A simulator for Spark scheduler

4TU.School for Technological Design

STAN ACKERMANS INSTITUTE

A simulator for Spark scheduler

Sarwan Dar

September **2016**

TU Delft
Delft University of Technology

TU/e Technische Universiteit Eindhoven University of Technology

UNIVERSITY OF TWENTE.

WAGENINGEN UR
For quality of life

A simulator for Spark scheduler

Sarwan Dar

September **2016**

A Simulator for Spark Scheduler

Eindhoven University of Technology
Stan Ackermans Institute / Software Technology

Partners

System Architecture and Networking Group (SAN) TU/e

Steering Group
PROJECT OWNER: Dr. Rudolf Mak
PROJECT MANAGER: Aleksandra Kuzmanovska
TU/e Supervisor: Dr. Ad Aerts

Date September 2016

Document Status Public

The design described in this report has been carried out in accordance with the Tu/e code of scientific conduct.

Contact Address	Eindhoven University of Technology Department of Mathematics and Computer Science MF 7.090, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands +31402474334
Published by	Eindhoven University of Technology Stan Ackermans Institute
Printed by	Eindhoven University of Technology *UniversiteitsDrukkerij*
SAI Number	Eindverslagen Stan Ackermans Instituut ; 2016/037
Abstract	Spark is a tool that is used for analyzing large amounts of data in parallel. Performing experiments on Spark is both time consuming and expensive. Furthermore, an application can be launched on Spark with several different configurations. Therefore, a simulator is required to simulate the functionality of Spark. This report presents a project conducted to create one such simulator. A design of the Spark simulator was developed, along with a working prototype. The simulator allows the user to create a representation of physical machines, components of spark and simulates the scheduling of applications on Spark. The simulator also supports several configurations. The project report also lists suggestions for future improvements.
Keywords	Spark, PDENG, Software Technology, Simulation, Software Design, Big data
Preferred reference	A Simulator for Spark Scheduler, SAI Technical Report, September 2016. (Eindverslagen Stan Ackermans Instituut ; 2016/037)

Partnership	This project was supported by Eindhoven University of Technology.
Disclaimer Endorsement	Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the Eindhoven University of Technology. The views and opinions of authors expressed herein do not necessarily state or reflect those of the Eindhoven University of Technology, and shall not be used for advertising or product endorsement purposes.
Disclaimer Liability	While every effort will be made to ensure that the information contained within this report is accurate and up to date, Eindhoven University of Technology makes no warranty, representation or undertaking whether expressed or implied, nor does it assume any legal liability, whether direct or indirect, or responsibility for the accuracy, completeness, or usefulness of any information.
Trademarks	Product and company names mentioned herein may be trademarks and/or service marks of their respective owners. We use these names without any particular endorsement or with the intent to infringe the copyright of the respective owners.
Copyright	Copyright © 2016. Eindhoven University of Technology. All rights reserved. No part of the material protected by this copyright notice may be reproduced, modified, or redistributed in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the Eindhoven University of Technology.

Foreword

Nowadays, the analysis of vast amounts of data is an important activity both for commercial enterprises and in scientific research. For this activity a variety of frameworks is available, such as Hadoop, Spark, Naiad and many more, that allow their users to perform these analyses on computational resources that are often situated in the cloud. Such resources have to be managed carefully. From the perspective of the resource users, i.e. the data analysts, to obtain fast turn-around time and a low cost for leasing, and from the perspective of the resource providers, i.e. cloud providers and data center owners, to obtain efficient usage of their resource pools, so that the maximum number of customers can be served from the same pool.

The System Architecture and Networking group (SAN) of the TU/e is doing research into resource management techniques. In particular, one of our PhD-students, Aleksandra Kuzmanovska is looking into resource management techniques for running multiple frameworks on one or more clusters within large data centers. In that context, she experiments with novel resource allocation and scheduling policies. Her research would greatly benefit from framework simulators that can accurately determine the resources they use under her policies, thereby reducing the need to actually perform experiments on real machines. For Hadoop such simulators exist, but for Spark there is none that serves her purpose.

It has been Sarwan's assignment to develop a Spark simulator with the desired properties. For this he had to build on previous work by another trainee of the SE PDeng-program. As it turned out, the desired integration with that project required a major refactoring of its software which slowed down the actual development of the simulator. As a consequence, Sarwan has, in mutual agreement with Aleksandra, directed his efforts to implementing only a subset of the simulator's behavior. As a result we now have a simulator that is capable of simulating scheduling between applications with reasonable accuracy on clusters whose hardware has been appropriately described, but not the scheduling jobs and tasks within an application. On the bright side, however, it can be mentioned that due to the refactoring effort, the latter and other extensions will be easy to implement.

Part of the requirement for the simulator was that it would accept input that resembles that of a real Spark run as closely as possible, and also that it would log and report events in the same manner as Spark does. To that end, Sarwan had to study the source code of Spark in detail, which is not an easy thing, given the lack of proper documentation and the fact that it is a very compact code written in Scala, a high level programming language that was new to Sarwan. He has managed to do so independently and successfully.

To conclude, we are looking forward to putting Sarwan's simulator into use to further our research in cluster resource management.

Rudolf Mak

September 12, 2016

Preface

The report was written in partial fulfillment of the PDEng Software Technology program at the Eindhoven University of Technology. The report provides a detailed account of the final project carried out by the author over a period of nine months. The project was carried out at the Eindhoven University of Technology and focused on simulating the behavior of the Spark scheduler.

The project was done in order to evaluate the design skills of the author. Therefore, this report contains a description of the problem, the domain, the design and an overview of the implementation, among other things.

Sarwan Dar
September 2016

Acknowledgements

The project could not have been done without the help of a lot of people. Therefore, I would like to thank all of the people who were involved with my project.

I would like to thank all of my Supervisors who helped me during the project. I would like to thank Dr. Rudolf Mak, Dr. Ad Aerts and Aleksandra Kuzmanovska for their continuous feedback and support during my project. Their support not only helped me with the technical aspects of the project but also the non-technical issues.

I am grateful to the management team of the Software Technology program, Dr. Ad Aerts and Dr. Johan Lukkien, for giving me an opportunity to be a part of this program. I am also thankful to both of the secretaries Maggy de Wert and Desiree van Oorschot for assisting me with my problems during the program.

I would like to thank my parents and my siblings in Pakistan for their continuous support. I am very thankful to my mother Rizwana and all she has done for me.

Finally, I would like to thank all of my friends in the Software Technology program for collaborating with me during the program and giving me some very nice memories.

September 2016

Executive Summary

Apache Spark is a framework that is used for analyzing large amounts of data using clusters of computers. Spark can have a lot of different configurations and sometimes it is not so obvious which configuration would yield the best results. Therefore, certain experiments need to be performed which can help in determining the best configuration with the given workload. The problem with performing experiment on real Spark is that one has to pay to get clusters to experiment on and may have to wait if the machines are not available. Sometimes, it may not even be possible to get the required amount of machines due to high demand of the machines and one may have to settle for less for their experiments. Some experiments also take a long time and could be running for weeks. This would make performing experiments on Spark expensive in terms of time, money and effort. Challenges like these make it hard to use real Spark for experiments and makes it more desirable to have a simulator for Spark. The simulator would allow the users to save time and money by only performing a selective number of experiments on the real Spark. This project was developed as result of this need. The project allows to simulate the behavior of the Spark scheduler and make it easier and faster to perform experiments.

The main output of this project is a simulator for the Spark scheduler. The simulator provides the ability to construct a virtual representation of cluster machines. The simulator makes it very easy to change the specifications of machines. The simulator allows to choose the number of cores and memory of the machines. Besides the physical machines, the simulator also constructs a representation of different Spark components such as a Master, workers and executors. These components are necessary to simulate application scheduling in Spark. The simulator has a representation of applications and allows scheduling of applications using different scheduling policies. The simulator currently uses the standalone cluster manager but is extensible and it can support more in the future.

The system explained in this report is based on a generic and extensible design that makes it easy to extend, maintain and change it. The need for this became evident during the implementation phase because there were a lot of changing requirements. The simulator reduces the time, costs and effort involved in doing experiments on real cluster machines by allowing users to carefully select the experiments that they would do on the real Spark.

Table of Contents

Contents

List of Figures

List of Tables

1.Introduction

In this chapter, we introduce the project context. Next, we discuss the goals of the project and give some background about our client. We end the chapter by giving an outline of the report. The aim of this project is to design a simulator for application scheduling in a data analytics framework called Spark. The simulator will allow us to analyze the performance of the existing Spark schedulers.

1.1 Context

The project titled “a Simulator for Spark” was designed and implemented by Sarwan Dar as part of his Professional Doctorate of Engineering (PDEng) in Software Technology program. The PDEng degree program is a part of the Mathematics and Computer Science group of Eindhoven University of Technology in association with the 4TU School for technological design, Stan Ackermans Institute.

The PDEng program is a two year, third cycle (doctorate level) degree program that is focused on improving the technical and non-technical skills of the trainee as a Software Designer. The degree program is focused on the design and development of large scale software intensive systems.

The project was initiated by the System Architecture and Networking group (SAN) at the Eindhoven University of Technology. Details about SAN and other stakeholders can be found in the “Stakeholder Analysis” chapter of the report. The project description and context are given in the next sections.

1.2 Big data

The advancement in technology and means of communication is leading to large amounts of data production. The data amount produced by applications such as Facebook [1], WhatsApp [2] and Twitter [2] is growing every day. According to a research done by IBM [2], 90 percent of the world’s data has been produced in the last two years. The large amount of data is referred to as “big data”, which means large volumes of structured or unstructured data produced by an organization. The big data is analyzed to find hidden patterns, get useful information and sometimes even to predict the future. Analyzing large amounts of data is not possible using a single computer system as the data is usually too large and requires to be split among multiple machines. So, parallel processing on clusters of machines is used to store and process data.

Clusters can also be used to analyze large amounts of data. A cluster is a collection of computers, connected together in a network, that work together to execute several tasks in parallel. Clusters allow to connect a large number of computers and use all of their processing power for solving a problem. Clusters also ensure fault tolerance because malfunctioning nodes can be simply replaced by other nodes.

1.3 Big data Frameworks

To analyze big data using clusters, we require the use of frameworks. I define the framework as a conceptual structure that allows the user to perform a specific task while minimizing the coding effort required by the user. A framework may also provide some services or a runtime environment. For example, in case of Spark, it provides a runtime environment to the users to run their applications and provides services such as resource allocation and application scheduling. A framework makes it easier to focus on the actual problem one is trying to solve rather than diving into hardware details. Frameworks help in carrying out the detailed tasks so the user does not need to worry about unnecessary details. Frameworks make development times shorter by providing reusable methods and the code produced by the frameworks is easy to maintain and change. A framework reduces the development time by providing commonly used components so the user can focus on the business rules.

There are many frameworks that allow us to use clusters to analyze large amounts of data in parallel. A few examples of such frameworks are Cloudera Impala [2], Hadoop [3], Mahout [4], Giraf [5] and Apache Hama [6]. Cloudera Impala is faster than all other frameworks when doing SQL based data analysis. Mahout is very efficient when used for machine learning. Giraf is used for graph processing and Apache Hama for bulk synchronous processing. The most popular framework among these is Hadoop, which is an implementation of Map-Reduce.

Map-Reduce is a programming model where a large amount of data is converted into smaller subsets of data, some processing is performed on the data and the results are combined in the end. Hadoop Map-Reduce works by creating key value pairs of data and then mapping, grouping the key-value pairs by a key and then reducing the group to a single value. Hadoop is the most popular implementation of Map-Reduce because it is very cheap to use. Hadoop is an open-source software and the cost per byte is really low. The Hadoop Distributed Files System (HDFS) [10] can deliver data to compute nodes at a very fast rate. Hadoop is designed to efficiently store and retrieve data.

Although Hadoop is the most used framework for big data analysis, it still cannot support all existing problems. Hadoop is the first implementation of the map reduce programming model and is an acyclic data flow model. Hadoop and most other popular frameworks are built on an acyclic data flow model [7]. Most popular applications, however, do not operate according to an acyclic data flow model. These applications, such as machine learning and interactive analysis, reuse data between multiple tasks running in parallel. Therefore, Hadoop is not suitable for such applications. To support these applications, a new framework called Spark [8] was introduced. Spark introduced a data abstraction called Resilient Distributed Datasets (RDD) [13]. An RDD is a read only set of data that is distributed among the clusters. An RDD provides the ability to cache data in memory that makes it easy to reuse data.

1.4 Spark

Spark is a fast framework for large scale data analytics. Spark can run up to 100 times faster than Hadoop in memory or up to 10 times faster on disk. Spark supports in-memory computation of large amounts of data and introduces an execution engine that supports a cyclic data flow model that is used by modern applications. Spark is very easy to use and provides methods that can help in making parallel applications. Spark can access data from a diverse range of data sources like H3, HDFS [10], and Cassandra [14].

Spark supports features like data streaming, SQL queries, machine learning and graph processing which makes it easy to program applications. Spark can hold data in memory by using RDDs rather than storing the results back to the disk. This can make the performance much better when one has to reuse the same dataset many times, like in many data mining applications or interactive analysis problems. A cluster manager is also needed which is responsible for scheduling individual applications on clusters.

Spark is a large scale distributed system and therefore it is difficult to predict the performance of a Spark application. To help predict the performance of a Spark application, we need to develop a discrete event simulator [15] that will allow people to simulate the behavior of Spark. The simulator will allow the users to predict the performance of Spark applications. A Simulator is also helpful to do a large number of experiments that would otherwise prove to be too expensive with real hardware.

The reason that we require a discrete event simulator is because we want to make calculations based on time without the time actually passing by. A discrete event simulation is a simulation of a real system that has a certain set of states which change with respect to time or due to the occurrence of some event. A discrete event simulation is usually used to study complex systems and to find improvement points in the system. An example of a discrete event simulator is the simulator for Hadoop, known as Mumak [8].

The goal of this project is to focus on the scheduling of applications in Spark. A Spark scheduler is used to allocate resources for computations. The simulator would be a discrete event simulator for the Spark scheduler. The simulator would deal with the deployment of multiple Spark instances and it would schedule according to the cluster manager specified in the Spark configuration file. There are three types of Cluster Managers in Spark:

1. Standalone
2. Mesos
3. Yarn

There are two types of resource allocation, static allocation and dynamic allocation. The simulator is also be used to compare to the actual schedulers inside Spark. Along with the scheduler simulator, we would also simulate the components of Spark that are necessary for running the scheduler. These components include the physical cluster, the application and application submission process and the Spark cluster.

Initially, the project started with a focus on scheduling across applications but the project was changed in the last months to also include scheduling within an application.

1.5 Project Goals

The main goals of this project is to create a Simulator for Spark. The simulator is further divided into different parts:

- The design and implementation of main components of Spark necessary for deployment of Spark.
- The simulation of the physical machines on which Spark is deployed on.
- Development of an application model for the simulator.
- Provide scheduling across applications.
- Validation of the simulator performance by comparison with Spark. Due to the lack of time, the accuracy percentage for the simulator was not defined.

1.6 Outline

The project report is organized as follows. It starts by explaining the project context and the goals of the project. Chapter 2 discusses the stakeholders involved with the project. Chapter 3 discusses about the problem this project is trying to tackle. Chapter 4 discusses the domain of the project. Chapter 5 shows a feasibility study for the project. Chapter 6 discusses the requirements of the project. Chapter 7 discusses the design of the project. Chapter 8 discusses the implementation of the project. Chapter 8 discusses the verification and validation portions of the project. Chapter 10 presents a conclusion of the report. Chapter 11 discusses the project management techniques applied during the project. Chapter 12 presents the project retrospective.

2. Stakeholder Analysis

The previous chapter introduced the project and gave an overview of the problem. This chapter will introduce the stakeholders involved with the project and their interests. There are two main groups of stakeholders in the project, namely the Eindhoven University of Technology and the System Architecture and Networking Group. The following sections introduce the stakeholders in each group and their interests,

2.1 The System Architecture and Networking Group (SAN)

The customer of this project is the system architecture and networking group (SAN) which is a part of the Mathematics and Computer Science department at TU/e. The system architecture and networking grouping works in a wide variety of domains like distributed media systems, automotive systems, smart lighting systems and wireless sensor networks. The group specializes in parallel and distributed resource constrained embedded systems. Some of the research problems being tackled by the group are creation and management of distributed applications, and performing distributed resource management. Most of the research projects are done in close collaboration with the industry. More information about SAN can be found on their website [9].

One of the major focuses of the SAN group is resource scheduling in big data systems. The project was initiated by Aleksandra Kuzmanovska who is a researcher in the group and is doing research on the interaction between different frameworks like Spark and Hadoop in a cluster. This project will be used by Aleksandra in her research to perform the experiments which require Spark.

The first interest of the group is to save time, money and effort by using a simulator instead of the actual Spark. Performing experiments on Spark requires the use of expensive clusters, waiting for the availability of clusters and waiting for the completion of experiments. The simulator would not require the use of expensive clusters or waiting for anyone or anything.

The second interest of the group is to be able to extract useful information from the simulator by performing experiments with different configurations. A Spark application can be run with a number of different configurations. The performance of these applications depends on these configurations and can be better or worse depending on the configuration. The simulator would be used by the stakeholders to study the effects of different Spark configurations on the application performance.

Additional details about this can be found in the next chapter, Problem Analysis.

2.2 Eindhoven University of Technology

The project is carried out to fulfill the requirements of the PDEng Software Technology (ST) program. Therefore, the University is responsible for making sure that certain standards must be met by the project. The candidate can only get the PDEng degree once he meets the required standards. The University is mainly interested in the use of a good design process, a high quality report and the creation of a high quality design.

Another stakeholder from the Eindhoven University of Technology is Sarwan Dar, the PDEng trainee responsible for the design and implementation of this project. The main interests of this stakeholder is to create a high quality design, implementation and report, in order to successfully complete the degree program.

Table 1 Project Stakeholders

Stakeholder	Role	Interest
Dr. Rudolf Mak	Project Owner	Save time, money by simulation
Aleksandra Kuzmanovska	Project Manager	Perform experiments by simulation
Dr. Ad Aerts	Supervisor/Director ST	Good quality design, report
Sarwan Dar	PDEng trainee	PDEng degree

3.Problem Analysis

In the introduction chapter, we gave an overview of Spark and the problems associated with using Spark for experiments. This chapter gives an overview of the problem that we are trying to solve. First, an introduction to the problem is given, followed by why we face this problem, a description of an existing project related to Spark Simulation and the design opportunities encountered in the project.

3.1 Problem Analysis

To explain about the problem, we first need to define some important terms related to application scheduling. The definitions of different application times and the figure 1 are taken from [18]. The arrival time “ai” is the time at which an application becomes ready for execution and requests resources for its execution. The start time “si” is the time at which the application gets the resources required to execute and it starts executing. When an application is granted resources then it executes for the duration of its execution cost or turnaround time “Ci”. The execution cost of an application is defined as the time necessary to the processor for executing the application without interruption. However, an application may not always get its required resources and it may have to wait for some time (si-ai), which is known as the waiting time of the application. The time at which an application finishes its execution is known as its finishing time or turn-around time “fi”. The turn-around time of an application is the sum of the waiting time and the execution time.

When applications are executed on Spark then their performance can be expressed in terms of their turn-around time or finishing time. An application might be waiting for some data or a free resource. So, if the applications are not scheduled with an appropriate configuration then this might increase the total waiting time of the applications which would increase their finishing time. Scheduling an application efficiently means allocating resources in such a way that applications don’t have to wait for a long time.

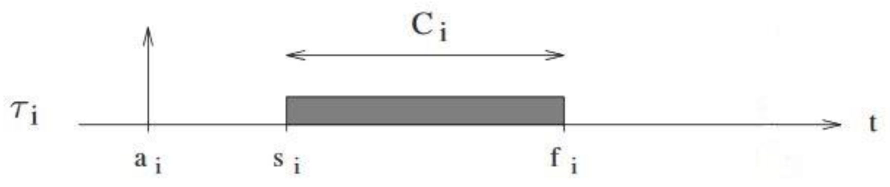


Figure 1 Timeline of an application

There are several problems with doing a large number of exploration or configuration experiments on real machines.

1). Experimenting on real machines is expensive because these machines cost a lot of money. A user has to pay a lot to get machines to experiment on and he might have to wait some time depending on the availability of the machines.

2). Even if someone can afford to run their experiments on the machines, they might not always be available. For example, all of the machines of a particular department might be occupied by other users and then they have to wait for the machines. This will prove to be unacceptable for people who are short on time and don't want to wait for the machines to be freed up.

3). One cannot change the specifications of the physical machines easily. For example, a user might want to run an experiment with machines having 8,16 or even 64 cores. However, machines with these specifications might not be available, temporarily or because they are not part of the hardware setup. When you are doing an experiment then you would like to try with multiple different configurations of the machine.

4). Running applications with the simulator would also be faster than running applications on real Spark. If the actual run time of an application is 2 hours then the user doesn't have to wait for 2 hours with the simulator. This is because the scheduler does not run in real time. The scheduler has a virtual clock that is updated based on events. This would allow the client to do a lot of configuration experiments in a relatively short time.

5). The user can try a lot of different configurations of Spark with the simulator. In real life this would be more difficult to accomplish. One of the reasons is that the deployment process is quite easy and much faster in the simulator. The user can change quickly and easily between different cluster managers and scheduling policies. The user can even easily add new scheduling policies.

3.2 Existing project to handle this problem

There is an existing project that mimics Spark [8]. The product produced by that project is only for scheduling tasks within an application. It doesn't support scheduling multiple applications at once. Therefore, the results from that project were not directly usable by this project. Some changes were made to that project to integrate some of the parts of both projects. The details about this are discussed in chapter 8 "Implementation".

3.3 Design opportunities

During the analysis of the Spark domain, a single design opportunity was identified. The design opportunity is extensibility. During the analysis phase and domain understanding phase, it was found out that Spark is highly configurable and it is possible to add a lot of new features and policies. This meant that a generic and extensible design would be needed for maintainability and extensibility purposes.

4.Domain Analysis

In the previous chapter, the problem analysis revealed the domain to be Spark. In this chapter, we explain the domain in detail to broaden the understanding of it. In this chapter, we discuss the Spark Framework, Spark components, Resilient Distributed Data Sets, parallel operations in Spark, Spark Deployment and Application scheduling in Spark.

4.1 Spark

Spark is a framework for parallel processing of large data sets. Spark is an open source framework that was originally developed by UC Berkeley but now it is an Apache project. Spark works by distributing the data over clusters of computers. Spark is an improvement over its predecessor Hadoop. Spark can perform up to 100 times faster than Hadoop because it offers in memory computations. Spark can also do computations on disk data when computations get too large. This is important because datasets are really large and it is a very time consuming operation to load the data in memory. Spark is also 10 times faster than Hadoop when on disk computations are done. Spark supports a wide variety of data sources such as NoSQL Databases [19], Hadoop Distributed File Systems and others. The performance of Spark and the fact that it can be used with multiple different data sources indicates that Spark has potential to become the best big data analytics tool. Spark has official support for multiple programming languages such as Java, Python and Scala. This allows one to write programs in different programming languages.

4.2 Spark Architecture

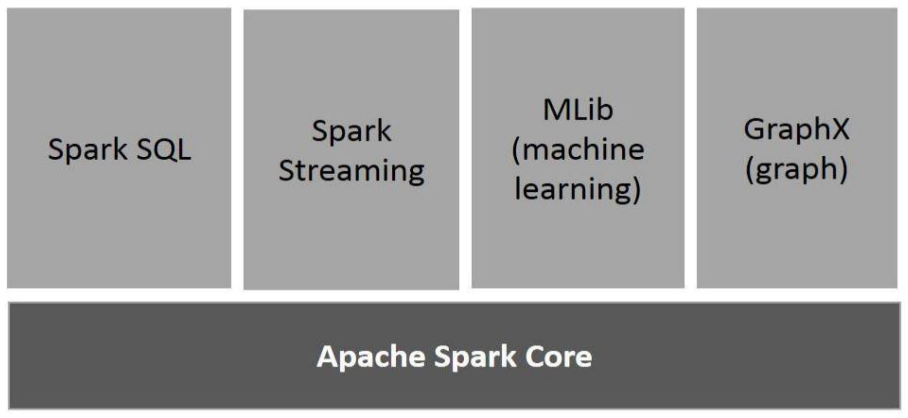


Figure 2 Spark Architecture

The Spark architecture is a layered architecture. The main component of the Spark Architecture is the Spark Core. Spark core is the execution engine of Spark and all of the functionality is built on top of this component. This component provides the capabilities such as in-memory calculation and external data referencing. The Spark core allows one to perform transformations on RDDs and to use libraries such as MLlib. The Spark core is also responsible for dispatching tasks, scheduling tasks and other input/output functionalities. It also allows to connect to different cluster managers such as
Standalone, Yarn or Mesos. Besides the Spark core, there are other components in the architecture. These components are explained below.

4.2.1. Spark SQL

Spark SQL is the component that provides support for structured and unstructured data for Spark. This library can be used to query data using the Structured Query language (Sql) or even the Hive Query Language (Hql) [20].

4.2.2. Spark Streaming

Spark streaming provides support for streaming data analytics. Streaming data analytics means fetching data in small groups and performing transformations on that data. This makes real time analysis of streams of data quite convenient.

4.2.3. Machine Learning Library

Spark has a machine learning framework known as MLlib [21]. The machine learning library comes with a selection of various algorithms such as Clustering, Filtering, Classification, Regression and many others.

4.2.4. GraphX

Spark has a distributed graph processing framework known as GraphX [22]. It allows the user to specify user-defined graphs. GraphX offers a combination of extraction transformation loading (ETL), exploratory analysis and iterative graph computation within a single system.

4.3 Resilient Distributed Data Sets

Spark introduces the concept of Resilient Distributed Datasets (RDDs) , which is an immutable collection of data that is distributed over the clusters. A single RDD can be divided into several logical partitions. An RDD can contain any type of objects from the languages Python, Java or Scala. This also includes user defined classes. The data inside an RDD can come from any place such as a text file, json file, CSV file, a database or even another RDD.

An RDD is a read-only collection of data and it can be created by performing operations on data or other RDDs. Once an RDD has been created then it cannot be modified. Performing an operation on an RDD results in the creation of a new RDD and the original RDD is not modified. An RDD allows the user to persist intermediate results in memory. RDDs provide automatic recovery in case of a failure. As RDDs are created after a sequence of transformations, these transformations are also stored as a sort of a graph. This graph is known as a lineage graph and allows the recovery of data in case of a failure. RDDs also have a lazy evaluation which saves a lot of memory and processing power. Lazy evaluation means that an expression is not evaluated till it is needed.

For Example:

```
val FileName=sc.textFile("FileName.txt");
```

The line mentioned above does nothing but create an RDD that says we will need to load this file. The RDD does not actually load the file at this moment.

The elements inside an RDD can be operated on in parallel. RDDS allow Spark to perform better than Hadoop. This is because RDDs make MapReduce operations faster due to in-memory data usage. The lack of multiple Input/Output operations makes RDDs much faster than a data format in other frameworks.

4.4 *Spark Deployment*

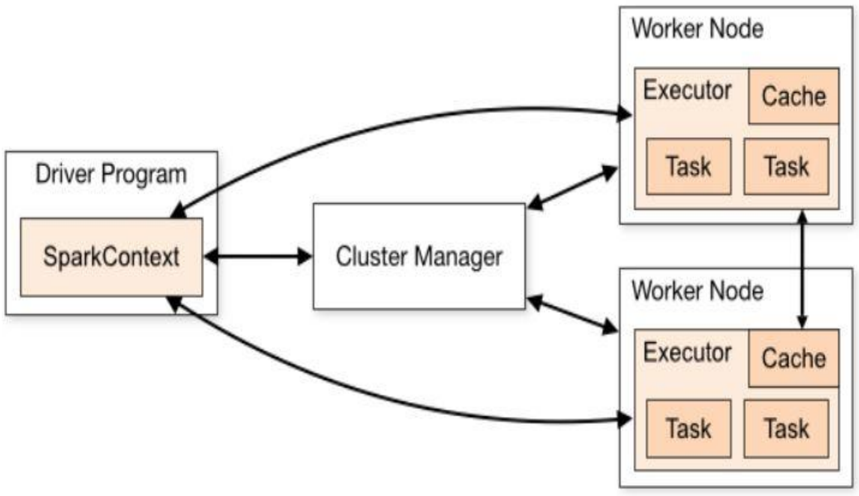


Figure 3 Spark Deployment

In this section, we explain about what is meant by a "Spark Deployment". A Spark Deployment is actually the creation of a set of processes known as workers and a single process, known as a Master of these workers, and their assignment to a computer. Spark can be deployed on a single computer or multiple computers connected by a network. The Cluster manager decides how resources are allocated across applications. Spark supports three types of Cluster managers:

1. StandAlone
2. YARN
3. MESOS

In this section, we will only explain about how deployment is done in Standalone mode.

In the beginning, we have to reserve a certain number of nodes in the cluster. A "node" in this case means a single computer. We need to reserve nodes because a cluster is usually being shared by a lot of people. So, this is done in order to have a dedicated set of resources for the duration of our session. A node can be reserved for a certain amount of time and the reservation is cancelled after that time expires.
After reserving nodes on the cluster, we start a Master process on one of those nodes. A Master process can be considered as a resource manager for applications. The Master process is the Cluster Manager in standalone mode. A Master process has information about which resources are available and how many cores are available. When an application arrives in the system, it will request the Master process for a certain number of cores. The Master will check if there are enough resources available and then allocate those resources to the application. After the Master process has

been started, it prints out an address which can be used to connect it to the Worker processes.

Next, we have to start a worker process on each remaining node. A Worker process is where the application gets resources to perform calculations and store its data. The node where the worker process resides is known as the Worker node. After a worker has been started then it must be connected to a Master process. So, when an application requests a Master for a certain number of cores then the Master will check whether the workers can satisfy the resources requirements of the application.

If the workers can satisfy the resource requirements, then the Master starts a Java Virtual Machine (JVM) on each Worker node. This JVM is known as an executor and it is specific to a single application. An executor cannot be shared between two applications and it only contains tasks and data for that application. A Worker on the other hand, can be shared between multiple applications. The executor will not take all of the resources of the worker node but it will only take the resources allocated by the Master to it. An executor stays alive as long as the application is alive.

Each application starts a driver process, which is the process running the main function of the application. The driver process is started on the Master node. So, when an application is submitted to the cluster then it will start a driver process on the Master node. The driver process will connect to the Master process to request resources for the application as explained earlier. Each driver program contains a Spark Context. The Spark Context has information about the connection to the cluster and it can be used to create RDDs on the cluster and send an application's tasks to its executors. The details about tasks (and other application components) and how these tasks are sent to the executors are explained in the next section.

4.5 Application Model

When a Spark application is launched then it is divided into a driver program and a set of independent processes called the worker nodes. The driver program is the main program of the application. The driver program is mapped to a driver process. The driver process converts the program into a set of Jobs, stages and tasks. Each application has only one driver and there is a one to one mapping between a driver and an application. The driver program is responsible for managing the graph of RDD transformations. It is also responsible for coordinating actions executed on the cluster. Each application has one driver which is responsible for dividing the application into different jobs.

Each Spark application is divided into a set of Jobs. Each Job is an action requested by the user. For example, "save" action in Spark is converted into a Job and this job is responsible for writing data to a file. A single Spark application can have many jobs. Each Job is further divided into stages that depend on each other. A stage can be considered as a computation that produces intermediate results. The results produced by the stage can be persisted on disk.

Each stage is composed of a set of tasks that are independent of each other. So all of these tasks can be executed in parallel. All tasks inside a stage are operating on a different set of data but they are performing the same calculations. A task is assigned to a task slot in the executors. The number of task slots in an executor is equal to the number of cores occupied by the executor.

For example:

Consider a Spark application that counts the number of Logs by their type. Let's say that we have the following log:

“Warning This is a warning”

“Error This is an error”

We use the following code [10] to perform count the number of logs of each type.

val input = sc.textFile("log.txt")
val splitedLines = input.map(line => line.split(" "))
val mappedRDD= splittedLines.map(words => (words(0), 1))
val shuffledRDD= mappedRDD.reduceByKey{(a,b) => a + b}

The sequence of commands will implicitly create a directed acyclic graph (DAG) of RDD objects. This DAG of RDDs is called as a lineage graph and will be used when an action is called.

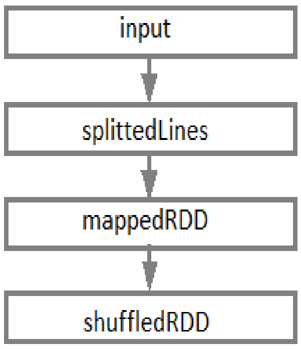


Figure 4 Directed Acyclic Graph of RDDs

The figure 4 shows a directed acyclic graph that is generated from our code. The DAG is generated by the application driver. When we use sc.textFile() function, a “input” RDD is created from the text file. This RDD is then mapped to the “splitted-Lines” RDD. No assignment actually happens here and the splittedLines RDD just keeps a reference to the input RDD. If we continue in the diagram, we can see that each map operation results in the creation of a new RDD and mapping it to its parent RDD. In the last step, we can see that a shuffled RDD is created. A Shuffled RDD is an RDD that is created by shuffling of data. Shuffling means that all with the same key should be on the same machine. The last operation “reduceByKey” is an action, as we discussed earlier for “save”, a job will be created corresponding to that action.

After the DAG has been created, it is sent to the DAG scheduler. The DAG scheduler is the entity that is responsible for dividing the DAG into stages. The division of the DAG into different stages depends on the type of transformations being done in the DAG and the dependencies between different transformations. There are two types of transformations, namely Narrow transformations and Wide transformations. Narrow transformations are those transformations that don't require the data to be shuffled across different partitions e.g. "map". Wide transformations are those transformations that require the data to be shuffled across partitions e.g. "reduceByKey". So, in this case, the narrow transformations will be grouped together in to a single stage and the wide transformation in another stage.

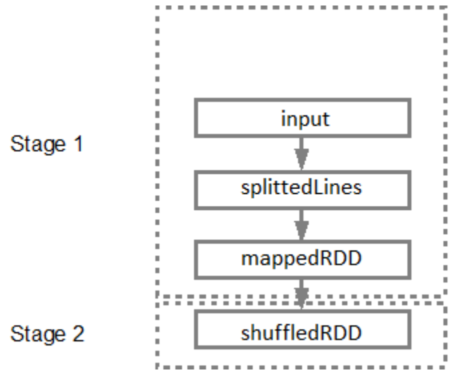


Figure 5 Conversion of DAG into stages

Figure 5 shows the mapping of DAG into different stages. Stage 1 contains all of the narrow transformations and Stage 2 contains the wide transformation. The DAG scheduler then submits these stages to the task scheduler. The DAG scheduler first divides the stages into a set of tasks and then sends this set of tasks to the task scheduler. So, let's say that in our case each stage can be divided in to four sets of tasks. These tasks can be executed in parallel. Figure 5 shows the conversion of stages to tasks.

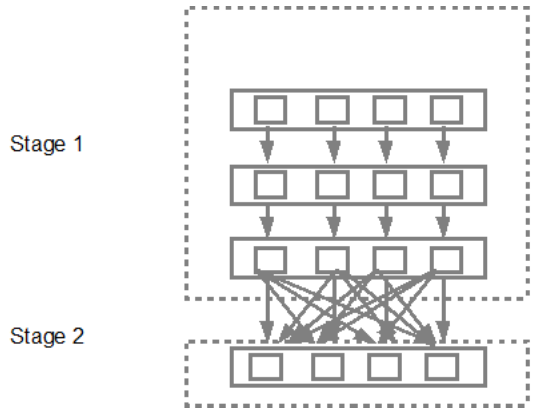


Figure 6 Conversion of Stages to tasks

4.6 *Scheduling Model*

The scheduling of applications is provided by the cluster manager that is running with Spark.

4.6.1. Static Resource allocation

The easiest option for resource allocation is static resource allocation. In static resource allocation, we can assign a fixed amount of resources to each application. The application will hold the resources for the entirety of its execution. The static resource allocation is supported by standalone, YARN and MESOS.

When an application is submitted in standalone mode then it will run in First In First Out (FIFO) mode. Each application will try to reserve all of the available nodes whether it needs them or not. It is possible to limit the maximum number of nodes an application will try to occupy by setting a parameter in the configuration file. It is also possible to limit the default maximum for all of the applications by changing a different parameter in the file. Each application's memory use can also be constrained by using a parameter.

When an application arrives in the system then it will request the Spark Master for a certain number of cores. The Master will check the amount of resources it has and if it has enough resources then it will ask each worker to start an executor with a specific number of cores. The number of executors created are dependent on the number of cores requested by the application. The newly created executors are assigned to the application.

4.6.2. Dynamic Resource allocation

In static resource allocation, the resources assigned to an application are fixed. This means that if an application has extra resources then it cannot give the resources to another application in need of resources. In dynamic resource allocation, an application can give back its resources to the cluster if they are not needed and it can demand more resources from the cluster when it has a heavy work load. This feature is useful if there are a large number of applications trying to share resources in the cluster. When a resource remains idle for some time then it can be returned back to the cluster.

Spark defines a set of heuristics that determine when an application can get more resources or when an application can release resources. When an application has a set of pending tasks and it does not have enough resources to execute the tasks then it requests the Master for more resources. The application requests resources in rounds and the request gets bigger after every interval. The request is first performed when an application has been idle for a time "t" that is defined in the configuration file and this continues every "t" seconds till the request has been satisfied. The amount of resources requested by the application also increases every "t" seconds. The resource removal policy on the other hand is quite simple. An application loses resources if the resource has been idle for more than a certain number of seconds defined in the configuration file. So, if an executor has been idle for a certain number of seconds then the application will request the Master to remove the executor. Due to shortage of time, the dynamic resource allocation was only partially completed.

4.6.3. Scheduling within an application

In the previous sections, we explained about scheduling across applications. In this section we will give a brief overview of how scheduling is done within an application. We will not go into details of this topic as this project is focused on scheduling across applications.

An application has multiple jobs with different arrival times. By default, jobs are scheduled in a First Come First Serve manner. Like we discussed in section 4.5, each job is composed of several stages and each stage has several tasks. The first job has the highest priority over all the resources while its stages still have tasks to launch, then the second job and so on. If one job does not need all of the resources then the next job can also start. A job can start if its resources requirements can be partially satisfied. All of the stages from each job are sent to the DAG scheduler. The DAG scheduler sends these stages to the task scheduler in a FCFS manner. These stages are sent as a collection of tasks known as task sets. The task scheduler schedules the tasks on the executors.

5.Feasibility Analysis

After doing the domain analysis in the previous chapter, a feasibility analysis is done in this chapter. The chapter covers the identified issues and the risks and the strategies used to avoid them.

5.1 Issues

5.1.1. Spark Domain

Understanding the complex Spark domain was a critical step in creating a design for the simulator. I had understanding of parallel programming concepts and big data but I had no idea of Spark. I had to spend a lot of time to understand and learn Spark and its concepts. There was also little to no documentation available for Spark. I also used the source code to understand more about Spark.

5.1.2. Existing Project

There was an existing product that the client wanted to be integrated with this project. There were some problems with the product. Firstly, the design was not scalable. The product had to be redesigned to be integratable to my new design by making it more generic and hence more extensible. Responsibilities also needed to be moved to the proper entities. More information about this can be found in section 8.4 of the "Implementation" chapter.

5.2 Risks

There were a series of risks identified in the project. The risks are explained below and also the strategies that were used to mitigate the risks.

5.2.1. Timeline

The PDENG projects have a fixed 9 month timeline. According to my initial estimation, the work load described in the beginning was more than the 9 month timeline. I tackled this risk by making a detailed planning for the project. Every few weeks I updated the plan of the project. I made a generic design so it was easy to add new components to the code. The design also made it easier if there were some changes.

5.2.2. Large Codebase

The Spark project has a very large codebase for a one man project. The entire project has more than a hundred and fifty thousand lines. Sometimes it was very hard to find something in code. The Spark codebase has meaningful names and code is arranged really nicely. The issue is that it is sometimes difficult to find something in the source code or to understand how a process works which creates the risk of lack of information or of misconceptions..

5.2.3. New technologies

There were several new technologies and concepts like Scala in the Spark project. It took me some time to grasp all the concepts. The entire code base was in Scala so understanding that was also challenging.

Conclusion

After doing the feasibility analysis, it was found that it is possible to complete the project within the given timeframe if an abstract representation of the Spark components is made. Further details about this can be found in the Requirements and the Project Management chapters.

6.System Requirements

In this chapter, the process used to obtain the requirements will be explained. After that, the requirements stated by the customers will be explained. Finally, the extended requirements identified by the author will be explained.

6.1 Process

The first process when gathering requirements is identifying which people use Spark, were already working on a similar project and/or would benefit by using the finished tool. These people comprise the stakeholders of the project. These stakeholders were then later divided into different groups based on their power and interest in the project during the stakeholder analysis. The list of these people served as the starting point of the requirements gathering process.

After dividing the stakeholders into groups, face to face meetings were scheduled with them. The frequency and length of these meetings depended on the amount of power/interest they had in the project.
These meetings had the following objectives:

1. To collect a list of basic requirements/goals for the project.
2. Provide the author with a better understanding of Spark and clear up any misconceptions about the project.
3. Verify the extended requirements generated by the author.

A List of basic requirements was created from these meetings. This list of requirements were later extended and revised several times in iterations because the project followed an agile development approach.

6.2 Functional requirements

This section discusses the basic customer requirements and goals for the project as explained by the customers during the meetings. The detailed Use Cases that encompass these requirements are presented in the next chapter.

6.2.1. Model the Hardware

The software system should have a representation of the physical hardware on which Spark is deployed. This means that the system must be able to create a representation of physical nodes with a certain number of cores and a certain amount of memory.

6.2.2. Simulate the components of Spark related to Spark Deployment

Spark is a very large system and capturing the entire functionality of Spark is neither possible in the given time frame nor required. Only the functionality necessary to support Spark deployment and application scheduling will be simulated. The required functionality is explained in the following sections.

6.2.2.1. Simulate the behavior of Spark Workers

The system should have a representation of Workers and it should be possible to "deploy" the workers on the hardware model. In the context of the simulator, deploying a worker means that the worker should be assigned to a hardware model.

6.2.2.2. Simulate the behavior of Spark Master

The system should have a representation of the Spark Master and it should be possible to "deploy" the Master on the hardware model. In the context of the simulator, deploying a Master means it should be assigned to a physical hardware model and it must be connected to all of the Workers.

6.2.2.3. Standalone Cluster Manager

The system should contain a standalone cluster manager. In the case of standalone mode, the Master acts as a cluster manager.

6.2.3. Application Scheduling

6.2.3.1. Application Model

The system should contain a representation of an application. An application model is necessary for simulating scheduling of applications. An application model will contain Execution time, priority, type and required resources. An application model will also contain Jobs, stages and tasks.

6.2.3.2. Executor Behavior

The system should contain a representation of an executor, which will be created on the worker and assigned to an application.

6.2.3.3. Scheduling across applications

The system should be able to simulate the behavior of application scheduler of Spark. The system should have a scheduler component which decides which application should be scheduled for execution. An application can be scheduled in the simulator if it can get the required resources. The system should also simulate the execution of an application. In the simulator, the "execution" of an application means that it holds resources for the duration of its defined execution time. It frees the resources after its execution time has expired.

6.2.3.4 Scheduling within an application

Dag Scheduler

The system should have a dag scheduler that schedules jobs of an application in a First Come First Serve manner. The first job gets priority over all the resources. In our simulator, “scheduling” of a job means that the job will wait till all the tasks in all of its stages have finished.

Task Scheduler

The system should support the task scheduler which is responsible for scheduling of tasks within the application. The tasks should be scheduled in such a way that the dependencies between stages are not violated and the priority of the jobs is not violated. “Scheduling” a task means that it will be assigned to a core and it will wait for a certain number of seconds generated by using a random number distribution. The task will free the core after its execution time is complete.

6.2.3.5. Scheduling policies

The system should support at least support the First Come First Serve scheduling policy.

6.2.3.6. Resource allocation policies

The system should support at least the static resource allocation policy. In static resource allocation, once resources have been assigned to an application they cannot be removed until the application finishes its execution and new resources cannot be added during its execution.

6.2.4. Validation

6.2.4.1 Application Generator

The system should support an application generation module whose purpose is to generate new applications. The application generation module will generate different types of applications based on the arrival rate. The arrival times of the applications will be generated by using an exponential distribution. The type of the application will be determined according to their probabilities. The execution costs of the applications would be determined by using a uniform distribution.

6.2.4.2 Validation with Spark

The system would be compared with the actual Spark system and the order of execution of applications should be same as the real Spark. We did not define an acceptable accuracy percentage of the simulator due to shortage of time while implementing.

6.2.5. Re-implement and Redesign existing project

The system should be integrated with another project that focuses on scheduling task within an application.

6.3 Non-functional requirements

In addition to the functional requirements, there are a certain number of Non-functional requirements that must be met by the system.

6.3.1. Performance

The number of applications created by the simulator should be equal to the maximum number of threads supported in Java. This is a requirement for the simulation library.

6.3.2. Configurability

The system should be configurable and components like cluster hardware, Deployment components and application should be configured by reading from files.

6.3.3. Extensibility

The system should be extendable to add new scheduling policies, multiple spark clusters and cluster managers.

7. System Architecture and Design

In this chapter, we begin by presenting a high level architecture of the system. Next, we explain the design of the system in terms of use cases, class diagrams, sequence diagrams and state machine diagrams.

7.1 Introduction

In this project we used the "4+1" view for architecture and design of the system. The "4+1" architecture [10] takes a single use case or user scenario and iterates it over all the 4 views. This provides integration of the 4 views. The 4 views of the 4+1 architecture are given below:

1. Logical view: Object oriented design and behavior diagrams such as a sequence diagram.
2. Process view: Deals with issues such as concurrency and multi -threading.
3. Development view: Shows the static organization of the software.
4. Deployment view: Presents a mapping of software to hardware.

The fifth view or the "+1" view is the user scenario, which is considered while making decisions for the rest of the 4 views and is iterated over all of the views. The fifth view is known as the scenario view and is presented through use cases.

In this chapter we will first present the use cases, and then illustrate the four other views for a particular use case.

7.2 *Architecture Overview*

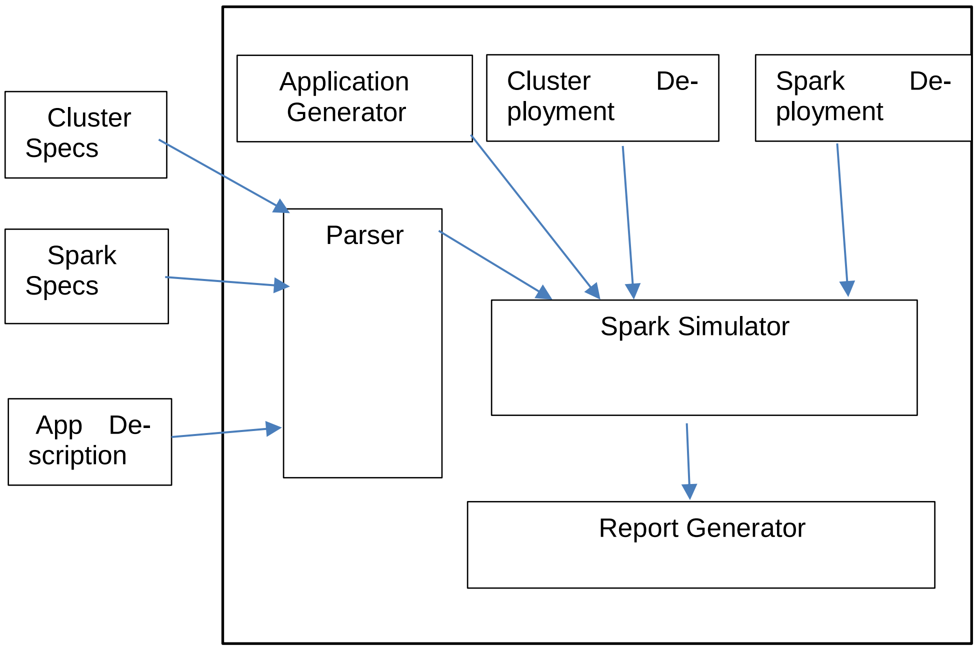


Figure 7 Architecture Overview

There are three different files that are used as input to the simulator. These files can be found in the Appendix. These files are shown in the diagram as “Cluster Specs”, “Spark Specs” and “App Description”. These files are read by the parser which passes the information to the simulator. Firstly, a representation of the physical cluster is created in the simulator. After a physical cluster has been created, a spark deployment is made on the physical cluster. Finally, a representation of the applications is created which is then fed to the Simulator. After all of these entities have been created then application scheduling is done according to the Spark heuristics and a report is generated by the simulator. These components are discussed in detail in the coming sections of this chapter and in chapter 8 “implementation.”

7.3 *Architecture Views*

As explained in the introduction section, we used the “4+1” view to obtain an architecture for the system. The different views of the architecture are explained in this section.

7.3.1. Use Case view

The user scenarios are presented by use cases in this section. A use case explains how the system and the user interact with each other. Four basic use cases were identified for the simulator. These four use cases are given below:

1. Create a virtual cluster.
2. Deploy Spark cluster.

3. Submit an application.
4. Schedule application.

The following is a system level use case diagram of the system. Each use case is explained in detail as well.

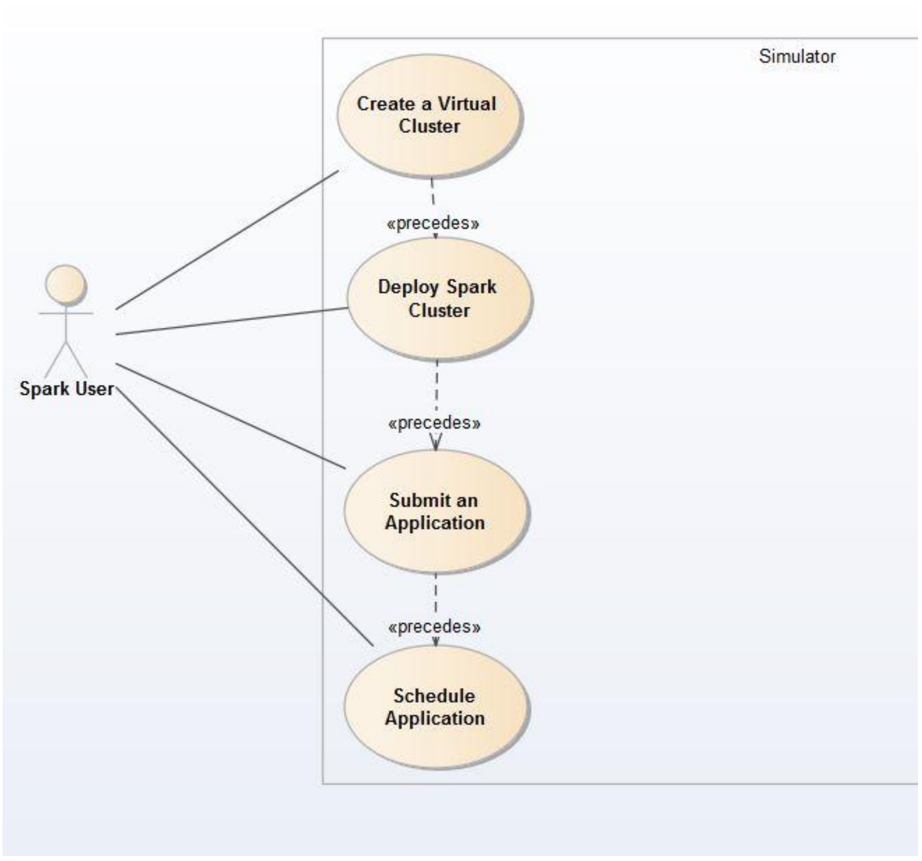


Figure 8 System Level Use Case Diagram of the simulator.

Use Case UC 1: Create a virtual Cluster

Description: A representation of the hardware platform is created in the simulator.

Pre Conditions: The required cluster specification is placed in the appropriate configuration file.

Actors: Spark user, System

Main Success Scenario:

1. **Spark user:** Place the cluster specifications such as number of nodes, processor and memory in a configuration file
2. **System:** Show the result of parsing the config file System: Show the result of the creation step
3. **System:** Show the result of setting the resource parameters
4. **System:** Show information to the user that the cluster creation was successful.

Post Conditions: A representation of a physical cluster is created in the system and a notification has been shown on the screen.

Use Case UC 2: Deploy Spark Cluster

Description: A representation of the spark components is created and they are mapped to the virtual cluster. This includes components such as a Master, workers and a cluster manager.

Pre Conditions: There should be enough physical nodes to accommodate the Spark instance to be deployed. Use case 1 should have been completed successfully as well.

Actors: Spark user, System

Main Success Scenario:

1. **Spark user:** Place the spark specifications such as number of workers in the configuration file.
2. **System:** Show that a Master and Worker objects have been created.
3. **System:** Show that the workers have been connected to the Master,
4. **System:** Show information to the user that the spark deployment was successful.

Post Conditions: Master and workers are created in the system, the workers are connected to the master and a notification is shown on the screen.

Use Case UC 3: Submit an application

Description: An application is submitted to the simulator in the form of a path to a configuration file. The configuration file contains information about the application. The arrival time and the type of application is generated by the application generator.

Pre Conditions: There should be enough worker nodes to satisfy the requirements of the application. Use case 1 and Use case 2 should have been completed successfully.

Actors: Spark user, System

Main Success Scenario:

1. **Spark user:** Place the application specifications such as the application name, priority, required resources and execution cost in the configuration file.
2. **System:** Generate an arrival time for the application using exponential distribution and show it on the screen.
3. **System:** Generate a type of the application using the uniform distribution and show it on the screen.
4. **System:** Load the application in the waiting applications queue.

Post Conditions: Application has been created and loaded into the waiting applications queue.

Use Case UC 4: Schedule the Applications

Description: An application is assigned resources by the scheduler and “executed” for a specific amount of time. In the simulator, “execution” means that an application holds resources for a certain amount of time and waits.

Pre-Conditions: Use cases 1,2 and 3 should have been completed.

Actors: Spark user, System

Main Success Scenario:

1. **System:** Check if there is an application waiting in the application queue.
2. **System:** If there is at least one application waiting in the queue then apply a scheduling algorithm and choose the next application to schedule.
3. **System:** Show the completed schedule and display starting times and ending times.

Post Conditions: An application has been scheduled and a notification is shown on the screen.

7.3.2. Logical view

The logical view is focused on the functionality required by the system. The logical view defines the components in the problem domain and the relationship of these components with each other. In this project, the logical view will be represented by a class diagram. The Process began by making an analysis model. This helped in getting a clearer understanding of the domain. The diagram began with all of the entities in the problem domain. This made sure that the project focused on only the required entities. This also ensured that the author had a correct understanding of the domain.

The Analysis model was slowly transformed into a model for the solution. The analysis model focused only on the entities in the problem domain but the Solution also focuses on entities required to develop a solution. The class diagram for the solution model also shows the relationship between these entities.

The Class diagram was designed with a focus on high cohesion and minimum coupling. The Design principles mentioned above made it easy to change and maintain the software. In addition to these design principles, some design patterns were also used. Some of those design patterns would be discussed in the upcoming sections. The use of these design principles made it easy to quickly respond to changing requirements. According to the feedback of the client, changes were implemented really quickly. It was easy to make changes because not a lot of classes had to be modified every time some requirements were changed. This also resulted in a stable design during the later phases of the project.

Before moving on to the next section, some important points have to be explained. While the principles of cohesion and coupling were followed, it was also important to simulate the actual behavior of the real spark. So if a component in spark had direct access to some components then it should also have direct access in my simulator. This chapter gives an overview of the architecture and design of the system and explain about the design process itself. Therefore, the chapter will not explain each and every diagram developed during the course of the project. The chapter will focus on the most important design aspects of the simulator and to explain the principles followed during the design process. The details and explanations of the rest of the diagrams can be found in the appendix section.

Design model

As it was explained in the previous section, the design began creating an analysis model. The analysis model contains all of the entities in the problem domain. The analysis diagram then slowly evolved to a class diagram. Here we show the class diagram of the entire system and explain the important design considerations.

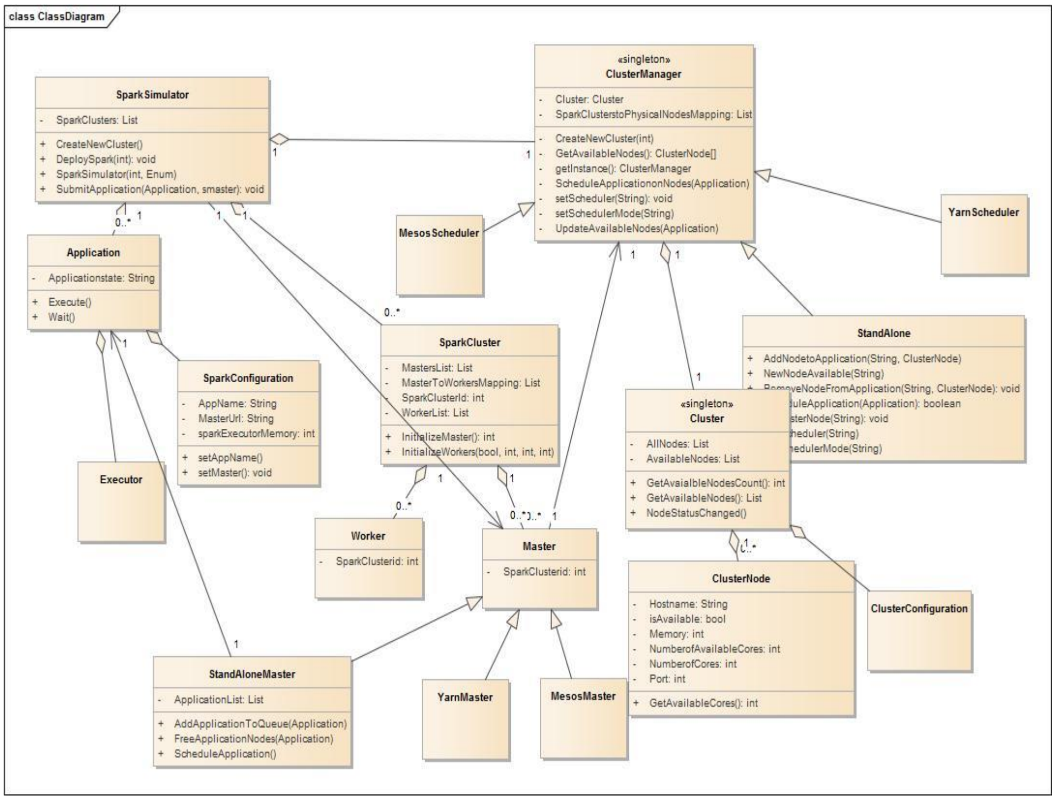

class ClassDiagram
SparkSimulator
- SparkClusters: List
+ CreateNewCluster()
+ DeploySpark(int): void
+ SparkSimulator(int, Enum)
+ SubmitApplication(Application, smaster): void
«singleton»
ClusterManager
- Cluster: Cluster
- SparkClusterstoPhysicalNodesMapping: List
- CreateNewCluster(int)
- GetAvailableNodes(): ClusterNode[]
- getInstance(): ClusterManager
- ScheduleApplicationonNodes(Application)
- setScheduler(String): void
- setSchedulerMode(String)
- UpdateAvailableNodes(Application)
MesosScheduler
YarnScheduler
Application
- Applicationstate: String
+ Execute()
+ Wait()
SparkConfiguration
- AppName: String
- MasterUrl: String
- sparkExecutorMemory: int
+ setAppName()
+ setMaster(): void
SparkCluster
- MastersList: List
- MasterToWorkersMapping: List
- SparkClusterId: int
- WorkerList: List
+ InitializeMaster(): int
+ InitializeWorkers(bool, int, int, int)
StandAlone
+ AddNodetoApplication(String, ClusterNode)
+ NewNodeAvailable(String)
+ RemoveNodeFromApplication(String, ClusterNode): void
«singleton»
Cluster
- AllNodes: List
- AvailableNodes: List
+ GetAvaialbleNodesCount(): int
+ GetAvailableNodes(): List
+ NodeStatusChanged()
Executor
Worker
- SparkClusterid: int
Master
- SparkClusterid: int
ClusterNode
- Hostname: String
- isAvailable: bool
- Memory: int
- NumberofAvailableCores: int
- NumberofCores: int
- Port: int
+ GetAvailableCores(): int
ClusterConfiguration
StandAloneMaster
- ApplicationList: List
+ AddApplicationToQueue(Application)
+ FreeApplicationNodes(Application)
+ ScheduleApplication()
YarnMaster
MesosMaster
1
0..*
1..*

Figure 9 Class Diagram of the Simulator

The “SparkSimulator” class serves as a Façade to the rest of the system. A Façade is an interface that serves to simplify the interface for the user to a complex system and makes thus it is easy to use. This provides an interface for not only this use case but for all of the other use cases as well.

Although, we only planned to focus on the Standalone cluster manager, the system is designed in such a way that it is easy to add new cluster managers in the future. From the figure 9, we can see that there is a “ClusterManager” class and three types of cluster managers are inheriting from that class. This is because Spark has three cluster managers namely, Standalone, Yarn and Mesos. Besides these cluster managers, it would be easy to add a new cluster manager in the future. Similarly, from figure 9, we can see that there is a parent Master and it has three children namely, “StandAloneMaster”,”YarnMaster” and “MesosMaster”.

The “Cluster” class represents a physical cluster in the simulator. A “Cluster” class is composed of several “ClusterNode” objects. A “ClusterNode” is a representation of a single processing node in the cluster. This node could have some processing power and a certain amount of memory. So if a cluster is a network of computers connected together then the cluster node is a single computer in the network. The “ClusterManager” maintains information such as which node is free and which node is not free.

The classes “StandAloneMaster” and “Worker” are important components of spark. The Worker is a representation of a machine resource in spark. This is different from a cluster node which is the representation of a real machine. A worker is deployed on a cluster node and gets some resources from the cluster node. The existence of a worker is important because at one time multiple users may be able to use a machine and each user can only get a certain amount of resources from the machine that is represented by the worker. A Worker gives resources to an application in the form of JVMs called executors.

The “StandAloneMaster” class is responsible for scheduling of resources across application. The “StandAloneMaster” schedules applications using a First Come First Serve Algorithm. Although the scheduling policy can be easily changed as we will see later in the design patterns section. The standalone master is also responsible for allocation of executors among the workers. As mentioned in the previous section, the standalone master inherits from the generic master class so that we can add more types of masters in the future.

The “SparkCluster” class maintains a mapping of a master and its workers. As it was mentioned earlier, it is important to have this class for scalability purposes. Currently, it is only possible to have a single deployment of spark but in the future it must be possible to have multiple deployments of spark.

Genericity vs Functionality

One of the most important design issues was keeping a balance between genericity and functionality. The simulator needs to be generic enough to be easily extensible but it should capture enough functionality of the real spark so that the clients can perform their experiments. If the simulator is too generic then it can be easily extensible but then it is too abstract to capture any real functionality. If the simulator is too focused on the functionality then it may not be easily extensible.

To maintain a balance between functionality and genericity, I introduced generic parent classes to the required components. The required components then inherited from the parent class. An example would be the “Master” class in Figure 9. Only the “StandAloneMaster” was required in the simulator because the simulator is only focused on the standalone mode. The addition of a generic “Master” class makes it easy to extend the simulator with other modes like a Yarn master, Mesos master or even a new type of master that hasn’t been implemented yet.

Another example of genericity is the existence of the “SparkCluster” class. The current requirement was to only support one user at a time. In the future, it should be possible to simulate time sharing of different spark clusters. The sparkcluster class was added to make it easy in the future to extend the simulator to include multiple deployments of spark.

Application Scheduling

Currently, there are three scheduling policies in the simulator. The three scheduling policies are as follows:

1. First Come First Serve Scheduling (FCFS)
2. Priority Scheduling
3. Shortest Job First Scheduling (SJF)

Out of all these scheduling policies, Spark only supports First Come First Serve Scheduling in Standalone mode. The other two scheduling policies were added to show how easy it was to add and switch between different scheduling policies. The simulator was made extensible for adding new scheduling policies by using the strategy [11] design pattern.

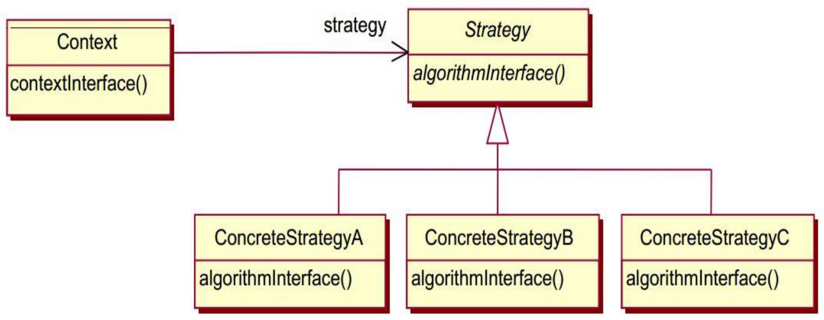


Figure 10 Strategy Design Pattern

Strategy is a design pattern that is used to define a family of algorithms and make them easily interchangeable with each other at run time. Strategy lets the algorithms vary independently from the clients that use it. The strategy design pattern has been implemented in this project.

Example:
You have to encrypt a file.

For small files, you can use "in memory" strategy, where the complete file is read and kept in memory (let's say for files smaller than 1 gigabyte). For large files, you can use another strategy, where parts of the file are read in memory and partial encrypted results are stored in temporary files. These may be two different strategies for the same task.

"Strategy" declares an interface to all supported algorithms. Context uses this interface to call the algorithm defined by a "ConcreteStrategy". "ConcreteStrategy" implements the algorithm using the "Strategy" interface. "Context" is configured with a "ConcreteStrategy" object. It maintains a reference to a "Strategy" object. It may define an interface that lets "Strategy" access its data.

The class diagram of Scheduling strategy can be seen in Figure 11. The "SchedulingStrategy" class provides an interface to the application scheduler. All three scheduling algorithms implement this strategy. Each scheduling strategy provides a scheduling algorithm that can be chosen at run time. The use of the strategy pattern makes it easier to add new scheduling policies to the code and switch between different scheduling policies.

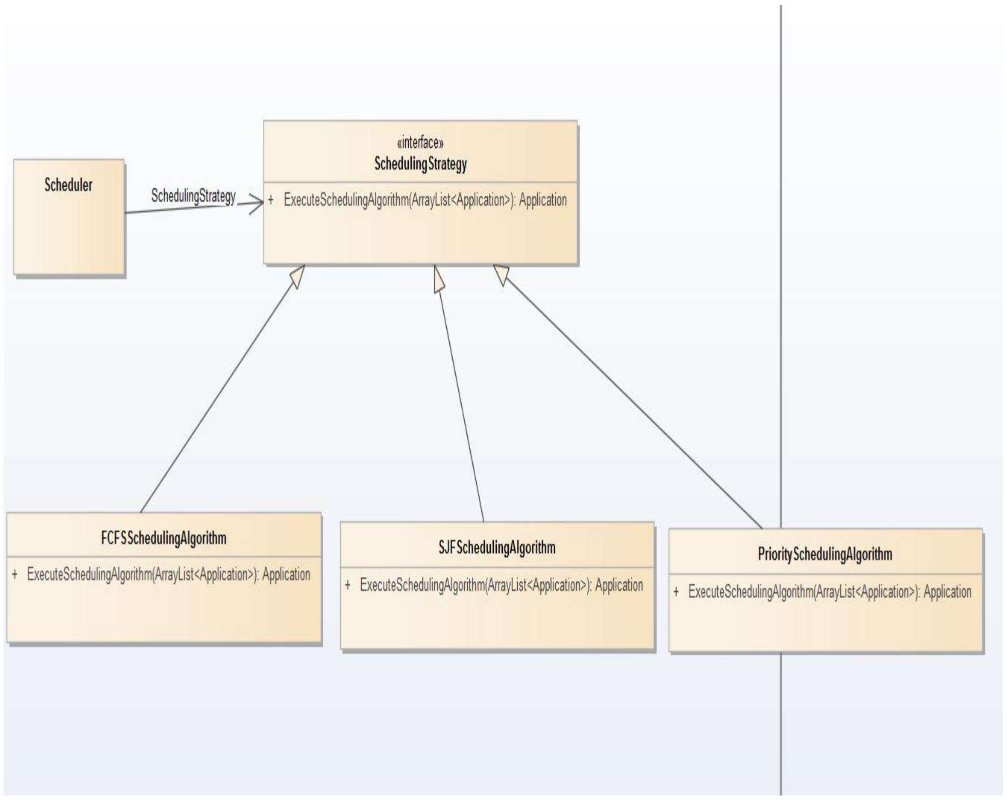

Scheduler
SchedulingStrategy
«interface»
SchedulingStrategy
+ ExecuteSchedulingAlgorithm(ArrayList<Application>): Application
FCFSSchedulingAlgorithm
+ ExecuteSchedulingAlgorithm(ArrayList<Application>): Application
SJFSchedulingAlgorithm
+ ExecuteSchedulingAlgorithm(ArrayList<Application>): Application
PrioritySchedulingAlgorithm
+ ExecuteSchedulingAlgorithm(ArrayList<Application>): Application

Figure 11 Scheduling Strategy

7.3.3. Process view

The process view describes the behavior of different entities at run time and how these entities interact with each other at run time. We present the process view by using a sequence diagram and a state machine diagram.

Sequence Diagram

A Sequence diagram is used to show the communication between different objects. The Diagram shows the actual messages exchanged between the different objects in the form of function invocations.

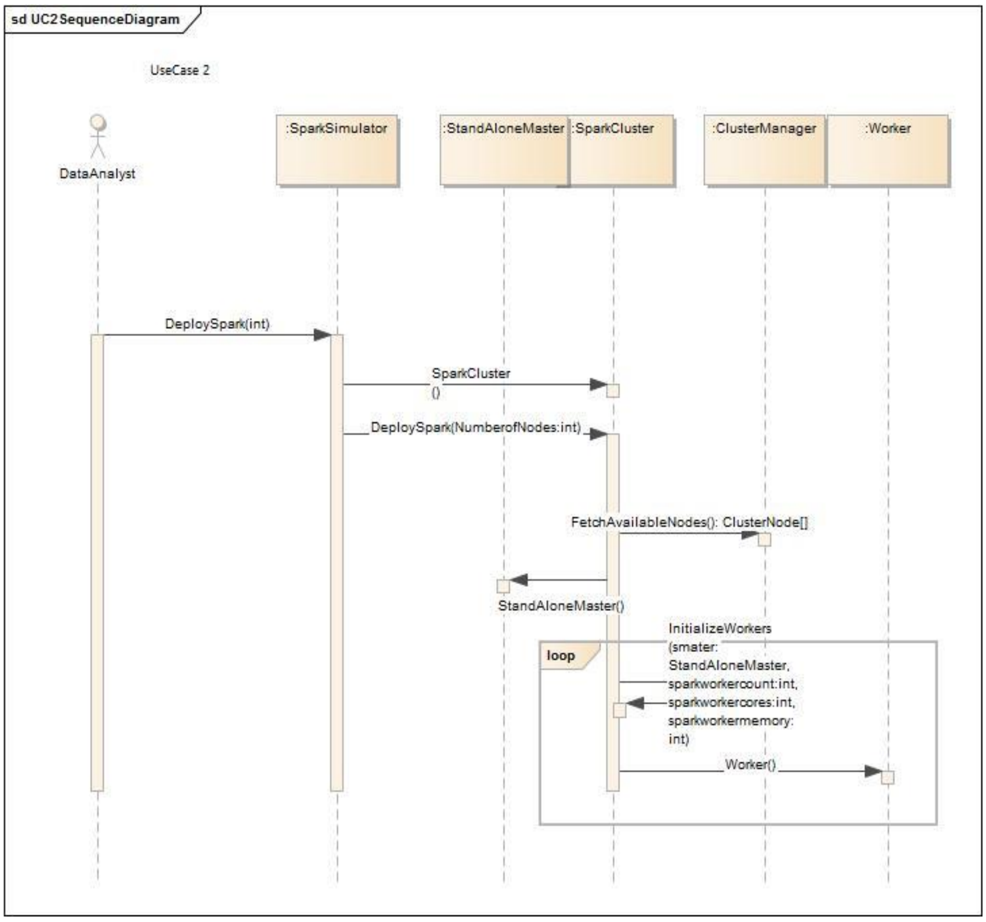

sd UC2SequenceDiagram
UseCase 2
DataAnalyst
:SparkSimulator
:StandAloneMaster
:SparkCluster
:ClusterManager
:Worker
DeploySpark(int)
SparkCluster()
DeploySpark(NumberofNodes:int)
FetchAvailableNodes(): ClusterNode[]
StandAloneMaster()
loop
InitializeWorkers(smater: StandAloneMaster, sparkworkercount:int, sparkworkercores:int, sparkworkermemory: int)
Worker()

Figure 12 Spark Deployment Sequence Diagram

The figure 12 shows the sequence diagram of the Spark deployment use case. The user requests the Spark simulator to make a deployment of spark with a specific number of nodes. The "SparkSimulator" object acts as a Façade to the rest of the system. It provides simple interfaces for the user to access the complex sub-components of the system. The method "Deployspark" is called with the number of required nodes as a parameter. The method creates a new Spark object and adds it to its list of Spark clusters. A Spark cluster contains information about one Spark deployment. It contains information about the Master and Workers created during the Spark deployment. The Simulator maintains a list of Spark clusters to make it easily extensible. Currently, the simulator only supports one Spark deployment at a time but in the future it should be able to support multiple Spark deployments. The list is maintained to make it easy to extend the current model to support multiple Spark deployments.

After the Spark cluster has been created then the "DeploySpark" method of the Spark cluster is invoked. The Deployment process begins by reading important configuration information from an xml file. The File includes information such as the number of workers per node and the number of cores required by each worker. The deployment process then continues by fetching the available physical nodes from Cluster manager. A Physical Node means a physical machine. If the number of available nodes are greater than the required number of nodes then the execution continues otherwise the method terminates its execution. If the required number of nodes are available then it assigns the master node to the first physical node and tries to create the specified number of workers on each node. If the resources required by a worker are not available then the method terminates otherwise it assigns the resources the workers to physical nodes. The deployment process finishes after all the workers have been created and assigned to the physical nodes. After the completion of the

Deployment process a dictionary is created which keeps track that which workers are assigned to a master.

State machine diagrams

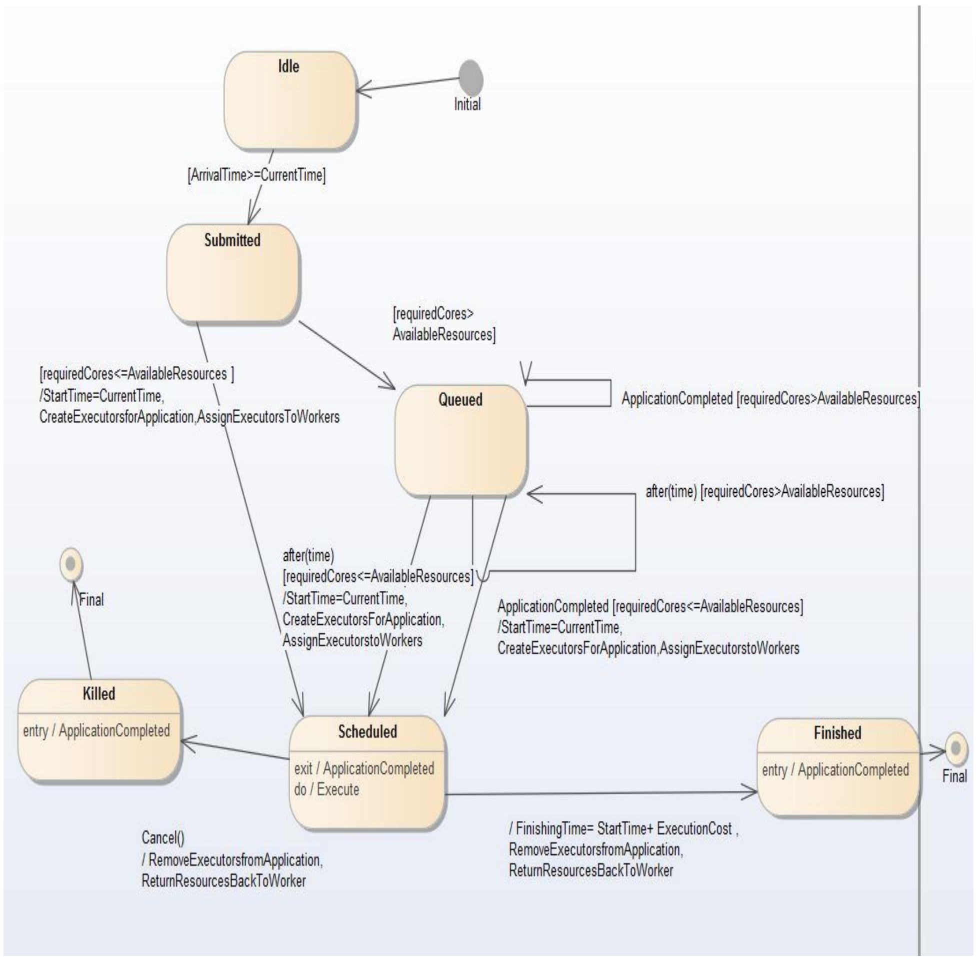

Idle
Initial
[ArrivalTime>=CurrentTime]
Submitted
[requiredCores> AvailableResources]
[requiredCores<=AvailableResources] /StartTime=CurrentTime, CreateExecutorsforApplication,AssignExecutorsToWorkers
Queued
ApplicationCompleted [requiredCores>AvailableResources]
after(time) [requiredCores>AvailableResources]
after(time) [requiredCores<=AvailableResources] /StartTime=CurrentTime, CreateExecutorsForApplication, AssignExecutorstoWorkers
ApplicationCompleted [requiredCores<=AvailableResources] /StartTime=CurrentTime, CreateExecutorsForApplication,AssignExecutorstoWorkers
Final
Killed
entry / ApplicationCompleted
Scheduled
exit / ApplicationCompleted
do / Execute
Finished
entry / ApplicationCompleted
Final
Cancel() / RemoveExecutorsfromApplication, ReturnResourcesBackToWorker
/ FinishingTime= StartTime+ ExecutionCost , RemoveExecutorsfromApplication, ReturnResourcesBackToWorker

Figure 13 State Machine Diagram of an application

The Simulator receives a file as input which includes information about the applications. The file includes information such as the total number of applications and the number of resources required by each application. The Simulator generates a list of applications after reading from the file. There is a class called “ApplicationGenerator” which generates the arrival times of all of the applications.

The Application is initially in the idle state. The Scheduler will check if the arrival time is greater than the current time. If so, then the application goes to the submitted state. The Scheduler will keep on calling submit application on every application till the condition Arrival Time>=Current Time holds true. After that it will setup an event for the time when the next application will arrive.

After the application state has been changed to the submitted state then the scheduler will try to schedule the application. An application can only be scheduled if the number of cores required by the application can be served by the Spark deployment. The spark cluster has a list of free workers and it will be checked if there are enough Workers to meet the resources requirements of the application. If there are enough workers then new executors are created for the application. The exact procedure of resources assignment to the applications is explained in more detail in the implementation chapter.

After the application has been scheduled then it will be removed from the queue and the free workers list of spark cluster will be updated. The application start time will be equal to the current time. Current time+ Execution cost will be equal to the finishing time. After finishing the application will fire a notification "ApplicationCompleted". This will make spark cluster check the queue again and more applications can be tried for scheduling.This is not really helpful now because applications are FCFS but it will be good for scalability.

7.3.4. Development view

The Development view, sometimes also referred as the implementation view, presents the system from the programmer’s perspective. We present the deployment view by using a package diagram and a state machine diagram. A package diagram presents the relationships between the different packages. A State machine diagram represents the different states an object goes through. A State machine basically shows how an object changes during its lifecycle.

Package Diagram

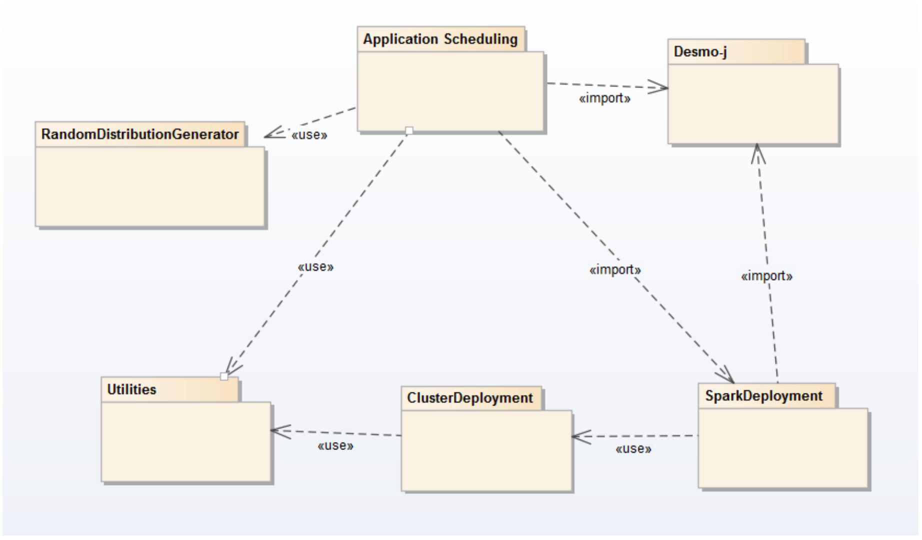


Figure 14 Package Diagram of simulator

The figure 13 shows the main packages in the simulator and their relationships. Desmo-j is the java library that is used for simulation in the project. The library is used by the Application Scheduling package to provide scheduling functionalities. Spark Deployment also needs to use this because the Master and Workers need to be active entities in the simulator. The Application Scheduling package is responsible for the application creation, submission and scheduling. The Application Scheduling package uses the RandomDistributionGenerator for experiments. The RandomDistributionGenerator provides an interface to the application to generate random “Executiontimes” for its tasks. The ClusterDeployment package contains implementation of the physical machines. The ClusterDeployment is used by the SparkDeployment package for deploying workers on machines. The utilities package is used by other packages to provide functionalities such as file parsing.

7.3.5. Deployment view

The Deployment view, sometimes also referred to as the physical view, describes the mapping between the system and the underlying hardware. The deployment view is helpful for system engineers and administrators because they should know which component should be deployed on which hardware.
In our case, the simulator is a desktop application and it would be deployed on a modern personal computer. The deployment is really simple and it is very helpful for the users of the system because they don't have to worry about deployment issues. The users of the simulator can focus on their experiments. The only requirement for the hardware is that it must support multi-threading.

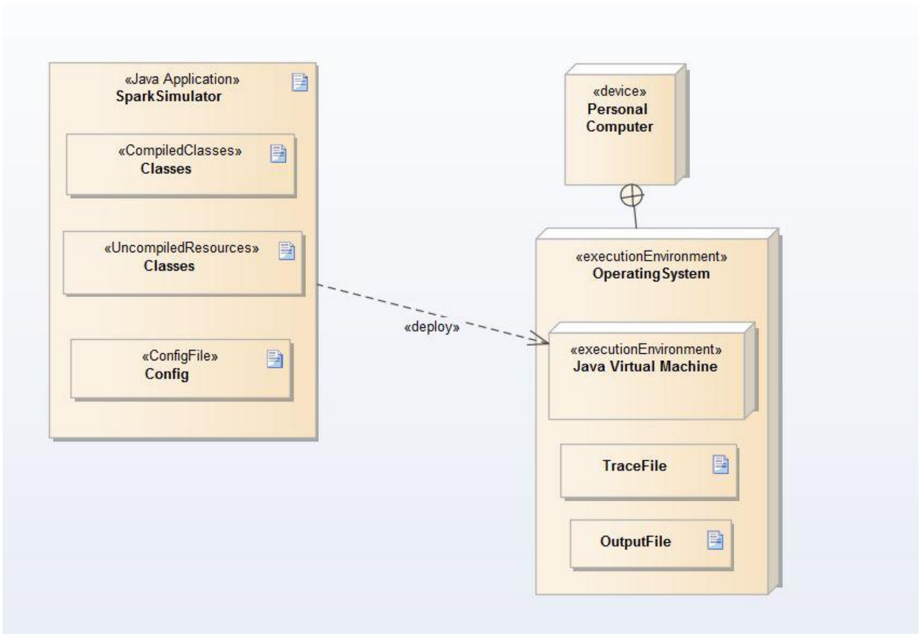


Figure 15 Deployment Diagram of the simulator

The figure 15 shows the deployment diagram of the simulator. The simulator can be deployed on any modern personal computer capable of multithreading. As we can see from the figure 15, "SparkSimulator" is our java application that is deployed on the personal computer. The application runs inside the java virtual machine. The application is deployed automatically when it is executed. There are no separate steps required for deployment. We can see two important files in the diagram, namely "TraceFile" and "OutputFile". The presence of a "TraceFile" is important because it contains experimental data from the real spark which is used as an input to the simulator. The "OutputFile" is the file generated by the simulator and it contains the results from the simulation.

8.Implementation

In this chapter, we discuss the implementation of the system. The implementation is divided into four parts based on the use cases mentioned in the Architecture and Design chapter. The implementation of the Spark deployment, application scheduling, application generation and implementation of clusters. The implementation of these parts is explained in this chapter.

8.1 Introduction

The implementation of the project was done in parallel with the design. A prototype was made in short sprints and feedback was received from the client after every sprint. This process was used to make an initial skeleton of the project. The requirements changed many times, so implementation changed also. It is easy to make changes because the project has a generic and flexible design. The implementation process was a process of continuous feedback.

8.2 Tools

There were many programing languages to choose from. Since, a similar product was done in Java and the client wanted integration with that product, Java is chosen as the programming language for the project. Java is also a good choice because it is quite close to Scala. It is possible to code in Scala in the java way, which will make it easy to connect the simulator to the actual Spark in the future.

8.3 Libraries

The Desmo-j [14] library is used for simulation in the project. Desmo-j stands for "Discrete Event Simulation Modelling in Java" and it has been developed at the Hamburg University. Desmo-j provides the ability to create a large number of threads. It is important because the simulator should be able to support a large number of threads. Desmo-j makes it easier to implement simulations by providing built-in support for common simulation artifacts such as, the implementation of a virtual clock, an event scheduler, event queue, conditional waiting and others.

Desmo-j was chosen for the following reasons:

1. Desmo-j is a free library and would not introduce any copyright issues in the project.
2. The Existing product is using Desmo-j which made it easier to integrate with the existing product and I needed a strong reason to switch to a different library.
3. The Library provides a flexible way to implement a simulation model because the library is simple and easy to change.
4. Desmo-j is a more efficient library for large simulations when compared with other libraries such as SSJ. This is more efficient because it allows one to create a large number of threads as compared to SSj.
5. Desmo-j is more actively developed as compared to all other simulation libraries.

8.4 Integration issues

As stated above, the client wanted this project to be integrated with an existing product. A number of adaptations were made to the existing product to improve its design and make it more extensible. There was no concept of Spark cluster in the existing product, so a Spark cluster is introduced. The purpose of the Spark cluster is to be able to have multiple Spark clusters in the future. The workers were not stored anywhere in the existing product. The implementation is changed to store the workers in the Spark Cluster.

In the existing product, the executors were created right after workers were created. This is changed to be more in line with the real Spark i.e. the executors are created when an application requests some resources. A lot of refactoring is done to remove variables from classes where they don’t belong. An example is that in the existing product, the clusters (physical machines) were creating the workers. This is changed to the Master creating the workers, as it is in the real Spark.

8.5 Configuration Parameters

The simulator needs three types of configuration files.

1. Infrastructure configuration.
2. Application configuration.
3. Spark configuration.

8.5.1. Infrastructure configuration

This infrastructure configuration contains parameters related to the construction of a representation of the physical nodes in the simulator. The file is named as “virtualNode” and is a simple text file. The file contains information such as the number of nodes to construct, the number of cores in each node and the memory in each node. In the current version of the simulator, the nodes are considered to be homogenous i.e. the only difference between the nodes is the number of cores. A snippet of the file can be found in the appendix section.

8.5.2. Application configuration

The application input contains parameters related to an application in the simulator. The Application input file is a CSV file and it is named as “ApplicationListCSV”. The application input file contains the application name, execution cost, priority and required resources. The application arrival time and type are generated by an application generator explained in the later sections. A snippet of the file can be found in the appendix section.

8.5.3. Spark configuration

This file contains information about the Spark deployment. It is named as “spark-env” and it is a simple text file. It contains information such as the number of workers that must be created, the number of cores in each worker and the memory allocated to each worker. A snippet of the file can be found in the appendix section.

8.6 Implementation

8.6.1. Spark Deployment implementation

The simulator begins the deployment of Spark by reading important configuration information from the Spark configuration file. The file includes such information as number of workers per node and number of cores required by each worker. The simulator then continues by fetching the available nodes from Cluster manager. If the number of available nodes are greater than the required number of nodes then the execution continues otherwise the method terminates its execution.

If the required number of nodes are available then it assigns the master node to the first node and tries to create the specified number of workers on each node. If the resources required by a worker are not available then the method terminates otherwise it assigns the resources to the worker. After all the workers have been created and assigned to nodes then the workers are assigned to a master. All of this information is a part of the Spark Cluster.

8.6.2. Application generator implementation

The application generator is used to supply input to the scheduler. The application generator reads a CSV file of applications. The CSV file includes the application name, priority, execution cost and required cores. The priority of the application is not used at the moment but “0” represents the highest priority. The priority can be used when using a priority based scheduling algorithm. The execution cost is the amount of time in seconds that the application hold its resources. The arrival time of the application and the type of the application are not put inside the CSV file but they are generated randomly. The arrival times are generated using an exponential distribution and the types are based on a probability given by the user.

So when the application generator reads the CSV file then it loads all of the applications into an application object array. After it has read all of the application objects then it will setup “arrival” events for all of the applications. These events will be set up using the Desmo-j library. When the application arrival event is launched the application generator will notify the scheduler that there is a new application in the queue.

8.6.3. Scheduling across applications

Currently, the model of an application is that it requires a certain number of cores for execution, it has an execution time and an arrival time. Applications are generated by the application generator based on their arrival times. When an application’s arrival time has been reached then the application enters the system. The scheduler selects the next application based on the scheduling algorithm. The Standalone mode has a First Come First Come scheduling algorithm. When an application has been selected for scheduling then the scheduler will check whether it has enough resources to give to the application. An application will request a certain number of cores. If the scheduler has enough resources for the application then it will check the list of workers and it will start enough executors for the application to satisfy its needs. When the application has been assigned its executors then it will wait for the duration of its execution cost and after that the application is finished. When an application frees up some resources then the scheduler is invoked to select a new application to schedule.

8.6.4. Resource Assignment

When Spark has enough resources then an application can be scheduled. The scheduling of the applications begins by assigning resources to the application. Resources can be assigned to the application in either “spreadout” mode or “non-spreadout” mode. The explanations of these modes are given in their subsections.

8.6.4.1 Spread-Out

When an application arrives in the system then it is checked that how many cores it requires for its execution. The application will also have a minimum number of cores per executor. For example, if an application requires four cores and the minimum number of cores per executor is two and there are four workers with one core free then the application cannot execute.

The Master will fetch all of the workers that satisfy the minimum requirements for the application. Now the master has a list of usable workers. After this step, the master will start executors in a load balanced way. The Master will go through every worker in a round robin fashion and start an executor with the number of cores specified in the configuration file. The master will attempt to launch an executor on every worker. This will continues till the application has received all of its required resources.

8.6.4.2 Non Spread-Out

The resource assignment process in non spread-out mode is almost similar to the process described above. The only difference is that now the master will try to use as few workers as possible. The Master will try to use all of the cores available on the worker.

First, the list of usable workers will be fetched based on the minimum requirements of the application. After the master has the list of usable workers then it will go through each worker and try to create an executor using all of the cores of the worker. This process will continue till the application has all of its required cores.

8.6.5. Scheduling within an application

The project initially began with a simple model of the application. Initially, an application had a fixed execution cost and when an application was launched then it held the resources for a fixed amount of time. Later, the application model was extended to include jobs, stages and tasks. This section will discuss the extended application model.

An Application has a set of jobs. Each job contains a number of stages and each stage contains a certain number of tasks. When an application is launched then it creates a Directed Acyclic Graph (DAG) scheduler and a task scheduler. The DAG scheduler iterates through the set of jobs belonging to that application and schedules them in a First Come First Serve (FCFS) manner. The first job gets the priority over all the resources. So, it needs all the resources assigned to the application then it will get those resources.

Each job has a set of stages and each stage contains a certain number of tasks. The stages inside a job may be dependent on other stages and cannot be scheduled till all of their predecessors are scheduled. The resource requirements of a job depends on all of the stages it can schedule at that time. The DAG scheduler will go through the list of jobs in a FCFS manner and start assigning resources to each job. A job is launched after resources have been assigned to the job. A job can be scheduled even if its resources requirements are partially satisfied.

After a job has been scheduled, the next step is to launch the tasks from all the independent stages inside that job. All of the tasks from the independent stages will be sent to the task scheduler. The task scheduler will begin by launching the tasks according to their priority. The priority of a task depends on the arrival time of its parent job. The task scheduler will launch one task on one core. If there are no more cores then the remaining tasks have to wait. The execution time of a task depends on a random number generated by the simulator. A job is finished once all of its tasks have been executed. An application is finished after all of its jobs have finished.

9.Verification & Validation

In this chapter, some experiments are performed with the simulator. The simulator prototype is compared with the output of the real Spark to show that the simulator provides the required functionality.

9.1 Experiments

A number of experiments were performed by using the simulator and the output of the simulator was compared with the output of the real Spark. We defined a Spark cluster with 10 worker nodes and each worker node had 8 cores. So the simulator had 80 cores in total. A number of applications were generated by using the Application generator module. The resource requirement and execution costs of the applications were generated by using an application generator. The arrival time of the applications was generated based on an exponential distribution. Since, the simulator was not detailed enough to compare with real applications, therefore, some dummy applications were created on the real Spark.

Three types of applications were generated for the system, namely “Type 1”, “Type 2” and “Type 3”. For each application, an execution cost was selected from the range of that application type and it was picked based on a uniform distribution. The execution cost range of each application type is given in table 3. For example, the execution cost range of application “Type 1” is between 20 and 40 seconds. A number is picked by the uniform distribution, let’s say 25 and assigned as the execution cost for that application.

All of these application types had different probabilities of generation as show in table 3. The “Type 1” applications had a large probability of 70 percent because they are really short applications as compared to the other types and we wanted to have a large number of very small applications. The “Type 2” applications had a probability of 25 percent and had more execution cost than “Type 1” applications. The “Type 3” applications were the largest applications in terms of execution cost and had the smallest probability i.e. 5 percent. Each application type also has a resource requirement. This resource requirement is described as the number of cores required by the application. The resource requirements of the application can also be seen in the table 3.

The arrival rate “lambda” was calculated for the applications by using the formula “1”:

$$\boldsymbol{\lambda} = \mu * \mathbf{C} * \mathbf{pl} \qquad 1$$

Where “μ” is the service rate of the system and is defined as (1/Average Execution Cost), “C” is the total number of cores in the system, “pl”, which is sometimes also referred to as “offered load” or “postload”, is the measure of traffic in the queue.

Table 2 Showing the execution cost range, required resources, probability of arrival and the average execution cost for each type

Application Type	Execution cost(seconds)	Required resources (cores)	Average Execution Cost(seconds)	Probability
Type 1	20-40	20	600	70 %

Type 2	300-500	40	16000	25 %
Type 3	1200-1400	60	78000	5 %

We already know that the total number of cores or “C” in our setup is 80. We set the postload or “pl” value to be 0.7. The value of μ can be calculated by using the formula “2”:

$$\mu = \frac{\mathbf{1}}{\boldsymbol{Exe_{Avg}}} \qquad 2$$

The term Exe_{Avg} is the average execution cost can be calculated by multiplying the probability of each application type with average execution cost of that application type. The formula for calculating the average execution cost is given in formula “3”

$$\boldsymbol{Exe_{Avg}} = \mathbf{Type1} * \mathbf{p1} + \mathbf{Type2} * \mathbf{p2} + \mathbf{Type3} * \mathbf{p3} \qquad 3$$

Where the types “Type1”, “Type2” and “Type3” represent the average execution cost of that type of application and “p1”, “p2” and “p3” represent the probability of occurrence of that application. So, formula “1” becomes formula “3” after inserting the value of μ:

$$\boldsymbol{\lambda} = \frac{\boldsymbol{C*pl}}{\boldsymbol{Exe_{Avg}}} \qquad 4$$

= (1/8320) * 80 * 0.7= 0.0067

The arrival times of applications were generated using an exponential distribution. The system ran for 6 hours and the applications only executed for that time.

9.2 Results

The output file generated by the simulator was compared with the output file generated by Spark. This was done in order to verify the correctness of the simulator and to make sure it fulfills the requirements.

After comparison between the simulator and Spark, it was found out that there was only a small difference of 5 percent between the total executors per application. When the starting and ending times of applications were compared, it was found out that there was only a difference of 13 percent between Spark and the simulator. This is due to the fact that in real Spark it takes some time to communicate between different components of Spark such as the Master, Workers and the driver. This is the delay due to network communication as these components exist on different physical nodes. This delay is also quite unpredictable and there is not a constant factor that can be added to the simulator to make the outputs more close. In our simulator, this communication is instant as there are no network delays in our simulator. Another comparison was done between the cores occupied by the system over time. There was a bigger difference in this measurement as compared to our previous measurement. The reason was that Spark has partial resource allocation whereas our model has all or nothing resource allocation. This can be improved in the future by extending the model to include partial resource allocation.

Figure 16 shows the graph generated by using data from the real Spark and figure 17 shows the graph generated by using the data from the simulator. The x-axis shows the values of time in seconds and the y-axis(top) shows the total number of cores.

The y-axis on the bottom shows the start time of the applications. The blue graph shows the occupied number of cores over time and the orange graph shows the total number of applications executing at that time.

Figure 16 Total occupied cores by time in Spark

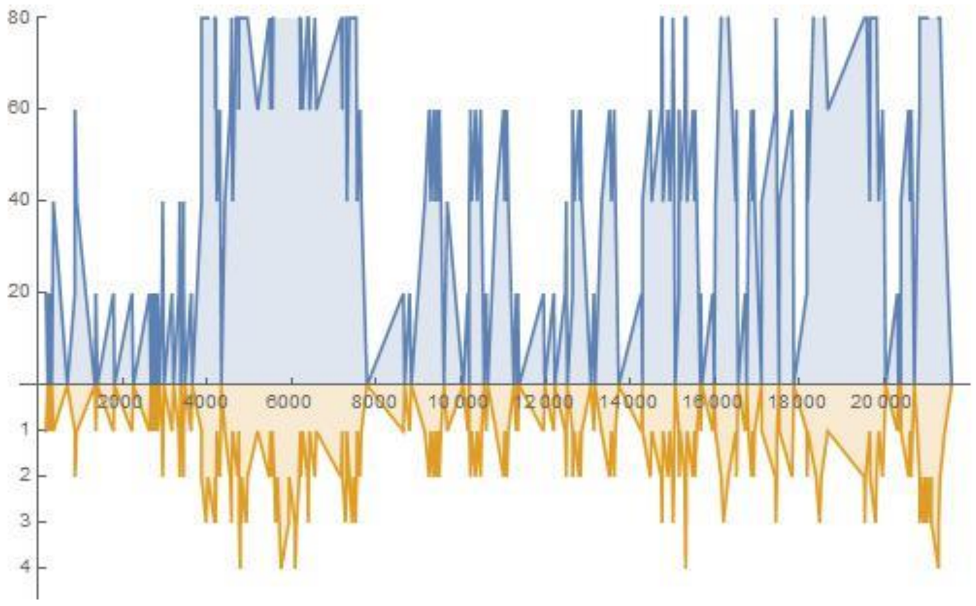
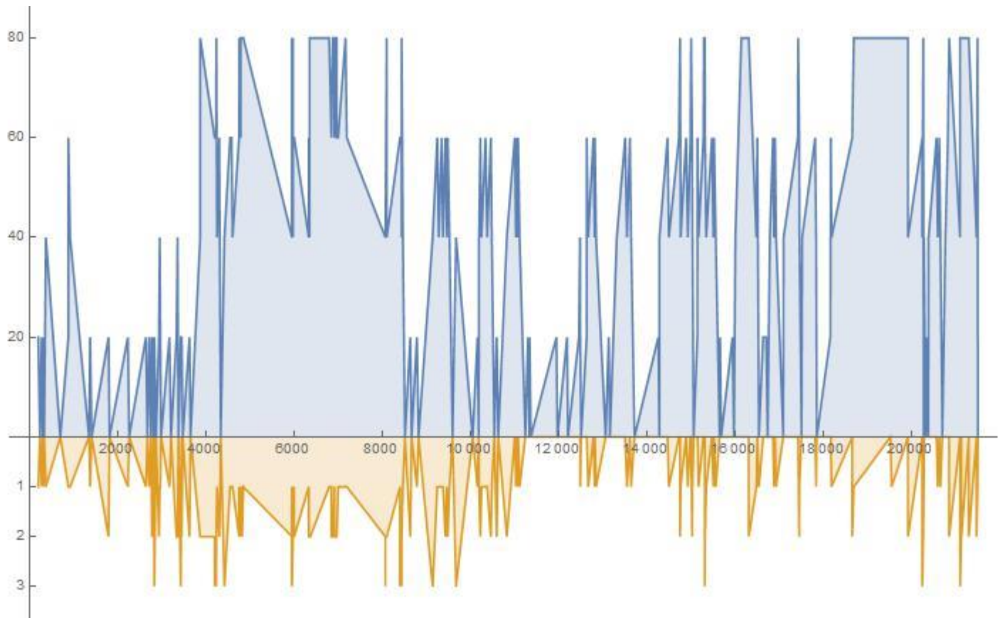


Figure 17 Total occupied cores by time in simulator



9.3 *Conclusion*

After comparing the output of the simulator with the traces generated by Spark, it was found out that the simulator satisfies the stakeholders' needs to a certain level but there is still room for improvement in the future. The important functionality expected from the simulator was successfully implemented given the limited time frame and the simulator looks promising for the future.

10. Conclusions

In this chapter we discuss the results produced by the project and any future work for the project.

10.1 Results

Spark has several configurations and selecting the appropriate configuration is important for achieving the best performance with Spark. To find the best configuration to run a particular set of applications, several experiments need to be performed. If these experiments are performed on the real Spark then they will be quite expensive and time consuming. The main focus of the project was to create a simulator for Spark, so, that the users can save time and money by running their applications with the best configuration options. This would save a lot of money because no money would be wasted on real Spark to find best ways to run an application. The users would also save time because the simulator provides results instantaneously as opposed to Spark where one has to wait for the end of the experiments.

The simulator allows the user to create a deployment of Spark, input applications to the deployment, and generate arrival times of those applications using an exponential distribution and schedule applications using the selected cluster manager and the scheduling policy. The simulator would save time and money of the users by allowing the users to find the best configuration for their applications. The use of a generic design makes it easy to add new scheduling policies in the future and add new cluster managers. A generic design also makes it easier to make changes to the simulator when required.

The design of the simulator is extensively documented using class diagrams, use cases and sequence diagrams. There are details of the diagrams and code in this report which will make it easier for people to work on and extend this simulator in the future. The correctness of the simulator has been verified by performing extensive test cases on the simulator.

10.2 Future work

The project was focused on creating an abstract implementation of the Spark simulator. Due to shortage of time, more focus was placed on implementing a representation of the Spark deployment and the standalone cluster manager. The project can be extended in the following ways.

1. **Mesos and Yarn Cluster managers**

The Mesos and Yarn cluster managers could be added as an alternative to the existing standalone cluster manager. Currently, there is only the standalone cluster manager in the simulator. The architecture is generic enough to add new cluster managers easily and without modifying a lot of things.

2. Connection with the real Spark simulator

The simulator could be extended to be connected with the real Spark. This could be used to compare the output of the real Spark with the simulator.

3. Extend the application model

Although, the currently existing application model is really good it can be extended to make it more realistic and closer to the real Spark.

4. Heterogeneous physical nodes

The physical nodes that are currently in the simulator are assumed to be homogenous. They are only differentiated based on the number of cores they have and not according to their processing power. In the future, we could have physical nodes with different processing powers. So a task scheduled on a node with 3GHz processing power will finish faster than a task scheduled on a node with a 2GHz processing power.

5. Multiple Spark deployments at the same time

Currently, there can be only one deployment of Spark at a time but in the future it can have multiple deployments at once. The architecture is generic enough to be easily extended. When one deploys Spark then a new Spark cluster is created with the information about the deployment. This Spark cluster is added to a list of Spark clusters and it can be fetched by using its name. One can similarly add multiple Spark clusters to the list to have multiple deployments.

11. Project Management

In this chapter, we explain the project management techniques used in the project.

11.1 Introduction

The project was developed using an agile methodology [12]. In the agile methodology, there are short development cycles, known as sprints. After each sprint, there was a meeting with the stakeholders to discuss the deliverables during the sprint. These deliverables could be anything such as a requirements document, a design or a working prototype. The deliverables were reviewed in the meeting and feedback was given for each deliverable.

In the beginning, we had a one week sprint but later we moved to a regular two week sprint. A short sprint was introduced in the beginning so that the clients can makes sure that I have a correct understanding of the domain and the requirements. I communicated with the clients using different diagrams like the Use Cases Diagrams and Analysis Class diagrams.

11.2 Work Breakdown Structure

The Work breakdown structure of the project is shown in figure 18. From the figure, we can see that there were three main activities during the project, namely domain analysis, design and implementation. However, these activities were not carried out in strict separation but rather in combination with each other.

The Design already began with the domain analysis portion of the project. As it was explained in earlier chapters, a problem analysis diagram was developed during the problem and domain analysis phases of the project. The diagram helped to make sure that the problem was defined correctly. The implementation was also carried out with design as with every design there was a small prototype.

The domain analysis consisted of the following activities:

- **Spark Domain**: Research and study about the main components of Spark and the architecture of Spark.
- **Standalone**: Research and understand how scheduling works in Spark, especially, the standalone scheduling.
- **Big data and Parallelism**: Research on big data and parallelism concepts such as multithreading etc.

The design phase consisted of the following activities:

- **Spark Domain**: Design the main components of Spark and how they would interact with each other.
- **Standalone**: Design of the Standalone scheduler in spark.

The implementation phase consisted of implementing everything that was designed. In the beginning, the implementation phase was coupled with the design phase as there was a prototype with every design. After a few iterations, the design became fixed and there was no overlap between design and implementation phases. More details about the implementation phase can be found in the implementation chapter.

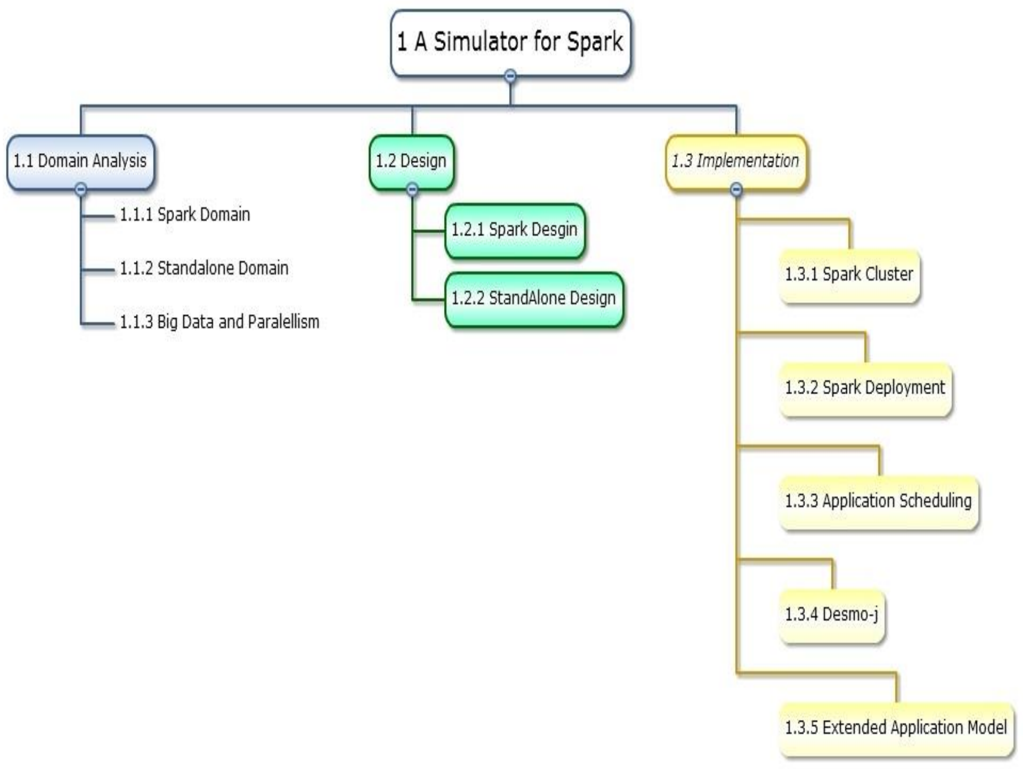
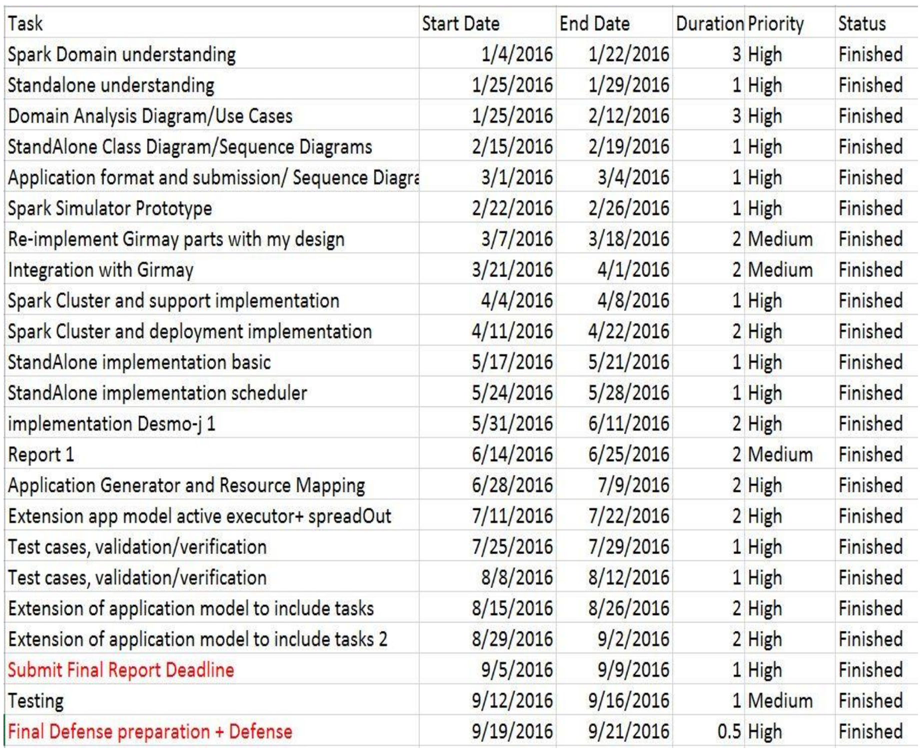


Figure 18 Work Breakdown Structure

11.3 Project Planning

A project plan was devised earlier on in the project. The project plan was based on the work breakdown structure presented in the previous section. The initial project plan was however changed and went through a lot of iterations. This is due to the fact that the project followed an incremental approach and there were also a lot of changing requirements. Only the final project plan is presented in this section. The initial project plan can be found in the appendices.



Task	Start Date	End Date	Duration	Priority	Status
Spark Domain understanding	1/4/2016	1/22/2016	3	High	Finished
Standalone understanding	1/25/2016	1/29/2016	1	High	Finished
Domain Analysis Diagram/Use Cases	1/25/2016	2/12/2016	3	High	Finished
StandAlone Class Diagram/Sequence Diagrams	2/15/2016	2/19/2016	1	High	Finished
Application format and submission/ Sequence Diagra	3/1/2016	3/4/2016	1	High	Finished
Spark Simulator Prototype	2/22/2016	2/26/2016	1	High	Finished
Re-implement Girmay parts with my design	3/7/2016	3/18/2016	2	Medium	Finished
Integration with Girmay	3/21/2016	4/1/2016	2	Medium	Finished
Spark Cluster and support implementation	4/4/2016	4/8/2016	1	High	Finished
Spark Cluster and deployment implementation	4/11/2016	4/22/2016	2	High	Finished
StandAlone implementation basic	5/17/2016	5/21/2016	1	High	Finished
StandAlone implementation scheduler	5/24/2016	5/28/2016	1	High	Finished
implementation Desmo-j 1	5/31/2016	6/11/2016	2	High	Finished
Report 1	6/14/2016	6/25/2016	2	Medium	Finished
Application Generator and Resource Mapping	6/28/2016	7/9/2016	2	High	Finished
Extension app model active executor+ spreadOut	7/11/2016	7/22/2016	2	High	Finished
Test cases, validation/verification	7/25/2016	7/29/2016	1	High	Finished
Test cases, validation/verification	8/8/2016	8/12/2016	1	High	Finished
Extension of application model to include tasks	8/15/2016	8/26/2016	2	High	Finished
Extension of application model to include tasks 2	8/29/2016	9/2/2016	2	High	Finished
Submit Final Report Deadline	9/5/2016	9/9/2016	1	High	Finished
Testing	9/12/2016	9/16/2016	1	Medium	Finished
Final Defense preparation + Defense	9/19/2016	9/21/2016	0.5	High	Finished

Figure 19 Project Plan

The initial plan was just an estimation and it was changed many times. There was no change in the domain analysis, problem analysis or the design phase but there were a lot of changes in the implementation phase. This is because the client changed the requirements many times. The presence of a generic design meant that there were not a lot of changes in the design. The plan shows a specific time slot for report but the report writing was a continuous activity. The report was written throughout the project and feedback was received from the supervisors.

Initially, the plan was to implement multiple cluster managers in simulator but later it was changed to only focusing on the standalone cluster manager. The reason was that it was not possible to complete multiple cluster managers in the given amount of time and the client wanted to have more detail in the standalone cluster manager. The presence of a generic design will make it easier to add new cluster managers in the future.

The first month was spent on reading literature and getting the high level requirements from the stakeholders. The high level requirements and the understanding of the domain was confirmed by showing use case and analysis diagrams to the stakeholders. Initially, there was a meeting every week with the stakeholders to get a clearer picture of the domain and to make sure I have the correct understanding. There was also a monthly project steering group meeting where all of the stakeholders were present. The project steering group meeting was to give the current status of the project to everyone and to make sure everyone was on the same page. There was an evaluation meeting every three months to make sure to give me feedback on my progress. After every meeting a detailed document of the meeting minutes was sent to everyone.

The second month was the formal beginning of the design phase. A design was already being created in the first month but it was more at the analysis class diagram phase, meaning the diagram at that time only focused on entities in the problem domain and did not focus on the solution. From the second month, the analysis diagram started evolving into a Class diagram. The Class diagram now also contained entities that comprise the solution to the problem. As the design became more mature, the implementation phase also began with the design. With every new design, there was a prototype of the design. The prototype helped the customers decide whether the design and implementation are correct or there should be some changes in the design.

The final few months were completely focused on the implementation with only minor changes in design. As the requirements were now fixed so there was no need to make a lot of changes in the design. The design was also generic enough to allow quick changes and additions to the existing code.

The project plan was helpful because it limited the scope of the project. The project plan, along with the feasibility study helped guide the scope of the project and determined what was possible to be implemented in the fixed time duration of 9 months. The project plan was also helpful for the customers as they could compare the actual progress of the project to the plan and see how things were progressing.

12. Project Retrospective

In this chapter, I look back at the project and how it progressed. I revisit the biggest challenges faced during the project and the design criteria chosen for the project.

12.1 Introduction

The Project was carried out in a timespan of nine months and it consisted of both familiar and unexpected challenges. Before beginning the project, I was already familiar with concepts such parallelism, multi-threading and process scheduling. On the other hand, I was not so familiar with concepts such as big data, Map reduce, Discrete Event Simulation and technologies like Spark. The Spark domain is very large and complex and it was a big challenge to understand the domain and translate the requirements into a final product.

The biggest challenge in the beginning was to understand the project domain. Incorrect understanding of the domain would lead to an incorrect simulation of Spark. Most of the time in the first two months was spent on understanding the domain. I tackled this challenge by making analysis diagrams of the problem domain. This made it easier for me to have a better understanding of the system and to communicate my understanding of the system to the clients. These analysis diagrams slowly evolved into the final design of the system.

The entire project was a great learning experience for me. I not only got to practice and improve my design skills but I also improved my communication skills as a result of the project. The project taught me the importance of a good design because there were a lot of frequent changes in the project requirements and having a good design made it easier for me to implement those changes.

12.2 Design opportunities revisited

In this section, I present the design criteria selected for the fulfillment of the project.

12.2.1. Functionality

The project succeeded in realizing this design criteria because it provides a working simulator that was evaluated by the client. Experiments conducted during the project also confirmed that the requirements presented earlier have been implemented successfully by the project.

12.2.2. Genericity

The design produced during the project is generic enough to easily change and extend. This was proved several times during the project when the client changed requirements and the time to implement the change was really short. The design is also generic enough to be easily extended. The reason is that the design is highly cohesive and there is minimal coupling between different entities. Existence of design patterns such as strategy pattern make it easy to switch different configurations and policies.

12.2.3. Complexity

As we discussed earlier, the project belongs to a complex domain. The challenge was to balance between the “Complexity” and “Genericity” design criterion. The simulator should be generic enough to be easily extended but is should be concrete enough to capture the complex functionality of Spark. The project succeeded in producing a simulator that captured enough of the functionality and complexity of Spark that it fulfilled the requirements of the client.

Glossary

This section shows the terms used throughout the report and the definition of those terms.

Table 2 Glossary

Term	Definition
PDENG	Professional Doctorate of Engineering
ST	Software Technology
SQL	Structured Query Language
RDD	Resilient Distributed Dataset
Desmo-j	Discrete Even Simulation Modelling in Java
UML	Unified Modelling Language
JSON	Javascript Object Notation
XML	Extensible Markup Language
HDFS	Hadoop Distributed File System
SAN	System Architecture and Networking Group
Application	A user program that is built on Spark
Driver	A process that contains the main () function of the Application.
Cluster	A collection of computers connected together by a network.
Worker	A virtual node that can run code inside the physical cluster
Executor	A Java virtual machine launched for an application on a worker. The executor is owned by a single application and it runs tasks and stores data for that application.
JVM	Java Virtual Machine
Task	A unit of work that is sent to the executor JVMs for execution
Job	A job is a parallel computation that consists of a collection of tasks that are grouped together in stages.
Stage	A job is divided into a set of tasks that are dependent on each other.

References

[1] Facebook, "Facebook," [Online]. Available: http://www.Facebook.com. [Accessed 1 June 2016].

[2] Whatsapp, "Whatsapp," [Online]. Available: http://www.whatsapp.com. [Accessed 1 June 2016].

[3] Twitter, "Twitter," [Online]. Available: http://www.twitter.com. [Accessed 1 June 2016].

[4] IBM, "IBM Webpage," [Online]. Available: https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html. [Accessed 10 August 2016].

[5] A. B. V. B. Marcel Kornacker, "Impala: A Modern, Open-Source SQL Engine for Hadoop," in *7th Biennial Conference on Innovative Data Systems Research*, California, 2015.

[6] T. White, Hadoop: The Definitive Guide, 3rd Edition, O'Reilly.

[7] R. A. T. D. Sean Owen, Mahout in Action.

[8] D. L. ,. R. S. Claudio Martella, Practical Graph analytics with apache giraph, 2015.

[9] A. Hama, "Apache Hama," [Online]. Available: https://hama.apache.org/. [Accessed 2 2 2016].

[10] Hadoop, "Hadoop hdfs," [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. [Accessed 1 July 2016].

[11] M. C. M. J. F. S. S. I. S. Matei Zaharia, "Spark: Cluster Computing with Working Sets," University of California, Berkeley.

[12] Apache, "Spark Webpage," [Online]. Available: http://spark.apache.org/. [Accessed 15 february 2016].

[13] Apache, "Spark Webpage," [Online]. Available: http://spark.apache.org/docs/latest/programming-guide.html#resilient-distributed-datasets-rdds. [Accessed 2016 3 July].

[14] P. Cassandra, "Spark Cassandra," [Online]. Available: http://www.planetcassandra.org/blog/kindling-an-introduction-to-

spark-with-cassandra/. [Accessed 1 June 2016].

[15] S. B. C. M. M. Karnon, "Modeling using Discrete Event Simulation," *Elsevier: Value in health,* pp. 821-827, 2012.

[16] Apache, "Apache Mumak," [Online]. Available: https://issues.apache.org/jira/browse/MAPREDUCE-728. [Accessed 2 2 2016].

[17] S. A. a. N. Group, "SAN," [Online]. Available: https://www.win.tue.nl/san/. [Accessed 10 August 2016].

[18] Buttazo, "Predictable scheduling algorithms and applications," in *Predictable scheduling algorithms and applications*, 2011, p. 27.

[19] N. d. reference, "Your ultimate guide to non-relational universe," [Online]. Available: http://nosql-database.org/. [Accessed 7 March 2016].

[20] Apache, "The hive language manual," [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/LanguageManual. [Accessed 9 March 2016].

[21] Apache, "Machine Learning Library," [Online]. Available: http://spark.apache.org/mllib/. [Accessed 9 March 2016].

[22] Apache, "Graphx," [Online]. Available: http://spark.apache.org/graphx/. [Accessed 9 March 2016].

[23] StackOverflow, "StackOverflow," [Online]. Available: http://stackoverflow.com/questions/25836316/where-to-learn-how-dag-works-under-the-covers-in-rdd. [Accessed 15 August 2016].

[24] P. Kruchten, "Architectural Blueprints—The “4+1” Model of Software Architecture," *IEEE Software,* pp. 42-50, 1995.

[25] T. Point, "Tutorials Point," [Online]. Available: http://www.tutorialspoint.com/design_pattern/strategy_pattern.htm. [Accessed 6 July 2016].

[26] U. o. Hamburg, "Desmo-j Home page," [Online]. Available: http://desmoj.sourceforge.net/home.html. [Accessed 1 June 2016].

[27] T. Point, "Tutorials Point," [Online]. Available: http://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm. [Accessed 2 June 2016].

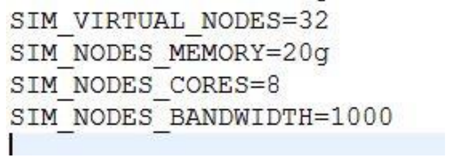
Appendix

Configuration files

This section contains a snippet of each configuration file of the simulator and explanation of the most important parameters.

1). virtualNode

A snippet from the virtualNode configuration file is shown in figure 20.

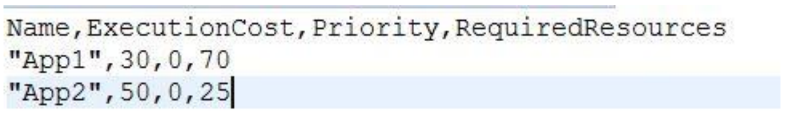


```
SIM_VIRTUAL_NODES=32
SIM_NODES_MEMORY=20g
SIM_NODES_CORES=8
SIM_NODES_BANDWIDTH=1000
```

Figure 20 Configuration of virtual node

The parameter "SIM_VIRTUAL_NODES" represents the total number of computing nodes in a cluster. "SIM_NODES_MEMORY" defines the total amount of RAM per node. "SIM_NODES_CORES" defines the total number of cores per node. The last parameter "SIM_NODES_BANDWIDTH" is supposed to represent the network bandwidth.

2). ApplicationListCSV

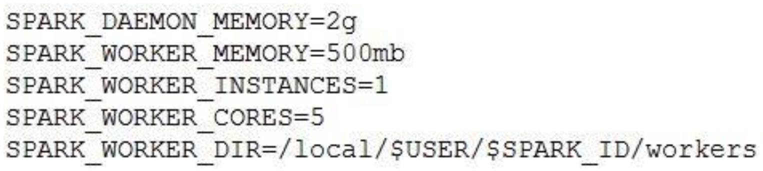


```
Name,ExecutionCost,Priority,RequiredResources
"App1",30,0,70
"App2",50,0,25
```

Figure 21 Configuration of applications

The application configuration file is shown in figure 21. "Name" represents the name of the application, "ExecutionCost" represents the execution time in seconds for the application. The application will hold the resources for the amount of time described in "ExecutionCost". "Priority" describes the priority of the application and "0" represents the highest priority. "RequiredResources" shows the number of cores required by the application.

3). Spark-env

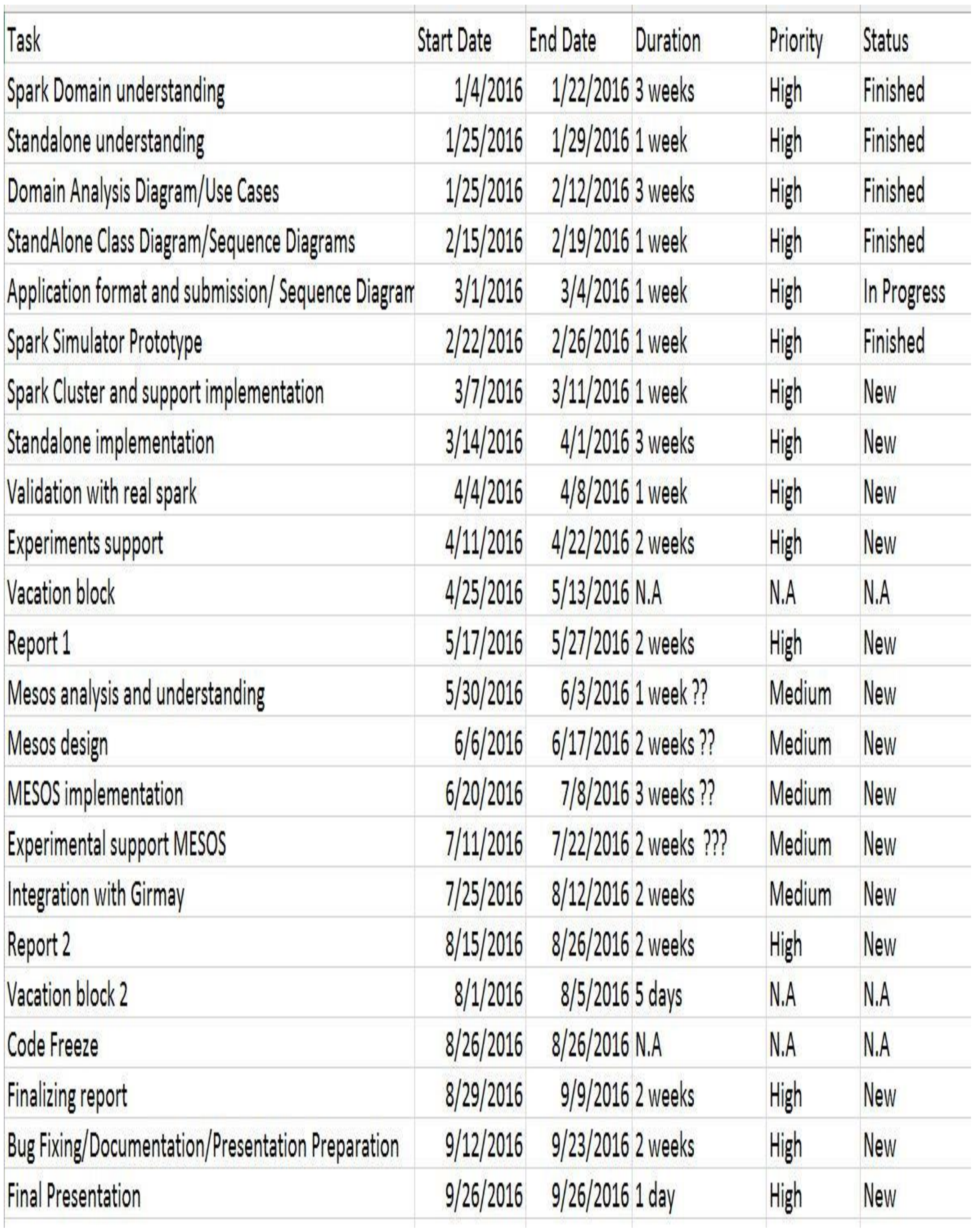


```
SPARK_DAEMON_MEMORY=2g
SPARK_WORKER_MEMORY=500mb
SPARK_WORKER_INSTANCES=1
SPARK_WORKER_CORES=5
SPARK_WORKER_DIR=/local/$USER/$SPARK_ID/workers
```

Figure 22 Configurtion of Spark

A snippet from the Spark configuration file is shown in figure 22. The parameter "SPARK_WORKER_MEMORY" shows the amount of memory allocated to each worker. "SPARK_WORKER_CORES" shows the number of cores per worker. "SPARK_WORKER_INSTANCES" shows the number of workers per node. The other two parameters in the picture are not being used by the simulator at the moment.

Initial plan



Task	Start Date	End Date	Duration	Priority	Status
Spark Domain understanding	1/4/2016	1/22/2016	3 weeks	High	Finished
Standalone understanding	1/25/2016	1/29/2016	1 week	High	Finished
Domain Analysis Diagram/Use Cases	1/25/2016	2/12/2016	3 weeks	High	Finished
StandAlone Class Diagram/Sequence Diagrams	2/15/2016	2/19/2016	1 week	High	Finished
Application format and submission/ Sequence Diagra	3/1/2016	3/4/2016	1 week	High	In Progress
Spark Simulator Prototype	2/22/2016	2/26/2016	1 week	High	Finished
Spark Cluster and support implementation	3/7/2016	3/11/2016	1 week	High	New
Standalone implementation	3/14/2016	4/1/2016	3 weeks	High	New
Validation with real spark	4/4/2016	4/8/2016	1 week	High	New
Experiments support	4/11/2016	4/22/2016	2 weeks	High	New
Vacation block	4/25/2016	5/13/2016	N.A	N.A	N.A
Report 1	5/17/2016	5/27/2016	2 weeks	High	New
Mesos analysis and understanding	5/30/2016	6/3/2016	1 week ??	Medium	New
Mesos design	6/6/2016	6/17/2016	2 weeks ??	Medium	New
MESOS implementation	6/20/2016	7/8/2016	3 weeks ??	Medium	New
Experimental support MESOS	7/11/2016	7/22/2016	2 weeks ???	Medium	New
Integration with Girmay	7/25/2016	8/12/2016	2 weeks	Medium	New
Report 2	8/15/2016	8/26/2016	2 weeks	High	New
Vacation block 2	8/1/2016	8/5/2016	5 days	N.A	N.A
Code Freeze	8/26/2016	8/26/2016	N.A	N.A	N.A
Finalizing report	8/29/2016	9/9/2016	2 weeks	High	New
Bug Fixing/Documentation/Presentation Preparation	9/12/2016	9/23/2016	2 weeks	High	New
Final Presentation	9/26/2016	9/26/2016	1 day	High	New

About the Author

Sarwan Dar received his Bachelors degree in Computer Science from Punjab University College of Information Technology, Pakistan, in 2011. He received his Masters in Computer Science from the same university in 2014. During his education he got multiple scholarships from the university. His Masters thesis focused on application partitioning for multi-core systems. The algorithm received a serial program and converted it into a parallel program based on the data dependencies between tasks. The algorithm at the time had the lowest time complexity of all the application partitioning algorithms. During his education he also worked as a software engineer for 2 years working on mobile app development and desktop applications.

In September 2014 he joined Eindhoven University of Technology and worked as a PDENG trainee in Software Technology. During this time he did projects for companies such as CERN, Océ, Nxp and Benteler. During his final project, he worked on the design and implementation of a Spark Simulator. The results of that project are discussed in this report.

4TU.School for Technological Design, Stan Akkermans Institute offers two-year postgraduate technological designer programmes. This institute is a joint initiative of the four technological universities of the Netherlands: Delft University of Technology, Eindhoven University of Technology, Universty of Twente and Wageningen University. For more information please visit: www.4tu.nl/sai