# Byzantine Fault-Tolerant and Locality-Aware Scheduling MapReduce

**Andrea Graziani**
andrea.graziani93@outlook.it
Università Degli Studi di Roma Tor Vergata
Rome, Italy

## ABSTRACT

*MapReduce* is a programming paradigm developed by Google that enables massive scalability across hundreds or thousands of servers allowing to process very large data-set [3].

Both academic literature and daily experience shows that *arbitrary faults* frequently occur corrupting our results [1]. Moreover, ignore data-set locality can lead to performance degradation and a pointless bigger network traffic [2].

In this paper we present an original MapReduce runtime system capable both to tolerate arbitrary faults and to recognize input data network locations and sizes in order to perform a *locality-aware task scheduling*, improving performance and diminishing network traffic.

Although job's execution using our system requires more resources respect to other implementations, we believe that this cost is acceptable for critical applications that require an higher level of fault tolerance.

## KEYWORDS

MapReduce, Fault tolerance, Arbitrary failure, Data locality

## 1 INTRODUCTION

Various data-intensive tasks, like seismic simulation, natural language processing, machine learning, astronomical data parsing, web data mining and many others, require a processing power that exceeds the capabilities of individual computers imposing the use of a *distributed computing* [2].

Nowadays many famous distributed applications use MapReduce framework as solution for processing large data sets in a distributed environment. In order to properly provide services to an increasing number of users worldwide, is necessary to connect together thousands of computers and hundreds of other devices like network switches, routers and power units, moving consequently an huge amount of data between computers.

However, as many studies confirm, *hardware component failures are frequent* and they will probably happen more often in the future due to increasing number of computers connected to internet [1]. Is been documented that in the first year of a cluster at Google there were 1000 individual machine failures and thousands of hard drive failures [5]. A recent study of DRAM errors in a large number of servers in Google data-centres for 2.5 years concluded that these errors are more prevalent than previously believed, with more than 8% DIMM affected by errors yearly, even if protected by error correcting codes (ECC) [6]. A Microsoft study of 1 million consumer PCs shown that CPU and core chipset faults are also frequent [4]. Moreover moving large amount of data repeatedly among distant nodes is becoming the bottleneck owing to an increased network traffic, causing significant performance degradation.

Therefore to construct a distributed system capable to provide its services even in the presence of failures, that is to design a *fault tolerant* cloud application, is a critical aspect of software engineering.

Moreover, due to very large data-set involved during data-intensive tasks, exploiting data locality becomes crucial aspect in order to mitigate network traffic and delay, improving so performance.

In these paper we will describe an *Arbitrary Fault-Tolerant Locality-Aware* (AFTLA) MapReduce system which is capable to resolve all issues described above.

Our AFTLA MapReduce prototype, which is capable to perform a word-count by a given text input file, is been implemented from scratch using Go programming language using some external libraries, including those necessary to interact with an Apache ZooKeeper$^{TM}$ cluster[1], which we have adopt for system's coordination purposes, and with Amazon AWS services, because we have adopt Amazon S3 for storage; note that all libraries used in our prototype are fully described in table 1.

Our prototype is been design to run into several Amazon EC2 instances hosted in a same region (`us-east-1`)[2], adopting AWS Elastic IP[3] to associate some of our EC2 instance to an public IPv4 address in order to enable communication with the internet and then to connect a local computer to our system.

[1] https://zookeeper.apache.org

[2] Note that, due to Account AWS Educate Starter limitations, we could not run Amazon EC2 instances on other regions.

[3] https://docs.aws.amazon.com/en_us/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html

Finally, we remember that our prototype's source code is fully available on GitHub[4], famous web-based hosting service for version control using `git`; we remind that LaTeX source code of this paper is available too[5].

## 2 ASSUMPTIONS

In order to properly describe how our algorithm can tolerate arbitrary faults and to exploit data locality, it is necessary to describe its architecture. To do that, in this section we start making some very important assumptions.

Our system is composed by a *set of distributed processes*, every of which run on different hosts in a same data-center.

We assume that our system runs *asynchronously*, that is no assumptions about process execution speeds or message delivery times are made; therefore we can normally use timeouts to state that a process has crashed although, occasionally, such conclusion is false [7].

All processes are connected by *reliable channels*, so no messages are lost, duplicated or corrupted; that feature is guaranteed by TCP connections. In our prototype, all system's processes use RPC protocol in order to interact with each other.

Clients are *always correct*, because if they are not there is no point in worrying about the correctness of our system's output. Finally we assume the existence of an hash function that is *collisions-resistant*, for which it is infeasible to find two inputs that produce the same output.

In this paper, we take for granted reader's knowledge of the MapReduce framework and main aspects of Hadoop framework.

## 3 SYSTEM'S MODEL

As you can see from figure 1, our system is made up of several kinds of processes each of which has its own features and purposes.

### Client Process

This process, which, as said previously, is assumed always correct, is the place where (word-count) jobs are submitted to our system.

It can interact with system's key-value storage service (i.e. DFS or an equivalently cloud key-value storage service) for job's input uploading and job's output downloading using HTTP-based RESTful APIs, according to which both an URL and the use of standard HTTP methods (`GET` and `PUT`) only are required in order to perform data transmission between our cloud system and client

Is very important to specify that, in our system, URLs are provided by current leader primary process (CLPP) for security and compatibility reasons with some cloud storage services. In fact, unlike Hadoop, which use HDFS (*Hadoop Distributed File System*) to store input and output data, our prototype adopts Amazon S3 for data storage purposes. According to Amazon S3 implementation, since all objects and directories (latter are called *buckets*) are by default private, clients cannot upload or download data without AWS security credentials or permissions.

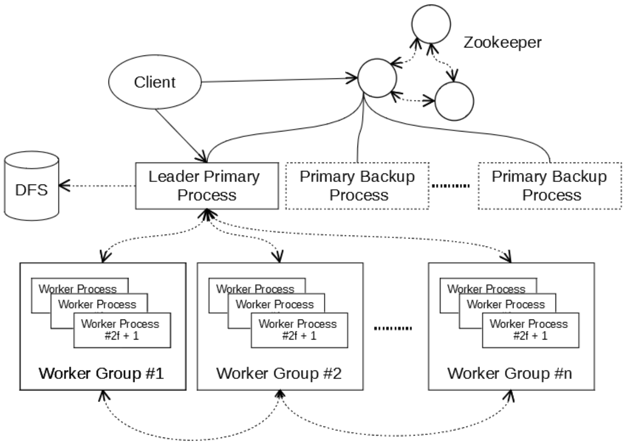

**Figure 1: System Architecture**

Therefore, rather than distribute AWS security credentials which can imply huge security risks, we have adopt a design according to which client processes have to request special URLs to CLPP to accomplish their data transmissions; these kind of URLs are called *pre-signed URLs*. According to Amazon AWS documentation, anyone who receives a valid pre-signed URL can upload (or download) an object from Amazon S3. However only someone that has a valid security credentials can create a pre-signed URL; this is the reason according to which CLPP, which has AWS security credentials to interact with Amazon S3[6], has to generate and distribute them to clients.

However clients process are unaware of CLPP network location which, as we will see in the next paragraph, can change during system activities due to either a crash or connectivity lost. To be more precise any system's process know network location of the others although Zookeeper cluster location, which we have adopt for membership managing and coordination purposes, must be known by everyone; therefore we can state that location transparency of our application is only partial.

[4] https://github.com/AndreaG93/SDCC-Project

[5] See https://github.com/AndreaG93/SDCC-Project-Report

[6] We remember that to give all privileges required for pre-signed URL creation to the EC2 instances where CLPP are running, we have to properly set all necessary IAM Roles thought "*AWS Identity and Access Management service*" (IAM)

**Table 1: Libraries used in our implementation**

| Name | Description | Link |
|---|---|---|
| `go-zookeeper/go` | Native Go Zookeeper Client Library | https://github.com/samuel/go-zookeeper |
| `logrus` | A structured logger for Go | https://github.com/Sirupsen/logrus |
| `aws-sdk-go` | AWS SDK for the Go programming language. | https://github.com/aws/aws-sdk-go |

Since, as we will see, is possible to know network location of a specified process querying Zookeeper cluster, clients are able to know where CLPP is both to require pre-signed URL and to submit jobs.

### Primary Process

At this point, is necessary to make some clarifications about PP's features.

Unlike Hadoop, according to which *JobTracker* process is assumed always correct, we have design our system believing that the host where PP is running may fail, for example by crashing or by losing network connectivity; in other words, as known in academic literature, PP represents for us a *single point of failure*.

Therefore, to ensure an high system availability, adopting the same design used in Apache Mesos$^{TM}$ [7], our system architecture is based on multiple PP backup copies run on *different machines* (in our prototype, on different EC2 instances); using a leader election algorithm, is possible to elect one of these copies as leader, becoming the *Leader Primary Process* (LPP), while all the others remain on stand-by. Using this approach, is possible to manage PP crash failure: when current LPP fails, a backup copy is promoted to become the new leader.

, only LPP can interact with WPs and modify data into the Zookeeper Cluster. Note that LPP schedules several task using a *push-based* approach (unlike Hadoop according to which JobTracker schedules task using a *pull-based* approach [2])

However, to perform its task, LPP manages various information about clients requests, many of which, as we will see later, are sent by WP after a map or reduce task. In other word, LPP has got a status.

Is very important to specify that LPP status is stored in memory, therefore they are permanently lost after a crash; this design makes our system easier to implement and helps to reducing overhead due to the disk I/O activities. However recover lost in-memory leader state after a failure is required in order to satisfy clients requests.

Therefore we have adopt a design according to which LPP make a sort of backup, or snapshot, of its current state regularly. To make our implementation easier, in our prototype, considering that LPP state contains a very small amount of data, LPP stores its state in a Zookeeper cluster after some events, like a map task termination. In case of crash failure of current LPP, when a primary process backup becomes leader, it retrieves all data from Apache ZooKeeper cluster, recovering last saved state.

### Worker Process and Groups

To make WP arbitrary fault-tolerant we have adopt a design according to which each task (map, reduce, etc.) is executed more than once by different processes running on different host. This approach requires to organize all our WPs into several *groups* through which became possible to mask one or more faulty processes of a given group.

Before explaining how we manage WP faults, we must explain what is a Worker Processes Group (WPG). A WPG is a set of identical worker processes, each of which execute, relatively to a given job, *the same commands using the same input data in the same order*; therefore we expected that all WPs belonging to a same group would produce the same output when no arbitrary fault occurs. All commands and data are sent by LPP.

Suppose to want tolerate at most $f$ arbitrary failures of all WPs belonging to a given group.

To achieve that level of fault-tolerance, we can apply to reach, for instance, *byzantine agreement* among all processes belonging to group. However we believe that solution is too expensive because it requires, as properly explain in literature[7], $3f + 1$ replicas.

Therefore we have adopt a protocol based on *majority voting* inspired to quorum-based protocols applied in replicated-write systems. As explained in literature[7], in this kind of systems, if processes exhibit arbitrary failures, continuing to run when faulty and sending out erroneous or random replies, a minimum of $2f + 1$ processes are needed to achieve $f$-fault tolerance. In the worst case, the $f$ failing processes could accidentally generate the same reply. However, the remaining $f + 1$ will also produce the same answer, so the LPP just believes the majority.

In our design each map or reduce task scheduled by current LPP are assigned to independent WPGs. Let's try to explain this statement formally using map task as example (but is the same of any task). Let $i = 1, 2, 3 \ldots n$, where $n \in \mathbb{Z}_+$.

[7] See http://mesos.apache.org/documentation/latest/architecture/

As MapReduce framework states, our system splits the input text file $S$ in $n$ parts $\{s_1, s_2, ...s_n\}$ called *split*; each split $s_i$ is associated to a map task $m_i \in \{m_1, m_2, ...m_n\}$ every of which must be replicated on multiple WPs to make them arbitrary fault tolerant. Each map task $m_i$ is assigned to a different WPG $g_i \in \{g_1, g_2, ...g_n\}$. As we will explain later, LPP, checking received outputs from each group, we can achieve arbitrary fault tolerance for every scheduled task. Obliviously to tolerate at most $f$ arbitrary failures of each task, each WPG must contains at least $2f + 1$ different processes. Then our design requires at least $n(2f + 1)$ different WPs.

This design is scalable in several ways: if the number of users or data input size grows, it should be possible to extend the system, at reasonable cost, adding more WPGs. If we want an higher level of arbitrary fault-tolerance, we should add more WP to each WPG, increasing the number PP backups too.

## 4 SYSTEM FEATURES

Given the system model described previously, a solution to tolerate at most $f$ arbitrary faults during each task would be the following: current LPP assigns a given map/reduce task to a WPG contacting its $2f + 1$ WPs; then, after retrieving the outputs from replicas, picks the most voted results which is expected correct if no more than $f$ arbitrary faults occur.

This simplistic solution is very expensive because it replicates everything $2f+1$ times: task execution, communication of map task outputs and so on. We use a set of techniques to avoid these costs and do things efficiently:

### Digest outputs

As explained above, in order to consider a map/reduce task correct, tolerating at most $f$ faulty processes, $f+1$ matching outputs, coming from at least $2f + 1$ WPs belonging to a given WPG, have to be received.

To validate task's output, since outputs computed by worker processes can be very large, in order to avoid pointless additional network traffic, we have adopted an approach according to which LPP fetches and compares *digests*, a bit string of a fixed size obtained using a hash function. The main advantage of this technique is that, digest size are many times smaller than task's data output, allowing us to reduce network traffic among servers, increasing system's performance.

### Locality-Aware Deferred Execution

Waiting for $2f + 1$ matching replies during a AFT Map-Task can be a waste of time especially when no faults occur or we have already received a sufficient number of reply to establish the output's correctness; in other words, we believe that there is no point in always executing $2f + 1$ Map-Task for each AFT Map-Task.

Therefore, to minimize both the number of Map-Task executed and the overhead due to network traffic, we have adopted a mechanism that we have called *Locality-Aware Deferred Execution.*

According to that solution, during a AFT Map-Task execution, only $f + 1$ Map-Task replicas are started checking if they all return the same result. If a time-out elapses or some return results do not match, more Map-Task replicas (up to $f$) are started, until we obtain $f + 1$ matching replies. When faults are unlikely, this approach, which we believe to be energy-saver[8], is capable to minimize the number of WPs involved into computations respect to basic scheme, improving overall system performance.

Moreover, this mechanism can reduce overall system response time because it is designed to minimize data transfer time too. When an AFT Map-Task starts $f + 1$ Map-Task replica on WPs, the latter are not chosen at random but according to their network location; in fact, Map-Task are scheduled firstly to the $f + 1$ nearest nodes giving priority to them. Only when we can't get $f + 1$ matching replies, the most distant nodes are used. The most difficult problem consists into determine WP's network location considering the variation of network traffic. In literature, many network proximity evaluation's methods exist which can be static (i.e. hop number) or dynamic (round-trip time evaluation, HTTP request emulation, etc.); in our prototype, we have adopt a very simple network proximity evaluation's method based on `ping` which able us to evaluate round-trip time among system's nodes.

### Speculative Execution

Although the advantages of the locality-aware deferred execution mechanism, we believe that waiting for $f+1$ matching Map-Task for every AFT Map-Task results before starting the Reduce-Phase can worsen the time for the job completion, especially when WPs arbitrary failures are uncommon. Formally, assuming that we have $n$ WPG, since we need at least $f + 1$ matching reply from each WPG, we have to wait for at least $n(f + 1)$ replies; all Map-Phase may require long time especially when network traffic or data set size are huge.

A way to increase system's performance reducing its response time is the following: when all scheduled AFT Map-Tasks have received the *first* Map-Task result, LPP starts Reduce-Phase immediately without waiting for each AFT Map-Tasks receive exactly $f + 1$ matching results; in other

[8] Assuming that idle WPs consume less energy respect to non-idle WPs, minimize the number of Map-Task executed involves minimize the amount of electricity used.

words, Reduce-Phase starts after one replica of all AFT Map-Tasks finish. Essentially, our system make a bet according to which arbitrary faults are highly unlikely therefore first Map-Task replies are considered as correct. This mechanism is called *Speculative Execution.*

Obviously validate the correctness of first Map-Task replies is critical otherwise job's output can be incorrect. In order to do that, all AFT Map-Tasks keep on waiting their $f + 1$ matching replies even when Reduce-Phase is started. When Map-Phase is fully finish, if is detected that the input used in the Reduce-Phase was not correct, Reduce-Phase will be restarted with the correct input, otherwise we can finish our job in much less time.

Then, when there are no fault or they are unlikely, this design is capable to reduce considerably system's response time.

### Data Locality-Aware Reduce Scheduling

Since in order to achieve fault tolerance we have replicate all task $2f + 1$ times, data transfers among servers are replicated too, generating an huge amount of network traffic. Evidently, when large data set are involved, move data repeatedly among server became a bottleneck.

Therefore we have adopted a design aware of data locations and sizes capable to reduce the overall amount of transferred data; that solution is based on a basic principles which states that "*moving computation towards data is cheaper than moving data towards computation*"; in other words "*moving computation towards data is cheaper than moving data towards computation*".

To be more precise, when a WP complete a map task, it send as output to LPP two very important information: a string containing data output digest and a value representing output data size. According to Hadoop terminology, map task output are called *partition* and, contrary to splits, they are of variable sizes: so why not just shuffle smaller partitions instead big one?

Assume for simplicity that no faults occurs and to have only two WPGs $\{g_1, g_2\}$. Let $P_{i:g_i}$ the partition owned by WPG $g_i$ which contains all key-value pairs which must be submitted to reduce task $R_i$. The problem is how to perform a locality-aware reduce task scheduling; in other word, which WPG $g_i$ have to run reduce task $R_i$ reducing shuffled data?

Suppose that $P_{1:g_1}$, $P_{2:g_1}$, $P_{1:g_2}$ and $P_{2:g_2}$ have a size of 300 MB, 150 MB, 150 MB and 300 MB respectively. Clearly, scheduling $R_1$ at $g_1$ and $R_2$ at $g_2$ reduce the amount of transferred data among WPGs by 50%. Obliviously using this approach is possible to perform a locality-aware shuffle task capable to reduce network traffic, applying locality to reduce task scheduling.

### Crash failure detection

To guarantee the correct functioning of the system, is critical to be able to detect WP and PP crash failure. To resolve this problem we have exploited some features of Apache Zookeeper.

As known, Apache ZooKeeper has the notion of *ephemeral nodes*, special znodes which exists as long as the *session* that created the znode is active. When session expiration occurs, Zookeeper cluster will delete all ephemeral nodes owned by that session and immediately notify all connected clients of the change.

When any client establishes a session with a Zookeeper cluster, a time-out value is used by the cluster to determine when the client's session expires. Expirations usually happens when the cluster does not hear from the client within the specified session time-out period (i.e. no heartbeat).

Therefore using that feature, we adopted a very simple design according to which any process must to "sign up" creating his own ephemeral node in Zookeeper Cluster in known paths (note that znode paths of PPs and WPs are different in order to not interfere with leader election of PPs).

It will be the responsibility of LPP to watch for WP's ephemeral node deletions; this approach makes WP status checking very simple. System coordination and membership is facilitated too, since any ephemeral node is used to store information about process that created (ID, Group ID etc.).

### Primary process leader election algorithm

**Algorithm 1**

```
(1) Let ROOT be a valid znode path. Any PP, which wants
    to become leader, creates a new znode $Z$ with path
    ROOT/$x$, where $x$ is a number. Using both SEQUENCE
    and EPHEMERAL flags, $x$ will be the sequence number
    greater than any one previously used.
(2) Let $C$ be the children of ROOT.
(3) If $Z$ is the smallest node in $C$, that is $x$ is the smallest
    sequence number used so far, then the PP that created
    $Z$ will become leader.
    (a) Otherwise watch for changes on znode ROOT/$j$ in C,
        where $j$ is the largest sequence number such that $j <$
        $x$.
    (b) Upon receiving a notification of znode deletion, go
        to step number 2.
```

Let's now describe in details how we have resolved leader election problem of a PP using Apache Zookeeper.

Suppose that `ROOT` is a valid path of an existing znode. To apply as possible leader, any PP creates a znode child into `ROOT` using `SEQUENCE` and `EPHEMERAL` flags. When `SEQUENCE` is used, ZooKeeper cluster automatically appends a sequence number to the new znode that is greater than any one previously appended to a child of `ROOT`. The PP, which created the znode with the smallest appended sequence number, become leader.

Obliviously is important to watch for failures of the leader in a such way that a new PP arises as the new leader when leader fails. Note that, since we have used `EPHEMERAL` flag during znode creation, when leader fails the smallest znode will be deleted by Zookeeper cluster. if we make sure that all PPs watch for the next znode down on the sequence of znodes, if a PP receives a notification that the znode it is watching is gone, then it becomes the new leader since there is no smaller znode.

A complete description of leader election algorithm is shown in Algorithm 1 while our implementation is in `SDCC-Project/aftmapreduce/cloud/zookeeper`

## 5 THE ALGORITHM

In this section we will provide a detailed description of our MapReduce algorithm used to fully accomplish any client's word-count request. Our algorithm is made up of five phases every of which have to be successfully completed in order to complete a request.

### Data upload phase

The first phase of our algorithm expects that client would upload their test input file into a DFS (*Distributed File System)* or, equivalently, into an external storage service.

### Acceptance phase

After data upload, client sends a word-count request to current LPP.

When that request is been received, LPP, after to have written into Zookeeper cluster all necessary information in order restore its state after a crash failure (request's status information, text input data digest etc.), notifies client that his request is been accepted by system, starting Map phase.

When LPP response is been received, client starts to wait for a new notification by Zookeeper cluster (watching a specified znode) which will signalize computation completion.

If any error occurs during this phase, client have to re-forward its request to LPP until it will receive acceptance confirmation.

Note that our system is able to manage duplicated requests because we have adopted a *most-once semantics*; in fact, LPP is able to recognize duplicates re-forwarding its reply to client while client, if it detects that LPP is been crashed, forwards a query to Zookeeper cluster in order to know who is the new LPP.

### Map phase

LPP splits received text into *n* splits where *n* is the number of WPGs currently available in the cluster (this number is checked at runtime). Then LPP starts *Map Phase* during which *n Arbitrary-Fault-Tolerant Map Tasks* (AFT-MTs) are initialized and assigned to every WPGs (note that each WPG manages a different AFT-MT).

To make this phase fault tolerant, each AFT-MT is replicated using the WPs available into a WPG specified by LPP. As explain in previous sections deferred execution and digest outputs are used. Our prototype is designed to check at runtime the number of currently available WPs into a WPG calculating automatically fault tolerance level (for example if there are three WP currently on-line of a specified group, the system behaves properly in order to tolerate one arbitrary failure).

When all AFT-MTs finish successfully, LPP perform a checkpoint storing into Zookeeper cluster a set of metadata used to restore LPP status in case of crash (these data include digest outputs from WPGs and some information about computed data outputs owned by WPGs).

In case of LPP crash (note that also an high amount of arbitrary fails of WPs determines a LPP crash), after using our leader election algorithm, a PP backup copy becomes leader. Then the new leader recovers previous leader's status from last checkpoint querying Zookeeper cluster. All pending requests are restarted according to last checkpoint while input text data are retrieved, if necessary, from DFS or an equivalent storage service.

### Locality-aware shuffle and reduce phase

Any mapper send to LPP information about computed data like its size. Using all informations collected from each online WP during previous phase, LPP perform a locality-aware data shuffle according to which our system avoids transferring large partitions, shuffling smaller ones and saving network bandwidth.

Since, according to our design, any WPG can receive a partition from any other, we need to wait until all mappers complete to certainly designate all the network locations of partitions. When map phase is fully done and all the network locations known, LPP starts to coordinates shuffle phase, ordering to some or all WGs to move part of their data to precisely location. These location are computed in order to shuffle smaller partition only.

When shuffle phase is fully complete, LPP starts to coordinate *n* Arbitrary-Fault-Tolerant Reduce Tasks (AFT-RTs) assigning them to every WPGs which now have all necessary data to perform this kind of operation. When all AFT-RTs fully finish, LPP perform another checkpoint in order to restore from an eventual LLP's crash.

If too many arbitrary errors occur, making AFT-RTs completion impossible, it's enough to restart current phase.

### Retrieve and Collect phase

When reducer phase is fully done, LPP start to retrieve all reduce outputs from WPs. To be more precise, for every available WPG in system LPP picks a single WP which, according to data received during last phase, has computed correct outputs.

Then all reduce outputs are finally concatenated in an single output file. LPP stores output file into Amazon S3 using its digest as key. Then LPP informs the client sending output file digest through which client can retrieve his output.

## REFERENCES

[1] Pedro Costa, Marcelo Pasin, Alysson N. Bessani, and Miguel Correia. 2011. Byzantine Fault-Tolerant MapReduce: Faults Are Not Just Crashes. (2011).

[2] Mohammad Hammoud and Majd F. Sakr. 2011. Locality-Aware Reduce Task Scheduling for MapReduce. (2011).

[3] IBM. [n.d.]. *What is MapReduce?* Retrieved September 2, 2019 from https://www.ibm.com/analytics/hadoop/mapreduce

[4] E. B. Nightingale, J. R. Douceur, and V. Orgovan. 2011. Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs. *Proceedings of the EuroSys 2011 Conference* (2011), 343—-356.

[5] B. Schroeder and G. A. Gibson. 2007. Understanding failures in petascale computers. *Journal of Physics: Conference Series* (2007), 78.

[6] B. Schroeder, E. Pinheiro, and W.-D. Weber. 2009. DRAM errors in the wild: a large-scale field study. *In Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems* (2009), 193—-204.

[7] Maarten van Steen and Andrew S. Tanenbaum. Version 3.01 (2017). *Distributed Systems.*