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

ESCALONAMENTO ADAPTATIVO PARA O APACHE HADOOP

DISSERTAÇÃO DE MESTRADO

Guilherme Weigert Cassales

Santa Maria, RS, Brasil

2016

ESCALONAMENTO ADAPTATIVO PARA O APACHE HADOOP

Guilherme Weigert Cassales

Dissertação apresentada ao Curso de Mestrado Programa de
Pós-Graduação em Informática (PPGI), Área de Concentração em
Computação, da Universidade Federal de Santa Maria (UFSM, RS),
como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

Orientadora: Profª. Drª. Andrea Schwertner Charão

Santa Maria, RS, Brasil

2016

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

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

ESCALONAMENTO ADAPTATIVO PARA O APACHE HADOOP

elaborada por
Guilherme Weigert Cassales

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

COMISSÃO EXAMINADORA:

Andrea Schwertner Charão, Profª. Drª.
(Presidente/Orientadora)

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

Patrícia Pitthan de Araújo Barcelos, Profª. Drª. (UFSM)

Santa Maria, 20 de Janeiro de 2016.

RESUMO

Dissertação de Mestrado
Curso de Ciência da Computação
Universidade Federal de Santa Maria

ESCALONAMENTO ADAPTATIVO PARA O APACHE HADOOP
AUTOR: GUILHERME WEIGERT CASSALES
ORIENTADORA: ANDREA SCHWERTNER CHARÃO
Local da Defesa e Data: Santa Maria, 20 de Janeiro de 2016.

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 distribuidas. O *framework Apache Hadoop* (Apache Hadoop, 2013), é uma delas e poupa o programador as terefas 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

Master's Dissertation
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 20$^{\text{th}}$, 2016.

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.

LISTA DE FIGURAS

LISTA DE APÊNDICES

SUMÁRIO

1 INTRODUÇÃO

Apache Hadoop é um *framework* de computação paralela e distribuída para processamento de grandes conjuntos de dados e implementa o paradigma de programação MapReduce (**?**). O Apache Hadoop é projetado para ser escalável de um único servidor a milhares de máquinas, cada uma oferecendo processamento e armazenamento local. Esta capacidade de utilizar grande quantidade de *hardware* barato e a crescente importância do processamento de dados não estruturados tornaram o Apache Hadoop uma ferramenta relevante no mercado (**?**).

Sem uma configuração específica pelo administrador, o Apache Hadoop assume que está sendo utilizado em um *cluster* homogêneo para execução de aplicações MapReduce. Uma vez que o desempenho geral do *cluster* está ligado ao escalonamento de tarefas, o desempenho do Hadoop pode diminuir significativamente quando executado em ambientes que não satisfaçam a suposição feita no projeto do *framework*, ou seja, em ambientes dinâmicos ou heterogêneos (**?**).

Esta é uma preocupação especial quando tenta-se utilizar o Hadoop em grids pervasivos. Grids pervasivos são uma alternativa aos *clusters* dedicados, dado que o custo de aquisição e manutenção de um *cluster* dedicado continua alto e dissuasivo para muitas organizações. De acordo com (**?**), grids pervasivos representam a generalização extrema do conceito de grid por possuírem recursos pervasivos, ou seja, recursos computacionais ociosos que são incorporados ao ambiente e então requisitados com objetivo de processar tarefas de maneira distribuída.

Estes grids podem ser vistos como grids formados por recursos existentes (desktops, servidores, etc.) que ocasionalmente contribuem para o poder de processamento do grid. Estes recursos são inerentemente heterogêneos e potencialmente móveis, entrando e saindo do grid dinamicamente. Sabendo disto, é possível afirmar que grids pervasivos são, em essência, ambientes heterogêneos, dinâmicos e compartilhados; tornando o gerenciamento dos recursos complexo e, quando executado de maneira ineficiente, dimnuindo o desempenho do escalonamento de tarefas (**?**).

Muitos trabalhos propuseram-se a melhorar a adaptabilidade do *framework* Apache Hadoop em ambientes que divergem da suposição inicial, cada um possuindo sua própria proposta e objetivos (**?**) (**?**) (**?**) (**?**) (**?**). Em geral os trabalhos dependem da aquisição de algum dado de contexto e subsequente tomada de decisão para maximização de algum objetivo específico.

Sabe-se que, o Apache Hadoop é baseado em configuração estática de arquivos e que

as versões correntes, apesar de possuírem tolerância a falhas, não adaptam-se a variações de recursos ao longo do tempo. Além disto, os procedimentos de instalação forçam o administrador a definir manualmente as características de cada recurso em potencial, como a memória e o número de *cores* de cada máquina, tornando a tarefa difícil e demorada em ambientes heterogêneos. Todos estes fatores impedem a utilização da capacidade total do Hadoop em ambientes voláteis, portanto, é essencial possuir sensibilidade ao contexto para tornar esta adaptação possível.

Este trabalho propõe-se a aumentar a adaptabilidade dos mecanismos de escalonamento do Apache Hadoop utilizando coleta e transmissão de dados de contexto como meio para solucionar os problemas gerados pelo compartilhamento de nós do *cluster*. Buscou-se atingir estes objetivos com o mínimo possível de intrusão e alterações nos mecanismos chave de escalonamento. A coleta de dados é feita com base no tempo médio de duração dos *containers* em execução e a informação só é transmitida caso hajam mudanças em relação à última coleta realizada. A transmissão de dados ocorre por meio de uma tabela Hash distribuída, quando esta tabela é alterada o escalonador é alertado e realiza uma atualização nas informações pertinentes. A solução implementada apresenta melhorias de até 40% em alguns casos de testes, os quais utilizam aplicações com características diferentes (CPU-bound, I/O-bound, CPU e I/O-bound), provando ser uma alternativa viável para o aumento da adaptabilidade do Apache Hadoop.

2 FUNDAMENTOS E REVISÃO DE LITERATURA

É necessário definir alguns termos, técnicas e/ou ferramentas para o entendimento deste trabalho. Por exemplo, a conceituação formal de sensibilidade ao contexto é muito importante, uma vez que constitue-se da técnica de obtenção dos dados. Como complemento à sensibilidade ao contexto define-se o que é o ZooKeeper, a ferramenta utilizada para transmissão dos dados coletados. Além disso, é relevante que exista a compreensão de como o Apache Hadoop, seu escalonamento e o paradigma de programação utilizado funcionam, bem como quais trabalhos já foram feitos neste âmbito.

2.1 Sensibilidade ao Contexto

Quando um usuário acessa um site por meio de um dispositivo móvel este site carrega automaticamente sua versão *mobile*, a qual possui alterações que aumentam a compatibilidade com o tipo de dispositivo sendo utilizado. Uma situação semelhante ocorre quando navegadores utilizam dados de localidade para melhorar os resultados de um motor de buscas, mostrando primeiro os resultados mais relacionados com a região ou idioma do usuário. Nota-se que nas duas situações a aplicação utiliza dados específicos, o tipo do dispositivo e a localização do usuário, coletados no momento da execução e utiliza-os para adaptar o seu funcionamento de maneira a oferecer maior conforto ou usabilidade ao usuário. Estes dados representam o contexto dos usuários, o qual é definido por (DEY, 2001) como qualquer informação que pode ser utilizada para caracterizar a situação de uma entidade (pessoa, lugar ou objeto) considerada relevante para a interação entre usuário e aplicação.

Sendo assim, se uma aplicação é capaz de coletar informações sobre a situação do sistema no qual está sendo executada, ela é capaz de coletar informações de contexto. Porém a simples coleta não aumenta, de fato, o desempenho desta aplicação, é necessário que ela seja capaz de responder às mudanças do ambiente detectadas. Esta capacidade de detecção e reação é caracterizada como sensibilidade ao contexto por (**?**) e vai ao encontro da definição de (BALDAUF; DUSTDAR; ROSENBERG, 2007) em que o sistema deve detectar mudanças e adaptar suas operações sem intervenção explícita do usuário, aumentando assim a usabilidade e eficácia da aplicação.

2.2 Hadoop

O *framework* Apache Hadoop origina-se de outro projeto da Apache, o Apache Nutch (Apache Nutch, 2013). O Apache Nutch iniciou em 2002 como um motor de buscas de código livre, porém o projeto encontrou problemas devido a sua arquitetura. Quando a Google publicou um artigo em 2003 descrevendo a arquitetura utilizada no seu sistema de arquivos distribuídos, chamado GFS (*Google File System*), os desenvolvedores do Nutch notaram que uma arquitetura semelhante resolveria seus problemas de escalabilidade. A implementação da ideia foi iniciada em 2004 e o resultado foi nomeado *Nutch Distributed Filesystem* (NDFS). Contudo, a medida em que o projeto se desenvolvia o propósito original do Nutch era deixado em segundo plano, o que culminou na criação de um novo projeto em 2006. Este novo projeto foi nomeado Apache Hadoop e possui o propósito de facilitar o processamento distribuído através do paradigma MapReduce.

2.2.1 Arquitetura geral do *Apache Hadoop*

O *framework* Apache Hadoop é organizado numa arquitetura de mestre-escravo e possui dois serviços principais, o serviço de armazenamento (HDFS - Hadoop Distributed File System) e o de processamento (YARN - Yet Another Resource Negotiator), os quais podem ser vistos na Figura 2.1.
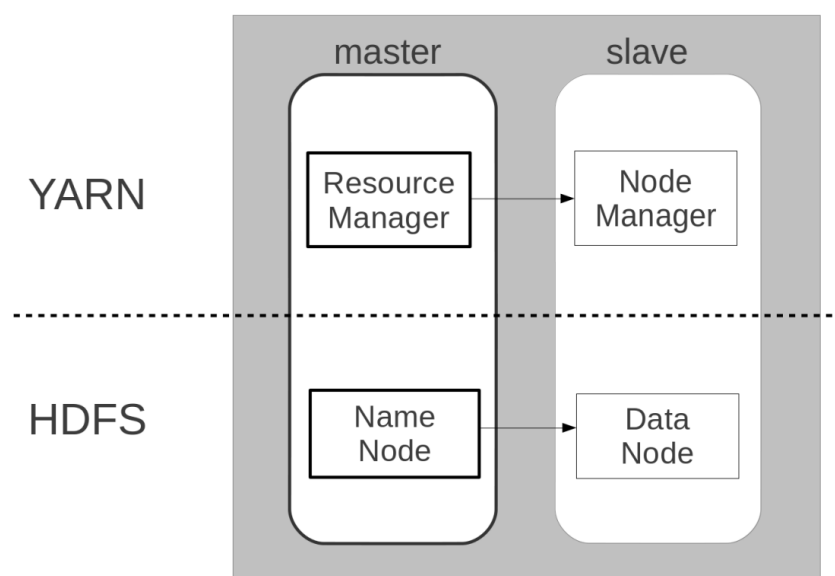
Figura 2.1 – Arquitetura geral do Apache Hadoop

Nesta figura, é possível notar a existência de 4 componentes do *framework*. Os componentes acima da linha pontilhada pertencem ao YARN, sendo o Resource Manager e o Node Manager os serviços mestre e escravo respectivamente. O HDFS é composto pelos componentes abaixo da linha pontilhada, sendo o Name Node e o Data Node os serviços mestre e escravo respectivamente.

2.2.1.1 HDFS

Grande parte do ganho de desempenho oferecido pelo Hadoop decorre do comportamento de levar o processamento até os dados, ou seja, todo processamento é feito com dados locais. Esta abordagem só é possível de ser realizada graças ao HDFS. Dentro do Name Node são mantidas informações de quais partes de quais arquivos estão em cada Data Node, ou seja, todos os arquivos estão divididos num grande HD distribuído e o mestre sabe exatamente quais blocos cada escravo possui. Dessa forma cada nó executa tantos Maps ou Reduces quanto a quantidade de arquivos locais permitir, diminuindo a necessidade de utilizar a rede para transferência de arquivos e deixando-a disponível para ser utilizada para a transferência dos resultados que são os dados já reduzidos. Um problema dessa abordagem é que o Hadoop possui uma latência muito alta, sendo desaconselhável o uso do Hadoop em aplicações críticas ou de tempo real (**?**). A Figura 2.2 apresenta um esquema básico da arquitetura do HDFS.
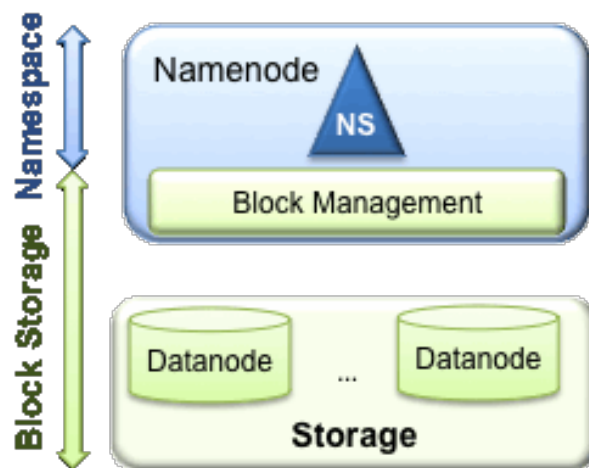
Figura 2.2 – Arquitetura geral do HDFS (HADOOP, 2013a)

2.2.1.2   YARN

Sendo o componente do Apache Hadoop responsável pela execução do *MapReduce*, o YARN realiza tarefas de gerenciamento e execução do processamento. Um dos objetivos do YARN é tornar a tarefa de processamento totalmente independente das tarefas de armazenamento, possibilitando que o *cluster* seja utilizado em conjunto com outras ferramentas que não utilizem o paradigma MapReduce (**?**). A Figura 2.3 apresenta um esquema básico da arquitetura do YARN.
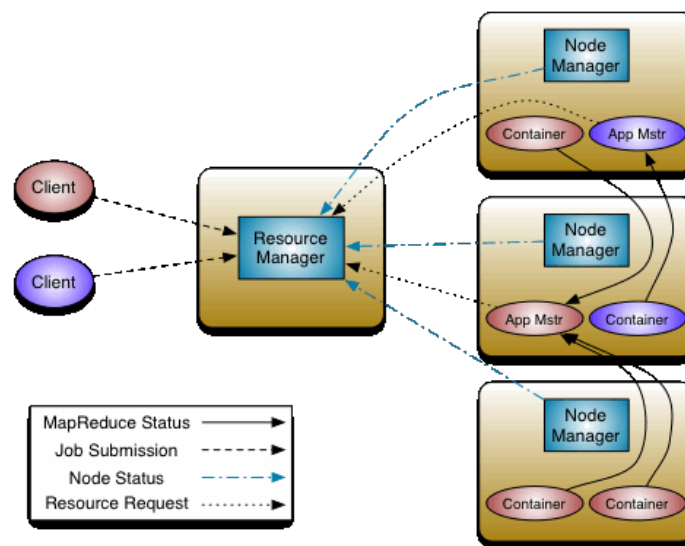
Figura 2.3 – Arquitetura geral do YARN (HADOOP, 2013b)

Na imagem é possível observar dois novos componentes do YARN. O primeiro é um componente do Node Manager, possuindo o papel de um escalonador interno de cada aplicação (Application Master - referenciado na imagem como App Mstr), sendo também conhecido por escalonador de tarefas. Considera-se que tarefas sejam uma fração do processamento das aplicações, ou seja, cada tarefa de Map ou Reduce corresponde a uma das divisões que serão processadas em paralelo. É importante não confundir este componente com o escalonador de aplicações, o qual é um componente do Resource Manager e pode ser melhor compreendido com a leitura da Seção 2.5. O outro componente presente na figura é o Container, o qual representa uma alocação de recursos em um nó qualquer do *cluster*. A importância do container vem do fato de que todas as tarefas são executadas em uma instância de container.

É importante notar que existem 2 instâncias de Application Master na figura e que elas estão, assim como os clientes e os containers, coloridas de rosa ou roxo para indicar que pertencem à mesma aplicação, ou seja, o cliente rosa lançou uma aplicação que possui o Application

Master e mais 3 containers de processamento. Nota-se também que o Resource Manager recebe informações tanto dos clientes quanto dos Node Managers e Application Masters, centralizando todas elas para um controle dos recursos disponíveis.

2.2.2 Configuração do ambiente de execução do Hadoops

Uma característica importante do Hadoop tem relação com sua configuração. Sabe-se que todo sistema necessita de uma configuração por parte de seu administrador, e o Hadoop não é diferente com relação a isso. O Hadoop utiliza uma série de arquivos em cada um dos nós do *cluster* para definir sua configuração. Estes arquivos de configuração são, na verdade, arquivos XML compostos por parâmetros de configuração e valores que influenciam o comportamento do *framework* no *cluster*. Para conhecimento, esses arquivos são: *core-site.xml, yarn-site.xml, mapred-site.xml* e *hdfs-site.xml*. Cada um destes arquivos possui propriedades de um serviço do Hadoop, como exemplo o arquivo *hdfs-site.xml* é responsável pela configuração do HDFS na máquina em questão.

É importante salientar que para a execução distribuída, ao menos algumas propriedades mais simples dos arquivos de cada nó, seja este mestre ou escravo, devem ser configuradas. No caso do administrador desejar fazer uma configuração mais específica de cada nó escravo ele deverá editar as propriedades nos arquivos de cada um dos nós. Caso o administrador não queira configurar os nós escravos, eles irão executar com base nos valores *default*, contudo esta decisão irá, provavelmente, afetar o desempenho do *cluster* devido à má configuração.

2.3 MapReduce

O paradigma de programação MapReduce é, geralmente, associado com implementações que processam e geram grandes conjuntos de dados. Neste paradigma, todo processamento é dividido em duas etapas, a etapa de Map e a de Reduce, originárias das linguagens funcionais. Como consequência da divisão em duas etapas, o resultado da primeira é a entrada da segunda e esta transferência de dados é baseada em de pares de chave e valor, o que torna possível expressar uma vasta gama de tarefas reais por meio deste paradigma (DEAN; GHEMAWAT, 2004).

O *work-flow* padrão de uma aplicação MapReduce inicia com a entrada de dados, que será dividida em *n* partes e cada parte será processada individualmente por uma tarefa Map. O

resultado das tarefas Map já será na forma de pares chave e valor, que serão passados como entrada para as tarefas de Reduce. Por sua vez, as tarefas de Reduce irão receber todos os pares com determinada chave e aplicar um algoritmo sobre os pares, fornecendo uma saída inteligível (**?**).

A naturalidade do paralelismo deste paradigma, torna a programação da aplicação mais simples. Ao utilizar o paradigma do MapReduce o programador deve apenas pensar numa solução que siga o *work-flow* do paradigma, e o parelelismo será inerente.

2.4 ZooKeeper

O ZooKeeper é um projeto da Apache e fornece ferramentas eficientes, confiáveis e tolerantes à falha para a coordenação de sistemas distribuídos (**?**). Inicialmente, o ZooKeeper foi implementado como um componente do Hadoop e virou um projeto próprio conforme cresciam suas funcionalidades e sua utilização em outras aplicações.

No caso deste trabalho, os serviços do ZooKeeper são utilizados para monitorar e transmitir as informações de contexto coletadas nos nós escravos. No ZooKeeper existe um componente chamado *znode* onde é possível inserir qualquer tipo de dado e todos as instâncias de clientes e servidores (no caso de um servidor replicado) possuem acesso à este *znode*. A comunicação é feita através de processos que atualizam e processos que lêem o conteúdo destes componentes.

2.5 Escalonadores para o Hadoop

Apesar de existirem dois níveis de escalonamento no Hadoop, escalonamento de aplicações e de tarefas, este trabalho influencia apenas o escalonamento de aplicações. O escalonador de aplicações gerencia qual será a primeira aplicação a receber um container para seu escalonador de tarefas e quais serão os escalonadores de tarefa que receberão recursos para execução (**?**). O Hadoop já inclui alguns escalonadores que oferecem maneiras diferentes para a realização do escalonamento de aplicações. Para alterar o escalonador utilizado é necessário alterar uma propriedade no arquivo *yarn-site.XML* e reiniciar o Resource Manager.

Dentre os escalonadores incluídos no Hadoop, o mais simples é o Hadoop Internal Scheduler que utiliza o algoritmo FIFO e tem boa performance em *clusters* onde não existe competição por recursos. Este escalonador suporta até 5 níveis de prioridade, porém a decisão da

próxima aplicação a ser executada sempre levará em consideração a hora de submissão.

Um pouco mais complexo que o Internal Scheduler, o Fair Scheduler é utilizado principalmente para o processamento de lotes de aplicações pequenas e rápidas, opera baseado em um escalonamento de dois níveis e possui o objetivo de realizar uma divisão justa dos recursos. O primeiro nível realiza o escalonamento na forma de filas para cada usuário ativo, dividindo os recursos do cluster igualitariamente entre as filas. Enquanto isso, o segundo nível realiza o escalonamento dentro de cada fila da mesma forma que o Internal Scheduler (Fair Scheduler, 2013).

A terceira opção, e também o padrão do Hadoop nas últimas versões, é o Capacity Scheduler, o qual foi projetado para a utilização compartilhada do Hadoop e busca a maximização do *throughput* e da utilização do *cluster*. Seu funcionamento baseia-se em garantias mínimas de capacidade para os usuários, ou seja, qualquer usuário terá sempre uma garantia mínima de recursos para utilização. Porém quando algum usuário está com seus recursos ociosos o escalonador repassa a capacidade deste usuário para aqueles que estão utilizando o *cluster*. Esta estratégia fornece elasticidade com um bom custo benefício, uma vez que diferentes organizações possuem diferentes horários de pico para o processamento de informações. Este escalonador é capaz de rastrear os recursos registrados no Resource Manager, embora esta informação pode não ser consistente com a realidade, e monitorar quais deles estão livres e quais estão sendo utilizados pelo *framework* (**?**).

A existência destes escalonadores adiciona flexibilidade no gerenciamento do *framework*. Apesar disso, os escalonadores disponíveis não detectam nem reagem à dinamicidade e heterogeneidade do ambiente. Para a utilização do Hadoop em ambientes pervasivos é necessário que exista uma capacidade de adaptação neste componente.

2.6 Trabalhos Relacionados

Existem outras implementações de escalonadores além dos 3 escalonadores já inclusos no Hadoop, nas quais é possível identificar diversas propostas de adaptação. Cada proposta possui métodos e objetivos específicos, tornando um estudo sobre estas implementações interessante como ponto de partida para a proposta deste trabalho. Logo, buscou-se identificar quais técnicas foram mais utilizadas e quais os tipos de adaptação mais explorados nestes escalonadores. A seguir encontram-se os trabalhos relacionados e um breve resumo sobre a proposta, método e objetivos esperados com a adaptação.

Os autores do escalonador CASH (*Context Aware Scheduler for Hadoop* - Escalonador Sensível ao Contexto para o Hadoop) (KUMAR et al., 2012) têm o objetivo de melhorar o rendimento geral do *cluster*. O trabalho utiliza a hipótese de que grande parte das aplicações são periódicas e executadas no mesmo horário, além de possuírem características de uso de CPU, rede, disco etc. semelhantes. O trabalho ainda leva em consideração que com o passar do tempo os nós tendem a ficar mais heterogêneos. Baseados nestas hipóteses e com objetivo de melhorar o desempenho geral, foi implementado um escalonador que classifica tanto as aplicações como as máquinas com relação ao seu potencial de CPU e E/S, podendo então distribuir as aplicações para máquinas que tem uma configuração apropriada para sua natureza.

No trabalho *A Dynamic MapReduce Scheduler for Heterogeneous Workloads* (Um Escalonador de MapReduce Dinâmico para Cargas de Trabalho Heterogêneas) (TIAN et al., 2009), os autores utilizam a técnica de classificar as aplicações e máquinas de acordo com a quantidade de E/S ou CPU. E assim como no CASH, o principal objetivo é a melhora de rendimento no *cluster*. Uma das diferenças, no entanto, é que esta implementação utiliza um escalonador com três filas.

Semelhante às propostas anteriores, a proposta COSHH (*A Classification and Optimization based Scheduler for Heterogeneous Hadoop Systems* - Um Escalonador Baseado em Classificação e Otimização para Sistemas Heterogêneos do Hadoop) (RASOOLI; DOWN, 2012) utiliza a classificação das aplicações e máquinas em classes e busca por pares que possuam a mesma classe. Esta busca é feita por um algoritmo que reduz o tamanho do espaço de busca para melhorar o desempenho. O objetivo desta solução é a melhora do tempo médio em que as aplicações são completadas, além de oferecer um bom desempenho quando utilizando somente a fatia mínima de recursos e, ainda, proporcionar uma distribuição justa.

O escalonador LATE (*Longest Approximation Time to End* - Aproximação do Tempo de Término mais Longo) (ZAHARIA et al., 2008), utiliza como informação de contexto o tempo estimado de término da tarefa baseado numa heurística que faz a relação de tempo decorrido e do *score* – um valor que indica quanto do processamento já foi feito. Essa informação também é utilizada para gerar um limiar que indique quando a lentidão de uma tarefa indica sintomas de erros e a partir desta informação iniciar uma tarefa especulativa em outra máquina possivelmente mais rápida. Este trabalho possui o objetivo de reduzir o tempo de resposta em *clusters* grandes que executam muitas aplicações de pequena duração.

Outro trabalho que utiliza a ideia de mensuração do progresso de uma tarefa é o SAMR

(*A Self-adaptative MapReduce* - MapReduce Auto Adaptativo) (CHEN et al., 2010), nesta implementação a informação de contexto é referente ao cálculo do progresso de uma tarefa com objetivo de identificar se é o lançamento de uma tarefa especulativa é necessário ou não. Uma diferença desta solução é com relação ao cálculo do progresso, o qual varia de acordo com informações do ambiente em que a tarefa está executando. O principal objetivo do trabalho é a redução do tempo de execução das tarefas. As informações do ambiente utilizadas para a tomada de decisão consistem de informações históricas contidas em cada nó, e a decisão é tomada após um ajuste do peso de cada estágio do processamento.

A proposta dos autores do *Quincy* (ISARD et al., 2009) difere-se de todos os outros trabalhos, pois possui um escopo muito maior e visa tanto o *Hadoop* como outras ferramenas. O trabalho possui como objetivo a melhora do desempenho geral de um *cluster*, e utiliza a distribuição de recursos como informação de contexto para alcançá-lo. A contribuição do trabalho é uma modificação da maneira tradicional de tratamento da distribuição dos recursos. A solução proposta mapeia os recursos em um grafo de capacidades e demandas, para então calcular o escalonamento ótimo a partir de uma função global de custo.

A proposta *Improving MapReduce Performance through Data Placement in heterogeneous Hadoop Clusters* (Melhorando o Desempenho do MapReduce em Clusters Heterogêneos com Hadoop Através da Localização dos Dados) (XIE et al., 2010), busca melhorar o desempenho de aplicações CPU-bound através da melhor distribuição destes dados. Esta solução utiliza principalmente a localidade dos dados como informação para tomada de decisões. O ganho de desempenho é dado pelo rebalanceamento dos dados nos nós, deixando nós mais rápidos com mais dados. Isso diminui o custo de tarefas especulativas e de transferência de dados pela rede.

Após estudo dos trabalhos, nota-se que muitos deles tem por objetivo a diminuição do tempo de resposta ou a melhoria do rendimento de maneira geral, os quais diferem dos objetivos do presente trabalho, que na verdade busca proporcionar uma melhor adaptação do *Hadoop* a um ambiente heterogêneo.

Constata-se também que há uma diversidade de contextos levados em consideração, contudo é possível identificar temas recorrentes como : a classificação dos *jobs* e dos nós quanto ao potencial de E/S ou de CPU, a avaliação do progresso da *task* na decisão de lançar ou não uma nova *task* especulativa.

2.7 TABELA

Situação	Default	Trabalho
Falha nó (morto)	perde tasks (precisa reiniciar quando tiver recursos) e desregistra NM (diminui recursos)	Idem default
Novo nó (registro)	registar novo NM. Aumenta recursos e escalona tasks que estavam esperando	Idem default
Nó inicia compartilhamento (- recurso)	**não reage**, continua alocando containers como se estivesse tudo disponível	**diminui o recurso do nó. Não afeta containers já alocados, mas só irá alocar quando possuir recurso disponível**
Nó termina compartilhamento (+ recurso)	**não reage**, como não alterou nada na quando perdeu recursos ele ficará agora com a info correta	**aumenta o recurso do nó. Inicia nova rodada de escalonamento, é como se um novo nó entrasse**
Cluster heterogêneo	tem que alterar as propriedades dos recursos no XML de todos escravos refletindo as configurações dos nós. Caso contrário existirão nós sobrecarregados ou subutilizados.	Cada nó tera a informação correta. Ganho de desempenho por não sobrecarregar nem subutilizar.

Tabela 2.1 – tabela de casos

Situação	Teste "Real"	Teste Controlado	Opção extra
Nó com compartilhamento (1 teste por vez)	retirar/devolver **de verdade** os recursos, mais fácil com **CPU**	"simula"a diminuição de recursos em todos nós tirando metade dos nós e dobrando a capacidade informada dos restantes. A "volta"é mais difícil, inicia job com todos nós e informação de 50% e depois volta ao normal	Importante é sobrecarregar os nós. Sem "tirar"recursos como teste real, daria para utilizar máquina virtual
Nó com compartilhamento (2 testes, - depois +)	possível, só organizar para sincronizar estas mudanças	possível, mas envolve muita gambiarra.	

Tabela 2.2 – tabela de testes

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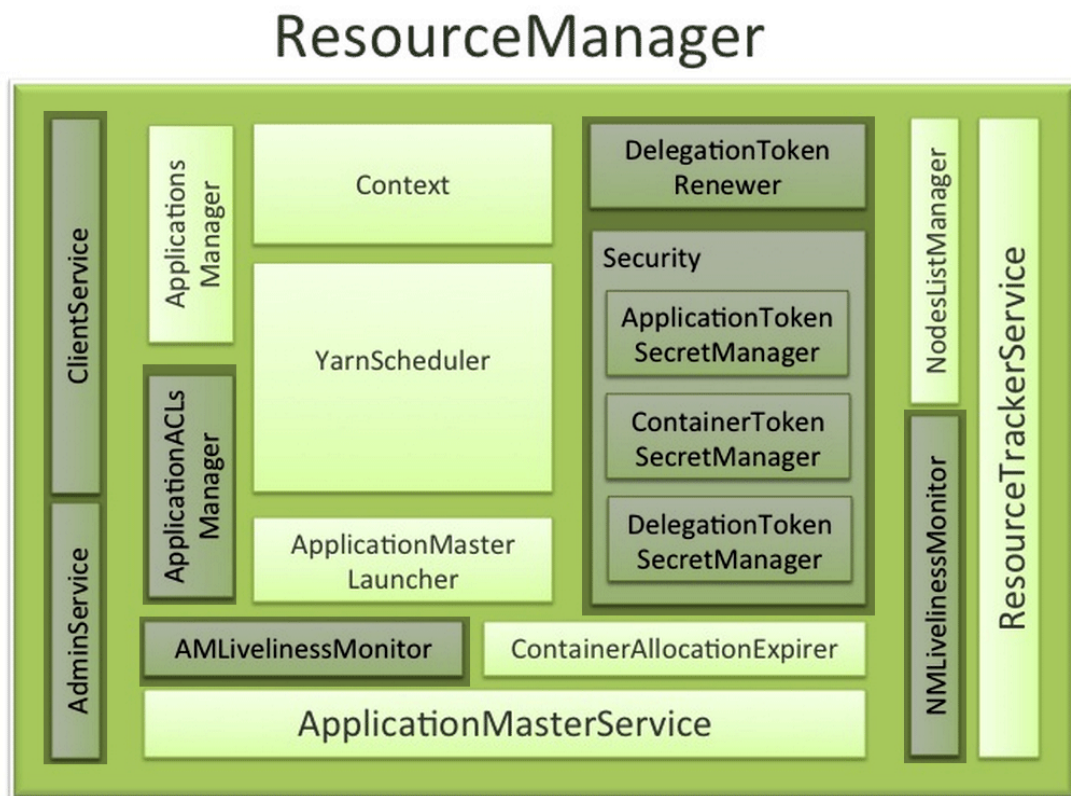
Figure 3.1 – ResourceManager Component Diagram (HortonWorks, 2014a)

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 Application-Master when the application has finished, either normally of 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 decomissioned 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.
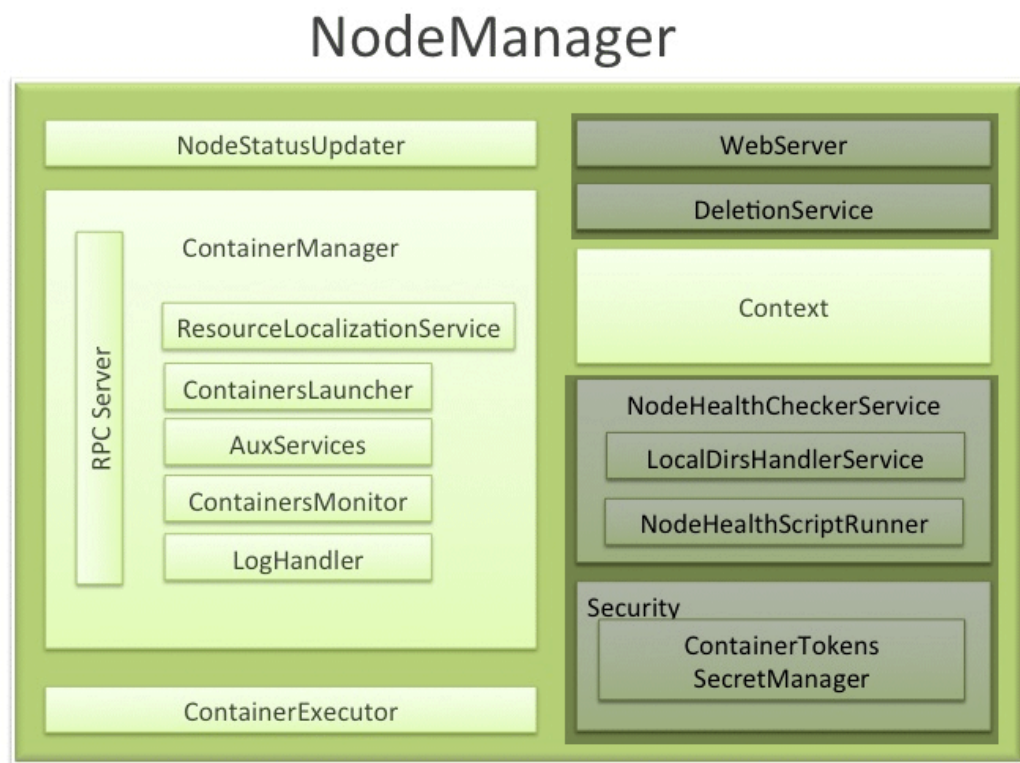
Figure 3.2 – NodeManager Component Diagram (HortonWorks, 2014b)

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, ResourceLocalizationService, 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.
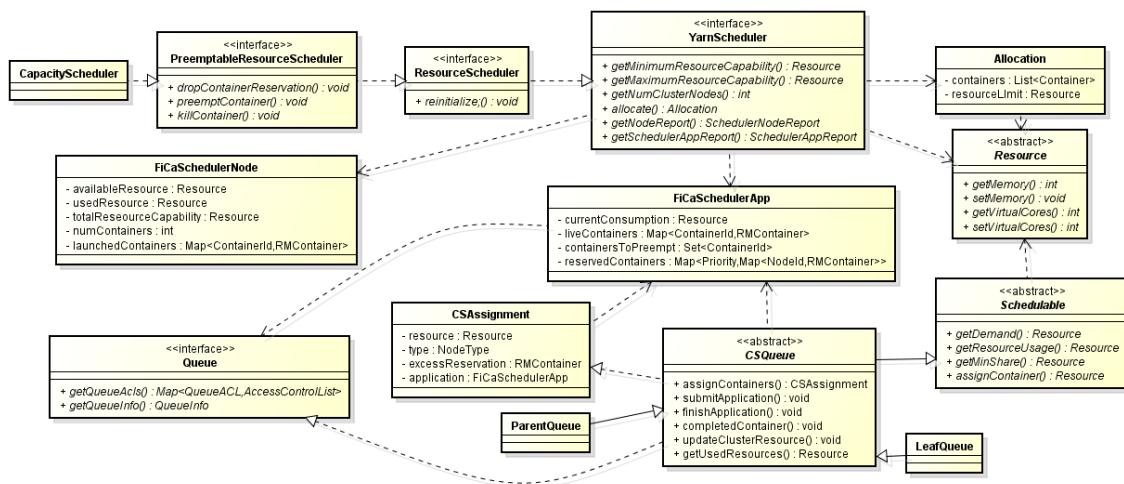
Figure 3.3 – Class Diagram with the main CapacityScheduler classes

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.

- PreemtableResourceScheduler: 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 RMApp/RMAppAttempt. 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. RMApp and RMAppAttempt represents the applications sent to the scheduler. This diagram is represented in the figure 3.4.
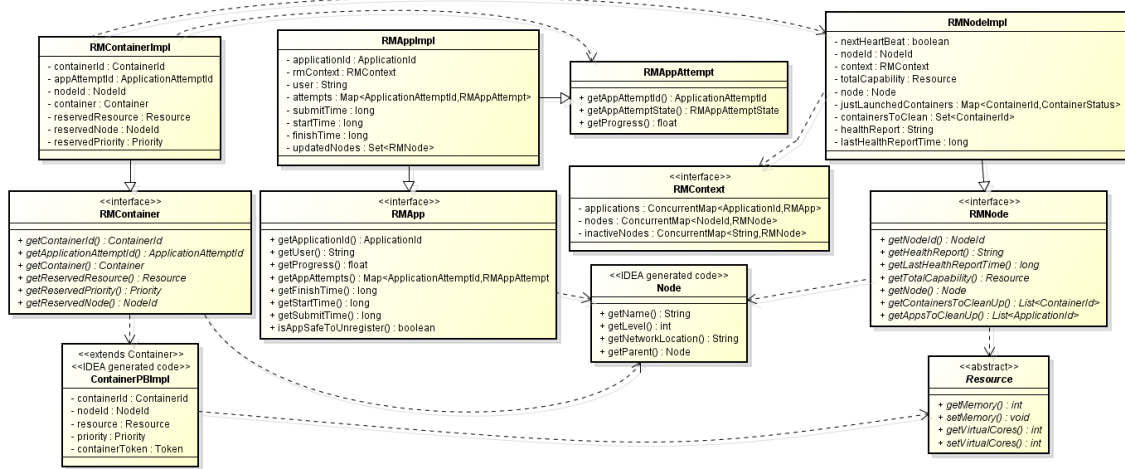
Figure 3.4 – Class Diagram with the main RM components

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 RMApplications.

The importance of RMContainers, RMNodes and RMApplications 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 RMApplication 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 Apêndice 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 it's 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 it's 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 table 3.1.

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 portraits the flow of operations executed by the Hadoop in order to determine the granted resources.

The default scenario just demonstrates how Hadoop allocation will behave in case there are no interventions.
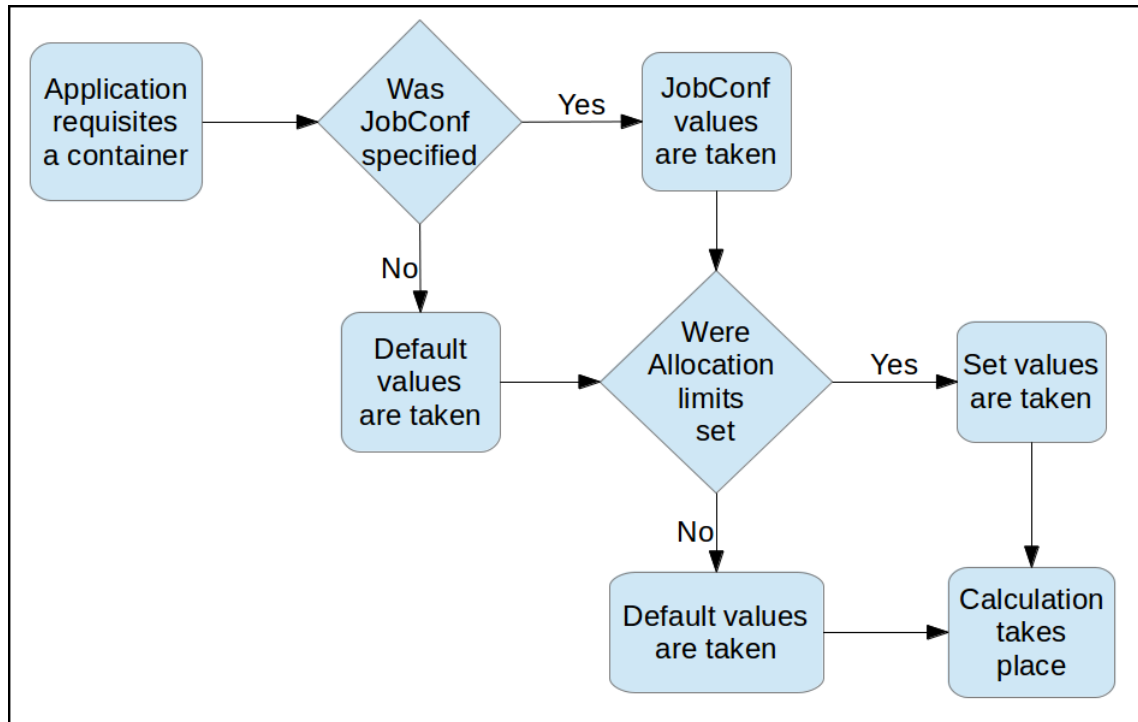
Figure 3.5 – Flow of operations for resource granting

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.

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
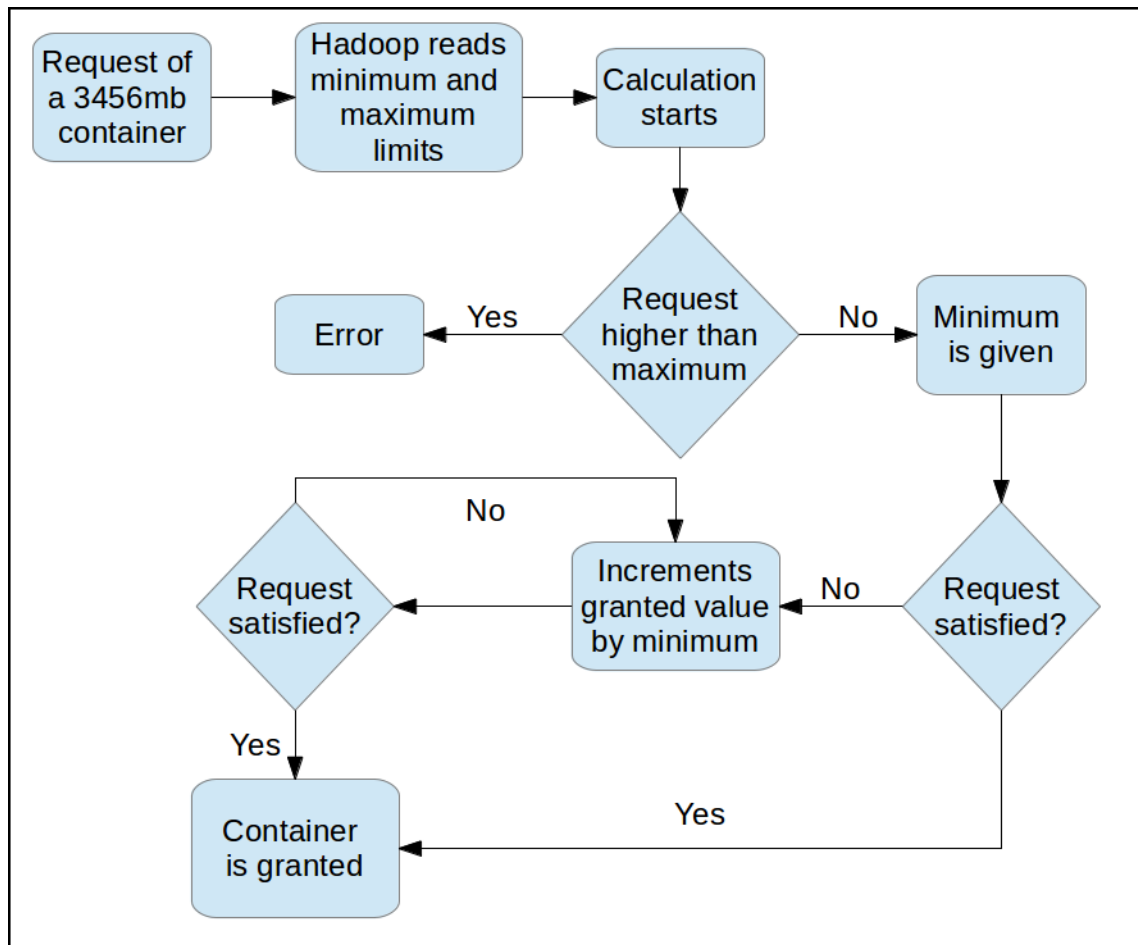
Figure 3.6 – Flow of operations for resource granting

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 opportu-

nity 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 **??**. 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 2.1 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 OperatingSystemMX-Bean, 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 PhysicalMemoryColletor 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.
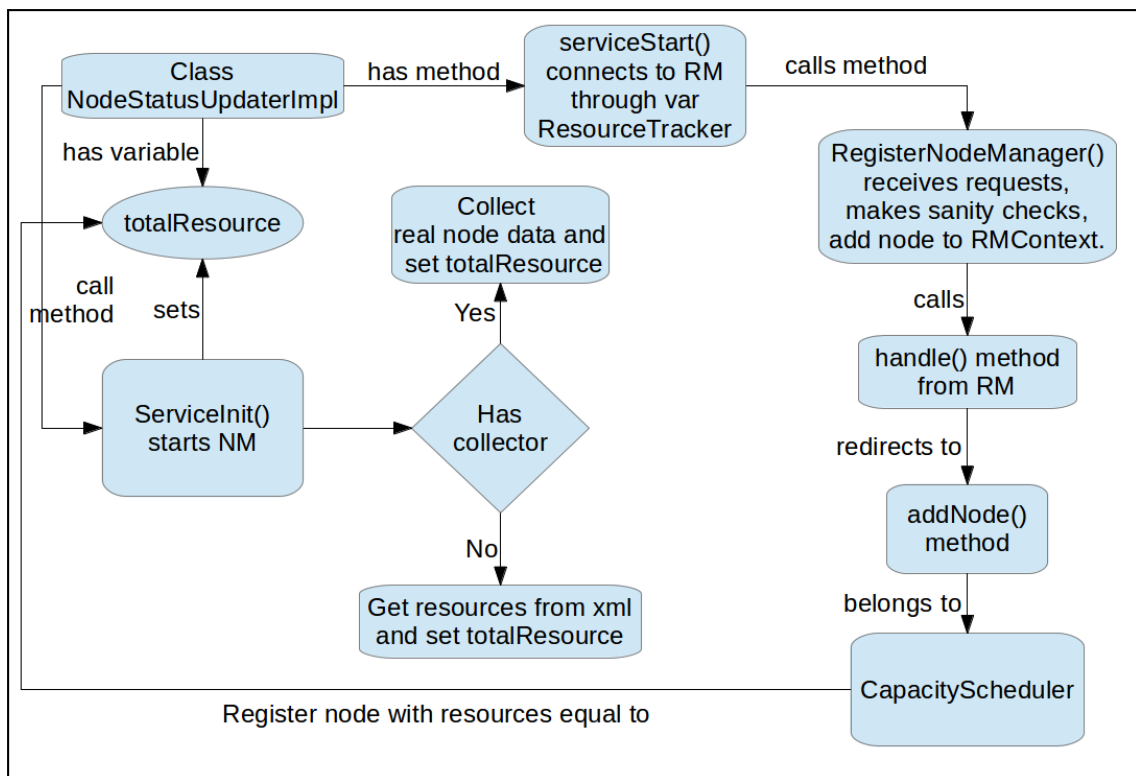
Figure 3.7 – Flow of operations for resource granting

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.
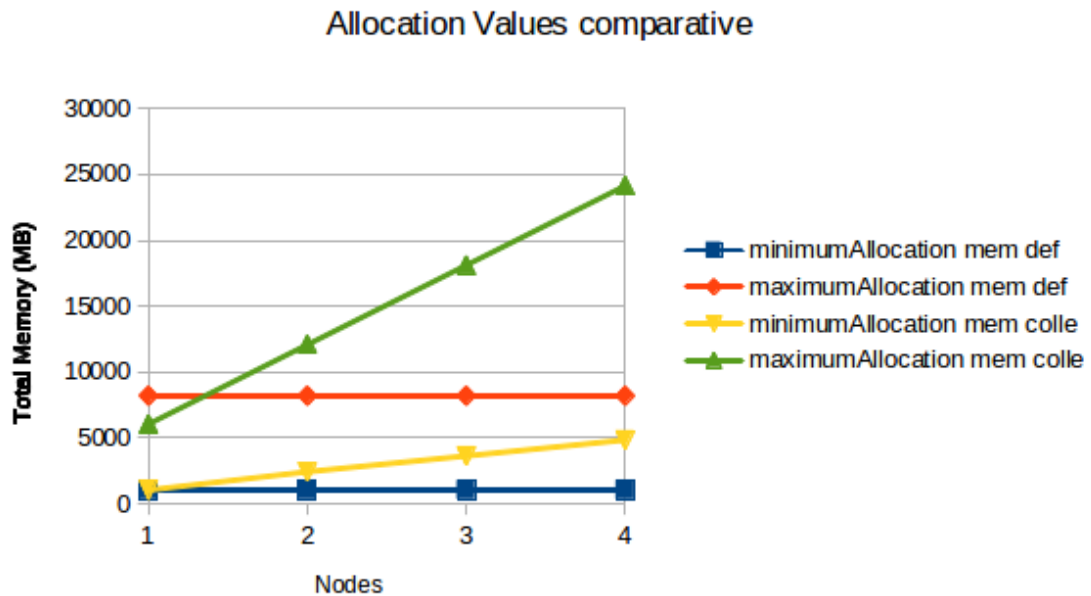
Figure 4.1 – Cluster memory available on default and collector implementation

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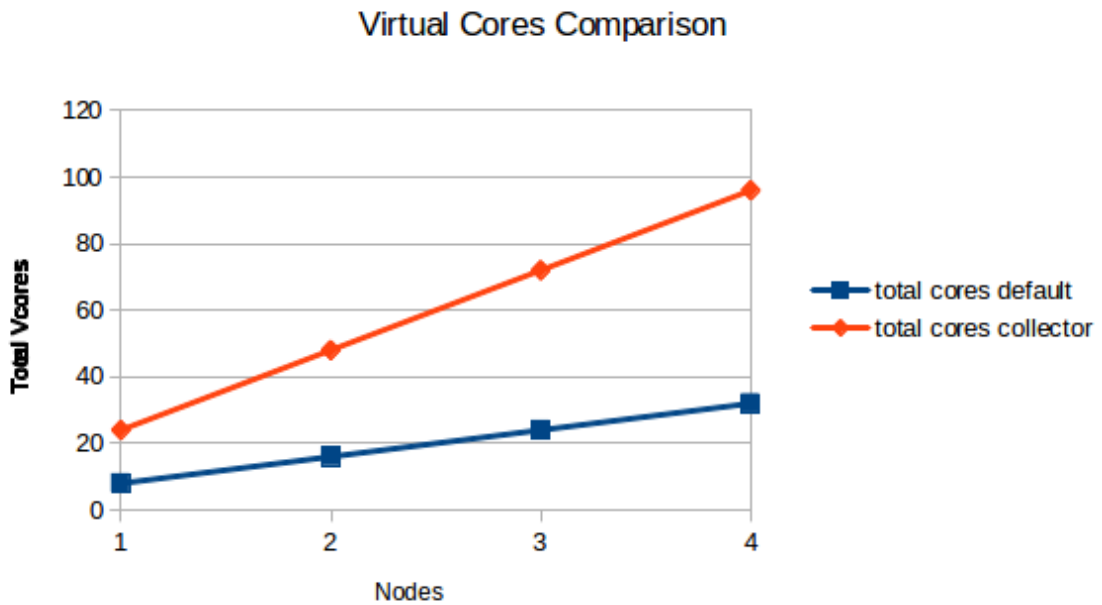e CapacityScheduler. The experiment consisted in executing a TeraSort in the cluster with the original CapacityScheduler and the context-aware CapacityScheduler. This was made in order to compare how the higher resource availability and higher allocation limits impacted the scheduling.

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 portraits 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.
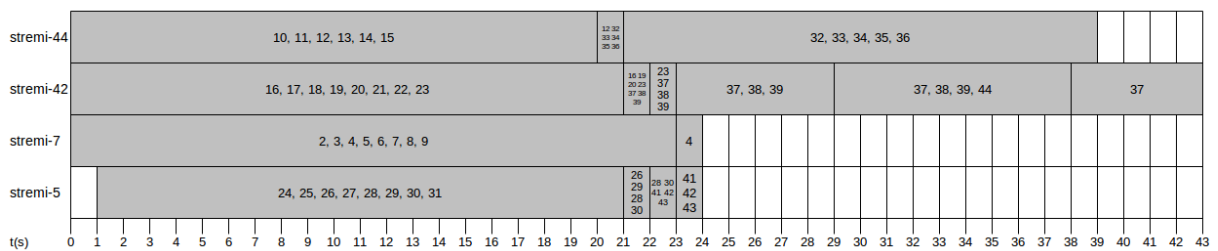

Figure 4.3 – Container assignment with default configuration

In order to illustrate how to interpret the Gantt charts, the node stremi-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 portraits 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.
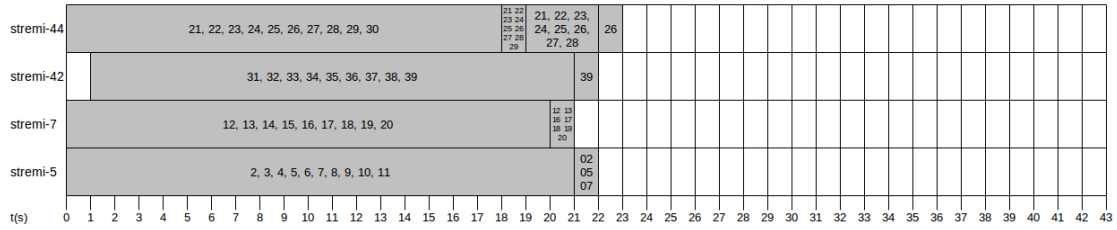
Figure 4.4 – Container assignment with the improved configuration

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 to 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 would 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 this 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 portraits 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.
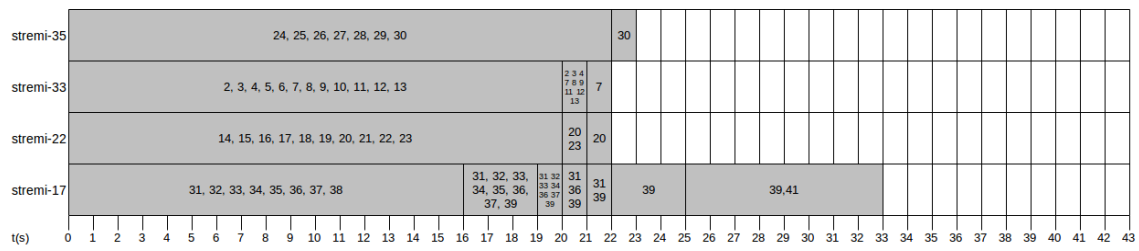
Figure 4.5 – Container assignment with the simulated heterogeneous environment

It is possible to note that the containers started the assignment with the node stremi-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 schedules 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.

REFERÊNCIAS

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).

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. In: CONFERENCE ON SYMPOSIUM ON OPEARTING SYSTEMS DESIGN & IMPLEMENTATION - VOLUME 6, 6., Berkeley, CA, USA. **Proceedings...** USENIX Association, 2004. p.10–10. (OSDI'04).

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.

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.

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.

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

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 he-terogeneous 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).

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.

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).

APÊNDICES

APÊNDICE 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.
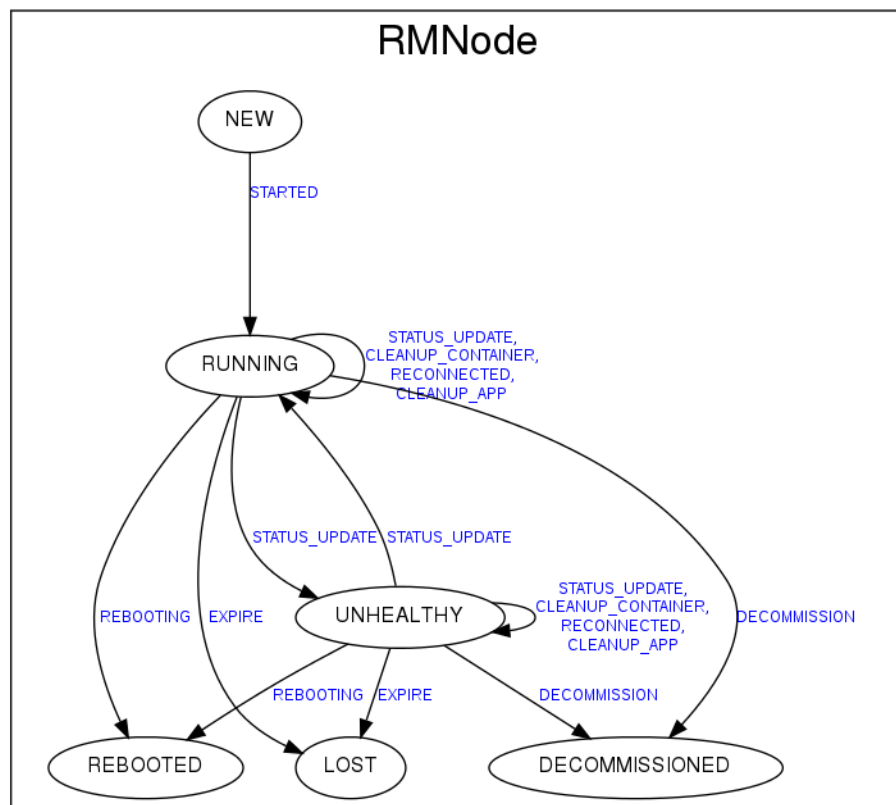

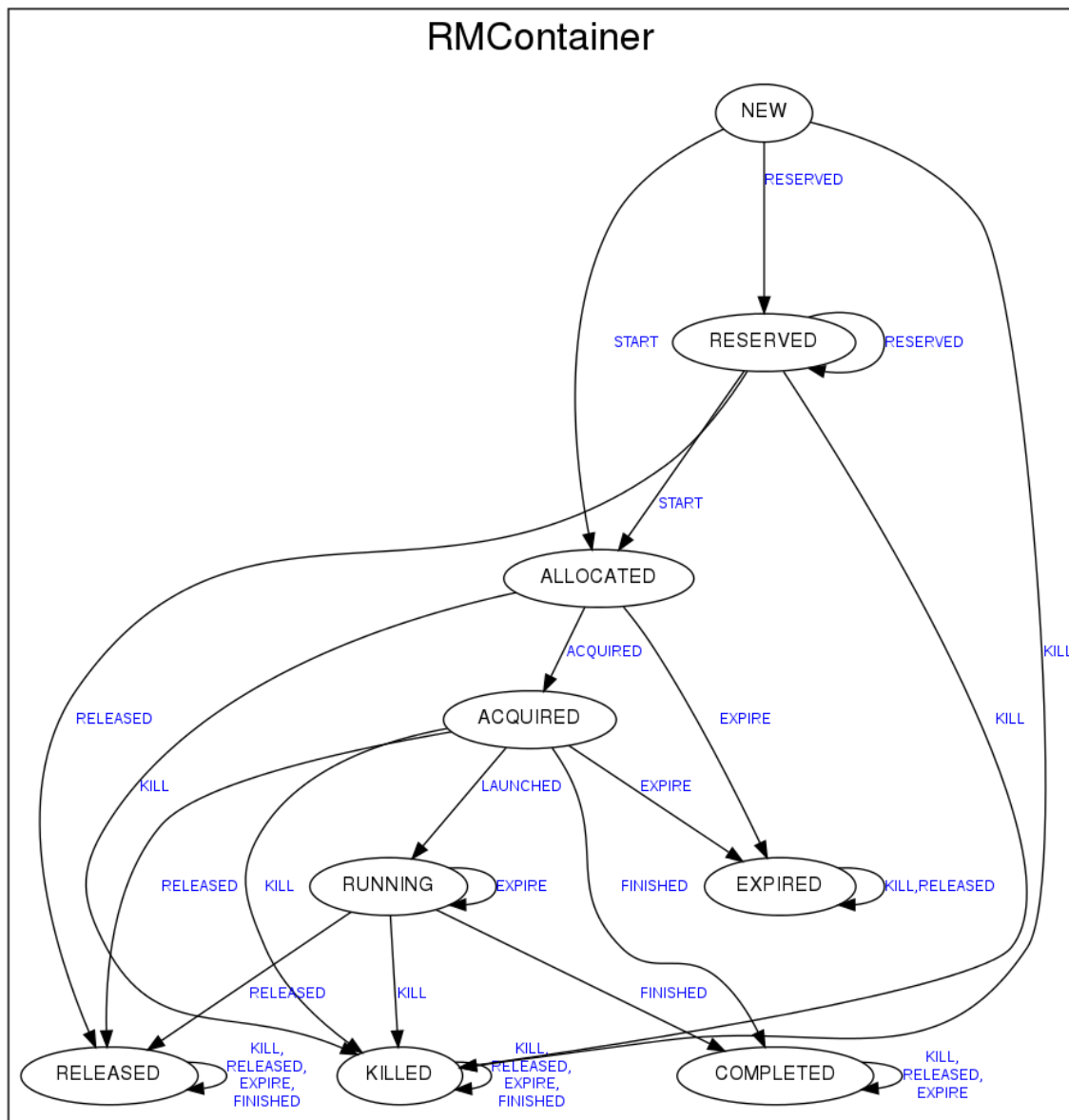
Figure A.1 – RMNode's state machine
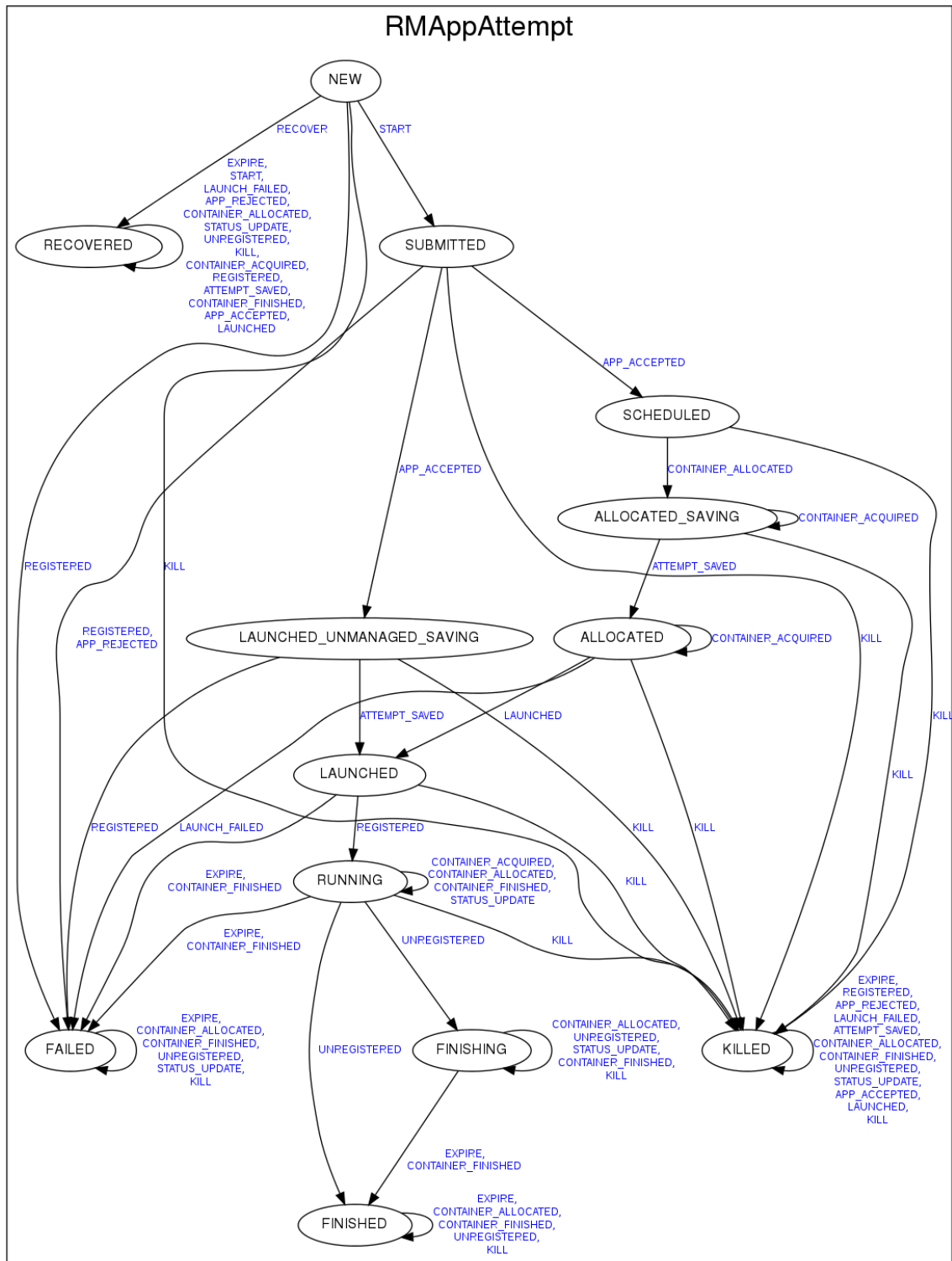
Figure A.2 – RMContainer's state machine
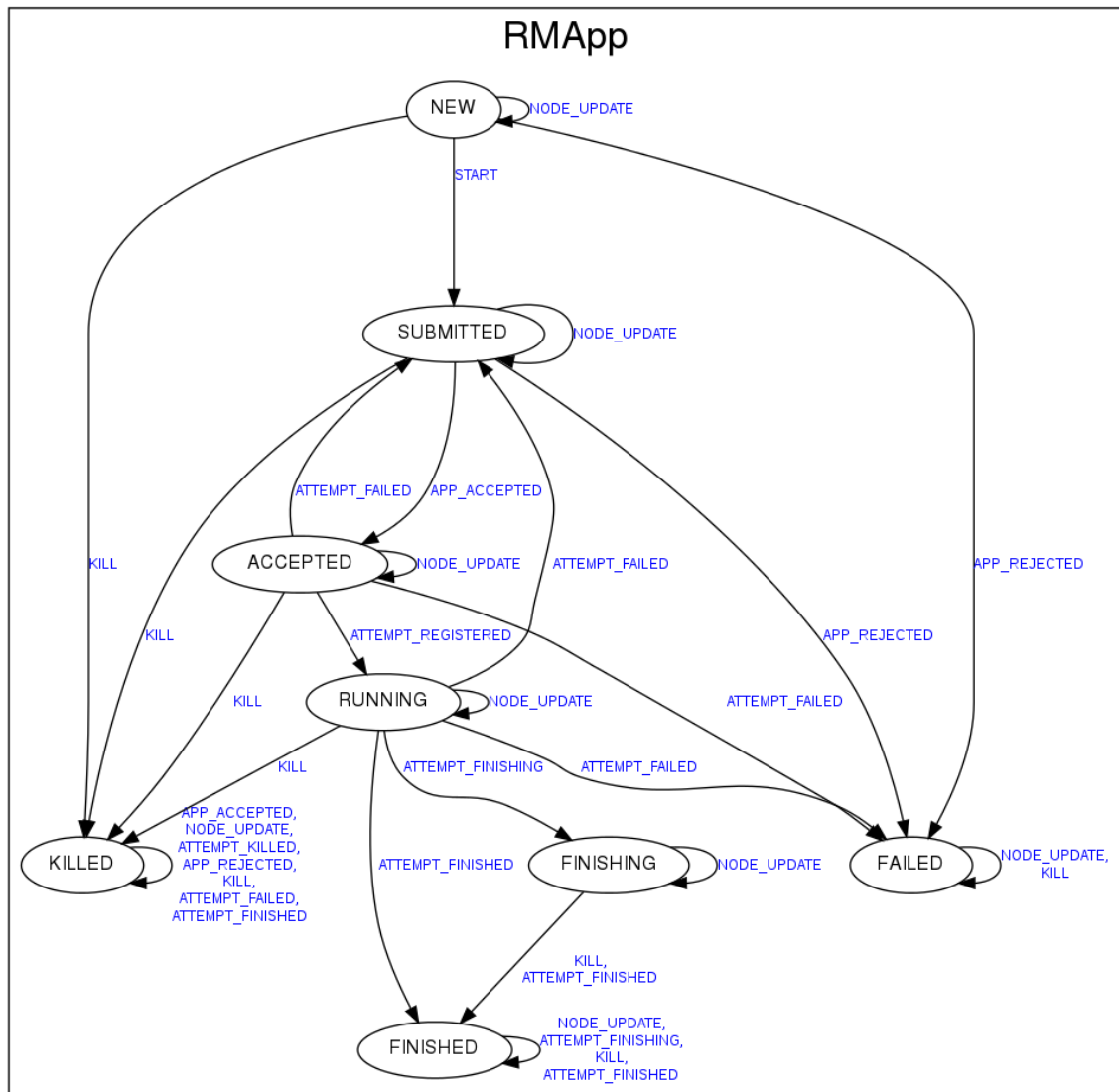
Figure A.3 – RMAppAttempt's state machine

Figure A.4 – RMApp's state machine

APÊNDICE 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 **??**, 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 it's 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.

APÊNDICE 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.

APÊNDICE 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>
```

APÊNDICE 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.

```
protected void serviceInit(Configuration conf) throws Exception {
    int memoryMb =
        conf.getInt(
            YarnConfiguration.NM_PMEM_MB, YarnConfiguration.
                DEFAULT_NM_PMEM_MB);
    float vMemToPMem =
        conf.getFloat(
            YarnConfiguration.NM_VMEM_PMEM_RATIO,
            YarnConfiguration.DEFAULT_NM_VMEM_PMEM_RATIO);
    int virtualMemoryMb = (int)Math.ceil(memoryMb * vMemToPMem);

    int virtualCores =
        conf.getInt(
            YarnConfiguration.NM_VCORES, YarnConfiguration.
                DEFAULT_NM_VCORES);

    this.totalResource = recordFactory.newRecordInstance(Resource.class);

    this.totalResource.setMemory(memoryMb);
    this.totalResource.setVirtualCores(virtualCores);
        metrics.addResource(totalResource);
    this.tokenKeepAliveEnabled = isTokenKeepAliveEnabled(conf);
    this.tokenRemovalDelayMs =
        conf.getInt(YarnConfiguration.RM_NM_EXPIRY_INTERVAL_MS,
            YarnConfiguration.DEFAULT_RM_NM_EXPIRY_INTERVAL_MS);

    // Default duration to track stopped containers on nodemanager is 10Min
        .
    // This should not be assigned very large value as it will remember all
        the
    // containers stopped during that time.
    durationToTrackStoppedContainers =
        conf.getLong(YARN_NODEMANAGER_DURATION_TO_TRACK_STOPPED_CONTAINERS,
            600000);
    if (durationToTrackStoppedContainers < 0) {
      String message = "Invalid configuration for "
        + YARN_NODEMANAGER_DURATION_TO_TRACK_STOPPED_CONTAINERS + " default
            "
            + "value is 10Min(600000).";
      LOG.error(message);
      throw new YarnException(message);
    }
    if (LOG.isDebugEnabled()) {
```

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

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
             PhysicalMemoryCollector();
3     TotalProcessorsCollector processorsCollector = new
           TotalProcessorsCollector();
4
5     int memoryMb = (int)(float)memoryCollector.collect().get(0)/1024;
6     int virtualCores = (int)(double)processorsCollector.collect().get(0);
7
8     this.totalResource = recordFactory.newRecordInstance(Resource.class);
```

The other change that had a strong impact in CapacityScheduler behavior was the inser-
tion 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(
           nodeManager,
3         usePortForNodeName));
4      Resources.addTo(clusterResource, nodeManager.getTotalCapability());
5      root.updateClusterResource(clusterResource);
6      ++numNodeManagers;
7    }
```

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(
           nodeManager,
3         usePortForNodeName));
4      Resource oldCap = Resources.clone(clusterResource);
5      Resources.addTo(clusterResource, nodeManager.getTotalCapability());
6      root.updateClusterResource(clusterResource);
7      ++numNodeManagers;
8      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.␣8GB␣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);
```