Available online at http://icsejournal.com/index.php/JCSE
Journal of Computer Science an Engineering (JCSE)
e-ISSN 2721-0251
Vol. 4, No. 1, February 2023, pp. 15-24

A fault-tolerance model for Hadoop rack-aware resource management system

Moses Timothy[1*], Oladunjoye John Abiodun[2]

[1]Department of Computer Science, Faculty of Computing, Federal University of Lafia, Nasarawa State, Nigeria.
[2]Department of Computer Science, Faculty of Computing, Federal University Wukari, Taraba State, Nigeria.
[1]moses.timothy@science.fulafia.edu.ng*; [2] oladunjoye.abbey@yahoo.com
* corresponding author

ARTICLE INFO

Article History:
Received March 3, 2023
Revised March 21, 2023
Accepted April 5, 2023

Keywords:
Hadoop YARN,
Fault-tolerant YARN,
Rack-aware resource manager,
Fault-tolerance resource management

Correspondence:
E-mail:
moses.timothy@science.fulafia.edu.ng

ABSTRACT

The central resource manager of Hadoop Yet Another Resource Manager (YARN) has posed a major concern to big data analysis and exploration. The central arbiter is overwhelmed whenever there are resource requests by application masters and heartbeat communication from several name nodes in the Hadoop cluster; thereby, degrading the performance of the framework. An attempt to decentralize the resource manager's responsibilities by introducing a new layer in the cluster named the Rack Unit Resource Manager (RU_RM) layer increased cluster performance but introduced a fault-tolerance concern. This work, therefore, developed a fault-tolerant model to allow for efficient and effective data analysis in the Hadoop cluster. A pseudo-distributed computation was set up with the help of the YARN Scheduler Load Simulator (SLS) and WordCount operation performed with varying input sizes. Two fault scenarios were presented and the results obtained showed that with an increase in input size (workload), the running time of the developed fault-tolerant model though slightly higher than that of the existing model, is significantly negligible when compared to the computation bottleneck incurred anytime RU_RM fails. The developed model, therefore, has good performance in the presence of failure of a unit (RU_RM) in the cluster.

1. Introduction

YARN, commonly known as MapReduce 2, is an acronym for "Yet Another Resource Negotiator". Because of its advancements over MapReduce 1, which suffers from a scalability bottleneck when cluster sizes exceed 4000 nodes, it is referred to as the next-generation MapReduce [1]. The fundamental goal of YARN is to divide the duties of the JobTracker into two parts; the Resource Manager is in charge of scheduling one part of the workload, while the Application Master is in charge of monitoring another part of the workload [1-2]. In YARN, job execution happens in phases. The job submission phase, the job startup phase, the task assignment phase, the task execution phase, the progress and update phase, and the work completion phase are among them [3-4]. Compared to traditional Hadoop, YARN is more scalable. Because YARN divides JobTracker's job into two halves, it is simpler to scale up worker nodes beyond 4000 [5]. In the same cluster setting as MapReduce, there may be another distributed architecture that improves resource use through the use of containers. Similar to slots in traditional Hadoop, containers are however configurable. A task in traditional MapReduce will have a set number of map and reduce slots, which are frequently underutilize, some spaces are not being used at all, while others are being overused. This is possible in YARN with the introduction of containers. Though the YARN configuration has improved overall scalability, there remain critical architectural concerns that limit the system's adaptability at extreme sizes. The central resource manager is one of the concerns. This focal resource supervisor is the fundamental component of the Hadoop framework that manages, provisions, and checks assets such as the CPU, memory, and network bandwidth of Hadoop compute nodes. These requirements represent a constraint for Hadoop's scalability. It also slows execution since all compute nodes submit and receive instructions from a single resource

http://dx.doi.org/10.36596/jcse.v4i1.651

management via heartbeat communication. If this central resource manager fails, all executions will come to a halt. Despite the fact that YARN provides Resource Manager High Availability to protect against a single point of failure, this method results in computation overhead since the resource manager must update the backup storage as frequently as feasible.

Previous research [6] developed an improved YARN model in response to this issue. The main objective of the concept was to provide a new layer called Rack Unit Resource Manager (RU RM) to decentralize the overall management of Resource Manager in the YARN framework. Instead of having a single Resource Manager controlling all compute nodes, this layer was created to allow compute nodes on each rack to be constrained by the associated Rack Unit Resource Manager. The methodology reduced a single point of failure that could occur with YARN global resource management and improved response and turnaround times for each job/application. The improved model, however, has a fault tolerance issue. Failure of a RU_RM will lead to a computation bottleneck in the corresponding rack because; all data nodes, their corresponding NameNodes, Application Master resource requests and heartbeats communication will halt. In order to ensure that all Rack Unit resource managers form a peer-to-peer architecture and that each Rack Unit resource manager holds the resources for which it is directly responsible as well as backup copies of those resources for the RU RM that precedes or succeeds it, this work developed a fault tolerance model. In the event that any RU RM fails therefore, the predecessor or successor will be able to administer the compute nodes in that rack until the RU RM succeeds.

2. Review of the architecture of Hadoop rack-aware system

According to research [6] created a rack-aware method that added a layer called Rack Unit Resource Manager (RU RM) to the YARN framework, decentralizing the overall management of Resource Manager (Fig. 1). In place of having a single Resource Manager control all of the network's worker nodes, the layer was introduced to the YARN architecture to allow worker nodes on each rack to be constrained by their own Rack Unit Resource Manager. So, this framework's main goals were to reduce turnaround times for jobs and to guarantee Resource Managers' high availability during job execution. The six (6) phases of the new framework are job submission, job initialization, task execution, progress/update, and job completion.

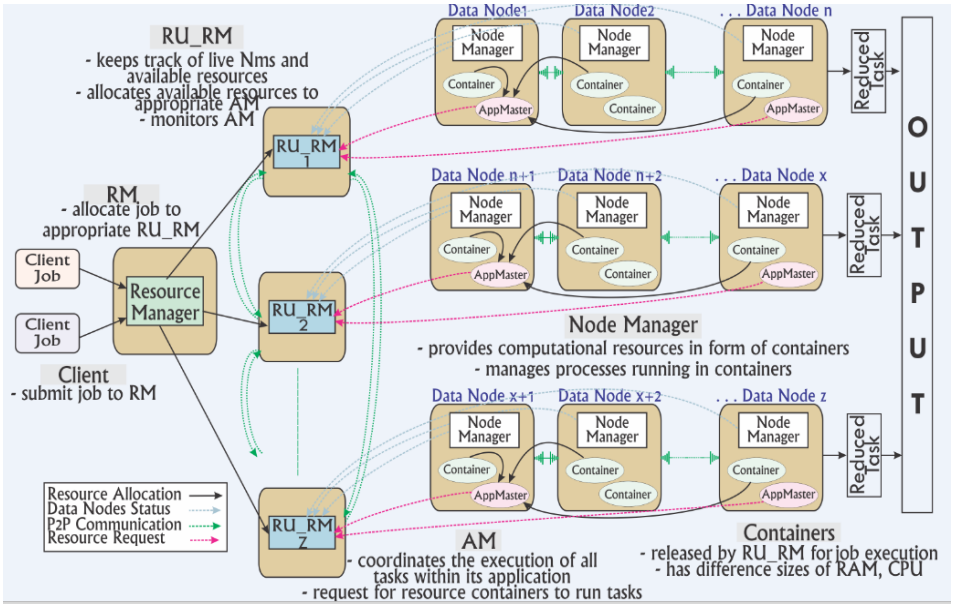

Fig. 1. MapReduce Job Execution on the existing framework by [6]

Fig. 2 details the system design as a whole. Resource Manager now delegates to Rack Unit Resource Manager the duties of scheduling jobs and keeping track of node status using a push-

http://dx.doi.org/10.36596/jcse.v4i1.651

based methodology. Two daemons—the central Resource Manager and the per-rack Resource Manager—serve as resource controllers for job execution as a result of the decentralization of the global resource manager's duties (described as Rack Unit Resource Manager). Failure of the rack unit resource manager will, however, lead to a performance bottleneck in the corresponding rack. The objective of this work is to eliminate performance issues that may occur as a result of RU_RM failure.

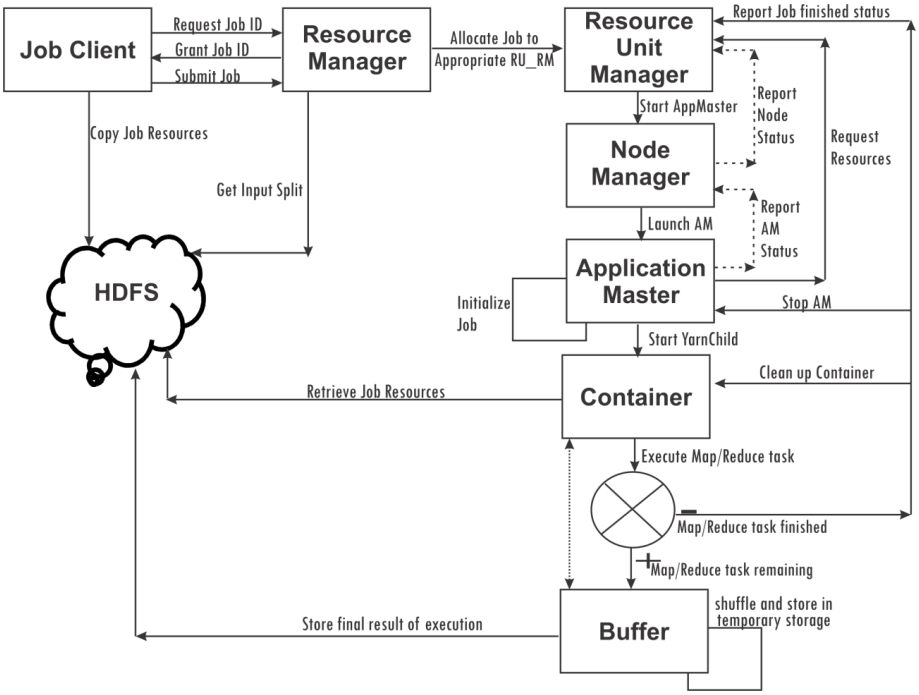

Fig. 2. Block diagram showing whole system design processes of the Hadoop rack-aware resource management system

According to research [6] used two measures (efficiency and average-delay ratio) to compare their work to the existing YARN framework. They define efficiency as the percentage of a task's total ideal finished time ($Total\text{-}T_{ideal}$) to total actual finished time (T_{actual}), as expressed by equation (1). T_{ideal} is run time derived by executing one data block (without overheads). T_{actual} is total running time derived by performing workloads on existing and built model (with overheads). This indicator aids in quantifying the average system usage of their model and the existing model. The average task-delay ratio (r_{td}) was calculated as the normalized difference between the average ideal task finished time (T_{itf}) and the actual task finished time (T_{atf}), as shown in equation (2). This indicator quantifies the speed between [6] work and the exiting YARN model from a task's perspective.

$$\frac{Total - T_{ideal}}{T_{actual}} \; x\,100\% \tag{1}$$

$$\frac{T_{atf} - T_{itf}}{T_{itf}} \tag{2}$$

Typical Hadoop workloads like Sort, WordCount, Terasort, PageRank, Naïve Bayes and k-means were used for the experiment.

Results obtained from their experiments showed that as file size increases, [6] model outperformed existing YARN framework. Though their work has improved scalability, it has fault tolerance issues. Failure of a rack unit resource manager leads to the failure of the entire data nodes in the corresponding rack. This will result in a performance bottleneck of the cluster. This problem, therefore, led to the development of a fault-tolerant model for Hadoop rack-aware system.

http://dx.doi.org/10.36596/jcse.v4i1.651

Other researchers have also attempted to provide solution to the performance bottleneck in Hadoop YARN. According to research [7], developed a model for simultaneous scheduling of map and reduce tasks in order to improve data locality and balance processors' load in both map and reduce phases. A similar work by [8] also proposed a two-stage scheduler called TMaR that schedules map and reduce tasks on servers in order to minimize task finish time during map and reduce operation. While [7] and [8] attempted to reduce network and data traffic, they still employ the global resource manager for the management of data nodes in the cluster. [9-10] and [15] developed architectures that attempted to solve problem of bandwidth utilization in Hadoop cluster. The three works helped curtail the number of intermediate records in shuffle phase of job execution which resulted in overall reduction in job latency and also minimized bandwidth cost. Other works by [11-14] also tried to solve YARN architecture. Common issue with all these works is that, none of the study focused on the decentralization of the central resource manager in YARN for improved cluster performance.

While [6] decentralized the global responsibilities of the Resource Manager in YARN, failure of its RU_RM unit can halt workload processing in the corresponding rack unit. This work therefore, developed a model to ensure fault-tolerant capability for the RU_RM layer in the work of [6].

3. Method

3.1 The developed fault-tolerance model

The pluggable scheduler's main duty in the developed model's central Resource Manager (RM) is to assign workloads to the appropriate RU_RM. It is a pure scheduler in that it doesn't monitor or maintain the status of jobs or applications and makes no promises that unsuccessful tasks will be restarted due to hardware or job failure. Based on the metadata that it received from NameNode, the scheduler carried out its task. After the work has been assigned to the proper RU_RM, RM is released from any further obligations for that job. The responsibility is pushed to the relevant RU_RM by RM using a push-based scheduling approach so that it can be executed. As there is no need for a periodic heartbeat mechanism between RM and RU_RM, RM can manage many tasks and produce superior results. The fault-tolerant model makes sure that all Rack Unit resource managers form a peer-to-peer architecture so that each Rack Unit Resource Manager has backup copies of the resources for the RU_RMs that come before and after it. Every job's predecessor and successor must also be updated in order for any of the RU_RM to be able to perform it. This is crucial to make sure that, in the event of a failure, the RU_RM's predecessor or succeeding unit can assume control of the failed unit. If at any time the predecessor/successor unit of any RU_RM does not receive an update, the model recognizes that an RU_RM has failed. Hence, the responsibilities of the failed RU_RM is assumed by the predecessor/successor unit. The step-by-step process of updates between RU_RM and its predecessor/successor node is described in Algorithm 1.

Algorithm 1 Execution of task and RU_RMs update

```
upon event <execute task>
   RU_RM = node(n)  // n = the number of RU_RM machines on the ring
   check <RU_RM with input splits>
   for node = 1 to n
       If RU_RM(node) == <RU_RM with input splits>
       allocate <job> RU_RM(node)
       Update <RU_RM(node – 1) && (RU_RM(node + 1)>
   next node
end event
```


http://dx.doi.org/10.36596/jcse.v4i1.651

Algorithm 2 made sure that once an RU_RM's predecessor or successor does not receive an update, it is assumed that the RU_RM is unavailable and that the predecessor or successor (depending on which is idle) would take over the failing RU_RM's responsibilities.

Algorithm 2 Rack Unit Resource Manager Failure

```
upon event <RU_RMnode failure>
   RU_RM = node(n)
   If heartbeat <not available> then <mark RU_RM(k)>
   for node = 1 to n
      If RU_RM(node) = RU_RM(k)
      {
        If RU_RM(node – 1) <not expanded>
        {
           If RU_RM(node – 1) <idle>
           transfer <responsibility> to RU_RM(node - 1)
        }
      elseif RU_RM(node + 1) <not expanded>
        {
           If RU_RM(node + 1) <idle>
           transfer <responsibility> to RU_RM(node + 1)
        }
      else
        transfer <responsibility> to RU_RM(node - 1)
      endif
      }
      endif
   next node
   update RM
end event
```

Figure 3 is a representation of the RU_RM ring topology for this design which aids in monitoring failure at the RU_RM layer.

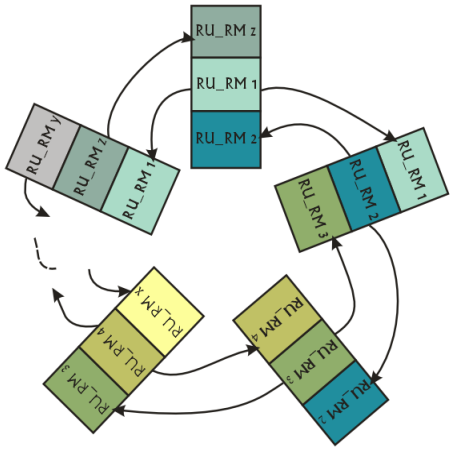

Fig. 3. Ring architecture for RU_RMs in the new model

By the diagram represented in Fig. 3, resources (data items) are replicated on nearby peers (RU_RMs) in the ring. Each RU_RM has resources that it is directly accountable for, as well as resources for RU RM that precedes it and succeeds it on the ring.

Every RU_RM in the ring updates its forerunner and heir. Any RU_RM is deemed dead if, after 3 seconds of execution, no signal is received from that RU_RM. The peer's boundary (RU_RM preceding/following it) will be widened. Fig. 4 details this procedure.

http://dx.doi.org/10.36596/jcse.v4i1.651

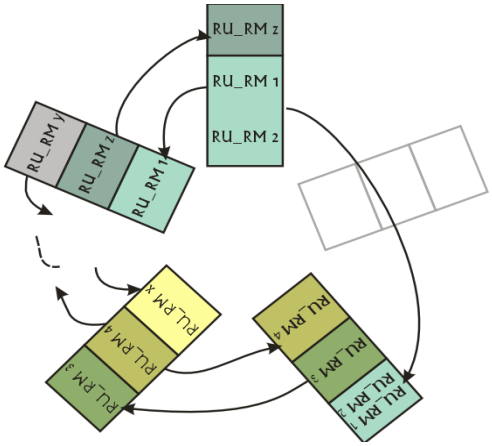

Fig. 4. Failure scenario in the ring architecture of RU_RMs

According to Figure 4, if RU_RM2 fails, the resource control boundary for RU_RM1 or RU_RM3 will be exhausted. The expansion of the boundary depends on two factors: (i) Whether the boundary has previously been increased as a result of another nearby node failing before the RU_RM succeeding the failed peer (RU_RM). (ii) Whether or not the peer (RU_RM) before/succeeding the failed peer is occupied. The optimal-case and worst-case scenarios are both conceivable.

Optimal Case Scenario: According to Figure 4, the peer (RU_RM) preceding the failed peer assumes the duties of the failed RU_RM if RU_RM1 and RU_RM3 have not been expanded as a result of the failure of their neighbor peers and none of them is idle. Now, RU_RM3 will receive updates for both RU_RM1 and RU_RM2, as seen in Fig. 4.

The Worst-Case Scenario: The worst-case scenario is when two consecutive peers occasionally fail, as shown in Fig. 5. A backup will be created at this time to enable execution prior to failure recovery. For instance, if RU_RM3 fails as in Figure 5, the resource control border for RU_RM4 will be widened. In this instance, RU_RM1 will be the only resource manager for RU_RM2. However, if RU_RM1 is the failing peer, RU_RMz's resource control border is expanded and RU_RM3 assumes ownership of RU_ RM2's resources.

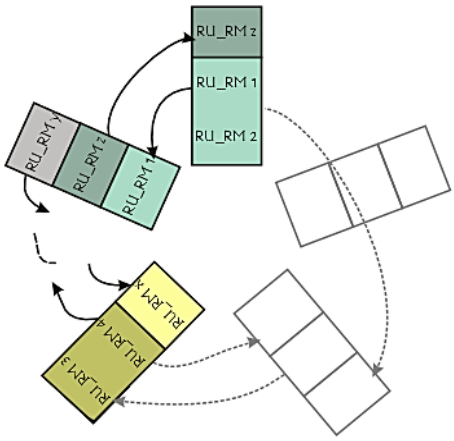

Fig. 5. Failure of two successive peers (RU_RMs) in the ring

http://dx.doi.org/10.36596/jcse.v4i1.651

When a failed peer in either of the two scenarios succeeds and joins the ring, the peer holding it will release its resources to the failed peer (recovered peer). Furthermore, the start of resource backup for peers that come before or succeeds it.

Upon an event join, as determined by Algorithm 3, the system looks for RU RM that isn't already occupying that location in the ring. It contrasts it with RU RM that is prepared to enter the ring. If the comparison is accurate, the RU RM joins the ring and notifies its predecessor and successor. Thereafter, the node receives its responsibility, which runs from its predecessor key + 1 to its key.

Algorithm 3 Rack Unit Resource Manager joining the ring

```
upon event <join>
   RU_RM = node(n)
   specify <RU_RM(k)> to join
   for node = 1 to n
    If RU_RM(node) <not available>
    {
     If (RU_RM(k) = RU_RM(node))
     {
     (RU_RM(node) = RU_RM(k)
     perform <join! RU_RM(node)>
     notify <RU_RM(node–1) && (RU_RM(node+1)>
     release <responsibility> to RU_RM(node)
     }
     endif
    }
    endif
   next node
end event
```

3.2 Experimental Setup

This section evaluates the performance of the fault-tolerant model integrated into the rack-aware model of Hadoop YARN. The intention is to test the performance of the system in the presence of failure. To allow for smooth simulation and to be able to introduce fault at the RU_RM layer during execution, a pseudo-distributed computation was set up with the help of YARN Scheduler Load Simulator (SLS). The architecture of YARN SLS was altered to allow for more RU-RM layers. Hadoop WordCount operation was carried out with three different RU_RMs and their corresponding Node Managers (NM), Application Masters (AM), and Data Nodes (DN). Two forms of execution were carried out in analyzing the results of this experiment. The first execution allows the three RU_RMs to run to a point and then one of the RU_RMs was interrupted (caused to stop working) after 80second. The second form of execution allows a RU_RM to fail at 80seconds but recover from failure at 95seconds to continue execution with its corresponding NM, AM, and DN. These two execution modes were carried out to analyze results obtained when a RU_RM fails and never recovers from failure until execution is completed, and when RU_RM fails but recovers from failure before the end of execution. 128MB was used as block size with block replication set at 3. The experiment was executed on a Toshiba Satellite C55-A, Intel Core-i3 running at 2.40GHz, 64 bits system with a memory capacity of 500GB.

4. Results and Discussion

The results of the experiment setup are shown in Table 1 and Table 2. The tables give a description of WordCount operation on the existing model and the developed fault-tolerant model with two fault scenarios. Input sizes 4GB, 8GB, 16GB, and 32GB were used to show the behavior of the model in the presence of failure.

http://dx.doi.org/10.36596/jcse.v4i1.651

Table 1. Results of wordcount operation for first scenario

Input Size	Run time for the existing model [R_TEM]	Run time for the First Scenario [R_TS_1]	Differences in run time for the First Scenario	
			The difference in run time for the each input size [G_1]	The difference in run time per GB [D_1]
4GB	232s	251s	19s	4.75s
8GB	416s	447s	31s	3.88s
16GB	853s	911s	58s	3.63s
32GB	1652s	1828s	176s	5.50s

Table 2. Results of wordcount operation for second scenario

Input Size	Run time for the existing model [R_TEM]	Run time for the Second Scenario [R_TS_2]	Differences in run time for the Second Scenario	
			The difference in run time for the each input size [G_2]	The difference in run time per GB [D_2]
4GB	232s	253s	21s	5.25s
8GB	416s	435s	19s	2.38s
16GB	853s	869s	16s	1.00s
32GB	1652s	1661s	9s	0.28s

Table 1 and Table 2 shows the running time of the existing model (R_TEM) and the two scenarios in the developed model (R_TS) for the WordCount operation with varying input sizes. This is also described in Fig. 6.

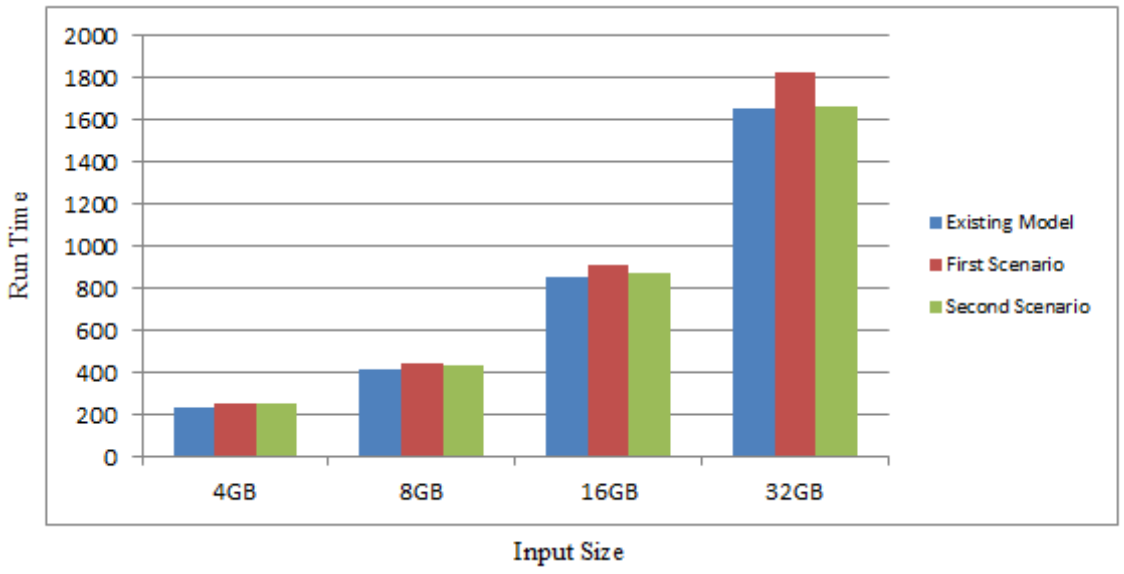

Fig. 6. Shows run time between the existing model and the two scenarios in the developed model.

Also, the differences in running time (G) for the entire cluster and the average response time (D) per GB of the difference in run time were calculated using equations (3) and (4) respectively.

$$G = \boldsymbol{R_T}S - \boldsymbol{R_T}EM \quad (3)$$

$$D = \frac{G}{Input\ Size} \quad (4)$$

http://dx.doi.org/10.36596/jcse.v4i1.651

G helps to understand the time difference (for each input size) between the existing model and the scenarios in the developed model, while D further explains average response time per GB of the difference in time between the existing model and the scenarios in the developed model.

From the first scenario, we observed that G_1 keeps increasing with an increase in the input size. This is because, from the setup, one of the RU_RMs was designed to fail at 80seconds and never to recover from failure. This means that, after 80seconds, only two RU_RMs will be responsible for all data nodes in the three racks contained in the cluster. The execution time will therefore increase since two (instead of three) RU_RM units take heartbeat communication and resource requests from the entire data nodes in the cluster. This scenario though gives a higher running time, it is better than the existing model where once a unit (RU_RM) fails, the entire nodes in the corresponding rack halt all operations. It is also observed from Table 1 that, while the difference in time from 4GB to 16GB in G_1 is considerably moderate, there was a surge in the difference in time when 32GB input size was run. There are two possible reasons. The first reason is due to the capacity of the system used for the experiment. The second reason is because of the overload on the cluster with just two functioning RU_RMs. D_1 shows a steady decrease in time per GB data run on the cluster for input sizes 4GB to 16GB. This implies that, the fault tolerance model developed for this system will improve response time in a larger cluster with numerous RU_RMs.

The second scenario in Table 2 is a case where the system fails but recovers from failure after 15seconds. G_2 shows a significant decrease in running time with an increase in input sizes. The average response time per GB (D_2) also shows the same pattern. The result shows that, with more input sizes, the running time for the existing model and the improved fault-tolerant model will be significantly negligible.

5. Conclusion

For an enhanced scalable resource management system for Hadoop YARN, a fault-tolerant ring design was devised. The developed model's running time showed that with a larger input size, the difference in running time between the existing model and the developed model is minimal. This is an improvement over the previous model, whose RU_RM(s) could fail during task execution. The developed fault-tolerant model, therefore, provides an efficient mechanism to guard against computation issues for worker nodes on any rack due to the failure of its corresponding RU_RM.

Further work may look at prfoviding a fault-tolerant model for Hadoop Distributed File System (HDFS 3.0), which also has more than one Name Node in a cluster.

References

[1] K.V. Vinod, C.M. Arun, D. Chris, A. Sharad, K. Mahadev, E. Robert, G. Thomas, L. Jason, S. Hitesh, S. Siddahart, S. Bikas, C. Carlo, O.M. Owen, R. Sanjay, R. Benjamin, and B. Eric "Apache Hadoop YARN: Yet Another Resource Negotiator". SOCC '13 Proceedings of the 4th annual symposium on Cloud Computing, New York, (2013) NY: ACM, 2013. http://dx.doi.org/10.1145/2523616.2523633

[2] K. Konstantinos, A. Suresh, and C. Douglas "Advancements in YARN Resource Manager". Encyclopedia of Big Data Technoligies: Springer International Publishing, 2018. https://doi.org/10.1007/978-3-319-63962-8_207-1

[3] S. Shenker, and I. Stoica "Hierarchical scheduling for diverse datacentre workloads". Proceedings of the 4th Annual Symposium on Cloud Computing, ACM, Santa Clara, California, 2013.

[4] Apache "Apache Hadoop". Retrieved from https://hadoop.apache.org/, on 3rd March, 2021.

[5] A.T.H. Ibrahim, B.A. Nor, G. Abdullah, Y. Ibrar, X. Feng and U. K. Samee "MapReduce: Review and Challenges". Springer Journal, 109(1), 389-421, 2016. http://www.doi.org/10.1145/1327452.1327492

[6] T. Moses, H.C. Inyiama and S.O. Anigbogu "A rack-aware scalable resource management system for Hadoop YARN". International Journal of High Performance Computing and Networking, 16(1): 1-13, 2020. http://dx.doi.org/10.1145/2523616.2523637

[7] O. Selvitopi, G.V. Demirci, A. Turk and C. Aykanati "Locality-aware and load-balanced static task scheduling for MapReduce". Future generation computer systems, 90: 49-61, 2018. https://doi.org/10.1016/j.future.2018.06.035

http://dx.doi.org/10.36596/jcse.v4i1.651

[8] N. Maleki, H.R. Faragardi, A.M. Rahmani, M. Conti and J. Lotstead "TMaR: a two-stage MapReduce scheduler for heterogeneous environments". Human-centric computing and information sciences, 10(42); 1-26, 2020. https://doi.org/10.1186/s13673-020-00247-5

[9] J. Rathinaraja, and V.S. Ananthanarayana "Multi-Level per Node Combiner (MLPNC) to minimize MapReduce job latency on virtualized environment". 33rd Association for Computing Machinery (ACM) Symposium on Applied Computing, SAC, Pau, France, 2018a. https://doi.org/10.1145/3167132.3167149

[10] J. Rathinaraja and V.S. Ananthanarayana "Dynamic aware reduce task scheduling in MapReduce on virtualized environment". IEEE Computer Society, Kunming, China June 13-15, 2018b. https://doi.org/10.1109/SERA.2018.8477195

[11] K. Hu, J. Hung, H. Chen and S. Rao "Scaling Linkedln's Hadoop YARN cluster beyond 10,000 nodes". Linkedln engineering, 2021. https://engineering.linkedin.com/blog/2021/scaling-linkedin-s-hadoop-yarn-cluster-beyond-10-000-nodes

[12] N. Orensa "A design framework for efficient distributed analytics on structured big data". *A thesis submitted to the College of Graduate and Postdoctoral Studies, Department of Computer Science,* University of Saskatchewan (2021). https://harvest.usask.ca/bitstream/handle/10388/13511/ORENSA-THESIS-2021.pdf?sequence=1&isAllowed=y

[13] GeeksforGeeks "Hadoop YARN architecture". Retrieved from https://www.geeksforgeeks.org/hadoop-yarn-architecture/ on 6th June, 2022.

[14] N.W. Ismahene, S. Boudouda and N. Zarour "A dynamic scaling approach in Hadoop YARN". International Journal of Organization and Collective Intelligence, 12(2):1-17, 2022. https://doi.org/10.4018/IJOCI.286176

[15] J. Rathinaraja, V.S. Ananthanarayana and P. Anand "Fine-grained data-locality aware MapReduce job scheduler in a virtualized environment". Journal of Ambient Intelligence and Humanized Computing, 11(10) :4261-4272, 2018. https://doi.org/10.1007/s12652-020-01707-7

http://dx.doi.org/10.36596/jcse.v4i1.651