CS322:Big Data

Final Class Project Report

Project (FPL Analytics / YACS coding): YACS Coding
Date: 01 – 12 - 2020

Sl. No	Name	SRN	Class/Section
1	Ameya Bhamare	PES1201800351	C
2	Atmik Ajoy	PES1201800189	A
3	Chethan Mahindrakar	PES1201801126	A
4	Dhanya Gowrish	PES1201800965	A

Introduction

As we all know, Big Data refers to huge volumes of data from various sources. The workloads are often too huge to run on a single machine with limited computing power. With Big Data frameworks, we are able to run these jobs on clusters of machines that are interconnected. We need scheduling frameworks to manage these clusters. An analogy would be the director of an organization managing all employees and resources.

Keeping that in mind, our final project requires us to build a centralized scheduling framework, called **YACS** (**Yet Another Centralized Scheduler**).

It consists of a **Master**, which is a node that essentially runs on a dedicated machine and manages the resources of the rest of the nodes in the cluster.

The other nodes in the cluster have a **Worker** process running on each one of them. The Master process makes scheduling decisions just like a director makes decisions in an organization while the Worker processes execute the tasks and inform the Master when a task completes its execution just like employees.

Related work

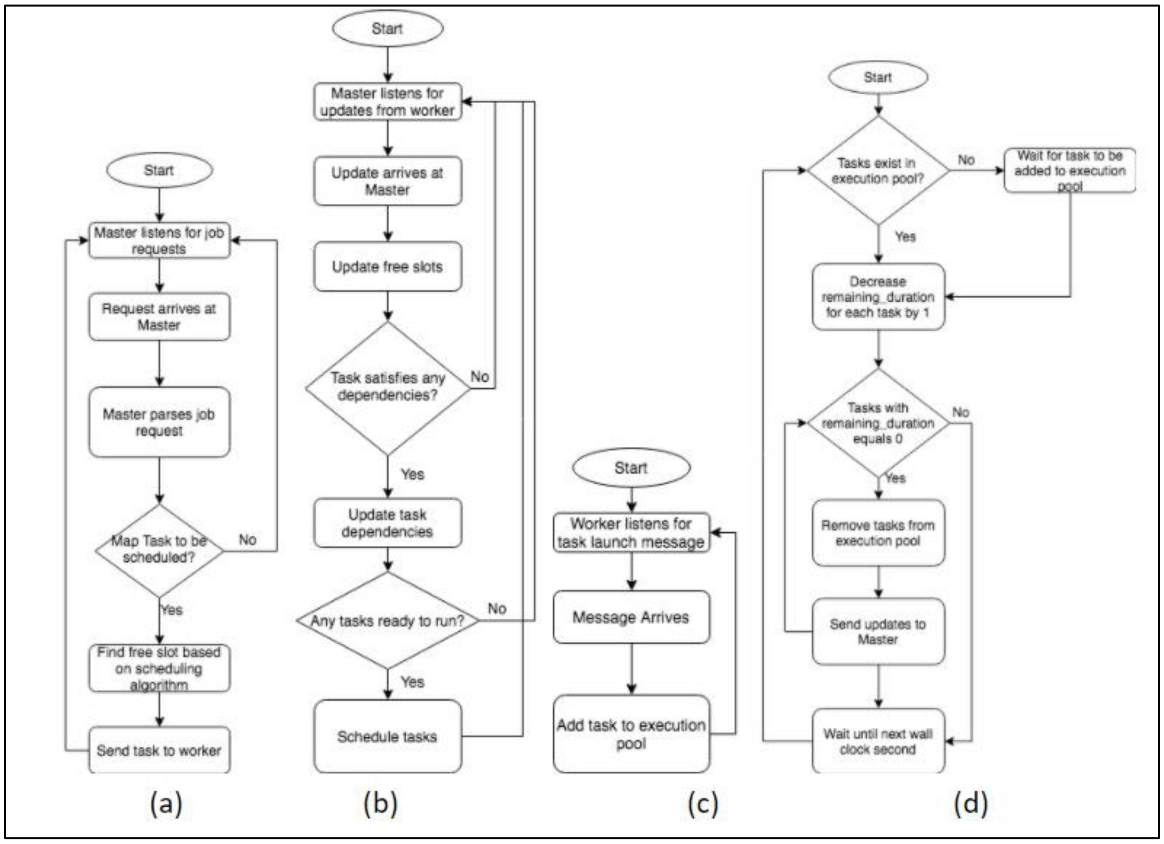
We referred to tutorials for multithreading, resource scheduling and socket programming.

Here are few of the links we referred to:

- https://www.edureka.co/blog/what-is-mutithreading/
- https://docs.python.org/3/howto/sockets.html
- https://www.youtube.com/watch?v=3QiPPX-KeSc
- https://towardsdatascience.com/schedulers-in-yarn-concepts-to-configurations-5dd7ced6c214#:~:text=There%20are%20three%20types%20of,placing%20them%20in%20a%20queue

Design

Since detailed implementation instructions were provided to us, we referred to the flowcharts given below.


Start
Master listens for job requests
Request arrives at Master
Master parses job request
Map Task to be scheduled?
No
Yes
Find free slot based on scheduling algorithm
Send task to worker
(a)
Start
Master listens for updates from worker
Update arrives at Master
Update free slots
Task satisfies any dependencies?
No
Yes
Update task dependencies
Any tasks ready to run?
No
Yes
Schedule tasks
(b)
Start
Worker listens for task launch message
Message Arrives
Add task to execution pool
(c)
Start
Tasks exist in execution pool?
No
Wait for task to be added to execution pool
Yes
Decrease remaining_duration for each task by 1
Tasks with remaining_duration equals 0
No
Yes
Remove tasks from execution pool
Send updates to Master
Wait until next wall clock second
(d)

The master is constantly listening for job requests on **Port 5000** i.e. **master.py**. The master can be started with any of the 3 scheduling algorithms specified in the specifications i.e. **Round-Robin**, **Random** or **Least-Loaded**.

As the job requests are received, the Master must schedule the tasks of the jobs on the Workers based on scheduling algorithm chosen.

In addition to taking care of the task assignment based on availably of slots in the workers, we also had to ensure that only once all the map tasks of a job were scheduled and completed, the reduce tasks could be scheduled.

The Master on another **Port 5001** listens to the workers which send updates as when tasks are finished.

We used locks to ensure that our shared data structures, for example the number of available slots in a worker could only be accessed and modified by one thread at a time.

We also maintained log files and wrote to them based on requirements, for example, we wrote to the log file every time the Master sent a task to a Worker.

We used a lock on the log file as well to ensure only one thread could write to the log file at a time.

The **worker.py** works with 3 ports identified by:

> (**port number**, **port id**) **i.e.** **(4000, 1)**, **(4001, 2)**, **(4002, 3)**.

The worker listens for task assignments from the master. It also sends updates to the Master as and when it completes execution of tasks in its slots. When the specified duration of a task becomes zero, it updates the master indicating that it has completed a task.

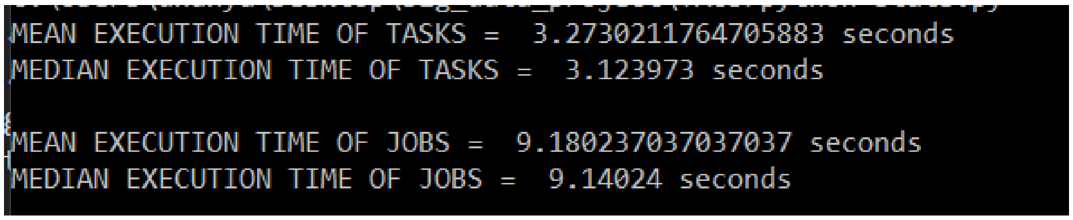
We maintained locks on the number of free slots and the log files in the worker as well.

To be specific, we had one log file for the Master and one log file for each of the workers.

Results

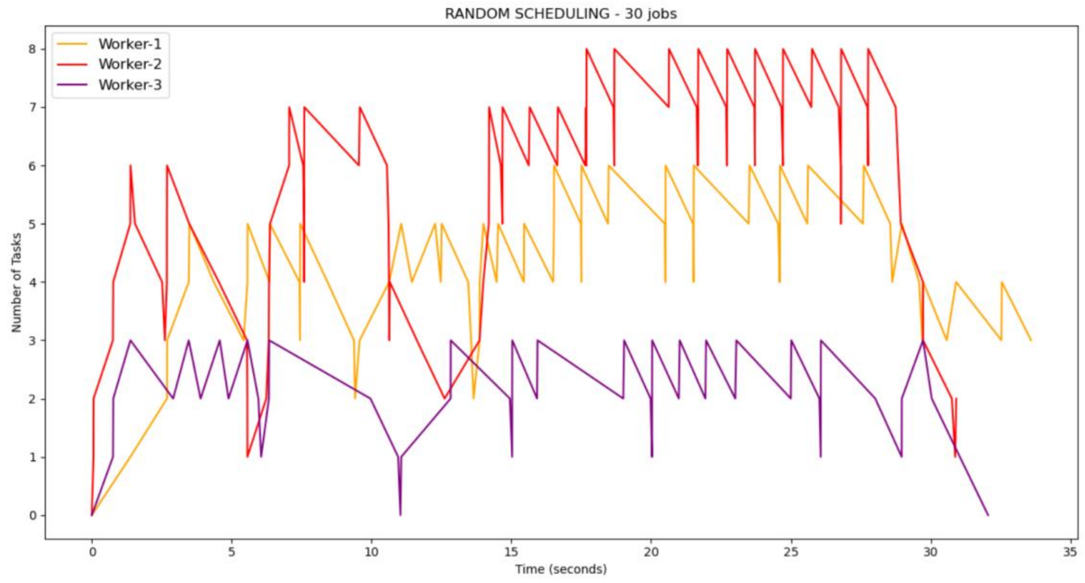
We have attached a few example screenshots for demonstration purposes i.e. Random and Round-Robin.

RANDOM SCHEDULER

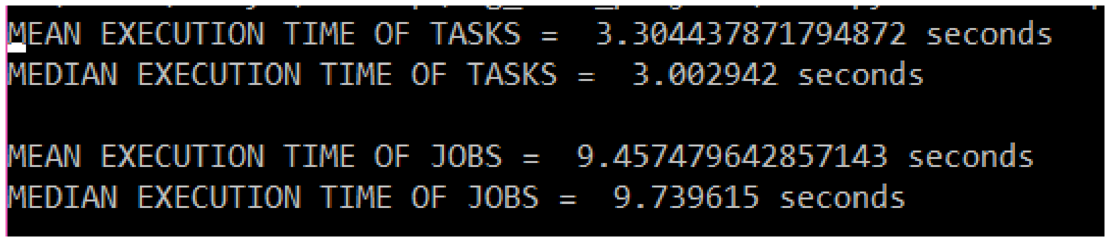


```
MEAN EXECUTION TIME OF TASKS =  3.2730211764705883 seconds
MEDIAN EXECUTION TIME OF TASKS =  3.123973 seconds

MEAN EXECUTION TIME OF JOBS =  9.180237037037037 seconds
MEDIAN EXECUTION TIME OF JOBS =  9.14024 seconds
```

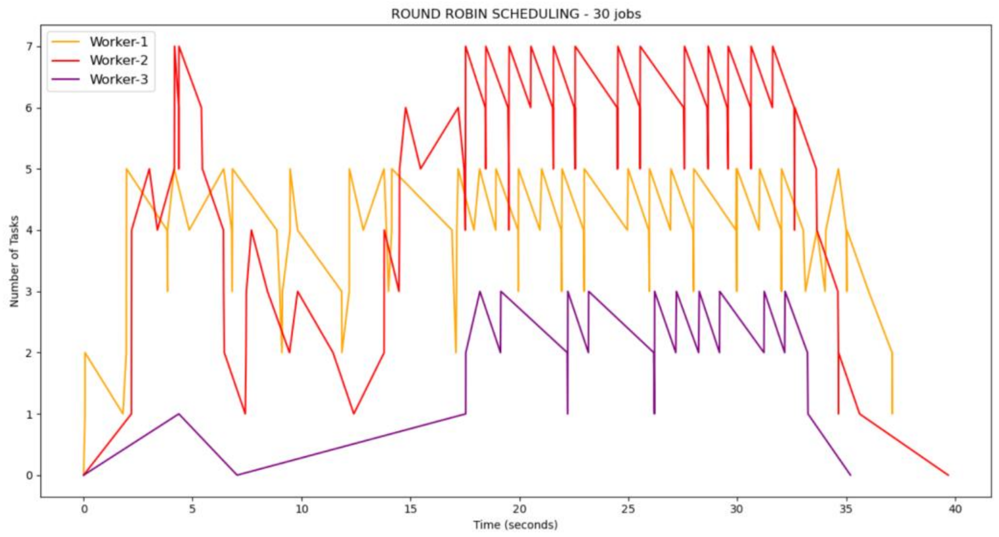


ROUND-ROBIN SCHEDULER



```
MEAN EXECUTION TIME OF TASKS =  3.304437871794872 seconds
MEDIAN EXECUTION TIME OF TASKS =  3.002942 seconds

MEAN EXECUTION TIME OF JOBS =  9.457479642857143 seconds
MEDIAN EXECUTION TIME OF JOBS =  9.739615 seconds
```



Problems

- Setting up socket communication was extremely time consuming. Turns out there were too many conflicting requests over a single port.
- We were encountering race conditions at first. Usage of locks in the right places helped us take care of it.

Conclusion

We understood how the YARN scheduler works, using it is very simple but implementing a basic scheduler requires a lot of intricacies to be taken care of.

EVALUATIONS:

SNo	Name	SRN	Contribution (Individual)
1			
2			
3			
4			

(Leave this for the faculty)

Date	Evaluator	Comments	Score

CHECKLIST:

SNo	Item	Status
1.	Source code documented	YES
2.	Source code uploaded to GitHub – (access link for the same, to be added in status →)	
3.	Instructions for building and running the code. Your code must be usable out of the box.	