

**UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO**

**DESENVOLVIMENTO DE UM
ESCALONADOR SENSÍVEL AO CONTEXTO
PARA O APACHE HADOOP**

TRABALHO DE GRADUAÇÃO

Guilherme Weigert Cassales

Santa Maria, RS, Brasil

2014

DESENVOLVIMENTO DE UM ESCALONADOR SENSÍVEL AO CONTEXTO PARA O APACHE HADOOP

Guilherme Weigert Cassales

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da
Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para
a obtenção do grau de

Bacharel em Ciência da Computação

Orientadora: Prof^a. Dr^a. Andrea Schwertner Charão

**365
Santa Maria, RS, Brasil**

2014

**Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação**

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

**DESENVOLVIMENTO DE UM ESCALONADOR SENSÍVEL AO
CONTEXTO PARA O APACHE HADOOP**

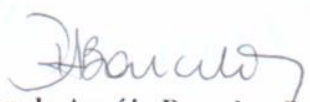
elaborado por
Guilherme Weigert Cassales

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:


Andrea Schwertner Charão, Prof^a. Dr^a.
(Presidente/Orientadora)


Benhur de Oliveira Stein, Prof. Dr. (UFSM)


Patrícia Pitthan de Araújo Barcelos, Prof^a. Dr^a. (UFSM)

Santa Maria, 20 de Janeiro de 2014.

AGRADECIMENTOS

Agradeço à minha família por todo apoio, não só durante os longos anos de graduação, mas em todas as etapas de minha vida.

À minha namorada Raíssa, não só pelo apoio em mais um TG e mais uma graduação, mas por todo crescimento conjunto que tivemos durante um ano e quase noventa meses.

Aos amigos mais próximos, que mesmo estando fisicamente distantes ou que a rotina tenha impedido um convívio diário, por sempre estarem dispostos a compartilhar experiências.

À professora Andrea Charão, por todo suporte prestado tanto no papel de orientadora como de coordenadora do curso.

Agradeço também, a todos que tornaram possível e/ou participaram na realização deste trabalho.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

DESENVOLVIMENTO DE UM ESCALONADOR SENSÍVEL AO CONTEXTO PARA O APACHE HADOOP

AUTOR: GUILHERME WEIGERT CASSALES

ORIENTADORA: ANDREA SCHWERTNER CHARÃO

Local da Defesa e Data: Santa Maria, 20 de Janeiro de 2014.

Hoje em dia, o volume de dados gerados é muito maior do que a capacidade de processamento dos computadores. Como solução para esse problema, algumas tarefas podem ser paralelizadas ou distribuídas. O *framework Apache Hadoop* (Apache Hadoop, 2013), é uma delas e poupa o programador as tarefas de gerenciamento, como tolerância à falhas, particionamento dos dados entre outros. Um problema no escalonador do *Apache Hadoop* é que seu foco é em ambientes homogêneos, o que muitas vezes não é possível de se manter. O foco deste trabalho foi na melhora de um escalonador já existente, possuindo como objetivo torná-lo sensível ao contexto, permitindo que as capacidades físicas de cada máquina sejam consideradas na hora da distribuição das tarefas submetidas. Optou-se por inserir coletores de informações de contexto (memória e cpu) no CapacityScheduler, tornando o comportamento desse sensível ao contexto. Através das mudanças feitas e de experimentos feitos usando um benchmark bem conhecido (TeraSort), foi possível demonstrar uma melhora no escalonamento em relação ao escalonador original com a configuração padrão.

Palavras-chave: Apache Hadoop. Escalonador. Sensibilidade ao Contexto.

ABSTRACT

Undergraduate Final Work
Undergraduate Program in Computer Science
Federal University of Santa Maria

DEVELOPMENT OF A CONTEXT-AWARE SCHEDULER FOR APACHE HADOOP

AUTHOR: GUILHERME WEIGERT CASSALES

ADVISOR: ANDREA SCHWERTNER CHARÃO

Defense Place and Date: Santa Maria, January 20th, 2014.

Nowadays the volume of data generated by the services provided for end users, is way larger than the processing capacity of one computer alone. As a solution to this problem, some tasks can be parallelized. The Apache Hadoop framework, is one of these parallelized solutions and it spares the programmer of management tasks such as fault tolerance, data partitioning, among others. One problem on this framework is the scheduler, which is designed for homogeneous environments. It is worth to remember that maintaining a homogeneous environment is somewhat difficult today, given the fast development of new, cheaper and more powerful hardware. This work focuses on altering the Capacity Scheduler, in order to make it more context-aware towards resources on the cluster. Making it possible to consider the the physical capacities of the machines when scheduling the submitted tasks. It was chosen to insert context information (memory and cpu) collectors on CapacityScheduler, making his scheduling more context-aware. Through the changes and experiments made using a common and well known benchmark (TeraSort), it was possible to notice a improvement on scheduling in relation to the original scheduler using the default configuration.

Keywords: Apache Hadoop. Scheduler. Context-aware.

FIGURE LIST

Figure 2.1 – General Apache Hadoop architecture	13
Figure 2.2 – General HDFS architecture (HADOOP, 2013a)	13
Figure 2.3 – General YARN architecture (HADOOP, 2013b)	14
Figure 3.1 – ResourceManager Component Diagram (HortonWorks, 2014a)	21
Figure 3.2 – NodeManager Component Diagram (HortonWorks, 2014b)	23
Figure 3.3 – Class Diagram with the main CapacityScheduler classes	24
Figure 3.4 – Class Diagram with the main RM components	25
Figure 3.5 – Flow of operations for resource granting	29
Figure 3.6 – Flow of operations for resource granting	30
Figure 3.7 – Flow of operations for resource granting	33
Figure 4.1 – Cluster memory available on default and collector implementation	35
Figure 4.2 – Cluster cores available on default and collector implementation	36
Figure 4.3 – Container assignment with default configuration	38
Figure 4.4 – Container assignment with the improved configuration	39
Figure 4.5 – Container assignment with the simulated heterogeneous environment	41
Figure A.1 – RMNode’s state machine	47
Figure A.2 – RMContainer’s state machine	48
Figure A.3 – RMAppAttempt’s state machine	49
Figure A.4 – RMApp’s state machine	50

APPENDIX LIST

APPENDIX A – Generated ResourceManager Graphs	47
APPENDIX B – Grid’5000 execution environment configuration	51
APPENDIX C – Source code edition and compilation.....	53
APPENDIX D – Example of semi-processed log from a experiment.....	54
APPENDIX E – Main code changes performed	56

INDEX

1 INTRODUCTION	10
1.1 Objective	11
1.2 Motivation	11
2 FOUNDATIONS AND LITERATURE REVIEW	12
2.1 Hadoop	12
2.1.1 Apache Hadoop Architecture	12
2.1.1.1 HDFS	13
2.1.1.2 YARN	14
2.1.2 Hadoop execution environment configuration	14
2.1.3 MapReduce	15
2.2 Context-awareness	15
2.3 Hadoop Schedulers	16
2.3.1 Hadoop Internal Scheduler	16
2.3.2 Fair Scheduler	17
2.3.3 Capacity Scheduler	17
2.4 Related work	17
3 DEVELOPMENT	20
3.1 Understanding Apache Hadoop internals	20
3.1.1 Code structure and class diagrams	21
3.1.2 State machines	25
3.2 Understanding Hadoop allocation pattern	26
3.2.1 Experiment	27
3.2.1.1 Employed methods, procedures and scenarios	27
3.2.1.2 Results and interpretation	28
3.3 Context-aware scheduling	31
3.3.1 Collector description	32
3.3.2 Collector integration with Hadoop	32
4 EXPERIMENTS AND RESULTS	34
4.1 Collector integration test	34
4.1.1 Hardware and Software configuration	34
4.1.2 Procedures	34
4.1.3 Results and interpretation	35
4.2 Original CapacityScheduler X Context-aware CapacityScheduler	36
4.2.1 Hardware and Software configuration	36
4.2.2 Procedures	36
4.2.3 Results and interpretation	37
4.3 Heterogeneity simulation	39
4.3.1 Hardware and Software configuration	39
4.3.2 Procedures	40
4.3.3 Results and interpretation	40
5 CONCLUSION AND FUTURE WORK	42
REFERENCES	43
APPENDIX	46

1 INTRODUCTION

One of the major IT companies nowadays, known as Google (Google, 2013), had the initial idea of a way to process a huge data volume generated by its servers. This approach would later be known as MapReduce, built by two separate steps, Map and Reduce, each step based on a functional language function. At the same time, a Yahoo! (Yahoo, 2013) led project was starting the implementation of MapReduce for its own system, which would then become a whole new project, named Apache Hadoop (Apache Hadoop, 2013).

Today the Apache Hadoop framework has a very active community of both developers and users, however there are some characteristics that weren't changed from the day the framework was first designed. Among these characteristics there is one very detrimental and prone to bad performance issues, which is the focus on homogeneous environments. It is known that maintaining a totally homogeneous environment is harder and harder as the time passes, requiring either a huge initial investment or a huge effort in order to replace faulty hardware without changing the component capability.

The MapReduce's task performance inside Hadoop is tightly tied to the scheduler (KUMAR et al., 2012). Since it is an open source project, it is possible to change the scheduler aiming to make it capable of better adapting to heterogeneity while at the same time presenting a performance improvement.

A key characteristic in Hadoop's transition to heterogeneous environments is the context-aware capability. The definition of context can vary from one application to another, but as a rule of thumb it is some information that the application can use as base for decision making. When an application is context-aware, it will detect and adapt to the changes in the environment (Kirsch-Pinheiro, M. and Steffemel, L. A., 2013).

In the present work, the context to which the application will have to adapt is related to the physical configuration of the machines that compose the Hadoop cluster, allowing the scheduler to work with real data collected from the machines and not suppositions as the default Hadoop configuration implies. In a more complex degree, the present work is just a part of a bigger project called PER-MARE (STEFFEMEL et al., 2013), which has the objective of adapting to more environment variations, as the insertion and removal of nodes in real time.

1.1 Objective

The main objective of this work is to improve Hadoop through a context-aware scheduling, which will provide better performance and adaptation on heterogeneous environments.

1.2 Motivation

Today, some processing tasks that used to be made through huge mainframes and servers are gradually transitioning to big clusters, which are composed of computers with more accessible prices and easily bought in the market.

Even though the Apache Hadoop framework has clusters as its target, it was designed and implemented under a specific assumption. The framework's better performance is achieved when it is running on a homogeneous cluster, in other words, when all nodes have the same resources. The problem is that given today's hardware development, it might take the system to a point where it is not possible or at least not profitable to maintain cluster homogeneity.

Since the default configuration of the scheduler tells that every node on the cluster has the same amount of resources, if a more powerful node is inserted all that extra capacity will be wasted. This happens because the cluster will not collect the real configuration, but the parameter set in a XML file as the node capacity. The opposite is also troublesome, if a less powerful node is inserted, the cluster will use it as if it had more potential, possibly overloading that node with more tasks that it can handle and causing errors or performance issues.

The present work is relevant, as it's objective is based on adaptation and improvement of an already existent technology. With the improved scheduler, not only will the Apache Hadoop clusters have a possibility to improve cluster's resource utilization, as the framework itself will be better prepared and capable of adapting to new heterogeneous environments in an easier and smoother way.

2 FOUNDATIONS AND LITERATURE REVIEW

The tools and paradigms used on the work are: framework Apache Hadoop, MapReduce, context-awareness, as well as related works. They will be further explained in the following sections.

2.1 Hadoop

The Apache Hadoop framework was actually originated from another Apache (Apache, 2013) project, the Apache Nutch (Apache Nutch, 2013), which was an open source web search engine started on 2002. Unfortunately, Apache Nutch was facing problems due to its architecture, that was happening almost at the same time that Google published an article describing the architecture used on their distributed file system (GFS). The Nutch developers saw that a similar architecture to the GFS would solve Nutch's scalability problem.

In 2004 the Nutch developers began to implement the idea and the result was named Nutch Distributed Filesystem (NDFS). As the project advanced it began to take bigger and bigger proportions until 2006, when a new project was made because the advancements were bigger than Nutch's purpose. This new project was named Hadoop. The Hadoop framework has the purpose of facilitating distributed processing through the MapReduce paradigm.

2.1.1 Apache Hadoop Architecture

In a general view, it is possible to break Apache Hadoop in two parts. These parts are named Hadoop Distributed File System (HDFS) and Yet Another Resource Negotiator (YARN). Figure 2.1 presents the aforementioned separation.

HDFS is responsible for the data storage, which is required in order to execute jobs. It is the component which will replicate the data in order to provide fault tolerance, data distribution according to the requirements of each node, among other things.

YARN is the other half of Apache Hadoop, it is responsible for processing tasks from submitted jobs. It is through YARN components that the MapReduce tasks are executed, which consequently makes YARN the manager for all cluster resources.

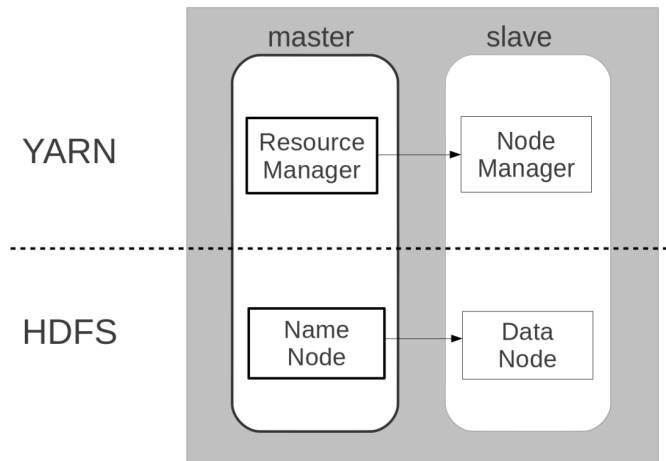


Figure 2.1 – General Apache Hadoop architecture

2.1.1.1 HDFS

HDFS is in huge part responsible for Hadoop’s good performance, because it has the task to not overloading the network with file transfers. Thanks to HDFS, the file access is always local, it means that every node will receive the file portion relative to its workload, thus preventing unnecessary replication other than for security and fault tolerance replication.

One problem of this approach is that Hadoop has a big latency, making its use inadvisable to critical or real time applications. The HDFS may be further divided in two services, NameNode and DataNode, responsible for the management of data in cluster level and local level, respectively. Figure 2.2 presents the basic HDFS architecture scheme.

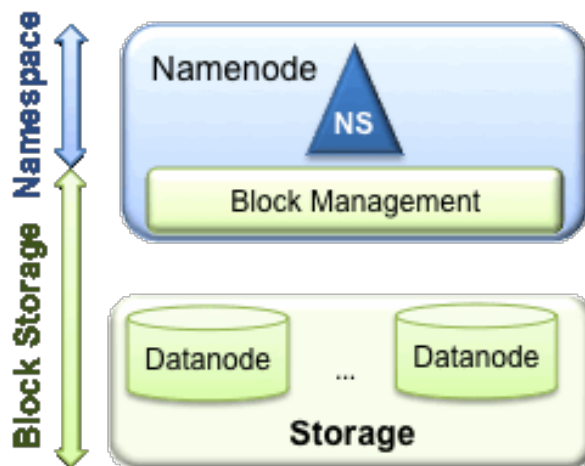


Figure 2.2 – General HDFS architecture (HADOOP, 2013a)

2.1.1.2 YARN

YARN is the portion of Apache Hadoop responsible for MapReduce execution, therefore the execution of management and processing tasks are controlled by YARN. In making the processing tasks totally independent from data storage tasks, the Apache Hadoop opens a lot of possibilities for its utilization. Just like HDFS, YARN can be further divided into two services, Resource Manager and Node Manager, responsible for system resource management and local resource management, respectively. Figure 2.3 presents the basic YARN's architecture scheme.

Although not shown on the figure, each service has many internal modules, for example the Resource Manager has the scheduler and the Applications Manager, which could still be further divided into sub-modules. The present work aims to present an improved solution for the scheduling and resource collection employed on Hadoop so it can adapt better to the environment.

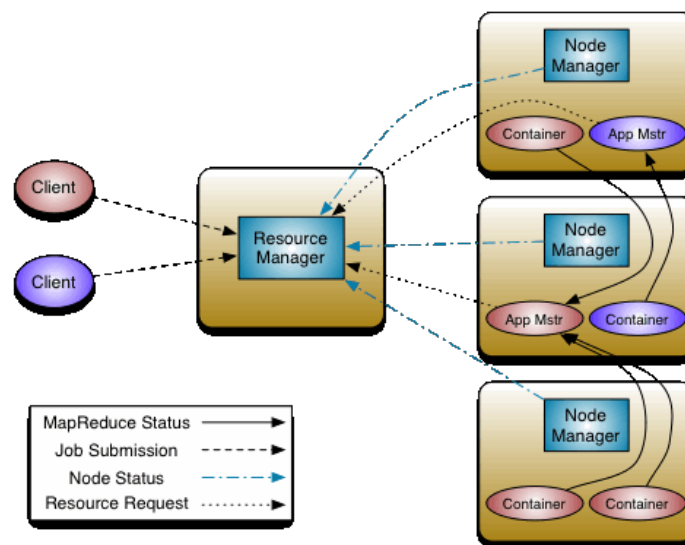


Figure 2.3 – General YARN architecture (HADOOP, 2013b)

2.1.2 Hadoop execution environment configuration

A correctly configured Hadoop environment has a few pre-requisites besides having access between the nodes in a network. Every node must have in its Hadoop installation many files, in XML format, containing the Virtual Machines' (VMs) configuration on that node.

The XML files are: core-site.xml, yarn-site.xml, mapred-site.xml, hdfs-site.xml. Each one of these files will contain one Hadoop service configuration. In case a parameter isn't set

on these files, the VM will get the value from one of the *-default.xml files which contains the default Hadoop configuration. Although it may seem like an automated configuration, it is only a default value Hadoop will use in case no other value has been specified.

2.1.3 MapReduce

The MapReduce paradigm, already mentioned many times on the present work, breaks down the processing of tasks in two steps. These two steps are derived from functional language functions Map and Reduce which, just like their original implementations, work based on key and value tuples. The standard work-flow of a MapReduce application starts with the Map function receiving an input file and searching for application desired information, after this information is found it creates key and value tuples.

Once the tuples are made the Map function sends them to Reduce function, where the keys will be processed and reduced to intelligible data. The Hadoop's greatest advantage is that, given a properly configured environment, the programmer can focus his attention and efforts on solving the tasks through MapReduce paradigm and not on making the tasks work properly distributed. Thus instead of having to worry about fault tolerance, scalability, and many other characteristics, the programmer will just focus on his algorithm.

2.2 Context-awareness

Given the interactivity of today's systems, it is possible to note some context-aware applications already in use on most of them. For example, when a site is accessed through a mobile device it will automatically load the mobile version on the device in order to have optimized spacing and content designed for that kind of devices. Another example is when the browsers take into account geographical data from the user point of access and refine the search results towards that area. Also, it is possible to use an user navigation historic to predict which products and offers will arouse more interest on that person.

All the above cases exemplified a situation where the application used some context in order to make a decision, making that application context-aware. But what really means to be a context-aware application? In order to answer this question there are two things that must be understood first, the definition of context and context-aware.

Starting with the first, what would the so called context mean. This definition is funda-

mental in order to understand what is a context-aware system and context-aware itself (Kirsch-Pinheiro, M. and Steffemel, L. A., 2013). Context can assume a lot of meanings depending on the situation, (DEY, 2001) defines context as any information possible to be used in order to characterize an entity (person, place or object) considered relevant to the interaction between user and application.

Knowing what context is, it is possible to understand the context-aware definition made by (MAAMAR; BENSLIMANE; NARENDRA, 2006) in which he claims that context-aware is the ability of an application to detect and respond to changes in the execution environment. Which then can be complemented by the following definition made by (BALDAUF; DUST-DAR; ROSENBERG, 2007), in which he defines a context-aware system as something capable of adapting it's operations to the actual context without explicit user intervention and therefore improve the usability and efficacy.

Although there are many different methods to improve the MapReduce's performance through employment of context information, (Kirsch-Pinheiro, M. and Steffemel, L. A., 2013) suggests three possible ways to do so. These three methods are: automated node configuration on Hadoop installation/start-up, management of node removal and addition in real time and finally through scheduler task distribution according to the available resources and already executing tasks at any given time. The present work used a hybrid approach between the second and third suggestions, as it will collect the real node data and also impact on scheduling resource distribution.

2.3 Hadoop Schedulers

One of the Hadoop's main components is the scheduler. It is responsible for the resource and workload distribution among the environment.

2.3.1 Hadoop Internal Scheduler

The standard Hadoop scheduler, was implemented aiming to support only batch job submission. It takes the first job received and executes it, making a queue for subsequent submissions. This scheduler supports five levels of priority, yet the choice for the next job to be executed will always take into account the submission time.

2.3.2 Fair Scheduler

Used for computing of small batch jobs that have the same input data size, using a two level scheduling in order to distribute the resources equally (Fair Scheduler, 2013). The superior level, usually allocates queues for each user, using a weighted fair algorithm. The second level allocates the resources inside each user queue, the scheduling in this level uses the same algorithm as Internal Scheduler.

2.3.3 Capacity Scheduler

This scheduler was implemented aiming the cases where the Hadoop environment is shared among various companies or has many distributed parts on places under multi-tenant responsibility. It focuses on guarantees that a minimum share will always be available for each company. The benefit comes from the fact that different organizations have processing peaks at different times, therefore the organization using more capacity, will use the idle capacity of the other organizations. This is the most complete scheduler provided by Hadoop. It is able to track the resources registered within the RM, although not consistent with the reality, and monitor which resources are free and which are being used.

2.4 Related work

Beyond the schedulers patched together with Hadoop, there are other implementations that sought to solve a specific necessity that the standard schedulers do not offer support. A bibliographic research was made aiming to analyze the works already published involving Hadoop and having as purpose adapting or changing the scheduling. Besides, it was sought to identify which techniques were the most used and which were the most common objectives of the developed work. Following, there is a list of the related works containing a brief abstract of their proposals, used context and expected objective with the interventions.

- CASH (Context Aware Scheduler for Hadoop) (KUMAR et al., 2012), this work had the objective of improving the cluster overall performance. Assuming the hypothesis that a huge part of the jobs are periodic and executed at roughly the same time, while also having similar CPU, network and disk usage characteristics. The work still takes into account that with time the nodes tend to become heterogeneous. With the intention to

solve this situation, a scheduler that classifies both jobs and nodes with respect to its CPU or I/O potential was implemented, it can then distribute the jobs for machines with a more appropriated configuration regarding the job's nature.

- LATE (Longest Approximation Time to End) (ZAHARIA et al., 2008), following what the name suggests, in this work context information is regarded to a job's estimated time to end, this time is based on a heuristic that makes the connection between elapsed time and the score. Score is a value that represents how much of the job has already been processed. This information is used to generate a threshold which will determine when a task is slow enough to start a new speculative copy on another possibly faster machine. The objective of this work was to reduce the response time in large clusters executing many jobs of short duration.
- A Dynamic MapReduce Scheduler for Heterogeneous Workloads (TIAN et al., 2009), the authors of this work used the technique of classifying the jobs and machines according to the I/O and CPU potential. Just like CASH the main objective is the improvement of cluster performance. One of the differences however, is that this implementation uses a three queue scheduler.
- SAMR (A Self-adaptative MapReduce) (CHEN et al., 2010), this implementation follow the same idea from LATE, where the context information refers to the job progress calculation and is used to identify if it is necessary to launch a speculative task. However, this solution changes a bit the calculation of progress by inserting information about the job's execution environment, the algorithm takes into account historical informations contained in each node and uses it to adjust weight of each processing step.
- COSHH (A Classification and Optimization based Scheduler for Heterogeneous Hadoop Systems) (RASOOLI; DOWN, 2012), a wider proposal if compared to the rest of the solutions, this one takes into account informations not specified about the system. It's performance gain is achieved through classification of jobs in classes, then it searches the whole cluster for machines that match the same class as the job. This search is made by an algorithm that reduces the search sample size, thus improving search response and performance. The objective of this solution is the improvement of medium job completion time, besides offering a good performance when using the minimum share and providing a fair distribution of resources.

- Quincy (ISARD et al., 2009), this solution was not proposed solely for Hadoop environments, but still applicable to it. Possessing the objective of improving general cluster performance, uses the distribution of resources as context information and modifies the traditional way of treatment to these. In using a more dynamic approach, the solution maps the resources in a capacity-demand graph and calculates the optimum scheduling from a global cost function.
- Improving MapReduce Performance through Data Placement in heterogeneous Hadoop Clusters (XIE et al., 2010), this solution aims to provide better performance on jobs through better data placement, using mainly the data locality as decision making information. The performance gain is achieved by the data re-balancing in nodes, leaving faster nodes with more data. This lowers the cost of speculative tasks and also of data transfers through the network.

After studying the related works, it is possible to note that many of them have the reduction of response time or improvement of overall performance as objective, which are slightly different from the present work that aims firstly for a better Hadoop adaptation in a heterogeneous environment, which will consequently provide a better performance. Regarding the context, there were many contexts used although, it is possible to identify recurrent contexts such as: job classification according I/O and CPU potential, job progress evaluation in order to launch or not speculative tasks.

3 DEVELOPMENT

This chapter describes the necessary steps to achieve this work's objectives. Given the complexity of the Hadoop framework, this work was divided in four stages, this decision was taken aiming mainly on understanding the context in which the scheduler will have to adapt and how it will be achieved, leaving the implementation stage to the end when every other prerequisite for the code insertion on Hadoop is already concluded. The first stage had the focus on installation and configuration of versions 1.0.4 and 2.0.3 (YARN), in order to enlarge the knowledge about the environment requirements necessary to utilize the framework. The second stage had the objective of installing and preparing the environment to compile the code, since the objective is changing the Apache Hadoop behavior and that is done by making changes in the code. The third stage was destined to the Hadoop's architecture study, through the standard Hadoop schedulers and related classes code and the Resource Manager and Node Manager state machines. Finally the fourth stage is the development and testing stage.

3.1 Understanding Apache Hadoop internals

Aiming to insert context-awareness on a scheduler, it is necessary that the architecture of Apache Hadoop is comprehended. This stage of the project had the objective to identify the dependencies between the classes, besides which classes would be necessary to implement and how would this implementation take place.

Since a working scheduler requires many abstract classes and interfaces to be implemented, it is a good practice to know the origins of these components and how they influence the final class. Also, through the architecture study it is possible to identify the resources supported by the framework, making it possible to decide the scheduling strategies.

Two methods were used in order to study the architecture. The first method consisted on source code study, while the second method was a study of the state machines that dictate the functioning of Resource Manager. This stage was planned aiming the identification of all components inside a scheduler, since the implemented interfaces and abstract classes to the scheduling criteria and how these are applied on available schedulers.

3.1.1 Code structure and class diagrams

Given the framework's complexity, it was decided to use an IDE on the execution of the first method, in this case the chosen IDE was IntelliJ IDEA (jetBrains, 2013). Once the study was finished, it was possible to generate a series of class diagrams. The generated diagrams were used in conjunction with HortonWorks Diagram in order to make the understanding of the whole YARN framework easier.

The first diagrams used in this step were the ResourceManager and NodeManager component diagrams, which provided a better high level understanding of the components that compose the ResourceManager and NodeManager.

The figure 3.1 illustrates the high level view of the ResourceManager. It is possible to note many modules which doesn't matter to the context of this work, such as: Security, DelegationTokenRenewer, among others. Even so, the notion this figure passes has a high value to the understanding.

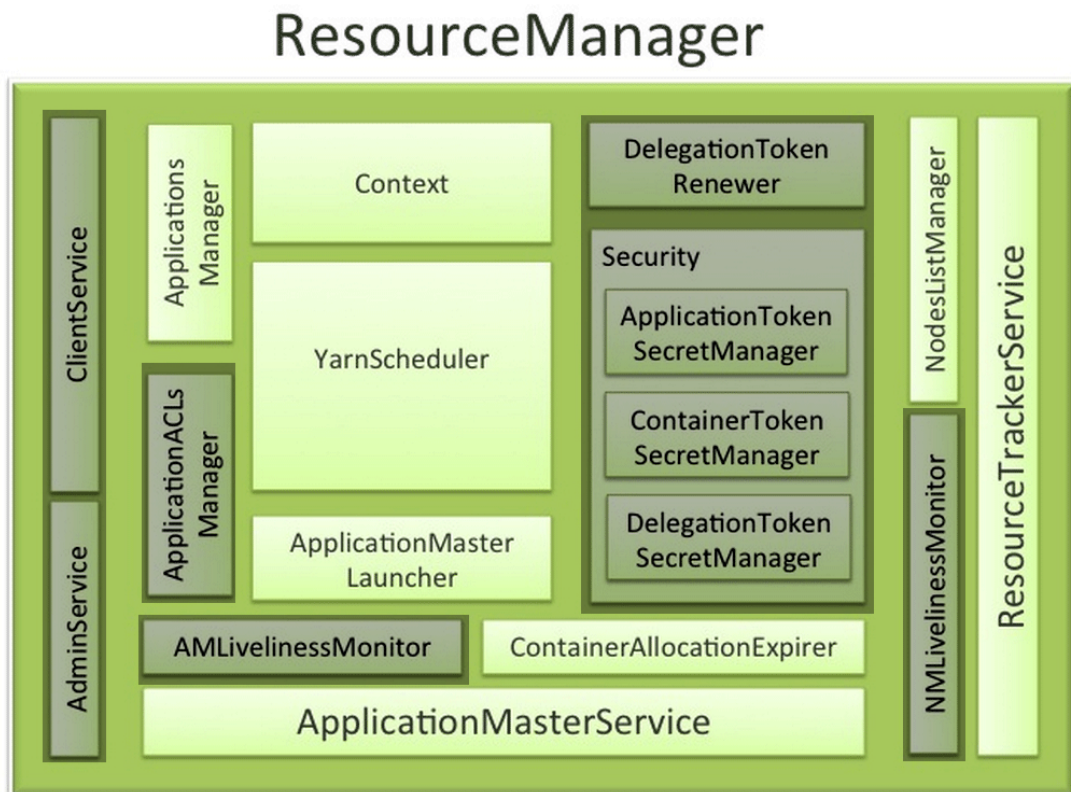


Figure 3.1 – ResourceManager Component Diagram (HortonWorks, 2014a)

The ResourceManager is the master that arbitrates all the available cluster resources and thus helps manage the distributed applications running on the YARN system. It works together

with the per-node NodeManagers (NMs) and the per-application ApplicationMasters (AMs) (HortonWorks, 2014a).

Except the scheduler itself, there are other components of fundamental importance to the scheduling process. Like the ApplicationsManager, which manages all the submitted applications. The ApplicationsManager component not only has a list of submitted applications, but also has information about all the completed applications and is able to provide this through either web UI or command line.

Another important component is the ApplicationMasterLauncher, that will be responsible to create the application specific ApplicationMaster, everytime an application is submitted. Another task left to the ApplicationMasterLauncher component is the deletion of ApplicationMaster when the application has finished, either normally or forcefully.

The ApplicationMaster itself is a concept that makes YARN completely different than the previous Hadoop versions. It is a key component on MapReduce tasks execution because it has one instance for every application being executed, making the ApplicationMaster the component responsible for the execution and monitoring of the containers and their resource consumption. Therefore, it has to communicate with the ResourceManager in order to ask and report the status and progress of its monitored containers.

It is through ApplicationMaster that YARN can achieve better scalability, since it takes some of the processing usually delegated to the Scheduler and ResourceManager components. One of the key tasks that was transferred to the ApplicationMaster is the fault tolerance. It is the ApplicationMaster that will decide when and/or if a speculative task is necessary and beneficial. Thanks to this shift in the responsibilities and the ApplicationMaster being a per-application manager, the cluster scaling potential is improved through the removal of a possible bottleneck.

Another crucial component to the present work is the ResourceTrackerService, that is responsible for the communication with the NodeManagers. This is the component that answers to the RPC calls, whenever a node wants to register with the RM, send a heartbeat, or many other tasks, this is the component that will be used. The node capacity is also passed on the registration of a node with the RM, this process of registration will store the node's information in the NodesListManager. The second component is a collection of all the nodes, both the valid and the decommissioned ones.

Finally, the Scheduler follows the same pattern as regular scheduler, comparing requests,

availabilities and then granting the resources based on some heuristic. However, most Hadoop schedulers use queues in order to help the task of managing the users and application submitted.

The other high level view that interests this work is the NodeManager. Just as the ResourceManager counterpart, even having some irrelevant modules presented in the figure 3.2, it facilitates the understanding of the service in a high level. Having a high level understanding is necessary in order to understand how the two services will interact.

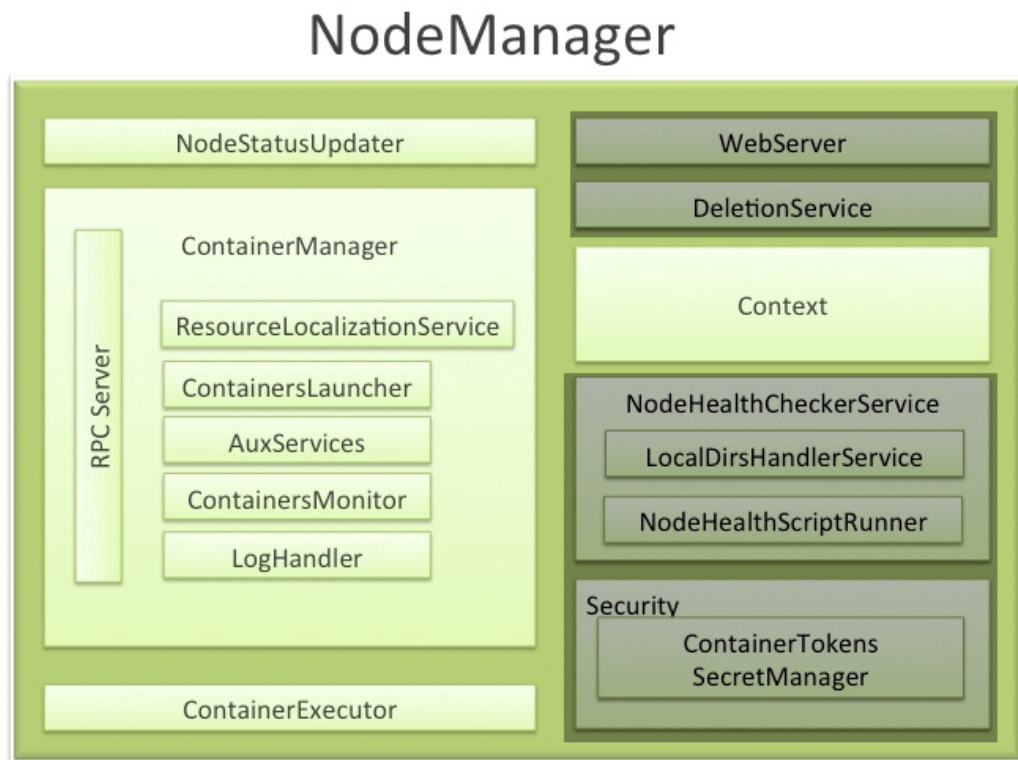


Figure 3.2 – NodeManager Component Diagram (HortonWorks, 2014b)

Each slave will have a NodeManager instance running, which is the local manager. This service main responsibility is keeping his information updated on the ResourceManager, but it has other attributions like managing the containers, monitoring the resource usage, monitoring node health, among others.

One key component is the NodeStatusUpdater, which will be responsible for the registration with the ResourceManager, it is through registration that the ResourceManager will be informed about the resources of a given node, making this component vital for the success of the approach in this work. Also, after the initial registration this component is expected to maintain the communication with the ResourceManager in order to provide updates on containers

launched, running or completed.

The largest component in the figure, the ContainerManager is as important as its size implies. With help from its sub-components, the ContainerManager has the hard but extremely necessary task of monitoring containers and informing whatever component needs this information. There are several ContainerManager sub-components, they are: RPC server, Resource-LocalizationService, ContainersLauncher, AuxServices, ContainersMonitor and LogHandler. Some like the ResourceLocalizationService is of vital importance to the MapReduce tasks, as it downloads the files that will be used on the tasks' execution, but won't interfere in this work's context.

The next diagram used to better understand the architecture was the scheduler components class diagram, which provided a helped to view classes related to the scheduling process. The Scheduler Components Class Diagram can be visualized in the figure 3.3.

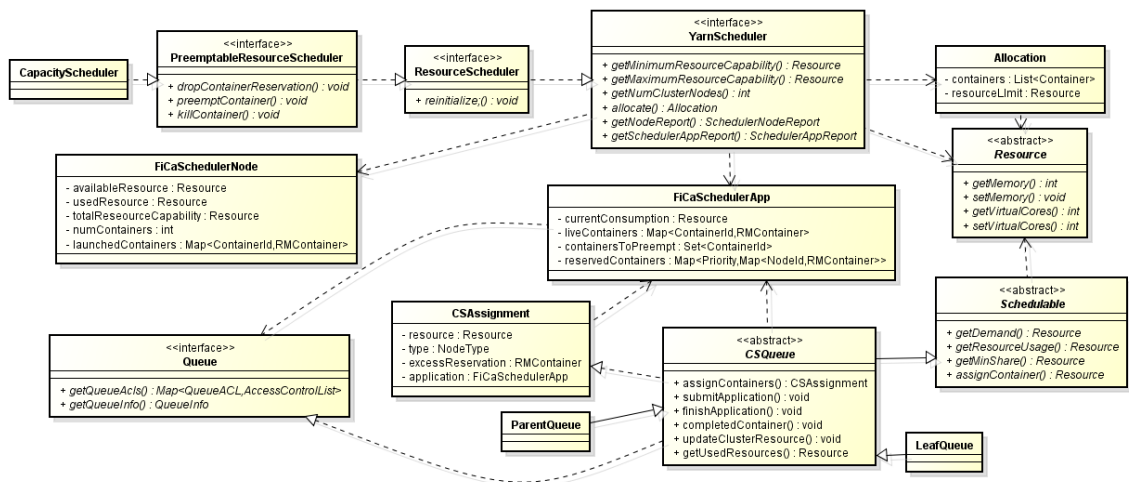


Figure 3.3 – Class Diagram with the main CapacityScheduler classes

Following is a description of the main components that compose the CapacityScheduler:

- **Schedulable:** an abstract class that represents an entity capable of launching either a job or a queue, it offers a simple interface whereby the algorithms can be applied either inside a queue as well as several queues.
- **Queue:** an interface that enables the basic control of all the queues in the scheduler. Possessing methods to allow reading of the queue info, and also queue management.
- **PreemptibleResourceScheduler:** another interface that allows the preemption of resources, through the scheduler.

- Resource: an abstract class responsible for modelling the resources capable of being use, which at this moment are only memory and core count.
- FiCaSchedulerNode and FiCaSchedulerApp: representations of the node and application, provides vital information about the structures. Uses a lot of the components present on the next Class Diagram.

Another key class diagram to the context of this work was the diagram that would represent three very important components, the RMContainer, RMNode and RMAApp/RMAAppAttempt. All of these components represents fundamental parts on understanding how this work has impacted the scheduling. RMContainer is the name given to the reservations of resources, making it also responsible for the task execution. RMNode is the representation of a whole NodeManager to the scheduler, and is through it that the scheduler will get access to the NodeManager real resource capacity. RMAApp and RMAAppAttempt represents the applications sent to the scheduler. This diagram is represented in the figure 3.4.

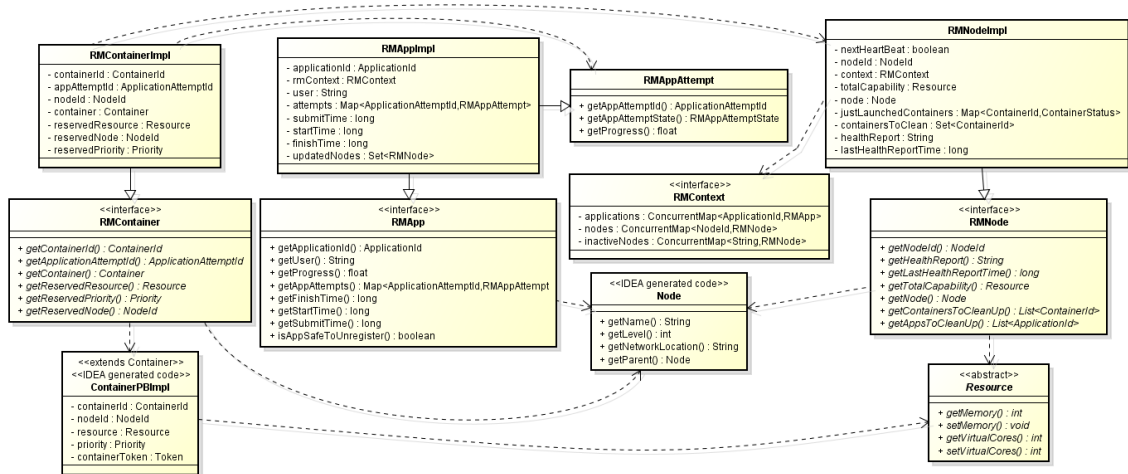


Figure 3.4 – Class Diagram with the main RM components

3.1.2 State machines

The second method was executed through an option offered by the Maven tool, in which it is possible to generate graphs that represents the Resource Manager and Node Manager state machines. Through analysis of the graphs it is possible to identify the life cycle of some key components, such as RMContainers, RMNodes and RMAApplications.

The importance of RMContainers, RMNodes and RMAApplications may be overlooked until further analysis of source code and state machines at the same time. It is not a coinci-

dence that all the components have a RM, referencing ResourceManager, in their names. In a brief explanation, RMNodes have all resource and other static information concerning a given NodeManager. RMContainers are the structures that represent the reservation of resources and also responsible for the processing. Finally, the RMAApplication is the component that provides ResourceManager a way to access application status, reports and updates. All state machines and a more thorough explanation can be found on Appendix A.

3.2 Understanding Hadoop allocation pattern

In order to improve Hadoop resource utilization and change the resource allocation behavior, it was necessary to understand how the allocation is made. That implies knowing and understanding the whole process of requisition and grant of resources, which is essentially the scheduling. After the experiments made with the objective of understanding this process, it was possible to identify which parameters influenced and how much impact would these parameters have once they were changed.

Once a quick research on official documentation and the parameters which can be set on XML files was made, it was found that there are five parameters that influence the allocation pattern for each resource. These parameters can be roughly divided into three classes: application request, cluster configuration towards containers and cluster configuration towards ApplicationMaster (AM). All of these parameters have default values in case the application did not set a request value or the cluster administrator did not set them properly on configuration files.

The application request parameters are set through the JobConf class when the user is coding his MapReduce job. If the parameters are not set at this time, the cluster will use the properties *mapreduce.map.memory.mb* and *mapreduce.reduce.memory.mb* either at the *mapred-site.xml* or *mapred-default.xml*, following default Hadoop behavior towards settings in the xml files. The default behavior is: if the properties are set in any **-site.xml* file that's the value they will assume, otherwise the values will be taken from **-default.xml* file. The default value for properties *mapreduce.map.memory.mb* and *mapreduce.reduce.memory.mb* is 1024.

The cluster configuration towards containers is composed by four parameters, these parameters express the floor and ceiling of valid allocations. The floor limit parameters receive their values from the properties *yarn.scheduler.minimum-allocation-mb* related to the memory and *yarn.scheduler.minimum-allocation-vcores* related to the cores, while the ceiling

limit parameters receive their values from properties *yarn.scheduler.maximum-allocation-mb* and *yarn.scheduler.maximum-allocation-vcores*. All properties will follow the default Hadoop behavior towards settings in XML files. These four parameters are related to two variables in the source code, which are named *minimumAllocation* and *maximumAllocation*, which represents the floor and ceiling limit respectively. The default value for *yarn.scheduler.minimum-allocation-mb* is 1024 and the default value for *yarn.scheduler.minimum-allocation-vcores* is 1. The default values concerning the ceiling properties is 8192 for *yarn.scheduler.maximum-allocation-mb* and 32 for *yarn.scheduler.maximum-allocation-vcores*.

The only parameter left is the AM request, this request will be the same for every application submitted to the cluster and cannot be configured through JobConf. This parameter will receive its value from properties *yarn.app.mapreduce.am.resource.mb* related to the memory and *yarn.app.mapreduce.am.resource.cpu-vcores* related to the cores, also following the default Hadoop behavior. The default value of the parameter *yarn.app.mapreduce.am.resource.mb* is 1536, and, the default value of the parameter *yarn.app.mapreduce.am.resource.cpu-vcores* is 1.

3.2.1 Experiment

In order to understand how the allocation process is made an experiment was performed. The experiment was made aiming to understand how the RM allocates memory for applications given requisition and minimum/maximum parameters changes. It consisted in altering some of the parameters and analysing the resultant behavior. As the AM request is the same across the cluster, it was excluded from the experiment. The reason being that different applications would have the same amount requested by the AM and its behavior follows the same pattern as the other requests, which are easier to manipulate.

3.2.1.1 Employed methods, procedures and scenarios

The experiment was performed in a cluster deployed on Grid'5000 environment. The cluster had five nodes, one master and four slaves, each node having the following configuration: 2 CPUs AMD@1.7GHz, 12 cores/CPU and 47GB RAM. All nodes were running an Ubuntu-x64-1204 standard image, having Sun JDK 1.7 installed. The Hadoop distribution was the 2.2.0 YARN version.

The procedure chosen as data acquisition method was the Hadoop Log System. The reason for such a choice was that Hadoop Log System is, by default, enabled in the INFO level

Parameters	Default	Higher	Smaller	Inside
Map Memory Requisition (MB)	1024	1024	1024	3456
Reduce Memory Requisition (MB)	1024	1024	1024	3712
Minimum Memory (MB)	1024	512	2048	512
Maximum Memory (MB)	8192	768	8192	8192
Map Memory Allocation (MB)	1024	ERROR	2048	3584
Reduce Memory Allocation (MB)	1024	ERROR	2048	4096

Tabela 3.1 – RM memory allocation pattern experiment results

and using the INFO level would be possible to insert small entries and extract useful information in real time.

Following there is a brief description of the four scenarios used in the experiment:

- **Default scenario:** no parameter was changed.
- **Requisition higher than maximum:** the application requested an amount of memory higher than the cluster maximum allocation parameter. The changed value was the maximum allocation memory. The minimum was also changed for consistency reasons.
- **Requisition smaller than minimum:** the application requested an amount of memory smaller than the cluster minimum allocation parameter. The changed value was the minimum allocation memory.
- **Requisition inside range:** the application requested an amount of memory inside the cluster valid range. The changed values were the minimum allocation memory and requisition from both map and reduce.

3.2.1.2 Results and interpretation

The results from the scenarios can be analysed on the figure ??.

Parameters	Default	Higher	Smaller	Inside
Map Memory Requisition (mb)	1024	1024	1024	3456
Reduce Memory Requisition (mb)	1024	1024	1024	3712
Minimum memory (mb)	1024	512	2048	512
Maximum memory (mb)	8192	768	8192	8192
Map Memory Allocation (mb)	1024	DNE	2048	3584
Reduce Memory Allocation (mb)	1024	DNE	2048	4096

Because of the experiment, it was possible to detect that Hadoop allocation pattern differs a bit from usual. Unlike usual pattern in which if a request is inside the minimum and maximum range, the amount granted is equal to the request, the requests on Hadoop will pass through a small set of calculations in order to determine how much memory will be granted.

The figure 3.5 portrays the flow of operations executed by the Hadoop in order to determine the granted resources.

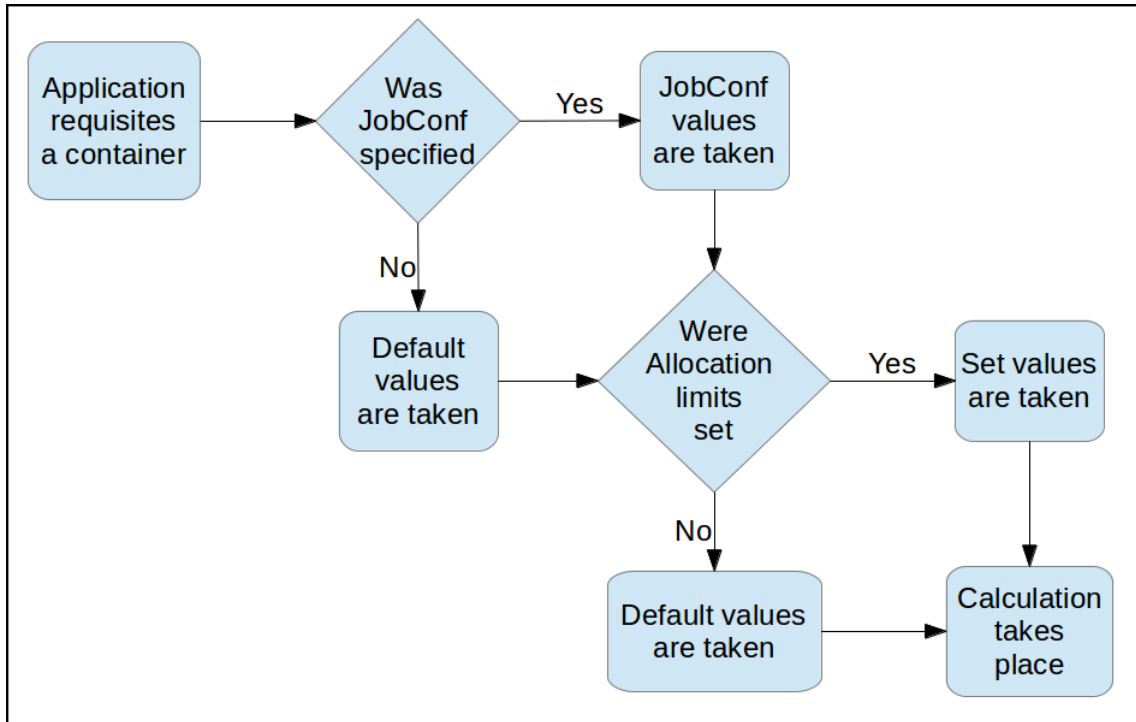


Figure 3.5 – Flow of operations for resource granting

The default scenario just demonstrates how Hadoop allocation will behave in case there are no interventions.

The second scenario shows what happens if the application requests a value higher than the maximum. The output is an error message and the job execution is aborted.

The third scenario shows what happens if the application requests a value smaller than the minimum. The cluster grants the minimum allowed.

The fourth scenario shows a case in which the requests are inside the valid range but although the requests were similar, the resources granted were different. This scenario is the one that makes it possible to fully comprehend the allocation process. Since the second and third scenarios provided evidence that a request can't be higher than the maximum and that at least the minimum allocation will be given, it is possible to infer that the allocation will always start with the minimum allocation value. In the case the minimum allocation is not enough

to satisfy the request, the value which will be granted is always incremented by the minimum allocation until it matches one of the following cases: the value is equal to the request, the value is higher than the request and lower than the maximum allocation or the value exceeds maximum allocation. Concerning the scenario, it is the second case that occurs.

The figure 3.6 is provided along with extensive explanation to facilitate the understanding of the whole calculation process.

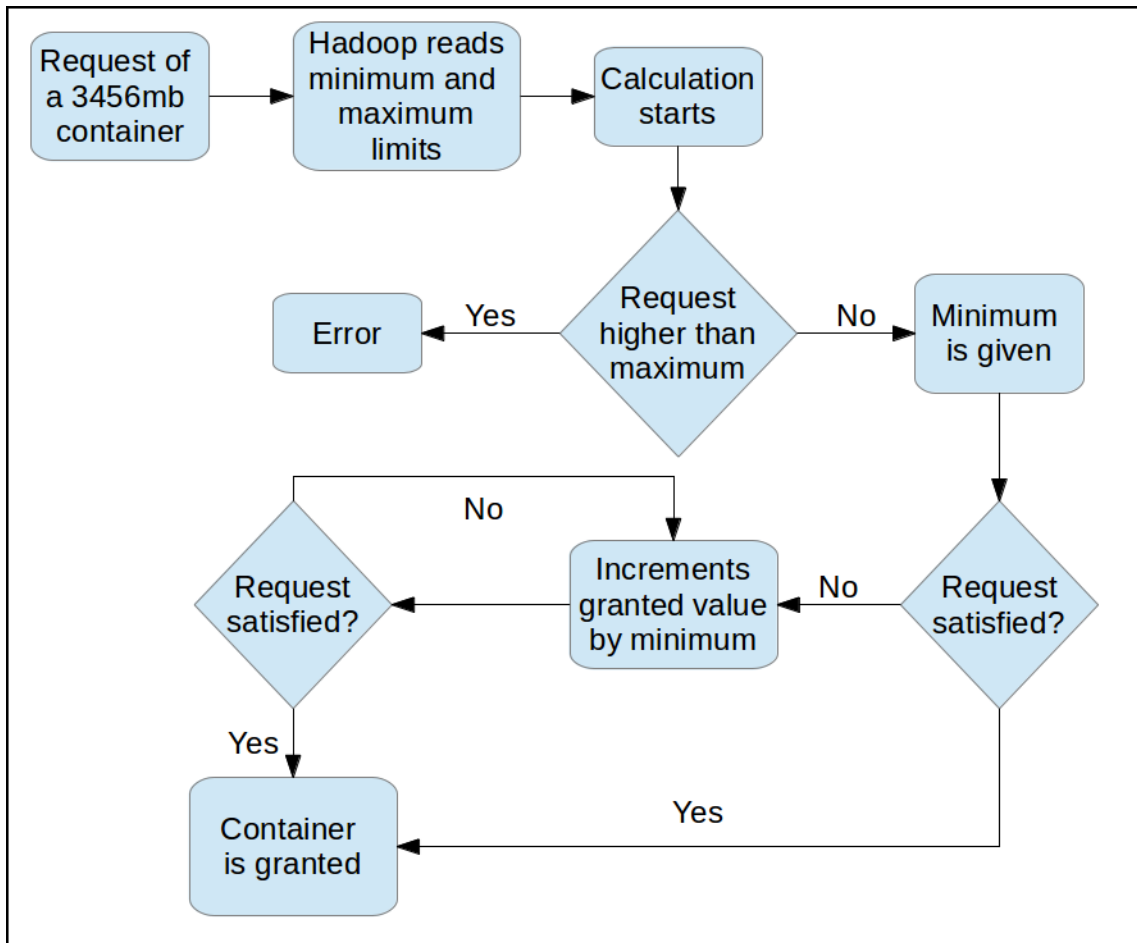


Figure 3.6 – Flow of operations for resource granting

The step by step calculation on scenario four will be demonstrated. Assuming M is the memory granted, or in the process of calculation, for the map request and R for the reduce. The calculation happens as follows:

Firstly the M and R are set to the minimum memory allocation property, which is 512. Since 512 does not satisfy any of the requests the M and R values are then incremented by the minimum memory allocation, assuming the value of 1024. As 1024 still smaller than both requisitions, they are again incremented by 512 and the process goes on as following:

M : 512, 1024, 1536, 2048, 2560, 3072, 3584.

R: 512, 1024, 1536, 2048, 2560, 3072, 3584, 4096.

Note that in both cases the value granted was the smallest multiple of minimum memory allocation (512) greater than or equal to the request (3456 and 3712).

3.3 Context-aware scheduling

Knowing that the scheduling task is closely related to the resources availability and, therefore, having a wrong information could ruin the performance of the algorithm, an opportunity for improvement was seen. After a quick study on the default schedulers, it was clear that CapacityScheduler already had the whole structure to support a better scheduling, as stated in the section 2.3. The flaw on CapacityScheduler wasn't really a scheduling flaw, but actually a wrong information received by the NodeManagers.

As stated numerous times in previous sections, Hadoop configuration is heavily dependent in XML files. While providing an easy way to configure a real cluster, sometimes it acts more as an obstacle than as a facilitator, the reason being that if one wants to change a parameter the whole service will have to be restarted. One huge restriction implied by XML files, is regarding the nodes capacity. In case one doesn't want to use Hadoop default configurations for node capacity, every node will have to have it's XML files edited. In a large heterogeneous cluster, modifying one file in each node will certainly be time consuming since each node will have a different configuration.

One possible solution to this case, is to overwrite the value gotten from XML file on the code. At first glance this brings in more problems than solutions, because the administrator would have to chose a certain hard coded value that would fit best as an average among all nodes. However, as one looks closer into the proposal, there is an option that, although involving more coding, would solve this problem.

It is clear that this solution requires the application to detect some context of the environment. The context, in this case, being the real information about node capacity. With this context at hands, it is a reasonable choice to make the scheduler context-aware. Therefore, improving the scheduling performance. As the section 2.2 implies, being context-aware requires the application to detect and adapt to changes in environment.

Regarding the detection of the node capacity, the chosen option was to integrate a collector into Hadoop, meaning that every node would be able to access it's true capacity. Thus, preventing the hard coding that was suggested.

Regarding the changes made once the context information was detected, the approach adopted was to scale the allocation limits together with the total cluster resource availability. This scaling meant that the containers would have more memory and cores at their disposal and, therefore, speculative task would hardly have to be launched. Also, by adapting to the cluster real resource, no resource would be wasted or left inactive while the scheduler is making tasks wait due to wrong information being received. Thus improving the cluster utilization.

3.3.1 Collector description

The collector of choice was PER-MARE's collector (Kirsch-Pinheiro, M., 2013). This collector uses a standard java package in order to access the real node capacity, therefore, no additional libraries would be necessary.

Only four files were necessary, an interface, an abstract class and two classes. Due to it's design, it would be easy to integrate new collectors and improve the resources available for the scheduling process, providing data about the CPU load or disk usage for example.

The base of the collector is the interface `Collector`, which has two access methods and the `collect` method.

The abstract class, called `AbstractOSCollector`, implements the interface and has some access methods of its own. The great usability comes from the usage of `OperatingSystemMXBean`, a public java interface that is used to access the operating system informations on which the Java virtual machine is running (Oracle, 2014).

The collector classes, called `PhysicalMemoryCollector` and `TotalProcessorsCollector` extends the `AbstractOSCollector` and have only constructor and collector method implemented.

3.3.2 Collector integration with Hadoop

The collector would have to be placed on `NodeManager` (NM), since this is the service responsible for processing tasks. The collected capacity is then sent to the `ResourceManager` (RM) in the moment that each NM is registering with the RM. It is possible to view the whole process of data acquisition until `CapacityScheduler` add the node in the figure 3.7.

After the collector integration, changing the scheduling behavior was possible due to the now available information about the real resources of a given node. Further information about the results gotten from the collector integration and modified scheduling can be found at the chapter 4.

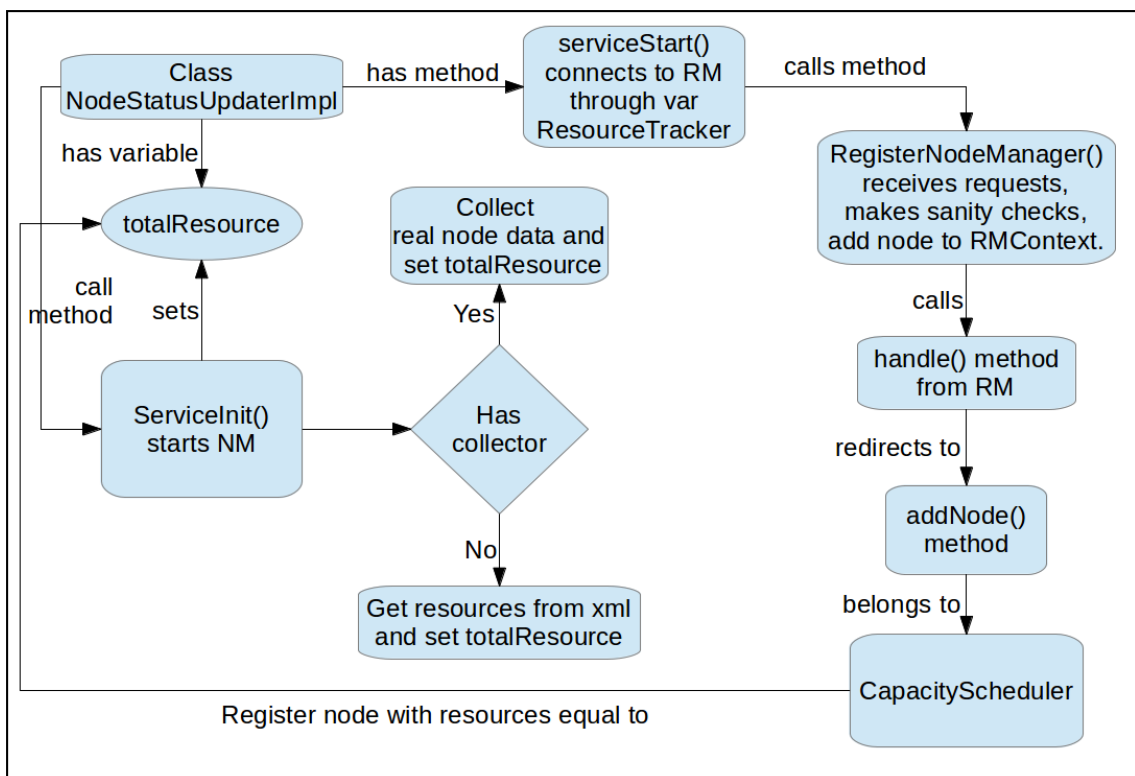


Figure 3.7 – Flow of operations for resource granting

4 EXPERIMENTS AND RESULTS

This chapter will provide information about the experiments that were capable of changing the Hadoop scheduling behavior and the results achieved.

4.1 Collector integration test

This experiment had the objective of testing the collector integration. The experiment consisted in deploying and starting Hadoop services in the cluster with original CapacityScheduler and with the context-aware CapacityScheduler in order to compare the change in total resources availability.

4.1.1 Hardware and Software configuration

The experiment was performed in a cluster deployed on Grid'5000 environment. The cluster had five nodes, one master and four slaves, each node having the following configuration: 2 CPUs AMD@1.7GHz, 12 cores/CPU and 47GB RAM. All nodes were running an Ubuntu-x64-1204 standard image, having Sun JDK 1.7 installed. The Hadoop distribution was the 2.2.0 YARN version.

The default Hadoop configuration is set on *yarn-default.XML* under the properties named *yarn.nodemanager.resource.memory-mb* and *yarn.nodemanager.resource.CPU-vcores* which have default values of 8192 and 8 respectively. The only difference being that one of the executions had the collector plugged.

4.1.2 Procedures

The procedure chosen as data acquisition method was the Hadoop Log System. The reason for such a choice was that Hadoop Log System is, by default, enabled in the INFO level and using the INFO level would be possible to insert small entries and extract useful information in real time. The data was acquired with the same call during the execution of services with both schedulers.

	Original CapacityScheduler	Context-aware CapacityScheduler
Node Memory	8192	48303
Node Vcores	8	24

Tabela 4.1 – Resources available on original and context-aware CapacityScheduler

4.1.3 Results and interpretation

The comparison of node memory from default and collector implementation can be seen in the table 4.1.

There are also two more figures that express how better the collector implementation scales given proper hardware. These figures are figure 4.1 and figure 4.2.

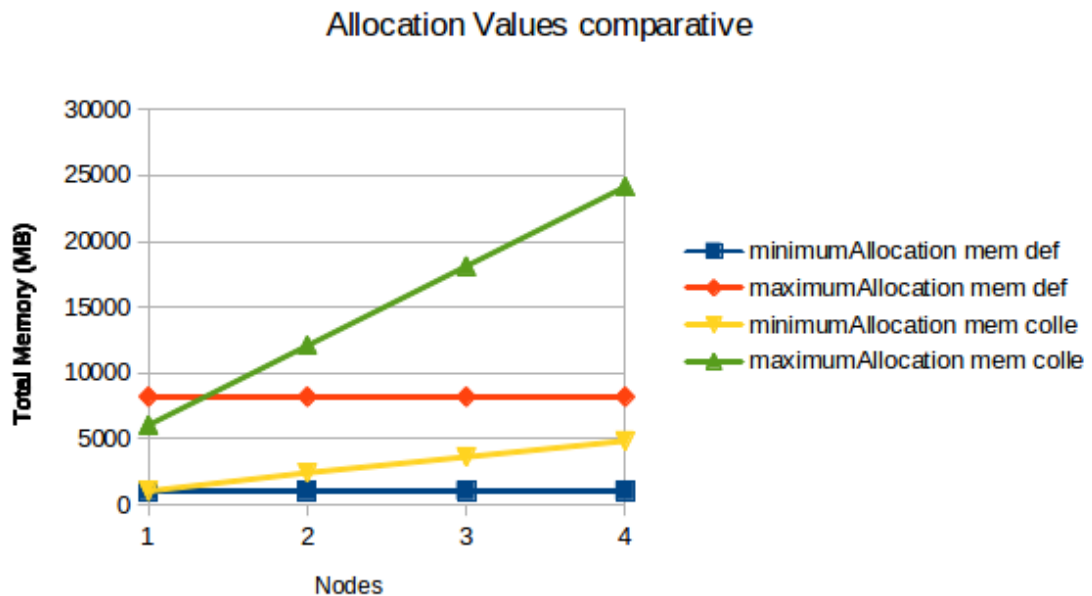


Figure 4.1 – Cluster memory available on default and collector implementation

Since this experiment was run on Grid'5000 and all nodes have the same configuration, it would be easy to discover the true capacity of a node, change the values on a XML file and replicate it to all other nodes inside the environment. The problem becomes huge once the environment is not homogeneous. It would require someone to discover the true capacity of each node and then separately edit the XML files in every node.

Consider that according to HadoopWizard (HadoopWizard, 2014) by July 2011 Yahoo! used a 42000 nodes Hadoop Cluster, and on the same month Facebook publicized it runs a 2000 nodes Hadoop cluster. Changing every node configuration manually would be quite challenging, therefore the collectors used would come in hand.

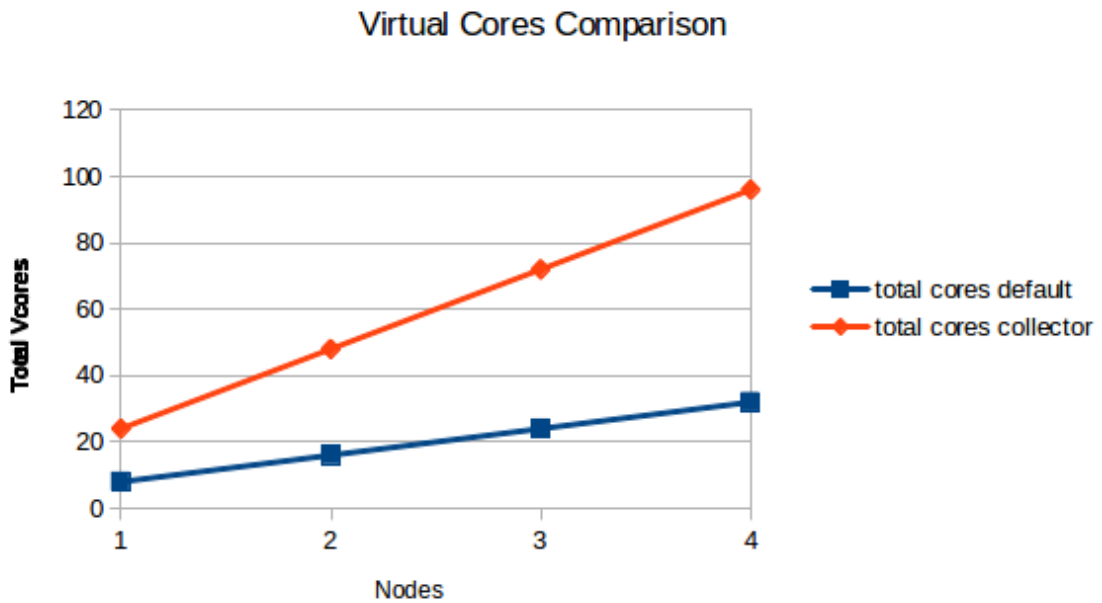


Figure 4.2 – Cluster cores available on default and collector implementation

4.2 Original CapacityScheduler X Context-aware CapacityScheduler

This experiment was performed in order to compare the container allocation pattern in the original CapacityScheduler and the context-aware CapacityScheduler. The experiment consisted in executing a TeraSort in the cluster with the original CapacityScheduler and the context-aware CapacityScheduler. This was made in order to compare how the higher resource availability and higher allocation limits impacted the scheduling.

4.2.1 Hardware and Software configuration

The experiment used the same hardware configuration from the previous one. Regarding the Hadoop configuration, there are new properties used. The properties are the minimum and maximum allocation values, which are set in properties stated on section 3.2. The only difference being that one of the executions had the collector plugged.

4.2.2 Procedures

The procedure chosen as data acquisition method was the Hadoop Log System. The reason for such a choice was that Hadoop Log System is, by default, enabled in the INFO level and using the INFO level would be possible to insert small entries and extract useful information in real time. The data was acquired with the same call during the execution of services with

both schedulers.

The application used to test the scheduling was a TeraSort with 5GB data to sort, requesting enough containers and providing enough data to be processed in order to stress the cluster.

4.2.3 Results and interpretation

Before going further into the interpretation of the results, there are some characteristics of jobs that need to be reminded. If the number of reduce tasks parameter isn't set on *mapred-site.XML*, the default value used is 1, making the whole reduce step forced to be executed on only one container.

Another thing to note is that the first allocated container is always the Application-Master, making this container not relevant in grant of resources for MapReduce tasks analysis. Thus both the ApplicationMaster and Reducer container were withdraw from the data analyzed, which was left only with the Map containers. All times are normalized related to the first Map container created.

The cluster configuration achieved with the original CapacityScheduler was:

- Total cluster resource of 32768mb and 32cores
- Minimum allocation of 1024mb and 1 core
- Maximum allocation of 8192mb and 32 cores.
- All Map containers were granted containers of 1024mb and 1 core, the minimum limit.

The cluster configuration achieved with the context-aware CapacityScheduler was:

- Total cluster resource of 193210mb and 96cores
- Minimum allocation of 4830mb and 2 cores
- Maximum allocation of 24151mb and 12 cores
- All Map containers were granted containers of 4830mb and 2 cores, the minimum limit.

Although a huge difference was achieved by only comparing the resources collected and the allocation limits, the main objective of this work is to impact the scheduling performance

in a Hadoop cluster. Therefore, a TeraSort execution was made and the results achieved are discussed below.

The following Gantt Charts are consolidated by resources, which are the NodeManagers. This means that the tasks, in this case portrayed as containers, will be consolidated to the resources they are tied. As stated before, the containers are allocated to a certain NodeManager. The consolidation works in a way that when a separation occurs in the segment, it means that a container has either started or finished on that NodeManager. That implies that many containers will be on more than one segment, and, the numbers inside the segment indicates which containers are running at that moment.

Figure 4.3 portrays the Gantt Chart of the TeraSort with original CapacityScheduler. It is easy to notice that some containers had to wait for the completion of others in order to start processing their tasks.

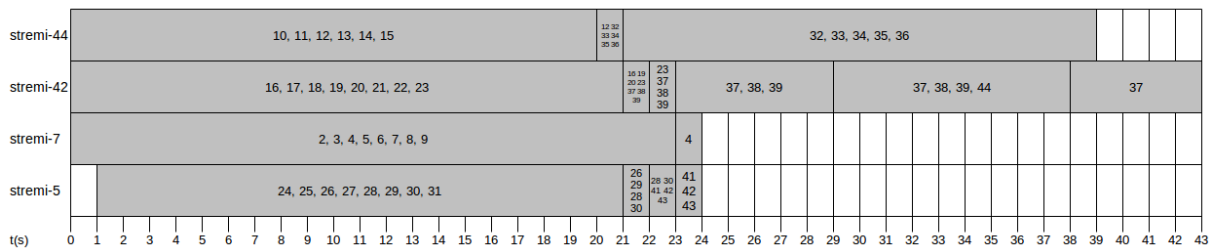


Figure 4.3 – Container assignment with default configuration

In order to illustrate how to interpret the Gantt charts, the node stre-mi-42 of figure 4.3 will be taken as example. It starts all its containers, numbered from 16 to 23, at the 0 seconds mark, then the segment ends at the 21 seconds mark, meaning that either a container started or finished. After a quick analysis of the containers in the first and second segments, it is possible to note that containers 17, 18, 21 and 22 are not in the second segment, meaning they have finished processing their tasks. Another thing to notice is that on the second segment, containers with numbers 37, 38 and 39 appeared for the first time, meaning they were started at this time. If the analysis is extended to the segment from 22 to 23 seconds, it is possible to note that containers 16, 19 and 20 have finished processing their tasks too, and the only running containers in this node at this moment are the containers 23, 37, 38 and 39.

Figure 4.4 portrays the Gantt Chart of the TeraSort with context-aware CapacityScheduler. In this case the overall completion time was reduced, this happened due to the fact that all containers could be started right after the arrival of the request, thanks to the higher resource availability.

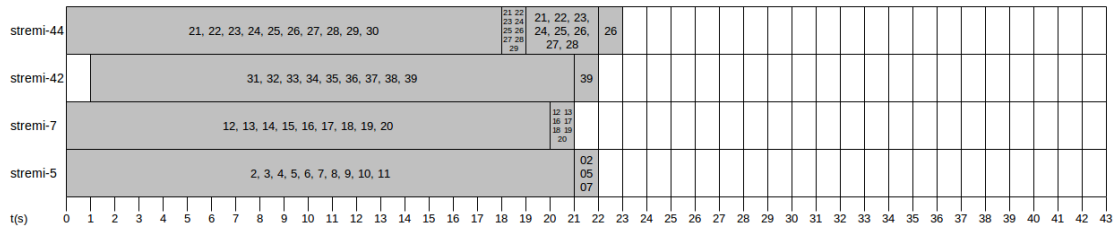


Figure 4.4 – Container assignment with the improved configuration

After an analysis and comparison of both charts, it is possible to notice that the default chart has containers 41-43 started on node stremi-5 and container 44 started on node stremi-42, while the context-aware chart has only the standard containers, which are numbered 2-39. This happens because these extra containers are, in reality, speculative tasks launched because other tasks were taking too long to finish. Without a better information acquisition it is hard to determine the match of original and speculative containers, but it is possible to infer which containers would make possible candidates, leaving containers 2-9, 23, 28 and 30 as possible staggers, responsible for the launching of speculative containers 41-43. Because of the same reasons, it is only possible to infer that container 44 was launched because the most likely stagger was one container in the 32-36 range.

By analysing the container numberings it is possible to notice how the scheduler decides which node is going to be used. The containers launched on a given node follow a logic numbering, meaning that the resources of that container are used until exhaustion before the scheduler starts launching containers on another node.

4.3 Heterogeneity simulation

This experiment was performed in order to simulate a heterogeneous environment and test how well would the context-aware scheduler adapt. The experiment consisted in executing a TeraSort in the cluster with the simulated heterogeneous environment using context-aware CapacityScheduler.

4.3.1 Hardware and Software configuration

The experiment used the same hardware configuration from the previous experiments. Regarding the Hadoop configuration, there are no changes. The only difference is that the nodes are purposely given false capacities when being added to the RM. Using these false values,

a heterogeneous cluster will be simulated.

4.3.2 Procedures

The procedure chosen as data acquisition method was the Hadoop Log System. The reason for such a choice was that Hadoop Log System is, by default, enabled in the INFO level and using the INFO level would be possible to insert small entries and extract useful information in real time. The data was acquired with the same call during the execution of services with both schedulers.

The application used to test the scheduling was a TeraSort with 5GB data to sort, requesting enough containers and providing enough data to be processed in order to stress the cluster.

4.3.3 Results and interpretation

As this experiment is a replication of the last one plus the simulated heterogeneity, the same principles applies regarding the container analysis.

It is important to firstly know the configuration of the simulated heterogeneity. The cluster had the following simulated configuration:

- stremi-17: 28981 MB of memory and 14 cores.
- stremi-22: 34715 MB of memory and 18 cores.
- stremi-33: 46287 MB of memory and 24 cores.
- stremi-35: 24151 MB of memory and 12 cores.
- Total Cluster Resources: 134134 MB of memory and 68 cores.
- Minimum Allocation: 3353 MB of memory and 1 core.

Figure 4.5 portrays the Gantt Chart of the TeraSort execution within the simulated heterogeneous environment, also using context-aware CapacityScheduler. Compared to the default case, the heterogeneous environment execution shows an improvement, but due to the lower cluster capacity, it is a slightly worse than the context-aware CapacityScheduler executing on a homogeneous environment.

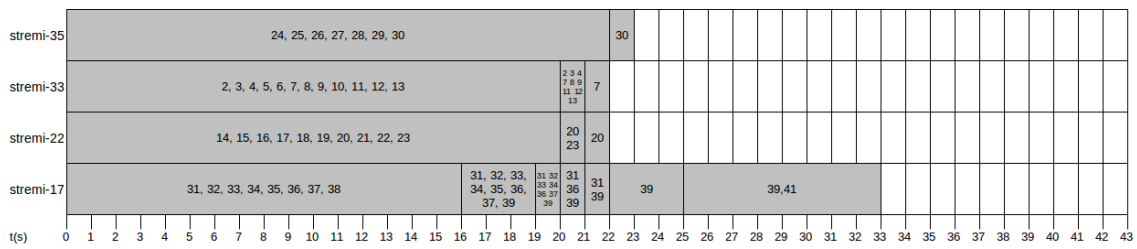


Figure 4.5 – Container assignment with the simulated heterogeneous environment

It is possible to note that the containers started the assignment with the node stremit-33, which is the node with the most capacity in the cluster and also was the first to be added in the node list. As in the other experiments, the scheduler launches containers on a node until its resources are all reserved, then mode to the next node on the list.

On this experiment a speculative task was launched. Contrary to the other experiments, its easy to infer which was the original stagger task, since there was only one container active at the moment that the container 41 was launched. It is also possible to note that the scheduler didn't change nodes to launch the speculative, that happens because the node had spare capacity when the request for the speculative arrived.

This experiment shows that it is possible to use this context-aware in a heterogeneous environment, the allocations were adapted to a slightly smaller cluster if compared to the real environment. As a future work, it is possible to set the allocation limits in function not only of total cluster resources but also of each individual node resource capacity.

5 CONCLUSION AND FUTURE WORK

This work had the objective of improving Hadoop scheduling, during the study process it was identified that the CapacityScheduler had already the base for a context-aware scheduling. However, this scheduler lacked some fundamental components in order to be context-aware, such as NodeManager real resource information and allocation limits scaling with the total cluster capacity.

Through development of this work changes have been made on original source code, these changes allowed Hadoop to be more aware of the context of nodes composing the cluster. The scheduling algorithm remained the same, however key limitations caused by Hadoop's default configurations were noticed. A new distribution containing a context-aware CapacityScheduler was generated in order to solve these issues.

The context-aware CapacityScheduler is capable of receiving the real capacity from each NodeManager, thanks to the collector plugged on NodeManager. This provides the cluster a better scaling potential while also using every node's full capacity. Using the context-aware CapacityScheduler, the allocations can be made to the full potential of the cluster instead of waiting for more resources when the cluster actually had almost 40GB of free memory per node.

Although the context-aware CapacityScheduler has better scaling potential and solves some problems on containers management, all contributions made are purely static and there are more ways to impact and improve Hadoop scheduling. Given Hadoop high modularity, it is possible to improve scheduling changing many areas that range from ApplicationMaster and Queues to NodeManager HeartBeat behavior.

Following there are some suggestions of future work:

- Extending the Resource class so it can track more resources like CPU load.
- Improving CapacityScheduler scheduling, taking into account other resources information.
- Modification of ApplicationMaster behavior.
- Implementation of a scheduler capable of starting containers directly on a NodeManager, and not dependant on queues.

REFERENCES

- Apache. **The Apache Software Foundation**. <http://www.apache.org>, Accessed August 2013.
- Apache Hadoop. **Apache™ Hadoop®**. <http://hadoop.apache.org/>, Accessed August 2013.
- Apache Nutch. **Apache Nutch™**. <http://nutch.apache.org>, Acesso August 2013.
- BALDAUF, M.; DUSTDAR, S.; ROSENBERG, F. A survey on context-aware systems. **Int. J. Ad Hoc Ubiquitous Comput.**, Inderscience Publishers, Geneva, SWITZERLAND, v.2, n.4, p.263–277, June 2007.
- CHEN, Q. et al. SAMR: a self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION TECHNOLOGY, 2010., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2010. p.2736–2743. (CIT '10).
- DEY, A. K. Understanding and Using Context. **Personal Ubiquitous Comput.**, London, UK, UK, v.5, n.1, p.4–7, Jan. 2001.
- Fair Scheduler. **Hadoop MapReduce Next Generation - Fair Scheduler**. <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html>, Accessed August 2013.
- Google. **Google**. <http://www.google.com/intl/pt-BR/about/company/>, Accessed August 2013.
- HADOOP, A. **Arquitetura do HDFS**. <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html>, Accessed November 2013.
- HADOOP, A. **Arquitetura do YARN**. <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>, Accessed November 2013.

HadoopWizard. **Which Big Data Company has the World's Biggest Hadoop Cluster?** <http://www.hadoopwizard.com/which-big-data-company-has-the-worlds-biggest-hadoop-cluster/>,

Accessed January 2014.

HortonWorks. **HortonWorks Hadoop YARN - ResourceManager.** <http://hortonworks.com/blog/apache-hadoop-yarn-resourcemanager/>, Accessed January 2014.

Accessed January 2014.

HortonWorks. **HortonWorks Hadoop YARN - NodeManager.** <http://hortonworks.com/blog/apache-hadoop-yarn-nodemanager/>, Accessed January 2014.

ISARD, M. et al. Quincy: fair scheduling for distributed computing clusters. In: ACM SIGOPS 22ND SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, New York, NY, USA. **Proceedings...** ACM, 2009. p.261–276. (SOSP '09).

jetBrains. **IntelliJ IDEA.** <http://www.jetbrains.com/idea/>, Accessed August 2013.

Kirsch-Pinheiro, M. **CaptureDonnees.** <http://per-mare.googlecode.com/svn/trunk/permare-ctx/>, Accessed December 2013.

Kirsch-Pinheiro, M. and Steffemel, L. A. **Requirements for Context-aware MapReduce on Pervasive Grids.** [S.l.]: STIC-AmSud PER-MARE, Université Paris 1, Université de Reims, 2013.

KUMAR, K. A. et al. CASH: context aware scheduler for hadoop. In: INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTING, COMMUNICATIONS AND INFORMATICS, New York, NY, USA. **Proceedings...** ACM, 2012. p.52–61. (ICACCI '12).

MAAMAR, Z.; BENSLIMANE, D.; NARENDRA, N. C. What can context do for web services? **Commun. ACM**, New York, NY, USA, v.49, n.12, p.98–103, Dec. 2006.

Oracle. **Interface OperatingSystemMXBean.** <http://docs.oracle.com/javase/7/docs/api/java/lang/management/OperatingSystemMXBean.html>, Accessed January 2014.

RASOOLI, A.; DOWN, D. G. Coshh: a classification and optimization based scheduler for heterogeneous hadoop systems. In: SC COMPANION: HIGH PERFORMANCE COMPUTING, NETWORKING STORAGE AND ANALYSIS, 2012., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2012. p.1284–1291. (SCC '12).

STEFFENEL, L. A. et al. PER-MARE: adaptive deployment of mapreduce over pervasive grids. In: IN PROCEEDING OF: 8TH INTERNATIONAL CONFERENCE ON P2P, PARALLEL, GRID, CLOUD AND INTERNET COMPUTING. **Anais...** [S.l.: s.n.], 2013. (3PGCIC'13).

TIAN, C. et al. A Dynamic MapReduce Scheduler for Heterogeneous Workloads. In: EIGHTH INTERNATIONAL CONFERENCE ON GRID AND COOPERATIVE COMPUTING, 2009., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p.218–224. (GCC '09).

XIE, J. et al. Improving MapReduce performance through data placement in heterogeneous Hadoop clusters. In: PARALLEL AND DISTRIBUTED PROCESSING, WORKSHOPS AND PHD FORUM (IPDPSW). **Anais...** IEEE International Symposium, 2010.

Yahoo. **Yahoo**. <http://info.yahoo.com/>, Accessed August 2013.

ZAHARIA, M. et al. Improving MapReduce performance in heterogeneous environments. In: USENIX CONFERENCE ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 8., Berkeley, CA, USA. **Proceedings...** USENIX Association, 2008. p.29–42. (OSDI'08).

APPENDIX

APPENDIX A – Generated ResourceManager Graphs

Following there are a brief explanation and the figures that were generated through the second method employed on the section 3.1.

The pertinence of these figures is validated by the fact that they describe all possible ways a given interface can take. The perfect case flow would be started with the submission of a job to the RM. There are some pre requisites that need to be fulfilled for the start of the job. From this point on, these figures are relevant.

Firstly an AppAttempt is created. The AppAttempt is literally a started application, through which the RM will try to allocate necessary resources (Node and Containers). If the resources are successfully allocated, the real App will be created. Then an ApplicationMaster will be launched in order to manage each Application allocated RMContainers and to which RMNode they belong.

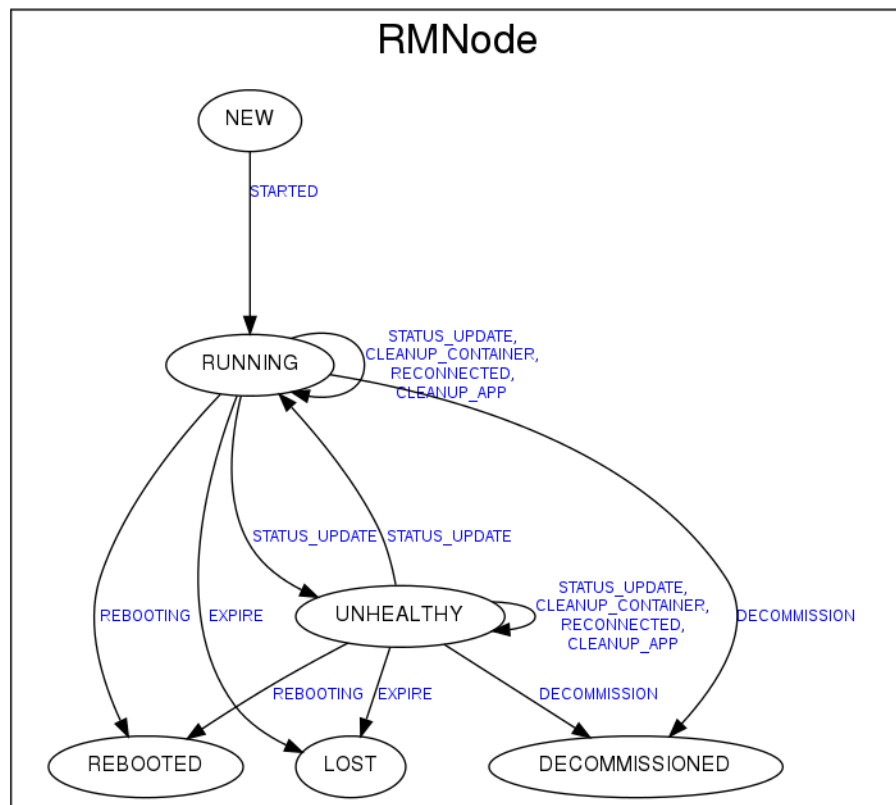


Figure A.1 – RMNode's state machine

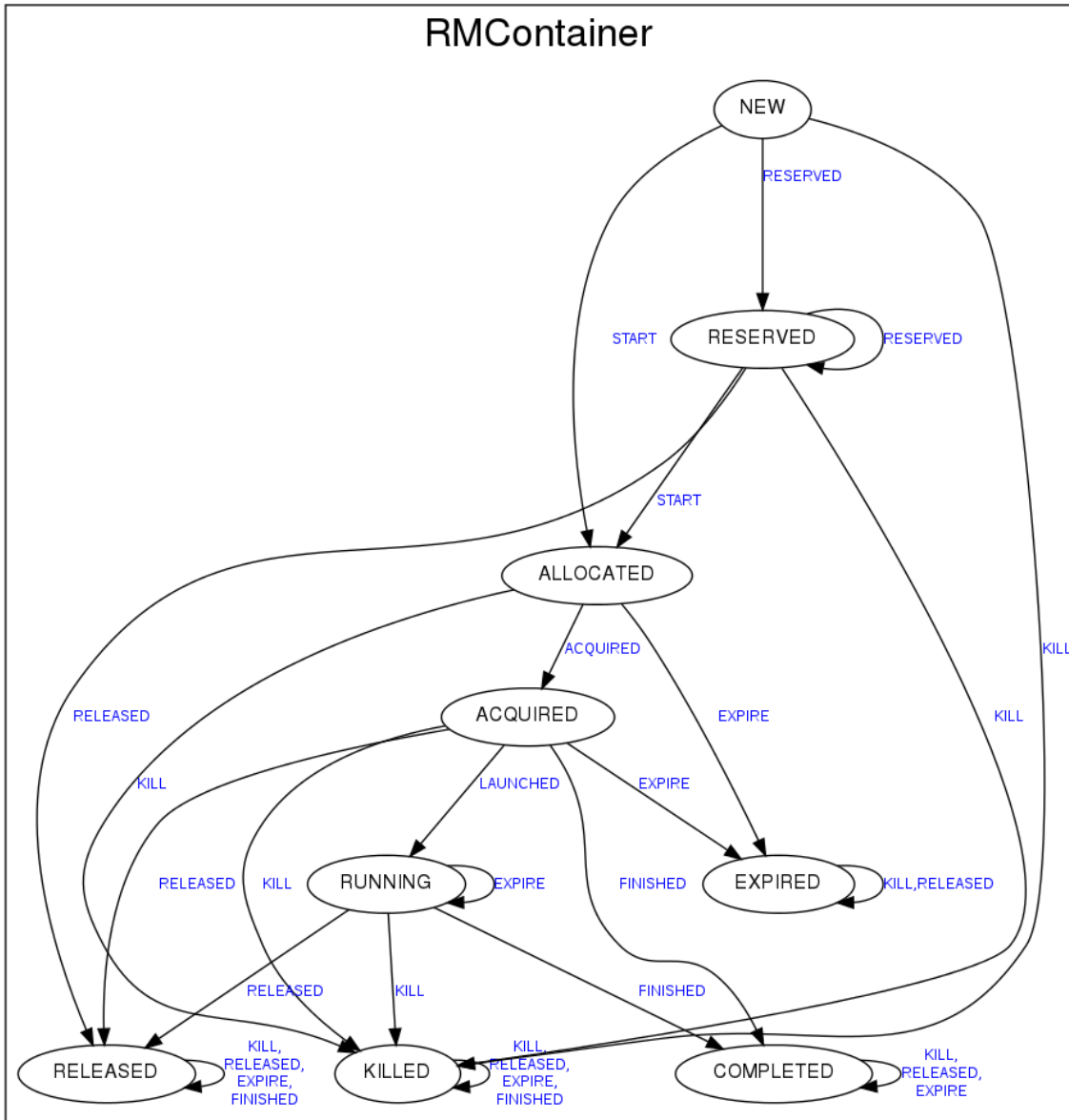


Figure A.2 – RMContainer’s state machine

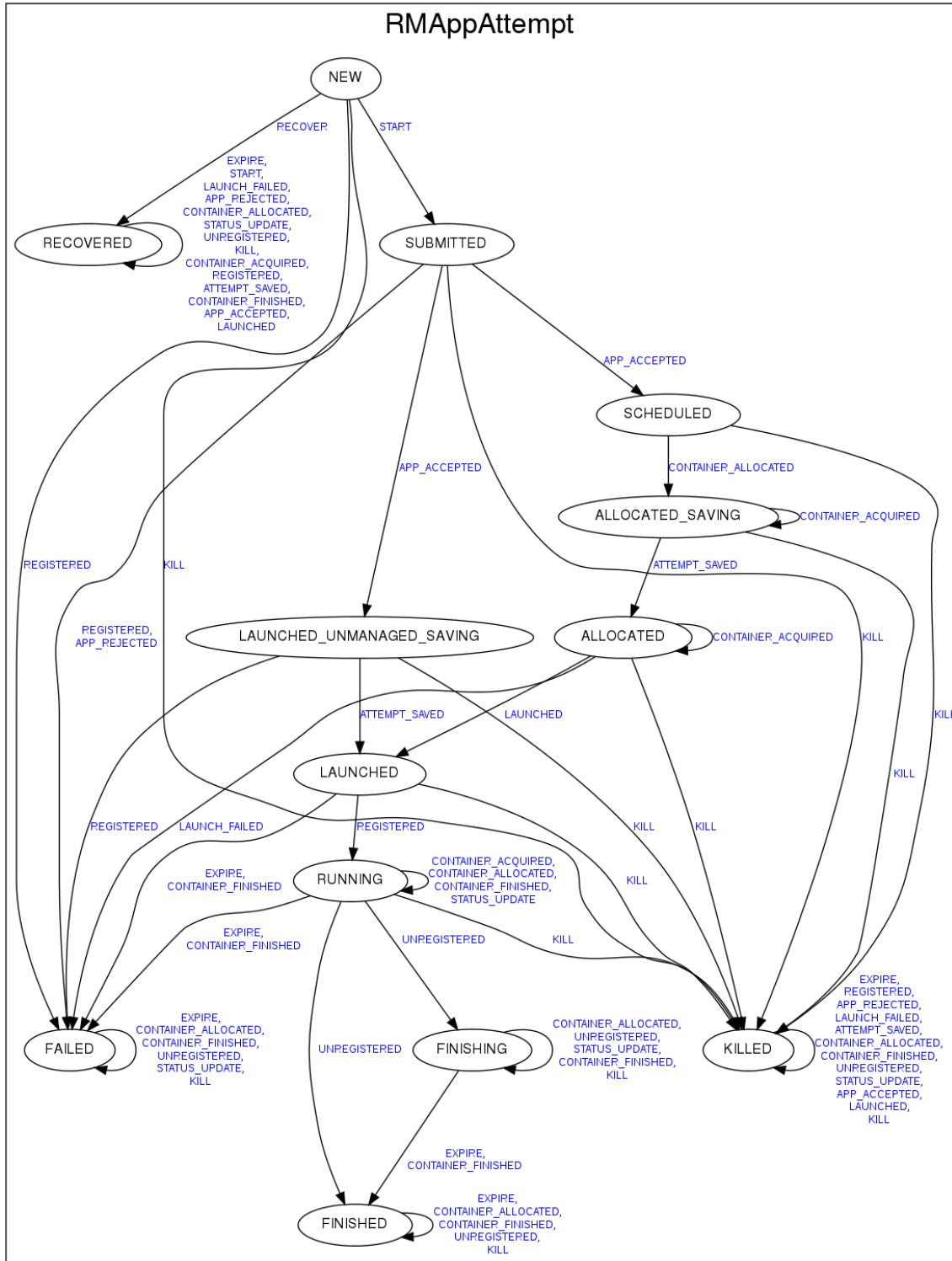


Figure A.3 – RMApAttempt's state machine

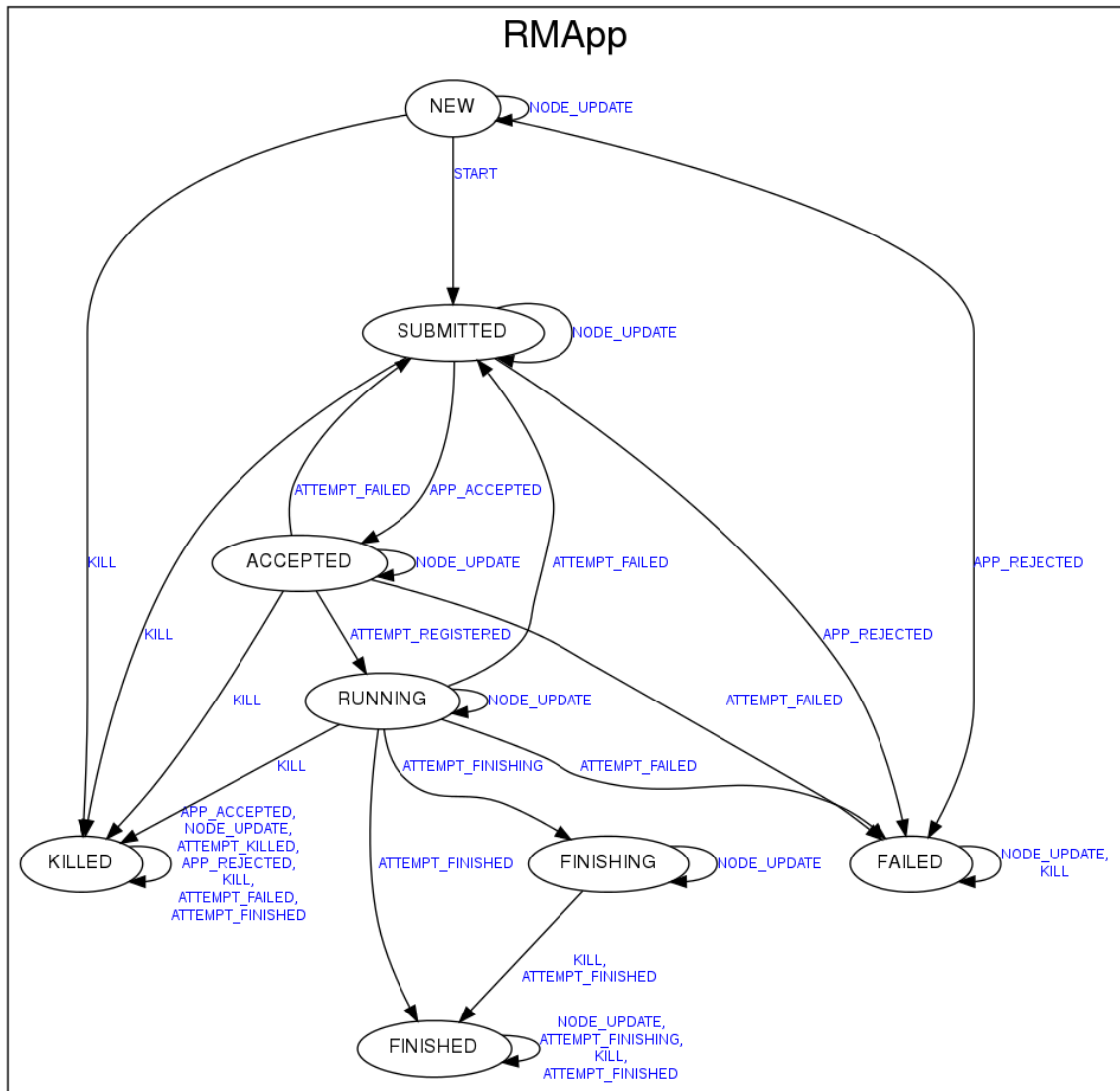


Figure A.4 – RMAApp's state machine

APPENDIX B – Grid’5000 execution environment configuration

This appendix has the necessary steps in order to setup a correctly working execution environment on Grid’5000 cluster.

The Apache Hadoop’s installation, contrary to end users standard program installations, doesn’t have a graphical interface. Actually the installation is just the extraction of files to a determined folder, however the environment configuration isn’t trivial and demands some network administration knowledge.

For the correct functioning of the framework, it is necessary to edit of some files responsible for describing the environment in which it will execute. Besides, it is necessary to have a network in which every node has access to the others. Knowing that Apache Hadoop has suffered some heavy changes in changing from 1.x to 2.x, it was expected that installing both versions would make the differences clearer.

In the beginning of this work the objective was to install and configure the Apache Hadoop in three situations:

- localhost, a single-node case.
- mini cluster, a multi-node case.
- Grid’5000, another multi-node case.

There are two configuration steps, the network step which refers to the configuration that doesn’t depend on Hadoop and the Hadoop step which refers to the parameters that influence Hadoop’s execution. This step is composed of tasks executed on the operational system, such as the user creation and ssh configuration. The second step, on the other hand, is the proper Hadoop environment configuration and, therefore, it has to change some configuration files in a way that Hadoop can be executed without errors.

The OS configuration is equal on both versions, since it has already been noted that it is independent of Hadoop. The Hadoop configuration, however, has a few differences between versions. Even so, both versions follow the same general rules. The Hadoop configuration is made through a bunch of xml files already mentioned in section 2, subsection 2.1.2, in which some properties containing name and value are inserted. This properties will be responsible for changing the default Hadoop behavior, but aside some properties changing their names, the biggest difference is that YARN version has a new xml file named *yarn-site.xml*, it contains all

parameters related to the YARN framework and therefore has a high influence on MapReduce and job execution performance.

Initially it was chosen to configure both versions on every specified instance, however, given the project focus the version 1.x was no longer a viable alternative and its use was discontinued.

The localhost environment configuration, is a simple process and occurred without problems after a basic contact with the framework. The mini cluster environment, was configured using two machines, already presenting a difference, in which the master could also be a slave on 1.x, in other words, it was possible to run NameNode and DataNode on the same node. On YARN version, the NameNode machine can't start a DataNode service, and the same is valid for ResourceManager and NodeManager. This change makes the tasks of processing and management totally apart and differentiable making the cluster more organized.

The real experiment starts when the Hadoop is deployed on a real cluster, like the Grid'5000, it was decided to use only the YARN version from this point onward. The installation of YARN requires only a few changes from mini cluster to Grid'5000 environment. The Grid also supplies a lot of tools that makes the job easier for Hadoop management.

At the end of this stage, some Hadoop peculiarities have already been cleared and the execution environment for posterior testing of the implementations was already deployed. Also, it is important to note that it was possible to identify some contexts that would present a high difficult to change the behavior. One of these behaviors would be the addition or removal of nodes on real time after the cluster initialization. The difficult comes from the way the service uses static reference to xml files that are read only on the initialization, making it impossible to use their values without restarting the services.

APPENDIX C – Source code edition and compilation

This appendix has the necessary steps in order to setup a correctly working compilation environment.

Given the project nature of generating a improved, context-aware, scheduler for the Apache Hadoop, it is necessary that this scheduler is included on the final distribution. Not only it has to be included, it has to be available for use by everyone who wants to try it. In order to achieve this, new jar files have to be generated from the modified code. Because of this a previous study was made about the necessary requisites in order to compile and generate these jar files.

The study began with the official documentation consultation, which included Apache Hadoop web site and help files included on the source code distributions. This way it was possible to create some steps required to compile the code. Starting from this list it was possible to identify the required dependencies to compile the code, which were then installed. The dependencies were: JDK 1.6 or higher, Maven 3.0, ProtocolBuffer 2.4.1 or higher and Cmake2.6 or higher.

The greatest objective of this process was to discover how the compilation took place and also how it would behave with the addition of new classes to standard code. Aiming to achieve this objective, a new but simple scheduler class was added. After the compilation process ended, the generated jar files were then copied to the Grid'5000 and deployed there in order to test if it would be possible to execute the compiled version in that environment.

Once the Hadoop services were deployed, it could be proven that the new scheduler was being used. It was also possible to identify the same vulnerability in the previous stage, in which the service has to be restarted in order to modify some of the Hadoop's parameters.

APPENDIX D – Example of semi-processed log from a experiment

The experiment results were collected using the Log System from Hadoop. Here is an example of a log already semi-processed, which means that this log has been filtered to show only the entries relevant to the analysis.

The following log snippet, shows the log when an application was submitted, in this case the application was the TeraSort. It is possible to note a lot of information, like the user who submitted and queue used, the applicationId, among others.

```

1 2014-01-13 13:19:21,517 INFO org.apache.hadoop.yarn.server.resourcemanager.
   scheduler.capacity.leafQueue: Application application_1389615375211_0002
   from user: hadoop activated in queue: default
2 2014-01-13 13:19:21,518 INFO org.apache.hadoop.yarn.server.resourcemanager.
   scheduler.capacity.leafQueue: Application added - appId:
   application_1389615375211_0002 user: org.apache.hadoop.yarn.server.
   resourcemanager.scheduler.capacity.leafQueue$User@5673e296, leaf-queue:
   default #user-pending-applications: 0 #user-active-applications: 1 #
   queue-pending-applications: 0 #queue-active-applications: 1

```

Another interesting log snippet is the one that show the assignment and completion of containers. The following snippet shows when the Reduce and ApplicationMaster containers are completed and the application is finished.

```

1 2014-01-13 13:24:05,016 INFO org.apache.hadoop.yarn.server.resourcemanager.
   scheduler.capacity.leafQueue: completedContainer container=Container: [
   ContainerId: container_1389615375211_0002_01_000040, NodeId: stremi-44.
   reims.grid5000.fr:34048, NodeHttpAddress: stremi-44.reims.grid5000.fr
   :8042, Resource: <memory:4830, vCores:1>, Priority: 10, Token: Token {
   kind: ContainerToken, service: 172.16.160.44:34048 }, ] resource=<memory
   :4830, vCores:1> queue=default: capacity=1.0, absoluteCapacity=1.0,
   usedResources=<memory:4830, vCores:1>usedCapacity=0.024998447,
   absoluteUsedCapacity=0.024998447, numApps=1, numContainers=1
   usedCapacity=0.024998447 absoluteUsedCapacity=0.024998447 used=<memory
   :4830, vCores:1> cluster=<memory:193212, vCores:96>
2 2014-01-13 13:24:11,146 INFO org.apache.hadoop.yarn.server.resourcemanager.
   scheduler.capacity.leafQueue: default used=<memory:0, vCores:0>
   numContainers=0 user=hadoop user-resources=<memory:0, vCores:0>
3 2014-01-13 13:24:11,147 INFO org.apache.hadoop.yarn.server.resourcemanager.
   scheduler.capacity.leafQueue: completedContainer container=Container: [
   ContainerId: container_1389615375211_0002_01_000001, NodeId: stremi-7.
   reims.grid5000.fr:58215, NodeHttpAddress: stremi-7.reims.grid5000.fr
   :8042, Resource: <memory:4830, vCores:1>, Priority: 0, Token: Token {
   kind: ContainerToken, service: 172.16.160.7:58215 }, ] resource=<memory
   :4830, vCores:1> queue=default: capacity=1.0, absoluteCapacity=1.0,
   usedResources=<memory:0, vCores:0>usedCapacity=0.0, absoluteUsedCapacity
   =0.0, numApps=1, numContainers=0 usedCapacity=0.0 absoluteUsedCapacity
   =0.0 used=<memory:0, vCores:0> cluster=<memory:193212, vCores:96>
4 2014-01-13 13:24:11,150 INFO org.apache.hadoop.yarn.server.resourcemanager.
   scheduler.capacity.leafQueue: Application removed - appId:
   application_1389615375211_0002 user: hadoop queue: default #user-pending
   -applications: 0 #user-active-applications: 0 #queue-pending-

```

```
applications: 0 #queue-active-applications: 0
```

Finally, there is something that influences a lot on the results, which is the time that an action took place. As it was possible to see on the above examples, the Hadoop Log System provides the hour, minute, second and milliseconds information. Thanks to this, two assignments that happened with mere milliseconds of difference were shown as 1 second delayed on chapter 4. The first container belongs to node stremi-44 and started at 13:19:29,995. The second container belongs to stremi-42 and started at 13:19:30:079

```
1 2014-01-13 13:19:29,995 INFO org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.LeafQueue: assignedContainer application=application_1389615375211_0002 container=Container: [ContainerId: container_1389615375211_0002_01_000030, NodeId: stremi-44.reims.grid5000.fr:34048, NodeHttpAddress: stremi-44.reims.grid5000.fr:8042, Resource: <memory:4830, vCores:1>, Priority: 20, Token: Token { kind: ContainerToken, service: 172.16.160.44:34048 }, ] containerId=container_1389615375211_0002_01_000030 queue=default: capacity=1.0, absoluteCapacity=1.0, usedResources=<memory:140070, vCores:29> usedCapacity=0.72495496, absoluteUsedCapacity=0.72495496, numApps=1, numContainers=29 usedCapacity=0.72495496 absoluteUsedCapacity=0.72495496 used=<memory:140070, vCores:29> cluster=<memory:193212, vCores:96>
2 2014-01-13 13:19:30,079 INFO org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.LeafQueue: assignedContainer application=application_1389615375211_0002 container=Container: [ContainerId: container_1389615375211_0002_01_000031, NodeId: stremi-42.reims.grid5000.fr:43999, NodeHttpAddress: stremi-42.reims.grid5000.fr:8042, Resource: <memory:4830, vCores:1>, Priority: 20, Token: Token { kind: ContainerToken, service: 172.16.160.42:43999 }, ] containerId=container_1389615375211_0002_01_000031 queue=default: capacity=1.0, absoluteCapacity=1.0, usedResources=<memory:144900, vCores:30> usedCapacity=0.74995345, absoluteUsedCapacity=0.74995345, numApps=1, numContainers=30 usedCapacity=0.74995345 absoluteUsedCapacity=0.74995345 used=<memory:144900, vCores:30> cluster=<memory:193212, vCores:96>
```

APPENDIX E – Main code changes performed

The changes that had the greatest impact on the behavior were the collector integration and the allocation re-scaling. The collector code is available on the link at the references, therefore, only the usage of the package will be inserted here in comparison to the original.

Starting with the original NodeManager creation, in which the totalResources are gotten. Note how the memoryMB and virtualCores variables are taken from the conf, which is the pointer to the default xml file. This method is from the NodeStatusUpdaterImpl class.

```

1
2 protected void serviceInit(Configuration conf) throws Exception {
3     int memoryMb =
4         conf.getInt(
5             YarnConfiguration.NM_PMEM_MB, YarnConfiguration.
6                 DEFAULT_NM_PMEM_MB);
7     float vMemToPMem =
8         conf.getFloat(
9             YarnConfiguration.NM_VMEM_PMEM_RATIO,
10                YarnConfiguration.DEFAULT_NM_VMEM_PMEM_RATIO);
11     int virtualMemoryMb = (int)Math.ceil(memoryMb * vMemToPMem);
12     int virtualCores =
13         conf.getInt(
14             YarnConfiguration.NM_VCORES, YarnConfiguration.
15                 DEFAULT_NM_VCORES);
16     this.totalResource = recordFactory.newRecordInstance(Resource.class);
17
18     this.totalResource.setMemory(memoryMb);
19     this.totalResource.setVirtualCores(virtualCores);
20     metrics.addResource(totalResource);
21     this.tokenKeepAliveEnabled = isTokenKeepAliveEnabled(conf);
22     this.tokenRemovalDelayMs =
23         conf.getInt(YarnConfiguration.RM_NM_EXPIRY_INTERVAL_MS,
24             YarnConfiguration.DEFAULT_RM_NM_EXPIRY_INTERVAL_MS);
25
26     // Default duration to track stopped containers on nodemanager is 10Min
27     // This should not be assigned very large value as it will remember all
28     // the
29     // containers stopped during that time.
30     durationToTrackStoppedContainers =
31         conf.getLong(YARN_NODEMANAGER_DURATION_TO_TRACK_STOPPED_CONTAINERS,
32             600000);
33     if (durationToTrackStoppedContainers < 0) {
34         String message = "Invalid_configuration_for_"
35             + YARN_NODEMANAGER_DURATION_TO_TRACK_STOPPED_CONTAINERS + "_default
36             + "value_is_10Min(600000).";
37         LOG.error(message);
38         throw new YarnException(message);
39     }
40     if (LOG.isDebugEnabled()) {

```



```

40     LOG.debug(YARN_NODEMANAGER_DURATION_TO_TRACK_STOPPED_CONTAINERS + "␣:
41         "
42         + durationToTrackStoppedContainers);
43     }
44     super.serviceInit(conf);
45     LOG.info("Initialized␣nodemanager␣for␣" + nodeId + ":" +
46         "␣physical-memory=" + memoryMb + "␣virtual-memory=" +
47         virtualMemoryMb +
48         "␣virtual-cores=" + virtualCores);
49 }

```

Then the changes made in the method to enable collectors. The rest of the method was not altered. The reason for the double casting is that the collector returns a Float and Double value and it's not possible to cast directly to int.

```

1 protected void serviceInit(Configuration conf) throws Exception {
2     PhysicalMemoryCollector memoryCollector = new
3     PhysicalMemoryCollector();
4     TotalProcessorsCollector processorsCollector = new
5     TotalProcessorsCollector();
6
7     int memoryMb = (int)(float)memoryCollector.collect().get(0)/1024;
8     int virtualCores = (int)(double)processorsCollector.collect().get(0);
9
10    this.totalResource = recordFactory.newRecordInstance(Resource.class);

```

The other change that had a strong impact in CapacityScheduler behavior was the insertion of recalculations of allocation limits inside the addNode method. This method belongs to the CapacityScheduler class. Starting with the original code.

```

1 private synchronized void addNode(RMNode nodeManager) {
2     this.nodes.put(nodeManager.getNodeID(), new FiCaSchedulerNode(
3     nodeManager,
4     usePortForNodeName));
5     Resources.addTo(clusterResource, nodeManager.getTotalCapability());
6     root.updateClusterResource(clusterResource);
7     ++numNodeManagers;
8 }

```

The same changes made on the addNode were also made on removeNode. Thus, when a node is killed, or is not accessible for a long period, it will be removed and the limits will be adjusted too.

```

1 private synchronized void addNode(RMNode nodeManager) {
2     this.nodes.put(nodeManager.getNodeID(), new FiCaSchedulerNode(
3     nodeManager,
4     usePortForNodeName));
5     Resource oldCap = Resources.clone(clusterResource);
6     Resources.addTo(clusterResource, nodeManager.getTotalCapability());
7     root.updateClusterResource(clusterResource);
8     ++numNodeManagers;
9     LOG.info("MEU␣Added␣node␣" + nodeManager.getNodeAddress() +

```

```

9      "clusterResource_before:" + oldCap + "nodecapability:" +
      nodeManager.getTotalCapability() + "clusterResource_now:" +
      clusterResource);
10 LOG.info("MEU_Changing_allocation_minimum_&_maximum_Actual_minimum:"
+ this.minimumAllocation + "actual_maximum:" + this.
maximumAllocation + ".\nDefault_settings: cluster_must_have_capacity
_for_at_least" + minimumContainers + "containers, and_no_more_than
" + maximumContainers + "containers.\n8GB_RAM_cluster_would_have_1GB
_minimum/maximum, 80GB_RAM_cluster_would_have_4GB_minimum_and_10GB_
maximum.");
11 this.minimumAllocation.setMemory(clusterResource.getMemory() /
maximumContainers);
12 this.minimumAllocation.setVirtualCores(clusterResource.getVirtualCores
() / maximumContainers);
13 this.maximumAllocation.setMemory(clusterResource.getMemory() /
minimumContainers);
14 this.maximumAllocation.setVirtualCores(clusterResource.getVirtualCores
() / minimumContainers);
15 if (this.minimumAllocation.getMemory() < minimumMemory)
16     this.minimumAllocation.setMemory(minimumMemory);
17 if (this.minimumAllocation.getVirtualCores() < minimumVcores)
18     this.minimumAllocation.setVirtualCores(minimumVcores);
19 if (this.maximumAllocation.getMemory() < this.minimumAllocation.
getMemory())
20     this.maximumAllocation = this.minimumAllocation;
21 LOG.info("MEU_New_minimumAllocation_settings:" + minimumAllocation + "
\nNew_maximumAllocation_settings:" + maximumAllocation);

```