# High availability for RoQ core components (Appendix A)

*Master's thesis in collaboration with EURA NOVA for the graduation of Master in Computer Science option Software engineering by Benjamin Van Melle*

*Promoter:* Peter Van Roy
*Copromoter:* Alexandre D'Erman
*Reader:* Richard Gil Martinez

Université Catholique de Louvain-la-Neuve
Belgium
Academic year 2014 - 2015

# Appendices

# Appendix A

# Spark

## A.1 Introduction

After the success of the Hadoop map/reduce framework to make computation over big sets of data, the Spark team began to think about its weaknesses. The most important one is its inefficiency to run cyclic data flows (see figure A.1). Therefore, Hadoop map/reduce is not an efficient solution to run such types of applications (e.g. many machine learning algorithms).

This part of my thesis presents Spark [1], a batch processing framework inspired from map/reduce and developed to run cyclic data flows efficiently. The solution is based on a powerful abstraction, the *resilient distributed datasets* (RDD), which is defined by the authors of [1] as: “a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost”.

The first section presents the components of Spark. Then we present the shared variables and the resilient distributed dataset. The third section is dedicated to the fault tolerance. Finally, we have a look of the solution to make the cluster manager highly available.

## A.2 Architecture overview

Spark is made of three fundamental components [2], the *driver program*, the *cluster manager* and the *worker nodes* (see figure A.2).

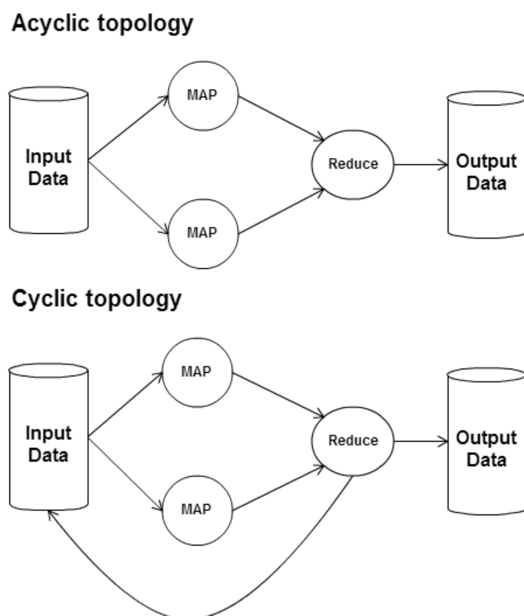


Figure A.1: A cyclic and an acyclic data flow

### A.2.1 Driver program

The driver program is written by the user and runs on the client machine. This one is made of a *SparkContext*. The SparkContext is responsible to run the user's program by assigning tasks to the workers. These tasks - which compose the data flow - are scheduled on the executors.

### A.2.2 Cluster manager

The cluster manager acquires the executors and provides them to the Spark-Context but for the duration of the application. An executor represents a pool of resources that can be used to run tasks. Spark proposes its own cluster manager, it also supports third party cluster managers such as Mesos and Yarn. The user can choose the most appropriate cluster manager for their application. For instance, Yarn offers the following advantages over the standalone cluster manager [3]:

- Since Yarn takes into account the tasks planned by the other applications on the cluster, it can allocate the resources efficiently and fairly.

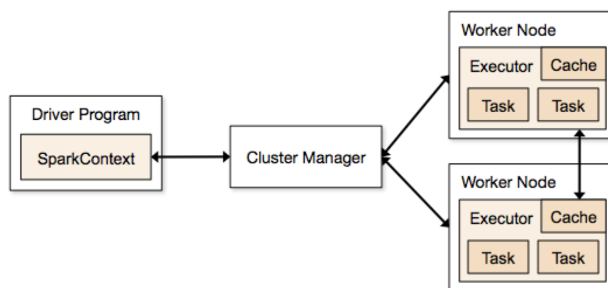


Figure A.2: Spark overview architecture, taken from [1]

- The Yarn scheduler is sophisticated and allows to prioritize, categorize and isolate the tasks.
- Yarn allows to specify the number of executors for each worker in the cluster.
- Yarn proposes security features to authenticate the connections among the nodes in the cluster.
- Yarn allows to run the Driver program on the cluster instead of on the client.

It is interesting to take into account these technologies when running a Spark cluster, because they can improve the system by delegating cluster management to specialized technologies.

### A.2.3 Workers

The worker nodes run concurrently the tasks assigned by the SparkContext. The code of these tasks has been previously sent by the driver to the executors. A task is an operation such as filter, mapper, reducer, etc. These tasks run in the context of an executor.

## A.3 Shared variables

Because workers are independent entities in the Spark architecture, we need a way to share some data among them. The solution is to use shared variables.

These variables are created by the user and copied on each worker. Spark provides two types of shared variables:

- **Broadcast variables**: This is a large read-only dataset which is stored on each worker in order to be used by many tasks. By making a shared variable with this data, we avoid the overhead due to the transfer of it to each task. These variables are distributed to the workers by the SparkContext (via a broadcast algorithm) and can be then used by the workers' tasks.

- **Accumulators**: This variable can only be modified (by the workers) by applying a commutative operation such as a "+=" (e.g. a counter). Only the driver program can read this variable.

## A.4 Resilient distributed dataset

Resilient Distributed Dataset(RDD) [4] is a distributed memory abstraction, this structure is at the origin of the performance of Spark on cyclic data flow. RDD is an immutable dataset which is distributed on nodes. Each node handles a partition of the dataset. RDD is fault tolerant, thanks to its ability to rebuilt a partition if one of them is lost. A key difference with Hadoop map/reduce is that data is not stored in physical memory in order to avoid the high latency access implied by distributed file systems such as HDFS. In Spark, a RDD is represented by a Scala object and can be created in different ways:

- By transforming a file.
- By dividing an array in slices. These slices can be handled concurrently.
- By transforming a RDD via operations such as flatmap or filter.
- By editing the persistence of the RDD (explained below).

RDD relies to a lineage mechanism for their fault tolerance. The lineage is the series of operations applied to thee original RDD in order to obtain the current set of data (see figure A.3). These operations can be replayed to rebuild lost partitions. In this way, RDD is fault tolerant.

The transformations applied to the RDD are lazy. Actually, the operations on the dataset are not computed directly. Spark just builds the RDD

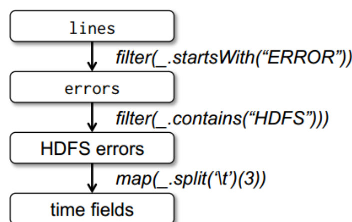


Figure A.3: RDD lineage, taken from [4]

lineage. A dataset is only computed when it has to be returned to the driver program. By default, the dataset must be recomputed at each time where you need it.

To avoid recalculating RDD multiple times, Spark provides caching operations.

The operations on the RDD are applied "exactly once". In others terms, re-computation leads always to the same result. Therefore, if a worker fails, the result will be the same.

## A.5 Fault tolerance

### A.5.1 Workers

The workers are designed to be fault-tolerant. We saw that the RDD makes its partitions fault tolerant via lineage. Therefore, the only dataset to don't lose is the input dataset because if we loose it, we are not able to rebuild a lost partition.

An *input dataset* is the original dataset provided to the program. This one can be submitted to Spark in two ways:

1. Using HDFS files. The HDFS guarantees the persistence of files. In this way, we cannot lost the input dataset.

2. Using a log system (such as Kafka or Flume). The input data received through these technologies is replicated on different nodes called *receiver nodes*. In this way, we can tolerate $n$ crashes without loosing the input dataset ($n$ is the number of receiver nodes).

## A.6 High availability

### A.6.1 Standalone cluster manager

By default the standalone scheduler (provided by the Spark cluster manager) is a single point of failure[2]. This is a master node which takes the scheduling decisions about the worker nodes such as jobs affectation, etc. Spark proposes two solutions to improve the availability of this component.

- **Standby Masters with ZooKeeper** This first solution is to use the leader election feature of ZooKeeper. In this way, we deploy multiple masters on the cluster and we elect one of them as the active master while the others are in standby. When the active master fails, one of the standby nodes is elected and recovers its state before taking next scheduling decisions.

  In order to register an application or add a worker to the cluster, a SparkContext needs to know the master address. To achieve this goal, we have to pass the masters address list to the SparkContext which will find the active one. If the the active master node fails, the new elected node will contact the applications and workers to inform them of the change.

- **Single-Node Recovery with Local File System** This second solution is less attractive in a production environment. This one involves to keep the master state in a file system to be able to recover it after a crash. This solution could be preferable than doing no recovery at all, but regarding to the mean time to recovery this is not acceptable for a system which aim to be highly available.

## A.7 Conclusion

We presented Spark, a framework to compute big datasets efficiently and which tackles the issues of Hadoop Map/Reduce thanks to is ability to deal with cyclic data flow efficiently. Spark relies widely on Resilient Distributed Dataset, a powerful abstraction which allows the system to be fault tolerant and to improve its efficiency by avoiding to use a distributed file system (such as HDFS). In addition, this abstraction thanks to its mechanisms (lazy evaluations and lineage) allows to provide an exactly once semantic.

# Bibliography

[1] M. J. F. S. S. I. S. Matei Zaharia, Mosharaf Chowdhury, “Spark: Cluster computing with working sets,” 2010.

[2] “Spark documentation.” https://spark.apache.org/docs/latest. [Online; accessed 10-Maart-2015].

[3] S. Ryza, “Apache spark resource management and yarn app models.” http://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/, 2014.

[4] T. D. A. D. J. M. M. M. M. J. F. S. S. I. S. Matei Zaharia, Mosharaf Chowdhury, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” 2012.