CS322:Big Data

Final Class Project Report

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

SNo	Name	SRN	Class/Section
1	Hridhay Kiran Shetty	PES1201800068	5 - 'B'
2	Danish Mohammed Ebadulla	PES1201800096	5 - 'A'
3	Rahul Raman	PES1201800146	5 - 'A'
4	Ashish Harish Shenoy	PES1201801447	5 - 'B'

Introduction

Our project YACS aims is to simulate the working of a centralized scheduling framework. The YACS framework is responsible for separating the resource management and the processing layer, for maximum utilisation and efficient usage of the resources. The YACS functionality is to keep track of the available resources and the jobs given from the client side and track the progress of the task execution in workers.

This is the general overview of our project which works as a miniature simulation of YARN, since our YACS framework can only handle map-reduce dependency jobs and runs on a single machine using socket(each worker application works on a different port) and multi-threading concepts. In this simulated version the map-reduce tasks aren't an actual program but just the duration of each task. The Application master splits the independent tasks (map tasks) and runs them on the available worker slot(based on scheduling algorithm). Once it has completed the map tasks for the given job, it deploys the respective job's reduce tasks.

Related work

We have referred through python's original documentation (https://docs.python.org/3/).

The modules that we have used:

- socket - for communication between client - master and master - worker, understanding the buffer size of the socket que.
- threading - Implementing multiple threads and thread locks used when accessing a shared variable
- logging - Used for logging all the events occurred in master and worker later used for debugging purpose and analysis.

Design

The YACS framework has two main components :

- **Master** :
    1. Accepts jobs from the client
    2. Schedules the map tasks and the reduce tasks after the completion of all the map tasks of a given job
    3. Assigns tasks based on the scheduling algorithm of choice(Random, Round Robin, Least Loaded)
    4. Assigns tasks to the respective worker and utilizes the resources(slots) of the workers efficiently
    5. Received updates from workers regarding the task completion and logs all functionality for later analysis and debugging
- **Worker :**
    1. Accepts tasks from the master
    2. Decrements the timer of each task
    3. Once the task duration is over, communicate back to the master regarding the status of the task
    4. Log all of the above functionality for later debugging

Master:

The master component functionality is distributed among two threads :

- One thread is responsible for setting up the simulation environment as provided in the config.json, creating the appropriate TCP sockets for communication between the master-worker (half duplex communication). Next this thread accepts the job sent over the socket (TCP connection) with port number 5000, where the master is the server and the requests.py is the client. The jobs are received in json format, the master extracts the necessary map task details in json format which contains the job_id, task_id, duration and then sends this json to a worker which was selected through the scheduling algorithm (Random, Round Robin, Least Loaded).
- Second thread is responsible for receiving the task completed execution status from the worker from a TCP socket with port number 5001, keeps tracks of the mapper task completion of each and every job, once the map tasks for a particular job are completed, it starts sending the reduce task details(job_id, task_id, duration) to the selected worker based on scheduling algorithm.

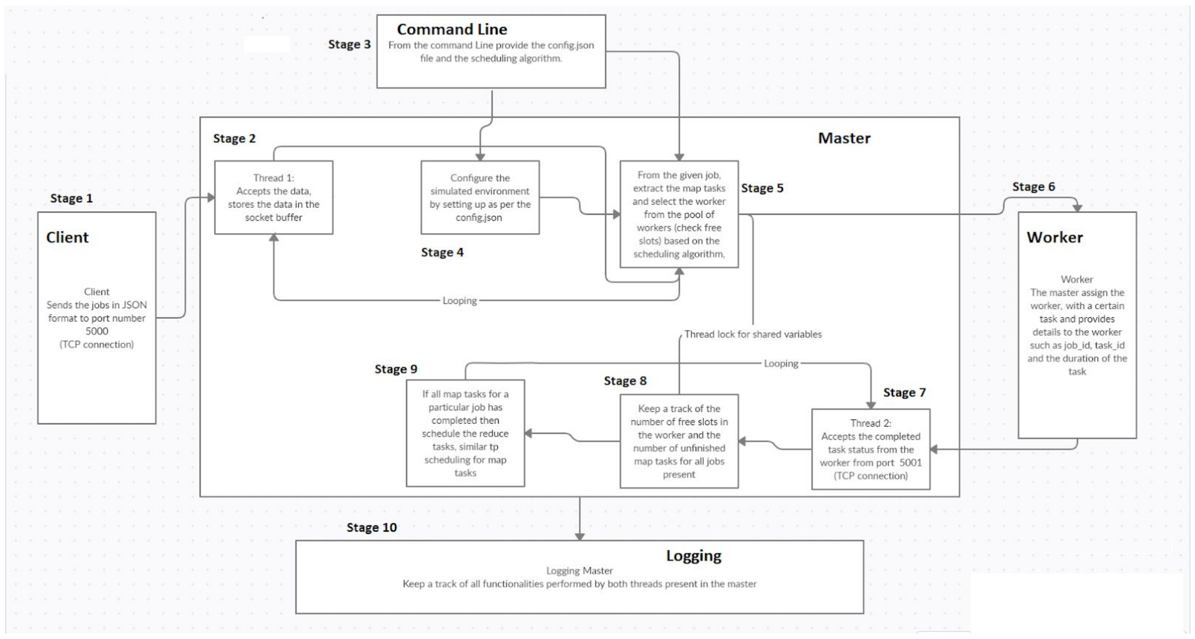
The tasks are scheduled as FCFS (first come first serve) i.e. as soon as the master receives the job from the client, extracts the map tasks and passes it to the selected worker. All worker resources (slots) are tracked in master and decrement the number of free slots for the worker once a task gets assigned. Once the worker communicates back the completed status of the task, free the given slot (increment the number of free slots) and since the increment and decrement are done in two different threads, to maintain synchronisation thread locks have been used.

Now regarding the types of scheduling algorithm, there are three different types and their implementation is as follows :

- **Random :** Using the random module, a random worker is picked from the given workers, check if the worker has free slots and then proceed with assigning the task, else pick another random worker.
- **Round Robin :** Each worker has an equal chance to be picked and they are picked in a cyclic manner and the tasks are assigned accordingly. If there are no free slots for the given worker then it moves to the next worker which has free slots.
- **Least Loaded :** Pick the worker with the maximum number of free slots and assign the task to this selected worker. Since the number of slots keeps incrementing/ decrementing hence selecting the worker is a critical section and is done inside a thread lock.

For each job the number of unfinished map tasks are kept track of, once this reaches zero then it indicates that map tasks for the particular job is completed and the reducer tasks are also scheduled along with other tasks and the scheduling algorithm. This procedure is similar to map task scheduling.

Regarding the logging, each and every functionality such as assigning a task to the worker, getting the completion status from the worker, number of slots assigned to a worker and few other debugging logs are done in real time.


Command Line
Stage 3
From the command Line provide the config.json file and the scheduling algorithm.
Master
Stage 2
Thread 1: Accepts the data, stores the data in the socket buffer
Configure the simulated environment by setting up as per the config.json
Stage 4
From the given job, extract the map tasks and select the worker from the pool of workers (check free slots) based on the scheduling algorithm,
Stage 5
Looping
Stage 1
Client
Client Sends the jobs in JSON format to port number 5000 (TCP connection)
Stage 6
Worker
Worker The master assign the worker, with a certain task and provides details to the worker such as job_id, task_id and the duration of the task
Thread lock for shared variables
Looping
Stage 9
If all map tasks for a particular job has completed then schedule the reduce tasks, similar tp scheduling for map tasks
Stage 8
Keep a track of the number of free slots in the worker and the number of unfinished map tasks for all jobs present
Stage 7
Thread 2: Accepts the completed task status from the worker from port 5001 (TCP connection)
Stage 10
Logging
Logging Master
Keep a track of all functionalities performed by both threads present in the master

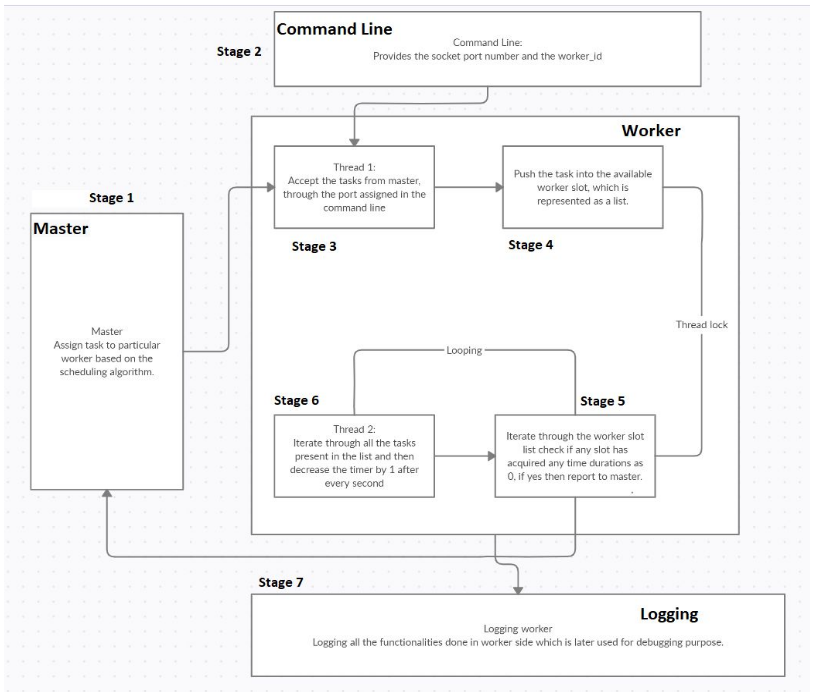
Worker:

The worker component functionality is distributed among two threads :

- One thread is responsible for accepting tasks from the master and assigning a slot for that task.
- Second thread is responsible for decrementing the timer of all the tasks present inside the slots and if any of the tasks have completed their duration then send a message to master regarding the completed status execution of the task by providing the task_id, job_id and the current worker_id in a json format to the master, through a TCP socket connection with port number 5001.

While adding a task given by master or removing a task from the slot, a thread lock mechanism is used to ensure that no race condition occurs. Through the command line the port number and the worker_id is provided for setting up the configuration of the worker.

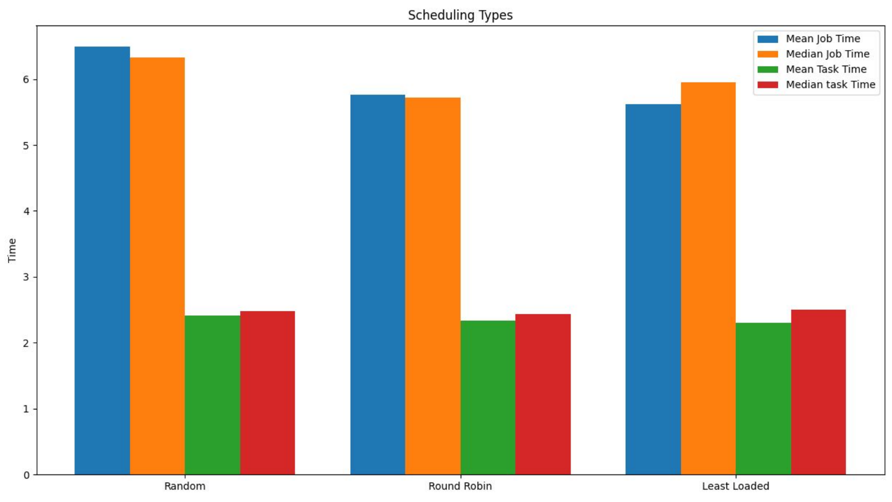
The task duration is reduced by using time.sleep(1), which is after every second the duration for each task present in the slots decreases by 1 second. Once a particular task duration reaches zero seconds, send the job_id, task_id, worker_id of the respective completed tasks to indicate to the master that the particular task is completed.



Stage 2
Command Line
Command Line:
Provides the socket port number and the worker_id
Worker
Thread 1:
Accept the tasks from master, through the port assigned in the command line
Stage 3
Push the task into the available worker slot, which is represented as a list.
Stage 4
Stage 1
Master
Master
Assign task to particular worker based on the scheduling algorithm.
Thread lock
Looping
Stage 6
Stage 5
Thread 2:
Iterate through all the tasks present in the list and then decrease the timer by 1 after every second
Iterate through the worker slot list check if any slot has acquired any time durations as 0, if yes then report to master.
Stage 7
Logging
Logging worker
Logging all the functionalities done in worker side which is later used for debugging purpose.

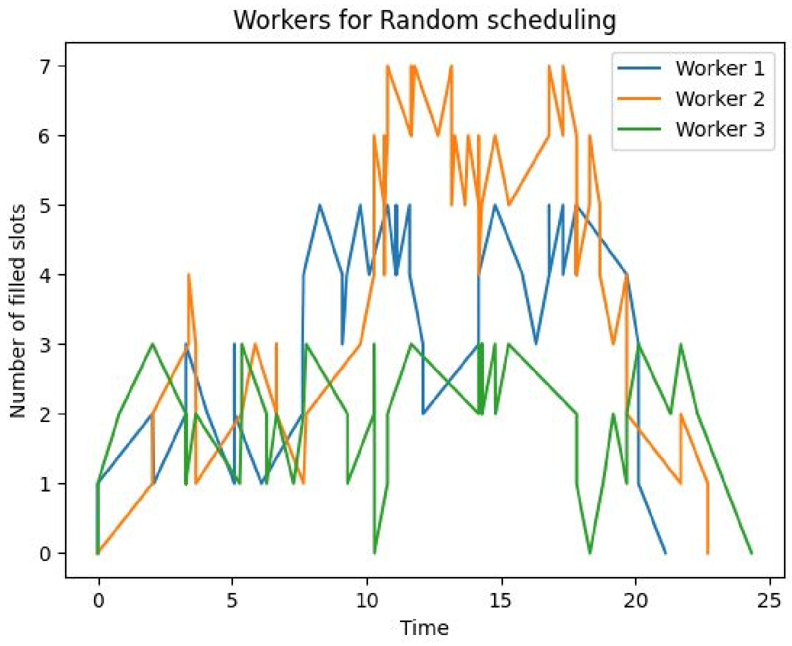
Results

Comparison between the different scheduling algorithms yielded the following results represented as a grouped bar chart : -

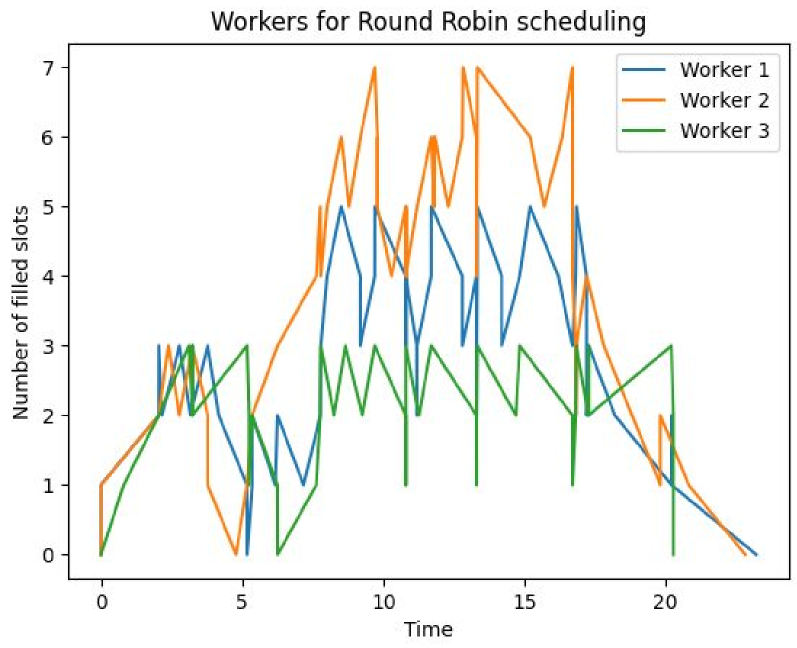


We also plotted a line plot to shadow the number of slots being used against time to observe how the scheduler assigns tasks.

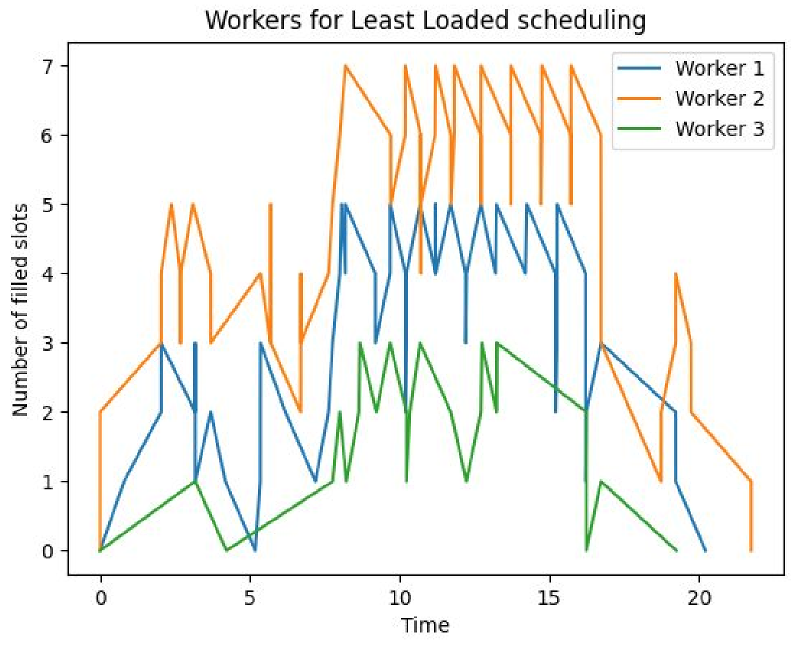
- Random Scheduling :-



- Round Robin Scheduling :-



- Least Loaded :-



Both the above results were observed for 20 generated jobs.

From the above graphs we can infer the following :

- Random scheduling has non uniform usage of the workers and thereby the sockets used vs time shows the sudden spikes in the usage of the worker.
- Round Robin uses all workers in a uniform manner, all workers get the equal number of tasks but then this doesn't utilise the worker slots to its maximum since there can be few workers which have more slots than the other workers.
- Least Loaded is one of the optimal algorithms , since in the case of Least Loaded, the tasks assigned to each worker is dependent on the number of free slots available during that instant. This will increase the efficient usage of the resources but at the same time if there is an imbalance in the number of slots each worker has, then the worker with the least number of slots will hardly be used thereby decreasing parallel computation.

In the current test case, we can observe that Least Loaded takes the least amount of time to compute, this is due to the efficient usage of all slots available for all workers, and another major factor is due to lower rejection penalty time as the number of rejections after selecting the worker (the selected worker only gets rejected if none of the workers had empty slots) is the least when compared to other scheduling algorithms.

Assumptions :

- The socket buffer size, might require a change if the job requests exceed 100
- All the worker port and worker_id assignment should be done through command line and is not automated
- This project only works on a single machine.

Problems

- Fair Scheduling : Currently our job scheduling uses First come first serve algorithm, i.e. the job which arrives first from the requests.py, it's mapper tasks are assigned first and due to this there can be starvation issues where the shorter jobs have to wait behind the longer jobs. We didn't rectify this issue since it's mentioned in the PDF that the jobs are exponentially distributed.
- Burst of worker communication packets: For a larger test case(100 requests) there are chances that the worker communication back to master can happen in a burst and due to this the packets will exceed the socket buffer queue and hence there will be packet loss. To rectify this we have increased the socket buffer size using the socket.listen()
- Fine gradient locking mechanism : Since two different threads were scheduling mapper and reducers our main aim was to ensure a locking mechanism between the mapper and reducer task scheduling, but initially our lock was over a large part of code which slowed down our performance. To overcome this issue we ensured

to keep only the critical section code (shared variables increment/decrement/append/delete) inside the thread lock.

Conclusion

We obtained valuable insights on how YARN schedules the tasks and the efficient utilisation of its resources. It also helped us dive deep into topics such as threading and socket programming.

EVALUATIONS:

SNo	Name	SRN	Contribution (Individual)
1	Hridhay Kiran Shetty	PES1201800068	Worker + Plotting graph + Log Analysis + report
2	Danish Mohammed Ebadulla	PES1201800096	Master + threading concepts + report +Worker Debugging
3	Rahul Raman	PES1201800146	Master + socket concepts + report +Master Debugging
4	Ashish Harish Shenoy	PES1201801447	Master + Worker + Framework designer + scheduling algorithm + Master, worker [Logging + 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.	