setting up the Scheduling Framework and running the simulation:

1. Implement the Master and the Workers as two separate python programs. One program for the Master, and one for the Worker.
2. The Master will be listening for incoming jobs on port **5000**. The path to the config file, and the scheduling algorithm (*RANDOM, RR, LL*) are given to the Master as command line arguments. E.g. `python Master.py /path/to/config RR`
3. The Master will listen for updates from Workers on port **5001**.
4. Each Worker will be listening for messages from the Master. The port and worker_id are supplied as command line arguments to each Worker program. These settings need to be made as per the config file supplied to the Master.
   E.g. `python Worker.py 4000 1`
5. Run the `requests.py` file to generate job requests with exponentially distributed inter-arrival times and send them to the Master. The program takes as command line argument the number of requests to be sent to the Master. E.g. `python requests.py 10`
6. The program requires the following python modules to be installed:
   - json
   - socket
   - time
   - sys
   - random
   - numpy

Scheduling Algorithms:

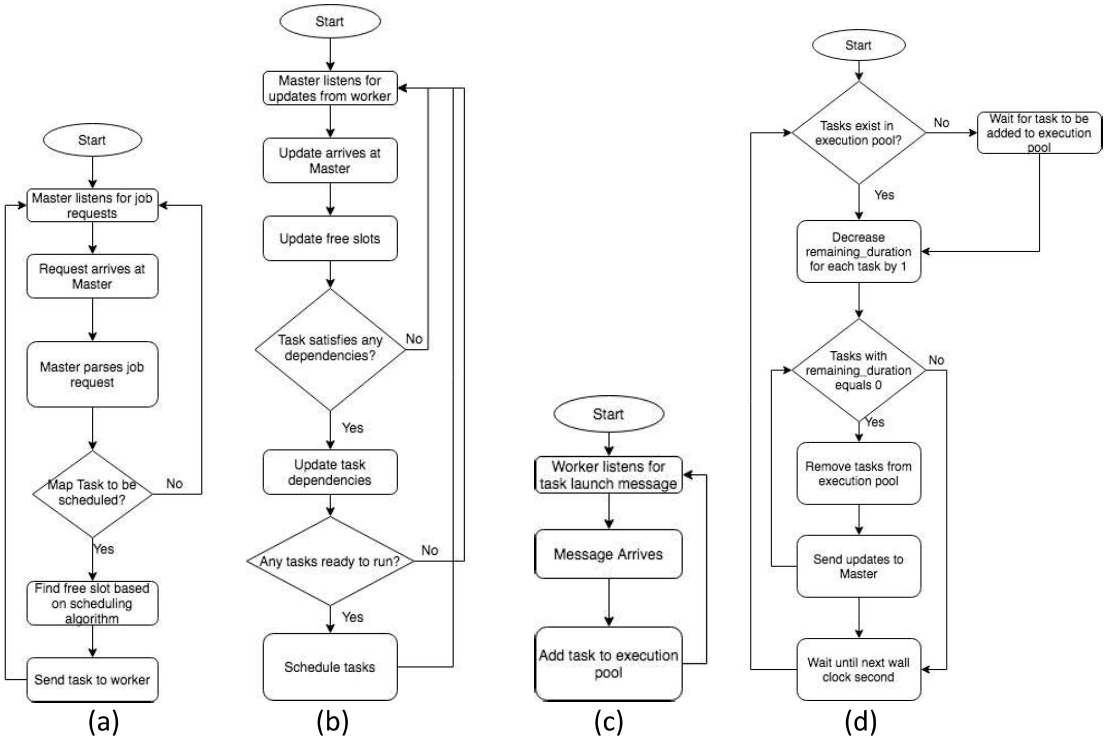
1. **Random:** The Master chooses a machine at random. It then checks if the machine has free slots available. If yes, it launches the task on the machine. Else, it chooses another machine at random. This process continues until a free slot is found.
2. **Round-Robin:** The machines are ordered based on worker_id of the Worker running on the machine. The Master picks a machine in round-robin fashion. If the machine does not have a free slot, the Master moves on to the next worker_id in the ordering. This process continues until a free slot is found.
3. **Least-Loaded:** The Master looks at the state of all the machines and checks which machine has most number of free slots. It then launches the task on that machine. If none of the machines have free slots available, the Master waits for 1 second and repeats the process. This process continues until a free slot is found.

Workflow

- The Master will need to have at least 2 threads- (a) to listen for job requests, and (b) to listen for updates from Workers.

- Each Worker will need to have at least 2 threads- (c) for listening for task launch messages from Master, (d) to simulate the execution of the tasks and to send updates to the Master.

The flowcharts below chalk out the workflow of the 4 threads.



To prevent race conditions, make sure to use locks for data structures that are shared between the threads in a component. *Even though you are running the Master and Worker processes on the same machine, do not make any assumptions which prevent the Master or any of the Workers from being deployed on separate machines*. Ensure that all data sharing between the various components takes this aspect into account. Do not rely on methods such as shared files or global variables for communicating between the Workers and Master.

Format of Job Request Message (JSON):

```
{
        "job_id":<job_id>,
        "map_tasks":[
                {
                        "task_id":"<task_id>",
                        "duration":<in seconds>
                },
                {
```

```
                "task_id":"<task_id>",
                "duration":<in seconds>
        }
        ...
    ],
    "reduce_tasks":[
        {
                "task_id":"<task_id>",
                "duration":<in seconds>
        },
        {
                "task_id":"<task_id>",
                "duration":<in seconds>
        }
        ...
    ]
}
```

Format of config file for Master (JSON):

```
{
 "Workers": [ //one worker per machine
   {
    "worker_id": <worker_id>,
    "slots": <number of slots>, // number of slots in the machine
    "port": <port number> // on which the Worker process listens for task launch message
   },
   {
    "Worker_id": <Worker_id>,
    "slots": <number of slots>,
    "port": <port number>
   },
   …
]
}
```

Part 2: Results to be analyzed from logs:

Write a program that does the following:

1. Calculates the mean and median task and job completion times for the 3 scheduling algorithms.
    a. task completion time:
    (end time of task in Worker process) – (arrival time of task at Worker process)
    b. job completion time:
    (*end time of the last reduce task) – (arrival time of job at Master)*

2. Plots the number of tasks scheduled on each machine, against time, for each scheduling algorithm.