

## CHAPTER ONE

### INTRODUCTION

#### 1.1 Background of the Study

Advances in ICT today has made data more voluminous and multifarious and its being transferred at high speed (Sergio, 2015). Applications in cloud like Yahoo weather, Facebook photo gallery and Google search index is changing the IT landscape in a profound way (Stone *et al.*, 2008; Barroso *et al.*, 2003). Reasons for these trends include scientific organizations solving big problems related to high performance computing workloads, diverse public services being digitized and new resources used. Mobile devices, global positioning systems, sensors, social media, medical imaging, financial transaction logs and lots of them are all sources of massive data generating large sets of complex data (Sergio, 2015). These applications are evolving to be data-intensive which processes very large volumes of data hence, require dynamically scalable, virtualized resources to handle them.

Large firms like Google, Amazon, IBM, Microsoft and Apple are processing vast amount of data (Dominique, 2015). International Data Corporation (IDC) survey in 2011 estimated the total world wide data size which they called digital data universe at 1.8 zebabytes (ZB) (Dominique, 2015). IBM observed that about 2.5 quintillion bytes of data is created each day and about 90% of data in the world was created in the last two year (IBM, 2012). This is obviously large. An analysis given by Douglas (2012) showed that data generated from the earliest starting point until 2003 represented close to 5exabytes and rose to 2.7zettabytes as at 2012 (Douglas, 2012). Type of data that has rapid increase is the unstructured data (Nawsher *et al.*, 2014). This is because, these data are characterized by human information such as high-definition videos, scientific simulations, financial transactions, seismic images, geospatial maps, tweets, call-centre conversations, mobile calls, climatology and weather records (Douglas, 2012). Computer world submit that unstructured information account for more than 70% of all data in an organization (Holzinger *et al.*, 2013). Most of these data are not modelled, they are random and very difficult to analyse (Nawsher *et al.*, 2014).

A new crystal ball of the 21<sup>st</sup> century that helps put all these massive data together, classifying them according to their kinds or nature is referred to as Big Data. Big data is a platform that helps in the storage, classification and analysing massive volume of data (Camille, 2015). Hortonworks (2016) defined big data as a collection of large datasets that cannot be processed using traditional computing techniques. These data includes black box data (data from components of helicopter, airplanes and jets), social media data such as facebook and twitter, stock exchange data that holds information about the “buy” and “sell” decisions made on a share of different companies, power grid data like information consumed by a particular node with respect to a base station, transport data which includes model, capacity, distance and availability of a vehicle. Big data can also be seen as accumulation of huge and complex datasets that is too hard to process using database management tools or traditional data processing application with the challenges of capturing, storing, searching, sharing, transferring, analysing and virtualization. Madeen (2012) also see big data as “too big, too fast or too hard for existing tools to process”. “Too big” from Madden’s explanation has to do with the amount of data which might be at petabyte – scale and come from various sources. “Too fast” is data growth, which is fast and must be processed quickly and “too hard” is the difficulties of big data that does not fit neatly into an existing processing tool (Madden, 2012).

The characteristics of big data are better defined by Gartner in Beyer and Laney (2012) as the three Vs (Volume, Velocity and Variety). Volume refers to the amount of data to be processed. Volume of data could amount to hundreds of terabytes or even petabytes of information generated from everywhere (Avita *et al.*, 2013). As organization grows, more data sources consisting large datasets increase the volume of data. Oracle gave the rate at which data grows. It was observed that data is growing at a 40% compound annual rate, reaching nearly 45ZB by 2020 (Oracle, 2012). Velocity is speed at which data grows. According to Sircular (2013), velocity is the most misunderstood big data characteristic. She described data velocity as the rate of changes and combining datasets that are coming with different speeds (Sircular, 2013). Variety has to do with the type of data. Big data accommodate structured data (relational data), semi-structured data (XML data) and unstructured data (word, pdf, text, media logs). From analytics perspective, data variety is seen as the biggest challenge to effectively gain insight in big data. Some researchers believe that taming data variety and volatility will be a key

to big data analytics (Nawsher *et al.*, 2014). IBM came with an additional V for big data characteristic which is “veracity” (IBM, 2012) Veracity addressed the inherent trustworthiness of data. Since data will be used for decision making, it is important to make sure that such data can be trusted (IBM, 2012). Some researchers mentioned “viability” and “value” as the fourth and fifth big data characteristics leaving “veracity” out of the Vs (Biehn, 2013).

These ever increasing data pools obviously have a profound impact not only on hardware storage requirements and user applications, but also on the file system design, implementation and the actual I/O performance and scalability behaviour of today’s IT environment. To improve I/O performance and scalability therefore, the obvious answer is to provide a means such that users can read/write from/to multiple disks (Dominique, 2015). Assume a hypothetical setup with 100 drives, each holding just 1/100 of 1TB data and all of these drives are accessed in parallel at 100MB/second. It then means that 1TB of data can be fetched in less than 2minutes. If same operation is to be performed with just a drive, then it will take more than 2½hours to accomplish same task. Today’s huge and complex semi-structured or unstructured data are difficult to manage using traditional technologies like RDBMS hence, the introduction of HDFS and MapReduce framework in Hadoop. Hadoop is a distributed data storage/data processing framework. Data sets processed by traditional database (RDBMS) solutions are by far much smaller compared with the data pool utilized in Hadoop environment (Dominique, 2015). While Hadoop adopts a brute-force access method, RDBMS solution only banks on optimized accessing routines such as indexes, read-head and write-behind technique (Dominique, 2015). Hadoop excels in an environment that reveals a massive parallel processing infrastructure where data is unstructured to the point where no RDBMS optimization techniques can be used to boost I/O performance (Dominique, 2015). Hadoop is therefore, designed to process efficiently, large data volumes by linking many commodity systems so that they can work as parallel entity. The framework was designed basically to provide reliable, shared storage and analysis infrastructure to the user community. Hadoop has two components – HDFS (Hadoop Distributed File System) and MapReduce framework (Nagina and Sunita, 2016). The storage portion of the framework is provided by HDFS while the analysis functionality is presented by MapReduce (Dominique, 2015). Other components also constitute Hadoop solution suite.

MapReduce framework was designed as a tool for data driven programming model which aims at processing large-scale data-intensive applications in cloud on commodity processors (Dean and Ghemawat, 2008). MapReduce has two components – Map and Reduce (Wang, 2015) with intermediate shuffling procedures and the data formatted as unstructured (key, value) pair (Dean and Ghemawat, 2008). HDFS replicates data unto multiple data nodes to safeguard the file system from any potential loss so that, if one data node gets fenced, there are at least two other nodes holding same data set (Dominique, 2015).

The first generation Hadoop called Hadoop\_v1 was an open source of MapReduce (Bialecki *et al.*, 2005). It has a centralised component called JobTracker that plays the role of both resource management and task scheduling. Another centralized component is the NameNode which is the file metadata server for HDFS that stores application data (Shvachko *et al.*, 2010). With Hadoop\_v1, scalability beyond 4000 nodes was not possible with the centralized responsibility of JobTracker/TaskTracker architecture. To overcome this bottleneck and to promote this programming framework so that it carries other standard programming models and not just implementation of MapReduce, the Apache Hadoop Community developed the next generation Hadoop called YARN (Yet Another Resource Negotiator). This newer version of Hadoop called YARN decouples resource management infrastructure from JobTracker in Hadoop\_v1. Hadoop YARN introduced a centralized Resource Manager (RM) that monitors and allocates resources. Each application also delegates a centralized per-application master (AM) to schedule tasks to resource containers managed by Node Manager (NM) on each compute node (Wang, 2015). The HDFS and its centralized metadata management remains the same on this newer programming model (Wang, 2015). Improvement made on Hadoop\_v1 (by decoupling the resource management infrastructure) enables YARN to run many application frameworks like MapReduce, Message Passing Interface (MPI), interactive applications and scientific workflows. This eases the resource sharing of the Hadoop cluster.

With the scheduler separated from RM and the implementation of per-application master (AM), Hadoop has achieved an unprecedented scalability. However, there are inevitable design issues that are preventing Hadoop from scaling to extreme scales. These issues are in the centralized paradigm in the implementation of some components in YARN framework. This research work therefore, seeks to develop a model solution

that will decentralize the responsibilities of resource manager for scalable resource management in YARN.

## **1.2 Statement of the Problem**

Data driven models like Hadoop have gained tremendous popularity in Big Data analytics. Though great efforts have been made through the implementation of Hadoop framework by decoupling of resource management infrastructure which has allowed Hadoop to scale to tens of thousands of commodity cluster processors, the centralized designs of resource manager and metadata management of HDFS has adversely affected Hadoop scalability (ability to expand the cluster) to tomorrow's extreme-scale datacentres. This challenge therefore, led us to the following problem definition.

- i. How to develop a model alternative that will ensure better scalable resource management in YARN.
- ii. To address scalability issues of Hadoop through decentralized resource management in order to improve response and turnaround time of clients' jobs.
- iii. How to provide a mechanism that will guard against failure of resource management daemons during job execution.
- iv. How to evaluate the scalability of the new model to the existing model using efficiency and average task-delay ratio as performance metrics.

## **1.3 Aim and Objectives of the Study**

The aim of this research work is to develop a model of an improved scalable resource management system for Hadoop YARN.

The objectives of the study are to;

- i. Decentralize the global control of Resource Manager in YARN framework by providing another layer called Rack Unit Resource Manager (RU\_RM) layer.

- ii. Configure RU\_RM layer to ensure that each RU\_RM controls resource requests for compute nodes within its rack instead of a single Resource Manager controlling all the compute nodes in the cluster.
- iii. Develop ring architecture in RU\_RM layer to ensure that all Rack Unit resource managers form a peer-to-peer architecture such that each Rack Unit resource manager holds resources for which it is directly responsible to and also have backup copies of resources for the RU\_RM preceding/succeeding it.
- iv. Carry out a performance evaluation test between the new model and YARN with Hadoop benchmark workload called WordCount.

#### **1.4 Significance of the Study**

The major significance of this research is to deliver an elastic, scalable and easy way to optimize and streamline operations in YARN so as to provide better quality of service to users. The research work seeks to make sure that no single point of failure exists in YARN framework. The global resource manager in YARN is a per cluster resource manager controlling all data nodes in the network. Once this daemon fails, all jobs will halt and have to be restarted. This process however, leads to delay in response time and the execution of jobs in the framework. With the introduction of per-rack resource manager layer in the new model, each rack unit resource manager is directly responsible for its corresponding data nodes. Single point of failure experienced in the existing framework therefore, is eliminated so that jobs will have lower response and execution time.

The introduction of novel ring approach in the rack unit resource manager layer of the new model ensured that all rack unit resource managers form a peer-to-peer architecture such that each rack unit resource manager holds resources for which it is directly responsible to and also have backup copies of resources for the RU\_RM preceding/succeeding it. This will ensure that resources are available to users on demand and efficiently utilized to provide greater turnaround time for safety critical jobs like computer controlled radiation machines in health sector. Failure of a single rack unit resource manager will no longer affect the processing of jobs since the rack unit resource manager preceding/succeeding it will take responsibility of all data nodes within the failed rack. The framework will enable surplus data to be streamlined for any

distributed processing system across cluster of computers. It will scale up single servers to a very large number of machines; each and every of these machines offering local computation and storage space. This will allow for rapid data transfer facilitated by well laid out distributed file system.

The introduction of rack-aware resource manager in this system can now provide a cost effective storage solution to business analysts. Its highly scalable storage/processing capability will facilitate businesses to easily access data sources and tap into different types of data to produce value for such data. Significantly noted is that, data management industry has expanded from software and web into retail, healthcare/hospitals, media and entertainment, information services, finance and government. This creates a huge demand for a more scalable system that will provide excellent data analytic services. Most enterprises/organizations use and analyse lower volume of data with a very large amount wasted. It is a bad practice to term data as unwanted as many part of these data can be put to good use by an organization/enterprise. This system therefore has the capability of storing and processing of large amount of data that can help organization improve the functionality of each and every business unit which includes research, design, development, marketing, advertising, sales and customers handling.

One vital component of data analytics is machine learning. Machine learning is significantly used in medical domain to predict cancer, natural language processing, search engine, recommendation engines, bio-informatics, image processing, text analytics and much more. Machine learning algorithms gain significance where data size is big; especially when it is unstructured, as it means making sense out of thousands of parameters, of billions of data values. Since this system processes large datasets across different cluster of cheap commodity servers, the place of big data comes into picture, which is also significant for running machine learning algorithms. With this rack-aware resource management system, data scientists and engineers will be able to ingest more data and processes into machine learning tools and be sure of lower response time during job execution. An efficient library can be developed to enable running various machine learning algorithms on this system in a distributed manner.

## 1.5 Scope of the Study

This research work considered the resource management of YARN framework and provided a model alternative that help to improve management of applications/jobs in this framework. The scope of this work therefore, is limited to resource manager daemon of YARN framework. Hadoop benchmark workload called WordCount was used to compare the new model and the existing model. The map task counts the frequency of each individual word in a subset data file while the reduce task shuffles and gather the frequency of all the words.

## 1.6 Limitation of the Study

The simulator for Hadoop YARN called YARNsim (Ning *et al.*, 2015) has very limited tools for carrying out this work. The current version (developed recently) does not support fault tolerance models (Ning *et al.*, 2015), only FIFO scheduling algorithm was built into the simulator and it has no plugins that allows modifications to suit research improvements in Hadoop hence, the need to use real test bed. However, this situation will not be a drawback to achieving desired objective for this research work.

## 1.7 Definition of Terms

**Datacenter (DC):-** Is a centralized repository for storage, management and dissemination of data and information organized around a particular body of knowledge.

**Framework:-** Is a layered structure in any programming model, which shows what kind of program can or should be built and how they would interrelate.

**Hadoop:-** Is an open source Java-based programming model which supports processing and storage of large pool of data sets in a distributed environment.

**HDFS:-** Hadoop Distributed File System use in Hadoop framework to store clients data.

**MapReduce:-** A component of Hadoop framework that processes clients data.

**Nodes:-** Just two or more different computers connected within a network.



**Scalable:-** Ability to change in size.

**Scalability:-** Ability of a system or application to continue to function well even when it changes in size or volume in order to meet its desired objectives.

**Task:-** A piece of work to be carried out by system.

**YARN:-** Yet Another Resource Negotiator is the newer version of Hadoop framework used in storage and processing of large data sets.

**Resource Manager:-** Is a core component of YARN responsible for scheduling of jobs and management of compute nodes in a cluster.

**Cluster:-** Is a special type of computational framework designed specifically for storing and analysing huge amounts of structured/unstructured data in a distributed computing environment.

## CHAPTER TWO

### LITERATURE REVIEW

#### 2.1 Theoretical Framework

Relational database technology has proved not to be effective for analysis of massive datasets hence; many organizations have developed technology to utilize large clusters of commodity servers to provide high performance computing capabilities for processing and analysing large datasets (Anthony, 2015). These clusters consist of hundreds or thousands of commodity machines which are connected by high bandwidth networks. The commodity servers are powerful than supercomputers which existed during the early 90's (Anthony, 2015). These commodity servers have led to new trend in super computer design for high performance using clusters of independent processors connected in parallel (Vinayak *et al.*, 2012). Computing problems today are suitable for parallelization, where problems can be divided into a manner that will allow independent processing nodes work on a portion of the problem in parallel. This is achieved by dividing the data to be processed into portions and then combining the final processing results for each portion (Nyland *et al.*, 2000). This method of parallelization is what is referred to as “data-parallelism” or “horizontal portioning” (Nyland *et al.*, 2000) and it is the potential solution to petabytes scale data processing requirements.

Data-parallelism can be seen as a computation applied independently to each data item of a set of data so as to allow a degree of parallelism to be scaled with the volume of data (Anthony, 2015). Why data – parallelism applications are developed, is to allow for scalable performance which result in magnitude improvement. The key issue with this development however, lies in the choice of algorithm, strategy for data decomposition, load balancing on processing nodes, communications between processing nodes and the overall accuracy of the results (Vinayak *et al.*, 2012). Cluster configuration makes it possible to partition data used by applications among available computing nodes and processed independently to achieve performance and scalability based on the amount of data (Anthony, 2015). This parallel processing approach is most times referred to as “shared nothing” approach because, each node consist of its own processor, local memory and disk resources. It shares nothing with other nodes in the cluster. Figure 2.1 gives a clear understanding of this new paradigm.

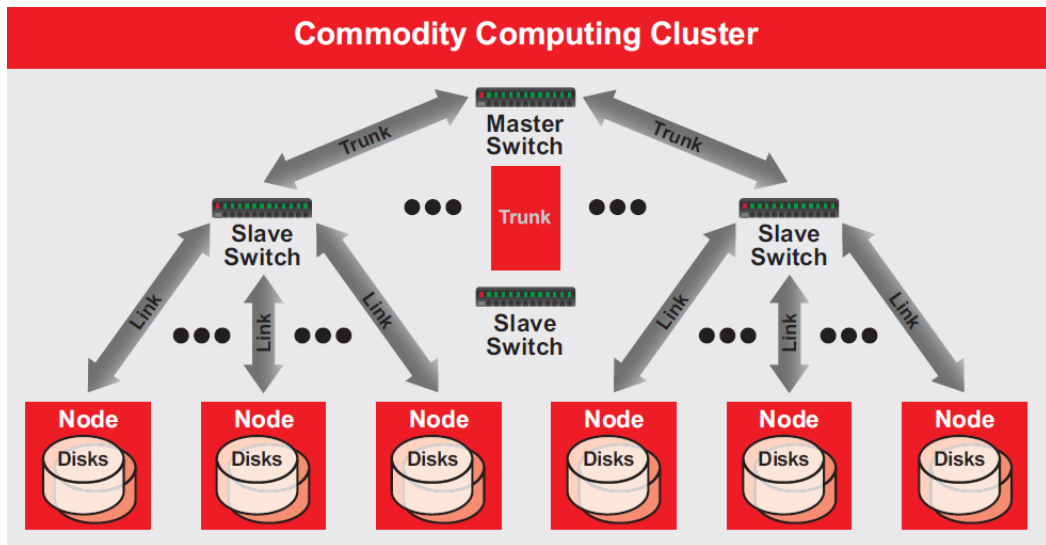


Figure 2.1: Commodity Computing Cluster (Anthony, 2015)

Clusters become extremely effective when it is easy to relatively separate problems into a number of parallel tasks with no dependency or communication required between these tasks other than overall management of the tasks. There are applications which are I/O bound or need to process large volumes of data. These data-intensive applications devote most of processing time to I/O and movement of data. Parallel processing of data-intensive applications requires partitioning of data into multiple segments and processing each segment in a distributed and parallel manner with same executable application program and then re-assembling the results from each node to produce the computed output data (Anthony, 2015). These applications uses distributed data and distributed file systems in which data is located across a cluster of processing nodes such that, instead of moving data to the source machine, program or algorithm is transferred to the nodes with the data that needs to be processed (Anthony, 2015). This approach of “move the code to the data” is extremely effective since program size is usually small in comparison to the large datasets to be processed. It also result in much less network traffic since data can be read locally instead of across the network. This technique makes executable program to process data on the nodes where these data resides thereby reducing system overhead and increasing performance (Anthony, 2015). One of the examples of these cluster technology that describe the characteristics and requirements of data-intensive applications is the Hadoop system.

Hadoop is an open-source Apache Software Foundation (ASF) project which is written in Java programming language that provides cost-effective and scalable infrastructure for distribution and parallel processing of large datasets across commodity of clusters (Shvachko *et al.*, 2010). The programming paradigm was inspired by Google File System (GFS) (Ghemawat *et al.*, 2003) and Google's MapReduce distributed computing environment. The idea was first conceived by Doug Cutting, an employee then with Yahoo and together with Professor Mike Cafarella of the University of Michigan, developed Hadoop later called Apache Hadoop (Shouvik and Daniel, 2013). Hadoop was named after Doug Cutting's son toy elephant (White, 2009). Hadoop has been used to process highly distributable problems across large amount of datasets with commonly available, inexpensive internal disk drives (Ibrahim *et al.*, 2016). There are basically two components in Hadoop (i) Hadoop Distributed File System (HDFS) and (ii) MapReduce framework.

This chapter is in seven sections. The theoretical framework which gives background theory to the research work has been discussed in the first section. The next section will discuss classic Hadoop and its limitations. This will be followed by the next generation Hadoop called YARN in section three. Section four will look at Hadoop clusters and their network topology while section five describes various components of Hadoop ecosystem. Section six will review related distributed data storage and parallel processing platforms and the last section will give clear distinction and research gap between these platforms.

## **2.2 Classic Hadoop**

Hadoop is a scalable, fault tolerant, open source framework for distributed storage and parallel processing of large sets of data on commodity hardware (Hortonworks, 2016). The idea was conceived after Google released a white paper in 2004 describing the use of MapReduce in solving big data problems. By 2005, Doug Cutting and Mike Cafarella (both working at Yahoo as at that time) developed Hadoop, which is a better version of distributed storage and parallel processing framework than Google File System (GFS) and MapReduce (MR) (Hortonworks, 2016). Yahoo donated Hadoop project to Apache in 2006 hence; the name Apache Hadoop. Hadoop has two basic

components; Hadoop Distributed File System (HDFS) and MapReduce (Hortonworks, 2016).

### 2.2.1 HDFS Architecture

HDFS is a master/slave architecture consisting of NameNode called master, secondary node called checkpoint and several DataNodes called slaves (Ibrahim *et al.*, 2016). The major/centralized controller that handles all file system operations is the NameNode hence; any request to the file system (like create, delete and read a file) must go through the NameNode. NameNode also handles block mappings of input files as shown in Figure 2.2. Each file is divided into blocks (default is 64MB) with each independently replicated across DataNodes for redundancy. Block creation, deletion and replication are managed by the DataNode upon instruction from the NameNode (Ibrahim *et al.*, 2016). A periodic heartbeat message is always sent from the DataNodes to NameNode (usually, default heartbeat is 3s) to be sure that there is no loss of connectivity between the two. If NameNode is unable to establish this periodic heartbeat from DataNode, it considers such DataNode out of service, unavailable or dead and hence, will not forward any new request to such DataNode. The NameNode at this point, schedules creation of new replicas of those blocks in the unavailable DataNode on another DataNode (Ibrahim, *et al.*, 2016).

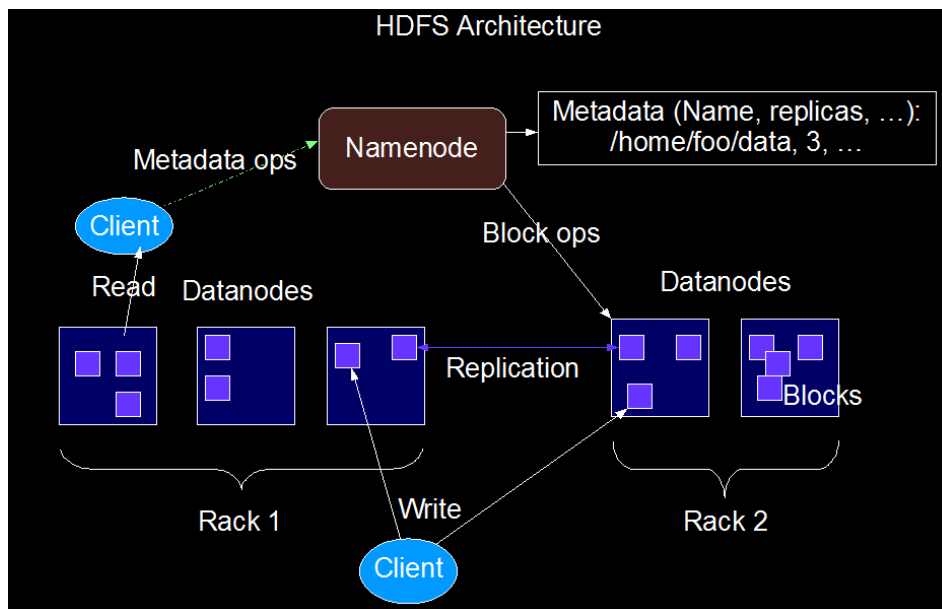


Figure 2.2: HDFS Architecture (Dominique, 2015)

HDFS also provides an option to configure Secondary NameNode, which periodically merges namespace image with the edit log in NameNode to prevent edit log from becoming too large (Dominique, 2015). Secondary NameNode normally is configured on a separate physical system which makes the merge activities CPU intensive. The Secondary NameNode stores merged copy of namespace image so as to provide backup when NameNode is fenced. However, the Secondary NameNode lags behind the primary hence, if the primary data is completely lost, the file system becomes non-functional. Based on the design principle of HDFS, a file is stored as a sequence of (same-sized) blocks, with the exception of the last file block which may be small for a block. Such file size is stored in a sub-size of a block (no internal fragmentation) (Shouvik and Daniel, 2013).

Placement of replicas is paramount to HDFS reliability and performance. Optimizing replica placement is considered as a feature that distinguishes HDFS from most common distributed file systems. A rack-aware replica placement policy will improve data reliability, availability and will optimize network bandwidth utilization. Most HDFS installations execute cluster environment that encompasses many racks (Dominique, 2015). Since inter-rack communication travels through switches and in most configurations, bandwidth potential among nodes in the same rack is greater than the network bandwidth among nodes hosted in different racks, there is need for HDFS replica placement to be rack-aware. One option is to place replicas in different racks so as to prevent data loss in a situation where an entire rack fails. Such policy will evenly distribute replicas in the cluster to provide efficient load balancing and prevent rack failure scenarios. However, it will increase the cost of writing data to different racks, as each write request will require transferring blocks to multiple racks. A common HDFS placement policy is to store 2/3 replicas on different nodes of the same local rack and the last on another node from a different rack (Dominique, 2015). This addresses the inter-rack write scenario and also eliminates the chances of a rack failure with no impact on data reliability and availability guarantees. To minimize network bandwidth consumption and maximize read/write latency without impacting data reliability and availability, we suggest that 2/3 replicas for each complete file block be placed on different nodes in same local rack. The reason is because, the chance of a rack failure is far less than that of a node failure and, a backup replica is in another rack in situation of failure.

## 2.2.2 MapReduce Architecture

Hadoop\_v1 MapReduce (MRv1) framework is based on centralized master/slave architecture (Ibrahim *et al.*, 2016). In this framework, there is a single master server called JobTracker and several slave servers called TaskTrackers (Dominique, 2015). JobTracker represents the centralized program that keeps track of all slave nodes through the TaskTracker and provides interface infrastructure for all job submissions. TaskTracker helps execute actual data stored on each slave nodes. Users submit jobs to JobTracker which inserts the job into pending job queue and execute them based on the type of scheduler chosen. Once a job is submitted, the JobTracker gives a job ID to the client program and starts allocating map task to idle TaskTracker as shown in Figure 2.3.

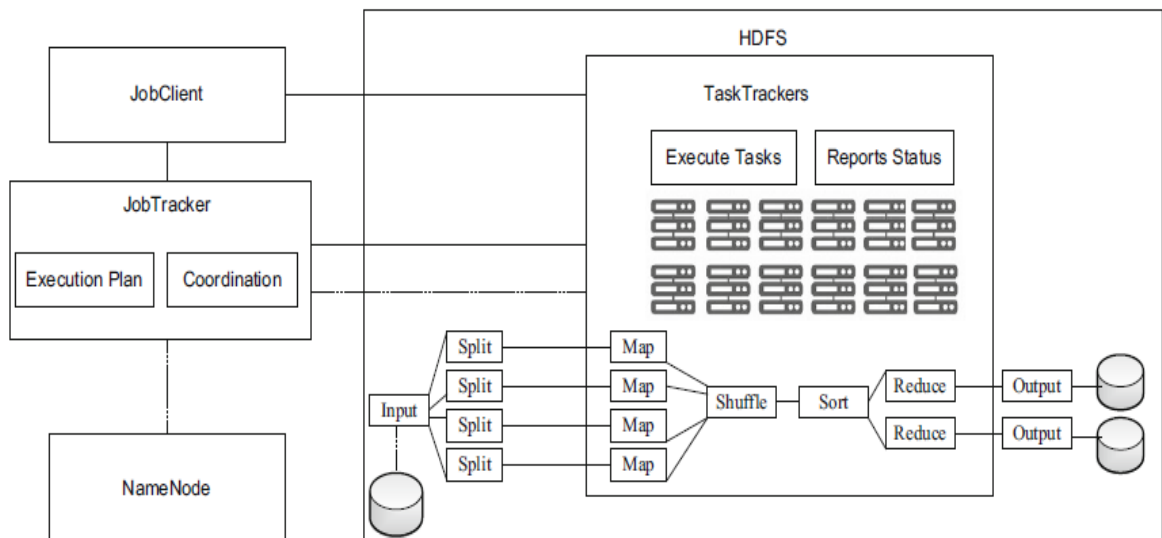


Figure 2.3: MapReduce Architecture (Ibrahim *et al.*, 2016)

Each TaskTracker has a defined number of task slots based on the node capacity. Through periodic heartbeat, the JobTracker knows the number of free slots in the TaskTracker hence; JobTracker can determine appropriate job setup for the TaskTracker. The assigned TaskTracker will then fork a map task to execute the map processing cycle (1 map task for each input split). Map task extracts input data from the split using Record\_Reader and an Input\_Format for the said job. This process invokes the user provided map function, which emits a number of <key, value> pairs in the memory buffer (Dominique, 2015). Once the map task has finished execution, the

commit process cycle is initiated, which flushes the memory buffer to the index and data file pair (Ibrahim *et al.*, 2016), where these index and data file pairs are merged into a single construct. The JobTracker initiates the reduce task through the TaskTracker where these files are concatenated into a single entity. As more map tasks are completed, JobTracker keeps notifying TaskTracker to concatenate files using reduce function until all tasks are completed and an output data generated. This process is shown in Figure 2.4.

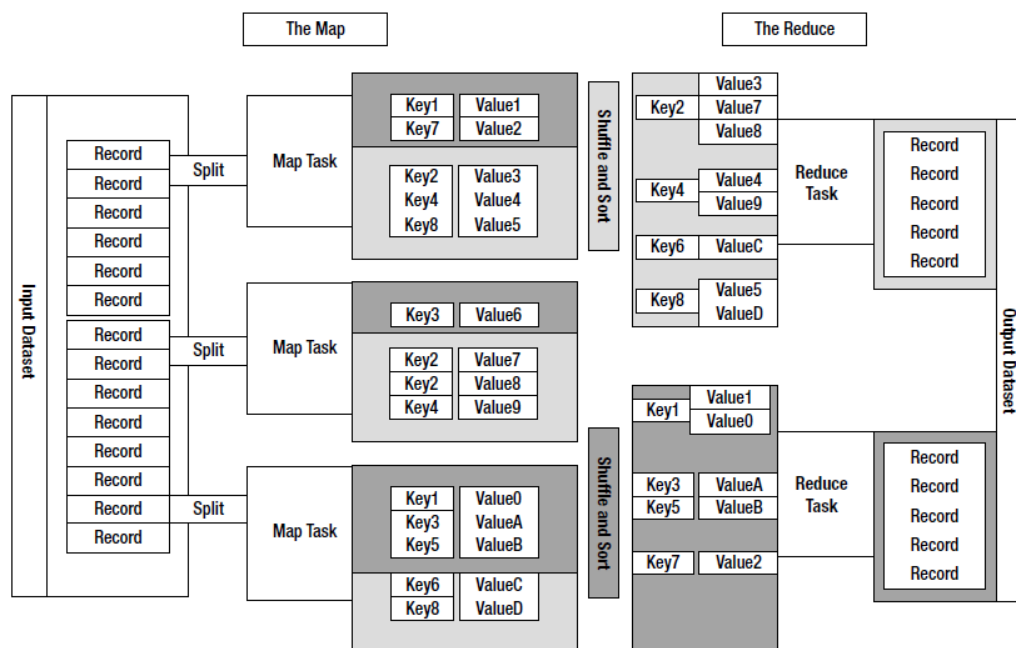


Figure 2.4: Representation of Map and Reduce Tasks (Wissem *et al.*, 2018)

Looking at the centralized nature of JobTracker and its responsibilities, there is a single point of failure. If JobTracker fails, all running jobs will halt. This means that, map tasks and incomplete reduce tasks will have to be re-executed hence, low throughput and high execution time. Facebook observed that single JobTracker in MapReduce framework is a bottleneck and most times, start-up time for a job is several tens of seconds (Dean and Ghemawat, 2008). It was obvious that JobTracker could not handle its dual responsibilities of managing the cluster resources and also scheduling users' job adequately. Facebook noticed that at peak load, cluster utilization dropped precipitously due to scheduling overhead experience in Hadoop framework (Dean and Ghemawat, 2008). Facebook also noticed that polling from TaskTrackers to the JobTracker during a periodic heartbeat is a bottleneck for Hadoop scheduler as it slows down processing



which thereby affect scalability (Dean and Ghemawat, 2008). Another issue is the slot based model. Most times, slot configuration does not match job mix. As a result, once there is need for upgrade to change the number of slots on node(s), all jobs will be killed which is unacceptable (Shouvik and Daniel, 2013). Because of these bottlenecks, Facebook saw the need for a better scheduling framework with better scalability and cluster utilization by lowering latency for small jobs and scheduling based on actual task resource requirements rather than a count of map and reduce tasks (Wang *et al.*, 2013).

Facebook developed Corona (Dean and Ghemawat, 2008), a new scheduling framework that separates cluster resource management from job scheduling. Corona introduced a cluster manager whose sole responsibility is to track nodes in the cluster and the amount of free resources available (Shouvik and Daniel, 2013). In Corona, a dedicated JobTracker was also introduced for each job, and can run either in that same process as the client (for small jobs) or as a separate process in the cluster (for large jobs). What differentiate Hadoop MapReduce framework from Corona is that, Corona uses push-based rather than pull-based scheduling (Shouvik and Daniel, 2013). In Corona, once the cluster manager receives resource requests from the JobTracker, it pushes the resource grants back to the JobTracker which creates tasks and then pushes these tasks to the TaskTrackers for execution. In Corona, cluster manager does not perform any tracking or monitoring of job's progress. The responsibility is left to the individual job trackers. This separation makes scheduling faster since JobTrackers tracks only one job each and has less code complexity. Corona therefore, manages a lot of jobs and achieves better cluster utilization. Figure 2.5 depicts a notional diagram of Corona's main components.

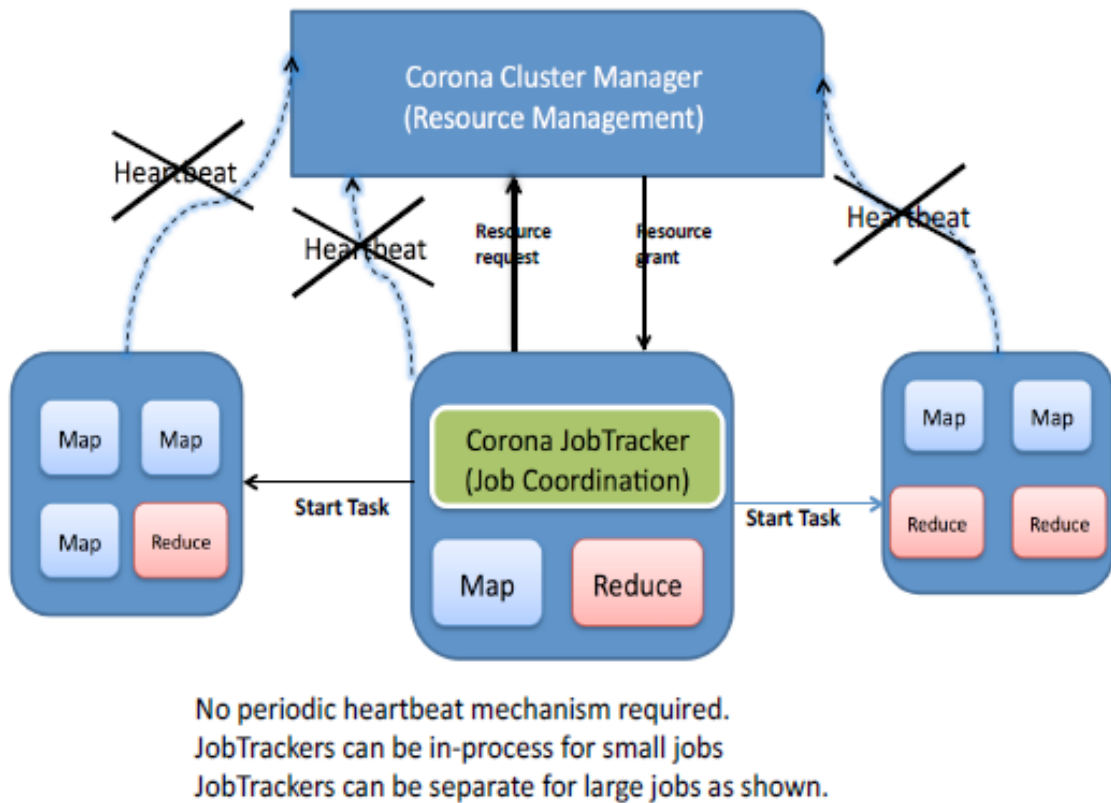


Figure 2.5: Corona's main components (Shouvik and Daniel, 2013)

Yahoo is also a major player in Hadoop ecosystem. When Yahoo discovered that JobTracker is a bottleneck in their huge Hadoop clusters, they developed MapReduce version-2. Hadoop\_v1 has indeed reached a scalability limit of around 4000+ nodes (Shouvik and Daniel, 2013). Scalability requirement of 20,000 nodes and 200,000 cores was required but was not possible with the current JobTracker/TaskTracker architecture (Shouvik and Daniel, 2013). In 2009, YARN was developed. YARN stands for Yet Another Resource Negotiator and it's the next generation of Hadoop (0.23 and beyond). YARN architecture decomposes two principal responsibilities of the old JobTracker into (i) resource management, (ii) job scheduling/monitoring entity. This new design is based on a global (yet centralized) Resource Manager (RM) and a per-ApplicationMaster (AM) daemon (Dominique, 2015). RM and per-node NodeManager (NM) reflects the new data computation framework of YARN. RM consists of ApplicationsManager (AppMg) and a scheduler (Shouvik and Daniel, 2013).

## 2.3 Yet Another Resource Negotiator (YARN)

YARN lifts some functions into a platform layer responsible for management of resources. It leaves the coordination of logical execution plans to a host of framework implementation (Vinod *et al.*, 2013). The per-cluster Resource Manager (RM) in YARN tracks resource usage and node liveness (Vinod *et al.*, 2013). It enforces allocation invariants and arbitrates contention among tenants. Separating units of JobTracker helps the RM use an abstract description of tenants' requirements but remain ignorant of the semantics of each allocation, the semantics of task allocation is the sole responsibility of Application Master (AM). AM coordinates the logical plan of a single job by requesting for resources from RM, generating a plan from resources received and coordinating the execution of the plan around possible faults that may occur (Vinod *et al.*, 2013).

### 2.3.1 Overview of YARN

Resource Manager of YARN run as a daemon on a dedicated machine, and act as a central authority managing resources among various competing nodes in the cluster (see Figure 2.6). RM enforces rich familiar properties such as fairness and locality across commodity servers. Base on the need of an application, scheduling priorities and availability of resources, RM dynamically allocates leases called containers to applications to run on particular nodes in the cluster (Vinod *et al.*, 2013).

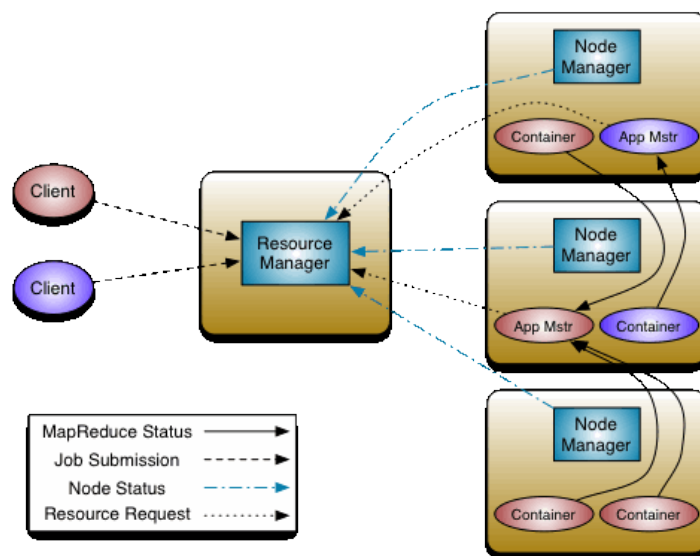


Figure 2.6: Architecture of YARN (Vinod *et al.*, 2013)

Containers are logical bundle of resources (e.g. 4GB of RAM, 2CPU) bound to a particular node (Vinod *et al.*, 2013). To track the amount of containers in the cluster, RM uses a special system daemon called Node Manager (NM) running on each commodity server. The Node Managers are responsible for monitoring of resource availability, containers lifecycle management (e.g. start, kill) and to report any possible fault to the RM. NMs achieve this through a heartbeat communication protocol (Ibrahim *et al.*, 2016).

Job submission to RM passes through a public submission protocol and goes through an admission control phase. This is done to make sure that security credentials are validated and various operational and administrative checks are performed (Vinod *et al.*, 2013). Once a job is accepted, a scheduler in YARN is triggered. If the scheduler has enough resources, the application is moved from accepted state to a running state. Apart from internal book keeping that helps in task management, the next step is allocating a container for the AM and spawning it on a node in the cluster (Vinod *et al.*, 2013). A record of accepted job is normally written to a persistent storage and recovered in case of RM restart or failure.

The AM serves as the “head” of a job (Vinod *et al.*, 2013). It manages all lifecycle aspects of the job including dynamically increasing and decreasing resource consumption, controlling the flow of execution (mappers and reducers), handling faults and computation skew and also perform local optimizations of commodity servers (Apache, 2016). Delegating all these functions to AM helps YARN architecture gained a great deal of scalability, flexible programming model, an improved upgrade/testing capability. To complete a job, AM will need to harness resources like CPU, RAM and disk available on multiple nodes. For AM to obtain a container, it issue resource request to RM with specification of locality preference and what properties a container should possess. Once a resource is released on behalf of an AM, RM will generate a lease for the resource, which is pulled by a subsequent AM heartbeat. To guarantee authenticity when AM presents the container lease to the RM, a token-based security mechanism is put in place (Vinod *et al.*, 2013). Once an AM discovers that a container is ready for its use, it encodes an application-specific launch request with the least (Apache, 2016). A running container communicates with AM through this application-specific protocol to report status and liveness and receive framework-specific commands. In all of these, YARN neither facilitates nor enforces any communication. YARN deployment only

provides basic, yet robust infrastructure for lifecycle management and monitoring of containers, while application-specific semantics are managed by each framework (Vinod *et al.*, 2013).

## **2.3.2 Resource Management components of YARN**

### **2.3.2.1 Resource Manager**

RM exposes two public interfaces and one internal interface (Apache, 2016). The public interfaces are client submitting applications and AM dynamically negotiating access to resources. The internal interface is towards NM's ability for cluster monitoring and resource access management (Apache, 2016). For this dissertation, focus is on the public interfaces as it best explain important frontier between YARN platform and various applications/framework running on it. RM is a global model of cluster state against the digest of resource requirements reported by running applications. This global model of cluster state makes it possible to tightly enforce global scheduling properties. Communication messages and scheduler state therefore, must be very effective and efficient for any RM to scale against application demand and the size of cluster (Vinod *et al.*, 2013). YARN has helped to achieve this with the scheduler handling only an overall resource profit for each application, ignoring local optimizations and internal application flow. While this approach has made YARN scale against application demand, a greater number of commodity hardware in a cluster together with a greater number of applications demand will cause delay in response time (by the scheduler) for each application demand thereby resulting in high turnaround time.

Application Masters codify their need for resources by making one or more ResourceRequests each of which track the number of containers (e.g. 200 containers), resource per container (4GB, 2CPU), locality preference and priority of requests within the application (Apache, 2016). ResourceRequests are designed in a way that it captures full detail of users' needs and/or a roll-up version of it (e.g. one can specify node-level, rack-level and global locality preferences). This approach makes communication and storage requests easier and efficient for the scheduler and also allows applications to express their needs clearly. ResourceRequest roll-up version also guides the scheduler

in a situation where perfect locality cannot be achieved; an alternative can be provided (e.g. rack-local allocation, if the desired node is busy). RM respond to AM request by generating containers together with tokens that grant access to resources (Apache, 2016). Once an application completes its execution, RM forwards an exit status of finished containers as reported by NMs to the corresponding AMs (Apache, 2016).

Looking at the responsibilities of RM, it is important to point out that RM is not responsible for coordinating application execution or task fault tolerance. It does not provide status or metrics for running applications (now part of AM) and it does not serve framework-specific reports of completed jobs (now delegated to a per-framework daemon). RM only handles live resource scheduling of applications with the heartbeat communication from AMs and NMs in the cluster. However, for a greater number of commodity servers and applications demand, response time from the global model of RM will be slow. It is therefore, necessary to provide a per-rack RM to handle all request/communication for NMs and AMs within the local rack with the global resource manager only assigning application demands to each of the Rack Unit Resource Manager (RU\_RM) and monitoring the liveliness of each of these units.

### **2.3.2.2 Application Master**

AMs are daemon inside the worker nodes that coordinate the execution of applications in the cluster. They are also in the cluster just like any other container (Vinod *et al.*, 2013). There is always a periodic heartbeat communication between AM and RM to be sure of liveliness of a container and to update the record of its demand. AM always encodes its preferences and constraints in a heartbeat message to the RM. After the first heartbeat communication, subsequent heartbeats make AM receive a container lease on bundles of resources bound to a particular workstation in the cluster (Apache, 2016). AM can update its execution plan so as to accommodate perceived abundance or scarcity based on the containers it receives from RM (Apache, 2016). Allocation to applications in YARN is late binding. Hence, probability that AM make a request may not remain true when it finally receives its resources, but the semantic of the container are fungible and framework specific. AM also updates its resource asks to the RM if the containers it receives from RM affect both its present and future requirements (Ibrahim *et al.*, 2016).

To explain the role of AM better, let us use MapReduce. In YARN, MapReduce AM optimizes for locality among map tasks with identical resource requirements. Any time AM gets a container, it first matches it against the set of pending map tasks by looking for a task with the input split close to the container. Once this is done, the two other nodes with the replicated input split for that block of data becomes less desirable. AM update its request to RM diminishing the weight on the other k-1 hosts. If the host processing the input split fails, AM will update the RM demanding for compensation from the k-1 hosts (Apache, 2016).

It is clear from the responsibilities of AM that RM does not interpret container status. It is the AM that determines the semantics of the success or failure of the container.

### **2.3.2.3 Node Manager**

Node Manager is a worker daemon in YARN that helps authenticate container leases, manages containers' dependencies, monitors containers' execution and also provides set of services to containers (Vinod *et al.*, 2013). It can be configured to report resources like CPU, memory etc. Once registration is confirmed between RM and NM, the NM heartbeats its status and receives instruction from RM. YARN has Container Launch Context (CLC) that describes all the containers (including NMs) in the framework. CLC has records that include a map of environment variables, dependencies stored in remotely accessible storage, security tokens, payloads for NM services and command necessary to create a process. CLC also include credentials to authenticate download. Any time AM request for a container and it is validated, the NM configures the environment for the container, which includes initializing its monitoring subsystem with the resource constraints specified in the lease (Vinod *et al.*, 2013).

NM is also responsible for killing containers as directed by the RM or the AM. Containers are killed based on; RM reporting its owning application as completed, scheduler decided to evict it for another tenant, AM detects that the container exceeded the limit of its lease or AM discovers that owning application is no longer needed (Apache, 2016). Anytime a container is killed or exits, NM cleans up its working directory in local storage. Also, at the completion of any application, resources own by

its containers are discarded on all nodes and all processes for that application still running in the cluster cancelled (Vinod *et al.*, 2013).

NM periodically monitors the health of physical nodes in the cluster. It checks if there is an issue with the local disk and frequently run an admin configured script to check any hardware/software issues. Anytime a problem is discovered with the node, NM will change its state to “unhealthy” and will report RM through heartbeat protocol about the status of the node. The scheduler makes specific decision of killing containers and/or stopping future allocations to the said node until its health issue is addressed (Apache, 2016).

Careful study of literatures has shown that RM remains a single point of failure. RM will recover from its failure reading its state from persistent storage. After recovery, it kills all running containers and live application and then restart them. This research work therefore, seeks to provide a technique on how to decentralize the functions of RM to each local rack in a cluster environment such that when RM fails, AM continue to work before RM is restored.

## **2.4 Hadoop Cluster and the Network**

The three major layers of machine roles in any Hadoop deployment are the client machines, master nodes and the slave nodes (Brad, 2011). The master nodes are responsible for overseeing two key functional processes that make up Hadoop framework; the HDFS that stores massive data and the MapReduce responsible for running parallel computation (Brad, 2011). The slave node layer makes up the vast majority of workstations that store and also process data. Client machine have Hadoop installed with all the cluster settings so as to enable for loading of data into cluster, submission of mapreduce/other applications describing how the applications should retrieve and process data and how it should view results once task is completed. Figure 2.7 shows typical server roles with these three major layers.



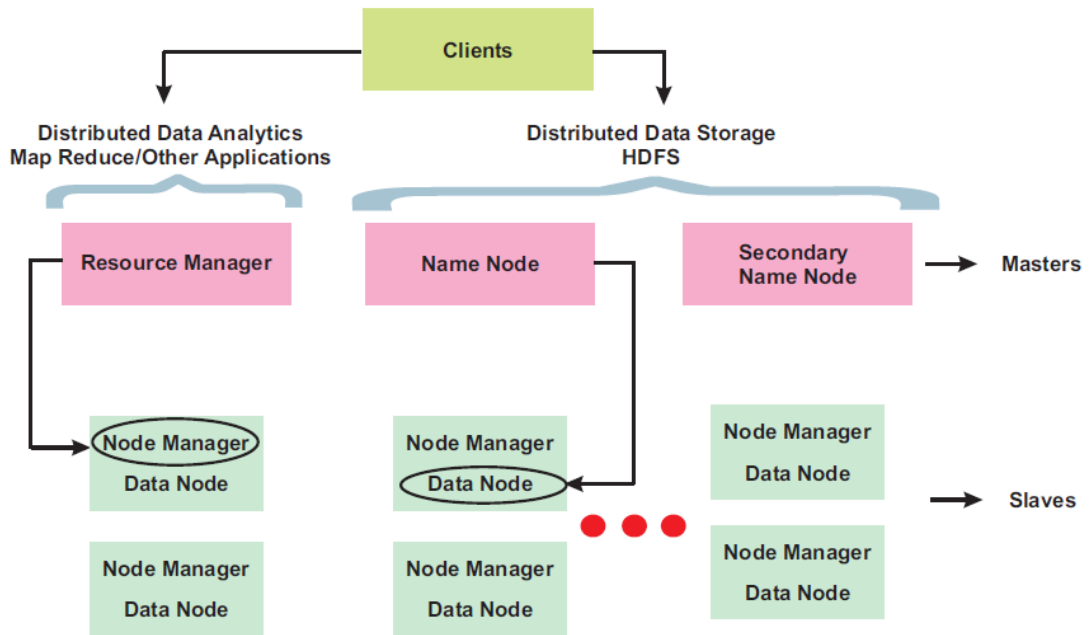


Figure 2.7: Hadoop Server Roles (Brad, 2011)

Typical architecture of Hadoop cluster has rack servers populated in racks connected to a top of a rack switch (Brad, 2011). The rack switch has uplinks which are also connected to another tier of switches, which connects all other racks with uniform bandwidth to form a cluster (see Figure 2.8).

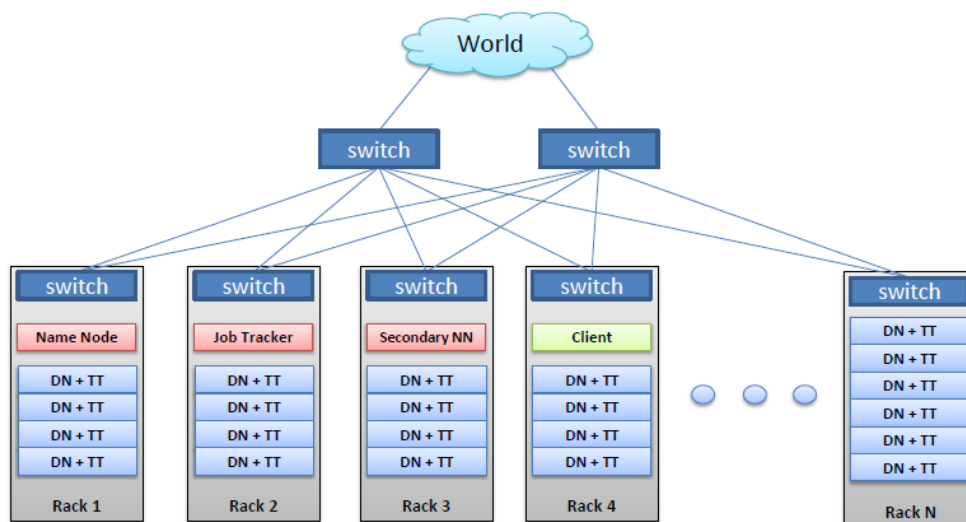


Figure 2.8: Hadoop Cluster (Brad, 2011)

Majority of these servers are slave nodes with lots of local disk storage, moderate amount of central processing unit and DRAM. Master nodes in the cluster have

different configuration favouring more CPU and DRAM. In a typical workflow of Hadoop framework, data are loaded into the cluster (HDFS writes), computations/analysis is carried out on data (MapReduce), results are stored in the cluster (HDFS writes) and results can be read from the cluster (HDFS reads).

Hadoop has the concept of “rack awareness”. The framework gives the client the option to manually define rack number of each slave node in the cluster. There are two reasons for setting rack awareness when storing data in HDFS; data loss prevention and network performance (Brad, 2011). Since data are replicated to avoid losing all copies of data, it is expected that while doing this, all data are not replicated at different nodes on same local rack. If this happened and the rack experiences a failure such as switch or power failure, then that data will be lost. It is also believed that two machines in same rack have more bandwidth and lower latency between each other than two machines in two separate racks. This is true because rack switch uplink bandwidth is usually less than its downlink bandwidth. Also, in-rack latency is lower than cross-rack latency. Hence, network performance can be enhanced if the framework is rack aware.

Figure 2.9 gives a description of how to load data (say file.txt) into the cluster. From the description, file.txt has three blocks (Blk A, Blk B and Blk C). To write Blk A to HDFS, the client consults the NameNode for permission to write file.txt to the cluster. The client gets permission from NameNode and will receive list of three (3) data nodes for the block. The Name Node uses the principle of rack awareness to influence decision as to which data node to provide for the client.

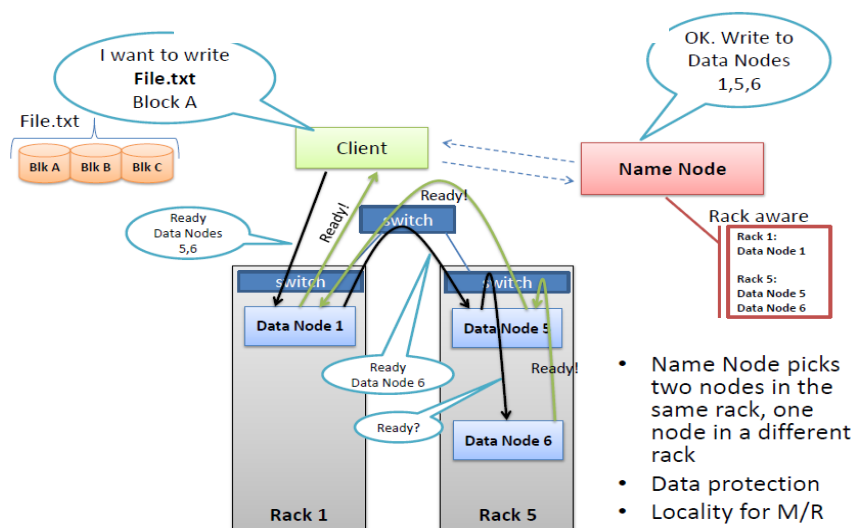


Figure 2.9: Preparing HDFS Writes (Brad, 2011)

Before the client writes Blk A of file.txt to the cluster, it picks the first data node (from the list of data nodes provided by Name Node) and opens a TCP connection to alert the data node to get ready to receive a block. It also release the list of the remaining two data nodes to the first data node to alert them and be sure they are ready to receive duplicate of the block. The first data node (say data node 1) opens a TCP connection to the second data node (say data node 5) to be ready for Blk A and the second data node also opens a TCP connection to the third data node (say data node 6) to be ready for Blk A of file.txt. The acknowledgements of readiness come back on the same TCP pipeline until the first data node sends a “ready” message back to the client. At this point, the client is ready to begin writing block data into the cluster (Brad, 2011).

From Figure 2.10a, a replication pipeline is created as data block is written into the cluster. This ensures that as a data node is receiving block data, it pushes a copy of that data to the next node in the pipeline. This approach shows a primary example of leveraging the rack awareness data in Name Node to improve cluster performance. You notice from Figure 2.10a that the second and third data nodes in the pipeline are in the same rack, hence the final leg of the pipeline does not need to traverse between racks but benefit from in-rack bandwidth and low latency.

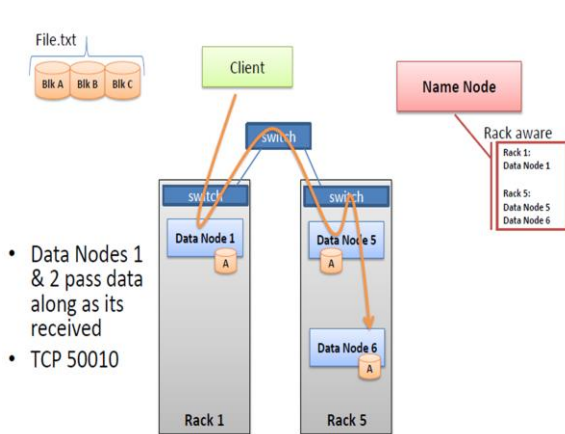


Figure 2.10a: Pipeline Write showing data nodes receiving block data (Brad, 2011)

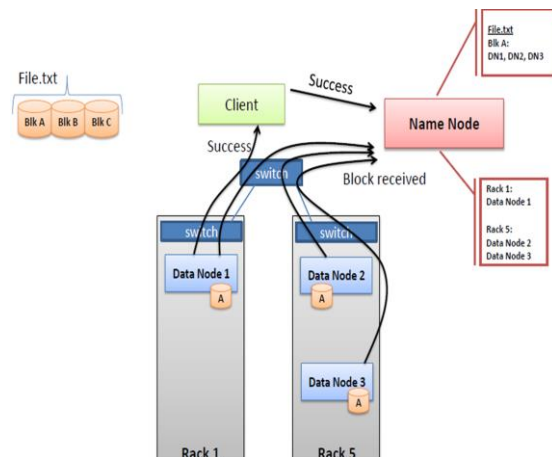


Figure 2.10b: Pipeline Write showing data nodes sending block received report (Brad, 2011).

Once all the three nodes have successfully received the block, the nodes will send a “block received” report to the Name Node as shown in Figure 2.10b. A “success” message is also sent back up the pipeline and TCP sessions close down. The client at

this point receives a success message and notifies the NameNode that data block has successfully been written. The Name Node updates its metadata information with the node locations of Blk A in file.txt. The client will then start the pipeline process for the other data blocks. As other blocks of file.txt are written, the first node in the pipeline vary from the one used for Blk A, spreading around the hot spots of in-rack and cross-rack traffic for replication. This is illustrated in Figure 2.11.

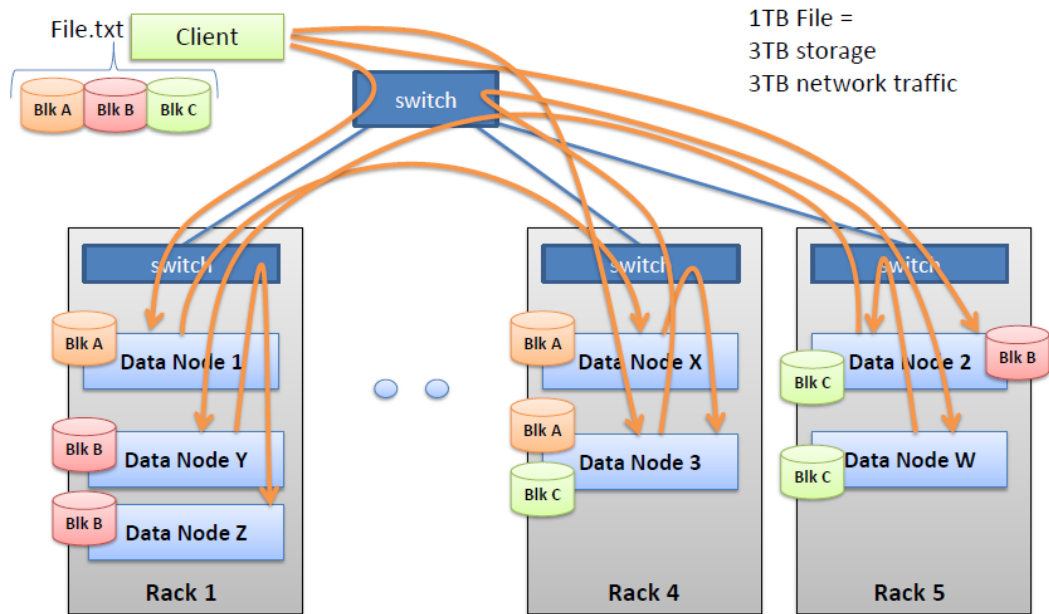


Figure 2.11: Multi-block replication pipeline (Brad, 2011)

Because of the replication factor in Hadoop framework, more bandwidth and storage are used. Assume you have 1TB file to be loaded into cluster, it is approximated that you consume 3TB of network traffic to successfully load the file and 3TB of disk space to hold the file. After successful completion of the replication pipeline of each block into the cluster, it is expected that the file is spread in blocks across the cluster of machines, each having a relatively small part of the data. The more blocks you have in a file, the more machines the data will positively spread. It is also expected that the more CPU cores and disk drive that have a piece of data, the more parallel processing power and faster results. This is the sole motive behind building large and wide clusters (Brad, 2011).

A cluster can scale wide or deep. When more machines are added to a cluster, then we say the cluster is scaling wide. It is expected also that, the network scale appropriately.

Another approach is scaling the cluster deep. In this approach, each machine in the cluster is scale up with more disk drives and more CPU cores. Instead of increasing the number of nodes, you increase the density of each machine. This approach however, requires that you put yourself on a trajectory where more network I/O requirements may be demanded of fewer machines (Brad, 2011).

The Name Node of the HDFS in a cluster holds all the file system metadata for the cluster. It oversees the healthy state of each data node in the cluster and coordinates access to them. It keeps track of the cluster storage capacity making sure that each block of data meets its minimum defined replica policy. Name Node is the central controller of HDFS. It does not hold cluster data itself but knows what blocks make up a file and where these blocks are located in the cluster (Brad, 2011). Anytime client wants to read data, the Name Node points the client to the Data Nodes they need to talk to. Data nodes send heartbeats to Name Node at interval of 3seconds through a TCP handshake using same port number that define the Name Node daemon. Every tenth heartbeats of Data Node to Name Node is a block report that tells Name Node about the blocks it has (Brad, 2011). This report makes Name Node build its metadata and ensure that three copies of each block of data exist on different nodes in different racks (Brad, 2011). Name Node forms a crucial component of HDFS without which client will be unable to read/write to HDFS and will be difficult to schedule map reduce jobs or other applications on Hadoop framework. Anytime heartbeat communication stops between Name Node and Data Node, it is presumed that such Data Node is dead and any data its holding gone as well. Previous block reports received from the said Data Node will help the Name Node to know which copies of blocks died along the node. Using rack aware policy, the Name Node will re-replicate those blocks on other data nodes. The limitation with this however, is when an entire rack of servers falls off the network due to rack switch failure or power failure. It then means that the Name Node will instruct the remaining nodes in the cluster to re-replicate all the data blocks lost in the rack. This process may mean that hundreds of terabytes of data will need to begin traversing the network (Apache, 2016).

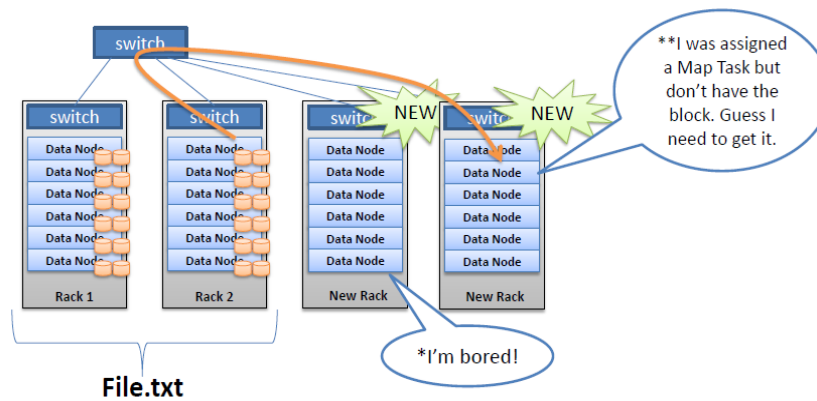
To guard against failure of Name Node, Hadoop has a server called the Secondary Name Node. There is a common misconception however about the responsibility of Secondary Name Node in Hadoop. Many think that its role is to provide availability backup for Name Node but it is not the case. The Secondary Name Node occasionally connects to

the Name Node (by default, every one hour) to fetch a copy of Name Node in-memory metadata and files used to store metadata, which sometimes both daemons may be out of sync. If perhaps the Name Node dies, the copy retained by Secondary Name Node can be used to recover the Name Node but may not be the exact copy of what the Name Node holds before failure. In a busy cluster however, some Hadoop administrators may configure the Secondary Name Node to provide this responsibility much more frequently than the default one hour setting (Apache, 2016).

For a client to retrieve a file from HDFS (say output of a job), the client will have to contact the Name Node and ask for the block location of the file (Brad, 2011). The Name Node will then release a list of each Data Nodes holding each block of file to be retrieved. From the list released to the client, it picks the first data node and reads a block at a time. Client will not progress to the next block until the previous block completes (Apache, 2016).

There are situations where a data node may also need to read a block from another data node in the cluster. This situation is possible if a data node has been asked to process a data that it does not have locally. In this case, the data node will need to retrieve the data from the data node that holds the block over a network before it can begin process. This situation is another key area that makes Name Node rack awareness principle an optimal solution for network performance. When the data node request from Name Node the location of the block to be processed, the Name Node checks if a data node on same rack has the block. If so, Name Node provides in-rack location from which to retrieve the block of data. In this situation, the flow of data does not traverse two or more switches to create a congested link between two racks. Data can be retrieved quicker in-rack and processing begins early. Jobs are also completed much faster.

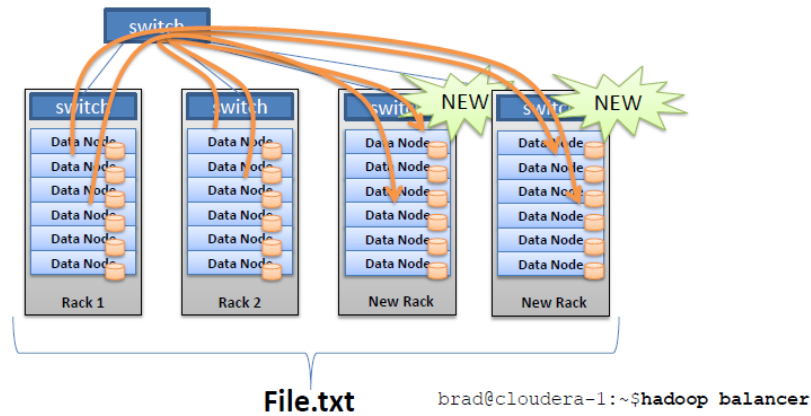
There are situations in Hadoop cluster that you need to add new racks full of servers and network to an existing cluster. It is possible to have what is called “unbalanced cluster” in this situation.



- Hadoop prefers local processing if possible
- New servers underutilized for Map Reduce, HDFS\*
- Might see more network bandwidth, slower job times\*\*

Figure 2.12: Unbalanced Cluster (Brad, 2011)

Figure 2.12 gives a pictorial representation of what it means to have an unbalanced cluster. As described in Figure 2.12, Rack 1&2 were the existing racks containing block of files. When two new racks were added to the cluster, no block of data was spread from the old racks to the new ones (Brad, 2011). This means that, the two new racks will remain idle until they are loaded with files. Again, if data nodes in Rack 1&2 are busy, they will have no choice than to assign jobs to data nodes in the new racks. The data nodes in the new rack will therefore attempt to grasp data over the network to begin processing. As a result, more network traffic and slower job completion times occur. To avoid this situation of unbalanced cluster, Hadoop includes a nifty utility called “balancer” in its framework. Balancer will look at difference in available storage space between data nodes in the cluster and will try to provide balance to a certain degree. New data nodes with free storage space will be detected by this utility and copying of data blocks off data nodes with less storage space to nodes with free space will begin. This scenario is described in Figure 2.13. This utility however, must be set by the administrator anytime the need arises and anytime the command is cancelled, the operation stops (Brad, 2011).



- Balancer utility (if used) runs in the background
- Does not interfere with Map Reduce or HDFS
- Default speed limit 1 MB/s

Figure 2.13: Balanced Cluster (Brad, 2011)

## 2.5 Hadoop Ecosystem

Hadoop ecosystem provides furnishings that turns Hadoop framework into a comfortable home for big data activities. Figure 2.14 gives an overview of some components built on top of Hadoop framework. The description in Figure 2.14 shows only few out of many components in Hadoop ecosystem. While some of these components in the ecosystem are intended to supplement one or two of Hadoop’s core elements (HDFS and MapReduce), others are commercially available framework solutions that provides more comprehensive functionality (Hortonworks, 2016).

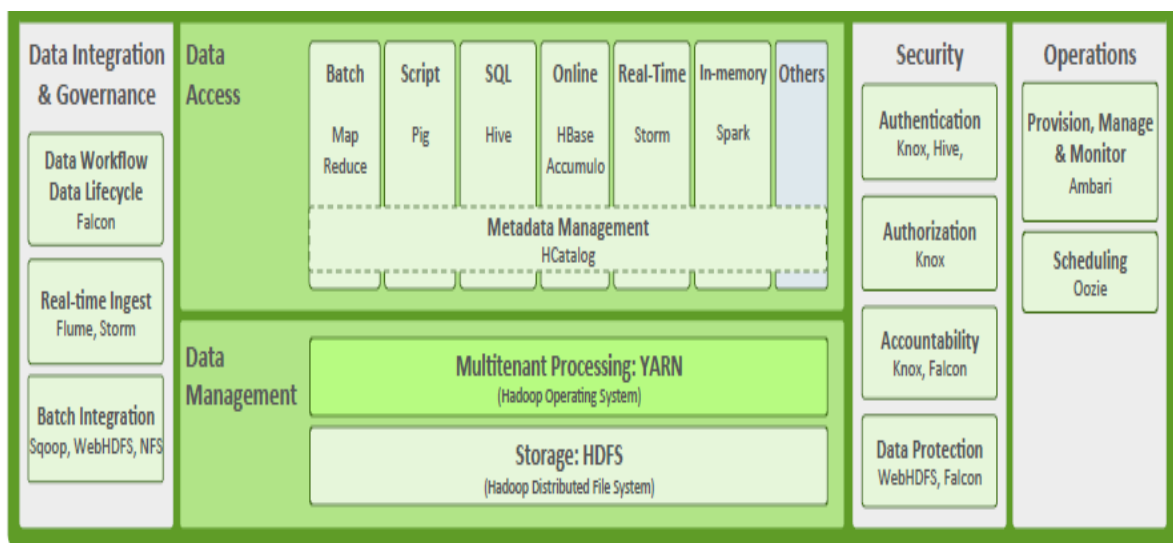


Figure 2.14: Hadoop Ecosystem (Garcia, 2014)



Hadoop ecosystem is in sections; data integration and government has mechanism for data workflow/lifecycle, real-time data ingest and batch integration mechanism. The data access section is for metadata management and allows the use of high-level programming languages to access data in HDFS. Data management section is the core of this framework. It stores, manages and process data. Security has mechanism for authentication, authorization, accountability and data protection while operations section is to allow addition of nodes to a cluster and for provisioning, managing, monitoring and scheduling of applications. We will have a closer look at some of these components.

**Falcon:-** This component is a data management framework for simplifying data lifecycle management and processing pipelines on Apache Hadoop. It enables users to configure, manage and orchestrate data motion, pipeline processing, disaster recovery, and data retention workflows. Instead of hard-coding complex data lifecycle capabilities, Hadoop applications can now rely on the well-tested Apache Falcon framework for these functions. Falcon's simplification of data management is quite useful to anyone building apps on Hadoop. Data Management on Hadoop encompasses data motion, process orchestration, lifecycle management, data discovery, etc. among other concerns that are beyond ETL. Falcon is a new data processing and management platform for Hadoop that solves this problem and creates additional opportunities by building on existing components within the Hadoop ecosystem (ex. Apache Oozie, Apache Hadoop DistCp etc.) without reinventing the wheel (Hortonworks, 2016).

**Sqoop/Flume:-** These components are distributed, reliable and available service in Hadoop ecosystem that efficiently collects, aggregate and move large amount of big data into HDFS (Hortonworks, 2016). It is important to note that Hadoop does not create data. Data is usually created in other systems and brought into Hadoop. There is need for a mechanism therefore, that will help get data into Hadoop. Two of these mechanisms are Sqoop and Flume. Sqoop is a mechanism to get data from relational databases like Oracle, SQL server and MySQL. It is not just a mechanism to get data into Hadoop but also help get data out of Hadoop to these relational database systems. It has an input/output utility which helps in carrying out these operations. The word 'Sqoop' is from 'Sq for SQL' and 'oop for Hadoop'. Most data brought into Hadoop may not be structured data. To help get unstructured data into Hadoop therefore, Flume is used. It has a simple and flexible architecture based on streaming data flows. It is

robust and fault tolerant with tuneable reliability mechanisms and many failover and recovery mechanisms. It uses a simple extensible data model that allows for online analytic application (Hortonworks, 2016).

**WebHDFS:-** This component support users of HDFS in operations that includes reading of files, writing to files, making directories, changing permissions and renaming. It is the first class built-in component of HDFS that runs inside Namenodes and Datanodes in the cluster (Hortonworks, 2016).

**Pig:-** This component is created by Yahoo. The purpose of this component is to provide a high-level API that can be written in English-like statement using words like join, soft, filter etc in Java codes. So, when a user writes in pig script and run it on Hadoop, it internally produces MapReduce equivalent making life easier for the developer.

**Hive:-** Is a component created by Facebook and use on top of Hadoop ecosystem. It was created to make Hadoop easier. Hadoop is created using Java language but most users are familiar with SQL hence, the need for Hive. It is also a high-level language created to assist developers familiar with SQ language.

**HBase:-** Is Google BigTable Inspired. Non-relational distributed database. Random, real-time read/write operations in column-oriented very large tables (BDDDB: Big Data Data Base). It's the backing system for MR jobs outputs. It's the Hadoop database. It's for backing Hadoop MapReduce jobs with Apache HBase tables (Hortonworks, 2016).

**Knox:-** It is a system that provides a single point of secured access for Apache Hadoop clusters. The goal is to simplify Hadoop security for both users (i.e. who access the cluster data and execute jobs) and operators (i.e. who control access and manage the cluster). The Gateway run as a server (or cluster of servers) that serves one or more Hadoop clusters (Hortonworks, 2016).

**Ambari:-** Is an open source mechanism that helps create a cluster, provision it, manage and monitor the cluster. For example, this mechanism can be used to add a node to a cluster.

**Oozie:-** Workflow scheduler system for MR jobs using DAGs (Direct Acyclical Graphs). Oozie Coordinator can trigger jobs by time (frequency) and data availability (Hortonworks, 2016).

Other components in Hadoop ecosystem are summarized in Table 2.1. The components are summarized based on their categories/functions.

**Table 2.1:** Hadoop Ecosystem

S/N	Category	Component
1.	Distributed File System	HDFS, Red Hat GlusterFS, Quanteast File System (QFS), Ceph File System, Lustre File System, Alluxio, GridGain, XtremFS
2.	Distributed Programming	Apache Ignite, MapReduce, Pig, JAQL, Spark, Storm, Flink, NetFlix, Apache REEF, Apache Twill, Apache DataFu, Apache Hama, Pache Beam
3.	NoSQL Databases	
	Column Data Model	Hbase, Cassandra, Hyper table, Accumolo, Kudu, Parquet
	Document Data Model	MongoDB, RethinkDB, ArangoDB
	Stream Data Model	Eventstore
	Key-value Data Model	Redis Database, LinkedIn, RocksDB, OpenTSDB
	Graph Data Model	ArangoDB, TitanDB
4.	SQL-on-Hadoop	Hive, HCatalog, Cloudera Impala, Facebook Presto, Splout SQL, Apache Tajo, Phoenix, MRQL
5.	Data Ingestion	Flume, Sqoop, Facebook Scribe, Kafka, Netflix Suro, Cloudera Morphline
6.	Machine Learning	Apache Mahout, Cloudera Oryx, MADLib, Apache SystemML
7.	Security	Apache Sentry, Apache Knox Gateway, Apache Ranger
8.	Metadata Management	Metascope
9.	System Development	Apache Ambari, Cloudera HUE, Apache Mesos, Hortonworks HOYA, Apache Helix, Apache Bigtop, Cloudbreak, Apache Eagle
10.	Applications	Apache Nutch, Apache OODT, HIPI Library

Source:- Hortonworks, 2016.

## 2.6 Review of Related Literatures

Information technology world has been facing big data challenges for over few decades. The term ‘big data’ has been for over four decades, it’s just that the definition has been changing (Vinayak *et al.*, 2012). In the 70’s big meant megabytes; at a time, big grew to gigabytes, then terabytes and petabytes. Today’s IT notion has grown to exabytes and zettabytes are presumably in the wings (Vinayak *et al.*, 2012). To perform any meaningful analysis on these voluminous and complex data therefore requires scaling up the hardware/software platforms. Multicore CPU is one of the early attempts developed to solve big data challenges.

Multicore allows a single machine has dozens of processing cores (Bekkerman *et al.*, 2012). These type of machine usually shared memory but only one disk. Multicore machines have gained internal parallelism over the past few years and more recently, the number of core per chip together with the number of operations a core can perform has increased significantly (Dilpreet and Chandan, 2014). Until the last few years, these hardware platforms together with their multithread operating systems are mainly responsible for accelerating the algorithms for big data analytics. This platform allows task to be broken down into threads with each thread executed in parallel on different CPU cores (Dilpreet and Chandan, 2014). Most of the programming languages in use also provided libraries to create threads and use CPU parallelism. The most popular among these programming languages is the Java language because of its existence for several years. A large number of software applications and programming environments are well developed for this platform (Dilpreet and Chandan, 2014).

Though Multicore CPUs is one of the earliest approaches to solving problem of massive data, its major drawback is in the limited number of processing cores and the primary dependence on system memory for data access. System memory will always be limited to a few of hundred gigabytes and this limitation affects the size of data that a CPU can process efficiently (Dilpreet and Chandan, 2014). At any point a data size exceeds system memory, access to disk becomes a big problem and even if data fits into a system memory, CPU core will attempt to process data at a much faster rate than the RAM speed hence; creating memory access bottleneck.

The development of CPUs however, is not at same pace with Graphic Processing Units (GPUs) (Dilpreet and Chandan, 2014). The number of cores per CPU is in double digits with processing power close to 10Gflops when compared with a single GPU which has more than 2500 processing cores with 1000Tflops of processing power (Dilpreet and Chandan, 2014). This massive parallelism in GPU makes it more appealing option for parallel computing applications. GPUs which were primarily intended for graphical operations such as video, image editing, and accelerated graphic processing became a general-purpose computing on graphic processing units (GPGPU). Recently, Nvidia launched Tesla series of GPUs which are specially designed for high performance computing. Nvidia has released Compute Unified Device Architecture (CUDA) framework that allows GPU programming accessible to all programmers without delving into the hardware details (Dilpreet and Chandan, 2014). GPU architecture usually has two levels of parallelism. At first level, there are several multiprocessors (MPs) and within each of these multiprocessors are several streaming processors (SPs). GPU programs are normally broken down into threads which are executed on SPs and these threads are grouped together to form thread blocks that run on a multiprocessor (Dilpreet and Chandan, 2014). Communication takes place between threads within a block. Each thread has access to small but extremely fast shared cache memory and larger global main memory. Threads from different block cannot communicate each other as they may be scheduled at different times. This implies that, any job to be run on GPU has to be broken down into blocks of computation that can run independently without communicating with each other (Hong and Kim, 2009). GPUs have been used in the development of faster machine learning algorithms. GPUminer is a good library used to implement machine learning algorithms on GPU architecture using CUDA framework (Dilpreet and Chandan, 2014).

Graphic Processing Unit (GPU) also has a major drawback of limited memory. Even with 12GB memory per GPU, which is the current architecture (Dilpreet and Chandan, 2014); it cannot handle terabyte scale data. At any point data exceeds this size; performance automatically decreases significantly as the disk access becomes a bottleneck. Also, there are limited number of software and algorithms available for GPUs (Dilpreet and Chandan, 2014). This is due to the way task is broken down in GPU hence, not many existing analytical algorithms are easily portable to GPUs.

Peer-to-peer network was another means intended by researchers to break through the challenges of big data (Steinmetz and Wehrles, 2005). Peer – to – peer network involves the connection of millions of machines in a network (Milojicic *et al.*, 2003). It is a decentralized and distributed network framework where nodes serve as consume resources for big data analytics. It is one of the oldest distributed computing platforms which uses Message Passing Interface (MPI) as its communication protocol to communicate between peers (Dilpreet and Chandan, 2014). Each node in the network can store data instance and the network can scale out to millions of nodes. One main feature of MPI which is the standard communication paradigm used in this network is its state processing process. Processes can live so long the system does not shutdown and there is no need to read same data again and again as in the case of other frameworks like MapReduce (Dilpreet and Chandan, 2014). All parameters in a P2P network are preserved locally which makes it suitable for iterative processing (Seivert and Casanova, 2004). MPI is also a master/slave paradigm (Dilpreet and Chandan, 2014). When deployed as a master – slave model, the slave machine can become the master for other processes. This feature is extremely useful for dynamic resource allocation where the slaves have large amounts of data to process (Dilpreet and Chandan, 2014). Although MPI in Peer – to – Peer network seems to be suitable for developing algorithms for big data analytics, its primary drawback is its inability to handle faults in a network. MPI has no mechanism to handle fault hence, when used on top of a P2P network which is itself completely unreliable hardware; a node failure may cause the whole system to shut down. With newer frameworks like Hadoop, MPI is no longer widely used.

Multicore CPU, GPUs and P2P network together with databases like Sybase, Informix, oracle and ingress where however, early attempts to solve big data challenges. Things took different turn with dynamic computations over ever-larger amounts of data becoming a necessity. SQL became inapt for the challenge, unstructured data needed to be analysed also to gain data science driven insights. This time gave birth to real ‘big data’ era with the introduction of Google File System and MapReduce by Google and subsequent release of Classic Hadoop by Yahoo.

Apart from Yahoo, other systems have recognized limitations in the classic Hadoop architecture and have provided alternative models to these limitations. Some of the efforts which closely resemble YARN are COSMOS, Mesos, Corona and Omega for

Microsoft, Twitter, Facebook and Google respectively (Chaiké *et al.*, 2008; Hindmand *et al.*, 2011; Facebook, 2012; Schwarzkopf *et al.*, 2013). Though these systems share a common inspiration of high-level goal of improving scalability, latency and programming model flexibility, they all have their architectural differences. These differences are most times in diverse design priorities and historical contexts. For instance, Mesos architectural design aim at providing a scalable and resilient core so that various frameworks can efficiently share clusters (Benjamin *et al.*, 2012). Due to the emergence of diverse frameworks, Mesos design philosophy is to provide a minimal interface that will enable efficient resource sharing across frameworks so that, control about task scheduling and execution can be pushed to the frameworks (Benjamin *et al.*, 2012). By pushing control to the frameworks, Mesos will allow each framework to implement its approach to solving problems in the cluster (for instance, achieving data locality and dealing with fault). Secondly, this approach will keep the design of Mesos simple and will help minimize the rate of changes required of the system. This design philosophy therefore, is necessary to make Mesos scalable and robust. Figure 2.15 shows the main components of Mesos.

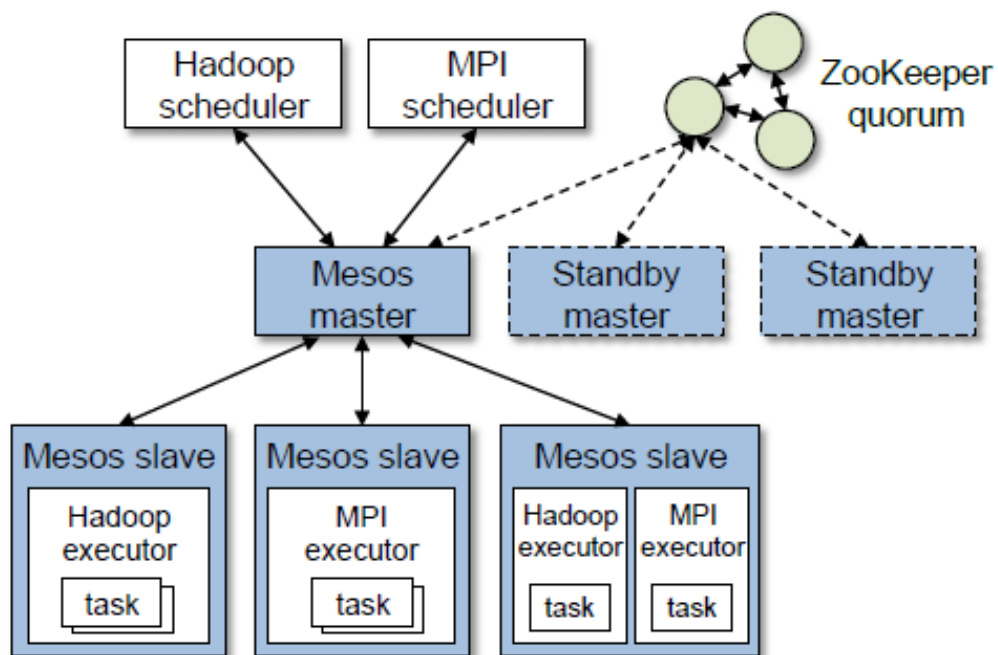


Figure 2.15: Mesos Architectural Design (Benjamin *et al.*, 2012)

Mesos has a master process responsible for managing slave daemons running on each cluster node, and also has frameworks that run tasks on these slaves (Benjamin *et al.*,

2012). The master daemon uses resource offers to achieve fine-grained sharing across frameworks. Resource offer is a list of free resources available on multiple slave nodes. Based on an organizational policy like fair sharing or priority, the master daemon in Mesos decides how many resources to be allocated to each framework. Each framework that will run on Mesos has a scheduler that registers the master to be offered resources, and has an executor process that launches slave nodes to run the framework's tasks (Benjamin *et al.*, 2012). The scheduler in each framework selects which offered resources to use. Once a framework accepts an offer, it passes Mesos a description of the tasks it wants to launch on them. Figure 2.16 gives an illustration of how framework gets scheduled to run tasks.

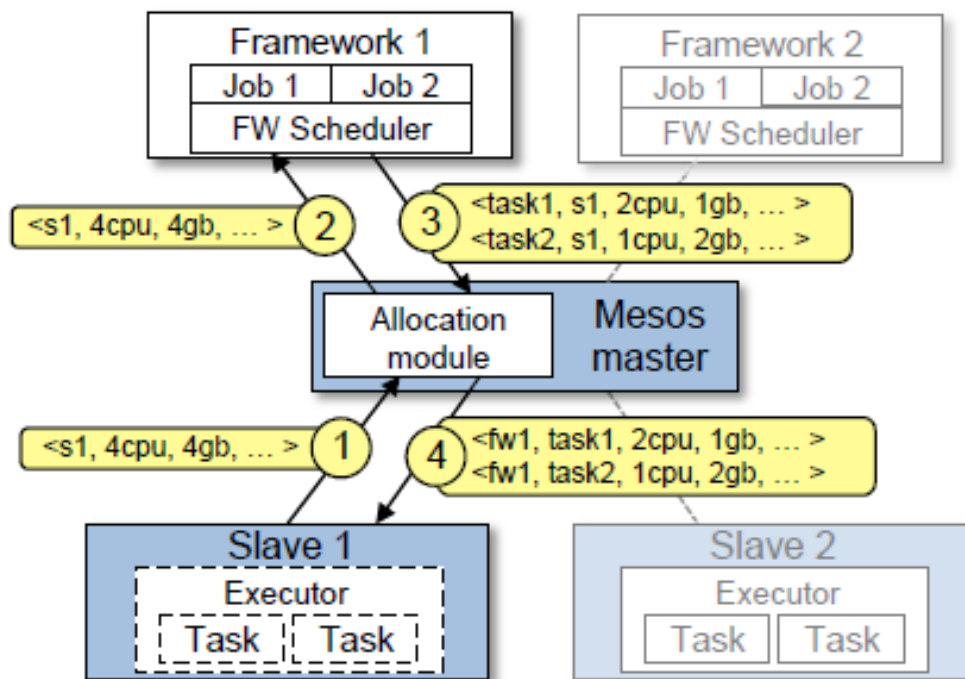


Figure 2.16: Resource offer in Mesos (Benjamin *et al.*, 2012)

In the first step, you notice Slave\_1 reports to the Master that it has 4GB memory and 4CPUs. The Master daemon in Mesos now invokes the Allocation\_Module which in turn informs framework that it has been offered all available resources. In the second step, the Master sends a resource offer describing these resources to the framework. The framework's scheduler in step 3 then replies the Master daemon with information about 2 tasks to be run on the slave node (2CPUs, 1GB RAM) and (1CPU, 2GB RAM) for task 1 and task 2 respectively. In the final step, the Master sends the tasks to the slave



which allocates appropriate resources to the framework's executor (Benjamin *et al.*, 2012).

Though the MapReduce frameworks of both Mesos and YARN have schedulers at two levels, they also have their significant differences. Mesos' architectural design implements an offered-based resource manager while YARN has a request-based resource manager (Hindman *et al.*, 2011). Mesos leverages a pool of central schedulers just like the type obtained in classic Hadoop but, YARN uses a per-job intra-framework scheduler which allows AM to request for resources depending on the criteria which includes location, CPU and memory demand. Allocation of resources in YARN are late binding, where application framework is obligated to use the resources provided by the container but does not have to apply them to a logical task on request. This framework/application decides which task to run with these resources. This is achieved through its own internal, second level scheduler.

Microsoft developed a computing platform called Collaborative Online Social Media Observatory (COSMOS) for storing and analysing massive data sets. The design philosophy aimed at running large clusters consisting of thousands of commodity servers (Chaiké *et al.*, 2008). Disk storage in this framework is distributed so that each server has one or more direct-attached disks. The main objectives behind COSMOS platform are;

- i. Availability:- To avoid whole system outages, Cosmos platform is resilient to multiple hardware failures. It does this by replicating data many times throughout the system with file metadata being managed by a quorum group of  $2f + 1$  servers so as to tolerate  $f$  failures (Chaiké *et al.*, 2008).
- ii. Reliability:- This framework is design in a way that it recognizes transient hardware conditions so as to avoid corrupting the system. A periodic scrub to detect corrupt or bit rot data is performed on disk data before it is used. Also, system components are check-summed end-to-end and a mechanism is applied to crash faulty components (Chaiké *et al.*, 2008).
- iii. Scalability:- Cosmos is capable of storing and processing petabytes of data and can accommodate more servers to the cluster without impacting performance.

- iv. Performance:- This framework is capable of running thousands of individual servers with data distributed among these servers. Each job is broken down into small units of computation and distributed across a large number of CPUs and storage devices.
- v. Cost:- It is cheaper to build, operate and expand than the traditional approaches.

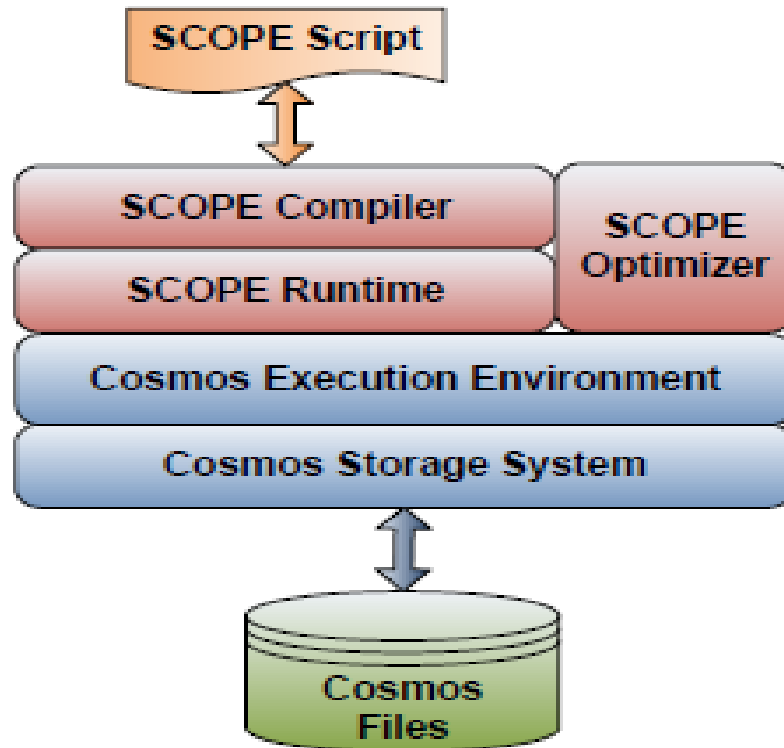


Figure 2.17: COSMOS Software Layers (Chaike *et al.*, 2008)

Cosmos platform has three basic components as described in Figure 2.17; COSMOS storage, COSMOS execution environment and SCOPE. COSMOS storage system is an append-only file designed to reliably and efficiently store extremely large sequential files (Chaike *et al.*, 2008). The platform is optimized for large sequential I/O. All writes are appended only with data distributed and replicated for fault tolerance. Data are also compressed to save storage and increase I/O throughput (Chaike *et al.*, 2008). COSMOS store is designed such that it provides a hierarchical namespace that stores sequential file of unlimited size. A file is composed of sequence of extents. Extent is a unit of space allocation which are most times in few hundred megabytes in size (Chaike *et al.*, 2008). A data within any extent consist of a sequence of append blocks with

block boundaries defined by application appends. Append blocks are also few megabytes in size and it contains a collection of application-define records stored in a compressed form with compression and de-compression done transparently at the client side (Chaike *et al.*, 2008). The second component of COSMOS platform is the COSMOS Execution Environment. The lowest level of this environment provides only the ability to run arbitrary executable code on a server (Chaike *et al.*, 2008). Through the execution protocol of this platform, clients can upload application code and resources onto the system. A recipient server then assigns the task a priority and executes it at an appropriate time. Building an efficient and fault tolerant application at this lowest level is difficult, error prone, time consuming and tedious (Chaike *et al.*, 2008). In COSMOS therefore, applications are programmed against execution engine that provides high-level program interface and a runtime that handles optimization details, data partitioning, parallelism, fault tolerance and resource management (Chaike *et al.*, 2008). Job Manager (JM) is the runtime component of the execution engine. It is the central and coordinating process for all processing vertices with the application.

Structured Computations Optimized for Parallel Execution (SCOPE) is the third component of COSMOS platform. It is a high-level scripting language for writing data analysis jobs in COSMOS. SCOPE compiler and optimizer help translate scripts to efficient parallel execution plans. It resembles SQL but has the expression of C-sharp. This design choice has several advantages. Its SQL resemblance reduces the learning curve for users and eases porting of existing SQL scripts into SCOPE. Its C-sharp expression can use C-sharp library hence; customer C-sharp classes can compute functions off scalar values or manipulate whole row sets (Chaike *et al.*, 2008).

COSMOS architectural framework closely resembles that of YARN in terms of storage and computer layers, their differences lies in the use of centralized resource manager. Though this framework have no central resource manager, its architectural design seems to be used for a single application type; SCOPE. If our desire is for a narrower target, COSMOS can leverage many optimizations such as native compression, indexed files and co-location partitions of datasets to speed up SCOPE.

Corona as earlier discussed, separates cluster resource management from job scheduling. It introduces a cluster manager whose sole responsibility is to track nodes in the cluster and to track the amount of free resources (Shouvik and Daniel, 2013). It has

a dedicated JobTracker for each job and can run either in the same process as the client (for small jobs) or as a separate process in the cluster (for large jobs). With Corona, Facebook measured some improvements over classic Hadoop. Some of these metrics includes average time to refill slots (Shouvik and Daniel, 2013). This metric gauges the amount of time a map/reduce slot remains idle on a TaskTracker. There was improvement in Corona compared to similar period for MapReduce. Cluster utilization was another improvement in Corona over Classic Hadoop. Cluster utilization improved along with refill-slot metric. Classic Hadoop system was also found to be unfair in its allocation and a dramatic improvement was seen with Corona's resource scheduling fairness (Shouvik and Daniel, 2013).

The approach used by Corona is a push based approach, which is different to the heartbeat based control-plane framework approach in YARN and other frameworks. Though latency/scalability tradeoff of these two frameworks deserves a detailed comparison, heartbeat communication protocol negotiates and monitors the availability of a resource in a cluster. It is intended to indicate the health of a machine hence; consideration between overload in YARN due to constant heartbeat between resource manager and other components and efficient fault tolerance in Corona since it is push based will have to look at.

Omega framework uses a shared state approach for its scheduling process, where each scheduler is granted full access to the entire cluster so that they can compete in a free-for-all manner, use opportunistic concurrency control to mediate clashes when they update the cluster state (Malte *et al.*, 2013). Omega has no central resource collector; all of its allocation decisions take place in the scheduler. Omega maintains a resilient master copy of the resource allocation in a cluster called cell state. Each scheduler in this framework has a private, local, frequently-updated copy of cell state for the purpose of scheduling decisions (Malte *et al.*, 2013). Each scheduler sees the entire state of the cell and can lay claim to any available resources in the cluster provided it has permission and priority to do so. Any time a scheduler makes a placement decision, update operation is done in the shared copy cell state in an atomic commit (Malte *et al.*, 2013). It is expected that one of the commit succeed in the case of conflict. But, whether or not the transaction succeeds, the scheduler will re-sync its local copy of cell state afterwards and if necessary, re-runs its scheduling algorithm and tries again (Malte *et al.*, 2013).

Omega schedulers operate completely in parallel; it does not have to wait for jobs in other scheduler and no inter-scheduler head of line blocking. To guard against any conflict causing starvation in the cluster, the schedulers uses incremental transactions. The scheduler uses all-or-nothing transaction to achieve gang scheduling where either all tasks of a job are scheduled together or none of the tasks is scheduled. If none of the task is scheduled, the scheduler must try to schedule the entire job again (Malte *et al.*, 2013). This process helps to avoid resource hoarding since a gang-scheduled job can pre-empt lower-priority tasks once sufficient resources are available and its transaction commits. In Omega framework, different schedulers can implement different policies but they must agree on what resource allocations are permitted, like a notion of whether a machine is full, common scale for expressing the relative importance of enforcement engine for high-level cluster-wide goals. The framework relies on the emergent behaviours that result from decisions of individual scheduler hence; fairness is not the sole concern of this framework but the need to meet business requirements (Malte *et al.*, 2013).

High Performance Cluster Computing platform is another distributed and parallel data processing system for big data. It was developed in 2000 by LexisNexis Risk Division and released as an open-source project in 2011 (Michael *et al.*, 2014). It is a data-intensive computing system platform created initially to reply to business needs of storing large volume of data (Camille, 2015). When HPCC system became fully operational, LexisNexis wanted to market it but Hadoop has already been implanted and widely used by companies (Patrick, 2011). Contrast to Hadoop platform, HPCC system is programmed in C++ language and executes natively on top of operating system leading to more predictable latencies and faster execution (Patrick, 2011). Enterprise control language is specifically designed for data management and query processing in HPCC system. The language is optimized for data-intensive operations, declarative, non-procedural and data flow oriented tasks. It uses syntax that is modular, reusable, extensible and highly productive (Camille, 2015).

HPCC system has two basic components; Thor Data Refinery Cluster and Roxy Rapid Data Delivery Cluster. Thor is a back-end batch oriented data workflow processing and analytics system equivalent to MapReduce in Hadoop framework (Camille, 2015). This component analyses and indexes huge amount of data. It uses a distributed file system called Thor DFS with parallel processing capability credit to its master/slave node

system (Camille, 2015). Roxie component of HPCC system is a front-end real-time data processing and analytics system. The component allows for real-time and analytics of data through parameterized Enterprise Control Language (ECL) queries (Camille, 2015). Roxie is like Hbase and Hive in Hadoop ecosystem (Anthony, 2015). It works with key/value store and is multi-threaded. It implements a master/slave node system like Thor. Roxie employs a distributed indexed-based file system called Roxie DFS with an index file to store file locations. The server to store file location and job scheduler in this component is Deli server, which function as system data store for job work unit information and provides naming services for distributed file systems (Michael *et al.*, 2014). With the introduction of Thor DFS, cluster can now scale from single node to thousands of nodes (Camille, 2015). However, neither data locality nor elastic scheduling needs of map and reduce phases were expressible with this framework. Perhaps the reason is that the framework was originally created to support MPI style and HPC application model and to run coarse-grained non-elastic workloads.

Disco is another attempt towards storage and processing of massive data. It was developed by Ville Tuulos in 2008 as a project of the Nokia Research Centre (Camille, 2015). The framework can distribute and replicate data, and also schedules jobs. It has tools that can index billion of data points and query them real-time (Camille, 2015). The framework can also analyse chunks of data in parallel and will collect intermediate results into a final result with MapReduce paradigm created by Google (Camille, 2015). Just like the Hadoop framework, Disco system is also based on a node system with master/slave architecture as described in Figure 2.18.

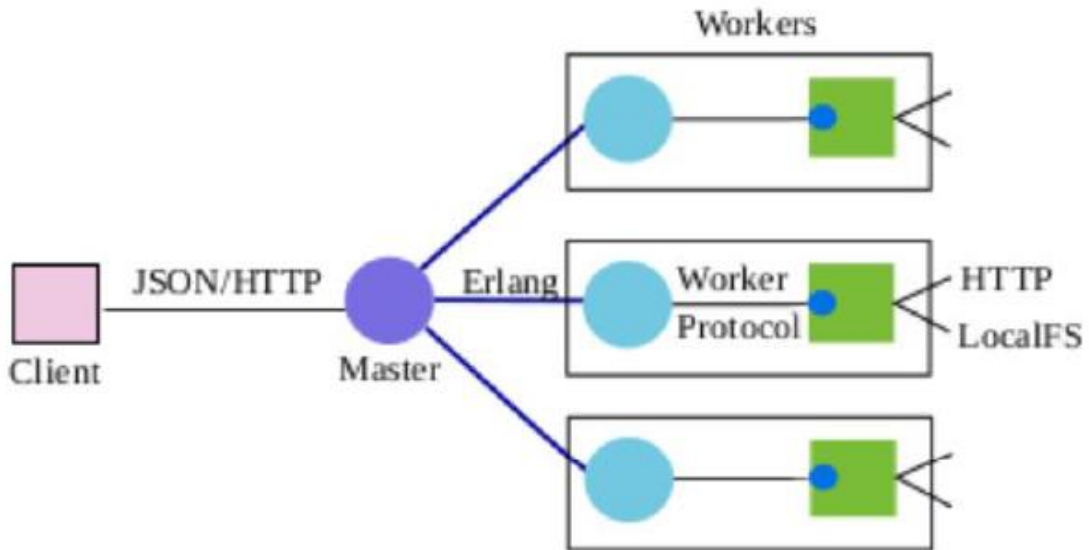


Figure 2.18: Disco Architecture (Prashanth *et al.*, 2011)

Disco core is written in Erlang (a functional language that allows fault tolerance for distributed applications) and it is the first big data framework that implements Python language jobs (Prashanth *et al.*, 2011). This framework was designed to process larger applications like web services in a way that tasks can be delegated to a cluster independently from the core application. It has a Python API called `disco.ddfs` with two components; REST-style Web API which helps in job control and Easy-to-use Web Interface that is used for status monitoring (Prashanth *et al.*, 2011). The worker protocol in Disco makes it easier to accept jobs written in other languages. It has a file system called Disco Distributed File System (DDFS). This file system adapt to MapReduce architecture and allow storage and processing of massive data such as structured data (worksheet and databases SQL) and unstructured data (text, documents, log of applications, sensors, pictures and videos).

DDFS has a special tool called tag-based file system which tags different files (Camille, 2015). For example, tags can be used to timestamp different versions of data or to denote who owns a data or from which source it came from. This way, DDFS provides flexible means to manage terabytes of data (Prashanth *et al.*, 2011). This file system is also schema-free which can be used to store arbitrary data provided the data are not fewer than 4KB or very often updated (Prashanth *et al.*, 2011). It is horizontally scalable with ability to add new nodes through Disco web interface. Disco framework

has DiscoDB as its own database system and Discodex which is a web front-end that allows indexing of data originating from MapReduce operations (Camille, 2015).

Spark is an in-memory distributed data analysis platform (Telmoda, 2015). The primary aim of this framework is to speed up batch analysis jobs, iterative machine learning jobs, interactive query and graph processing tasks (Telmoda, 2015). It is a next generation paradigm for big data processing developed by researchers at University of California, Berkeley (Anthony, 2015). Spark was designed as an alternative to Hadoop to help overcome disk I/O limitations and to improve performance. The framework allows data to be cached in memory which helps reduce disk overhead in Hadoop for iterative tasks (Anthony, 2015). Spark uses Resilient Distributed Datasets (RDDs) which are great for pipeline parallel operators for computation and are also immutable, allowing for fault tolerance based on lineage information (Telmoda, 2015). Spark supports a rich set of higher-level tools as shown in Figure 2.19. These tools include Shark SQL for SQL and structured data processing, Spark Streaming for development of parallel applications, MLib for machine learning and GraphX for graph processing (Apache, 2016).

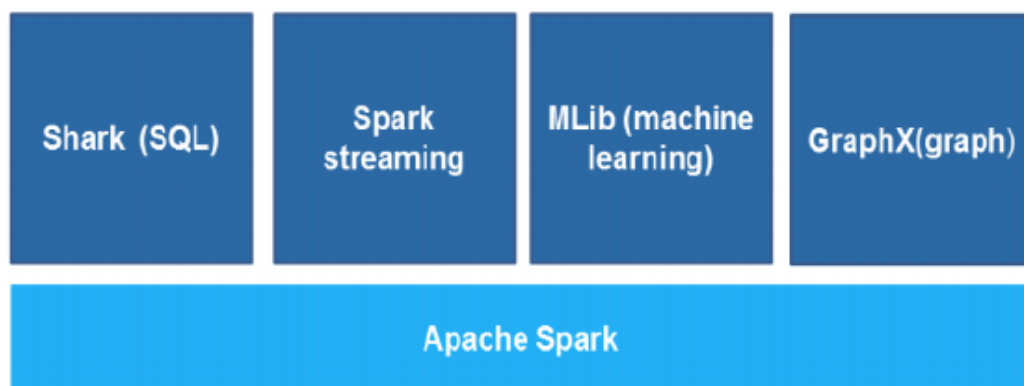


Figure 2.19: Spark Framework (Telmoda, 2015)

The main abstraction of this framework is its resilient distributed datasets which are immutable, partitioned collections that can be created through various data-parallel operators. Each of these RDDs can be a collection stored in an external storage system like HDFS or a derived dataset which is created through the application of operators to other RDDs. There are three options for persist RDDs; in-memory storage serialized data (with limited space), in-memory storage as de-serialized Java objects (fastest, JVM can access RDD natively) or on-disk storage (RDD too large to keep in memory and its



costly to recompute). The main goal of the framework is to treat streaming computations as a series of deterministic batch computations on small time interval (Telmoda, 2015). Any input data received at each interval of computation is reliably stored across clusters to form dataset for that interval. Once the time slice completes, the dataset is processed through deterministic parallel operation like map and reduce operations. The operation produces new datasets representing either program outputs or intermediate state and these outputs are stored in RDDs (Zaharia *et al.*, 2012).

Storm is a free and open source distributed real-time computation system for big data. The framework focuses on stream processing or better put, complex event processing (Telmoda, 2015). It uses a fault tolerant method to perform pipeline multiple computations on an event as it flows into a system (Apache, 2016). It can be used to transform unstructured data, as it flows into a system (into desired format). Just like classic Hadoop is known for batch processing, Storm reliably processes unbounded streams of data in distributed real-time computation. The framework has many use cases like online machine learning, real – time analytics, ETL and distributed RPC (Telmoda, 2015).

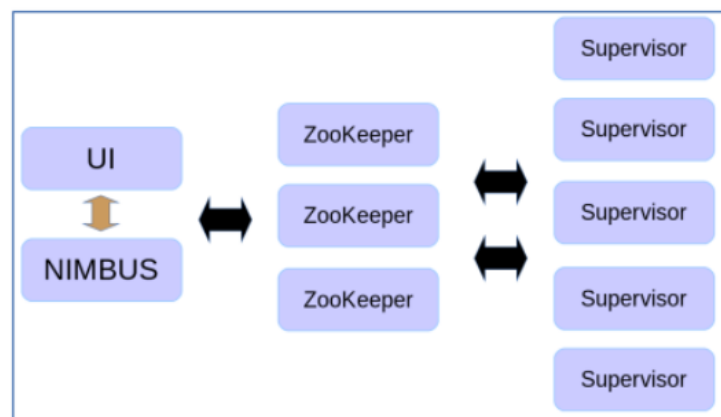


Figure 2.20: Storm framework system architecture (Telmoda, 2015)

Figure 2.20 shows Storm system architecture with Nimbus (just like JobTracker in Hadoop), Supervisor (manages worker nodes), Zookeeper (stores metadata) and UI (for Web-UI). Two type of nodes exist in Storm cluster; master node which runs the daemon ‘Nimbus’ and is responsible for distributing code around the cluster, assigning tasks to worker nodes and monitoring of failures (Apache, 2016). The slave (worker) node runs the daemon ‘Supervisor’ which receives work assigned to it, start and stops worker processes as instructed by Nimbus (Apache, 2016).

Apache Giraph is also a big data tool running on top of Hadoop framework (Bakshi and Sonali, 2016). It is an open source version of Google Pregel most suitable for large scale graph processing. Examples of these graph processing are analysis of interconnected web (for page ranking) or social media (like facebook and twitter) interaction that are only graph of interconnected vertices either by web page to another through the edge (hyperlink) or users connected to each other through edges representing friendship or some kind of fans or business (Bakshi and Sonali, 2016).

A framework close to YARN architecture is the data processing stack designed by Spark developers called Berkeley Data Analytics Stack (BDAS) shown in Figure 2.21. This stack has Tachyon at its lowest level which is based on HDFS (Dilpreet and Chandan, 2014). It is fault tolerant and enables file sharing at memory-speed (data I/O speed comparable to system memory) across a cluster (Dilpreet and Chandan, 2014).

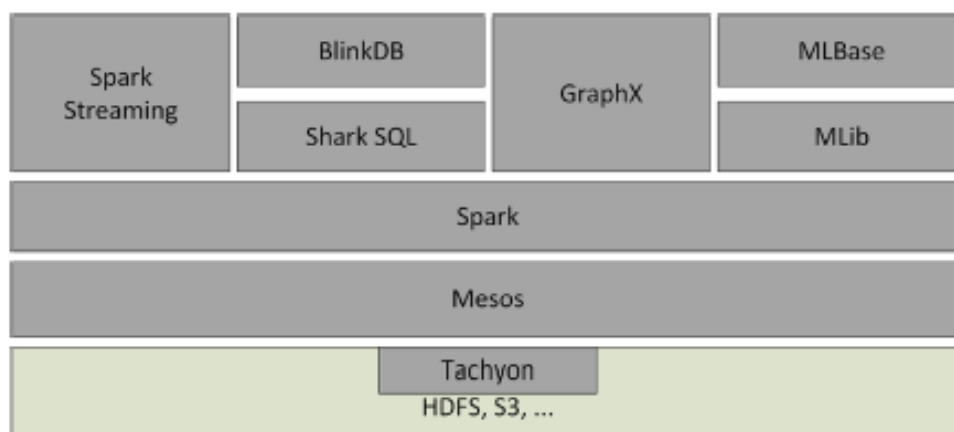


Figure 2.21: An illustration of Berkeley Data Analysis Stack and its various components (Dilpreet and Chandan, 2014)

The second layer of this stack has a component called Mesos which is a cluster manager that provides efficient resource isolation and sharing across distributed applications/frameworks (Dilpreet and Chandan, 2014). This component also supports Hadoop, Spark, Aurora and other applications on a dynamically shared pool of resources with scalability of tens of thousands nodes (Anthony, 2015). Its API is available in Java, Python and C++ and it has multi-resource scheduling capabilities. The third layer has Spark that takes the place of Hadoop MapReduce and on top of this layer are Spark wrappers; Stack Streaming, MLib, Stack SQL, GraphX, BlinkDB for queries with bounded errors and bounded response time on very large data, MLbase for distributed machine learning library based on Spark (Kraska *et al.*, 2013).

Cloudera Impala is another big data analytics that helps enterprise exploit benefits of SQL tools in achieving real-time analytics potentials when working with massive data that are either structured or unstructured (Kornacker *et al.*, 2015). This framework can be used by business analyst and IT experts over a range of supported data types and large volume of data to interact in real time with a HBase or a HDFS data store for the sake of analytics (Bakshi and Sonali, 2016). Interestingly, Cloudera Impala can be integrated into Hadoop stack. Figure 2.22 shows Cloudera Impala’s position in Hadoop stack.

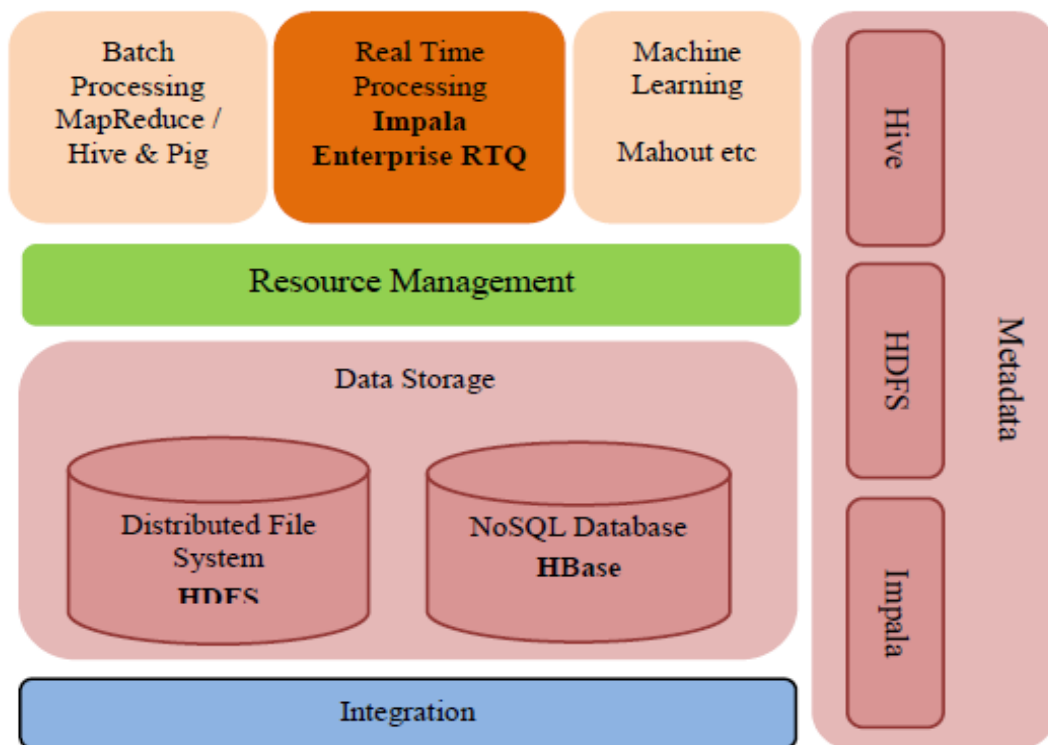


Figure 2.22: Cloudera Impala status in Hadoop Stack (Bakshi and Sonali, 2016)

This framework has flexible data model, support real-time interaction through Cloudera Enterprise RTQ (reduces response time of queries to seconds unlike HiveQL or Map Reduce), offers effective security measure through Kerberos authentication support (Bakshi and Sonali, 2016).

Pentaho is an open source business intelligence framework which provides range of tools that can help customers manage their business better (Pasula, 2016). These tools include mining tools, online analytic processing options, dashboard applications and data integration tools. Pentaho is a multi-purpose business intelligence platform helping

enterprise in analysing, integrating and presenting data through comprehensive report and dashboards. Business analytics now rely on Pentaho to identify barriers that block company's ability to extract value from data (Pasula, 2016). It was initially developed as a report generating engine but branch into big data analytic tool to help enterprise have insight into their business (Vidhya *et al.*, 2014). It was integrated with most NoSQL databases like Cassandra and MongoDB. This tool can fetch IT and business users together through classic sorting and sifting tables with firmly coupling data integration thereby permitting both IT and business users access, build, virtualize and analyse data that makes an impact on business results (Vidhya *et al.*, 2014). Pentaho helps in reducing plan time and complexity needed to acquire and deploy big data analytics thus helping companies to know business value of a large bit of diverse data (Vidhya *et al.*, 2014). It does this through the execution of data and preparation of big data and traditional data types in any infrastructure, along with the range of analytics from data dictionary to production analytics (Vidhya *et al.*, 2014). Pentaho complements Greenplum distribution of Hadoop which provides an end to end data integration and business intelligence suite that enables easy to use, graphical environment for managing data movement in and out of Hadoop. The integration of Pentaho into Hadoop framework therefore, makes Hadoop a more robust framework for big data analytics.

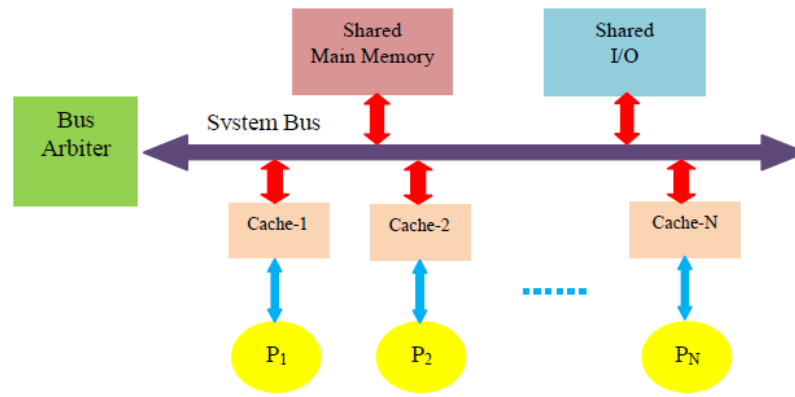
Jaspersoft is also an open source business intelligence platform suitable for better decision making with the help of highly interactive reports, analytics and dashboards. Though initially designed for small business, Jaspersoft is now moving into big data for huge businesses (Vidhya *et al.*, 2014). Jaspersoft server provides software that utilizes data from diverse storage platforms like MongoDB, Cassandra and Redis. To gain access to HBase, the Hive connector is provided by Jasper reports which is well represented by Hadoop (Vidhya *et al.*, 2014). Jaspersoft like Pentaho is an open source business intelligence platform. Though this platform has helped in providing big data analytics solutions in business, it is not an all-encompassing big data framework. In October 24<sup>th</sup> 2011 at San Francisco, Jaspersoft announces new Hadoop-based big data analytics solution (TIBC, 2011) making the platform part of Hadoop cluster. With this development, Hadoop is still a more robust big data framework.

Another big data platform is Splunk. Splunk is a log analysis platform. The platform can be used with other databases like SQL (Vidhya *et al.*, 2014), specifically used in

monitoring of ordered and unordered machine data. As a business intelligence tool, Splunk with stored data can visualize data (Siya, 2013). Splunk can define data emitted by machines in great volumes (Vidhya *et al.*, 2014). Splunk makes machine-generated data accessible, usable and valuable to users. It does this by organizing and extracting real-time insights from huge amounts of machine data provided from servers, sensors, websites, social media platforms and open source data stores. Once these data are in Splunk, the platform searches, monitors, reports and analyse the data without considering how unstructured, huge or diverse the data may be (Vidhya *et al.*, 2014). Splunk DB has a powerful connectivity for real-time connection between one or many relational databases. It is also used for bi-directional connectivity with Hadoop (Vidhya *et al.*, 2014).

Karmasphere is also a big data tool originally developed as a set of plug-ins for Eclipse (Vidhya *et al.*, 2014). It is a specialized IDE for creating and running Hadoop jobs easily. One major feature of Karmasphere is that, it shows test data at each step while setting up workflow, thus making users understand the outlook of temporary data as it is been analysed and reduced (Vidhya *et al.*, 2014). Karmasphere Analyst is a tool in Karmasphere developed to ease procedure of plotting through data in Hadoop cluster. Just like subroutine that uncompressed zipped log files, Karmasphere Analyst has features for creating good Hadoop jobs (Vidhya *et al.*, 2014).

IBM is not an exception in the development of big data frameworks. One of the big data frameworks develop by this company is IBM Netezza. Netezza can be seen as either a storage or computing framework. The reason being that, it provides both data warehouse as well as analytics appliance (Bakshi and Solani, 2016). This framework is a shared-nothing architecture premised on Asymmetric Massively Parallel Processing (AMPP). It is two-tier architecture as described in Figure 2.23a and 2.23b.



P: Independent Homogeneous Processor

Figure 2.23a: IBM Netezza Tier -1 (Francisco, 2011)

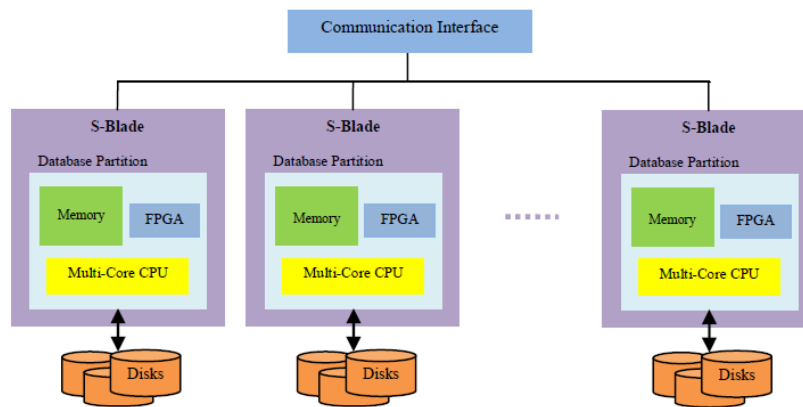


Figure 2.23b: IBM Netezza Tier -2 (Francisco, 2011)

This framework handles massive complex queries very quickly. The first tier as shown in Figure 2.23a employs a high performance Linux based symmetric multi-processing host which is responsible for compiling data query jobs so as to generate execution plans (Francisco, 2011). It does this by breaking down the original query into sub-task that is suitable for parallel execution. The suitable sub-tasks are distributed over the second tier for execution. As described in Figure 2.23b, the second tier contains hundreds of intelligent snippet processing blades called S-Blades that form the Massively Parallel Processing (MPP) engine of the framework (Bakshi and Solani, 2016).

Nephele – PACT framework is also a big data framework that offers Parallelization Contracts (PACTs) which are generalization of map and reduce primitive of the MapReduce framework (Sharanjit *et al.*, 2014). This framework helps handle complex

data flows in a cluster. Dissimilar with execution strategy obtained in MapReduce, the PACT compiler generates multiple execution plans from where it selects optimal one. These execution plans are evaluated and stored as direct acyclic graphs (DAG) where the vertices of DAG are instances of PACT, while the edges denote data transportation mechanism between the PACTs. This programming model centred on key/value pairs and Parallelization Contracts (PACTs). The PACTs are second-ordered functions that describe properties of the input and output of the associated first-ordered functions (Sharanjit *et al.*, 2014).

Microsoft Dryad is another high performance, distributed computing framework that supports writing and execution of data-centred parallel programs (Dongyao *et al.*, 2017). This framework allows programmers to use resources in a cluster to run data-parallel programs. It is possible to write simple programs which can be executed concurrently on thousands of machines while hiding the complexity of concurrency with this framework. The language used for execution is DryadLINQ (Yuan *et al.*, 2016). DryadLINQ is a sequential program which composed of LINQ expressions performing arbitrary side-effect-free transformations on datasets (Yuan *et al.*, 2016). This language translates data-parallel portions of a program into a distributed execution plan which is passed to the Dryad execution platform. Figure 2.24 describes Dryad system architecture.

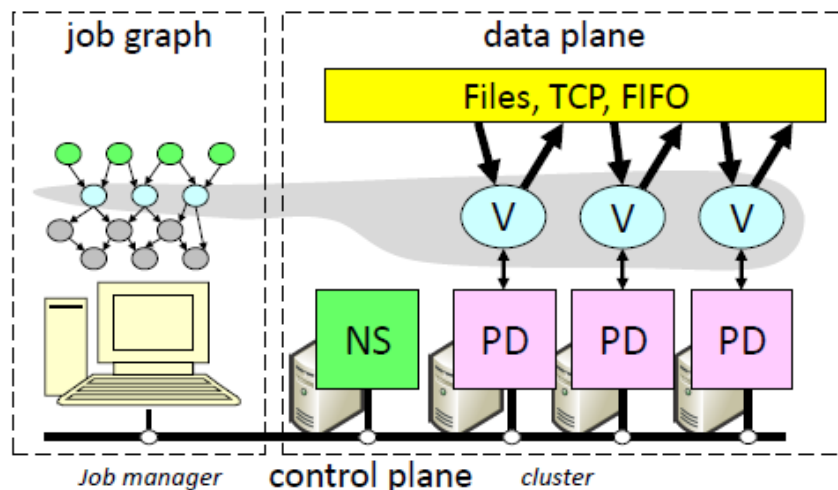


Figure 2.24: Dryad Architecture (Yuan *et al.*, 2016)

Dryad has a centralized job manager responsible for job execution. The job manager instantiate a job dataflow graph, schedules processes on cluster computer, provides fault

tolerance for failed or slow process by re-executing them, monitors job and collects statistics and also, transform job graph dynamically base on user's supplied policies. A task scheduler helps control the cluster by managing batch queue of jobs and executing them at a time subject to cluster policy (Yuan *et al.*, 2016).

Condor is another batch system framework presented by Tannenbaum (2010) for high throughput computing. This system is used extensively in High Energy Physics (HEP) community for management of computing tasks on dedicated compute farms. The scalability of Condor makes it an important consideration for administrators of large and expanding compute farms that depend on this framework to integrate large collections of computer across multiple institutions and end users with increasingly large workflows to process. HEP community has been a large driver in recent effort to advance the scale at which Condor can operate due to its scalability features (Bradly *et al.*, 2011). The architectural design of Condor has a pool defined by a daemon called *collector*. *Collector* serves as a registry for the rest of distributed daemon in Condor pool. Job execution nodes in the pool are represented by a daemon called *startd*; responsible for carrying out job execution requests. *Startd* helps divide machine into one or more logical sub-divisions called execution slots. A collection of jobs that have been submitted by users is maintained by another daemon called *schedd*. This daemon obtains a lease to run jobs on an execution slot. The lease is gotten from *negotiator*, which is also a daemon responsible for pool-wide user priority management and for matchmaking (finding compatible execution slots for resource request). *Schedd* has pool of jobs submitted by users and is also responsible for communicating with other daemons in Condor pool for running of jobs (Tannenbaum, 2010). Clients communicate with *Schedd* for job management and operations such as submitting, modifying, examining and aborting jobs. A Condor pool can have more than one *Schedd*, each running on a separate computer. For manageability and ease of use however, few number of *Schedds* are required (Tannenbaum, 2010).

It has been observed by users of Condor that, instantiating another *schedd* involves numerous steps which include the purchase of new hardware and configuring software that will interact with the daemon. In this framework, the status of every job is logged to files on disk so that workload managers can rely on the semantics of queued jobs (same job will not run in a multiple instances at the same time and once a job is submitted, it will not disappear from the queue without atleast one record in the job's



even log indicates that it is finished or cancelled). Because of this semantic guarantee about job logging, transactions requiring durability are synced to the disk. Syncing transactions to disk however can be quite expensive hence; in many situations, this has been found to be a limiting factor for scalability of *schedd*.

A dispersed cloud infrastructure that uses voluntary edge resources for computation and data storage was presented by Jonathan *et al.*, (2017). This system called “Nebula” described a light-weight architecture that allows distributed data intensive computing through a number of optimizations which include location-aware data and computation placement, replication and recovery (Jonathan *et al.*, 2017). The design goal of this framework is to support distributed data-intensive computing, location-aware resource management by enabling efficient execution of distributed data-intensive applications and sandboxed execution environment. The architectural design is shown in Figure 2.25.

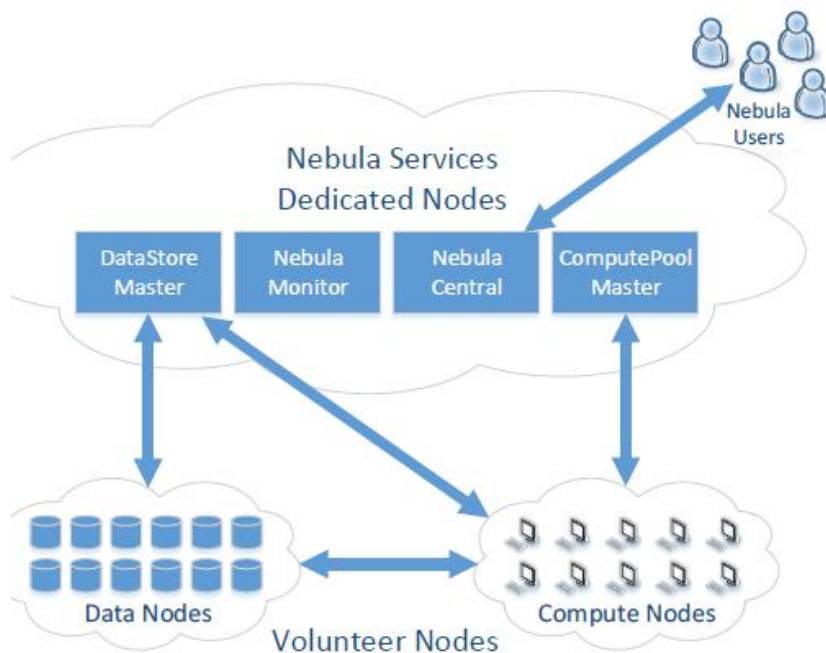


Figure 2.25: Nebula System architecture (Jonathan *et al.*, 2017)

The system consist of volunteer nodes that donates computation and storage resources along with a set of global and application specific services which are hosted on dedicated, stable nodes that has four major components; Nebula Central, Nebula Monitor, DataStore Master and ComputePool (Jonathan *et al.*, 2017). Nebula Central serves as the front-end daemon for Nebula ecosystem providing simple and easy-to-use

web-based portal so that volunteers can join the system and application writers can also inject applications into the system. DataStore component of this system is a simple per-application storage service which is used to support efficient and location-aware data processing in Nebula. Each DataStore has a volunteer node that stores actual data and a DataStore Master that keeps system metadata for data placement decisions (Jonathan *et al.*, 2017). ComputePool component of Nebula provides per-application computation resources with the help of volunteer compute nodes. ComputePool Master coordinates the execution of applications within the compute nodes. The compute nodes access and retrieve data with the help of DataStore and they are assigned tasks by ComputePool based on application specific computation requirements and data location. Nebula monitor monitors volunteer nodes and network characteristics. It checks node computation speeds, memory and storage capabilities, network bandwidth and also checks health information of each node and possible link failures (Jonathan *et al.*, 2017). Figure 2.26 describe the control and data flow and steps involved in executing a task on the Nebula framework.

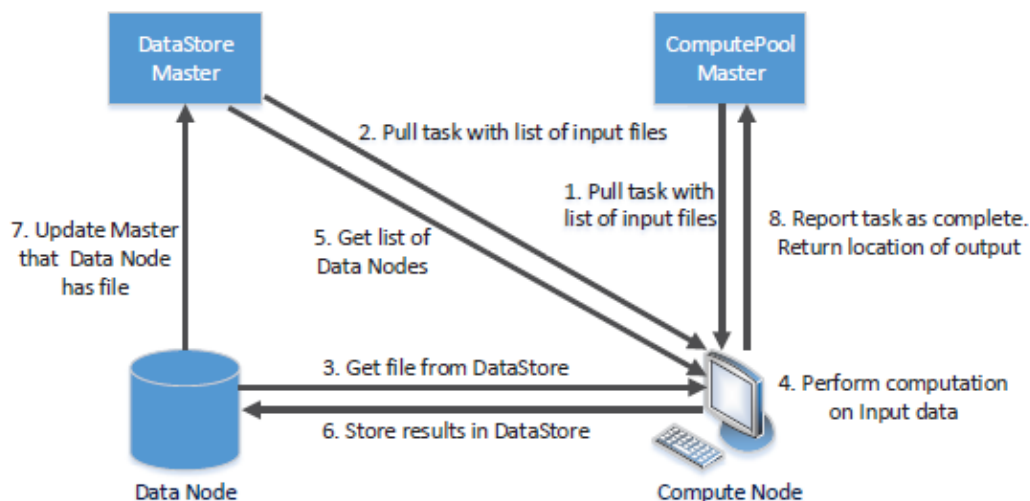


Figure 2.26: Control and data flow for job execution in Nebula system (Jonathan *et al.*, 2017)

To execute a task (application), the application will have to be injected into the system via Nebula Central and the input data placed within Nebula DataStore. The compute nodes contact ComputePool periodically and ask for tasks. Once a task is assigned to

compute node, the node will download the application code and the input data from the DataStore and then starts computation. At the end of computation, the outputs are uploaded back to the DataStore. Finally, bandwidths between DataStore and compute node together with location of output files are provided to the ComputePool Master (Jonathan *et al.*, 2017). A prototype running across edge volunteers on PlanetLab testbed was carried out with this system and an evaluation of MapReduce on Nebula was performed and compared against other edge-based volunteer systems. Nebula MapReduce significantly outperformed other edge-based volunteer systems (Jonathan *et al.*, 2017)

Outerhout *et al.*, (2013) proposed a distributed, low latency scheduling framework which demonstrates a decentralized, randomized sampling approach for near-optimal performance while avoiding throughput and availability limitations of a centralized design. Outerhout *et al.*, (2013) presented Sparrow; a stateless distributed scheduler that adapts the power of two choices load balancing technique to the domain of parallel task scheduling. The choices require scheduling each task by probing two random servers and placing task on server that is less busy or has fewer queued tasks (Outerhout *et al.*, 2013). Sparrow focused mainly on fine-grained task scheduling for low latency applications. The framework provides task scheduling which is complimentary to the functionality provided by cluster managers. Instead of launching new task, the framework assumes that a long running execution process is already running on each compute node for each framework hence; it only sends a short task description when a task is launched (Outerhout *et al.*, 2013). The framework makes approximations when scheduling tasks thereby trading off many of the complex features supported by sophisticated, centralized scheduler so as to provide higher scheduling throughput and lower latency. For instance, Sparrow does not allow certain types of placement constraints like “my job should not be run on machines where user A’s jobs are running”. It also does not allow bin packing and gang scheduling (Outerhout *et al.*, 2013). The framework however, supports basic constraints over job placement, such as per-task constraint (i.e. task needs to be co-resident with input data) and per-job constraints (i.e. task must be placed on machines with appropriate cores). This feature set in Sparrow is similar to the one in Hadoop and Spark scheduler.

This framework does not support gang scheduling typically implemented by bin packing algorithm which searches for a reserve time splits on which an entire job can be

run. Because Sparrow queue tasks on several machines, it lacks a central point from which to perform bin packing hence, deadlocks between multiple tasks that require gang scheduling may occur. Currently, this framework only supports FIFO order, adding other query-level scheduling policies may improve end-to-end query performance of the framework. It is also important that when a compute node fails, all schedulers with outstanding requests at that node be informed. A centralized state that relies on heartbeat protocol so as to maintain a list of nodes that are alive may be needed in this framework.

Zhao *et al.*, (2014) proposed FusionFS that has a distributed storage layer local to compute nodes. This layer allows for most I/O operations and saves extreme amounts of data movement between compute and storage resources (Zhao *et al.*, 2014). Zhao *et al.*, (2014) idea was to see how to colocate compute and storage nodes in High Performance Computing (HPC) to enable applications manipulate their intermediate results and checkpoints rather than transferring data over network. Zhao *et al.*, (2014) observed that in HPC systems, fault tolerance is achieved through some check pointing. The system will periodically flush memory to external persistent storage during check-pointing and will occasionally load same data back to memory so as to roll back to the most recent correct checkpoint after a failure. Doing this makes file writes outnumber file reads in terms of both frequency and size in HPC system. To improve write performance therefore, will significantly reduce overall I/O cost. FusionFS was designed to disperse metadata to all compute nodes so that, maximal concurrency of metadata operations can be achieved. Every client of FusionFS optimized write operations with local write, an approach that reduces network traffic and makes the aggregate I/O throughput highly scalable (Zhao *et al.*, 2014). FusionFS was deployed and evaluated on 16K compute nodes of IBM Blue Gene/P supercomputers, showing a significant improvement over other file system in HPC (Zhao *et al.*, 2014).

FusionFS was specifically designed to overcome bottleneck in file systems of HPC. HPC systems as discussed earlier are expensive systems with vertical scaling technique in focus. In this technique, systems have to be more powerful to handle future workloads. Initial addition of more processors, memory and faster hardware in an attempt to have better performance are mostly not fully utilized at the initial stage. Collocation of compute and storage nodes in FusionFS is a special feature in Hadoop framework that makes it stand out. Fault tolerance of data centres is also a unique

feature in Hadoop. Though fault tolerance is achieved through re-computing affected data chunks that are replicated on multiple nodes and not through check-pointing memory states, Hadoop technique is a better option since data are stored in in-expensive commodity servers.

Wang *et al.*, (2015) proposed a task execution framework called MATRIX to overcome Hadoop scaling limitations through distributed task execution. Though MATRIX was originally developed to schedule executions of data-intensive scientific applications of many-task computing on supercomputers, Wang *et al.*, (2015) saw the need to use same framework to address scalability issues of Hadoop through decentralizing the responsibility of resource manager. Figure 2.27 shows the architecture of MATRIX.

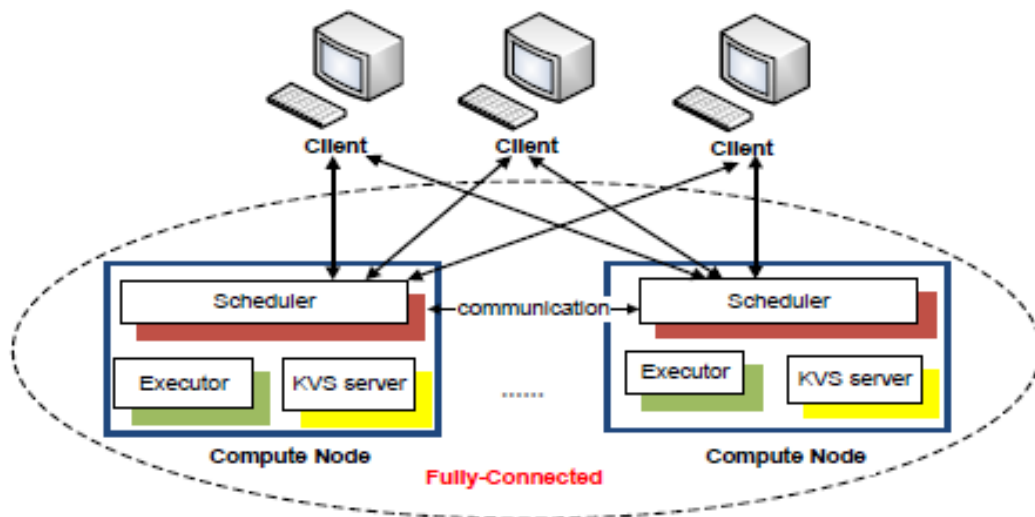


Figure 2.27: MATRIX Architecture (Wang *et al.*, 2015)

The framework is fully distributed by delegating one scheduler on each compute node (Wang *et al.*, 2015). For each compute node, there is an executor and a key-value store (KVS) server. The scheduler on each of these nodes has the responsibility of managing local resources for optimizing load balancing and data-locality. The executor is saddled with the responsibility of executing tasks while the KVS server keeps the metadata of tasks and data files in a scalable way (Wang *et al.*, 2015). Each scheduler in this framework has four task queues; waiting queue (WaitQ), dedicated ready queue (LReadyQ), shared ready queue (SReadyQ) and complete queue (CompleteQ). These queues store tasks in different states. For instance, WaitQ keeps tasks waiting for their parents to be processed, LReadyQ holds ready tasks whose majority of required data is

local (i.e. tasks that are executed locally). SReadyQ keeps ready tasks migrated from other compute node(s) while finished tasks are moved into the CompleteQ. Wang *et al.*, (2015) claimed that MATRIX outperformed YARN by 1.27x for typical workload (WordCount) and has the potential to enable Hadoop scale to extreme-scale data centers for fine-grained workloads (Wang *et al.*, 2015).

MATRIX was originally designed for scheduling fine-grained many-task data-intensive applications on supercomputers. An attempt to leverage MATRIX distributed design wisdoms to overcome Hadoop scaling limitations for arbitrary data processing applications is a good approach but not robust for frameworks needed for distributed and parallel processing. From the architecture of MATRIX, it is clear that the framework has a per-node resource manager (each scheduler maintains a local view of the resources on an individual node). For any framework to have a per-node RM, all data blocks for single file must be resident on that compute node.

Albert *et al.*, (2016) proposed a framework called Awan; a resource manager that helps share computing resources across multiple frameworks in an Edge Cloud environment (Albert *et al.*, 2016). The main goal of this system is to provide a general resource management mechanism that will allow each framework to schedule its job with high locality in a geo-distributed environment. To achieve this goal, Awan implements a resource lease abstraction to allocate resources to individual framework schedulers. These schedulers can in turn make better scheduling decisions by considering the availability of desirable local resources (Albert *et al.*, 2016). Awan has File Master, Node Manager, Resource Manager and Framework Scheduler as shown in Figure 2.28.

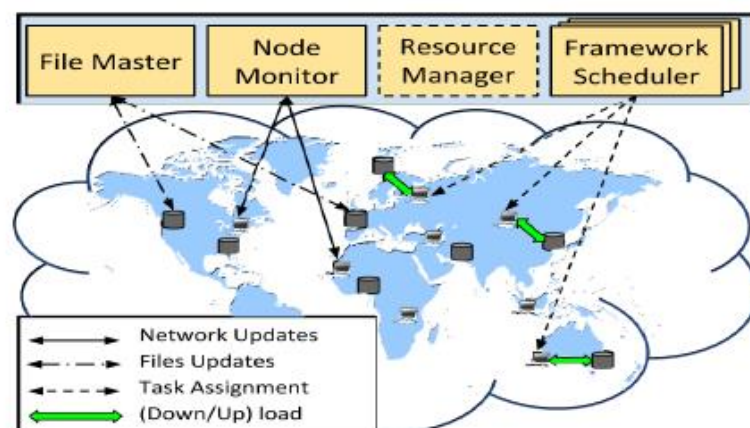


Figure 2.28: Component of Awan (Albert *et al.*, 2016)

The File Master manages all files stored in the system. It is responsible for maintaining file metadata, handling file replication and determining which storage nodes are responsible for storing specific files and its replicas (Albert *et al.*, 2016). The Node Manager monitors the health of each node and the network bandwidth (up-link and down-link) between nodes. The bandwidth monitoring helps file master during data placement decisions and framework scheduler during scheduling of task locally. The framework scheduler helps in task scheduling logic for specific computing framework. The scheduler will always attempt to schedule task with high locality since network bandwidth is a dominant factor in task running time. The Resource Manager provides a resource sharing service among framework schedulers. Resource Manager keeps record of live nodes by a communication protocol with the Node Manager (Albert *et al.*, 2016). To ensure that Awan does not have a centralized resource manager, the framework provides a two-level architecture as shown in Figure 2.29.

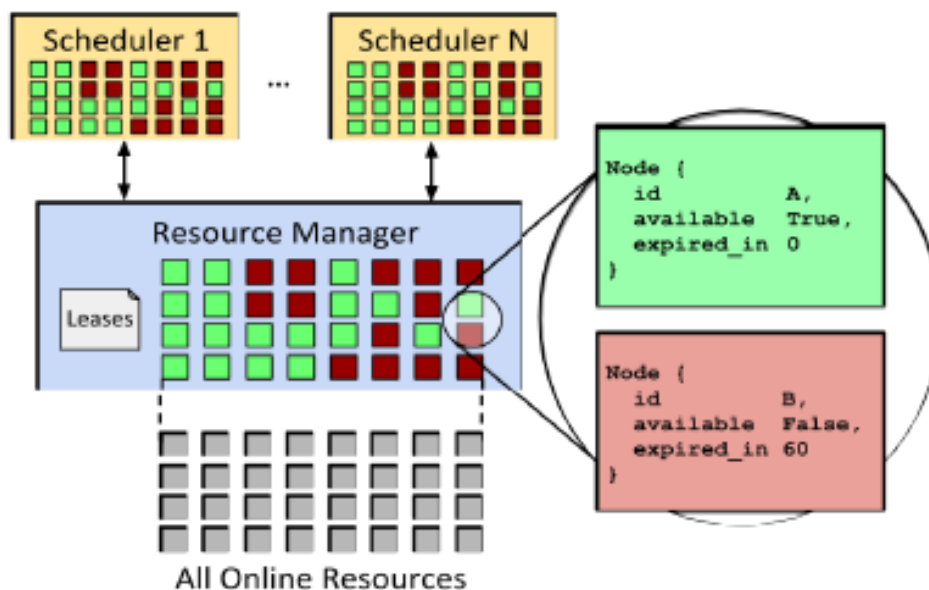


Figure 2.29: Two-level architecture of Awan (Albert *et al.*, 2016)

This two-level architecture incorporates shared-state architecture by sharing the states of all the resources to every framework scheduler. The Resource Manager in Awan provides the states of all resources instead of providing the resources that are available (Albert *et al.*, 2016). The File Master daemon in this framework behaves the same way as the HDFS in Hadoop while the Node Manager has similarity with the Node Manager in Hadoop with the exception of bandwidth monitoring. However, in an attempt to

provide a shared-state mechanism where all framework schedulers have global knowledge of all resources in the cluster (both available and non-available resources), resource lease conflicts are bound to occur between schedulers. Though a mechanism is in place to resolve these conflicts by RM in Awan, an extra overhead will frequently be incurred in running applications in this framework. Also, with many tools built on top layer of YARN and its widespread implementation, Hadoop is still the most widely used distributed data processing framework. Konstantinos *et al.*, (2018) proposed more features in YARN resource manager. In this work, they added new features to YARN architecture as shown in Figure 2.30. The new features appear in orange.

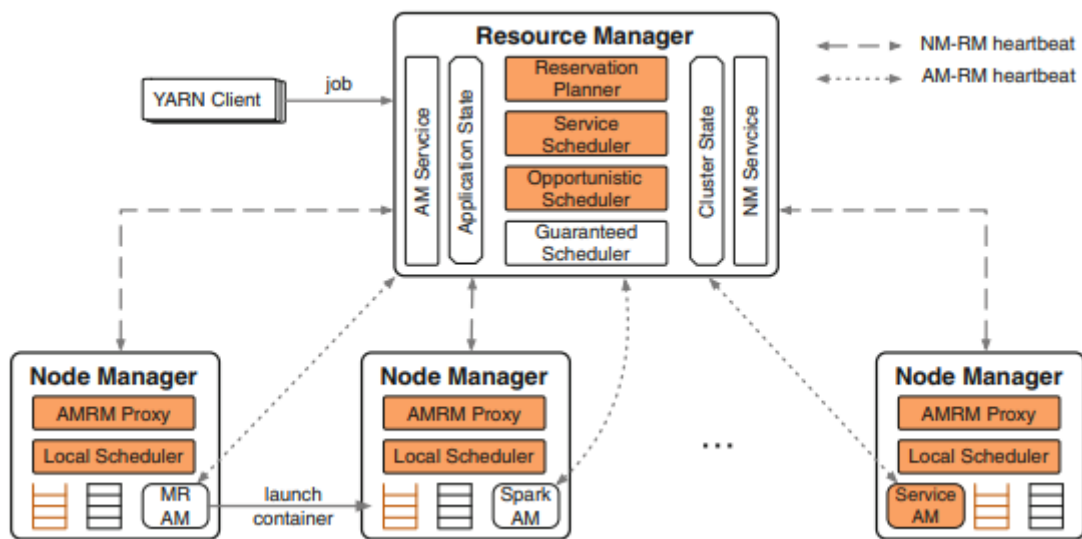


Figure 2.30: Advancement in YARN Resource Manager (Konstantinos *et al.*, 2018)

From Figure 2.30, reservation planner determines the resource needs and temporal requirements of a job and translates job's completion deadlines into a Service Level Objectives (SLOs) over predictable resource allocations (Konstantinos *et al.*, 2018). Service scheduler gives service owners the ability to control container placement so as to optimise performance of their applications while optimistic scheduler allows containers to be dispatched to Node Managers even if there are no available resources (jobs) on that node. At any point that job arrives at the Node Manager, optimistic containers will be picked from the queue and execution will start immediately, avoiding any feedback delays (Konstantinos *et al.*, 2018)



YARN framework is the most widely used and powerful tool for big data analytics. It is the architectural centre of Hadoop, extensible and very easy to integrate with many components. It allows several engines like interactive SQL, real-time streaming, data science and batch processing to handle data stored in a single platform. The popularity of this framework is largely because of its ability to store, analyse and access massive data more quickly, and its cost effectiveness across clusters of commodity servers. Hadoop YARN is actually not a single product but a collection of several components providing resource management and central platform to deliver consistent operations, security and data governance tools across Hadoop cluster. This has made it an all-round framework for big data solution.

## 2.7 Summary of Literature Review and Knowledge Gap

**MultiCore** (Bekkerman *et al.*, 2012; Dilpreet and Chandan, 2014) and **Graphic Processing Units** (Hong and Kim, 2009; Dilpreet and Chandan, 2015) can only be scaled up (a vertical scaling method where you install more processors, memory and faster hardware). This technique requires substantial financial investment and it is impossible to scale up after a certain limit. MPI in **Peer-to-Peer** (Steinmetz and Wehrles, 2005; Milojicic *et al.*, 2003) has no ability to handle faults in a network. In **Mesos** (Hindman *et al.*, 2011; Benjamin *et al.*, 2012), task description will have to be sent upon accepting a resource hence, no second-level scheduler to determine framework/application's own internal resource management. Again, because Mesos offers resources to framework, locality preference is hindered. Though **COSMOS** (Chaiké *et al.*, 2008) and **Corona** (Shouvik and Daniel, 2013) has similar architectural framework with YARN, for multiple applications/frameworks however, both frameworks will find it significantly difficult to handle. **Omega** (Malte *et al.*, 2013) architectural design geared towards distributed, multi-level scheduling which reflects a greater focus on scalability. It is however, hard to enforce global properties such as capacity/fairness/deadlines on this system (Schwarzkopf *et al.*, 2013). To Google, this approach is sensible. But for an open source platform like Hadoop, it is not amenable. This is because, arbitrary framework from diverse independent sources share the same cluster in Hadoop.

**High Performance Cluster Computing (HPCC)** (Patrick, 2011; Micheal *et al.*, 2014; Anthony, 2015; Camille, 2015) systems were initially designed with vertical scaling technique in focus. In this technique however, system has to be more powerful to handle future workloads. Addition of more processors, memory and faster hardware in an attempt to have better performance are mostly not fully utilized at the initial stage. **Disco** (Prashanth *et al.*, 2011; Camille, 2015) framework is a young big data tool which must evolve to become even more effective. Its distributed system cannot be compared with HDFS which is fault-tolerant. Losing a disk or a machine in Hadoop typically does not spell disaster for the data under consideration. **Spark** (Telmoda, 2015; Apache, 2016) is part of Hadoop big data ecosystem, which makes Hadoop YARN more robust than this framework when used alone. **Storm** (Telmoda, 2015; Apache, 2016) cluster is superficially similar to Hadoop cluster (Telmoda, 2015). Whereas you run ‘MapReduce jobs’ in Hadoop, you run ‘topologies’ in Storm. These two are very different. While jobs eventually finish in MapReduce, topology processes forever until it is killed (Telmoda, 2015). **Giraph** (Bakshi and Sonali, 2016) is still in a very immature phase of development which lack complete set of offered algorithms hence, can only be runned on Hadoop framework. **Berkeley Data Analysis Stack (BDAS)** (Kraska *et al.*, 2013; Dilpreet and Chandan, 2014) emerged to attack challenges of advanced analytics and machine learning on big data. Though this stack consist of many useful components in the top layer for various applications, many of these components are still at early stage of development hence; it support is limited. The cluster manager in BDAS is also a centralized resource manager hence; start-up time for a job is several tens of seconds.

All ‘joins’ operation in **Cloudera Impala** (Kornacker *et al.*, 2015; Bakshi and Sonali, 2016) are performed in memory capacity not sufficient by the smallest memory node present in the cluster. Cloudera Impala does not support querying streaming data as with Apache Spark in Hadoop cluster. There is also single point of failure in query execution with this framework. **Pentaho** (Vidhya *et al.*, 2014; Pasula, 2016) and **Jaspersoft** (Vidhya *et al.*, 2014) are designed specifically as Business Intellegence (BI) platform. With it self-explanatory design interface, this frameworks has made valuable contributions by providing business suggestions to business experts. The frameworks however, are not suitable for other big data analytic solutions like diagnostic analytics. **Splunk** and **Karmasphere** (Vidhya *et al.*, 2014) are part of hadoop framework which enhance the development of several Apache Hadoop-based applications to produce

insights from users' data. **Natezza** (Francisco, 2011; Bakshi and Sonali, 2016) is not suitable for online transactional processing and does not employ any query turning mechanism. The framework supports models like Hadoop, still making Hadoop the most robust big data framework. **Nephele-PACT** (Sharanjit *et al.*, 2014) framework is not as robust as Hadoop framework. This framework is still in its infancy stage and cannot accommodate big data tools like Hadoop. Hadoop has general acceptance and usage over Nephele-PACT. **Dryad** (Yuan *et al.*, 2016) has a centralized job manager which is a bottleneck for scalability. Scalability of each instance of *schedd* in **Condor** (Tannenbaum, 2010) is a major concern. Interaction with disk is another limiting factor for *schedd*.

**Nebula** (Jonathan *et al.*, 2017) is still at early stage and only a prototype has been demonstrated. There are limited numbers of applications and frameworks that can be ported to this system. Scalability of this system is not guaranteed because injection of external data and techniques for both aggregation and decomposition across distributed resources may crash the system. During execution of task, compute nodes still make download of application codes and input data from DataStore Master, this will incur additional overhead. **Sparrow** (Outerhout *et al.*, 2013) is suitable for distributed, low latency scheduling workloads but does not support gang scheduling. **FusionFS** (Zhao *et al.*, 2014) has vertical scale up which is very expensive. The per-node resource manager in **MATRIX** (Wang *et al.*, 2015) does not support distributed and parallel processing technique for big data analytics. **Awan** (Albert *et al.*, 2016) architecture is somewhat similar to YARN architecture but extra overhead is frequently incurred while running applications in Awan.

Though YARN framework stands out among most of the big data analytics due to its ability to run several other frameworks/applications, the responsibilities of global resource manager to handle requests from all of these applications/frameworks obviously constitute a bottleneck for scalability of Hadoop. Even with the addition of features in YARN resource manager as proposed by Konstantinos *et al.*, (2018), the global resource management in this framework slows down execution since all Node Managers (NM) from each compute nodes in the cluster and all Application Managers (AM) send/receive instructions/request from a single resource manager through heartbeat protocol. This process will reduce response time and total turnaround time for each job in the cluster. The aim of this research work is to decentralize the global

resource manager in YARN by having another layer called Rack Unit Resource Managers (RU\_RM) responsible for resource management of nodes in their corresponding rack.

## CHAPTER THREE

### SYSTEM ANALYSIS AND METHODOLOGY

#### 3.1 Analysis of the Existing System

YARN is acronym for “Yet Another Resource Negotiator” also called MapReduce 2. It is considered as the next generation MapReduce, considering its improvement over MapReduce 1 which has scalability bottleneck when cluster size grows beyond 4000 nodes. The main idea in YARN is to split the JobTracker’s responsibilities into two:

- i. Resource Manager: Does job scheduling portion of the workload.
- ii. Application Master: Does the task monitoring portion of the workload.

#### Entities in YARN

- i. Client:- Responsible for submission of jobs and also interact with MapReduce and HDFS framework.
- ii. Resource Manager:- Responsible for allocating computing resources and data required by the job. It has two units – Resource Scheduler and Application Manager.
- iii. Node Manager:- This is present at the slave nodes and is responsible for creating execution containers and monitoring containers’ usage.
- iv. Application Master:- Coordinates and manages MapReduce jobs, negotiates with Resource Manager to schedule tasks. The tasks are started by Node Manager.
- v. YARN child:- Responsible to send status of task to the application master.
- vi. Distributed File System:- Shares resources and job’s artefacts between YARN components.

#### 3.1.1 Data Flow of the existing system

Job execution in YARN is in phases as shown in Figure 3.1. These phases include job submission phase, job initialization phase, task assignment phase, task execution phase, progress and update phase and job completion phase.

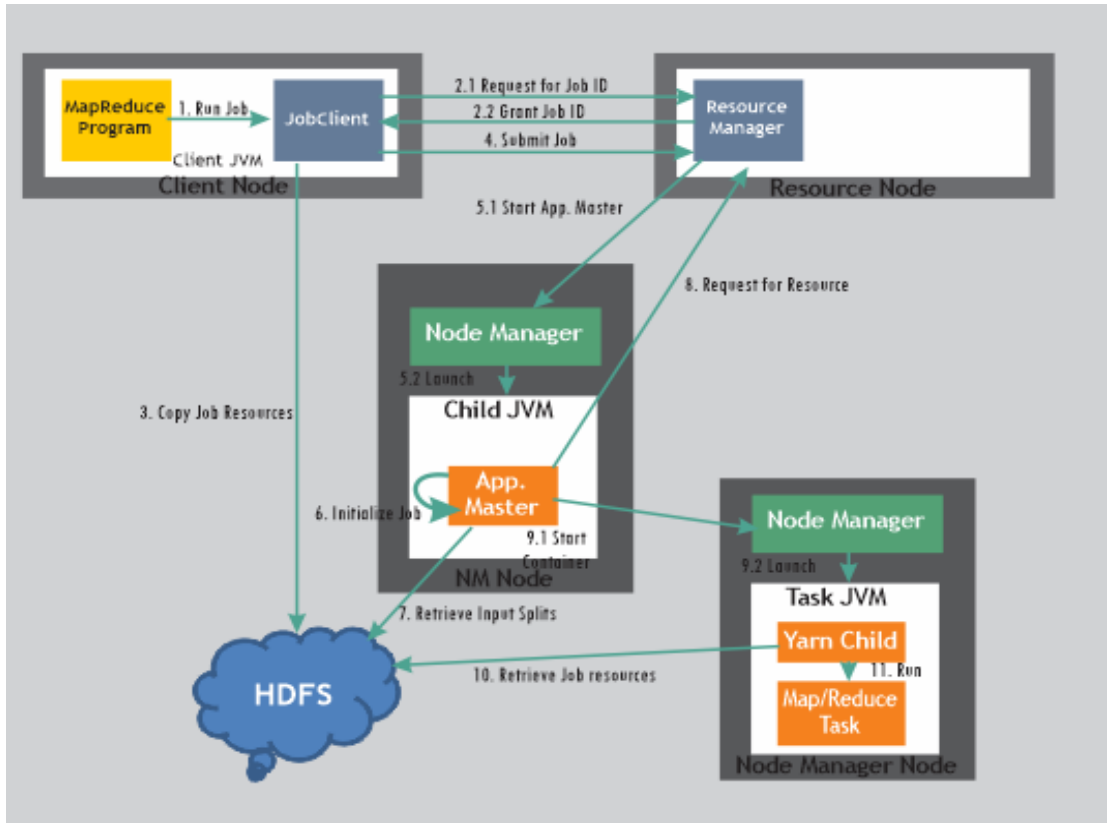


Figure 3.1: MapReduce Job Execution in YARN

a. Job Submission Phase

- Step 1: The job gets submitted to job client.
- Step 2: The job client request for a new job id.
- Step 3: The job client then checks if output directory has been created. After verifying this, it copies the job resources to the HDFS.
- Step 4: The job client then submits the job to the Resource Manager.

b. Job Initialization Phase

Remember that, Resource Manager has two units – Resource Scheduler and Application Manager. The scheduler schedules and allocates resources while Application Manager monitors status and process of the job.

- Step 5: As soon as the scheduler picks a job, it contacts the Node Manager to allocate a container and launch Application Master for the job.

Step 6: Application Master creates object for the job. This is done for book keeping purposes and task management.

Step 7: The Application Master retrieves input splits from HDFS and creates 1 map per split. The Application Master at this point decides how to execute the job. If the job is a small task, the Application Master runs the job in its JVM to avoid unnecessary overhead. These type of tasks are called Uber tasks in Hadoop framework.

c. Task Assignment Phase

Step 8: If the job is large, Application Master requests the Resource Manager to allocate the computing resources needed. Scheduler at this point knows where the resources are located. It gathers this information from the heartbeat it gets from each worker node. It uses this information to consider data locality while assigning a task. The scheduler tries as much as possible to assign a task to where the data are located. If this is not possible, it assigns the task to another node within the cluster.

d. Task Execution Phase

Step 9: Application Master contacts the Node Manager assigned to execute the task, to start a container. The Node Manager then launches the YARN child. YARN child is a Java program which has main class "YarnChild". It runs a separate JVM to isolate user code from long running system.

Step 10: YarnChild retrieves all job resources from the HDFS.

Step 11: YarnChild now runs the map and reduce tasks.

e. Progress and Update Phase

In this phase, YarnChild sends the progress report every 3 seconds to the Application Master. Application Master in turn aggregates and sends update directly to the job client.

f. Job Completion Phase

Application Master and task containers clean up their working state.

### 3.1.2 Advantages of the existing system

From the analysis of the existing system, we observed that;

- i. There is increased scalability in YARN as compared with classic Hadoop. The reason is because, YARN decouples the work of JobTracker into two making it easier to scale up worker nodes beyond 4000.
- ii. Along with MapReduce, there can be another distributed framework on the same cluster environment.
- iii. Better utilization of resources with the concept of containers. Containers are like slots in classic Hadoop but, slots are fixed for each task while containers are flexible. In classic MapReduce, a task will have specific number of map and reduce slots which most times are not fully utilized. While some slots are under-utilized, others are over-utilized.

### 3.1.3 Disadvantages of the existing system

Though YARN architectural design has improved scalability significantly, there are fundamental design issues that cap the scalability of this framework towards extreme scales. Some of these design issues include,

- i. Centralized Resource Manager:- Resource manager, which is the core component of Hadoop framework offers the functionalities of managing, provisioning and monitoring resources like the CPU, memory and network bandwidth of compute nodes. These responsibilities obviously, are a bottleneck for scalability of Hadoop towards extreme scales. It also slows down execution since all compute nodes send/receive instructions from a single resource manager through heartbeat protocol. Once resource manager fails, all execution will halt. Although YARN provides RM High Availability to protect against single point of failure, this technique causes computation overhead because, resource manager needs to update the backup storage.



- ii. Hadoop replication factor:- Replication factor in Hadoop framework is such that  $\frac{2}{3}$  of each block (of a whole file) are replicated into different data nodes across racks in a cluster. Since Application Master is expected to monitor the execution of a job/application (with its complete number of blocks) in a cluster, AM will need to communicate data nodes with input splits of the corresponding job/application across racks to be able to monitor this execution. Communication across racks will result to higher latency in job execution.
- iii. Job completion time:- Since only resource manager coordinates the release of resources for execution of jobs, several Application Masters (AMs) polling from Resource Manager of this framework during resource request is a bottleneck for the system. It slows down processing, which means that total turnaround time (job completion time) for each job will be high.

### **3.2 Analysis of the New System**

The main aim of this model is to decentralize the global control of Resource Manager in YARN framework by providing another layer called Rack Unit Resource Manager (RU\_RM) layer. The aim of this layer is to make compute nodes on each rack to be controlled by their corresponding Rack Unit Resource Manager instead of a single Resource Manager controlling all the compute nodes in the network. We believe that this will help improve response and turnaround time for each job/application and will eliminate single point of failure which makes jobs halt in the existing global resource manager.

The second idea is to ensure that all Rack Unit resource managers form a peer-to-peer architecture such that each Rack Unit resource manager holds resources for which it is directly responsible to and also have backup copies of resources for the RU\_RM preceding/succeeding it. This will ensure that, if any RU\_RM fails, the predecessor or successor can continue with the management of compute nodes in that rack until such RU\_RM recovers from failure.

The major aim of this new framework therefore, is to provide lower turnaround time for jobs and to ensure high availability of Resource Manager during job execution. The

new framework has five (6) phases – Job submission, job initialization, task assignment, task execution, progress/update and job completion phase. Figure 3.2 explains MapReduce job execution on our new framework.

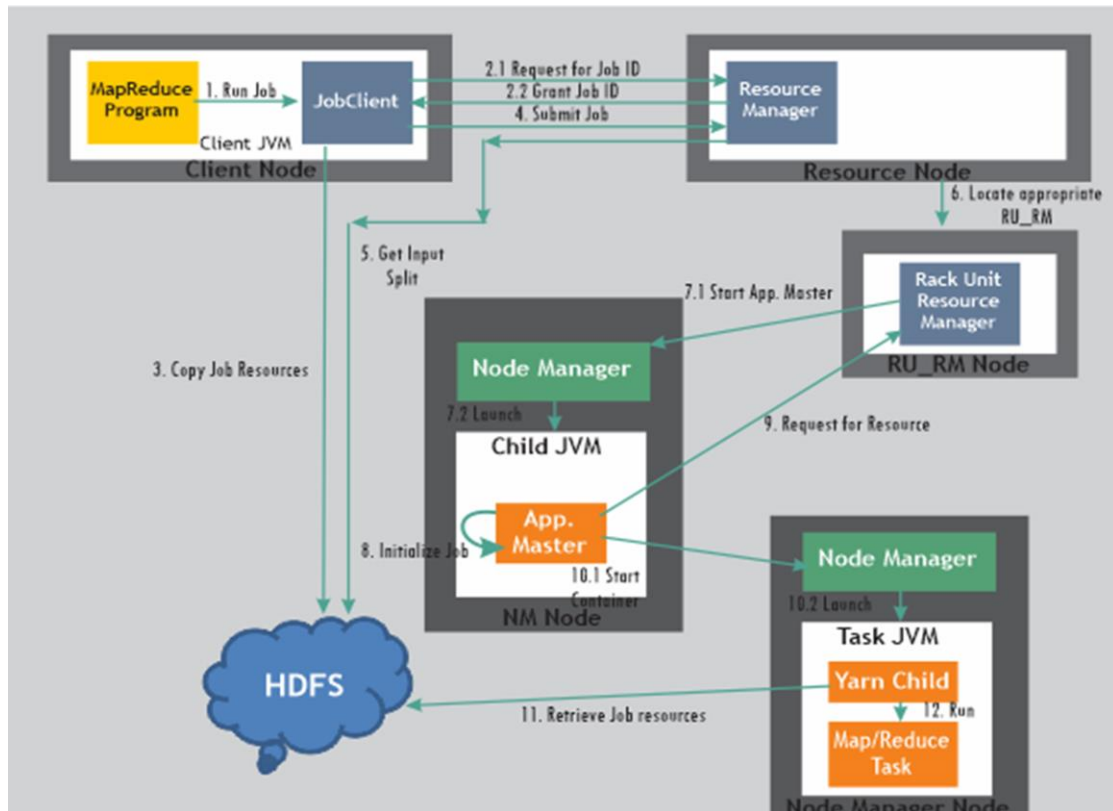


Figure 3.2: MapReduce Job Execution on the new framework

a. Job Submission Phase

- Step 1: The job gets submitted to job client.
- Step 2: The job client request for a new job id.
- Step 3: The job client then checks if output directory has been created. After verifying this, it copies the job resources to the HDFS.
- Step 4: The job client then submits the job to the Resource Manager.

b. Job Initialization Phase

- Step 5: Resource Manager gets input splits for the said job.

- Step 6: With the information in Step 5, the Resource Manager schedules appropriate Rack Unit Resource Manager (RU\_RM) with 2/3 of the input split to execute the job.
- Step 7: The scheduler at the RU\_RM picks the job and contacts the appropriate Node Manager to launch Application Master for the job.
- Step 8: Application Master creates object for the job. This is done for book keeping purposes and task management. The Application Master creates 1 map per split from each input split on data node. The Application Master at this point decides how to execute the job. If the job is a small task, the Application Master runs the job in its JVM to avoid unnecessary overhead.

c. Task Assignment Phase

- Step 9: If the job is large, Application Master requests the Rack Unit Resource Manager to allocate the computing resources needed (container). Scheduler at this point knows where the resources are located. It gathers this information from the heartbeat it gets from each worker node in the rack. It uses this information to consider data locality while assigning a task. The scheduler tries as much as possible to assign a task to where the data are located. If this is not possible, it assigns the task to another node within the rack.

d. Task Execution Phase

- Step 10: The Rack Unit Resource Manager through the appropriate Node Manager launches the YARN child.
- Step 11: YarnChild retrieves all job resources from the HDFS.
- Step 12: YarnChild now runs the map and reduce tasks.

e. Progress and Update Phase

In this phase, YarnChild sends the progress report every 3 seconds to the Application Master. Application Master in turn aggregates and sends update directly to the job client.

f. Job Completion Phase

Application Master sends output to HDFS

Application Master and task containers clean up their working state with the help of Container Expirer.

### 3.2.1 Justification of the New System

To justify the working methodology of our new model, a hypothetical evaluation is carried out, which analyse the results obtained in this new model to results from the YARN model. Let us assume to have three jobs (applications) to be processed and that, each step in executing any of these job takes 0.01ns (assume that three jobs are of the same size). For each job therefore, the first 5 steps in the existing framework holds as obtained in Figure 3.1.

Since there is only one resource manager, the last seven steps will require Resource Manager communicating with Node Manager to launch containers and with Application Master for resource requests. Since no more than one instruction can be given at a time, it means that Resource Manager will interleave these instructions between the three jobs (applications).

Assume that the time taken for each job to be attended to is 3ns and the interveaning of process follows FIFO order. It means that *Job1* get resources immediately hence, delay time is zero (0). *Job2* will get assess to resource at time 3ns while *Job3* at time 7ns. The overall time it will take to process the three jobs will be as follows:

$$\text{Job 1} = (0.01\text{ns} \times 5) + 0\text{ns} = 0.05\text{ns}$$

$$\text{Job 2} = (0.01\text{ns} \times 5) + 3\text{ns} = 3.05\text{ns}$$

$$\text{Job 3} = (0.01\text{ns} \times 5) + 7\text{ns} = 7.05\text{ns}$$

**Total instruction time needed to process the three jobs = 0.05ns + 3.05ns + 7.05ns = 10.15ns**

With the new model, the first 7 steps holds for all the three jobs as obtained in Figure 3.2. Since each RU\_RM node executes just one job at a time, the last 5steps therefore are carried out at the same time on different RU\_RM node.

Therefore, if it takes 3ns for the three jobs to be attended to in the existing system; it will take *1/3ns of 5steps* for these jobs to be attended to in the new model. Hence, the process time for the three jobs will be as follows:

Job 1 = (0.01ns x 7) + *1/3ns of 5* on RU\_RM1 = 0.07ns + 1.67ns = 1.74ns

Job 2 = (0.01ns x 7) + *1/3ns of 5* on RU\_RM2 = 0.07ns + 1.67ns = 1.74ns

Job 3 = (0.01ns x 7) + *1/3ns of 5* on RU\_RM3 = 0.07ns + 1.67ns = 1.74ns

**Total instruction time needed to process these three jobs = 1.74ns x 3 + = 5.22ns**

From our analysis; ignoring bottlenecks associated with network bandwidth and communication overhead, we observe that it takes 10.15ns to pass instruction that will execute three jobs in YARN model, whereas it takes 5.22ns to do same with our new model. This shows that, our new model promises a better response time and lower turnaround time compared to the existing model.

### **3.3 The Organization and its Environment**

Apache Software Foundation (ASF) who is the custodian of Hadoop project is a non-profit public charity organization incorporated in the United States of America in 1999 primarily to provide a foundation for open and collaborative software development projects so as to create an independent legal entity to which companies and individuals can donate resources and be assured that those resources will be used for the public benefit (Apache, 2016). The foundation provides a means through which individual volunteers can be sheltered from legal suit directed at the foundation's projects and also protect the 'Apache' brand as applied to its software product, from being abused by other organizations (Apache, 2016).

The organizational governance at a higher level of this foundation is fairly simple; members are the ones to elect Board of Directors, the board will appoint various

officers and creates Project Management Committees (PMCs) who report periodically to the board. Most other officers report to the board through the president of the foundation. The foundation’s corporate government reporting structure is shown in Figure 3.3.

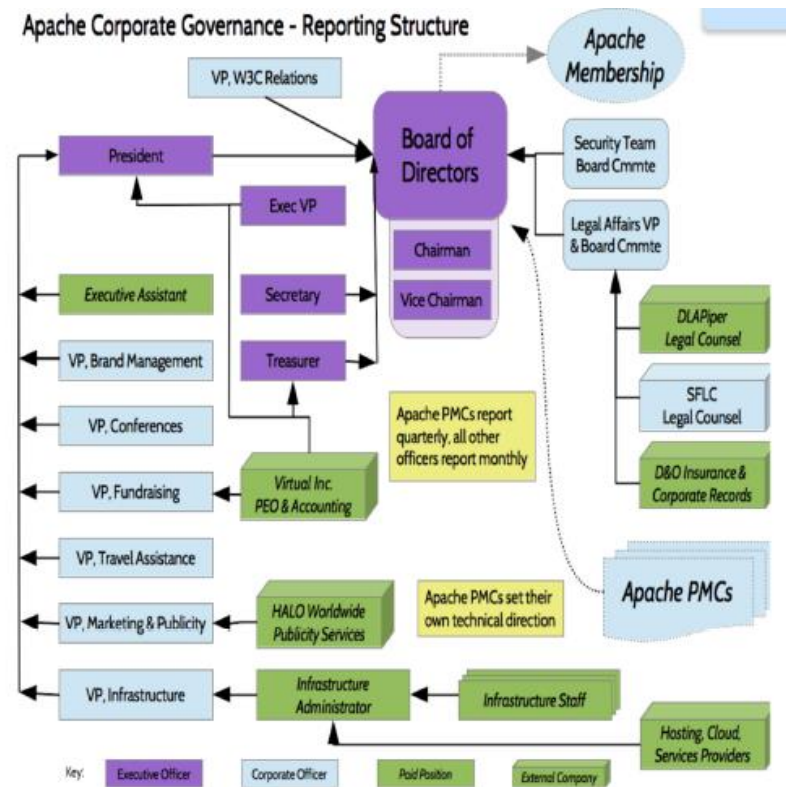


Figure 3.3: Apache Corporate Governance reporting structure (Apache, 2016)

For elections and appointments purposes, the following procedures are followed in the foundation.

- i. Existing members are charged with the responsibility of nominating and electing new members periodically, and they nominate and elect nine (9) directors to the board annually.
- ii. The board are charged with the responsibility of appointing operational officers. They delegate responsibility for specific policy/operational areas to each officer.
- iii. The board has the sole power to appoint executive officers including the President, Secretary, Treasurer etc who are responsible for day-to-day operations of the foundation.

- iv. Most officers report to the President on a monthly basis and the President in turn reports an overall operational status to the board also on a monthly basis. Figure 3.4 shows the foundation’s corporate government – elections and appointments.

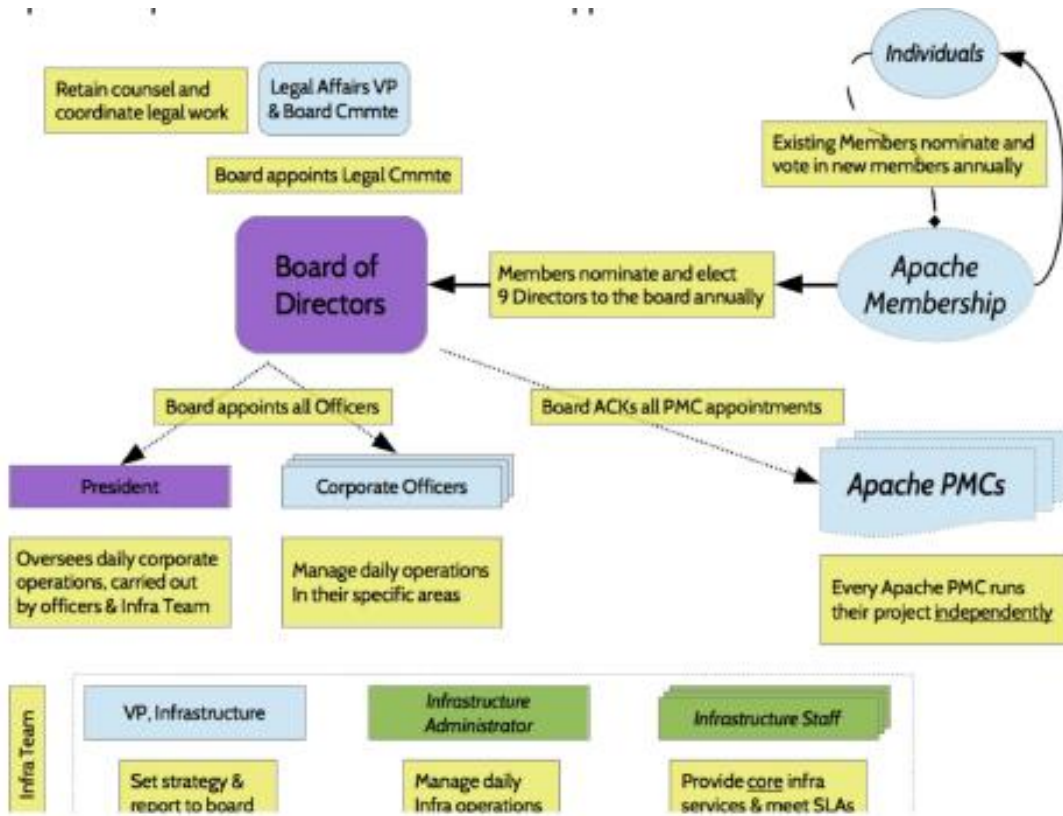


Figure 3.4: Apache Corporate Governance—Elections and Appointments (Apache, 2016)

Every Apache PMC manages their project independently. The following procedures are followed for project governance.

- i. PMCs reports progress of work directly to the board quarterly. The organizational oversight of PMCs and its functioning as a healthy community and to ensure they follow Apache policies is done by the board. For technical governance however, PMCs take full charge.
- ii. The chair of each PMC is a Vice President for that project and thus, an officer of the foundation. The VP ensures that project reports are complete and submitted to the board.

- iii. PMCs vote on software product releases. This is to ensure that all source code releases are acts of the foundation itself, through properly governed PMC.
- iv. PMCs can nominate and elect new committees to their project.
- v. Apache incubators called special IPMC help to mentor new podling communities to help them learn the Apache way. Figure 3.5 gives a view of the foundation's project governance structure.

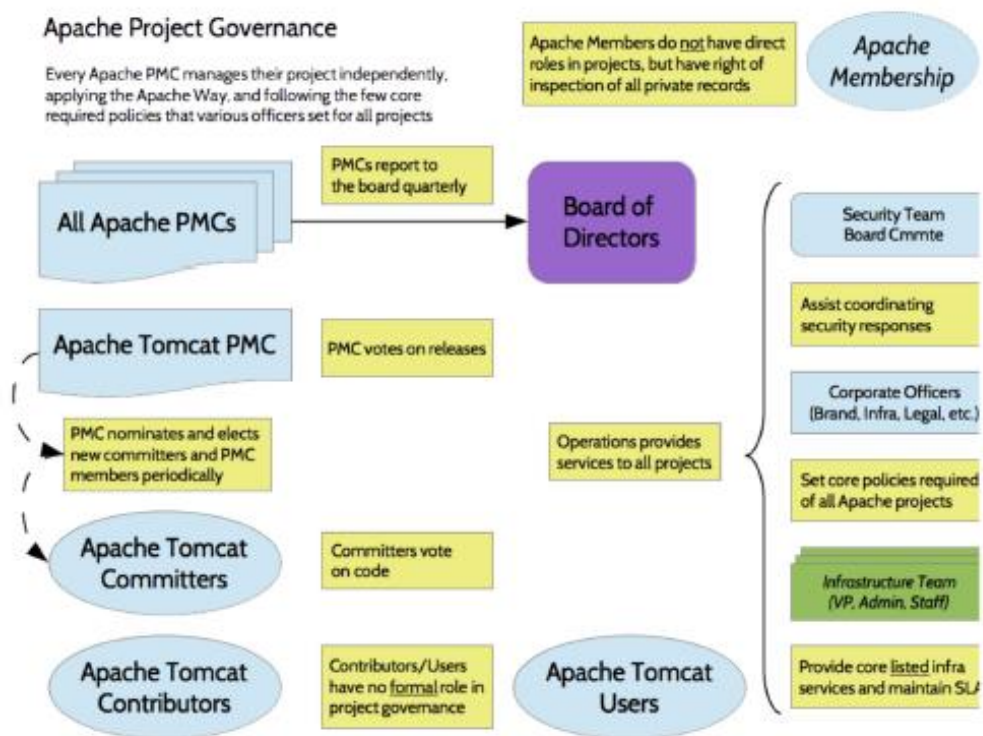


Figure 3.5: Apache Project Governance Structure (Apache, 2016).

### 3.4 Methodology

We live in the world of objects. These objects exist in such a way that they can be created, organised, categorized, described and manipulated upon. Hence, Object Oriented Methodology (OOM) has come into picture for developing software. OOM is a new system development methodology that encourages and facilitates reuse of software components. This methodology makes it easier for a system analyst to determine what objects are required in a system, how each of these objects behave over time or in response to an event, responsibilities and relationships of these objects with each other, their commonalities, differences and how the system will manipulate them.



Because of the nature of this research, the new model used this methodology because it provides nice structures for thinking and abstraction, which leads to modular design. Also, the methodology encourages reusability and provides inheritance feature of object-orientation. Inheritance will allow a program to use the existing classes in new application. So many OOM exist, popular among them are Booch methodology with the concept of Object Oriented Analysis and Object Oriented Design (OOA/OOD), Responsibility Driven Design (RDD) methodology, Object Oriented Software Engineering (OOSE) methodology, Object Modelling Techniques (OMT) methodology.

This research work used OMT methodology because it describes a method for analysis, design and implementation using object-oriented technique. It is fast and provides intuitive approach that identifies and model objects making up a system. To justify the proposed methodology, OMT three major viewpoints was considered in the design phase of this work, each viewpoint capturing important aspects of the system. The three viewpoints are static, dynamic and functional behaviours of the system also described as object model, dynamic model and functional model of OMT.

### **UML description of the new model with methodology adopted**

UML are meant to provide model for computer applications/software. UML notation set provides several diagrams that when used within a given methodology will increase the understanding of a system under development. The UML diagrams used follow the object oriented methodology for this work. The UML description looked at the three major viewpoints of OMT in object oriented methodology with each view point capturing important aspects of the system. The three viewpoints are static, dynamic and functional behaviours of the system also described as object model, dynamic model and functional model of OMT.

Object model describes objects in a system and their inter-relationships. It gives more attention to objects as static entity and does not pay attention to object's dynamic nature. This model describes the structure of object in the system. From analysis of the new model, key objects in this model include;

- a. Client
- b. Resource Manager (RM)
- c. Rack Unit Resource Manager (RU\_RM)

- d. Node Manager (NM)
- e. Application Master (AM)
- f. Hadoop Distributed File System (HDFS)

Class diagram was used to describe the structural and data aspects of this system. Figure 3.6 describes the class diagram for the new model, the relationship of objects, their operations and also depicts the primary view of the overall decomposition of the system.

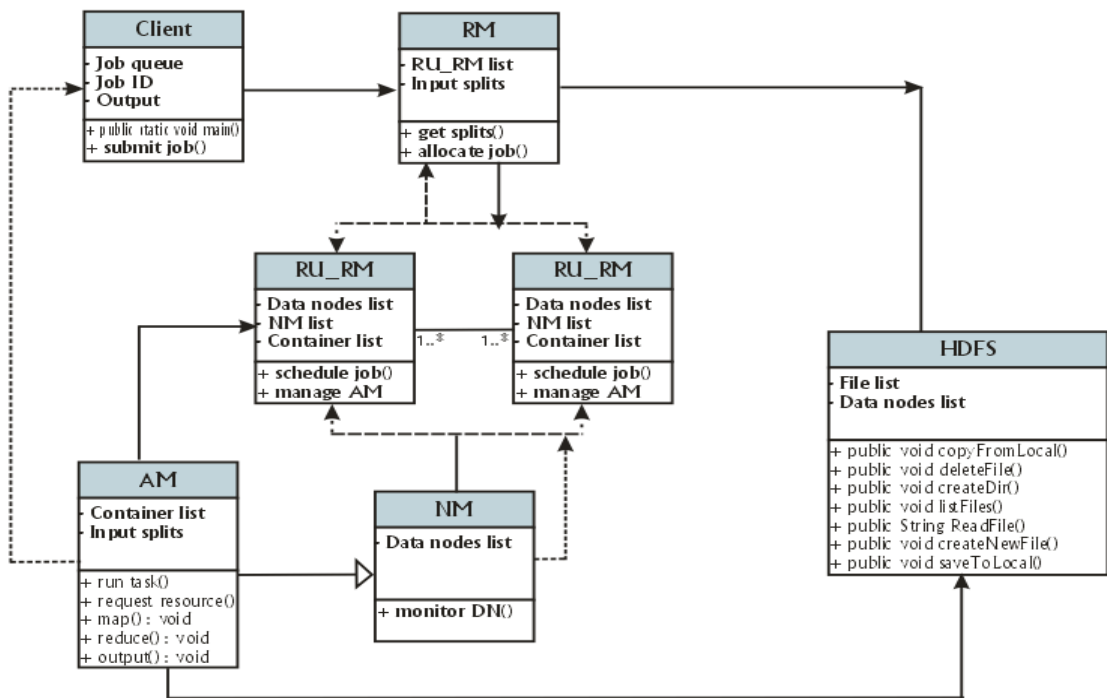


Figure 3.6: Class diagram for the new model

Dynamic models are used to represent behaviour of the static constituents of software. Static constituents are objects and their relationships. It represents the interaction, workflow and different states of these static constituents in software. Diagrams used in dynamic models include interaction diagrams (sequence or communication diagram), object diagram and activity diagram. The new model describes those areas of the system that changes using sequence diagram as shown in Figure 3.7. Sequence diagram captures the sequence of messages flow from one object to another.

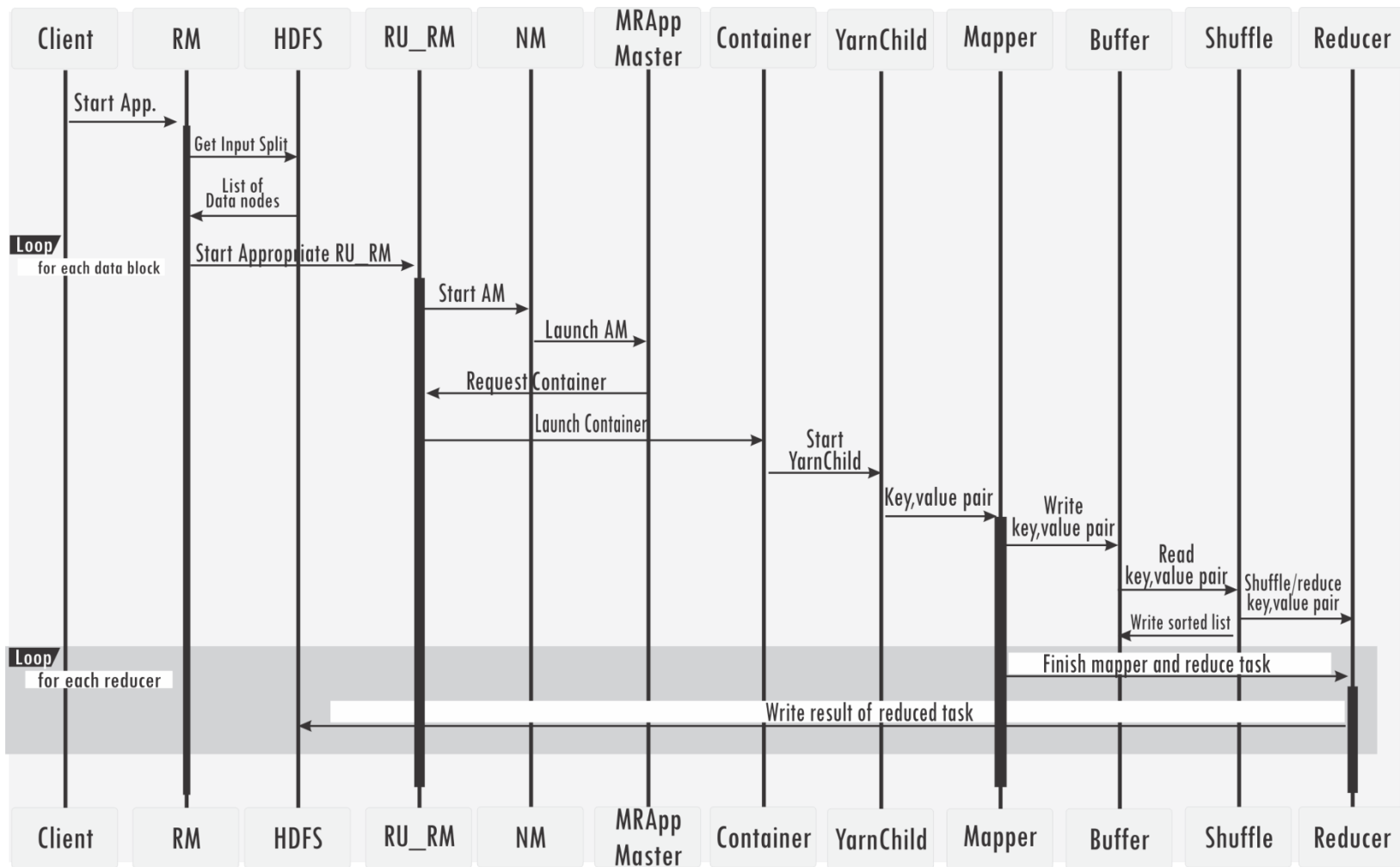


Figure 3.7: Sequence diagram for the new model

Functional Model:- This model describes the transformational and functional aspects of the new system. It captures what the system does without regard to how or when it is done. This model uses Data Flow Diagram (DFD) to show the flow of data through the system. It views system as a function that transforms input into desired outputs. Figure 3.8 shows DFD for the new model.

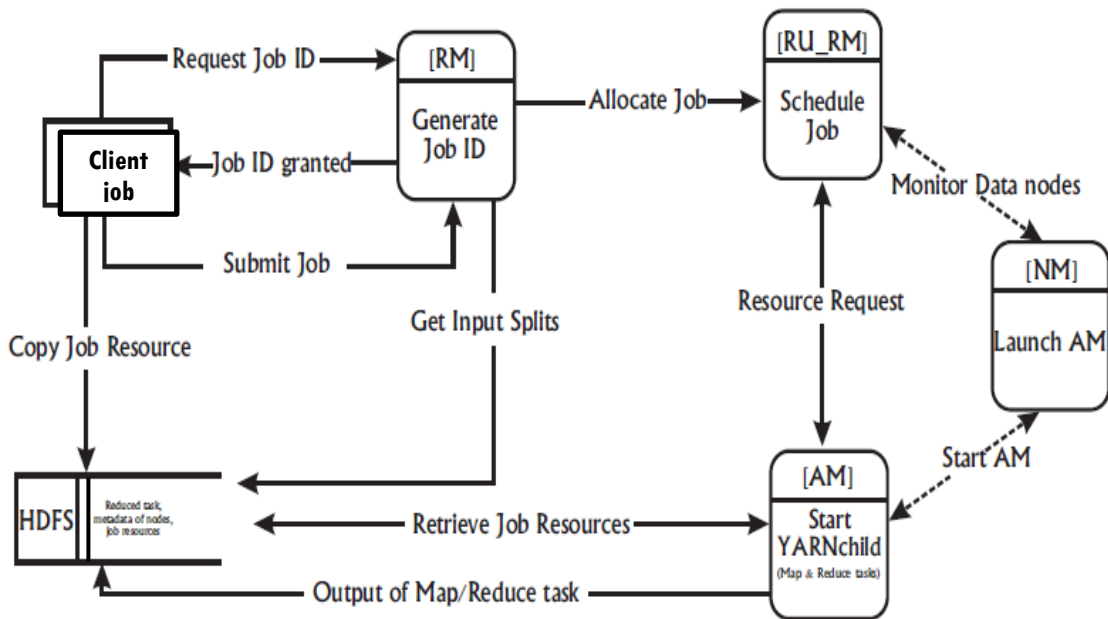


Figure 3.8: DFD for the new model

### 3.5 High Level Model of the New System

To capture and precisely state requirements and domain knowledge of the new framework, a high level model is presented in Figure 3.9. This model serve to focus the thought process and to capture requirements needed for the system design.

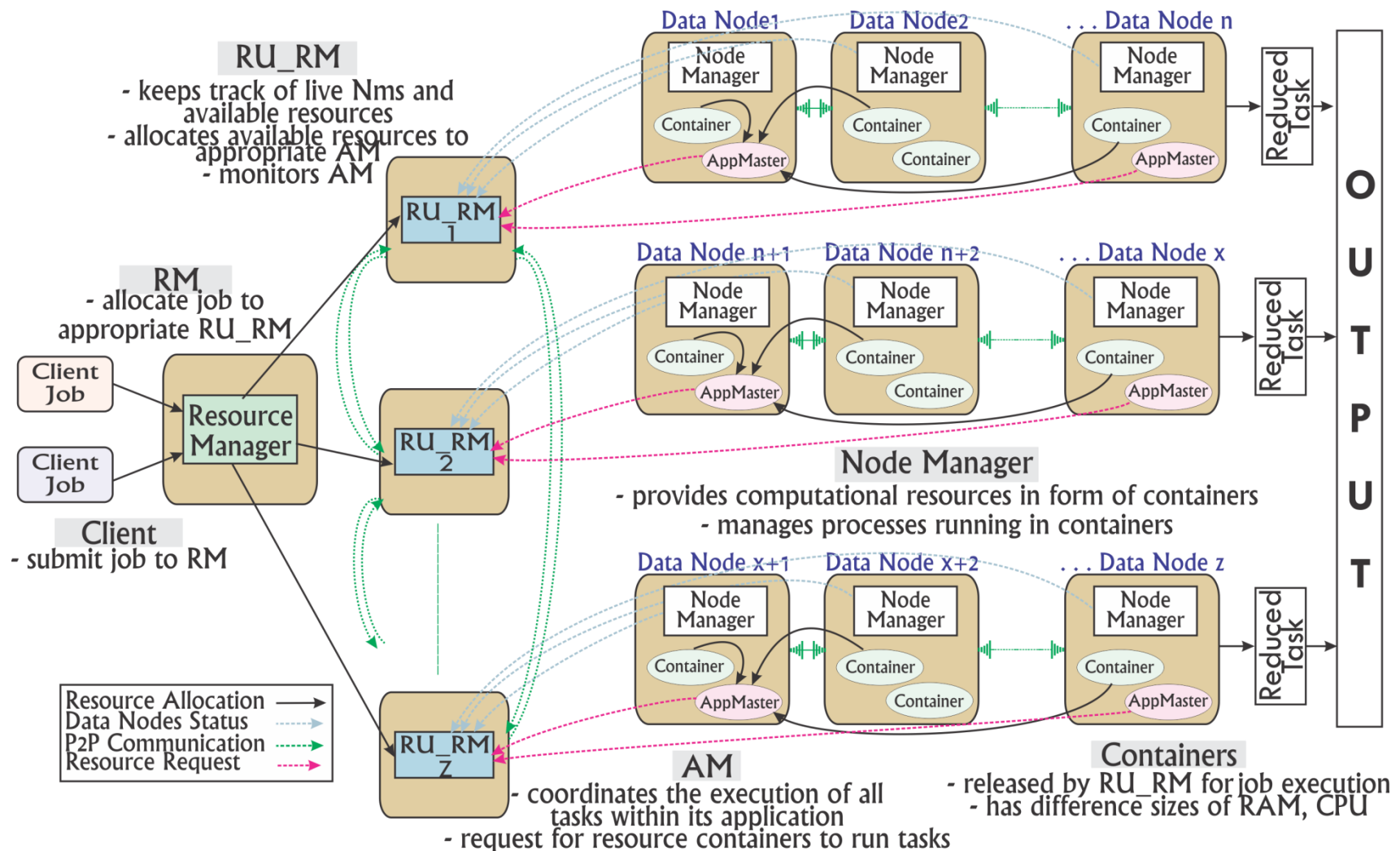


Figure 3.9: High-Level Model of the new rack-aware resource manager framework

The architectural framework of the new model as described in Figure 3.9 decouples the responsibility of Resource Manager by providing another layer where each daemon called Rack Unit Resource Manager (RU\_RM) carry out the responsibility of allocating resources to data nodes. This will allow high bandwidth and low latency for large files on data nodes within the same local rack. The illustration below will help understand our new model.

Assume we have three different files (sales, music and video file) to be stored in HDFS. The NameNode gets these files from the client, split the files into blocks and allocates them on free data nodes. Assume we have the following input splits.

**First File:** sales

**Blocks:** a.sales, b.sales, c.sales

**Second File:** music

**Blocks:** a.music, b.music

**Third File:** video

**Blocks:** a.video

With the new model, compute node replicates each block on different node but on same local rack. Compute node with the replicated block will replicate the third copy on a node in another rack. Each of the three compute nodes will communicate the NameNode once replication is over. The NameNode will then update the metadata server. Metadata server keeps record table for the three files as described in Table 3.1.

**Table 3.1:** Metadata table for all input splits in HDFS

Jobs	RU_RM1				RU_RM2			
	DN1	DN2	DN3...	DN n	DN n+1	DN n+2	DN n+3	...DN p
a.sales	+	+				+		
b.sales			+	+				+
c.sales		+	+		+			
a.music				+	+	+		
b.music		+				+		+
a.video	+						+	+

“+” represent data locality on a node.

Remember that from Figure 3.2, Resource Manager (RM) gets input split with their corresponding local racks from HDFS. For resource manager to schedule the appropriate RU\_RM, input split for each job is considered. If 2/3 of the replicas belong to a particular rack, RM allocates the job to the RU\_RM which in turn, allocates the block to the appropriate compute node for execution. From Table 3.1, the file “sales” will be allocated to RU\_RM1 while “music” and “video” will be allocated to RU\_RM2.

For a reliable, fault tolerant system and to guarantee lookup consistency in the presence of failure of nodes, the RU\_RM layer of the new model introduced novel relaxed-ring architecture. This approach help eliminate single point of failure experienced in the existing system. The new model provides that, every step in the ring needs the agreement of two RU\_RM nodes which is guaranteed with a point-to-point communication. Our first invariant is that, every RU\_RM node must have a predecessor and a successor in the ring. Secondly, the responsibility of a RU\_RM node starts with the key of its predecessor + 1 and ends with its key. Thirdly, a RU\_RM accommodates its neighbouring peers for back-up purpose. Fourthly, every RU\_RM node must communicate its predecessor and successor and update them with its responsibilities to remain in the ring. The last condition is that, every RU\_RM node must communicate with the central Resource Manager in situation where its neighbour node fails.

## CHAPTER FOUR

### SYSTEM DESIGN AND IMPLEMENTATION

#### 4.1 Objectives of the design

The objective of this design shall follow the objectives of the research study mentioned at the beginning of this work.

- (i) To decentralize the global control of Resource Manager (RM) in YARN framework by providing another layer called Rack Unit Resource Manager (RU\_RM) layer. By decentralizing this component, there will be two daemons serving as resource control for job execution; the central Resource Manager and the per-rack resource manager (described as Rack Unit Resource Manager in this work).
- (ii) The second objective of this design is to ensure that all Rack Unit resource managers form a peer-to-peer architecture such that each Rack Unit Resource Manager holds resources for which it is directly responsible to and also have backup copies of resources for the RU\_RM preceding/succeeding it.

The whole system design is described in Figure 4.1. Resource Manager now uses push-based approach to transfer responsibilities of scheduling jobs and monitoring node status to Rack Unit Resource Manager.



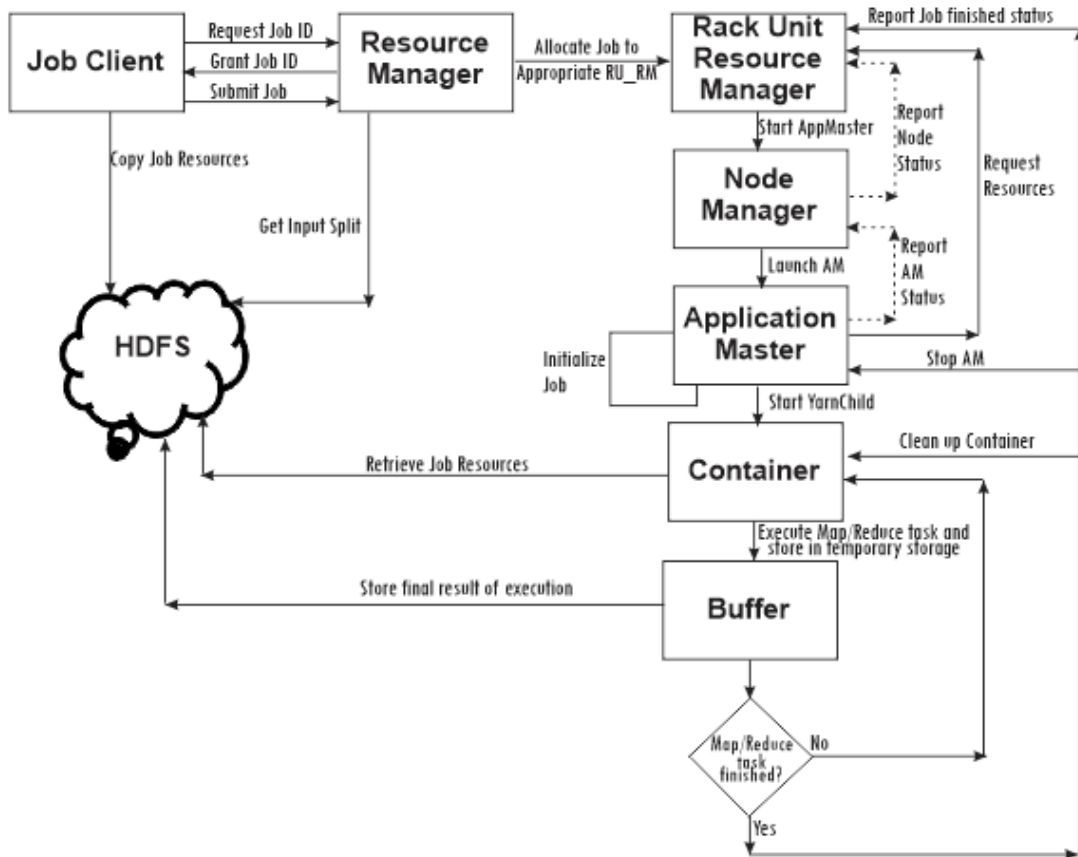


Figure 4.1: Block diagram showing whole system design processes

#### 4.2 Decomposition and Cohesion of High Level Model

The new model has three basic layers; the global resource manager layer, rack-units resource managers' layer and the compute nodes layer. The RM, whose responsibility is to allocate job to appropriate RU\_RM is highly cohesive. This module performs exactly one task which is allocation of job to appropriate RU\_RM. Each RU\_RM also is high cohesive; its responsibility is tied to nodes in its rack. In case of failure, where responsibility is extended beyond RU\_RM's boundary, the module ensures that data (contents) of the extended boundary remain within the rack(s) they are originally resident. Node managers and AMs in each data node are also highly cohesive with each AM solving just a task at a time.

While there is high cohesion within modules in each layer, the layers are loosely coupled. This is important so that modification at any layer of the model does not affect/change modules in other layers. With this, maintenance becomes easier and if additional rack (with compute nodes) is added to the cluster, RU\_RM, NM and AM can be picked and re-used without having to build them from the beginning.

### 4.2.1 Control Centre/Main Menu

The control centre/main menu of this system is an improved YARN Resource Manager GUI. Figure 4.2 shows the main interface. Cluster metric are displayed in the top row while the left hand portion of the menu provides navigations to sub-menus of the system.

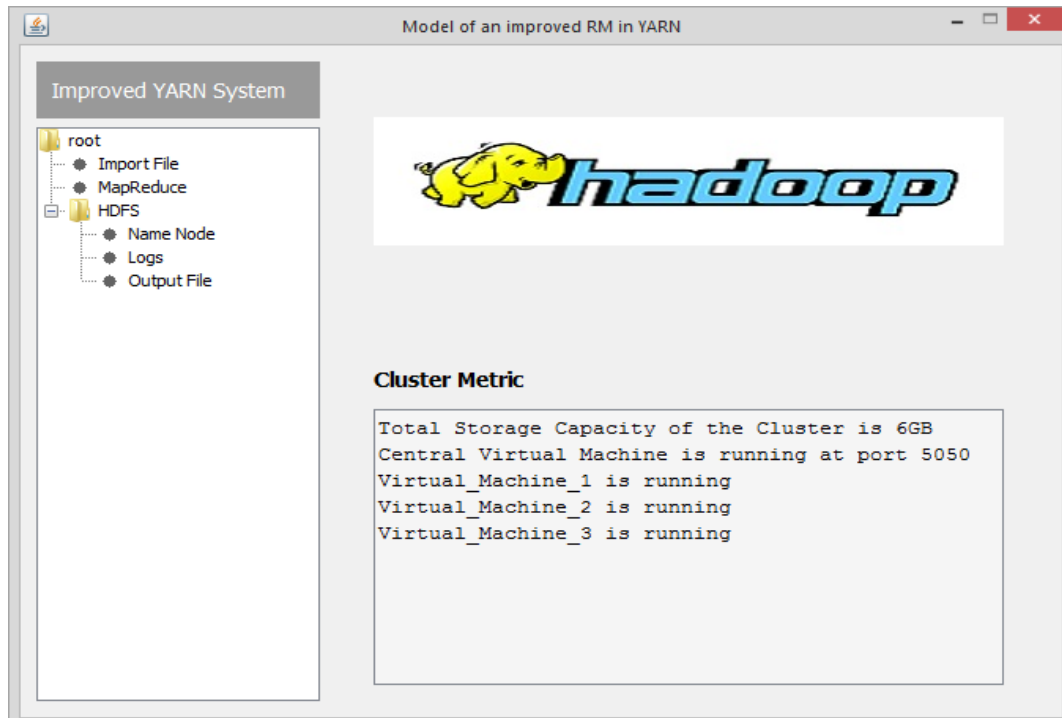


Figure 4.2: Control Centre/Main Menu for the New System

The cluster metrics provides information about size of data uploaded for word count operation, number of virtual machines connected to the server and the total time elapsed for each word count operation. The sub-menus on the left of the control centre shows operations that are performed on this framework.

### 4.2.2 The Sub-Menus/Sub-Systems

The sub-menu as shown in Figure 4.2 has the data submission, HDFS and MapReduce sub-menus. Each of these sub-menus has specific responsibility in the model.

#### 4.2.2.1 Data Submission Implementation

This sub-menu allows users to import text file to be stored in the server. Figure 4.3 shows the data submission sub-system.

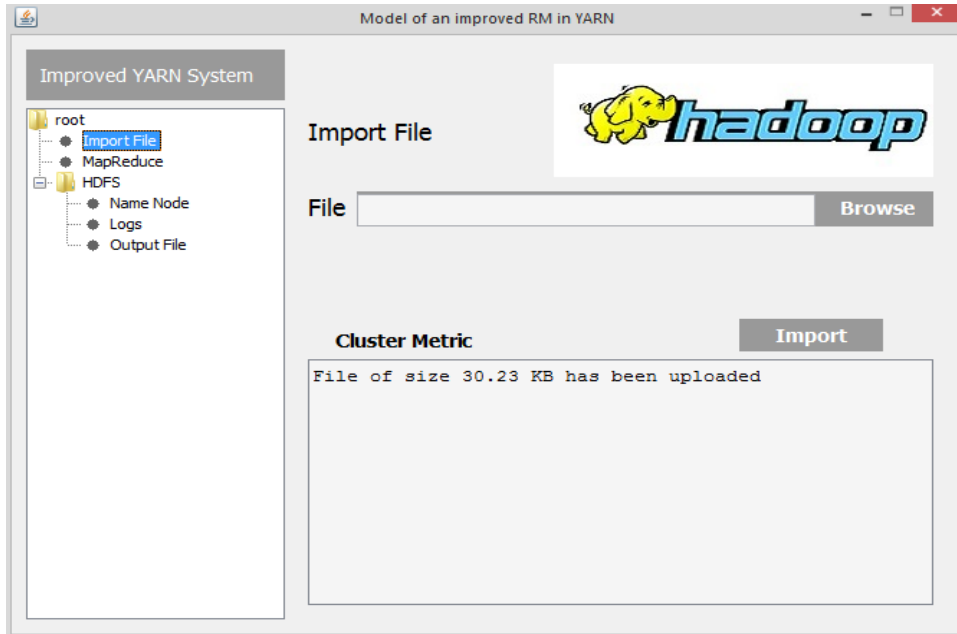


Figure 4.3: Data submission sub-system

From Figure 4.2, the user selects file to be imported from the client’s system. Once the user clicks on ‘Import’ button, the size of file (in kB) is displayed in the cluster metric.

#### 4.2.2.2 MapReduce Sub-system Implementation

This sub-menu allows users to run Hadoop workload called WordCount. The user can select either the existing model or the improved model. The two models are built into this system to allow for comparison and for the purpose of evaluation. The existing model contains re-usable modules hence, the architecture was not altered. The user is expected to select a file for the WordCount operation and click on the submit button. Figure 4.4 shows the mapreduce sub-system.

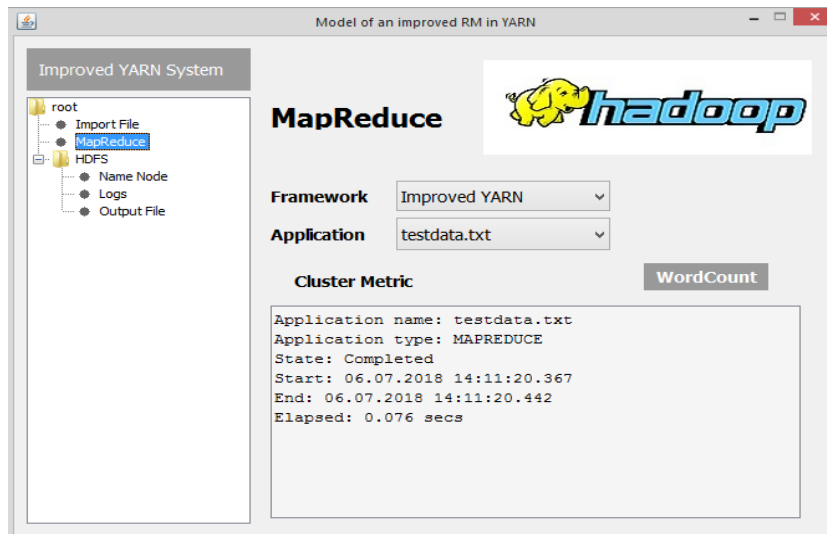


Figure 4.4: MapReduce Sub-system

### 4.2.2.3 HDFS Sub-system Implementation

The HDFS sub-menu has the Name Node, Logs and Output file list. The Name Node as described in Figure 4.5 shows number of files that have been uploaded on the server, total number of blocks and cluster size. It also gives description of which node is holding a block partition.

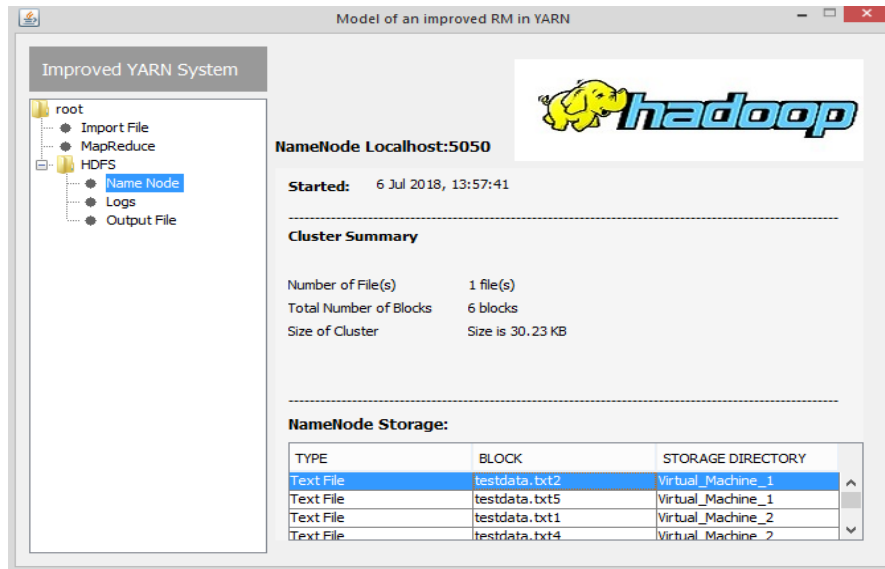


Figure 4.5: Name Node Sub-system

To view each block partitions, user double-clicks on the virtual machine holding that block partition. Example of a block partition is shown in Figure 4.6.

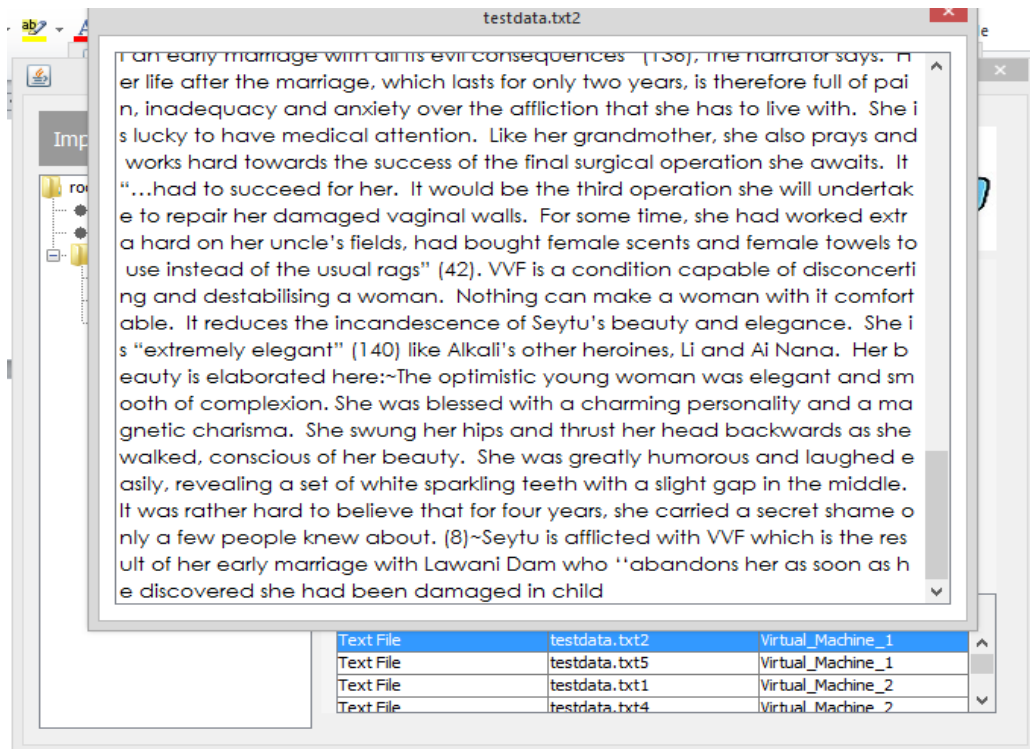


Figure 4.6: Block partition from Virtual Machine\_1

Logs in HDFS sub-menu shows intermediate mapreduce jobs during run time. It keeps track of the time for WordCount operation on each block partition. Viewing the log requires that yarn.log.aggregatios enable variable in yarn\_site.xml file be set.

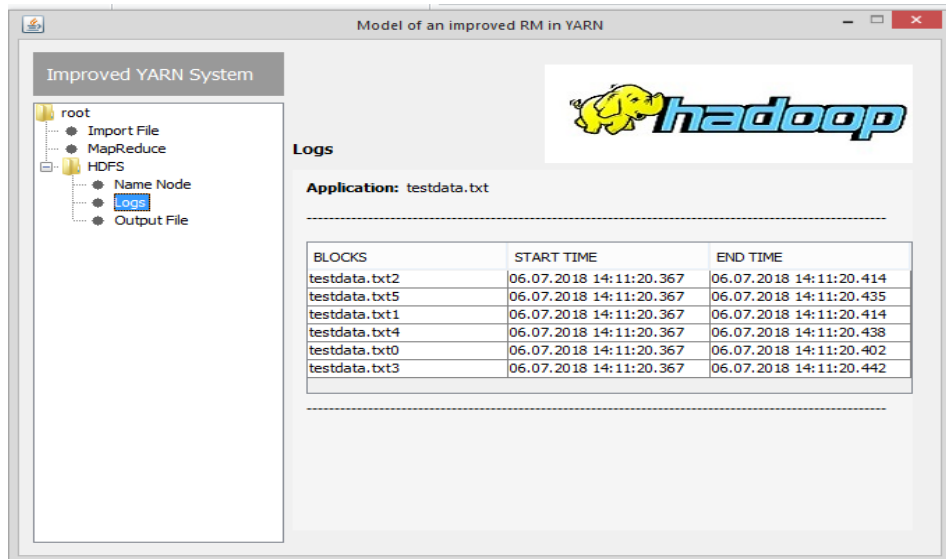


Figure 4.7: Improved YARN MapReduce Logs

The output file described in Figure 4.8a and 4.8b shows summary of completed job in the model. It shows the name of virtual machine with their corresponding block partitions and results of WordCount operation.

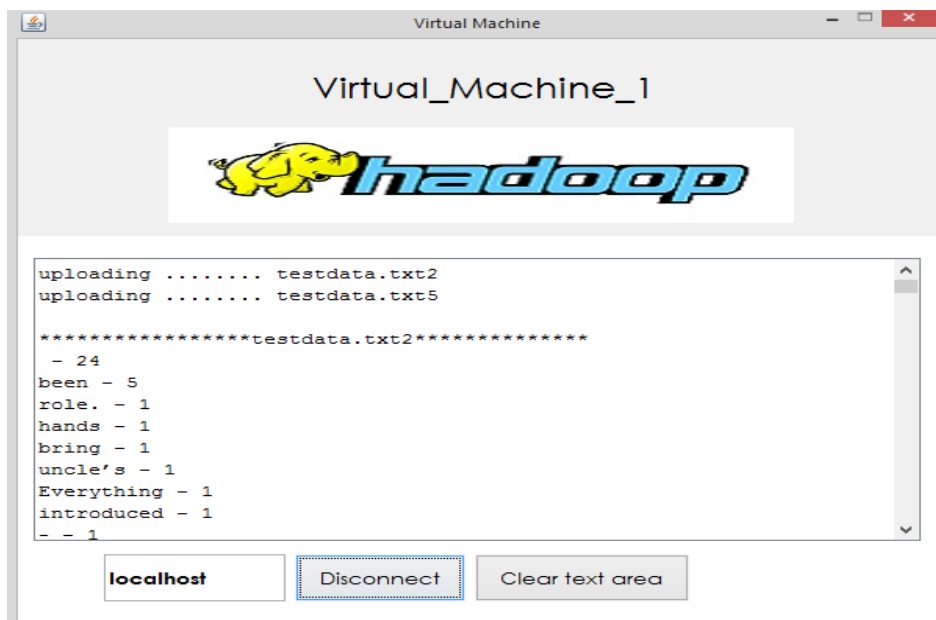


Figure 4.8a: Output from Virtual Machine\_1

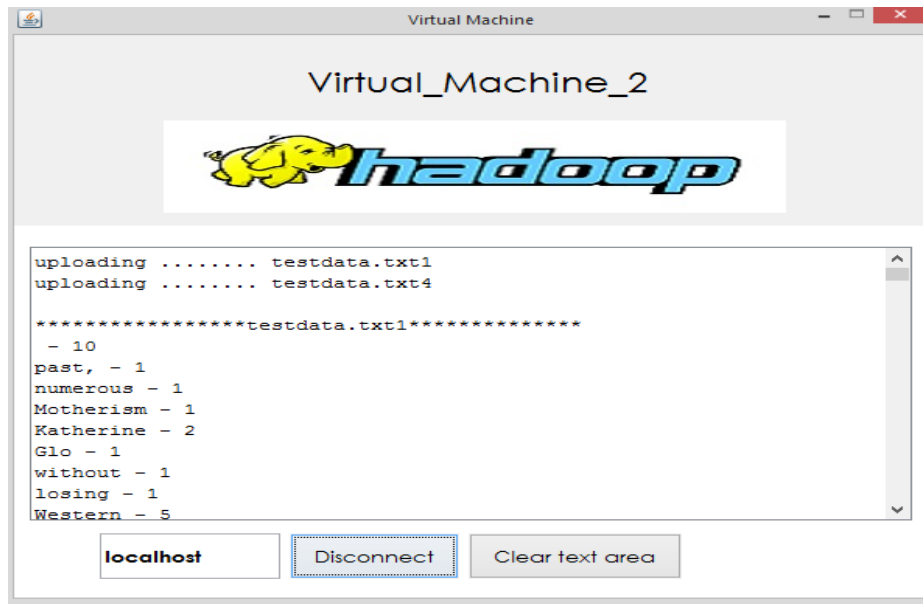


Figure 4.8b: Output from Virtual Machine\_2

### 4.3 Specifications

#### 4.3.1 Database Development Tool

The database development tool used for this system is the Computer Assisted Software Engineering (CASE) tool. There are a number of CASE tools that provides extensive functionality for database development. This new system uses Microsoft Visual Studio.Net Enterprise Architect Edition called Microsoft Office Visio. It is a forward and reverse engineering tool for databases and UML. It supports data dictionary to accompany entity relationship template and also supports name, data type, required, primary and notes properties.

#### 4.3.2 Database Design and Structure

Database design and structure for this system follows the four database development phases; conceptual data modelling, logical database design, distributed database design and physical database design. While conceptual data modelling and logical database design focuses on the information content of the database, distributed and physical database design focuses on efficient implementation of the system.

- i. Conceptual Data Model:- The conceptual data model for this system identifies the highest level relationship between entities. Features of this data model include the important entities and the relationship among them. No attribute or

primary/foreign key is specified in this model. The conceptual data model of this system is shown in Figure 4.9.

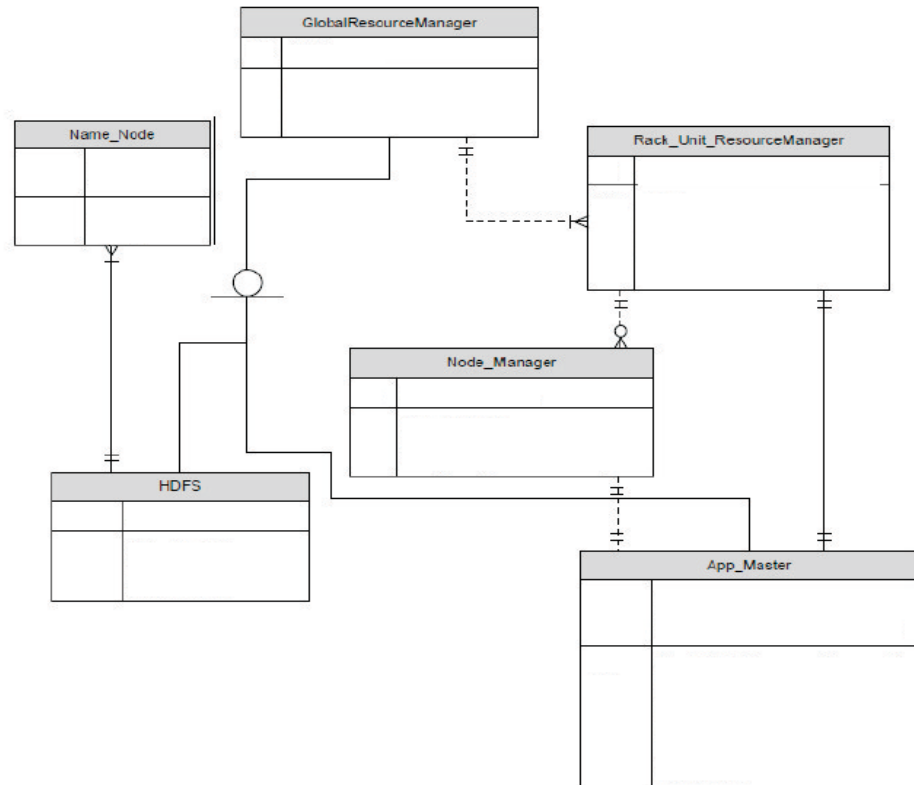


Figure 4.9: Conceptual Data Model showing relationship between entities. The rectangles (GlobalResourceManager, Rack\_Unit\_ResourceManager, HDFS, Data\_Node, Node\_Manager, App\_Master) represent entity types while label lines shows relationship between entities.

- ii. Logical Data Model:- The logical data model for this system is a more detailed description of the conceptual data model without regard to its physical implementation in the database. Features here include all entities and relationships among them, attributes for each entity are specified, primary and foreign keys (keys identifying relationship between different entities) are specified. To ensure that there is no redundancy, normalization is carried out in the table design constraints (dependencies among columns). Data requirements for the new model were used to produce ERD using Microsoft Office Visio. Figure 4.10 shows the logical data model for this system.

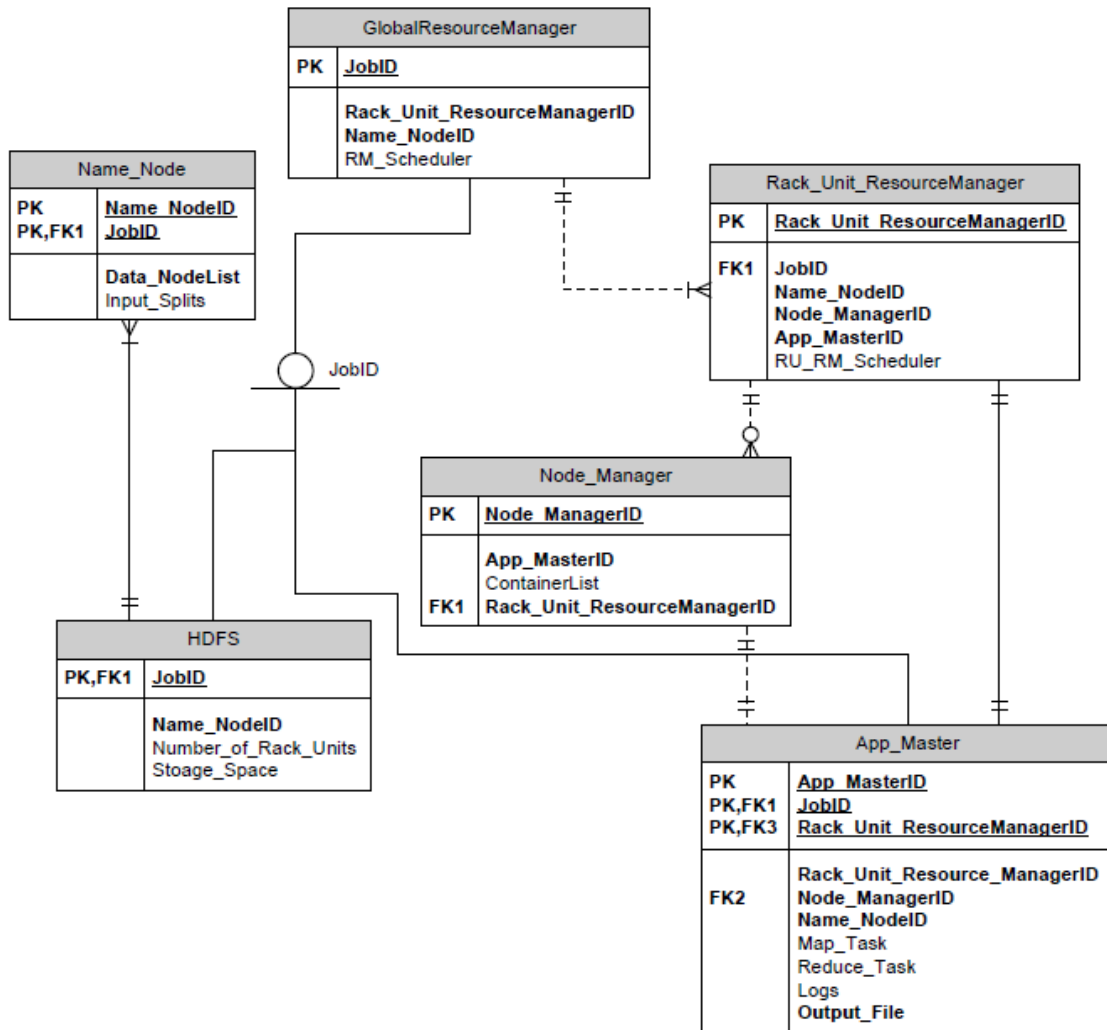


Figure 4.10: Logical data model showing attributes, primary and foreign keys for each entity

Attributes/properties of these entities are listed inside the rectangle. The diagram also shows primary keys for each entity and foreign keys that connects entities.

- iii. Distributed Data Model:- To ensure that reduced response time, improved availability of data and improved control is achieved, data are located in their appropriate positions. The actual input splits (blocks) for each file are resident in the appropriate data nodes within a rack. The system ensures that 2/3 of these blocks occupy different nodes in the same rack while the third block is in another node for fault tolerant situation. Also, 2/3 of all blocks are in a rack to ensure high bandwidth and low-latency during job execution. The metadata file, which keeps record of all data nodes and files they contain, is in a central location. This is done to ensure improved control of location and movement of data. With each RU\_RM



handling all compute nodes and their data, improvement on control and efficient response time was achieved. The index of files which is one of the important aspects of this phase has been implemented with the introduction of Name Node. This node keeps metadata of all data nodes in the cluster thereby, enhancing efficient implementation of this model.

- iv. **Physical Data Model:-** This phase describes how the data model was built in the database. It shows table structures with column names and data types used for database implementation. Figure 4.11 shows the physical data model for this system.

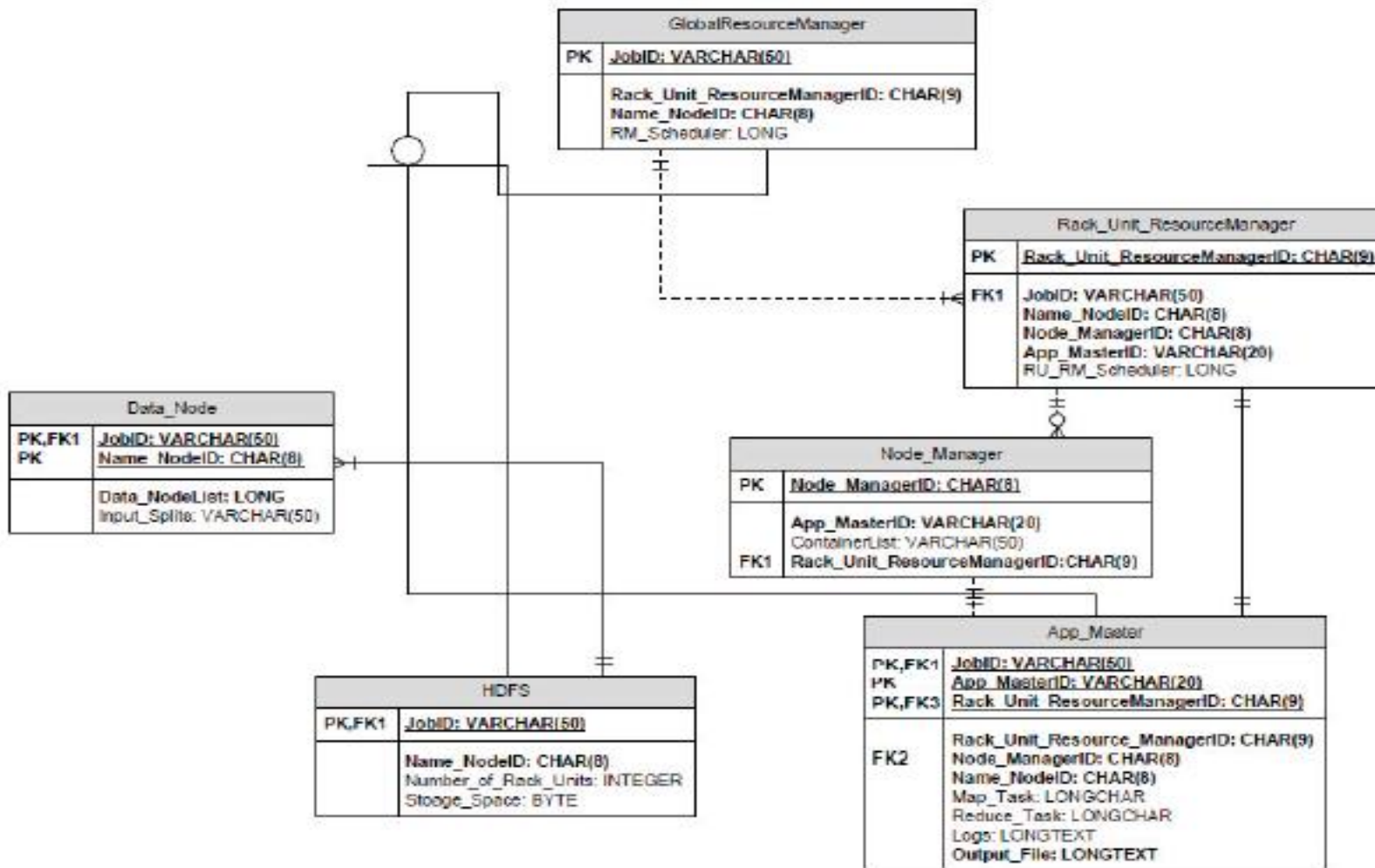


Figure 4.11: Physical data model showing data types

### 4.3.3 Program Module Specification

With the proposed rack unit resource manager layer introduced in the new model, the central resource manager is no longer responsible for monitoring of data nodes in the cluster. Resource request and container lease are sole responsibilities of each rack unit resource manager. Application Master now communicates corresponding rack unit resource manager for container lease and monitoring of job process. Going by the design objectives of this work, five different modules/components describe the whole architecture of the new model. These components are;

- Resource Manager allocation of job to appropriate Rack Unit Resource Manager.
- Rack Unit Resource Manager responsibility in executing job.
- Description of task execution and update in RU\_RM layer.
- Description of RU\_RM failure.
- RU\_RM description for joining the ring architecture

**Central RM allocation of job to appropriate RU\_RM:-** Upon job submission by client to RM, RM retrieves input splits from HDFS. With metadata information from Name Node on which rack holds these input splits, RM allocates job to the appropriate RU\_RM for processing. Figure 4.12 shows an event-tracing diagram describing how RM allocates job to appropriate RU\_RM.

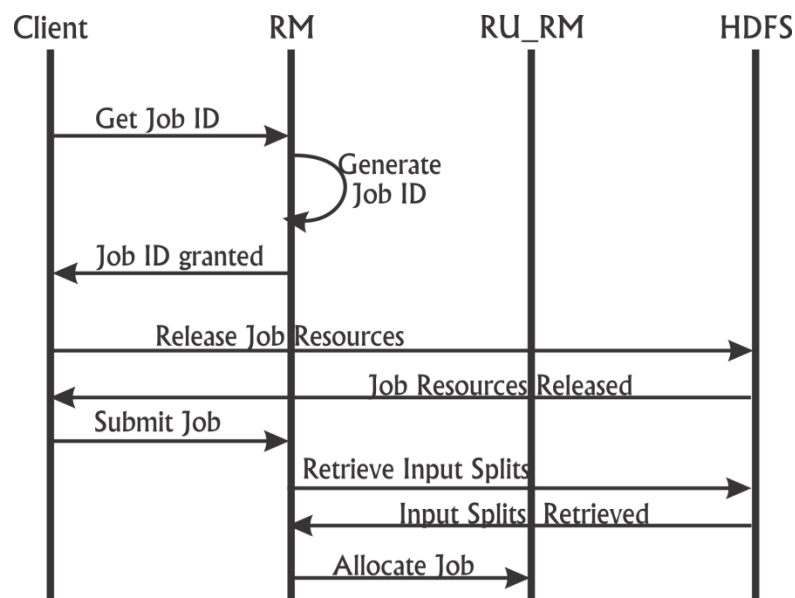


Figure 4.12: Event-tracing diagram for RM allocation of job to appropriate RU\_RM

**Rack Unit Resource Manager (RU\_RM) responsibility in executing job:-** RU\_RM module is responsible for executing job allocated to it by RM. RU\_RM is a per-rack framework only responsible for execution of jobs within the rack, except failure occur of its neighbouring unit. Figure 4.13 shows a scenario between RU\_RM, Application Master and the Node Manager during job execution.

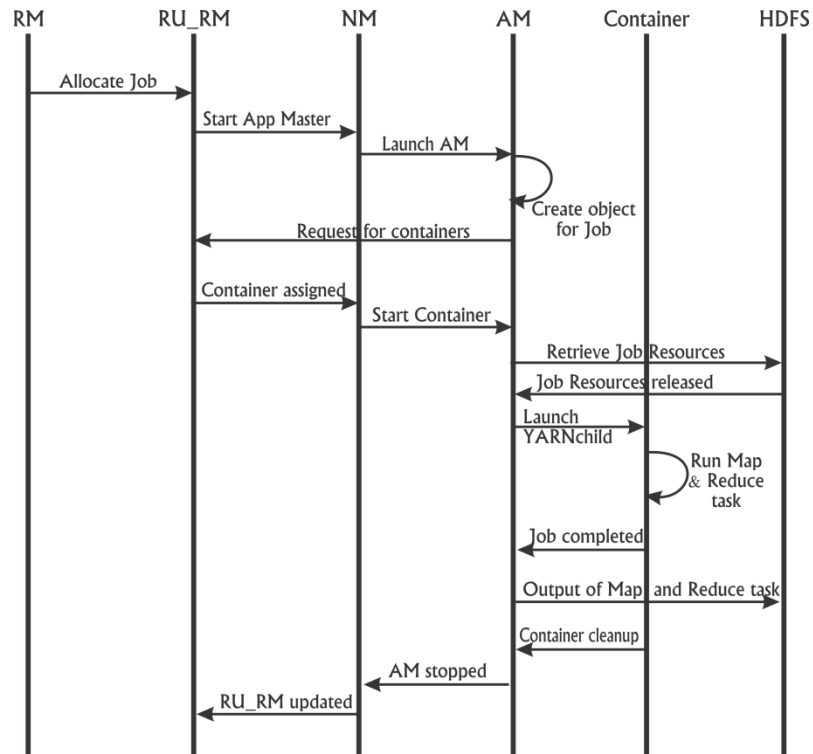


Figure 4.13: Event-tracing diagram for RU\_RM responsibility

**Description of task execution and update in RU\_RM layer:-** This module ensured that, for every job to be executed by any of the RU\_RM, its predecessor and successor must also be updated. This is important to ensure that if failure occurs, the RU\_RM's predecessor/successor can take over the responsibility of the failed unit. To describe this module, state machine diagram was used. State machine shows the behaviour that specifies sequence of states an object visits during its life time in response to events. Figure 4.14 shows the state machine diagram for this module.

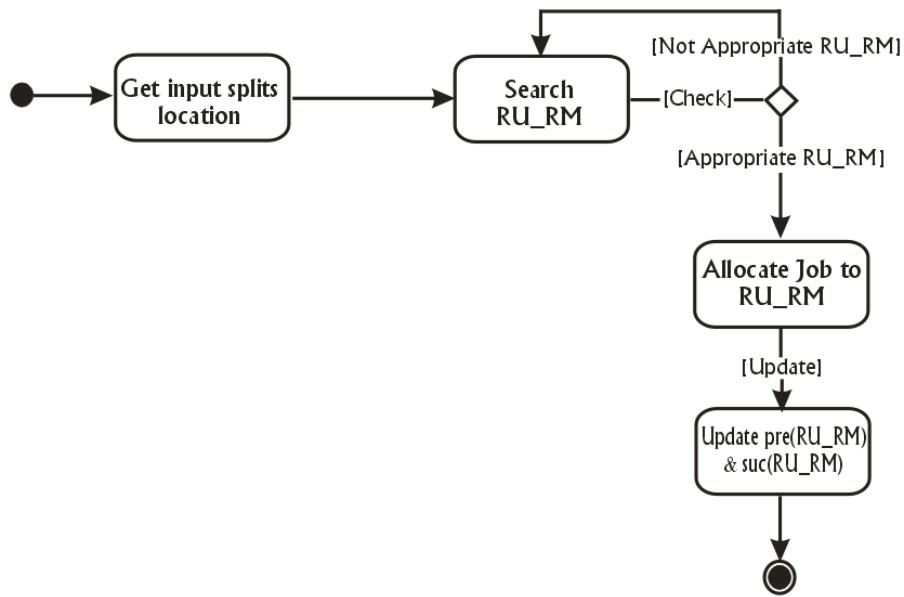


Figure 4.14: State machine diagram for execution of task in a rack

**Description of RU\_RM failure:-** This module ensures that at any point the predecessor/successor unit of any RU\_RM do not receive update, its means that such RU\_RM is not available. The predecessor/successor unit (as described in the objective of the design) therefore, takes over the responsibility of the failed RU\_RM. Figure 4.15 shows the state machine diagram for this module.

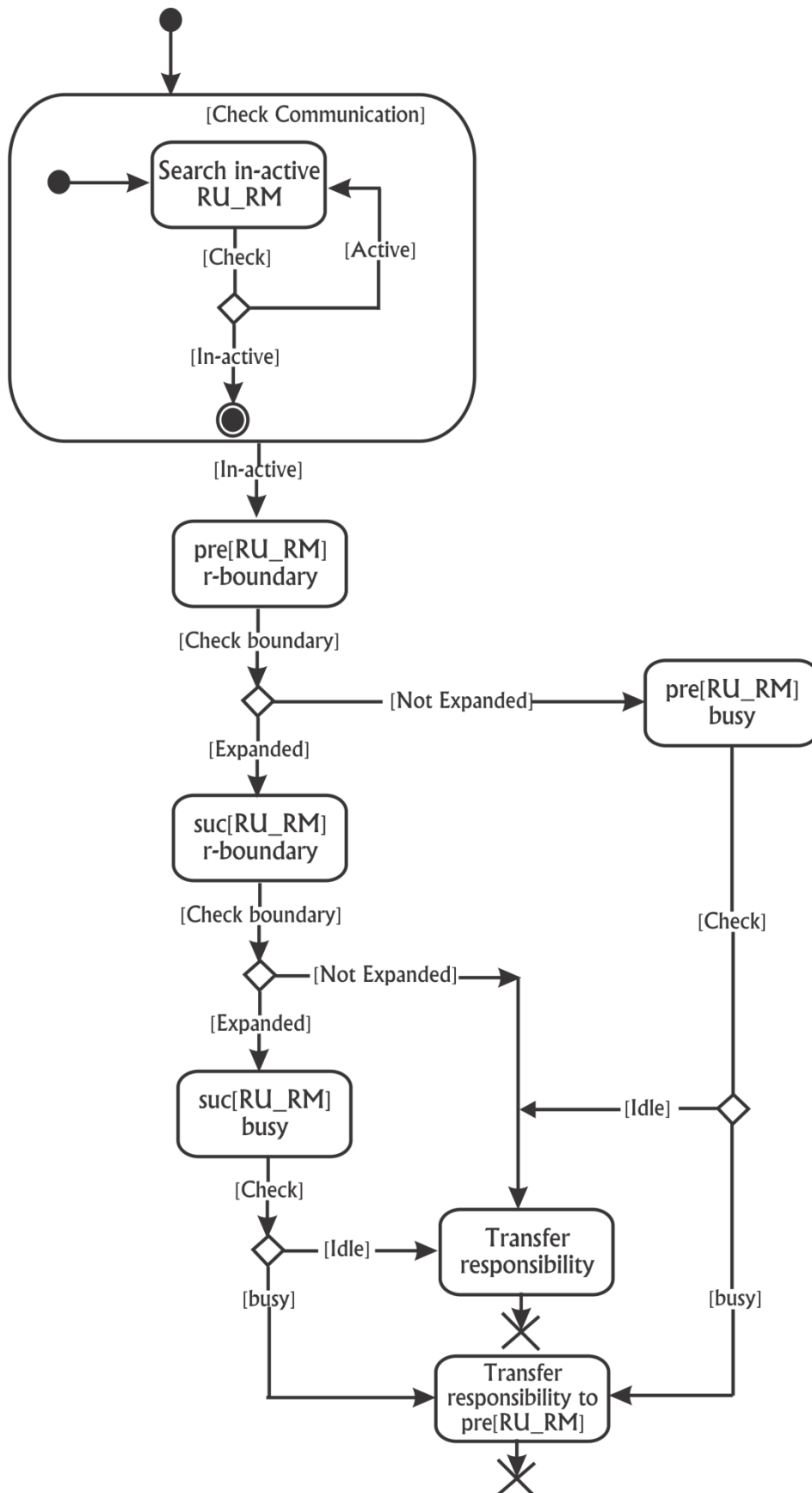


Figure 4.15: State machine diagram for RU\_RM failure

**RU\_RM description for joining the ring architecture:-** For a failed RU\_RM to join the ring after failure recovery, this module searches for RU\_RM that is not available in its position in the ring. It compares it with the RU\_RM ready to join the ring. If the comparison is correct, the RU\_RM joins the ring and its predecessor and successor notified. Its responsibility (which starts with predecessor key + 1 and ends with its key) is then release to the node. Figure 4.16 gives a state machine description of this module.

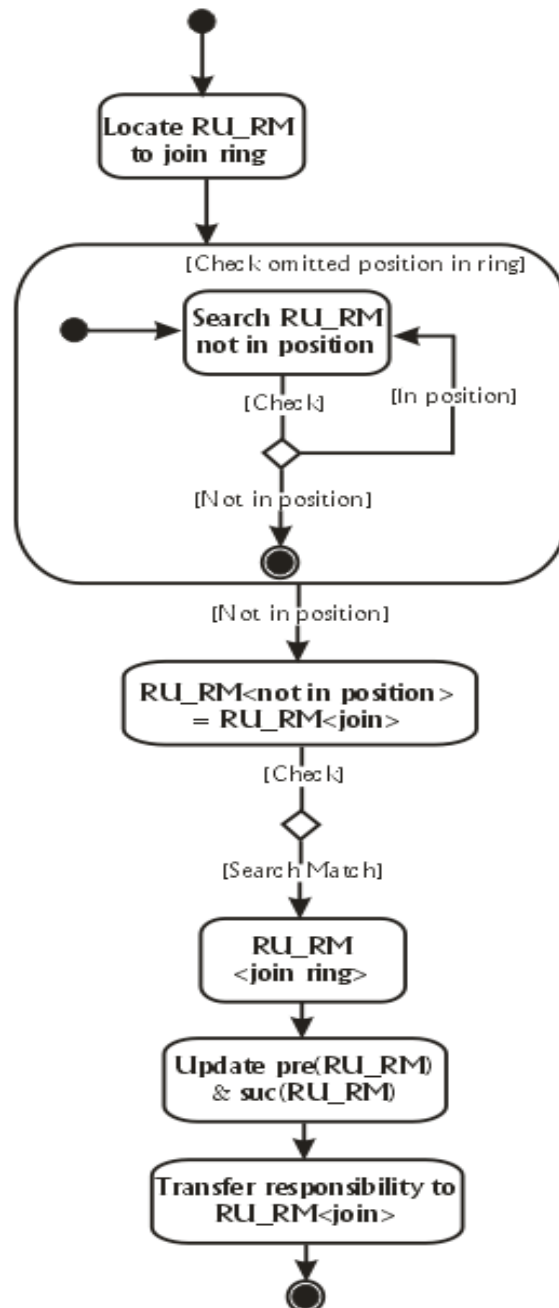


Figure 4.16: State machine diagram for RU\_RM joining the ring architecture

### 4.3.4 Input/Output Format

#### 4.3.4.1 Input Format

WordCount is the Hadoop benchmarking workload used for this research. WordCount is a typical two-phase Hadoop workload with the map task counting the frequency of individual words in a subset data file while the reduce task shuffles and gather the frequency of all the words. The input data for this research work therefore is any text file. Input format is described in Table 4.1.

**Table 4.1:** Typical input data format for map and reduce task

Hi, how are you
How is your job
How is your sister
How is your brother
What is the time now
What is the strength of Hadoop

To perform wordcount operation on the input data in Table 4.1, let us assume that the file name is file.txt and the size is 140MB. If 64MB is the size of each block to be stored in HDFS, the text file will be partitioned into 2blocks of 64MB and a block will contain 8MB as shown in Table 4.2.

**Table 4.2:** Block partitions for input data

Hi, how are you How is your job	64MB
How is your sister How is your brother	64MB
What is the time now What is the strength of Hadoop	8MB

The number of input splits for a file depends on the number of blocks you have for the job. Since there are three blocks in Table 4.2, we have three input splits and there are three corresponding mappers and reducers (one input split to a mapper and a reducer).



From the blocks in Table 4.2, block1 will be allocated to the first input split, block2 to the second and block3 to the third input split. Hadoop runs MapReduce jobs in the form of (key, value) pair. For the text file to be read and converted into (key, value) pair, there is an interface called RecordReader. The RecordReader reads each line in the text and converts it into (key, value) pair with the format (byteoffset, entireline). The 'byteoffset' represent row number in the text while 'entireline' is the whole text in the line. For example, to read block1 in the file.txt, (byteoffset, entireline) will be (0, hi how are you). The RecordReader gets the next byteoffset by reading the number of characters in the first row. The first row has the text 'hi\_how\_are\_you\_' = 15characters including spaces. Hence, the next (byteoffset, entireline) = (16, how is your job).

#### 4.3.4.2 Output Format

The generated data from mapper form another (key, value) pair which is referred to as intermediate data (output). Once these intermediate data are generated, reducer function is triggered to combine all intermediate data into final output. Shuffling phase combines all single key to produce these intermediate data described in Table 4.3.

**Table 4.3:** Format of intermediate output from MapReduce task

Hi, [1]
how, [1,1,1,1]
are, [1]
you [1]
is [1,1,1,1,1]
your [1,1,1]
job [1]
sister [1]
brother [1]
the [1,1]
time [1]
now [1]
strength [1]
of [1]
Hadoop [1]

The reducer has RecordWriter. Once the intermediate data has been shuffled, the reducer will sort it and pass it to RecordWriter which produces the output file into HDFS. Table 4.4 describes output file format.

**Table 4.4:** Output file format

Hi, 1
how, 4
are, 1
you, 1
is, 5
your, 3
job, 1
sister, 1
brother, 1
the, 2
time, 1
now, 1
strength, 1
of, 1
Hadoop, 1

Screen display of the output file shows the application name, application type, state of execution, finished time, total processing time and a link to where the output file is stored.

#### **4.3.5 Overall Object Diagram for the New System**

The object diagram in Figure 4.17 shows a snapshot of the detailed state of this system as point of task execution. The diagram encompasses objects and their relationship at point of execution.

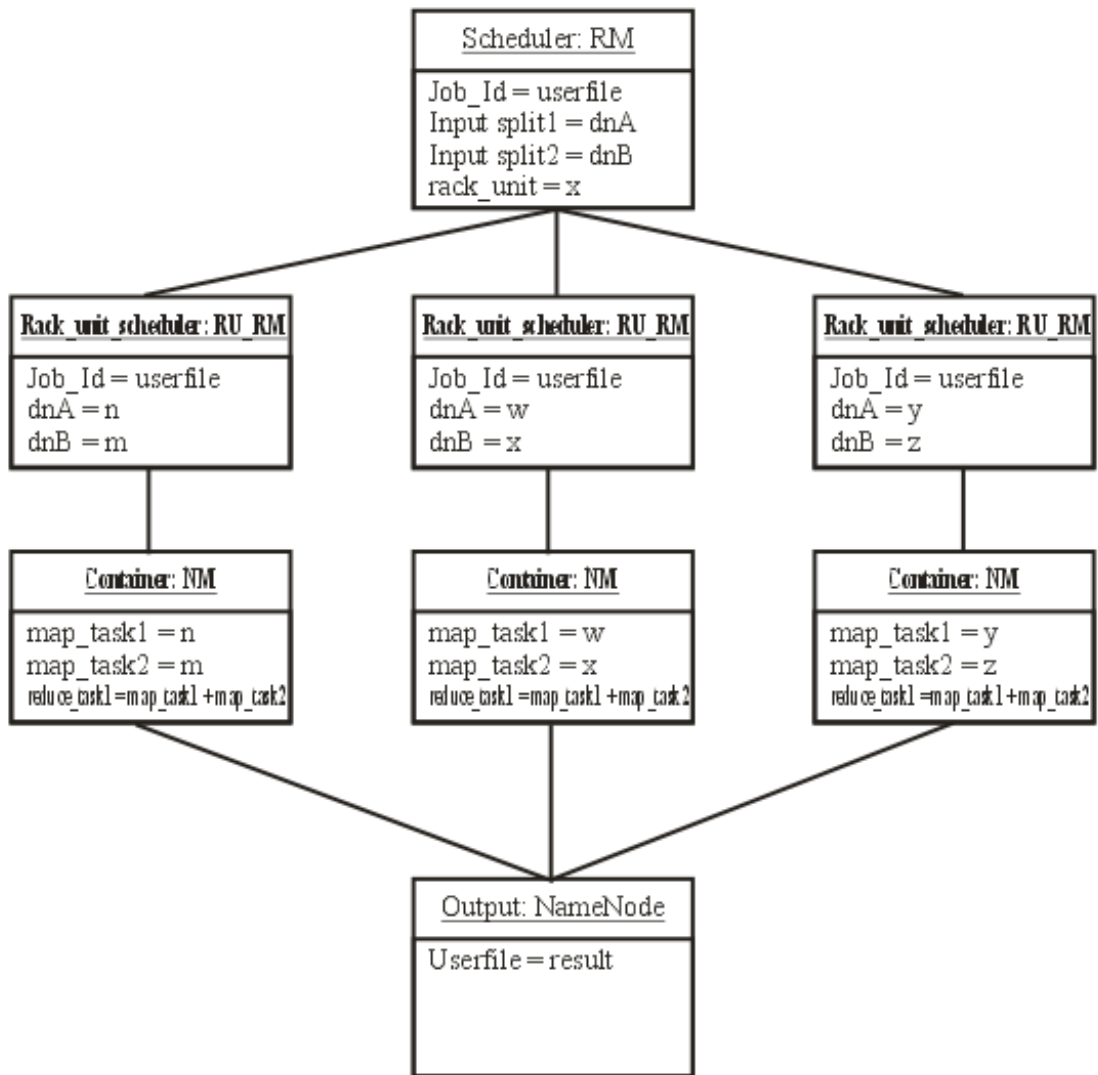


Figure 4.17: Object diagram of the new system at point of task execution

Figure 4.17 show that the global resource manager only allocates job to appropriate rack unit resource manager. Rack unit resource managers execute jobs simultaneously, each executing jobs within its rack. This approach allows for faster response time as in contrast to the existing approach where the global resource manager is responsible for job execution in the whole cluster.

#### 4.3.6 Algorithm

Going by the design objectives of this work, five different algorithms described the whole architecture of the new model. Algorithm 3.1 described the work of Central Resource Manager which is allocation of job to appropriate RU\_RM. Once allocation is

done, all responsibilities and processes involved in the execution of that job are controlled by the corresponding RU\_RM.

---

**Algorithm 3.1:** Allocation of job to appropriate RU\_RM

---

**upon event** <job submission>  
    client receive Job ID  
    client copies Job resources to HDFS  
    client submits job to RM  
    RM retrieves input split from HDFS  
    RM allocate job to appropriate RU\_RM  
**end event**

---

The central Resource Manager has a pluggable scheduler whose sole responsibility is to allocate jobs to the appropriate RU\_RM. It is a pure scheduler in the sense that it performs no monitoring or tracking of jobs/applications status, offering no guarantees to restart failed task either due to job or hardware failure. The scheduler performed its function based on the metadata received from Name Node. Once job has been allocated to the appropriate RU\_RM, RM is free from any other responsibility for that job. RM uses a push-based scheduling technique where it pushes responsibility to appropriate RU\_RM for execution. No periodic heartbeat mechanism is required between RM and RU\_RM hence; RM can manage a lot of jobs and achieves better cluster utilization.

Algorithm 3.2 shows scenario between rack unit resource manager, application master and node manager during job execution. RU\_RM is a per-rack resource manager as compared to the per-cluster resource manager in the existing framework. 2/3 of the complete data block for a file is stored in the same rack. This is to ensure that AM do not need monitor job across racks in the cluster. This process ensured lower job latency as opposed to the existing system where AM monitors job across racks in the cluster.

---

**Algorithm 3.2: RU\_RM Responsibility**

---

**upon event** <execute job>

RU\_RM receive job allocation from RM  
RU\_RM scheduler prompt Node Manager to launch AM for the job  
AM creates object for the job  
AM retrieves and creates lmap per split from input splits  
AM request for container from RU\_RM  
RU\_RM through NM launches YARNchild  
YARNchild retrieves all job resources from HDFS  
YARNchild runs map and reduce task  
YARNchild sends update of execution to AM  
AM aggregates and send update to RU\_RM through NM  
AM send completed task to HDFS  
AM and task container clean up working state and RU\_RM notified

**end event**

---

Algorithm 3.3 ensured that, for every job to be executed by any of the Rack Unit Resource Manager (RU\_RM), its predecessor and successor must also be updated. This is to make sure that if failure occurs, any of predecessor/successor can take over the responsibility of the failed RU\_RM.

---

**Algorithm 3.3: Execution of Task**

---

**upon event** <execute task>

RU\_RM = node(n) // n is 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**

---

Algorithm 3.4 ensured that once the predecessor or successor of any RU\_RM does not receive update, it therefore means that such RU\_RM is not available hence, the predecessor or successor (depending on which is idle) takes over the responsibility of the failed RU\_RM.

---

**Algorithm 3.4:** 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
```

---

The RU\_RM ring architecture for this design shown in Figure 4.17 helps to monitor failure at the RU\_RM layer.

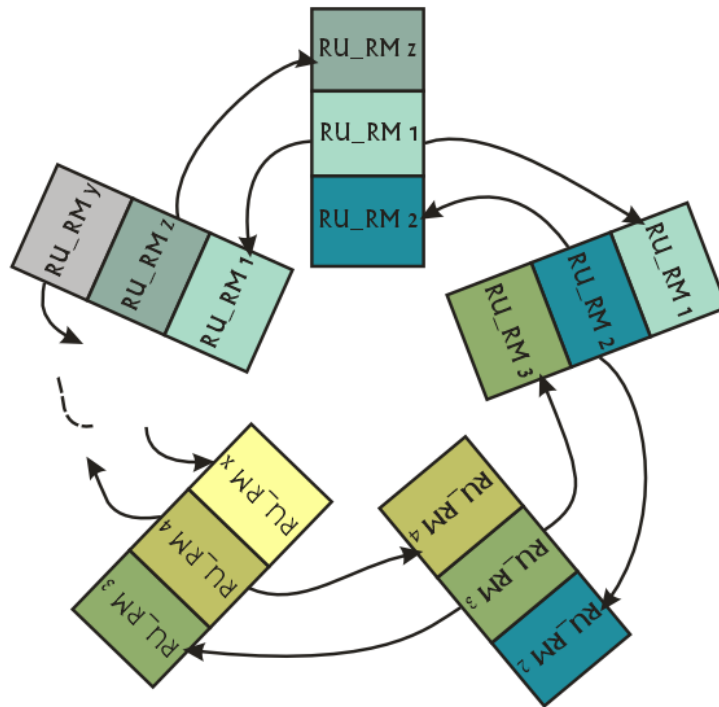


Figure 4.18: Ring architecture for RU\_RMs in the new model

From Figure 4.18, the architecture ensured that data items (resources) are replicated on neighbouring peers (RU\_RMs) on the ring. Each of RU\_RM holds resources for which it is directly responsible and also hold resources for RU\_RM proceeding and succeeding it on the ring.

Each RU\_RM on the ring updates its predecessor and successor. If after 3seconds of execution, no signal is obtained from any of RU\_RM, such RU\_RM is considered dead. Boundary of the peer (RU\_RM proceeding/succeeding it) will be extended. This process is described in Figure 4.19.

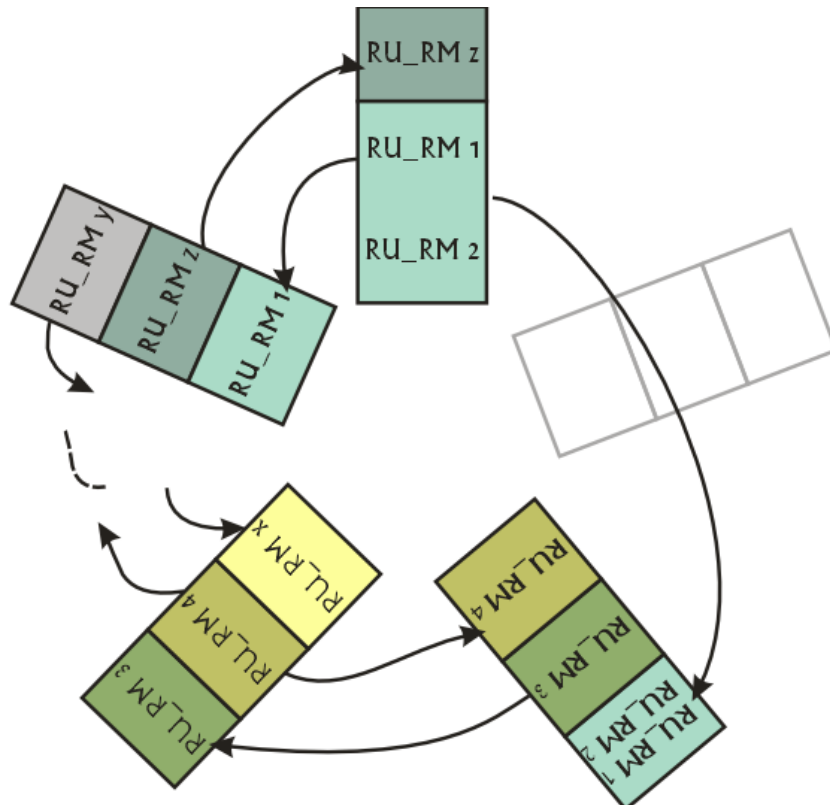


Figure 4.19: Failure scenario in the ring architecture of RU\_RMs

As described in Figure 4.19, if RU\_RM2 fails, RU\_RM1 or RU\_RM3 boundary of resource control will be expanded. Expansion of boundary is dependent on two conditions;

- a. Whether RU\_RM proceeding/succeeding the failed peer (RU\_RM) has already been expanded due to failure of other neighbouring node.
- b. Whether the peer (RU\_RM) proceeding/succeeding the failed peer is idle or busy.

Two scenarios are possible; best case scenario and worst case scenario.

**Best Case Scenario:-** From Figure 4.19, if RU\_RM1 and RU\_RM3 have not been expanded due to failure of their neighbour peers, and none of them is idle, the peer (RU\_RM) preceding the failed peer takes over the responsibilities of the failed RU\_RM. At this point, update of both RU\_RM1 and RU\_RM2 will be done on RU\_RM3 as shown in Figure 4.19.

**Worst Case Scenario:-** A worst case scenario is the situation where two successive peers fail intermittently as described in Figure 4.20. At this point, a back-up will be made to allow for execution before recovery from failure is performed. For instance, if



from Figure 4.20 RU\_RM3 also fails. RU\_RM4 resource control boundary will be expanded. In this case, only RU\_RM1 will hold resources for RU\_RM2. If however, RU\_RM1 is the peer that fails, resource control boundary of RU\_RMz is expanded while RU\_RM3 takes control of resources in RU\_RM2.

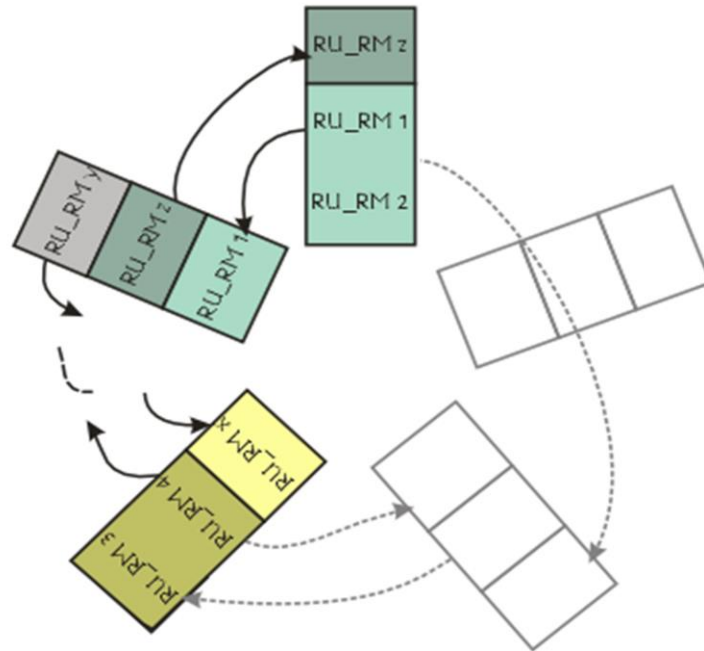


Figure 4.20: Failure of two successive peers (RU\_RMs) in the ring

In either of the two case scenarios mentioned, once a failed peer recovers from failure and joins the ring, its resources from the peer holding it will be released to it (recovered peer). Also, back-up of resources for peers preceding/succeeding it begins.

Upon event join as obtained in Algorithm 3.5, the system searches for RU\_RM that is not available in its position in the ring. It compares it with the RU\_RM ready to join the ring. If the comparison is correct, the RU\_RM joins the ring and its predecessor and successor notified. Its responsibility (which starts with predecessor key + 1 and ends with its key) is then release to the node.

---

**Algorithm 3.5:** 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**

---

### 4.3.7 Data Dictionary

Data dictionary provides detailed information about data elements and their meanings and allowable values. It gives information about each attribute of a data model. Data dictionary for this system are obtained from six entities in our data model as shown in Table 4.5 to Table 4.10.

**Table 4.5:** Global Resource Manager

S/N	Column Name	Data Type	Constraints (Key)	Notes
1.	JobID	VARCHAR(50)	Primary	It identifies RM to HDFS for input splits.
2.	RU_RM_ID	CHAR(9)	Not Null	It identifies RM to appropriate RU_RM for job allocation.
3.	Name_NodeID	CHAR(8)	Not Null	Keep track of input splits for each job
4.	RM_Scheduler	LONG	Not Null	Schedules jobs based on RU_RM_ID information.

**Table 4.6:** Hadoop Distributed File System (HDFS)

S/N	Column Name	Data Type	Constraints (Key)	Notes
1.	JobID	VARCHAR(50)	Primary, Foreign	Track job input splits on NameNodeID and candidate key that identifies RM for DataNodeList and input splits
2.	Name_NodeID	CHAR(8)	Not Null	For DataNodeList and Input splits.
3.	Number of RUs	INTEGER	Not Null	Keeps track of number of racks in cluster
4.	Storage_Space	BYTE	Not Null	Keeps track of storage space in cluster

**Table 4.7:** Rack Unit Resource Manager

S/N	Column Name	Data Type	Constraints (Key)	Notes
1.	RU_RM_ID	CHAR(9)	Primary	Identifies appropriate RU_RM from RM. Candidate key for RM.
2.	JobID	VARCHAR(50)	Foreign	Identifies job allocation traced to HDFS through RM
3.	Name_NodeID	CHAR(8)	Not Null	Identifies DataNodeList and input splits from HDFS through RM
4.	Node_ManagerID	CHAR(8)	Not Null	Is of RU_RM for task monitoring
5.	App_MasterID	VARCHAR(20)	Not Null	Is of RU_RM for task execution
6.	RU_RM_Scheduler	LONG	Not Null	Schedules job input splits to appropriate NodeManager

**Table 4.8:** Name Node

S/N	Column Name	Data Type	Constraints (Key)	Notes
1.	Name_NodeID	CHAR(8)	Primary	Identifies HDFS for data node list and input splits
2.	JobID	VARCHAR(50)	Primary, Foreign	Identifies job input splits and respective data nodes traced to HDFS through RM
3.	Data_NodeList	LONG	Not Null	Is for Name_Node for list of data nodes in cluster
4.	Input Splits	VARCHAR(50)	Not Null	Is for Name_Node for list of input splits for a job.

**Table 4.9:** Node Manager

S/N	Column Name	Data Type	Constraints (Key)	Notes
1.	Node_ManagerID	CHAR(8)	Primary	Identifies App_MasterID in connection with RU_RM_ID
2.	App_MasterID	VARCHAR(20)	Not Null	Identifies each job for a single Application Master
3.	ContainerList	VARCHAR(50)	Not Null	For Node Manager to track containers
4.	RU_RM_ID	CHAR(9)	Foreign	Identifies RU_RM for job monitoring

**Table 4.10:** Application Master

S/N	Column Name	Data Type	Constraints (Key)	Notes
1.	App_MasterID	VARCHAR(20)	Primary	Identifies job allocation in connection with RU_RM_ID
2.	RU_RM_ID	CHAR(9)	Primary, Foreign	Identifies RU_RM_ID directly via App_MasterID
3.	JobID	VARCHAR(50)	Primary, Foreign	Identifies allocated Job directly via App_MasterID
4.	Node_ManagerID	CHAR(8)	Foreign	Identifies Node Manager for task monitoring
5.	Name_NodeID	CHAR(8)	Not Null	Keeps record of processing on Data Nodes
6.	Map_Task	LONGCHAR	Not Null	Is of App_Master for map tasks
7.	Reduce_Task	LONGCHAR	Not Null	Is of App_Master for reduce task
8.	Logs	LONGTEXT	Not Null	Is of App_Master for intermediate results
9.	OutputFile	LONGTEXT	Not Null	Is of App_Master for final result

#### 4.4 System Flowchart

System flowchart for this system is shown in Figure 4.21.

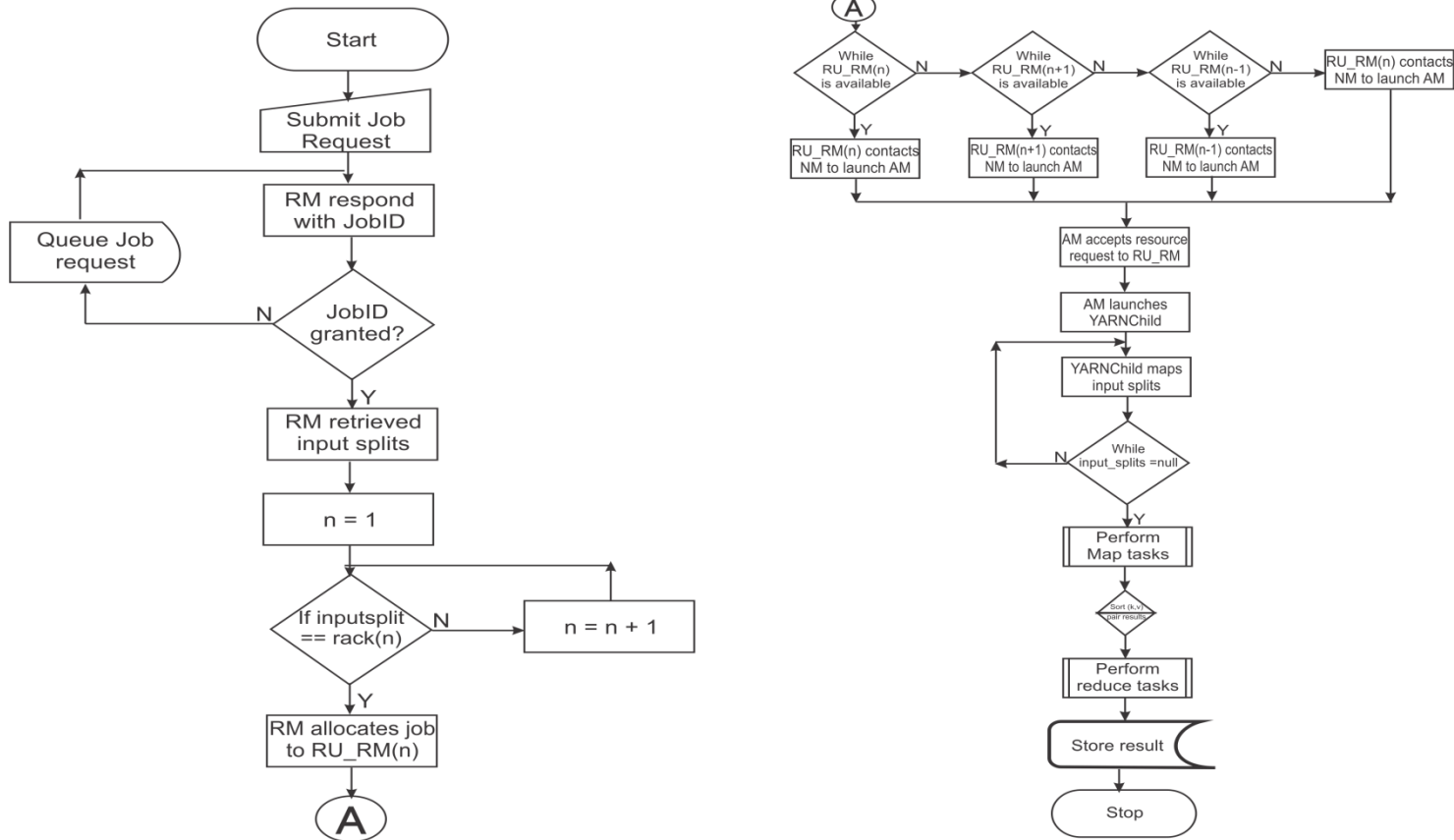


Figure 4.21: System flowchart for the new model

## **4.5 System Implementation**

System requirements for implementing the new model described features and necessary specifications required to run this application. Requirements for the new system are hardware and software specifications.

### **4.5.1 Hardware Requirements**

Hardware requirements for setting up this new rack aware resource management application in Hadoop include:

- i. RAM size – minimum of 4GB
- ii. Hard disk size – minimum of 50GB
- iii. Processor – Intel Core 2 dual/quad/hex/octa or higher end 64bit processor with operating frequency of 2.5GHz or higher.
- iv. Graphic Adapter – NVIDIA

### **4.5.2 Software Requirements**

Software requirements for the new system are;

- i. Java Development Kit (JDK) 1.6 or later version
- ii. Cygwin software – packages to install are Open ssl, Open ssh, tcp wrappers and diffutils.
- iii. Install Virtual Machine ware (VMware)
- iv. Windows SDK – a .NET framework Software Development Kit (SDK) for Microsoft.
- v. Maven Protocol Buffers Plugin – a tool that generate Java source files from .proto (protocol buffer definition).

## **4.6 Program Development**

Program development deals with various tools, methods and procedures required for controlling the complexity of software development, management and its maintenance. This will be considered under choice of programming environment and justification for the language used.

#### **4.6.1 Choice of Programming Environment**

Integrated Development Environment used for this system is NetBeans IDE. Programming language used is Java (Oracle JDK 1.7). Other supported environment used to run this application is Windows SDK, which is a .NET Framework Software Development Kit (SDK) from Microsoft that contains documentation, header files, libraries and tools required for developing this system. To successfully run this application on Windows, Maven Protocol Buffers Plugin, which is a tool that helps generate Java source files from .proto (protocol buffer definition) was used. Cygwin was also installed with packages like openssh, openssl, tcp wrappers and diffutils.

#### **4.6.2 Language Justification**

Java programming language was used for this system because Hadoop core architecture was also developed using Java. Hence, to modify the architecture so as to suit the design objectives for this application, Java language is best fit. Though there are IDEs for Java development, NetBeans IDE supports scripting language like PHP. It also supports Maven and is cross-platform that runs on Microsoft Windows, Mac OS X, Linux, Solaris and other platforms supporting a compatible JVM. Windows SDK was installed to help build windows native component (winutils.exe). Cygwin is a large collection of GNU and open source tools that provides functionality similar to Linux distribution on Windows. It is needed to run scripts supplied with Hadoop because they are all written for Linux platform.

### **4.7 System Testing**

Thorough test plan and proper execution helps deliver quality software. Test plan act as a road-map for system testing within a project. It describes overall test strategy drawn up for testing components of a system. This section details test plan and test data used for the new system.

#### **4.7.1 Test Plan**

The test plan for this new model started with components test. Each component was tested to see if it meets its design objectives. All components were later coupled to form a whole system. This was done to simplify error localization and to ensure interleaving



of processes. The first component developed was Global Resource Manager. Its relationship with client on the local machine and cloud server used for storage were tested to ensure jobs are granted appropriate JobID. This process was the first test plan carried out as shown in Figure 4.22.

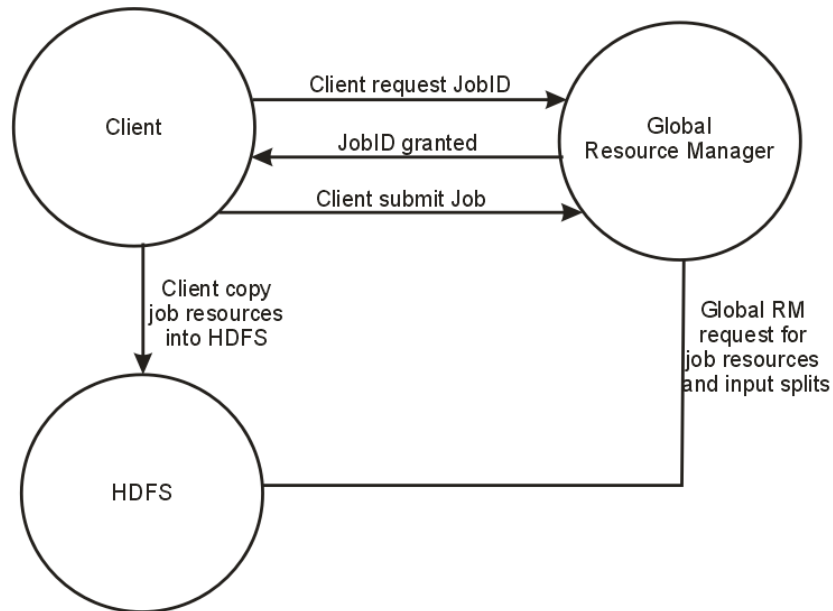


Figure 4.22: Module and Unit Code Test(1) showing interaction between global resource manager, client on local machine and HDFS

The other three components developed and tested are Rack Unit Resource Manager, Node Manager and Application Master. These three components work exactly like components in the existing system. Hence, they are all re-usable components picked from existing architecture. The interaction between these three components is described in Figure 4.23.

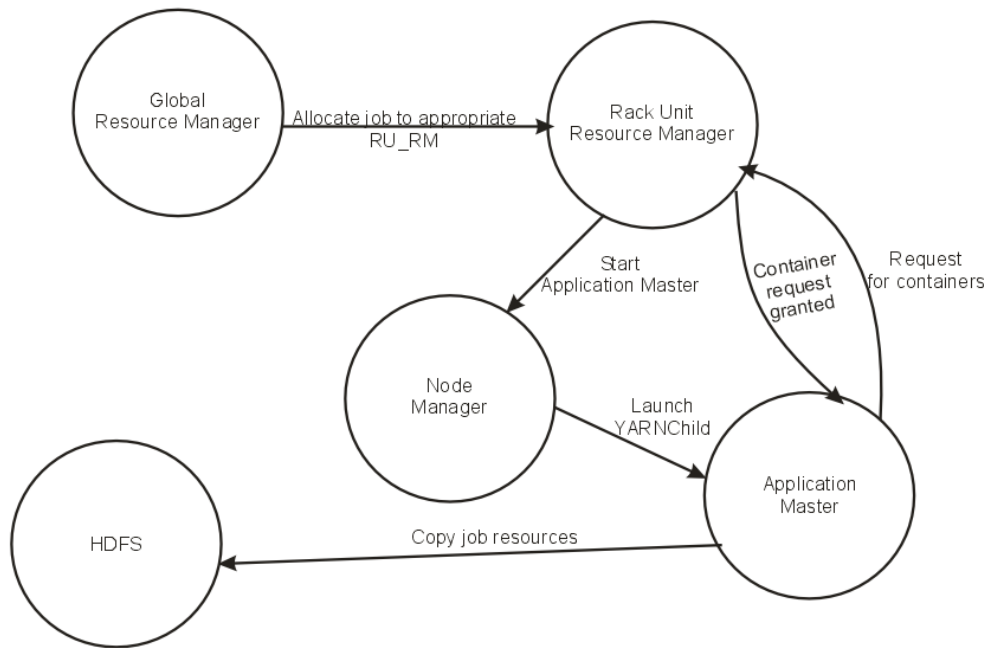


Figure 4.23: Module and Unit Code Test (2) showing interaction between Rack Unit Resource Manager, Node Manager and Application Master

The novel ring architecture of rack unit resource managers was also tested to ascertain continuous execution of jobs in case any rack unit resource manager fails. This test process is shown in Figure 4.24.

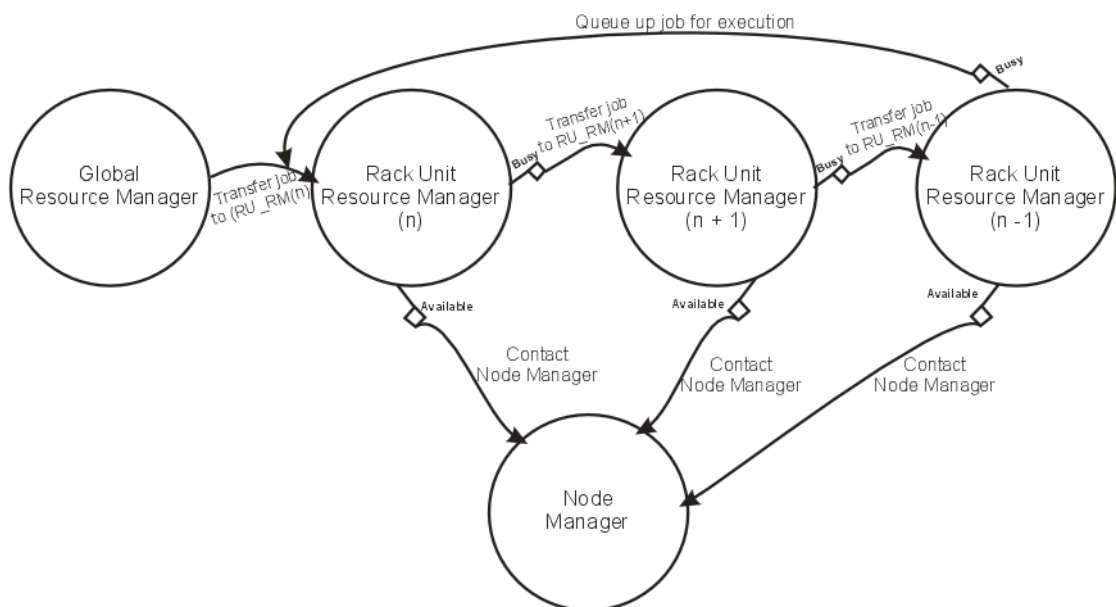


Figure 4.24: Module and Unit Code Test (3) showing interaction between Rack Unit Resource Managers

Figure 4.25 describe how the three module and unit code test were coupled and integration/acceptance test carried out to ensure that the whole model meets its design objectives.

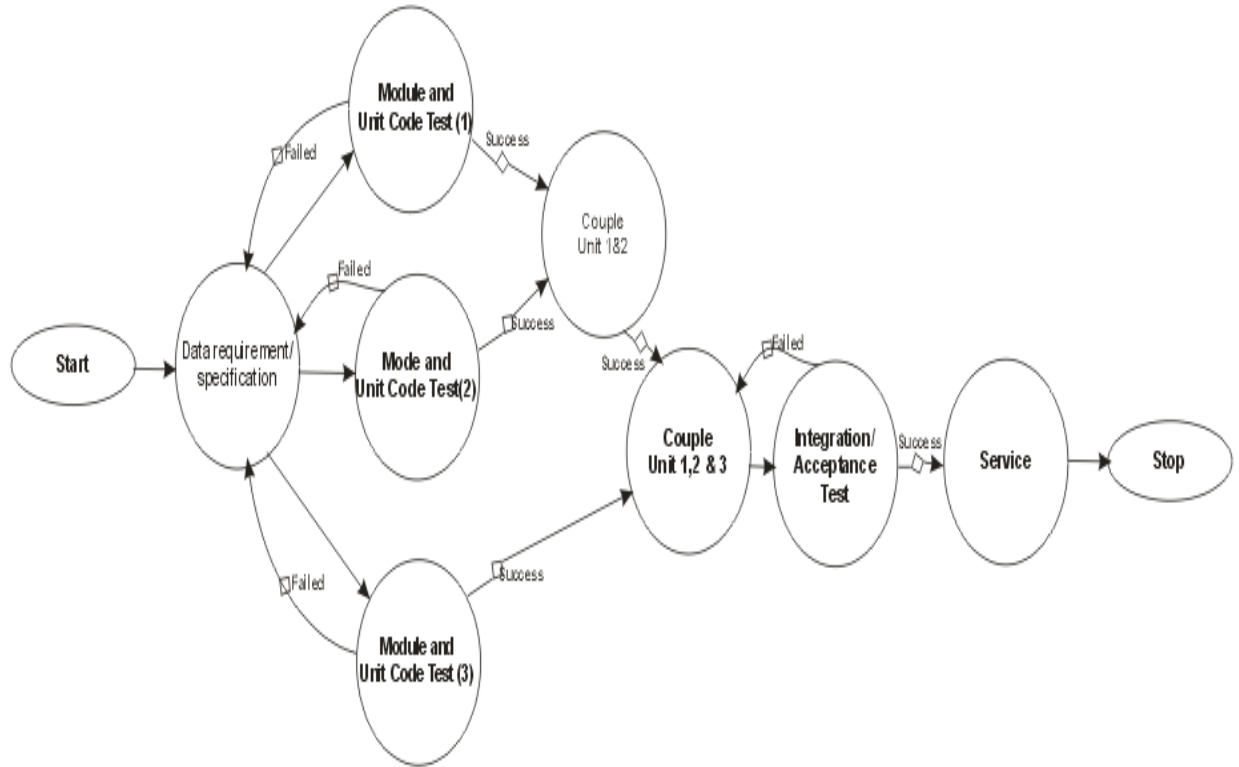


Figure 4.25: Whole system test plan

#### 4.7.2 Test Data

One of the popular workload for Hadoop benchmark is WordCount. The workload helps count the occurrence of each word in a text file. The process is to see how efficient and fast this operation will be, so as to determine processing and response time possible for tasks run in Hadoop framework. For this model therefore, WordCount was used as evaluation metric to determine processing and response time between the new and existing system. Table 4.11 shows the text file used for this evaluation. It is important to note that, any text file can run on this application for WordCount MapReduce task.

**Table 4.11:** Test Data

Women writers and readers have always had to work against the establishment. Aristotle, in his time, declared that the female is a female because she lacks certain qualities that the male, the supposed perfect being, has. St. Thomas Aquinas believed that a woman was the imperfect form of man. In pre-Mendelian days, men regarded their sperm as the active seed which gave form to the waiting ovum of the woman which lacked identity till it received the male's sperm. All these were established formations by the society that found themselves in some of the writings of the classical period. Throughout its long history, feminism has sought to disturb these established trends and complacent conventions of such cultures rooted in patriarchy. Although the word feminism may only have come into English usage in the 1890s, women's conscious struggles to resist patriarchy go much earlier than that. Feminism as we have it today started developing only in the 18<sup>th</sup> century. Hence, feminism, on a general note, is basically concerned with the struggle for the emancipation of women and the expression of issues regarding the ordeals of women in society. The early activities of feminism, which largely surrounded issues concerning suffrage, are commonly referred to as first wave feminism, which started in England. At this point, feminists like Mary Wollstonecraft and Virginia Woolf articulated what it meant to be a woman in the society and worked towards changing the limitations imposed upon women. In the United States of America, feminists like Margaret Fullers in 1850 and Olive Schreiner in 1848 respectively, advocated women rights. Through the works of Virginia Woolf and Simone de Beauvoir, the first wave of feminism challenged the conventions of their days and paved way for the emergence of the second wave. By the 19<sup>th</sup> century, second wave feminism began to build on the successes recorded by the first wave feminists. This was when feminism was developed into theories and strategies aimed at giving the women a voice in the society and a place equal to that of men. The efforts of the first and second wave feminists laid the foundation for the emergence of contemporary feminism. During the second wave, Michele Barrette, through her book, *Women's Oppression Today: Problems in Marxist Feminist Analysis*, announced what is today known as Marxist Feminism. In Britain, this brand of feminism was already popular in the late 1960s and the 1970s. It sought to extend Marxism's analysis of class struggles into the woman history of material and economic oppression, and especially, how the family and the woman's domestic labour were constructed by and reproduced the sexual division of labour. There was also Gynocriticism that was started by Elaine Showalter which emphasised how distinctive women writing was, saying that the women literary tradition differ from that of the men in the range of syntax, semantics and pragmatics since the woman is physiologically different from the man. Hence, women have their own culture of writing in such a way that behind the writing, the gender could be recognised. The French feminist scholar Alice Jardine preferred to see that distinct nature of gender in the writing of women as Gynesis, which did not emphasise the gender of the writer but the feminisation of the text. That is, the feminine effect of the text, by its syntactic, semantic and pragmatic substances, made on the writing. Another French feminist, Monique Wittig, took a more radical stand, rejecting the use of the term "woman" because, in its socially constructed form, it would not include a lesbian, who is not a "woman" in the sense of sexuality. She thus preferred the term "Lesbian" because it suggests an un-oppressed sexual identity and allows the woman to name and to redefine herself in sexuality and sexual roles. This gave rise to what is today known as lesbian feminism. Black writers and scholars living in the United States of America and Britain embarked on appropriating feminism to their own peculiar situation, resulting in what today is seen as Black/African feminism. In her book, *In Search of Our Mother's Gardens*, Alice Walker deconstructs the racial sense inherent in the word feminism, substituting it with what she calls womanism, to replace black feminism. Given the peculiar demands of the society on the African woman, Walker thought that the African woman could not totally, as feminism demanded; rejects the man in her life. Womanism, therefore, advocates a room in which the woman and the man can co-habit.

### 4.7.3 Actual Test Result versus Expected Test Result

Table 4.12 and Table 4.13 summarize actual and expected results obtained from module/unit code test and the whole system test plan carried out for this model.

**Table 4.12:** Results from Components test

Components	Expected Result	Actual Result
Global Resource Manager	It is expected that this component receive client's job request, grant JobID, retrieve input splits from HDFS and allocate job to appropriate RU_RM	The component does all functions as it is expected.
Rack Unit Resource Manager	It is expected of this component to receive job from RM and contact Node Manager for job monitoring and execution.	The component locates appropriate Node Manager with input splits for task execution.
Node Manager	This component is expected to launch Application Master for job execution and to monitor job execution process.	The component does just as it was expected.
Application Master	It is expected of this component to execute job by launching YARNChild which create mappers for each input split and subsequently reducers for each intermediate result. It is also expected that final result be stored in HDFS	Application Master does all these functions through the help of RU_RM that releases resource containers for task execution.
HDFS	It is expected of this component to receive job	Component does as expected.

	resources from client's local machine, store location of input splits and job resources to be retrieved by RM for task execution on appropriate RU_RM.	
Name Node	It is expected that this component stores list of data nodes and input splits for each job.	Component does as expected by keeping metadata of each job input split.

**Table 4.13:** Whole system test result

Test Data	Expected Result	Actual Result
Hadoopproject.txt	Expected that the system counts each word in the text file and output the number of occurrence for each. Also expecting to know the processing and response time for the new system and the old system.	System does as expected. Process and response time for new system is less compared to that of the existing system.

#### 4.7.4 Performance Evaluation

This section evaluates the new model and the existing model for scalability and efficient resource management. Though there are other works (Albert *et al.*, 2016; Konstantinos *et al.*, 2018) carried out after Hadoop YARN was developed, their major focus was to guard against RM failure and not on scalability (ability to expand the cluster with more data nodes). Albert *et al.*, 2016 and Konstantinos *et al.*, 2018 architectures still have a centralized resource manager, which is a scalability bottleneck. Also, Hadoop YARN is an open source framework, which has a standard benchmark workload called

WordCount for testing scalability hence; the need to use this framework for evaluation test. WordCount is a typical two-phased Hadoop workload which involves map task counting the frequency of individual word in a text file while the reduce task shuffles and sum up the number of times each word appears in the text. This shows a representation of a large subset of real-world MapReduce jobs that transforms data from one representation to another and further extracts a small amount of interesting data from these large datasets. To fairly capture the timestamp of each task, execution time for each block of a single task was recorded. This was done for both improved model and the existing model. Results of the total finished time are presented in a bar chart. Two different file sizes were used for this experiment, with finished time for each block of file recorded.

**Experiment 1:-** Figure 4.26 shows results of WordCount operation with text file of 30.5kB in size.

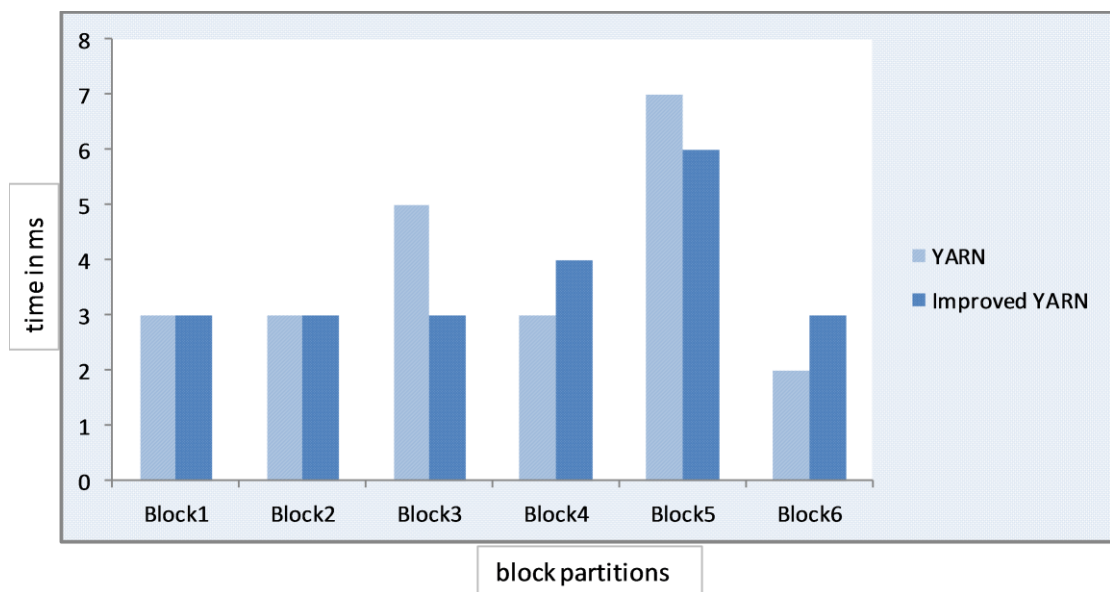


Figure 4.26: Block partitions result of wordcount operation for improved and existing model with file size = 30.5kB

**Experiment 2:-** The second experiment shows a larger file size of 92kB. Figure 4.27 shows results of WordCount operation performed on this file.

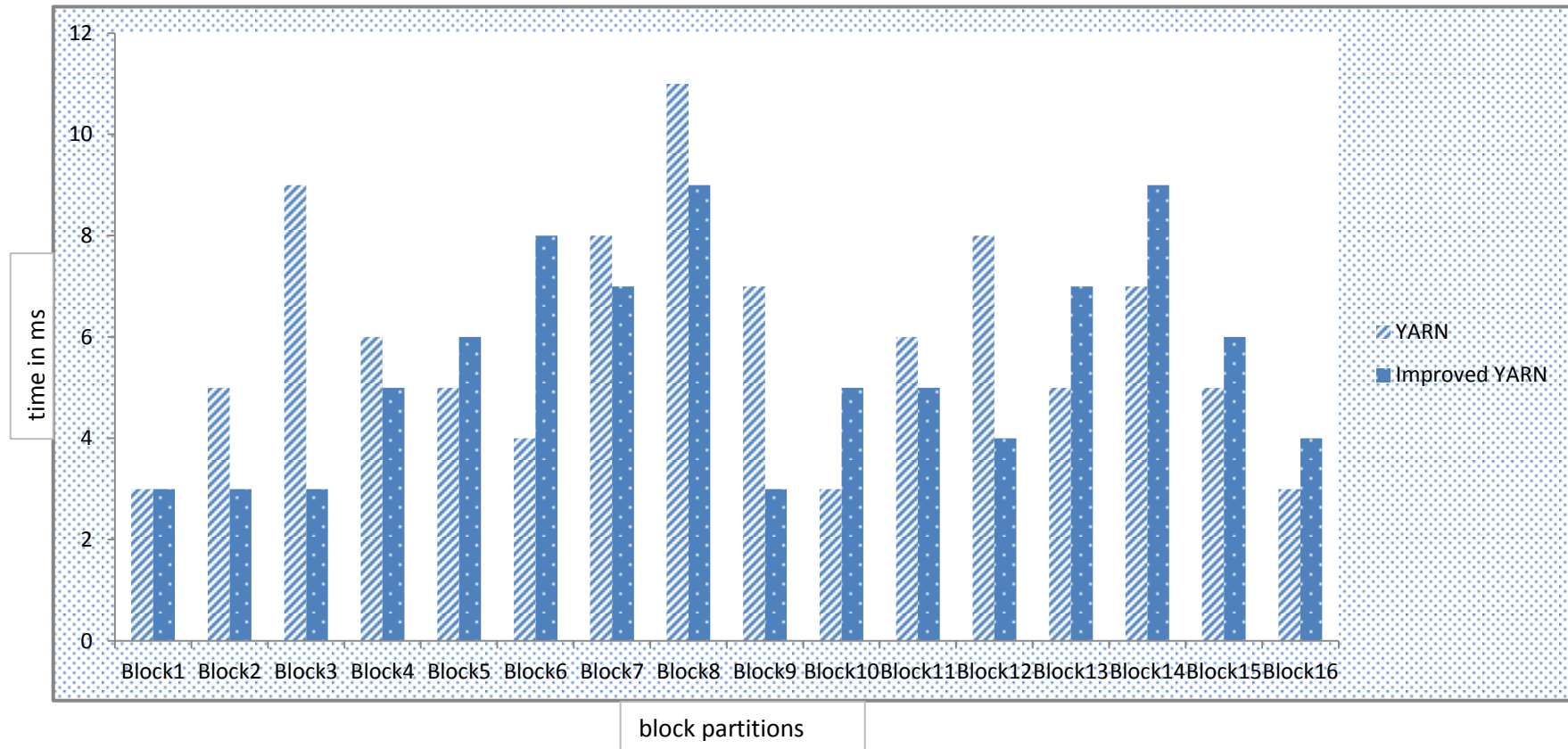


Figure 4.27: Block partitions result of wordcount operation for improved and existing model with file size = 92kB



## Discussion

Two performance metrics were defined for this work; efficiency and average task-delay ratio.

- i. Efficiency:- Efficiency in this work represents the percentage of the ideal finished time ( $T_{ideal}$ ) to the actual finished time ( $T_{actual}$ ) of a task. This metric helps quantify the average system utilization of the new and existing model.

$$\text{Efficiency} = (T_{actual} / T_{ideal}) * 100\% \quad \text{eqn(4.1)}$$

Higher efficiency therefore, indicates less scheduling overhead hence; a better turnaround time.

To obtain  $T_{actual}$  for this work, we run a task of file size = 6kB. Since this workload is contained in just 1block partition, it was assumed that no scheduling overhead is needed. Hence, finished time (which is approximately 3ms = 3000000000ns) forms  $T_{actual}$  for our work.

### Performance analysis for Experiment 1

$T_{ideal}$  for existing and new model are 23000000000ns and 22000000000ns respectively.

$$T_{actual} \text{ for Exp.1} = 3000000000\text{ns} * \text{number of blocks} \quad \text{eqn(4.2)}$$

$$T_{actual} = 3000000000\text{ns} * 6\text{blocks} = 18000000000\text{ns}$$

Efficiency of existing model from eqn(4.1)

$$\begin{aligned} &= 18000000000\text{ns}/23000000000\text{ns} * 100\% \\ &= 0.783 * 100\% \\ &= 78.3\% \end{aligned}$$

Efficiency of new model from eqn(4.1)

$$\begin{aligned} &= 18000000000\text{ns}/22000000000\text{ns} * 100\% \\ &= 0.818 * 100\% \\ &= 81.8\% \end{aligned}$$

Performance analysis for Experiment 2

$T_{ideal}$  for existing and new model are 95000000000ns and 87000000000ns respectively.

$T_{actual}$  for Exp.2 = 3000000000ns \* number of blocks

$T_{actual} = 3000000000ns * 16blocks = 48000000000ns$

Efficiency of existing model from eqn(4.1)

$$= \frac{48000000000ns}{95000000000ns} * 100\%$$

$$= 0.505 * 100\%$$

$$= 50.5\%$$

Efficiency of new model from eqn(4.1)

$$= \frac{48000000000ns}{87000000000ns} * 100\%$$

$$= 0.552 * 100\%$$

$$= 55.2\%$$

Performance evaluation for the two analyses above is represented in the bar chart in Figure 4.28.

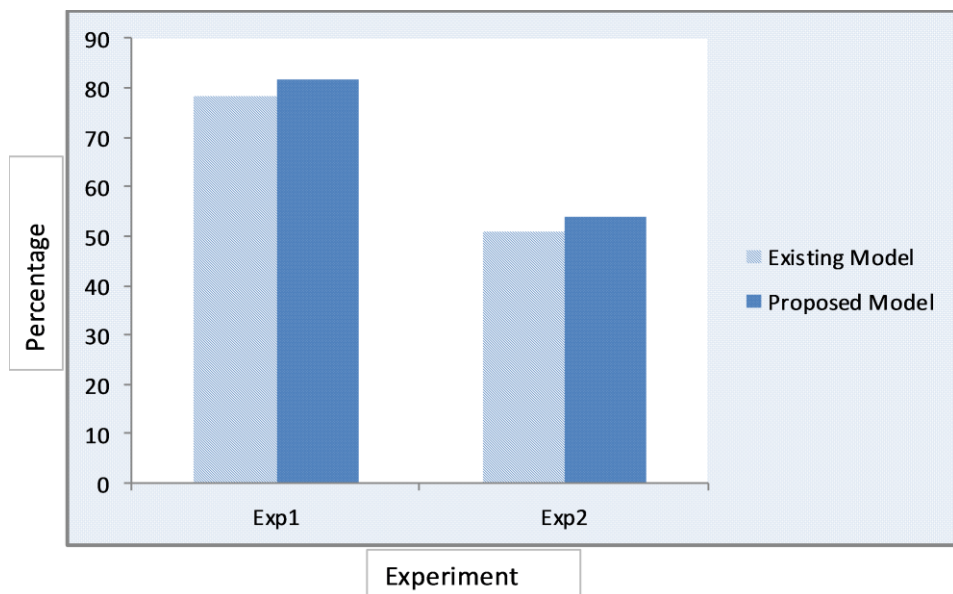


Figure 4.28: Percentage difference between the efficiency of existing and new model

- ii. Average Task-Delay Ratio:- Average Task-Delay Ratio ( $r_{td}$ ) for this work is computed as the normalized difference between average ideal task finished time ( $T_{itf}$ ) and actual task finished time ( $T_{atf}$ ). This is represented by below.

$$r_{td} = (T_{itf} - T_{atf}) / T_{atf} \quad \text{eqn(4.3)}$$

Where

$$T_{itf} = T_{ideal} / \text{number of blocks} \quad \text{eqn(4.4)}$$

$$T_{atf} = T_{actual \text{ of a single block}} \quad \text{eqn(4.5)}$$

This metric measured how fast the models can respond from a task's perspective. If rtd is small, we conclude that there is faster response time and lower scheduling overheads.

#### Average Task-Delay Ratio performance analysis for Experiment 1

From eqn(4.3), Existing Model

$$\begin{aligned} r_{td} &= [(23000000000\text{ns}/6) - 3000000000\text{ns}] / 3000000000\text{ns} \\ &= (3833333333.3\text{ns} - 3000000000\text{ns}) / 3000000000\text{ns} \\ &= 833333333.3\text{ms} / 3000000000\text{ns} \\ &= 0.278\text{ns} \end{aligned}$$

From eqn(4.3), Improved Model

$$\begin{aligned} r_{td} &= [(22000000000\text{ns}/6) - 3000000000\text{ns}] / 3000000000\text{ns} \\ &= (3666666666.7\text{ns} - 3000000000\text{ns}) / 3000000000\text{ns} \\ &= 666666666.7\text{ns} / 3000000000\text{ns} \\ &= 0.222\text{ns} \end{aligned}$$

#### Average Task-Delay Ratio performance analysis for Experiment 2

From eqn(4.3), Existing Model

$$\begin{aligned} r_{td} &= [(95000000000\text{ns}/16) - 3000000000\text{ns}] / 3000000000\text{ns} \\ &= (5937500000\text{ns} - 3000000000\text{ns}) / 3000000000\text{ns} \\ &= 2937500000\text{ns} / 3000000000\text{ns} \\ &= 0.979\text{ns} \end{aligned}$$

From eqn(4.3), Improved Model

$$\begin{aligned} r_{td} &= [(87000000000\text{ns}/16) - 3000000000\text{ns}] / 3000000000\text{ns} \\ &= (543750000\text{ns} - 300000000\text{ns}) / 300000000\text{ns} \\ &= 243750000\text{ns} / 300000000\text{ns} \\ &= 0.813\text{ns} \end{aligned}$$

Average Task-Delay Ratio performance evaluation for the two experiments is shown in Figure 4.29.

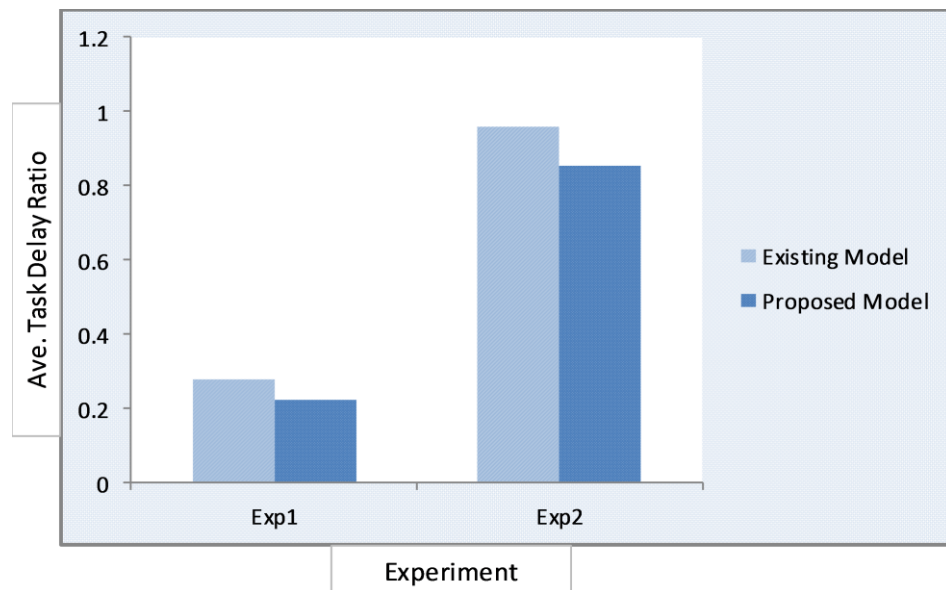


Figure 4.29: Average Task-Delay Ratio of existing and new model

#### 4.7.4. Limitation of the System

The need to add more physical machines at the rack unit layer to serve as rack unit resource managers is the limitation of this system. With results obtained from the analysis of the existing and new system however, the benefits of this new system outweigh the cost involved in obtaining these machines. Moreover, with the novel ring architecture for rack unit resource managers adopted in this system, machines to be used can be inexpensive commodity computers.

## 4.8 System Conversion

### 4.8.1 Changeover Procedures

Four approaches for system changeover are possible. These four approaches; their descriptions, advantages, disadvantages and implications of using them are as follows;

- i. **Direct Cut Over:-** Is a direct approach where existing system is cut and overwrite by a new system. This approach immediately stops the old system to allow new system becomes operational. The approach is not expensive but has high risk of data loss with no option to revert to old system as backup.
- ii. **Parallel Operation:-** This allows for both old and new systems to run simultaneously for a specified period of time. The old system can only be terminated at a point where all stakeholders are satisfied with the new system. The approach has low risk of engagement and allows for backup in situation where the new system fails but, it is the most expensive changeover procedure. There is also, increased workload and delay in processing because users need to work on both systems.
- iii. **Pilot Operation:-** This approach implements new system at a selected location of the company. This location is referred to as pilot site. The old system will be allowed to run for the entire company (including the pilot site). At any point the new system proves successful at the pilot site, it is implemented in the rest of the company (usually using direct cut over method). The approach reduces risk of failure, it is less expensive than parallel and it is a safer method.
- iv. **Phased Operation:-** This approach implements new system in modules or stages. The approach is similar to pilot procedure but here, part of the system is provided to all users instead of the entire system released to some users. Though less expensive and limited risk of failure than parallel approach, it can cost more than pilot approach if the system involves a large number of separate phases.

#### **4.8.2 Recommended Procedure**

From the four possible changeover procedures, it is clear that direct and parallel approach will not be suitable for our new model. This is because, Hadoop project is a very large project and drawback like high risk or high cost cannot be accommodated. Still, phased approach is not suitable for this work because, to achieve desired result and the purpose of comparison and analysis of both systems, implementing the new system in modules will not yield fruitful result. Pilot approach therefore, is the most suitable and recommended procedure for this system. Hadoop cluster in a particular geo-graphical location can be chosen to implement the new system and once the system is proved successful, other cluster centres can implement the system through direct cut over. This will reduce cost and risk of system failure.

#### **4.9 System Security**

Hadoop has evolved to address security concerns as it pertain authentication, authorization, accounting and data protection. It is been used securely and suffessfully today in sensitive financial service applications, private healthcare initiatives and range of other security-sensitive environments. Since this system will be part of the entire project with improvement on its resource management, key security issues as addressed in the old system will also be part of this system. The system provides two modes of authentication. First is the simple or pseudo authentication, which places trust in user's assertion about who they are. The second provides a fully secured cluster. Authorization in this system gives access privilege for user or system and accounting provides the ability to track resources used within the system. Two authorization mechanism put in place to secure this system are 'Admin Panel' login access and 'Cluster' login access. The 'Admin Panel' login is as described in Figure 4.30.

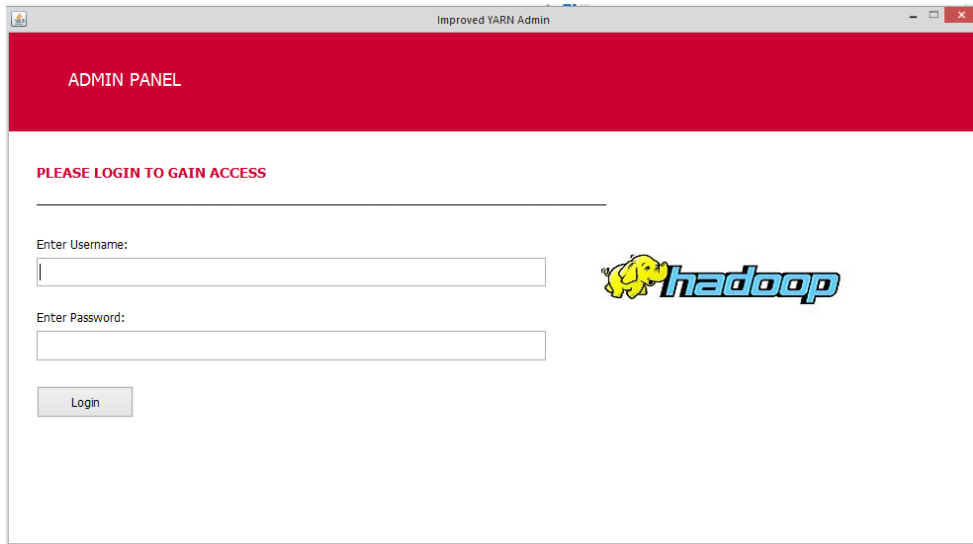


Figure 4.30: Admin Panel access

Figure 4.30 gives access to the only the Admin of this system. The admin can add users who will gain access to the system. The Admin will also, be able to know number of files and blocks that has so far been uploaded in the cluster. This is shown in Figure 4.31.

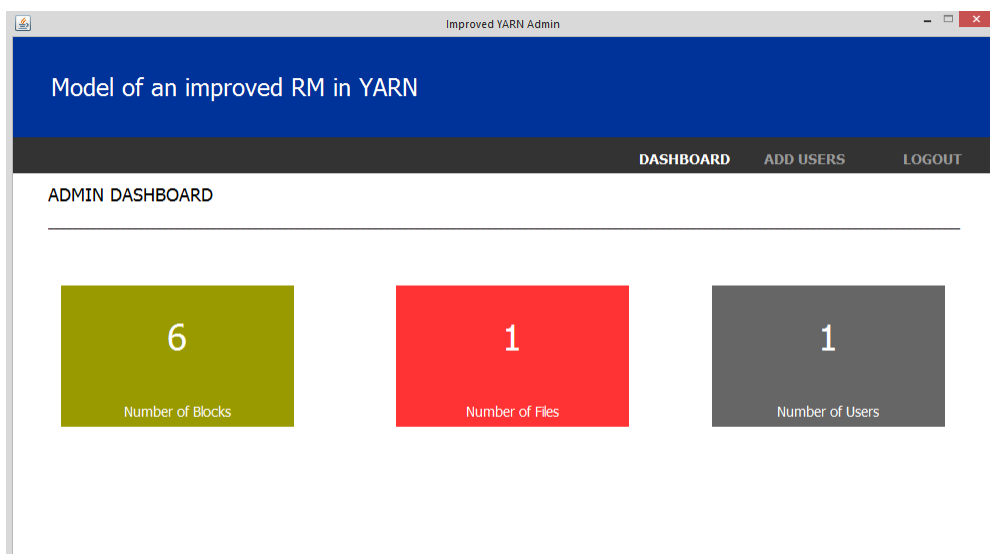


Figure 4.31: The Admin Dashboard

The second level of authorization is at the cluster as shown in Figure 4.32. This allows a user to gain access to the cluster for MapReduce task.

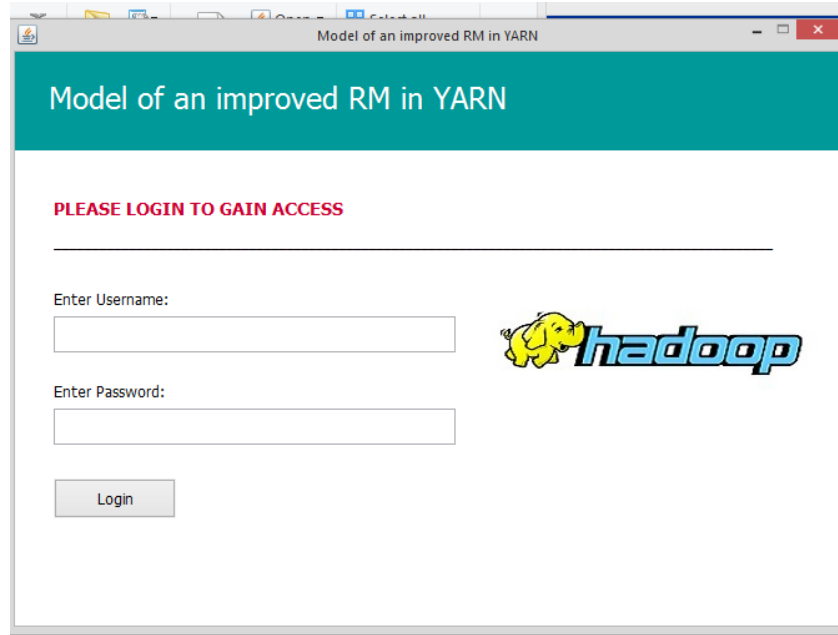


Figure 4.32: Cluster login

#### 4.10 Training

To launch this application, you will need to pass through all security checks as mentioned in the previous section. You can now click Central Machine to gain access to the cluster. The main menu shown in Figure 4.33 will be displayed.

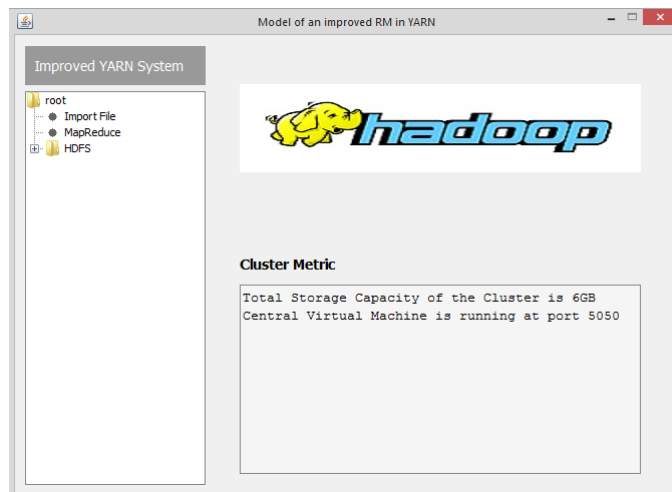


Figure 4.33: Main menu for Improved YARN

On the left hand corner are functions/submenu for this application and on the right hand is information about the cluster metric.



To import a text file to be stored in HDFS, the Import File page displayed in Figure 4.34 is used.

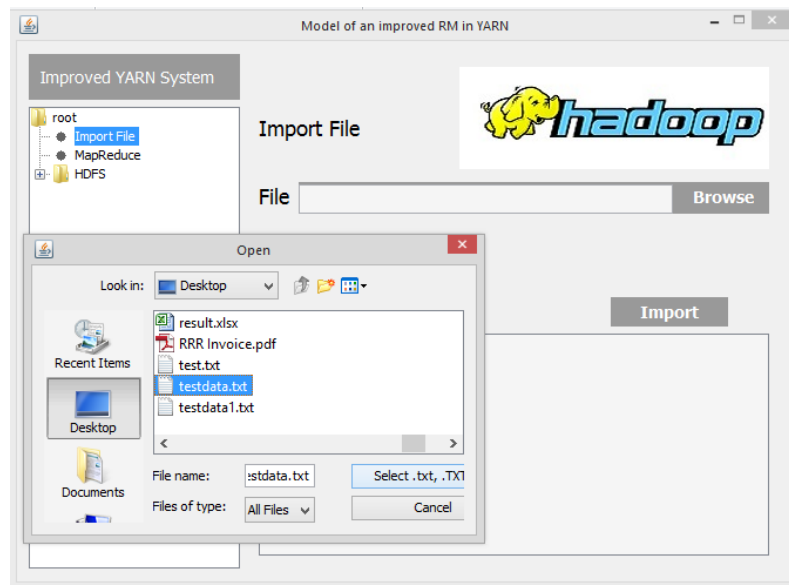


Figure 4.34: Import File page

From Figure 4.34, the user will browse to locate text file (in his local machine) to be imported into the compute nodes. The text file is broken down into blocks of 6kB each with the last block partition saved as a block even if the block size is less than 6kB.

To perform WordCount operation on any text file, the MapReduce function page as shown in Figure 4.35 is launched. This function count each word stored in the virtual nodes and gives statistics of the total of each word in a text file.

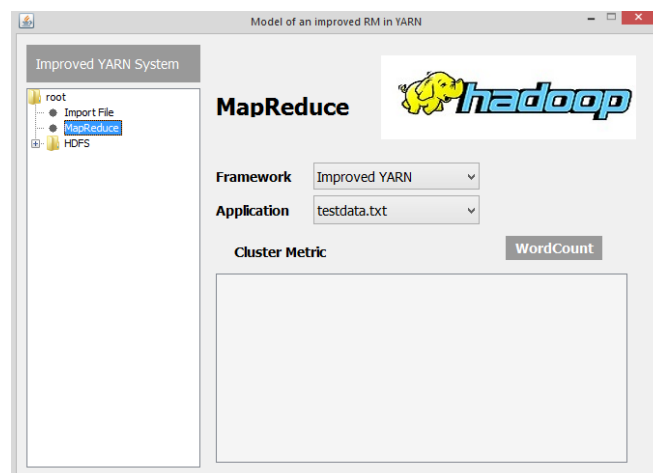


Figure 4.35: MapReduce page

Figure 4.35 has two options (Improved YARN and the existing system called YARN)

*If you select Improved YARN, the system does the following*

1. You enter the **application** (text file) you want to process e.g. abstract.docx
2. Click on WordCount

*If you select YARN, the system does the following*

1. You enter the **application** (text file) you want to process e.g. abstract.docx
2. Click on WordCount

Once WordCount operation is completed, the output screen will be displayed as shown in Figure 4.36.

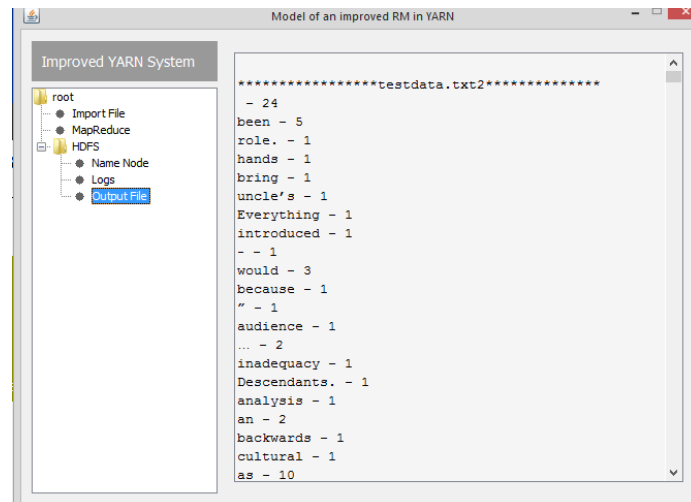


Figure 4.36: Result of WordCount operation

Figure 4.36 shows counts for each word. To see finished time for block partitions of this file will require the user to click on Logs. To also know the total number of blocks stored in cluster, user will need to click on NameNode. The output menu display result of WordCount operation in a file.

## 4.11 Documentation

The documentation for this system describes how you install and run the application. To install this application however, there are basic tools that need to be in place.

1. JDK installation – you will need to install JDK 1.6 or later version of Java. JDK can be downloaded from <http://www.oracle.com/technetwork/java/index.html>
  2. Cygwin installation – First, you will need to download Cygwin setup from <https://cygwin.com/install.html>. To install, you click on setup from the folder you downloaded.
  3. The next is to download and install Windows SDK from <https://developer.microsoft.com/en-us/windows/downloads> Windows SDK provides the tools, compilers, headers and libraries needed to run the new system.
  4. Maven and Protocol Buffer will be the next tool to install. Install Maven 3.0 or later version and protocol buffer 2.5.0 into the directory c:/maven and c:/protobuff respectively.
  5. Next step will be to setup environment path for JAVA\_HOME, M2\_HOME (for Maven) and platform (x64 or win32 depending on your system architecture). Edit the path variable under system variables to add the following: c:/cygwin64/bin; c:/cygwin64/usr/sbin; c:/maven/bin; c:/protobuff.
- To setup environment variable for JAVA\_HOME for instance, you right-click on ‘my computer’ on your desktop, click ‘properties’ and locate ‘advanced system settings’ as shown in Figure 4.36.

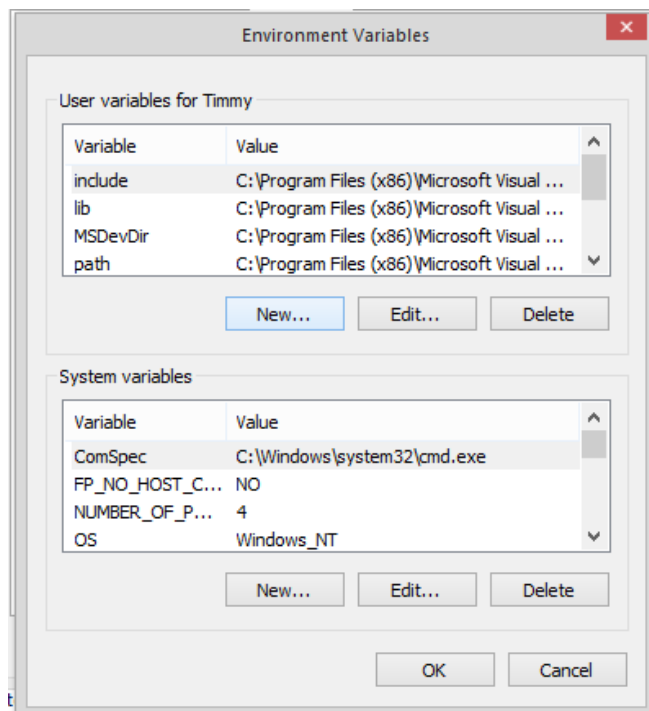


Figure 4.37: Environment Variable Setup

Click on 'New' button on 'user variables' section. On the 'variable name' space, type JAVA\_HOME. On the 'variable value' space, go to your 'program file' from your 'c – drive'. Locate Java, then jdk. Copy the values on your URL bar and paste in the 'variable value space'.

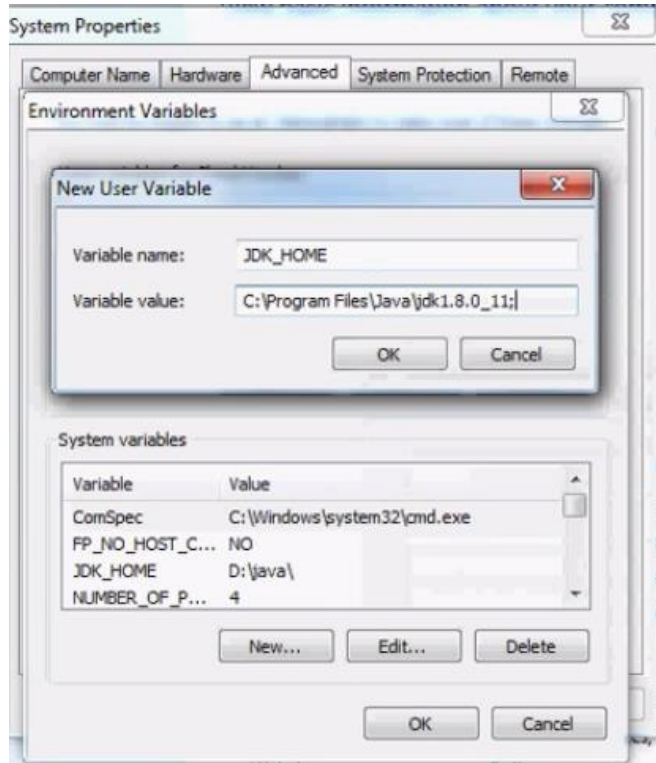


Figure 4.38: User Variable setup

6. Once the basic tools described in step 1 – 5 has been successfully installed, you can now launch this new application. The procedure for this is as contained in section 4.10 of this work.

## CHAPTER FIVE

### SUMMARY, CONCLUSION AND RECOMMENDATION

#### 5.1 Summary

Big data has brought in an era of data exploration and utilization with Hadoop MapReduce computational paradigm as its major enabler. Though great efforts through the implementation of Hadoop has made computation scale to tens of thousands commodity cluster processors, the centralized architecture of resource manager has adversely affected response time in large datacenters. Decentralizing the responsibilities of resource manager to address scalability issues of Hadoop for better response, processing, turnaround time and to eliminate single point of failure is therefore necessary; hence, this study. The aim of this research was to develop a model of an improved scalable resource management system for Hadoop YARN. The objectives were to; decentralize the responsibilities of Resource Manager (RM) by creating Rack\_Unit Resource Manager (RU\_RM) layer; configure RU\_RM layer to ensure that each RU\_RM controls resource requests for compute nodes within its rack; develop a ring architecture in RU\_RM layer to guard against failure; and to carry out a performance evaluation test between the developed model and existing model.

Object Modeling Technique (OMT) methodology was adopted with parallelization and push-based techniques used to decentralize the responsibilities of RM. Java Remote Method Invocation was implemented to maintain resource requests control within racks. A self-stabilizing Peer-to-Peer (P2P) topology was used in the RU\_RM layer so that, if one RU\_RM fails, the unit preceding/succeeding it takes over the responsibilities of compute nodes in that rack. Hadoop benchmark workload called WordCount was used to compare the efficiency and average task-delay ratio of the developed and existing models.

Decentralized RM showed that cluster average execution times of the developed model for file sizes 30.5kB and 92kB were less compared to execution times of the existing model. 12kB containing 2 blocks from file size 30.5kB and 30kB containing 5 blocks from file size 92kB were independently computed on two RU\_RMs for a cluster configuration of three RU\_RMs. The remaining 6.5kB containing 2 blocks and 32kB containing 6 blocks were computed on the third RU\_RM. Ring architecture deployed

showed that at-least one ( $RU\_RM \geq 1$ )  $RU\_RM$  was available during cluster execution. Efficiency of new model for file sizes 30.5kB and 92kB showed a difference of 3.5% and 4.7%, respectively better than the existing model. The new model had lower average task-delay ratio of 0.056ns and 0.166ns for file sizes 30.5kB and 92kB, respectively compared to the existing model.

## **5.2 Conclusion**

A model of improved scalable resource management system for Hadoop Yet Another Resource Negotiator (YARN) is an improvement over the existing Hadoop YARN model. The new model decouples the responsibilities of resource manager in YARN by providing another layer called  $RU\_RM$  layer. The layer forms a peer-to-peer architecture to guard against failure. This new model was developed and tested in Java programming language. Hadoop benchmark workload called WordCount was used for comparing existing and new model. Finished time of block partitions recorded in log file for a single text file was used as basis for comparing the models. Results obtained from computation of efficiency and average task-delay ratio showed that as file size increases, the developed model performed better than existing model. Since Hadoop was developed for big data analytics, this work is recommended as a better solution for a scalable and efficient resource management framework.

## **5.3 Recommendation**

### **5.3.1 Application Areas**

This new model can be used in various big data activities. It can be used to configure, manage and orchestrate data motion, pipeline processing, disaster recovery and data retention workflows. The system can be used in applications like yahoo weather, facebook photo gallery and google search index. The system can also help analyse life-threatening risks. This is possible where patient's medical history together with series of test and results are analysed with the help of big data tools. Identifying warning signs of data security breaches is another area of application. Before data breaches occur, there are typical early warning signs such as unusual server pings, suspicious emails or other forms of communication that could suggest internal collusion. With ability to

mine and correlate people, business and machine-generated data all in one seamless analytic environment, the new system can help provide complete picture of who is doing what and when.

### **5.3.2 Suggestions for Further Research**

Further design issue that can be looked at in YARN framework is the centralized metadata management of the framework. HDFS is the default distributed file system responsible for keeping all files/blocks metadata in a centralized daemon called Name Node. This daemon monitors all compute nodes through its metadata information. With the rate at which this framework is growing, the number of data files will increase significantly that will lead to a much higher demands of memory footprints and metadata access ratio. This will obviously overwhelm the centralized metadata management. Things may become worst for abundant small data files. To avoid this situation therefore, further research should be carried out to have efficient metadata management in this framework.

Another area of interest is data replica system for rack failure in Hadoop framework. Anytime heartbeat communication stops between NameNode and Data Node, it is presumed that such Data Node is dead and any data its holding gone as well. Previous block reports received from the said Data Node will help the NameNode to know which copies of blocks died along the with the node. Using rack-aware policy, the NameNode will re-replicate those blocks on other data nodes. The limitation with this however is, when an entire rack of servers falls off the network due to rack switch failure or power failure, it then means that the NameNode will instruct the remaining nodes in the cluster to re-replicate all the data blocks lost in that rack. This process may mean that hundreds of terabytes of data will need to begin traversing the network.

## **5.4 Contribution to Knowledge**

Decentralizing the global control of resource manager in YARN framework by providing Rack\_Unit Resource Manager (RU\_RM) layer and the peer-to-peer ring

architecture designed to guide against RU\_RM failure has added positively to the existing knowledge.

- i. There is improved scalability:- With the breaking down of Resource Manager's responsibilities into Rack Unit Resource Manager (RU\_RM), more nodes can easily be added to each rack without predicting future scalability bottleneck as observed with the existing framework.
- ii. Easy fault/failure detection:- Isolation of slave nodes using rack aware technique will help administrators locate faulty nodes easily.
- iii. Fast execution/response to client job:- Since each RU\_RM now handles the responsibility of allocating and monitoring resources in a rack, response/execution time for each job becomes faster compared to when a central Resource Manager has to respond to each job needs in the cluster.
- iv. Easy Recovery:- With relaxed-ring topology built with this framework, if any Rack Unit Resource Manager fails, the predecessor or successor can continue with the management of compute nodes in that rack until such RU\_RM recovers from failure. This process will ensure that jobs do not halt at a point where any of the daemons (central resource manager or rack unit resource manager) fails.



## REFERENCES

- Albert, J., Abhishek, C. and Jon, W. (2016). Awan: Locality-aware resource manager for geo-distributed data-intensive applications. *IEEE International Conference on Cloud Engineering (IC2E)*, Berlin, 2016, Germany: IEEE Computer Society.
- Anthony, M. (2015). Introduction to High-Performance Computing cluster. White paper from Lexis Nexis Risk Solutions, USA. Retrieved from <https://docs.huihoo.com/hpcc/Introduction-to-HPCC.pdf> on 25th September, 2016.
- Apache (2016). Apache Hadoop. Retrieved from <https://hadoop.apache.org/>. on 3rd March, 2017.
- Avita, K., Mohammad, W. and Goudar, R. H. (2013). Big Data: Issues, challenges, tools and good practices”. *IEEE International Conference on Robotics and Automation*, Karlsruhe, 2013, Germany: IEEE Computer Society.
- Bakshi, R. P. and Sonali, A. (2016). Comparative study of big data computing and storage tools: A review. *International Journal of Database Theory and Application*, 9(1), 45-66.
- Barroso, L. A., Dean, J. and Holzle, U. (2003). Web search for a planet: The Google cluster architecture. *IEEE Computer Society*, 23(2) 22-28.
- Bekkerman R., Bilenko M., and Langford J. (2012). *Scaling up Machine Learning: Parallel and distributed approaches*. Berkeley, California: University of California.
- Benjamin, H., Andy, K., Matei, Z., Ali, G., Anthony, D. J., Randy, K., Scott, S. and Ion, S. (2012). *Mesos: A platform for fine-grained resource sharing in the data centre*. Berkeley, California: University of California.
- Beyer, M. A., and Laney, D. (2012). The Importance of 'Big Data: A Definition. Retrieved from <https://www.gartner.com/doc/2057415/importance-big-data-definition> on January 18<sup>th</sup>, 2017.
- Bialecki, A., Cafarella, M., Cutting, D. and O'Malley, O. (2005). Hadoop: A framework for running applications on large clusters built of commodity hardware. Retrieved from <http://lucene.apache.org/hadoop/> on 6<sup>th</sup> June, 2015.
- Biehn, N. (2013). The missing V's in big data: Viability and value. Retrieved from <http://www.wired.com/insights/2013/05/the-missing-vs-in-big-data-viability-and-value/>. on June 20<sup>th</sup>, 2017.

- Brad, H. (2011). Understanding Hadoop clusters and the network. Retrieved from <http://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/> on 23<sup>rd</sup> May, 2017.
- Bradly, D., Clair, S. T., Farrellee, M., Guo, Z., Livny, M., Sfiligoi, I. and Tannenbaum, T. (2011). An update on the scalability limits of Condor batch system. *Journal of Physics*, 331(6), 1-6.
- Camille, R. (2015). "Big Data open platforms". Project final report for Department of Information and Systems Engineering, Polytechnic Institute of Coimbra. Retrieved from <https://slidex.tips/download/big-data-open-platforms> on 24th June, 2017.
- Chaiké, R., Jenkins, B., Larson, P. A., Ramsey, B., Shakib, D., Weaver, S. and Zhou, J. (2008) Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endowment*, Auckland, 2008, New Zealand: ACM.
- Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *OSDI '04: 6th Symposium on Operating Systems Design and Implementation*, Berkeley, 2008, CA: USENIX Association.
- Dilpreet, S. and Chandan, K. R. (2014). A survey on platforms for big data analytics. *Journal of Big Data: A Springer Open Journal*, 1(1), 8-27. Retrieved from <https://www.journalofbigdata.com/content/1/1/8>
- Dominique, A. H. (2015). Hadoop design, architecture and MapReduce performance. *DH Technologies*. Retrieved from [www.dhtusa.com](http://www.dhtusa.com) on 10<sup>th</sup> March, 2015.
- Dongyao, W., Sherif, S. and Liming, Z. (2017). *Big data programming models*. Australia, Sydney: Springer International Publishing.
- Douglas, K. (2012). Big Data Infographic: Solve your big data problems. Retrieved from <http://www.intel.in/content/www/in/en/big-data/solving-big-dataproblems-infographic.html>. On June 20<sup>th</sup>, 2017.
- Facebook (2012). Under the hood: Scheduling MapReduce jobs more efficiently with Corona. Retrieved from <http://on.fb.me/TxUsYN> on 21st October, 2015.
- Francisco, P. (2011). *The Netezza data appliance architecture: A platform for high performance data warehousing and analytics*. USA, IBM Redbooks.
- Garcia, J. (2014). The BBT Sessions: Hortonworks, big data and the data lake. Retrieved from <http://www.dofthings.com/2014/04/the-bbbt-sessions-hortonworks-big-data.html> on 18th January, 2017.

- Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The Google file system. *Paper presented at the ACM SIGOPS operating systems review*, New York City, 2003, NY: ACM.
- Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S. and Stoica, I. (2011). Mesos: a platform for fine-grained resource sharing in the data center. *In Proceedings of the 8th USENIX conference on Networked systems design and implementation*, Berkeley, 2011, CA, USA: USENIX Association.
- Holzinger, A., Stocker, C., Ofner, B., Prohaska, G., Brabenetz, A. and Hofmann-Wellenhof, R. (2013). *Combining HCI, natural language processing, and knowledge discovery—potential of IBM content analytics as an assistive technology in the biomedical field*. Berlin, Germany: Springer.
- Hong, S. and Kim, H. (2009). An analytical Model for a GPU Architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, New York, 2009, NY: ACM.
- Hortonworks (2016). Apache Hadoop YARN. Retrieved from <https://hortonworks.com/apache/yarn> on June 18th, 2017.
- IBM (2012). IBM Analytics: The real-world use of big data. Retrieved from [www.m.ibm.com](http://www.m.ibm.com) on 12<sup>th</sup> April, 2015.
- Ibrahim, A. T. H., Nor, B. A., Abdullah, G., Ibrar, Y., Feng, X., and Samee, U. K. (2016). MapReduce: Review and Challenges. *Springer Journal*, 109(1), 389-421. <http://www.doi.org/10.1145/1327452.1327492>
- Jonathan, A., Ryden, M., Oh, K., Chandra, A., and Weissman, J. B. (2017). Nebula: Distributed edge cloud for data intensive computing. *IEEE Transaction on Parallel and Distributed Systems*, 28(11), 3229-3242.
- Konstantinos, K., Suresh, A., and Douglas, C. (2018). Advancements in YARN Resource Manager. *Encyclopedia of Big Data Technologies: Springer International Publishing*, DOI: [https://doi.org/10.1007/978-3-319-63962-8\\_207-1](https://doi.org/10.1007/978-3-319-63962-8_207-1).
- Kornacker, M., Behm, A., Bittorf, V., Bobrovitsky, T., Ching, C., Choi, A. and Erickson, J. (2015). Impala: A modern, open-source SQL engine for Hadoop. *Proceedings of the Conference on Innovative Data Systems Research*, California, 2015, USA: Innovative Data Systems Research.

- Kraska, T., Talwalkar, A., Duchi, J., Griffith, R., Franklin, M. J., and Jordan, M. (2013). MLbase: A Distributed Machine-learning System. *In: Proceedings of Sixth Biennial Conference on Innovative Data Systems Research*, California, 2013, USA: Innovative Data Systems Research.
- Madden, S. (2012). From Databases to Big Data. *IEEE Internet Computing*, 16(3), 4-6. <https://doi.org/10.1109/MIC.2012.50>
- Malte, S., Andy, K., Michael, A. and John, W. (2013). Omega: Flexible, scalable schedulers for large compute clusters. *Google Inc*, Prague, 2013, Czech Republic: ACM.
- Michael, E., Payne, L. B., Ngo, F. V. and Amy, W. A. (2014). Managing the academic data lifecycle: A case study of HPC. *IEEE International Conference on Big Data*, Washington, 2014, USA: IEEE Computer Society.
- Milojicic, D. S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., and Xu, Z. (2003). Peer-to-peer computing. *Technical Report HPL-2002-57, HP Labs*, California, CA: Hewlett-Packard Company.
- Nagina, D. and Sunita, D. (2016). Scheduling algorithm in big data: A Survey. *International Journal of Engineering and Computer Science*, 5(8), 17737-17743.
- Nawsher, K., Ibrar, Y., Ibrahim, A. T. M., Zakira, I., Waleed, K. M. A., Muhammad, A., Muhammad, S. and Abdullahi, G. (2014). Big data: Survey, technologies, opportunities and challenges. *The Scientific World Journal*, 14(7) 1-18. <http://dx.doi.org/10.1155/2014/712826>
- Ning, L., Xi, Y., Xian-He, S., Jonathan, J., and Robert, R. (2015). YARNsim: Simulating Hadoop YARN. *15<sup>th</sup> IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Shenzhen, 2015, China: IEEE.
- Nyland, L. S., Prins, J. F., Goldberg, A. and Mills, P. H. (2000). A design methodology for data-parallel applications. *IEEE Transactions on Software Engineering*, 26(4), 293-314.
- Oracle (2012). From overload to impact: An industry scorecard on big data business challenges. Retrieved from [www.oracle.com](http://www.oracle.com) on 16<sup>th</sup> April, 2015.
- Outerhout, K., Patrick, W., Matei, Z. and Ion, S. (2013). Sparrow: Distributed, low latency scheduling. *Hertz Foundation Fellowship*, Pennsylvania, 2013, USA: ACM. [dx.doi.org/10.1145/2517349.2522716](https://doi.org/10.1145/2517349.2522716)

- Pasula, S. (2016). What is the difference between Hadoop and Pentaho? *Education Management & Administration*, Retrieved from <https://www.quora.com/What-is-the-difference-between-Hadoop-and-Pentaho> on 28th June, 2018.
- Patrick, T. (2011). Hadoop challenger works to add developers. Retrieved from <http://www.computerworld.com/article/2500651/business-intelligence/hadoop-challenger-works-to-add-developers.html> on June 20th, 2017.
- Prashanth, M., Ville, T., and Jared, F. (2011). Disco: A computing platform for large-scale data analytic. *Proceeding of the 10<sup>th</sup> ACM SIGPLAN workshop on Erland*, New York, 2011, NY, USA: ACM.
- Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M. and Wilkes, J. (2013). Omega: flexible, scalable schedulers for large compute clusters. *In Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys '13*, New York, 2013, NY, USA: ACM.
- Sergio, C. G. (2015). “What about big data?” A project carried out at Computer Science and Engineering Department of the Open University of Catalonia. Retrieved from [www.openaccess.uoc.edu/webapps/o2/bitstream/.../scruzguocTFG1215memoria.pdf](http://www.openaccess.uoc.edu/webapps/o2/bitstream/.../scruzguocTFG1215memoria.pdf) ... on 25<sup>th</sup> July, 2016.
- Sharanjit, K., Rakhi, S., Dhriti, K. and Vasudha, B. (2014). Comparing data processing frameworks for scalable clustering. *Proceedings of the Twenty-Seventh International Florida Artificial Intelligence Research Society Conference, Association for the Advancement of Artificial Intelligence*, Pensacola Beach, 2014, Florida: The AAAI Press.
- Shouvik, B., and Daniel, A. M. (2013). The anatomy of MapReduce jobs, scheduling and performance challenges. *Proceedings of the 2013 conference of the Computer Measurement Group*, San Diego, 2013, CA: Semantic Scholar.
- Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. *2010 IEEE 26th symposium on Mass Storage Systems and Technologies (MSST)*, Washington D. C., 2010, USA: IEEE Computer Society.
- Sievert, O., and Casanova, H. (2004). A simple MPI process swapping architecture for iterative applications. *International Journal of High Performance Computing Application*, 18(3), 341–352.
- Sircular, S. (2013). “Gartner's big data definition consists of three parts” *Forbes*, 27 March, 2013. Retrieved from <http://bit.ly/2sIuBrA> on 28<sup>th</sup> April, 2016.

- Siya, K. (2013). What is the difference between Hadoop and Splunk? Retrieved from <https://www.quora.com/What-is-the-difference-between-Hadoop-and-Splunk> on 20<sup>th</sup> June, 2018.
- Steinmetz, R., and Wehrles, K. (2005). *Peer-to-Peer Systems and Applications*. Berlin, Heidelberg: Springer.
- Stone, Z., Zickler, T., and Darrell, T. (2008). Auto-tagging facebook: Social network context improves photo annotation. *IEEE Conference on Computer Vision and Pattern Recognition Workshop*, Anchorage, 2008, Alaska: IEEE Computer Society.
- Tannenbaum, T. (2010). What's new in Condor: What's coming up? *Condor Week*, Madison, 2010, Wisconsin: Center for High Throughput Computing. Retrieved from [https://research.cs.wisc.edu/htcondor/CondorWeek2010/condor-presentations/tannenba\\_roadmap\\_2010.pdf](https://research.cs.wisc.edu/htcondor/CondorWeek2010/condor-presentations/tannenba_roadmap_2010.pdf) on 30<sup>th</sup> March, 2016.
- Telmoda, S. M. (2015). Survey of frameworks for distributed computing: Hadoop, Spark and Storm. *Proceedings of the 10<sup>th</sup> Doctorial Symposium in Informatics Engineering DSIE '15*, Porto, 2015, Portugal: DSIE.
- TIBC (2011). Jaspersoft announces new Hadoop-based big data analytics solution. Retrieved from <https://www.jaspersoft.com/press/jaspersoft-announces-new-hadoop-based-big-data-analytics-solution> on 26th June, 2018.
- Vidhya, S., Sarumathi, S. and Shanthi, N. (2014). Comparative analysis of diverse collection of big data analytics tools. *International Journal of Computer and Information Engineering*, 8(9), 21-40.
- Vinayak, R. B., Michael, J. C. and Chan, L. (2012). Big data platforms: What's next?. *ACM Transactions on Accessible Computing*, 9(1), 44-49.
- Vinod, K. V., Arun, C. M., Chris, D., Sharad, A., Mahadev, K., Robert, E., Thomas, G., Jason, L., Hitesh, S., Siddharth, S., Bikas, S., Carlo, C., Owen, O. M., Sanjay, R., Benjamin, R., and Eric, B. (2013). Apache Hadoop YARN: Yet Another Resource Negotiator. *SOCC '13 Proceedings of the 4<sup>th</sup> annual symposium on Cloud Computing*, New York, 2013, NY: ACM. <http://dx.doi.org/10.1145/2523616.2523633>
- Wang, K. (2015). *Scalable Resource Management System Software for Extreme-Scale Distributed Systems*. (PhD Dissertation). Retrieved from [http://datasys.cs.iit.edu/publications/2015\\_IIT\\_PhD-thesis\\_Ke-Wang.pdf](http://datasys.cs.iit.edu/publications/2015_IIT_PhD-thesis_Ke-Wang.pdf)

- Wang, K., Ma, Z., and Raicu, I. (2013). Modelling many-task computing workloads on a Petaflop IBM BlueGene/P Supercomputer. *IEEE International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum*, Massachusetts, 2013, USA: IEEE Computer Society.
- Wang, K., Ning, L., Imam, S., Xi, Y., Xiabing, Z., Tonglin, L., Michael, L., Xiantte, S. and Ioan, R. (2015). Overcoming Hadoop scaling limitations through distributed task execution. *Computer Science Department, Illinois Institute of Technology*, Illinois, 2015, USA: Amazon AWS Research Grant.
- White, T. (2009). *Hadoop: The definitive guide*. Sebastopol, CA: O'Reilly Media.
- Wissem, I., Sabeur, A., Haithem, M., Mondher, M. and Engelbert, M. N. (2018). An experimental survey on big data frameworks. *34<sup>th</sup> Conference on Principles, Technologies and Application*, Bucarest, 2018, Romania: Conference Proceedings of BDA.
- Yuan, Y., Michael, I., Dennis, F. and Mihai, B. (2016). DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *8<sup>th</sup> USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, 2016, CA: USENIX Association.
- Zaharia, M., Das, T., and Li, H. (2012). Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. *4<sup>th</sup> USENIX Workshop on hot topics in cloud computing*, Boston, 2012, MA: USENIX Association.
- Zhao, D., Zhang, Z., Zhou, X., Li, T., Wang, K., Kimpe, D., Carns, P., Ross, R. and Raicu, I. (2014). FusionFS: Towards supporting data-intensive scientific applications on extreme-scale High-Performance Computing Systems. *IEEE International Conf. on Big Data*, Washington DC, 2014, USA: IEEE Computer Society.

## APPENDIX A

### PROGRAM LISTING

```
package org.apache.hadoop.yarn.server.resourcemanager;

import java.io.IOException;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.classification.InterfaceAudience.Private;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.metrics2.lib.DefaultMetricsSystem;
import org.apache.hadoop.security.SecurityUtil;
import org.apache.hadoop.util.ReflectionUtils;
import org.apache.hadoop.util.StringUtils;
import org.apache.hadoop.yarn.YarnException;
import org.apache.hadoop.yarn.api.records.ApplicationAttemptId;
import org.apache.hadoop.yarn.api.records.ApplicationId;
import org.apache.hadoop.yarn.api.records.NodeId;
import org.apache.hadoop.yarn.conf.YarnConfiguration;
import org.apache.hadoop.yarn.event.AsyncDispatcher;
import org.apache.hadoop.yarn.event.Dispatcher;
import org.apache.hadoop.yarn.event.EventHandler;
import org.apache.hadoop.yarn.security.ApplicationTokenSecretManager;
import org.apache.hadoop.yarn.security.client.ClientToAMSecretManager;
import org.apache.hadoop.yarn.server.RMDelegationTokenSecretManager;
import org.apache.hadoop.yarn.server.resourcemanager.amlauncher.AMLauncherEventType;
import org.apache.hadoop.yarn.server.resourcemanager.amlauncher.ApplicationMasterLauncher;
import org.apache.hadoop.yarn.server.resourcemanager.recovery.Recoverable;
import org.apache.hadoop.yarn.server.resourcemanager.recovery.Store;
import org.apache.hadoop.yarn.server.resourcemanager.recovery.Store.RMState;
import org.apache.hadoop.yarn.server.resourcemanager.recovery.StoreFactory;
import org.apache.hadoop.yarn.server.resourcemanager.rmapp.RMApp;
import org.apache.hadoop.yarn.server.resourcemanager.rmapp.RMAppEvent;
import org.apache.hadoop.yarn.server.resourcemanager.rmapp.RMAppEventType;
import org.apache.hadoop.yarn.server.resourcemanager.rmapp.attempt.AMLivelinessMonitor;
import org.apache.hadoop.yarn.server.resourcemanager.rmapp.attempt.RMAppAttempt;
import org.apache.hadoop.yarn.server.resourcemanager.rmapp.attempt.RMAppAttemptEvent;
import
org.apache.hadoop.yarn.server.resourcemanager.rmapp.attempt.RMAppAttemptEventType;
import org.apache.hadoop.yarn.server.resourcemanager.rmcontainer.ContainerAllocationExpirer;
import org.apache.hadoop.yarn.server.resourcemanager.rmnode.RMNode;
import org.apache.hadoop.yarn.server.resourcemanager.rmnode.RMNodeEvent;
import org.apache.hadoop.yarn.server.resourcemanager.rmnode.RMNodeEventType;
import org.apache.hadoop.yarn.server.resourcemanager.scheduler.ResourceScheduler;
import org.apache.hadoop.yarn.server.resourcemanager.scheduler.event.SchedulerEvent;
import org.apache.hadoop.yarn.server.resourcemanager.scheduler.event.SchedulerEventType;
import org.apache.hadoop.yarn.server.resourcemanager.scheduler.fifo.FifoScheduler;
import org.apache.hadoop.yarn.server.resourcemanager.security.DelegationTokenRenewer;
import org.apache.hadoop.yarn.server.resourcemanager.webapp.RMWebApp;
import org.apache.hadoop.yarn.server.security.ApplicationACLsManager;
import org.apache.hadoop.yarn.server.security.ContainerTokenSecretManager;
import org.apache.hadoop.yarn.server.webproxy.AppReportFetcher;
```



```

import org.apache.hadoop.yarn.server.webproxy.ProxyUriUtils;
import org.apache.hadoop.yarn.server.webproxy.WebAppProxy;
import org.apache.hadoop.yarn.server.webproxy.WebAppProxyServlet;
import org.apache.hadoop.yarn.service.AbstractService;
import org.apache.hadoop.yarn.service.CompositeService;
import org.apache.hadoop.yarn.service.Service;
import org.apache.hadoop.yarn.webapp.WebApp;
import org.apache.hadoop.yarn.webapp.WebApps;
import org.apache.hadoop.yarn.webapp.WebApps.Builder;

/**
 * The ResourceManager is the main class that is a set of components.
 *
 */
public class ResourceManager extends CompositeService implements Recoverable {
    private static final Log LOG = LogFactory.getLog(ResourceManager.class);
    public static final long clusterTimeStamp = System.currentTimeMillis();

    protected ClientToAMSecretManager clientToAMSecretManager =
        new ClientToAMSecretManager();

    protected ContainerTokenSecretManager containerTokenSecretManager =
        new ContainerTokenSecretManager();

    protected ApplicationTokenSecretManager appTokenSecretManager =
        new ApplicationTokenSecretManager();

    private Dispatcher rmDispatcher;

    protected ResourceScheduler scheduler;
    private ClientRMService clientRM;
    protected ApplicationMasterService masterService;
    private ApplicationMasterLauncher applicationMasterLauncher;
    private AdminService adminService;
    private ContainerAllocationExpirer containerAllocationExpirer;
    protected NMLivelinessMonitor nmLivelinessMonitor;
    protected NodesListManager nodesListManager;
    private EventHandler<SchedulerEvent> schedulerDispatcher;
    protected RMAppManager rmAppManager;
    protected ApplicationACLsManager applicationACLsManager;
    protected RMDelegationTokenSecretManager rmDTSecretManager;
    private WebApp webApp;
    protected RMContext rmContext;
    private final Store store;
    protected ResourceTrackerService resourceTracker;

    private Configuration conf;

    public ResourceManager(Store store) {
        super("ResourceManager");
        this.store = store;
        this.nodesListManager = new NodesListManager();
    }

    public RMContext getRMContext() {
        return this.rmContext;
    }

    @Override

```

```

public synchronized void init(Configuration conf) {

    this.conf = conf;

    this.rmDispatcher = createDispatcher();
    addIfService(this.rmDispatcher);

    this.containerAllocationExpirer = new ContainerAllocationExpirer(
        this.rmDispatcher);
    addService(this.containerAllocationExpirer);

    AMLivelinessMonitor amLivelinessMonitor = createAMLivelinessMonitor();
    addService(amLivelinessMonitor);

    DelegationTokenRenewer tokenRenewer = createDelegationTokenRenewer();
    addService(tokenRenewer);

    this.rmContext = new RMContextImpl(this.store, this.rmDispatcher,
        this.containerAllocationExpirer, amLivelinessMonitor, tokenRenewer);

    addService(nodesListManager);

    // Initialize the scheduler
    this.scheduler = createScheduler();
    this.schedulerDispatcher = createSchedulerEventDispatcher();
    addIfService(this.schedulerDispatcher);
    this.rmDispatcher.register(SchedulerEventType.class,
        this.schedulerDispatcher);

    // Register event handler for RmAppEvents
    this.rmDispatcher.register(RMAppEventType.class,
        new ApplicationEventDispatcher(this.rmContext));

    // Register event handler for RmAppAttemptEvents
    this.rmDispatcher.register(RMAppAttemptEventType.class,
        new ApplicationAttemptEventDispatcher(this.rmContext));

    // Register event handler for RmNodes
    this.rmDispatcher.register(RMNodeEventType.class,
        new NodeEventDispatcher(this.rmContext));

    //TODO change this to be random
    this.appTokenSecretManager.setMasterKey(ApplicationTokenSecretManager
        .createSecretKey("Dummy".getBytes()));

    this.nmLivelinessMonitor = createNMLivelinessMonitor();
    addService(this.nmLivelinessMonitor);

    this.resourceTracker = createResourceTrackerService();
    addService(resourceTracker);

    try {
        this.scheduler.reinitialize(conf,
            this.containerTokenSecretManager, this.rmContext);
    } catch (IOException ioe) {
        throw new RuntimeException("Failed to initialize scheduler", ioe);
    }

    masterService = createApplicationMasterService();
}

```

```

addService(masterService) ;

this.applicationACLsManager = new ApplicationACLsManager(conf);

this.rmAppManager = createRMAppManager();
// Register event handler for RMAppManagerEvents
this.rmDispatcher.register(RMAppManagerEventType.class,
    this.rmAppManager);
this.rmDTSecretManager = createRMDelegationTokenSecretManager();
clientRM = createClientRMService();
addService(clientRM);

adminService = createAdminService(clientRM, masterService, resourceTracker);
addService(adminService);

this.applicationMasterLauncher = createAMLauncher();
this.rmDispatcher.register(AMLauncherEventType.class,
    this.applicationMasterLauncher);

addService(applicationMasterLauncher);

new RMNMInfo(this.rmContext, this.scheduler);

super.init(conf);
}

protected EventHandler<SchedulerEvent> createSchedulerEventDispatcher() {
    return new SchedulerEventDispatcher(this.scheduler);
}

protected Dispatcher createDispatcher() {
    return new AsyncDispatcher();
}

protected void addIfService(Object object) {
    if (object instanceof Service) {
        addService((Service) object);
    }
}

protected ResourceScheduler createScheduler() {
    return ReflectionUtils.newInstance(this.conf.getClass(
        YarnConfiguration.RM_SCHEDULER, FifoScheduler.class,
        ResourceScheduler.class), this.conf);
}

protected ApplicationMasterLauncher createAMLauncher() {
    return new ApplicationMasterLauncher(
        this.appTokenSecretManager, this.clientToAMSecretManager,
        this.rmContext);
}

private NMLivelinessMonitor createNMLivelinessMonitor() {
    return new NMLivelinessMonitor(this.rmContext
        .getDispatcher());
}

protected AMLivelinessMonitor createAMLivelinessMonitor() {
    return new AMLivelinessMonitor(this.rmDispatcher);
}

```

```

}

protected DelegationTokenRenewer createDelegationTokenRenewer() {
    return new DelegationTokenRenewer();
}

protected RAppManager createRAppManager() {
    return new RAppManager(this.rmContext, this.clientToAMSecretManager,
        this.scheduler, this.masterService, this.applicationACLsManager,
        this.conf);
}

@Private
public static class SchedulerEventDispatcher extends AbstractService
    implements EventHandler<SchedulerEvent> {

    private final ResourceScheduler scheduler;
    private final BlockingQueue<SchedulerEvent> eventQueue =
        new LinkedBlockingQueue<SchedulerEvent>();
    private final Thread eventProcessor;

    public SchedulerEventDispatcher(ResourceScheduler scheduler) {
        super(SchedulerEventDispatcher.class.getName());
        this.scheduler = scheduler;
        this.eventProcessor = new Thread(new EventProcessor());
        this.eventProcessor.setName("ResourceManager Event Processor");
    }

    @Override
    public synchronized void start() {
        this.eventProcessor.start();
        super.start();
    }

    private final class EventProcessor implements Runnable {
        @Override
        public void run() {

            SchedulerEvent event;

            while (!Thread.currentThread().isInterrupted()) {
                try {
                    event = eventQueue.take();
                } catch (InterruptedException e) {
                    LOG.error("Returning, interrupted : " + e);
                    return; // TODO: Kill RM.
                }

                try {
                    scheduler.handle(event);
                } catch (Throwable t) {
                    LOG.error("Error in handling event type " + event.getType()
                        + " to the scheduler", t);
                    return; // TODO: Kill RM.
                }
            }
        }
    }
}

```

```

@Override
public synchronized void stop() {
    this.eventProcessor.interrupt();
    try {
        this.eventProcessor.join();
    } catch (InterruptedException e) {
        throw new YarnException(e);
    }
    super.stop();
}

@Override
public void handle(SchedulerEvent event) {
    try {
        int qSize = eventQueue.size();
        if (qSize != 0 && qSize % 1000 == 0) {
            LOG.info("Size of scheduler event-queue is " + qSize);
        }
        int remCapacity = eventQueue.remainingCapacity();
        if (remCapacity < 1000) {
            LOG.info("Very low remaining capacity on scheduler event queue: "
                + remCapacity);
        }
        this.eventQueue.put(event);
    } catch (InterruptedException e) {
        throw new YarnException(e);
    }
}

@Private
public static final class ApplicationEventDispatcher implements
    EventHandler<RMAppEvent> {

    private final RMContext rmContext;

    public ApplicationEventDispatcher(RMContext rmContext) {
        this.rmContext = rmContext;
    }

    @Override
    public void handle(RMAppEvent event) {
        ApplicationId appId = event.getApplicationId();
        RMApp rmApp = this.rmContext.getRMApps().get(appId);
        if (rmApp != null) {
            try {
                rmApp.handle(event);
            } catch (Throwable t) {
                LOG.error("Error in handling event type " + event.getType()
                    + " for application " + appId, t);
            }
        }
    }

    @Private
    public static final class ApplicationAttemptEventDispatcher implements
        EventHandler<RMAppAttemptEvent> {

```

```

private final RMContext rmContext;

public ApplicationAttemptEventDispatcher(RMContext rmContext) {
    this.rmContext = rmContext;
}

@Override
public void handle(RMAppAttemptEvent event) {
    ApplicationAttemptId appAttemptID = event.getApplicationAttemptId();
    ApplicationId appAttemptId = appAttemptID.getApplicationId();
    RMApp rmApp = this.rmContext.getRMApps().get(appAttemptId);
    if (rmApp != null) {
        RMAppAttempt rmAppAttempt = rmApp.getRMAppAttempt(appAttemptID);
        if (rmAppAttempt != null) {
            try {
                rmAppAttempt.handle(event);
            } catch (Throwable t) {
                LOG.error("Error in handling event type " + event.getType()
                    + " for applicationAttempt " + appAttemptId, t);
            }
        }
    }
}

@Private
public static final class NodeEventDispatcher implements
    EventHandler<RMNodeEvent> {

    private final RMContext rmContext;

    public NodeEventDispatcher(RMContext rmContext) {
        this.rmContext = rmContext;
    }

    @Override
    public void handle(RMNodeEvent event) {
        NodeId nodeId = event.getNodeId();
        RMNode node = this.rmContext.getRMNodes().get(nodeId);
        if (node != null) {
            try {
                ((EventHandler<RMNodeEvent>) node).handle(event);
            } catch (Throwable t) {
                LOG.error("Error in handling event type " + event.getType()
                    + " for node " + nodeId, t);
            }
        }
    }
}

protected void startWepApp() {
    Builder<ApplicationMasterService> builder =
        WebApps.$for("cluster", ApplicationMasterService.class, masterService, "ws").at(
            this.conf.get(YarnConfiguration.RM_WEBAPP_ADDRESS,
                YarnConfiguration.DEFAULT_RM_WEBAPP_ADDRESS));
    if(YarnConfiguration.getRMWebAppHostAndPort(conf).
        equals(YarnConfiguration.getProxyHostAndPort(conf))) {
        AppReportFetcher fetcher = new AppReportFetcher(conf, getClientRMService());
        builder.withServlet(ProxyUriUtils.PROXY_SERVLET_NAME,

```

```

        ProxyUriUtils.PROXY_PATH_SPEC, WebAppProxyServlet.class);
        builder.withAttribute(WebAppProxy.FETCHER_ATTRIBUTE, fetcher);
    }
    webApp = builder.start(new RMWebApp(this));
}

@Override
public void start() {
    try {
        doSecureLogin();
    } catch (IOException ie) {
        throw new YarnException("Failed to login", ie);
    }

    startWepApp();
    DefaultMetricsSystem.initialize("ResourceManager");
    try {
        rmDTSecretManager.startThreads();
    } catch (IOException ie) {
        throw new YarnException("Failed to start secret manager threads", ie);
    }

    super.start();

    /*synchronized(shutdown) {
        try {
            while(!shutdown.get()) {
                shutdown.wait();
            }
        } catch (InterruptedException ie) {
            LOG.info("Interrupted while waiting", ie);
        }
    }*/
}

protected void doSecureLogin() throws IOException {
    SecurityUtil.login(this.conf, YarnConfiguration.RM_KEYTAB,
        YarnConfiguration.RM_PRINCIPAL);
}

@Override
public void stop() {
    if (webApp != null) {
        webApp.stop();
    }
    rmDTSecretManager.stopThreads();

    /*synchronized(shutdown) {
        shutdown.set(true);
        shutdown.notifyAll();
    }*/

    DefaultMetricsSystem.shutdown();

    super.stop();
}

protected ResourceTrackerService createResourceTrackerService() {
    return new ResourceTrackerService(this.rmContext, this.nodesListManager,

```

```

        this.nmLivelinessMonitor, this.containerTokenSecretManager);
    }

    protected RMDelegationTokenSecretManager
        createRMDelegationTokenSecretManager() {
        long secretKeyInterval =
            conf.getLong(YarnConfiguration.DELEGATION_KEY_UPDATE_INTERVAL_KEY,
                YarnConfiguration.DELEGATION_KEY_UPDATE_INTERVAL_DEFAULT);
        long tokenMaxLifetime =
            conf.getLong(YarnConfiguration.DELEGATION_TOKEN_MAX_LIFETIME_KEY,
                YarnConfiguration.DELEGATION_TOKEN_MAX_LIFETIME_DEFAULT);
        long tokenRenewInterval =
            conf.getLong(YarnConfiguration.DELEGATION_TOKEN_RENEW_INTERVAL_KEY,
                YarnConfiguration.DELEGATION_TOKEN_RENEW_INTERVAL_DEFAULT);

        return new RMDelegationTokenSecretManager(secretKeyInterval,
            tokenMaxLifetime, tokenRenewInterval, 3600000);
    }

    protected ClientRMService createClientRMService() {
        return new ClientRMService(this.rmContext, scheduler, this.rmAppManager,
            this.applicationACLsManager, this.rmDTSecretManager);
    }

    protected ApplicationMasterService createApplicationMasterService() {
        return new ApplicationMasterService(this.rmContext,
            this.appTokenSecretManager, scheduler);
    }

    protected AdminService createAdminService(
        ClientRMService clientRMService,
        ApplicationMasterService applicationMasterService,
        ResourceTrackerService resourceTrackerService) {
        return new AdminService(this.conf, scheduler, rmContext,
            this.nodesListManager, clientRMService, applicationMasterService,
            resourceTrackerService);
    }

    @Private
    public ClientRMService getClientRMService() {
        return this.clientRM;
    }

    /**
     * return the scheduler.
     * @return the scheduler for the Resource Manager.
     */
    @Private
    public ResourceScheduler getResourceScheduler() {
        return this.scheduler;
    }

    /**
     * return the resource tracking component.
     * @return the resource tracking component.
     */
    @Private
    public ResourceTrackerService getResourceTrackerService() {

```



```

    return this.resourceTracker;
}

@Private
public ApplicationMasterService getApplicationMasterService() {
    return this.masterService;
}

@Private
public ApplicationACLsManager getApplicationACLsManager() {
    return this.applicationACLsManager;
}

@Override
public void recover(RMState state) throws Exception {
    resourceTracker.recover(state);
    scheduler.recover(state);
}

public static void main(String argv[]) {
    StringUtils.startupShutdownMessage(ResourceManager.class, argv, LOG);
    try {
        Configuration conf = new YarnConfiguration();
        Store store = StoreFactory.getStore(conf);
        ResourceManager resourceManager = new ResourceManager(store);
        Runtime.getRuntime().addShutdownHook(
            new CompositeServiceShutdownHook(resourceManager));
        resourceManager.init(conf);
        //resourceManager.recover(store.restore());
        //store.doneWithRecovery();
        resourceManager.start();
    } catch (Throwable t) {
        LOG.fatal("Error starting ResourceManager", t);
        System.exit(-1);
    }
}

}

}

/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Database;

import Model.Block;
import Model.Log;
import com.mysql.jdbc.CommunicationsException;
import java.math.BigInteger;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Arrays;

```

```

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javax.swing.JOptionPane;

/**
 *
 * @author user
 */
public class DatabaseHelper {
    private static final String classname = "com.mysql.jdbc.Driver";
    private static final String url = "jdbc:mysql://localhost:3306/improvedyarndata";
    private static final String username = "root";
    private static final String password = "";

    private static final String BLOCK = "blocks";

    private final String blockTable = "create table if not exists "+BLOCK+" (ID int auto_increment not
    null primary key, virtualmachine varchar(100), filename varchar(100), filecontent LONGTEXT, size
    varchar(20), file varchar(100), username varchar(100))";

    private Connection connection;

    public DatabaseHelper(String user)
    {
        try{
            Class.forName(classname);
            //creating database file if not exist
            connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mysql",
            username, password);
            PreparedStatement pst = connection.prepareStatement("create database if not exists
            "+user);
            pst.execute();
            pst.close();

            connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/"+user,
            username, password);
            Statement stmt = connection.createStatement();
            stmt.execute(blockTable);

            stmt.close();

        }catch(ClassNotFoundException ex)
        {
            ex.printStackTrace();
            JOptionPane.showMessageDialog(null, ex);
        }catch(SQLException sq)
        {
            sq.printStackTrace();
            if(sq instanceof CommunicationsException)
                JOptionPane.showMessageDialog(null, "Connection lost");
            else
                JOptionPane.showMessageDialog(null, sq);
        }
    }

    public void addBlock(Block block)

```

```

    {
        //String sql = "insert into "+block.getVirtualMachine()+"(virtualmachine, filename, filecontent,
size, file, username) values('"+block.getVirtualMachine()+"', '"+block.getFilename()+"',
 '"+block.getFilecontent()+"', '"+block.getSize()+"', '"+block.getFile()+"',
 '"+block.getUsername()+"')";
        String sql2 = "insert into "+BLOCK+"(virtualmachine, filename, filecontent, size, file, username)
values('"+block.getVirtualMachine()+"', '"+block.getFilename()+"', '"+block.getFilecontent()+"',
 '"+block.getSize()+"', '"+block.getFile()+"', '"+block.getUsername()+"')";

        try{
            Statement stmt = connection.createStatement();
//            stmt.executeUpdate(sql);
            stmt.executeUpdate(sql2);
            stmt.close();
            //JOptionPane.showMessageDialog(null, "New sales added");
        }catch(SQLException sq)
        {
            sq.printStackTrace();
        }
    }

    public boolean isBlock(Block block){
        String sql = "select * from "+BLOCK+" where filename = '"+block.getFilename()+"' AND file =
 '"+block.getFile()+"' AND username = '"+block.getUsername()+"'";
        try{
            Statement stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery(sql);
            if(rs.next()){
                //if(us.equalsIgnoreCase(rs.getString("username"))) ||
username.equalsIgnoreCase("email"))
                return true;
            }
            rs.close();
            stmt.close();
        }catch(SQLException sq)
        {
            sq.printStackTrace();
        }
        return false;
    }

    public int getNumberOfBlocks(String machine, String username, String filename){
        String sql = "select * from "+BLOCK+" where username = '"+username+"' and virtualmachine =
 '"+machine+"' and file = '"+filename+"'";
        int total = 0;
        try{
            Statement st = connection.createStatement();
            ResultSet rs = st.executeQuery(sql);
            while(rs.next()){

                total = total + 1;
            }
        }catch(SQLException sq){
            sq.printStackTrace();
        }
        return total;
    }

    public int getNumberOfBlocks(String machine, String username){

```

```

String sql = "select * from "+BLOCK+" where username = '"+username+"' and virtualmachine =
 '"+machine+"'";
int total = 0;
try{
    Statement st = connection.createStatement();
    ResultSet rs = st.executeQuery(sql);
    while(rs.next()){

        total = total + 1;
    }
}catch(SQLException sq){
    sq.printStackTrace();
}
return total;
}

public long getBytesCount(String username){
String sql = "select * from "+BLOCK+" where username = '"+username+"'";
long total = 0;
try{
    Statement st = connection.createStatement();
    ResultSet rs = st.executeQuery(sql);
    while(rs.next()){
        total += Long.parseLong(rs.getString("size"));
    }
}catch(SQLException sq){
    sq.printStackTrace();
}
return total;
}

public long getBytesCount(String file, String username){
String sql = "select * from "+BLOCK+" where username = '"+username+"' AND file = '"+file+"'";
long total = 0;
try{
    Statement st = connection.createStatement();
    ResultSet rs = st.executeQuery(sql);
    while(rs.next()){
        total += Long.parseLong(rs.getString("size"));
    }
}catch(SQLException sq){
    sq.printStackTrace();
}
return total;
}

public String getFilecontent(String filename, String username){
String content = "";
String sql = "select * from "+BLOCK+" where username = '"+username+"' AND filename =
 '"+filename+"'";
try {
    Statement st = connection.createStatement();
    ResultSet rs = st.executeQuery(sql);
    while (rs.next()) {
        content = rs.getString("filecontent");
        break;
    }
    rs.close();
    st.close();
}
}

```

```

    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
    }
    return content;
}

public List<String> getServernames(String username){
    List<String> servernameList = new ArrayList<>();
    String sql = "select * from "+BLOCK+" where username = '"+username+"'";
    try {
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            servernameList.add(rs.getString("virtualmachine"));
        }
        rs.close();
        st.close();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
    }
    return servernameList;
}

public List<String> getNames(String username){
    List<String> nameList = new ArrayList<>();
    String sql = "select * from "+BLOCK+" where username = '"+username+"'";
    try {
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            nameList.add(rs.getString("filename"));
        }
        rs.close();
        st.close();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
    }
    return nameList;
}

public List<byte[]> getBlock(String username){
    List<byte[]> blockList = new ArrayList<>();
    String sql = "select * from "+BLOCK+" where username = '"+username+"'";
    try {
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            blockList.add(rs.getString("filecontent").getBytes());
        }
        rs.close();
        st.close();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
    }
    return blockList;
}

public List<Block> getBlock(String machine, String username){

```

```

List<Block> blockList = new ArrayList<>();
String sql = "select * from "+BLOCK+" where username = '"+username+"' and virtualmachine =
'+machine+'";
try {
    Statement st = connection.createStatement();
    ResultSet rs = st.executeQuery(sql);
    while (rs.next()) {
        Block block = new Block(rs.getString("virtualmachine"), rs.getString("filename"),
rs.getString("filecontent"), rs.getString("size"), rs.getString ("file"), rs.getString ("username"));
        blockList.add(block);
    }
    rs.close();
    st.close();
} catch (SQLException ex) {
    JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
}
return blockList;
}

public Block getBlockObj(String machine, String username, String filename){
    Block block = new Block();
    String sql = "select * from "+BLOCK+" where username = '"+username+"' and virtualmachine =
'+machine+' and filename = '"+filename+'";
    try {
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            block.setVirtualMachine(rs.getString("virtualmachine"));
            block.setFilename(rs.getString("filename"));
            block.setFilecontent(rs.getString("filecontent"));
            block.setSize(rs.getString("size"));
            block.setFile(rs.getString ("file"));
            block.setUsername(rs.getString ("username"));
        }
        rs.close();
        st.close();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
    }
    return block;
}

public List<String> getBlockFilesNames(String machine, String username){
    List<String> blockList = new ArrayList<>();
    String sql = "select * from "+BLOCK+" where username = '"+username+"' and virtualmachine =
'+machine+'";
    try {
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            blockList.add(rs.getString("filename"));
        }
        rs.close();
        st.close();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
    }
    return blockList;
}
}

```

```

private List<byte[]> listDataFromBytes(byte[] source){
    int chunksize = 6144;
    List<byte[]> result = new ArrayList<byte[]>();
    int start = 0;
    while(start < source.length){
        int end = Math.min(source.length, start + chunksize);
        result.add(Arrays.copyOfRange(source, start, end));
        //System.out.println("start: "+start+", end: "+end);
        start += chunksize;
    }
    return result;
}

public String getCapacity(BigInteger size) {
    String hrSize = "";
    double b = size.longValue();
    double k = size.longValue() / 1024.0;
    double m = ((size.longValue() / 1024.0) / 1024.0);
    double g = (((size.longValue() / 1024.0) / 1024.0) / 1024.0);
    double t = ((((size.longValue() / 1024.0) / 1024.0) / 1024.0) / 1024.0);

    DecimalFormat dec = new DecimalFormat("0.00");

    if (t > 1) {
        hrSize = dec.format(t).concat(" TB");
    } else if (g > 1) {
        hrSize = dec.format(g).concat(" GB");
    } else if (m > 1) {
        hrSize = dec.format(m).concat(" MB");
    } else if (k > 1) {
        hrSize = dec.format(k).concat(" KB");
    } else {
        hrSize = dec.format(b).concat(" Bytes");
    }

    return hrSize;
}

public BigInteger getRemainingCapacity(BigInteger size){
    BigInteger sixGB = BigInteger.valueOf(1073741824).multiply(BigInteger.valueOf(6));
    BigInteger remain = sixGB.subtract(size);
    System.out.println("remain: "+remain);
    System.out.println("used: "+size);
    return remain;
}

public List<String> getServernames(String machine, String userid){
    List<String> servernameList = new ArrayList<>();
    String sql = "select * from "+BLOCK+" where virtualmachine = '"+machine+"' and username = '"+userid+"'";
    try {
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            servernameList.add(rs.getString("virtualmachine"));
        }
        rs.close();
        st.close();
    }
}

```

```

    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
    }
    return servernameList;
}

public List<String> getNames(String machine, String userid){
    List<String> nameList = new ArrayList<>();
    String sql = "select * from "+BLOCK+" where virtualmachine = '"+machine+"' and username = '"+userid+"'";
    try {
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            nameList.add(rs.getString("filename"));
        }
        rs.close();
        st.close();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
    }
    return nameList;
}

public List<String> getServernames(String machine, String userid, String filename){
    List<String> servernameList = new ArrayList<>();
    String sql = "select * from "+BLOCK+" where virtualmachine = '"+machine+"' and username = '"+userid+"' and file = '"+filename+"'";
    try {
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            servernameList.add(rs.getString("virtualmachine"));
        }
        rs.close();
        st.close();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
    }
    return servernameList;
}

public List<String> getNames(String machine, String userid, String filename){
    List<String> nameList = new ArrayList<>();
    String sql = "select * from "+BLOCK+" where virtualmachine = '"+machine+"' and username = '"+userid+"' and file = '"+filename+"'";
    try {
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            nameList.add(rs.getString("filename"));
        }
        rs.close();
        st.close();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
    }
    return nameList;
}

```



```

}

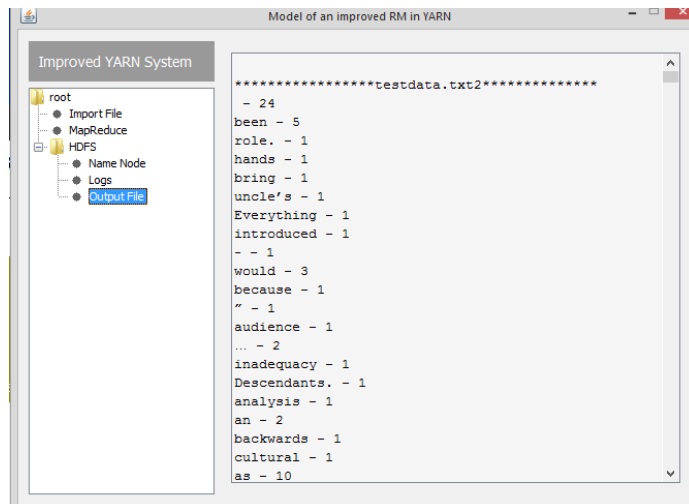
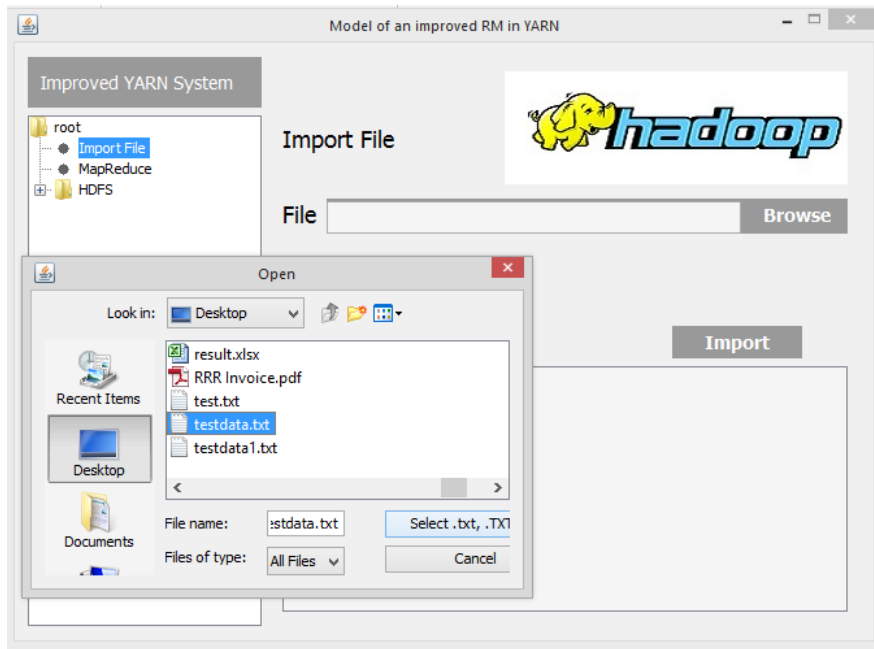
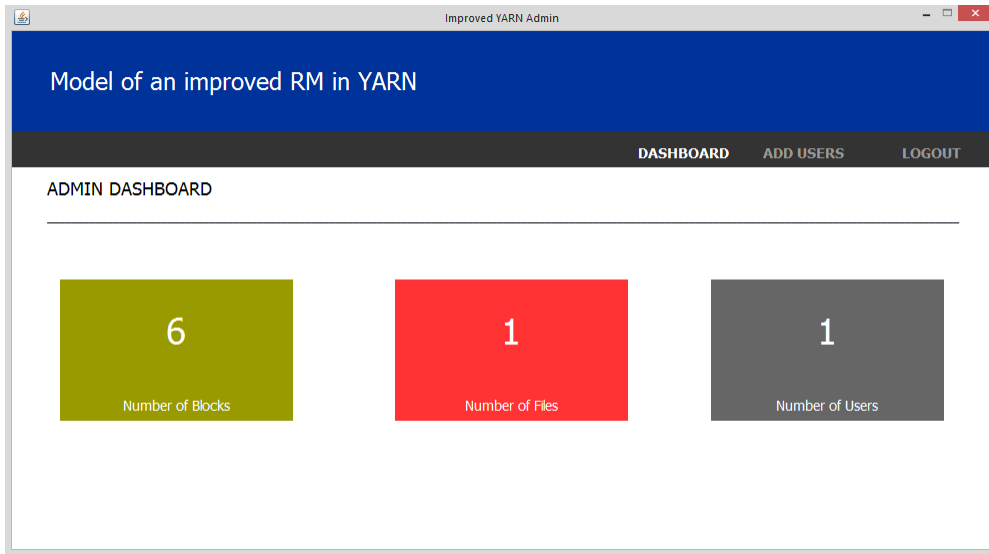
public List<byte[]> getBlock(String machine, String userid, String filename){
    List<byte[]> blockList = new ArrayList<>();
    String sql = "select * from "+BLOCK+" where virtualmachine = '"+machine+"' and username =
"+userid+"' and file = '"+filename+"'";
    try {
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            blockList.add(rs.getString("filecontent").getBytes());
        }
        rs.close();
        st.close();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(null, ex.getLocalizedMessage());
    }
    return blockList;
}

public void close(){
    try{
        connection.close();
    }catch(SQLException sq){
        JOptionPane.showMessageDialog(null, sq.getLocalizedMessage());
    }
}
}

```


## APPENDIX B

### SAMPLE OUTPUT



Virtual Machine

Virtual\_Machine\_1



```

uploading ..... testdata.txt2
uploading ..... testdata.txt5

*****testdata.txt2*****
- 24
been - 5
role. - 1
hands - 1
bring - 1
uncle's - 1
Everything - 1
introduced - 1
-- 1


```

localhost Disconnect Clear text area

Model of an improved RM in YARN

Improved YARN System

- root
  - Import File
  - MapReduce
  - HDFS
    - Name Node
    - Logs
    - Output File



Logs

Application: testdata.txt

BLOCKS	START TIME	END TIME
testdata.txt2	06.07.2018 14:11:20.367	06.07.2018 14:11:20.414
testdata.txt5	06.07.2018 14:11:20.367	06.07.2018 14:11:20.435
testdata.txt1	06.07.2018 14:11:20.367	06.07.2018 14:11:20.414
testdata.txt4	06.07.2018 14:11:20.367	06.07.2018 14:11:20.438
testdata.txt0	06.07.2018 14:11:20.367	06.07.2018 14:11:20.402
testdata.txt3	06.07.2018 14:11:20.367	06.07.2018 14:11:20.442