CS322: Big Data

Final Class Project Report

Project (FPL Analytics / YACS coding): YACS **Date: 1.12.2020**

SNo	Name	SRN	Class/Section
1	Anind Kiran	PES1201800052	5I
2	Rithwika Mantri	PES1201801081	5C
3	Srija Sistla	PES1201801955	5I
4	Nikitha Jayakumar	PES1201801996	5I

Introduction

This project is focused on implementing a centralized scheduling framework, named YACS (Yet Another Centralized Scheduler). This framework comprises of **Master**, a dedicated machine that manages all resources allocated to the rest of the machines working in the cluster. The other machines are referred to as **Workers**. The Master accepts job requests and implements a scheduling algorithm to assign the tasks, which make up these jobs, to the Workers. The Workers execute these tasks on its slots. The number of slots on each Worker machine is fixed and each slot has enough resources to execute one task at a time. On completion of these tasks, the Workers send an update to the Master.

Related work

First, we had to understand the working of socket programming and threading before starting this project. Then, we built our knowledge regarding the scheduling algorithms. The web was a very useful source for us to learn from, more specifically stack overflow. This along with our knowledge regarding python data structures enabled us to successfully complete the project.

https://www.youtube.com/watch?v=IEEhzQoKtQU

https://pythonprogramming.net/sockets-tutorial-python-3/

Design

Consists of 1 master and 3 workers. Ports have been assigned to the workers, the sockets have been created accordingly and stored in a dictionary.

The first socket is for accepting the job requests and the second one is for listening to the task completion status from the worker. The rest of the sockets are for communication with the respective worker.

---------- initConnect: Master + initRecv: worker ----------

The workers were then launched and connection with the workers was initiated. The resources/slots were allocated to the workers as per the specifications given in the config file which was stored. The workers, on getting the information about the slots, launch the threads representing slots that are used to carry on the tasks.

---------- masterListenerJob ----------

On receiving a job request, the time of arrival, job id along with the **status code – 00** is noted in a log file, the job is then stored in a dictionary with the job id being the key and [no of map tasks, no of reducer tasks, reducer tasks] being the value. The map tasks are first dispatched. The completion of map tasks is checked simultaneously by a completion checker and the reducer tasks for the job are accordingly dispatched.

---------- masterDispatcher: Master + getFromMaster: Worker ----------

The dispatcher finds an appropriate worker to assign the task to using the schedulers. It then assigns the task to that worker and reduces the number of available slots of that worker. On dispatching the task, the time, task id and worker id along with the **status code – 01** is noted in a log file. The worker listens for tasks from the master, from various slots and on receiving a task, run_task is activated.

---------- masterListenerWorker: Master + updateMaster: Worker ----------

Once the task is completed on the worker, it updates the Master along with its port number sent as starboard, which is used to recognize the worker (that has completed the task) by the master. The Master listens for the task completion status from the workers and on receiving a message, it logs the task as completed with **status code – 10** and increments the number of available slots of the worker. If the task completed is a mapper, it decrements the mapper count of the job and if the task completed is a reducer, the reducer count is decremented. If the reducer count comes down to 0, the job is considered complete and this is logged into the file with the **status code – 11**. Once all the jobs received are completed, the threads are joined and the master terminates.

---------- analysis.py ----------

Performance is calculated by reading the log file. The time taken for each task and job is noted in dictionaries, for each scheduling algorithm, along with the worker it was assigned to. This helps in calculating the mean and median time taken for completion of tasks and jobs by each scheduler. The number of tasks assigned to workers by each scheduling algorithm when plotted against time helps analyze the pattern of scheduling.

THREADS:

MASTER	WORKER
1. Master Listener Job 2. Master Listener Worker 3. Completion checker 4. 3 for connection with each worker to allocate the resources.	1 thread per slot to listen for tasks, run them and update the master.

SHARED RESOURCES AND LOCKS FOR MASTER –

1. loglock –

On the logfile, which is shared between threads. It is updated in each of the four cases: when request for a job arrives, when a task is dispatched, when the task is completed by the worker and when the job is completed.

2. requestsdataLock –

On the dictionary of requests data. It consists of the job information, that is the job id – no. of mapper tasks, no. of reducer tasks and the reducer tasks. It is shared between the threads and updated when – a new job request arrives and when a task is completed, by decrementing the number of tasks for that particular job id.

3. configdataLock –

On the list of configuration data, it consists of the configuration details of every worker, that is the port number and slots assigned. It is shared between the threads and updated when – a task is dispatched by decrementing the number of available slots for the worker and when the task is completed by incrementing the number of available slots for the worker.

ASSUMPTIONS:

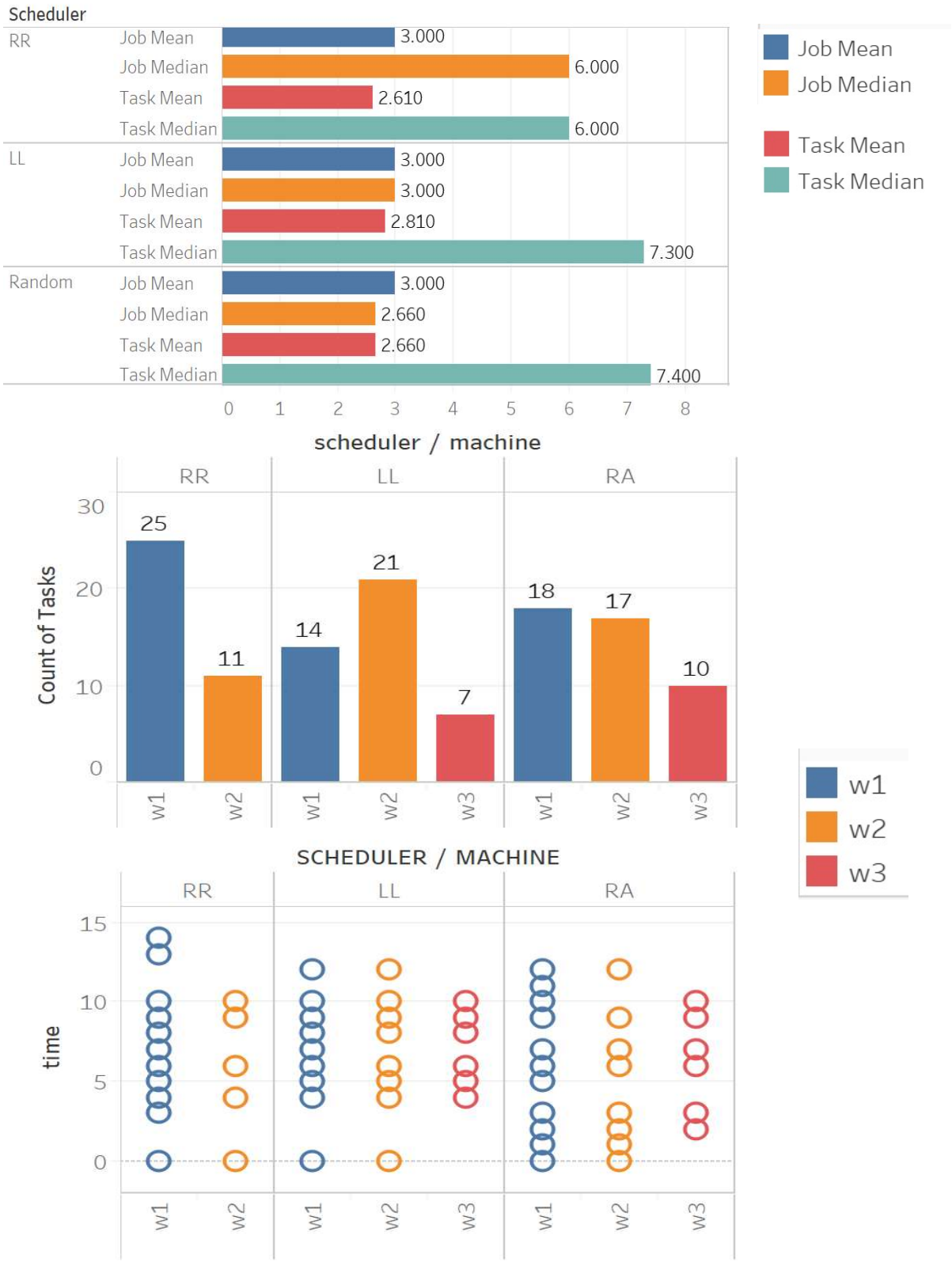
1. The number of workers was hardcoded.
2. The maximum message size was considered to be 1024bytes.
3. Either the master or the worker can be launched first.
4. Each slot of the worker was implemented as a thread.

Result

Round Robin scheduling algorithm puts a heavier load on the first worker as compared to the others. **Least loaded** divides the work according to the resources/slots allocated to each worker, this ensures uniform workload across the machines. Nothing specific can be said about the **random scheduler** as the tasks are randomly allocated.

No solid conclusion could be derived about their efficiency with respect to the job and task completion time from our analysis.

As all the specifications for the workers and jobs were given, we have not experimented with the parameters.


Scheduler
RR
Job Mean
Job Median
Task Mean
Task Median
3.000
6.000
2.610
6.000
LL
Job Mean
Job Median
Task Mean
Task Median
3.000
3.000
2.810
7.300
Random
Job Mean
Job Median
Task Mean
Task Median
3.000
2.660
2.660
7.400
0 1 2 3 4 5 6 7 8
scheduler / machine
Job Mean
Job Median
Task Mean
Task Median
RR
LL
RA
Count of Tasks
30
20
10
0
25
11
14
21
7
18
17
10
w1
w2
w3
SCHEDULER / MACHINE
time
15
10
5
0

Problems

1. Establishing the connections via sockets and ensuring its safe running.
2. Prevention of race conditions.

Both of which were successfully handled.

Conclusion

This project helped us gain a thorough understanding about the working of a centralized scheduler along with the some of the scheduling algorithms and its performance.

It enhanced our knowledge about socket programming and working of threads.

EVALUATIONS:

SNo	Name	SRN	Contribution (Individual)
1	Anind Kiran	PES1201800052	Master implementation
2	Rithwika Mantri	PES1201801081	Implementation of threads
3	Srija Sistla	PES1201801955	Logs and analysis
4	Nikitha Jayakumar	PES1201801996	Worker implementation and socket programming

(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.	