Design and Implementation of a MapReduce Framework

Azim Afroozeh, Qiru Tao, Wesley Geniz Shann, You Hu
{a.afroozeh, q.tao, w.genizshann, y.hu}@student.vu.nl

***Abstract*—The increase in data volume led to the need for more sophisticated data processing techniques. In the last decade, the MapReduce was introduced to resolve such tasks, by executing distributed computation in a huge volume of data. The framework was a success and is still relevant in the current days. Playing a key role in this scenario, distributed systems evolved to be essential in today society. Therefore, in this project, we seek to understand some of the key properties of distributed systems by designing and implementing a simple MapReduce framework. Overall, the implementation provides nice levels of scalability, however, it does not have a golden rule to achieve it regardless of the input data. This report presents the first version of a simple MapReduce framework based on the original MapReduce paper.**

I. Introduction

DATA have been growing fast in the last decades, it became large enough to the point that it not possible to handle this large amount of data with traditional techniques [1], [2].

There were attempts to address this problem in the past 30 years. Two solutions that stand out are the MapReduce and Spark.

Introduced by Google in 2004, MapReduce is a programming model designed to support parallel computing in large collections of data in computer clusters [3]. It allowed the possibility of processing a large amount of data of any kind with the implementation of only two functions, a function to map the input, generating an intermediate result and a function to reduce this result to the final output. As such, the community have accepted this solution with a highly positive response and multiple researches on top of the MapReduce were made since then, in December of 2018, according to the Google Scholar platform, the total number of citations of the paper accounted to 26048 times [4]. Today there are multiple implementations of MapReduce available, a few examples are Apache Hadoop[1], Apache CouchDB[2], Disco Project[3], Infinispan[4] and Riak[5], all open source.

Presented by Team in 2012 is the Resilient Distributed Datasets (RDD), which was implemented in Spark. As defined in their paper, an RDD is "a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner" [5]. They proposed it to address a few items that were inefficient in the MapReduce, such as the need for saving the intermediate results. The community have also positively received this solution, with the paper having a total of 3350 citations at the end of December, 2018 [6]. Spark is available in Apache Spark[6] and Databricks[7].

The objective of this work is to design and develop a distributed system, understanding all the individual components that comprise such systems. To achieve this, we choose to implement a simple MapReduce framework, as it has a simple and nice design, while including several important concepts of distributed systems, such as scheduling, fault tolerance, communication and scalability.

There are a number of problems that can be solved using the MapReduce framework, such as: Counting each word frequency in a given text; distributed grep; distributed sort and counting the URL access frequency in a given log of web pages requests [3].

This report is organised with the following structure: In Section II we describe with more details the background of the proposed MapReduce framework and its requirements. The framework features, architecture and components are described in Section III. The experiments executed to test the performance, functionality and quality of the application are specified in Section IV, along with the obtained results. In Section V we discuss these results and the proposed framework altogether. Lastly, our final considerations and improvement suggestions are presented in Section VI.

II. Background on Application

The application will run on Amazon EC2 [8] and it must support computation of large volume of data. Users will have a few actions only. A map function, a reduce function and, at least, one input file must be provided to the application. Multiple input files can also be accepted. After running the MapReduce task, the user will receive multiple reduce files. In summary, there are 4 major steps in the application: (1) read the input files and split it to the mapper nodes; (2) run the map phase; (3) after all the map jobs are finished, run the reducer phase and (4) store the results.

This report was submitted for the lab assignment of the 2018-2019 edition of the Vrije Universiteit Amsterdam, Computer Science Master Degree, Distributed System course, under super- vision of prof. dr. ir. Alexandru Iosup (a.iosup@vu.nl), dr. Animesh Trivedi (animesh.trivedi@gmail.com) and ir. Laurens Verluis (l.f.d.versluis@vu.nl

[1] https://hadoop.apache.org/
[2] https://couchdb.apache.org/
[3] http://discoproject.org/
[4] http://infinispan.org/
[5] http://basho.com/products/#riak
[6] https://spark.apache.org/
[7] https://databricks.com/
[8] https://aws.amazon.com/ec2/

Since the purpose of the MapReduce framework is to process a large amount of data, it is important that the solution is scalable. A sufficient number of nodes should be available be able to execute the task independently of the size of the data. Furthermore, the application must be able to efficiently split the task among the available nodes. As the reducer phase can only start executing after the mapper phase is completed, if a mapper node receives significantly more jobs than the others mapper nodes, then the application will be inefficient keeping several idle resources.

Fault tolerance and recovery are also an important aspect that the application must be able to provide in the case of nodes failure. If any node fails, the application should be able to recover from the fail without stopping the execution. The jobs that were assigned to the failed node should be split among other nodes. For the first version of the framework, fault tolerance and recovery is only provided for the worker nodes (maps and reduces jobs), while the master node is considered to not fail.

A master node will be responsible to automatically manage all these tasks internally. Consequently, all these operations will be transparent to the user, who does not need to be concerned with how the input files will be split among the nodes, failures, etc. The user only needs to be concerned in providing the input files, map and reduce functions. Thus, making the application simple and easy to use.

These requirements are essential. While fulfilling them, the framework should have an acceptable level of performance, as the number of worker nodes varies, data volume increases or decreases and in the presence of failure. The acceptable level of performance is discussed in sections IV and V.

III. System Design

Most of the system design was inspired by the original MapReduce implementation, with slight modifications to present a simplified MapReduce framework. The framework was implemented using the programming language Python[9] version 3.7 and is publicly available in a GitHub repository[10].

To give a simpler view of the implementation, we selected the most important parts of the code and separated them in a directory[11] in the root. For each of the relevant features in this section, a link to the implementation is given.

A. Architecture

The MapReduce is implemented using a centralized Master-Worker architecture, as illustrated in Figure 1. The architecture is essentially the same as introduced by the original MapReduce paper in [3]. There are 3 major components in this architecture:

- The **master node** are responsible to manage all the components and make sure that everything is running as it should be. It ensures that the input and intermediate files are correctly sent to the map and reduce nodes, as well as the functions themselves. The master node also guarantees the execution of the application even in the presence of failures.
- The **worker nodes** are responsible to execute all the computation job. In this framework, there are two types of worker nodes, the mappers and reducers. The mappers are responsible to read the piece of the input file assigned to it, execute the map function and store the intermediate result in their local storage. Similarly, the reducers are responsible to read all the intermediate results, execute the reduce function and store the final result in the shared storage.
- The **client** is able to submit new MapReduce tasks and is responsible to provide a map and a reduce function and the input files path.

B. System Operation

As illustrated in Figure 1, the application flow can be described in 8 steps:

1) The user provides a map and a reduce function and submit a new MapReduce task to the master node.
2) The master node assigns the map and reduce functions to the respective map and reduce worker nodes.
3) The master split the input files into several partitions and assign them to the map nodes.
4) The map nodes receive the input and execute the map phase.
5) Once each map node completes the execution, it stores the intermediate result in their local storage, in the format $< key, value >$. In this stage, duplicated keys are stored.
6) After all map nodes have completed their execution, the master node notifies the reduce nodes to start their computation and inform the nodes where the intermediate result is stored.
7) When finishing the execution, each reduce node will write the result in a new output file, in the format $< key, totalvalue >$. In this stage, the keys are unique.
8) The user is notified by the master node that the MapReduce task has successfully complete.

A set of classes to handle the core of the flow are at[12].

C. Implementation Details

Job Scheduling: The map/reduce jobs scheduling to the worker nodes, is achieved by implementing a priority queue structure in the master node. The priority range is from one to five, being 5 the highest and 1 the lowest level of priority. The queue is sorted from the lowest to the highest priority. New map/reduce tasks are inserted with the lowest priority. In the case a task fail, its priority is increased. The scheduling can be found at[13]. The *SchedulerThread* will run on background. It checks the task queue and resource queue to verify if there are any new task and available resources. If there is, a task will be assigned to the available resource.

[9]https://www.python.org/
[10]https://goo.gl/BmrYAY
[11]https://goo.gl/rr81AM
[12]https://goo.gl/k9Fsuo
[13]https://goo.gl/yX37Ro

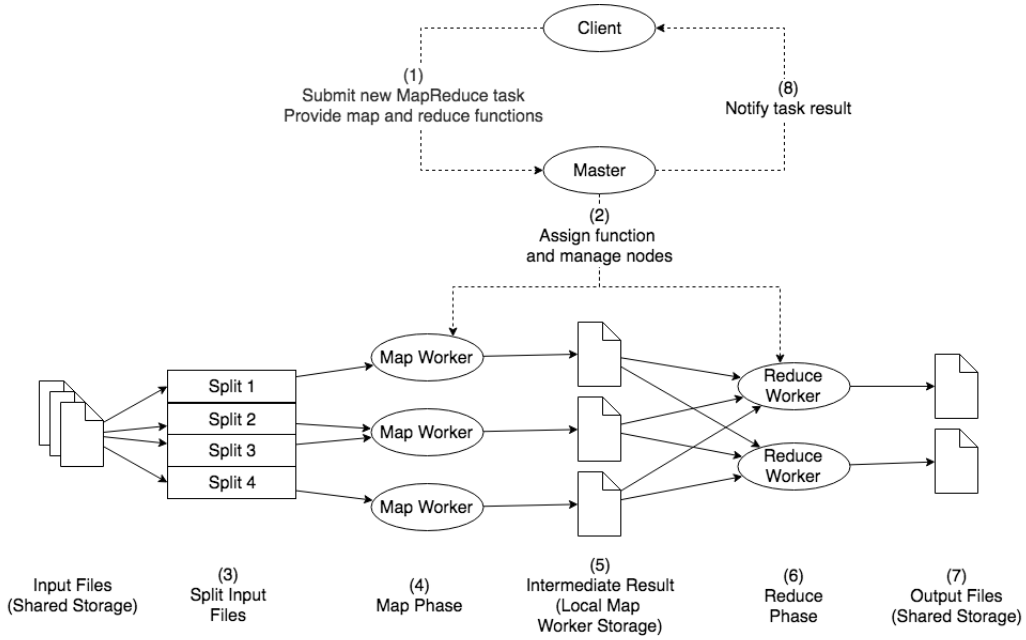

Fig. 1: MapReduce framework architecture. Adapted from [3].

Similarly, a priority queue to select the resource to assign the task to is also used. The load balancing is achieved through the use of both queues. With slightly alterations and improvements, the current implementation can be updated to be able to implement a dynamic resource detector,

Communication: The communication between the application nodes is designed with asynchronous Remote Procedure Call (RPC). This can be achieved by the use of the library RPyC[14] version 4.0.2.

Concurrency Control: A mutual exclusion (mutex) is set to a number of threads containing finished map jobs, to provide more secure access to it and so also be able to identify which map jobs have finished in a given moment.

Implementation is available at[15]. To determine the moment in which the map or reduce phase have completed, we have a variable that counts the number of finished tasks. For each variable, there are 2 threads sharing the access to this value. To avoid conflict when updating the variable value, we defined lock in n the implementation on the critical section that tries to update the value.

D. Faulty Tolerance

The detection of node failure is achieved by a simple gossip-based technique: the master node will periodically send a signal (heartbeat) to each worker node, which will answer the master to state their service availability. If the master does not receive an answer, it considers that the worker node has failed. [7]. There are two possibilities to identify failures in the master node. The first is to have a second node exchanging information with the master node. Both would send periodically signal to each other to indicate that they are running. The second option is to make use of a few randomly selected worker nodes to exchange information with the master node. In this case, these selected worker nodes would monitor the signals sent by the master node. If they do not receive any signal from the master node in a long period of time, they would conclude that the master node have failed. For all scenarios, the signal frequency is configured to be sent every 5 seconds.

There are four levels of fault recovery to be considered in a MapReduce framework:

1) Failure in a map node in the map phase: After identifying that a failure occurred in any map node, the master node will first identify the map jobs sent to this particular node, and then re-insert them in the job scheduling queue. When inserting again the jobs from a failed node in the priority queue, the priority of the jobs is increased. Once inserted in the queue, the scheduling component will handle the jobs.
2) Failure in a reduce node in the reduce phase: A recovery in this scenario would be similar as in the previous case.
3) Failure in a map node in the reduce phase: This is a tricky failure, due to the fact that the intermediate results from the map phase are stored in the local storage of the map node. Thus, a failure here would imply that the intermediate results are lost, consequently, this specific map phase would need to be computed again. This scenario recovery involves two steps. First, the map phase to the particular portion of the input needs to be executed again. Once the intermediate results are stored, the reduce nodes that required this file are re-executed.
4) Failure in a master node in any phase: Failures in the master node would need more sophisticated techniques. In this case, a backwards-error recovery technique could be used, for instance, coordinated checkpointing combined with pessimist logging protocol. The most relevant information to be logged would be the active map/reduce

[14]https://rpyc.readthedocs.io/en/latest/
[15]https://goo.gl/whXoTM

nodes and each node job.

For the first version of the framework, we opted to provide failure detection and recovery only for the worker nodes. Therefore, we currently do not provide a fault tolerance for the master node, assuming that this node will never fail.

The overall code to achieve fault detection can be verified at[16]. The *HeartBeatThread* thread also runs in the background. Every 5 sec it aks the group of workers with their service is available and executing. If any case of failure is identified, the task will be reassigned to new nodes. The designed MapReduce framework has 2 different versions for this service, one for map nodes and a second for reduce nodes.

E. Scalability

The main aspect of the framework scalability is the splitting of the files to the N worker nodes. By providing the liberty to configure the number of worker nodes, the framework allows a better of the resources according to the size of the files to be processed. If the size is relatively small, fewer resources can be used, whereas for a huge volume of data, more workers should be allocated for more efficient computation.

IV. Experimental Results

A. Experimental Setup

Workload: To run the experiments, several test files in ASCII format were created using an online tool to generate Lorem Ipsum[17] dummy text. Each file was generated with a random number of words, this number, was generated using the Google RNG[18] (Random Number Generator). For each dataset, a different range was used to generate a number, 10 to 500 for small, 501 to 2500 for medium, 2501 to 5000 for larger and 100000 for huge. However, due to the low size of the generated files resulting in no significative difference in the results, the content of each file was duplicated until reaching a reasonable size.

In Table I, is defined the amount and size of the files created. They were classified into four datasets: small[19], medium[20], large[21] and huge[22]; with the files size approximately of 500KB, 1MB, 10MB and 20MB, respectively. Each dataset contain 10 test files.

TABLE I: Testing datasets.

Dataset Name	Files Quantity	Average File Size	Smaller File Size	Larger File Size
Small	10	0.52 MB	0.37 MB	0.64 MB
Medium	10	1.05 MB	0.75 MB	1.3 MB
Large	10	10.04 MB	7.9 MB	12.5 MB
Huge	10	23.36 MB	21.0 MB	27.7 MB

[16]https://goo.gl/R1dbYt
[17]https://www.lipsum.com/
[18]https://www.google.com/search?q=rng&oq=rng
[19]https://goo.gl/Dnnsdt
[20]https://goo.gl/jrsNwM
[21]https://goo.gl/Pd8c3f
[22]https://goo.gl/cbBy7N

Testing application: We use two MapReduce applications to perform the experiments and test the implemented framework, both introduced in the original MapReduce paper: Word Count and Inverted Index [3].

- **Word Count**: Given a repository of text files, the map phase will parse each document and generate entries of the type *[word, 1]*. The reduce phase will, for each word, sum the total frequency count and generate the following output *[word, total count]*.
- **Inverted Index**: Given a repository of text files, the map phase will parse each document and generate entries of the type *[word, document ID]*. The reduce phase will, for each word, create an array with all the documents in which that particular word is present in the following format *[word, list of document IDs]*.

A briefly pseudo code for Word Count[23][24] and Inverted Index[25][26] applications is given next.

Listing 1: Word Count pseudo code.

```
# Implementation in word_count_map
map(file_split, worker_id){
  lines = read_file(file_split, worker_id)
  foreach word w in lines:
    write_count(w, 1)
}

# Implementation in word_count_reduce
reduce(workers, partition){
  foreach worker w in workers:
    values += data_intermed(partition)
  foreach value v in values:
    sum_count(w, 1)
}
```

Listing 2: Inverted Index pseudo code.

```
# Implementation in word_count_map
map(file_split, worker_id){
  lines = read_file(file_split, worker_id)
  foreach word w in lines:
    emit_word(w, file_split)
}

# Implementation in word_count_reduce
reduce(workers, part){
  foreach worker w in workers:
    values += data_intermed(partition)
  foreach value v in values:
    word_list += v
}
```

[23]https://goo.gl/565YdQ
[24]https://goo.gl/HQMEX8
[25]https://goo.gl/E41Niz
[26]https://goo.gl/wM5HD2

Testing Environment: To execute the experiments, the following setup was used:

- The master node resided in the client machine.
- The worker nodes resided in the remote Amazon machines.
- The testing was conducted using a variable number of worker nodes.
- Each worker node was running in a different EC2 instance.

The remote server configuration is described in the Table II.

TABLE II: Remote Amazon EC2 configuration.

Configuration	Value
Instance Type	t2.micro
IPV4	Unique by node
vCPU	1
Shared Storage	EFS
RAM	1 GB

Elastic File System (EFS)

Monitoring: The experiments monitorization was simple. In the experiments, the interesting information to register is the running time of the application. For that, no additional library or tool was required, the native library Time[27] was enough to handle the execution time. The time at the beginning of the execution was stored and subtracted from the time at the end of the execution. However, an important note regarding the execution time is that a delay time was introduced to the execution, as the master and worker nodes are not executed in the same network.

B. Experiments

System Operation: To test the correctness of the application, we used all datasets and both MapReduce application. We executed the dataset with a correct MapReduce algorithm to generate the correct output. Then we executed our implementation of the MapReduce framework and used the *diff* command to verify if there were any difference in the output files.

We verified that except for the huge dataset, all results were correct. An error occurred in the huge dataset in the reduce phase due limitation in the file size imposed by the libraries used.

Scalability: The framework scalability was tested using all described datasets, the word count application and varying the number of worker nodes. The running time was recorded to check how well the system executed under different workload size and resources available.

TABLE III: Map phase runtime in seconds.

Dataset	1 Node	2 Nodes	3 Nodes	4 Nodes
Small	16.867314	12.44766151	12.69698219	12.83730044
Middle	21.54382512	13.24713168	13.09334471	13.10187846
Large	83.68333106	47.5053001	35.4752633	29.57062249
Huge	174.0856512	98.77408644	81.05009178	63.92450104

[27]https://docs.python.org/3/library/time.html

In Table III is the average total execution time recorded in seconds for the map phase. Better visualization of the data is illustrated in Figure 2.

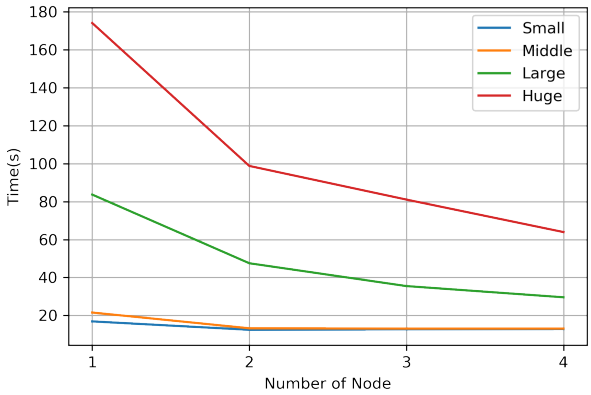

Fig. 2: Map phase runtime in seconds.

To illustrate how much faster the framework can execute MapReduce tasks using multiple nodes instead of just one, we calculate the speed up in terms of $N1/Ni$. $N1$ is the runtime with one node and Ni is the runtime with all of the tested nodes quantity, both available Table III. With the results, a speed up graph was created in Figure 3.

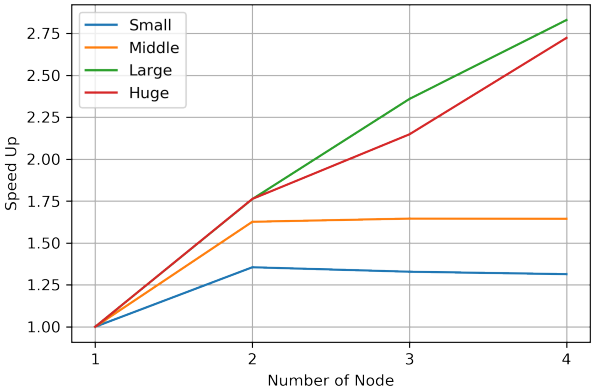

Fig. 3: Map phase runtime speed up.

There is no significant difference in the execution time when using more nodes for smaller datasets. In contrast, for larger datasets, the more nodes, the better the execution is. Although if we keep adding nodes, it will reach the point that there will be no significant difference as well for the larger datasets.

Fault Tolerance: To test the fault identification and recuperation, the small dataset and word count application were used. In particular, as mentioned in section III-D, we provide fault tolerance for all described scenario except the last (master failure). In this experiment, we manually fail a worker node process, then we verify and register what is the framework response in this situation, when the worker node stop receiving and sending messages.

The results were showed that the framework provide a good

fault tolerance for the tested scenarios. The framework was able to successfully identify the failure, get the jobs previously assigned to the failed nodes and schedule them again to other nodes and to compute again any necessary step.

V. Discussion

One important aspect of the MapReduce framework is to be able to identify the best quantity of worker nodes for the given input size. If the input is too small, using a large number of worker nodes will just add unnecessary overhead to split the files and schedule the jobs. In reality, for some small inputs, the utilization of the MapReduce framework is not even needed. The same logic applies for larger datasets. If the input is huge, using few worker nodes might not be very effective.

For the experiments realised in section III-D, if we continued the experiment for more nodes and larger datasets containing gigabytes to terabytes of data, the result would be the essentially the same: a curve with very high time to execute with fewer nodes, tending to get an almost constant runtime when using several nodes.

Therefore, there is no perfect number of words to use for all applications and datasets. This value should be configurated and adapted according to what the application demands.

One important observation to be done is that the fact that the Master node and Worker nodes are executed in different networks, surely influential the measured runtime results. Consequently, besides the nodes amount and dataset size, this also needs to be taken in consideration.

Regarding the framework correctness, the current implementation seems to fail when parsing datasets larger than 20 MB. Although it is able to correctly execute for smaller datasets, in real usage cases, the input is expected to have GB if not TB of information. Therefore, a revision in the current version of the framework to execute MapReduce tasks in larger datasets is essential.

Concerning the fault tolerance, as described in section III-D, there are four failure scenarios identified that can occur in a MapReduce task. To demonstrate a simple MapReduce framework, we have provided fault tolerance for the first scenario only. However, to be able to use the implementation for real-world usage case, a complete fault tolerance framework including all four described scenarios is absolutely required.

To be able to get better experiments, three points should be addressed. First, instead of simply generating Ipsum Lorem dummy text, search for two or more online data dump, such as the Pizza&Chili Corpus[28]. Secondly, the testing input size should be increased. In the presented experiments, we used inputs of a few KB and MB only, while in the real world, you can easily get data reaching TB. Thirdly, MapReduce application is not exclusively for text data, there are other applications, such as sorting and matrices operation. Executing the experiments with such applications would demonstrate a highly application diversity in the implemented framework. These three points, if accomplished, would lead to more interesting results and would bring the framework closer to real-world usages.

[28]http://pizzachili.dcc.uchile.cl/texts/nlang/

VI. Conclusion

This report described a simple MapReduce framework. Of the various characters of a distributed system, is given more importance to fault tolerance, scalability, scheduling and communication. Experiments prove that the MapReduce framework is scalable to a certain point, in which the application is not able to increase performance anymore, creating instead an undesired overhead in the communication and job scheduling. The experiments were, however, inconclusive regarding fault tolerance, due to the absence of a complete faulty tolerance implementation. Overall, the presented implementation is a prototype that has multiple points to be improved.

References

[1] M. Cox and D. Ellsworth, "Application-controlled demand paging for out-of-core visualization," in *Proceedings of the 8th Conference on Visualization '97*, VIS '97, (Los Alamitos, CA, USA), pp. 235–ff., IEEE Computer Society Press, 1997.

[2] S. Mukherjee and R. Shaw, "Big dataconcepts, applications, challenge sand future scope," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 5, pp. 66–74, Feb 2016.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, (San Francisco, CA), pp. 137–150, 2004.

[4] Google Scholar, "Jeff dean papers." Accessed at 13/12/2018. Available at https://scholar.google.com/citations?user=0KF6ZC8AAAAJ.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, (San Jose, CA), pp. 15–28, USENIX, 2012.

[6] Google Scholar, "Matei zaharia papers." Accessed at 13/12/2018. Available at https://scholar.google.nl/citations?user=I1EvjZsAAAAJ.

[7] M. R. v. Steen and A. S. Tanenbaum, *Distributed systems*. Pearson Education, Inc., 3 ed., 2017.

APPENDIX A
PROJECT SOURCE CODE

The source code for this project is publicly available in a Github repository[29] for anyone interested in contributing or developing their version of a MapReduce framework.

APPENDIX B
TIME SHEETS

In the Table IV is registered the approximate time spend in this project, splitted in 6 categories. The *design* on how to implement the application. The *development* of the framework implementation. The planning and execution of the *experiments*. The *analysis* of the obtained results. This *report* writing time. Lastly, *wasted* refers to any activity related to the experiment that don't fit in any of the other activities, for instance, selecting a LaTeXreport template and adjusting it as needed or communicating with the supervisors.

TABLE IV: Spend time by activity.

Activity	**Time (Hours)**
Design	25.30
Development	132.48
Experiment	11.57
Analysis	22.41
Report	23.05
Wasted	26.50
Total Time	238.01

[29] https://goo.gl/BmrYAY