

CS322:Big Data

Final Class Project Report

**Project : \_\_\_\_\_YACS coding\_\_\_\_\_**  **Date: \_\_30-11-2020\_\_\_**

|  |  |  |  |
| --- | --- | --- | --- |
| SNo | Name | SRN | Class/Section |
| 1 | Aneesh Nagesh Bhat | PES1201800216 | G |
| 2 | Apoorv Kashyap | PES1201800183 | G |
| 3 | Praveen Kumar | PES1201800141 | G |
| 4 | Rohit Vishwakarma | PES1201800152 | H |

## Introduction

YACS – Yet Another Centralized Scheduler

This project is about scheduling different tasks on multiple machines.

A job is made of one or more tasks.This scheduling framework receives job requests and launches the tasks in the jobs on the machines in the cluster.As and when a task finishes exectution, the scheduling framework is informed, and the resources are freed. The framework can threreafter assign these freed resources to other tasks.

## Related work

We had to learn about socket programming, Multi-Threading and Mutex Locks in order to implement this scheduling framework.

## Design

Talk about the design of the system, algorithms used, and models implemented. Block diagrams are preferred wherever applicable.

The framework has the option to use one of the following scheduling algorithms:

1.Random scheduling(RANDOM):-

This algorithm chooses a worker machine at random and checks if has any available free slot for task exectution, if it does then launches the task on that machine, else chooses another machine at random, and continues the process.

2.Round-Robin scheduling(RR):-

This algorithm chooses a worker machine in round-robin fashion, and chooses the first machine it encounters which has a free slot for task execution.

3.Least-Loaded scheduling(LL):-

This algorithm chooses a worker machine, which has maximum number of free slots available

and assigns the task to that machine.

If none of the machines have any free slot left then the job scheduler waits for 1 second before trying again.

This framework consists of two main files.

1. master.py

2. worker.py

1.master.py

This file contains the scheduler.It has 3 threads running, for different tasks.

1st thread handles receiving of the job from requests.py or any other external resource, and listens on port 5000. It adds any incoming job to the job pool for further processing

2nd thread extracts mapper and reducer tasks from the job pool, and adds them to their own separate pool. This thread also schedules mapper tasks and sends them to the worker machine.

3rd thread listens for any update from worker machines i.e task completion and frees up the resources, which could be used by other tasks. This thread also schedules reducer tasks and sends them to the worker machine.

It contains 3 important functions for scheduling

1. schedule\_worker

This function first uses a mutex Lock so that threads don’t overlap and might cause a race condition. Based on the scheduler type argument it chooses the scheduling algorithm and chooses the worker machine for task assignment. Once the machine is chosen it releases the lock so that other thread is able to use this scheduler.

2. schedule\_mapper

This function first calls schedule\_worker to get the worker id to assign the task, then chooses the mapper task to be assigned, and marks it as ‘submitted’ in the mapper task pool and returns the mapper task and the worker id to the 2nd thread to be sent to its respective worker machine.

3. schedule\_reducer

This function first calls schedule\_worker to get the worker id to assign the task, then it checks if all the mapper task on which this reducer chosen is has completed execution, only then it chooses reducer task to be assigned, and marks it as ‘submitted’ in the reducer task pool and returns the reducer task and the worker id to the 3rd thread to be sent to its respective worker machine.

The time for each task is the difference in time between when the task was assigned to the worker and when the worker returned it back to the master. The time for each job is the difference in time when the job was assigned to the master and when the last reducer of that particular job completes the exectution.

2. worker.py

This file contains the logic for worker task exectution simulation. It has 2 threads running, for different tasks.

1st thread accepts the task from the master machine and adds the task to the task pool for futher exectution (task simulation)

2nd thread uses the task pool. It first sleeps for 1 second and reduces the ‘duration’ value of each task present in the task pool by 1, to simulate task exectution. If the duration of any of these tasks becomes 0 then it is removed from the task pool and taks completion message, along with the task’s info is sent back to the master machine. It repeats this process until the task pool becomes empty, and waits for further tasks to be sent to the task pool.

## Results

Result- proper exectution of the tasks given to the master. Race condition was unexpected, but later was fixed using mutex locks.

The statistics below are for 10 jobs given by requests.py

Median time taken by tasks in LL is 2.78204607963562

Median time taken by tasks in LL is 7.837675094604492

Average time taken by tasks LL is 3.5517009690750476

Average time taken by jobs LL is 13.957263398170472

Median time taken by tasks in RR 2.1744751930236816

Median time taken by tasks RR 7.256169080734253

Average time taken by tasks RR 2.4967362880706787

Average time taken by jobs RR 6.432512187957764

Median time taken by tasks in RANDOM 3.109616756439209

Median time taken by tasks in RANDOM 7.693694829940796

Average time taken by tasks in RANDOM 2.7008211863668343

Average time taken by jobs in RANDOM 7.332086682319641

## Problems

Race condition was one of the problems face, it was resolved using mutex locks, which prevented thread overlapping.

## Conclusion

Learned about 3 different scheduling algorithms.

Usage of sockets – client and server sockets. One server socket communicating with multiple client sockets

Multi-Threading – learnt how to use threads for parallel processing, writing codes in a way that two threads don’t overlap with each other and cause unexpected results.

Mutex Lock – used in the critical section of the code to avoid race condition between two threads.

## EVALUATIONS:

|  |  |  |  |
| --- | --- | --- | --- |
| SNo | Name | SRN | Contribution (Individual) |
| 1 | Aneesh Nagesh Bhat | PES1201800216 | Wrote most of the code.  Wrote job assignment, listening for task updation code. Communication between master and multiple workers.  Solved race condition using mutex lock  Wrote the entire worker.py file |
| 2 | Praveen Kumar | PES1201800141 | Wrote the code for calculating  time of exectution of each task and job completion |
| 3 | Apoorv Kashyap | PES1201800183 | Wrote all three scheduling algorithm. Wrote important functions like schedule mapper and schedule reducer |
| 4 | Rohit Vishwakarma | PES1201800152 | Plotting graph |

## (Leave this for the faculty)

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Evaluator | Comments | Score |
|  |  |  |  |

## CHECKLIST:

|  |  |  |
| --- | --- | --- |
| SNo | Item | Status |
| 1. | Source code documented | Done |
| 2. | Source code uploaded to GitHub – (access link for the same, to be added in status ) | https://github.com/aneeshnbhat/BD\_YACS\_BIG\_DATA |
| 3. | Instructions for building and running the code. Your code must be usable out of the box. | First run worker.py  syntax: python worker.py port\_num worker\_id  Then run master.py  syntax: python master.py config.json name\_of\_scheduler(RR,LL,RANDOM)  Then run request.py  syntax: python request.py number\_of\_jobs  Once all of the jobs have completed exectution  Then run analysis.py |