CS322:Big Data

Final Class Project Report

Project (FPL Analytics / YACS coding): **<u>YACS_Coding</u>** **Date:** **<u>30-11-2020</u>**

SNo	Name	SRN	Class/Section
1	Taran Singhania	PES1201800333	5I
2	Pranav Bhatt	PES1201800764	5G
3	Rehan Vipin	PES1201801655	5J
4	Romaanchan Skanda	PES1201801958	5J

Introduction

The project is mainly an attempt to mimic the working of YARN(Yet another Resource Negotiator). YARN is a framework which is present in the hadoop ecosystem. The main functionality of YARN is to keep a track of the available resources and the creation of an application master to keep a track of the jobs. We mimic the same by the use of threads and sockets. The requests are sent to the master by a client. The master adds the tasks into separate queues (wait, ready and running). Now the master schedules the jobs(tasks) into different workers and keeps listening for completion of the tasks. Once completed, notifies the client about completion.

Related work

We referenced the python docs for documentation on:

- The socket library for finding out how to bind a port to a socket and how to listen and send data through them
- The threading library for implementing multiple threads to perform the specified tasks
- The locking mechanism provided by the threading library to prevent race conditions. Here we also referred to medium articles (https://medium.com/python-features/using-locks-to-prevent-data-races-in-threads-in-python-b03dfcfbadd6) for the working

Design

Master

For executing the code for the master, the configuration file along with the scheduling algorithm is passed in as input. The configuration file is a json file which contains the number of slots available and its corresponding port number for every worker. We have 5 separate threads in the master which are used for the following

· Debugging by analyzing states of execution queues

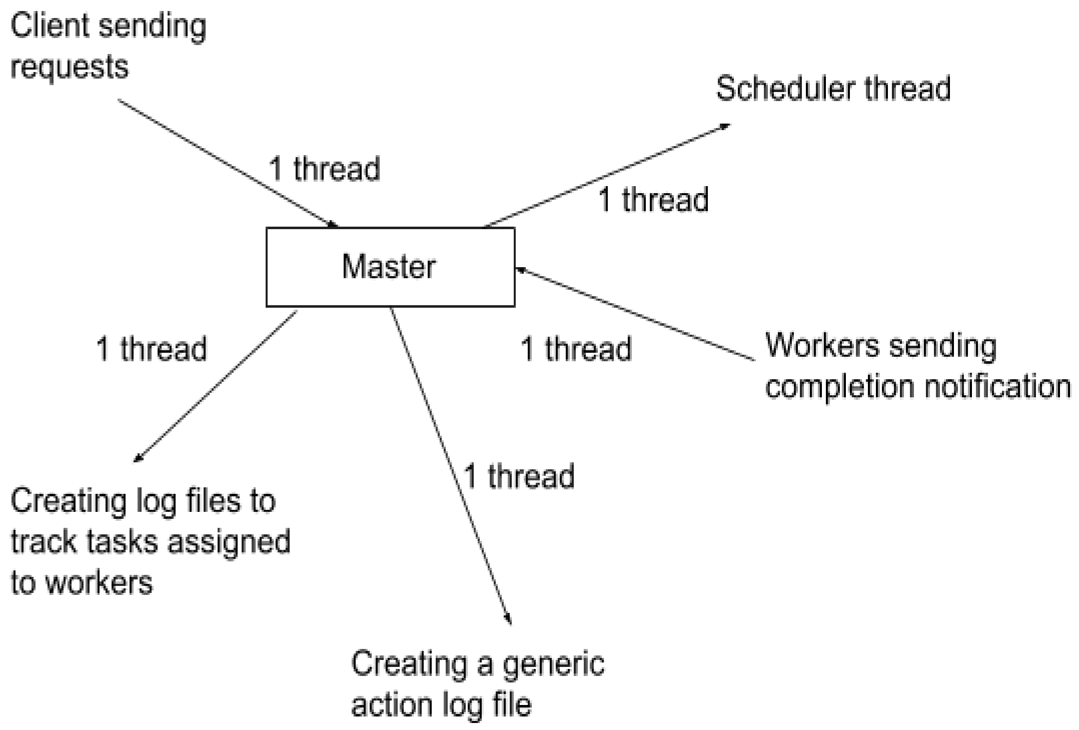
· Collecting logs about load on workers for analysis later on

· Accepting requests for jobs and placing tasks in queues

· Dispatching jobs to the workers

· Scheduling the jobs(the scheduler and dispatcher are separate)

The job requests are sent to the master through a socket at port 5000. The master receives task updates from workers on port 5001.Every job is split up into 2 tasks, the mapper and the reducer. All the tasks are split into wait_queue, ready_queue and running queue.The workers are made as worker objects containing the info about their configuration such as worker id, port on which they are functioning and the number of slots they have.


Client sending requests
1 thread
Scheduler thread
1 thread
Master
1 thread
1 thread
Workers sending completion notification
1 thread
Creating log files to track tasks assigned to workers
Creating a generic action log file

All the incoming jobs are referenced by their job ids. All of the queues are equipped with the locking mechanism in order to maintain synchronisation. The log files are created in a separate directory. The requests are received from the client as a json object through the job incoming port. Each job has a job_id, the arrival time for each job is saved and the number of mappers and reducers it has. The set of map and reduce tasks are first extracted. If there are no map tasks then all the reduce tasks are added into the ready queue.

If there are map tasks then all the reduce tasks are added into the wait queue and the map tasks are put into the ready queue.The tasks which have been dispatched and are scheduled are stored on the running queue(transferred from ready queue to running queue). The master also keeps listening for the completion of tasks and updates the number of free slots for each worker (The info for each worker is saved on the master and is referenced by the worker id).

Once it receives the notification for the completion it removes the task from the running queue and decreases the corresponding count of map or reduce tasks. Along with this the log files are also updated with the time taken for completion of each task, which worker it was assigned to. There is also a log file created in order to keep a track of all the actions performed for debugging purposes.

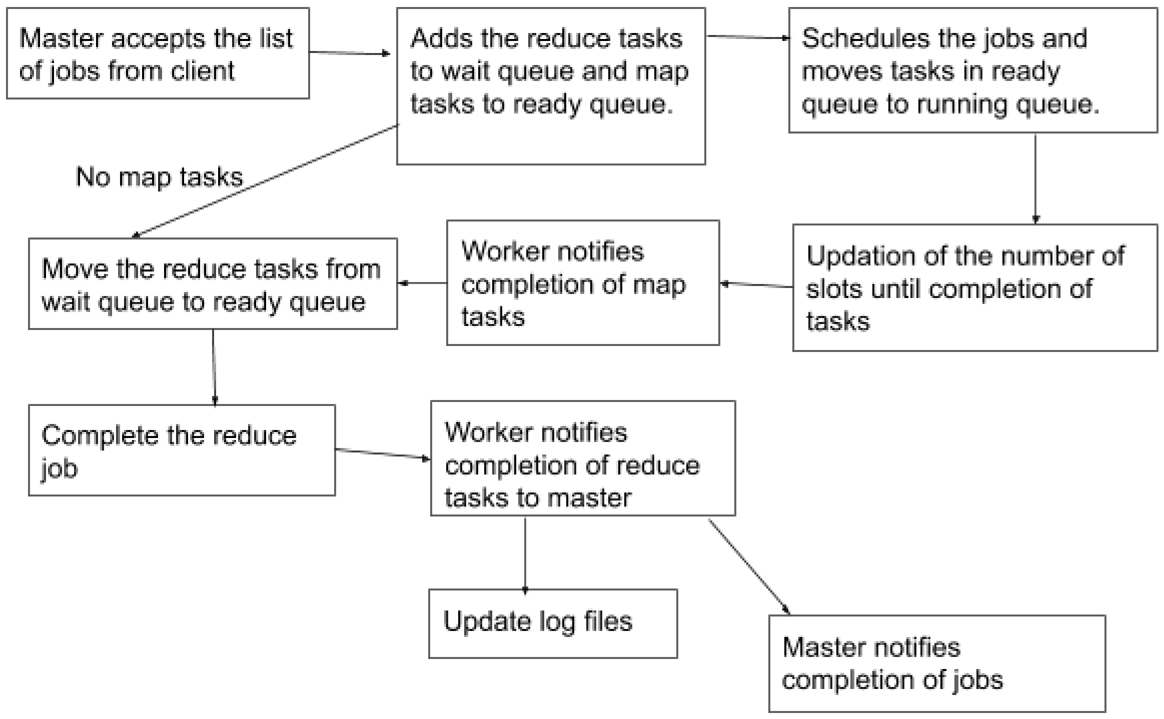
Note about logging

There are two dedicated threads for logging, the listener thread also logs task and job completions.

This third method of logging is different, while the other two log updates periodically, the listener threads logs only task and job completion times.

The logging for load on workers could also be done in the scheduler thread, the results would be consistent with expectations but it does not give a look at real-time performance of the workers.

General WorkFlow


Master accepts the list of jobs from client
Adds the reduce tasks to wait queue and map tasks to ready queue.
Schedules the jobs and moves tasks in ready queue to running queue.
No map tasks
Move the reduce tasks from wait queue to ready queue
Worker notifies completion of map tasks
Updation of the number of slots until completion of tasks
Complete the reduce job
Worker notifies completion of reduce tasks to master
Update log files
Master notifies completion of jobs

Worker

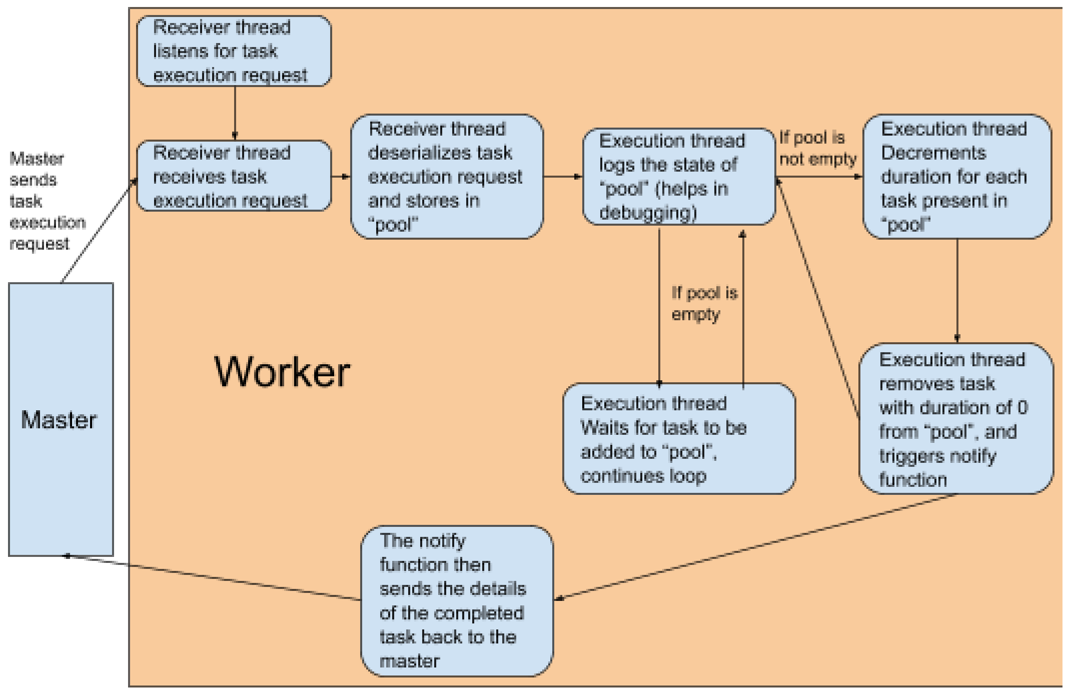
Each worker uses 2 threads for performing the necessary actions; a receiver thread meant to receive tasks execution requests, and an executor thread meant to simulate the running of a task. The task executions are of the form "identifier" (communication token with master), "task_hash" (unique identifier for tasks) and "duration" (execution time for the task).

When a task is received by the receiver, it is first deserialized, converted to a dictionary object, and stored in a list called "pool".

Every second, the executor reads the tasks present in "pool" and decrements the duration by one. It is also responsible for logging the tasks being processed on a worker every second.

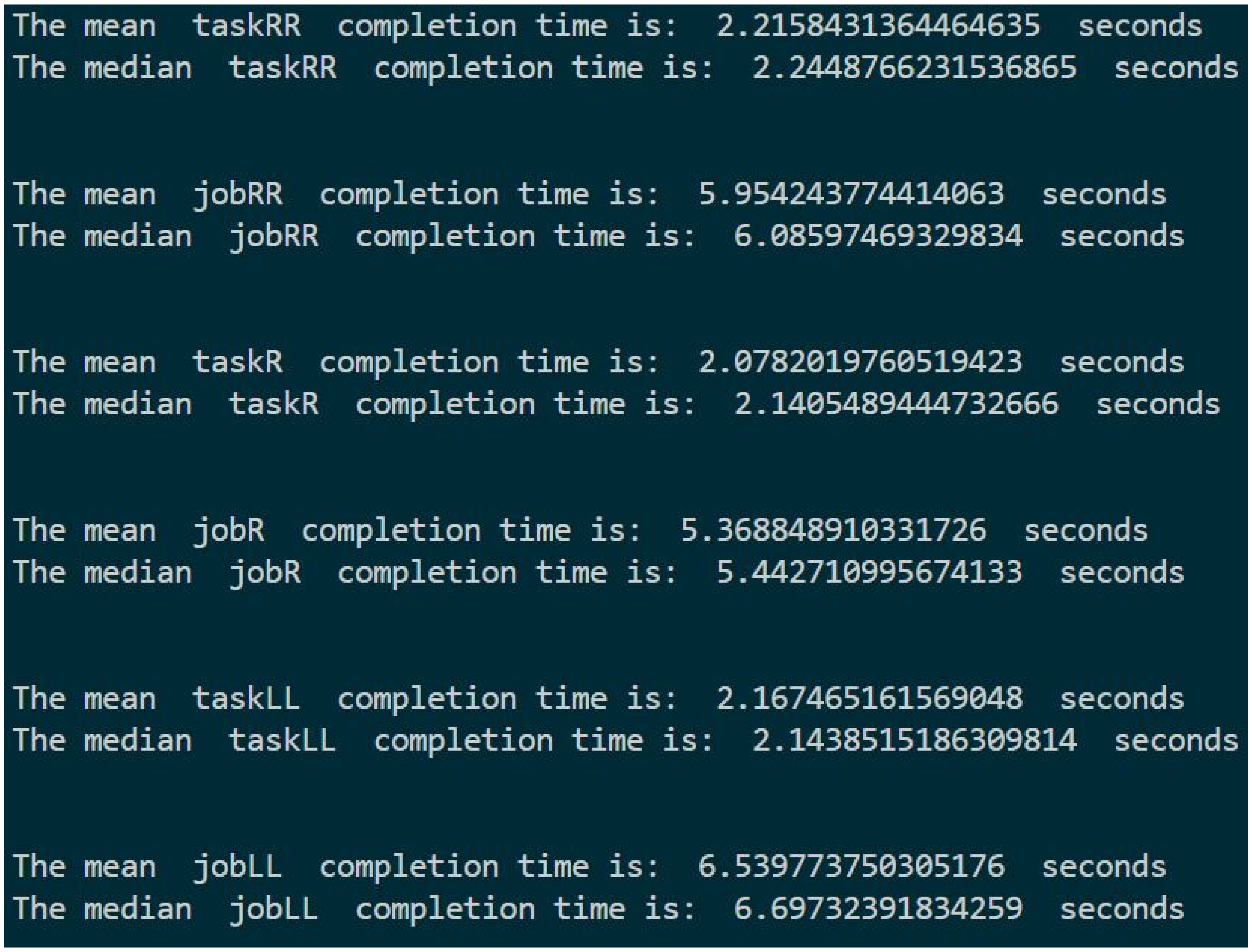
If the executor discovers a task with a duration of '0', it sends the details of the task back to the master using the notify function, which serializes the task and sends it to the appropriate port.

Worker workflow

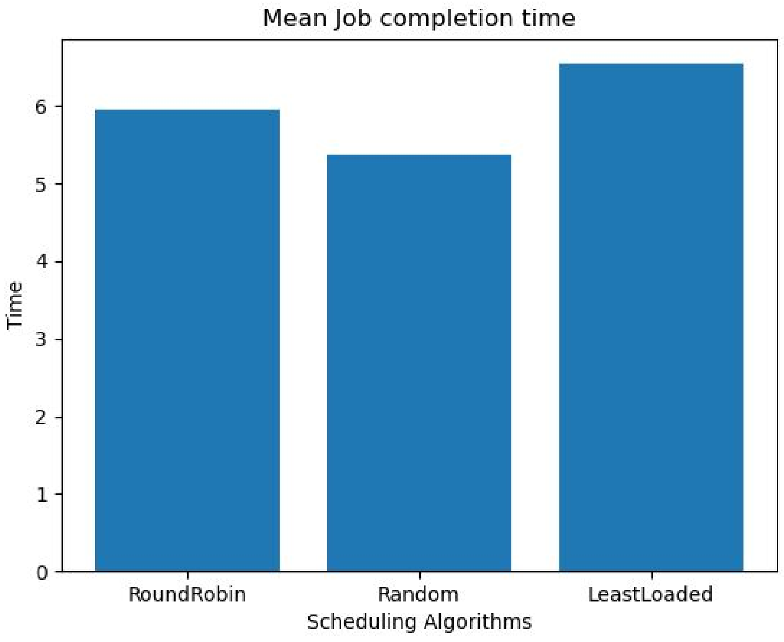


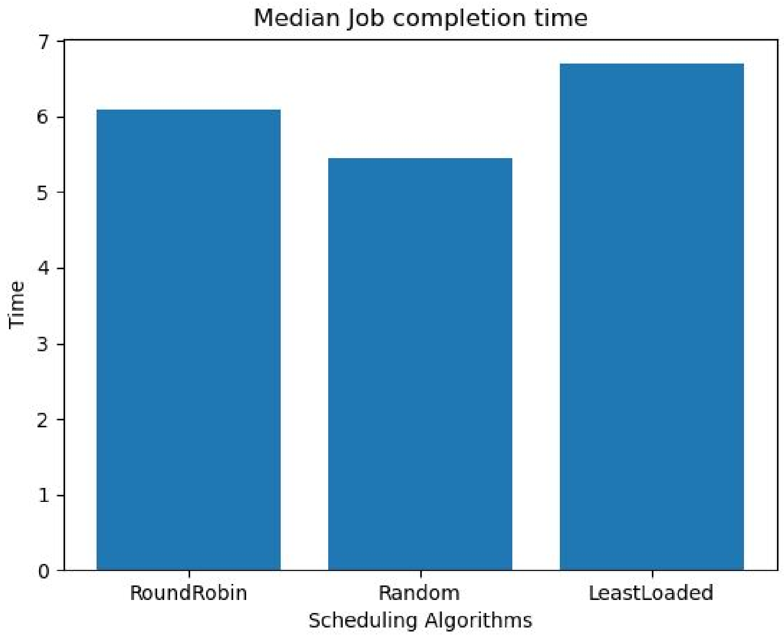
Receiver thread listens for task execution request
Master sends task execution request
Receiver thread receives task execution request
Receiver thread deserializes task execution request and stores in "pool"
Execution thread logs the state of "pool" (helps in debugging)
If pool is not empty
Execution thread Decrements duration for each task present in "pool"
If pool is empty
Worker
Master
Execution thread Waits for task to be added to "pool", continues loop
Execution thread removes task with duration of 0 from "pool", and triggers notify function
The notify function then sends the details of the completed task back to the master

Results



```
The mean  taskRR  completion time is:  2.2158431364464635  seconds
The median  taskRR  completion time is:  2.2448766231536865  seconds


The mean  jobRR  completion time is:  5.954243774414063  seconds
The median  jobRR  completion time is:  6.08597469329834  seconds


The mean  taskR  completion time is:  2.0782019760519423  seconds
The median  taskR  completion time is:  2.1405489444732666  seconds


The mean  jobR  completion time is:  5.368848910331726  seconds
The median  jobR  completion time is:  5.442710995674133  seconds


The mean  taskLL  completion time is:  2.167465161569048  seconds
The median  taskLL  completion time is:  2.1438515186309814  seconds


The mean  jobLL  completion time is:  6.539773750305176  seconds
The median  jobLL  completion time is:  6.69732391834259  seconds
```


Mean Job completion time
Time
0
1
2
3
4
5
6
RoundRobin
Random
LeastLoaded
Scheduling Algorithms


Median Job completion time
Time
0
1
2
3
4
5
6
7
RoundRobin
Random
LeastLoaded
Scheduling Algorithms

Mean and median time

From the above image we infer that for tasks,

Mean time: Round Robin > Least Loaded > Random

Median time: Round Robin > Least Loaded > Random

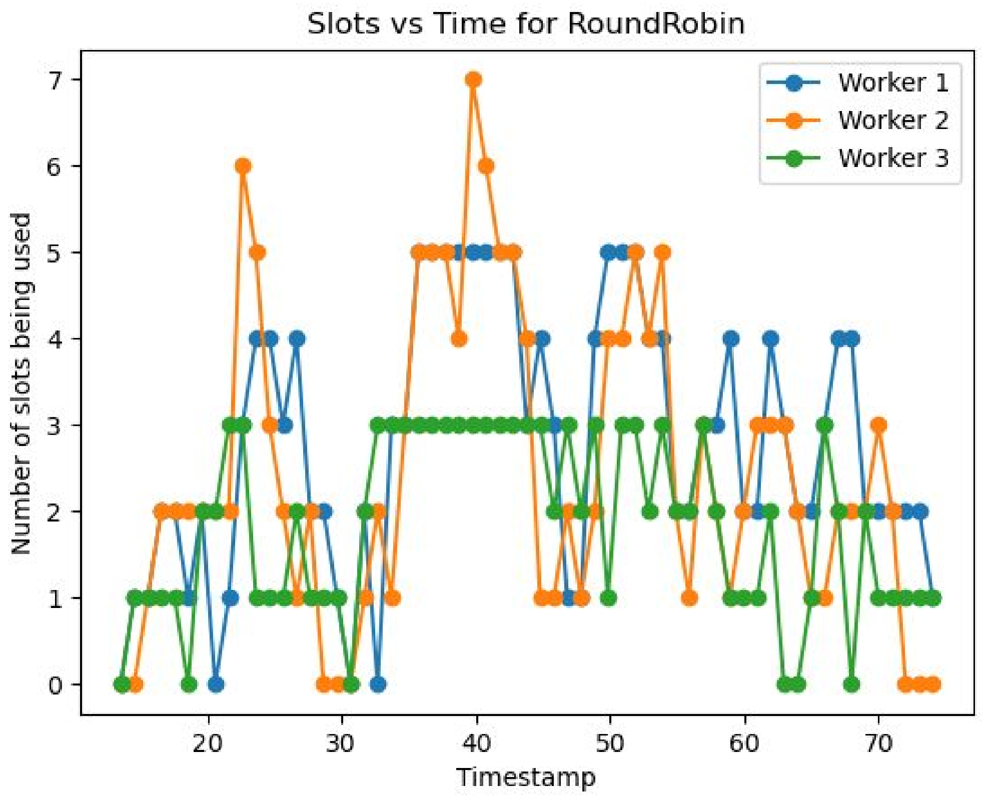
For jobs:

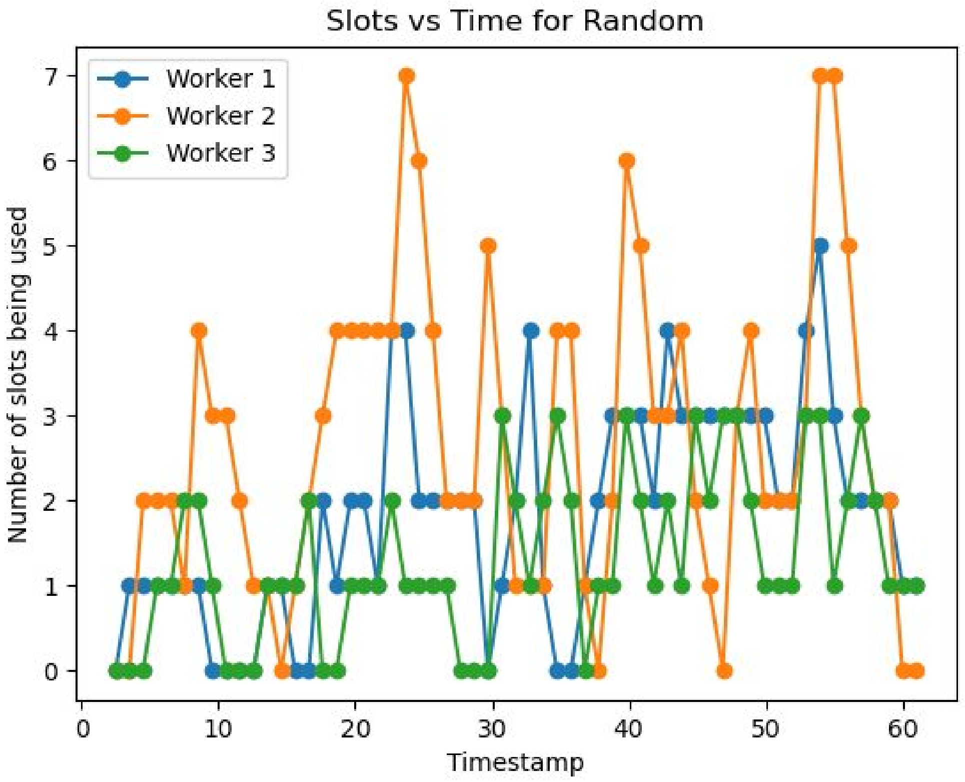
Mean time: Least Loaded > Round Robin > Random

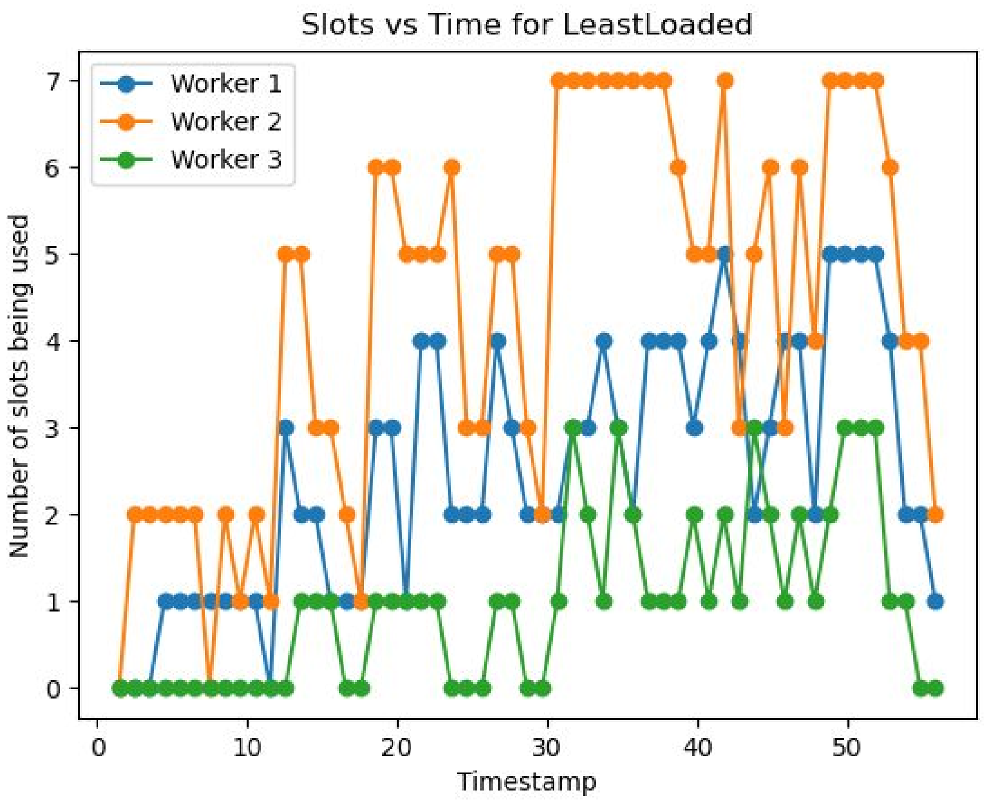
Median time: Least Loaded > Round Robin > Random

Graphs:

These slots vs timestamp graphs were obtained by running the requests.py with 50 job requests for each scheduling algorithm. The requests are all random and so are the durations they are sent. To get a fair evaluation between schedulers we set the random seed so the requests and durations are the same for all.




Slots vs Time for Random
Worker 1
Worker 2
Worker 3
Number of slots being used
Timestamp
0
1
2
3
4
5
6
7
0
10
20
30
40
50
60



From the images above we can conclude that scheduling algorithms are working correctly. For example, in Least Loaded, worker 2 with the most number of slots, has the highest area under curve and is most utilized. Similarly, in Round Robin, slots are being allocated one by one as seen in the graph.

At task level: Random performs best, Round Robin performs worst

At job level: Random performs best, Least Loaded performs worst

Inferences regarding the results about the performance of the scheduling algorithms -

1. Least Loaded finishes first. This is expected because it is shown to be operating at max efficiency most of the time. But even then it's job completion time on average is the worst, we think this is because of the wait it performs when workers are full.
2. Round Robin finishes last, this too is expected because of worker slots being underutilized. The area under the curve is smaller than the other two.

Problems

- Selective waiting while scheduling - We passed locks to all schedulers, only the least loaded scheduler was allowed to release and acquire them while waiting for slots to be freed up. All others exited if a worker was not found.
- Figuring out how to maintain map-reduce order - Keeping track of number of map tasks and reduce tasks. (Refer general workflow diagram)
- Infinite waiting while using the least loaded scheduler. We found out that we needed more fine grained locking and needed to release it while it was sleeping.

Assumptions

- The request size is 64KB or less, but this can be increased or decreased.
- The format of the request made from the requests.py client.
- No worker specified in the config file dies.
- If workers are moved to different machines, this must be specified in the config file.

Conclusion

This project helped us learn more about the rough internal workings of schedulers like YARN at a fundamental level. It also helped us explore scheduling algorithms, threading, locking and socket programming.

EVALUATIONS:

SNo	Name	SRN	Contribution (Individual)
1	Taran Singhania	PES1201800333	Master + Log Analysis + Report + Debugging
2	Pranav Bhatt	PES1201800764	Worker + Report + Debugging
3	Rehan Vipin	PES1201801655	Master + Worker + Report + Debugging
4	Romaanchan Skanda	PES1201801958	Worker + Report + Debugging

(Leave this for the faculty)

Date	Evaluator	Comments	Score

CHECKLIST:

SNo	Item	Status
1.	Source code documented	
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.	