Arbitrary Fault-Tolerant and Locality-Aware 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 a MapReduce runtime system as a solution to these problems. Although job's execution using our system requires more resources respect to other implementations due to process replication, we believe that this cost is acceptable for critical applications that require an higher level of fault tolerance.

KEYWORDS

MapReduce, Arbitrary Fault Tolerance, Data-Locality Awareness, Speculative Execution, Leader Election Algorithm, Deferred Execution

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.

For these reasons, to construct a distributed system capable to provide its services even in the presence of failures, mitigating network traffic and delay, is become a critical aspect of software engineering.

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

2 ASSUMPTIONS AND REMARK

In order to properly describe how our system can achieve a certain level of arbitrary fault tolerance without unduly compromising performance, let's start making some very important assumptions and observations.

We have designed our cloud application as a *set of processes*, every of which have to both run on *different* servers and have got a certain replication degree in order to achieve a certain level of arbitrary fault tolerance, therefore we have deploy our processes in several Amazon EC2 instances hosted in a same region (`us-east-1`)[1].

In order to enable communication with the Internet and therefore allow a local computer to connect to our system, we have adopt AWS Elastic IP[2] which able us to associate to some of our EC2 instance to a public static IPv4 addresses.

Observe that processes are not all the same but several kind of these exist, with its own features and aims: although we will describe them in detail in following section, just note that we distinguish three process types: *Client Process*, *Primary Process* (PP) and *Worker Process* (WP);

Note that our MapReduce system, capable to execute word-count job only, is been implemented from scratch using Go

[1]Note that, due to Account AWS Educate Starter limitations, we couldn't run Amazon EC2 instances on other regions.
[2]https://docs.aws.amazon.com/en_us/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html

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

programming language[3] using some external libraries, which are all reported in table 1.

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 a time-out to state that a process has crashed although, occasionally, such conclusion is false [7].

All processes are connected by *reliable channels*, that is no messages are lost, duplicated or corrupted. In our prototype that feature is guaranteed by the use of Go implementation of RPC[4] (*Remote Procedure Call*) which is based on TCP connections. Mostly all RPC calls, except those made by the client process, are asynchronous[5].

Clients are *always correct*, because if they are not there is no point in worrying about the correctness of our system's output. 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.

Finally, we remember that our prototype's source code is fully available on GitHub[6], famous web-based hosting service for version control using `git`; we remind that LaTeX source code of this paper is available too[7]. In this paper, we take for granted reader's knowledge of the MapReduce framework and part of Hadoop framework.

3 SYSTEM'S MODEL

Let's start now the analysis of all processes of our system.

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 a 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. To perform

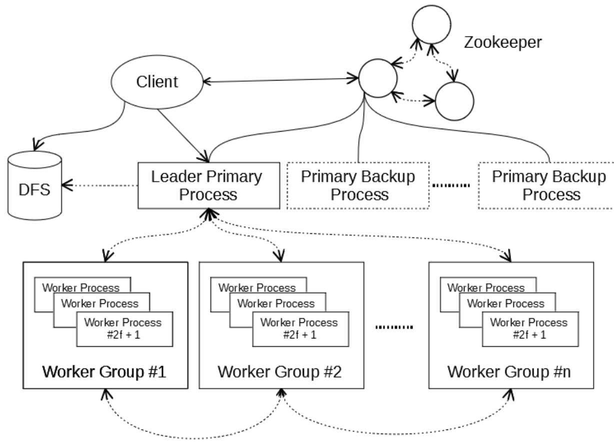

Figure 1: System Architecture

data transmissions, client processes use HTTP-based RESTful APIs, according to which both an URL and standard HTTP methods (`GET` and `PUT`) only are required.

Is very important to specify that, in our system, URLs are provided by a PP 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 (*buckets*) are by default private, clients cannot upload or download data without AWS security credentials or permissions.

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 PP 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 PP, that has AWS security credentials to interact with Amazon S3[8], has to generate and distribute them to clients.

[3]https://golang.org
[4]https://golang.org/pkg/net/rpc/
[5]Although Go's `rpc` package provides its own support for RPC asynchronous calls, we have preferred to adopt an equivalent approach based on properly used goroutine and channels.
[6]https://github.com/AndreaG93/SDCC-Project
[7]See https://github.com/AndreaG93/SDCC-Project-Report

[8]We remember that to give all privileges required for pre-signed URL creation to the EC2 instances where PP is running, we have to properly set

Observe that client processes are unaware of PP network location, that is they doesn't know its internet address (which can change during system activities due to either a crash or connectivity lost because to Amazon EC2 policies). To be more precise any system's process know network location of the others although an Apache Zookeeper^{TM} cluster location, which we have adopt for membership managing and coordination purposes, must be known by everyone. According to our design, is possible to know network location of a specified process querying Zookeeper cluster, therefore clients are able to know where PP is and then it can both require pre-signed URL and submit jobs.

Primary Process

Most of the activities carried out by a Primary Process are very similar to those performed by *JobTracker* process used in Apache Hadoop framework. These activities including *Map-Task* or *Reduce-Task* scheduling, pre-signed URLs generation, managing of worker nodes replies, clients request managing and so on.

However unlike Apache Hadoop framework, according to which *JobTracker* process is assumed always correct, we have design our system believing that PP, and the host where it 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, inspired by Apache Mesos^{TM} architecture[9], we have adopt a design according to which multiple PP backup copies exist and they 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 a PP crash failure: when current LPP fails, a backup copy is promoted to become the new leader.

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 saves its state regularly. Since LPP state consists on a very small amount of data, we have design LPP to store its state into Apache Zookeeper cluster upon the occurrence of certain events. When a LPP crash failure occurs, the new leader can easily retrieves last saved state from ZooKeeper cluster. Note that Apache Zookeeper's processes is been replicated too in order to tolerate a certain number of faults upon Zookeeper cluster.

Another very important aspect of PP is its ability to recognize duplicated requests coming from client (which can re-transmit a same request) re-forwarding responses; in other words, we have adopt a *most-once semantics*. Note that WP too are designed according to most-once semantics.

Another difference with Hadoop *JobTracker* process regards task scheduling approach. In our design LPP schedules tasks to WP using a *push-based* approach, contrariwise Hadoop JobTracker schedules task using a *pull-based* approach where WPs ask for tasks[2].

Worker Process and Groups

To achieve fault tolerance we have adopt a design according to which each task is executed more than once by different processes running on different host. To do this, as you can see from figure 1, all worker processes has been organized in several groups which we have used to mask one or more faulty processes: these groups are called *Worker Processes Group* (WPG). To be more precise, an WPG is a set of identical WP, each of which execute *the same commands using the same input data*; therefore we expected that all WPs belonging to a same group would produce the same output when no arbitrary fault occurs.

According to our system's design, any word-count job is made up of two main phases: *Map-Phase* and *Reduce-Phase*. To fully complete the Map-Phase of a job is necessary to successfully perform a certain number of special tasks called *Arbitrary Fault-Tolerant Map-Tasks* (*AFT-Map Tasks*): a AFT-Map represents a fault tolerant version of classical map task, in fact it simply consist on a replicated map task among a certain number of WP. To be more precise, our system is been design so that each WPG performs its own AFT-Map Task independently: to do that, each WPs, belonging to a given WPG, will perform the same map task in order to mask faulty processes. Clearly all this is also valid for Reduce-Phase.

Suppose now to want tolerate at most f arbitrary failures. To achieve that level of fault-tolerance is clearly necessary that any AFT-Map Task or AFT-Reduce Task assigned to each WPG are able to tolerate at most f arbitrary failures. At this point we can apply, for instance, *byzantine agreement* among all processes of a given WPG, but 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

all necessary IAM Roles thought "*AWS Identity and Access Management service*" (IAM)

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

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.

Formally, let $i = 1, 2, 3...n$, where $n \in \mathbb{Z}_{+}$. LPP splits the input text file S in n parts $\{s_1, s_2, ...s_n\}$ called *splits*; each split s_i is associated to an AFT-Map Task $m_i \in \{m_1, m_2, ...m_n\}$. Each AFT-Map Task m_i is assigned to WPG $g_i \in \{g_1, g_2, ...g_n\}$. Then $2f + 1$ different WPs belonging to WPG g_i perform the same map task replica. LPP, checking received outputs from each WP of each group, picks the most voted results which is expected correct if no more than f arbitrary faults occur. Obviously Reduce-Phase is performed in similar ways.

Clearly, this design requires more resources respect to classic MapReduce scheme: to tolerate at most f arbitrary failures all WPGs have to contain at least $2f + 1$ different WPs, therefore a total of $n(2f + 1)$ different WPs is required.

Although its very high costs, 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. te

4 SYSTEM'S FEATURES

At this point should be clear to reader that our design, despite its simplicity, is very expensive because almost everything is replicated $2f + 1$ times like map/reduce tasks, data transmissions and so on. Can we have a good level of fault tolerance without worsening performance? How can we do AFT-Map Task and AFT-Reduce Task efficiently?

In this section we will explain a set of solutions and techniques which we have adopt in our system to optimize it reducing costs and doing things efficiently.

Digest outputs

As explained above, in order to consider any AFT-Map Task/AFT-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 to which we have assign the aforementioned task, have to be received.

Since outputs computed by WPs can be very large, we have adopted an approach according to which, instead of to compare raw data, LPP fetches and compares *digests*, that is fixed size strings obtained from raw data outputs using an hash function. The main advantage of this technique is that digest size are many times smaller than task's data output, allowing us to validate task's output reducing network traffic among servers.

Locality-And-State-Aware Deferred Execution

It is frequently a good idea to distinguish between the most common case and the worst possible case and treat them differently. In our system the occurrence of f arbitrary faults represents the worst possible case, not the common in which less than f errors occur. To improve considerably system's performance is important to make the common case fast. When worst case occurs, is sufficient to make it correct.

Therefore waiting for $2f + 1$ matching replies during any AFT-Map Task/AFT-Reduce Task can be a waste of time especially when fewer 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 $2f + 1$ replicas for a given AFT-Map Task/AFT-Reduce Task.

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

According to that solution, during a AFT Map-Task/AFT-Reduce Task execution, only $f + 1$ replicas are started checking if they all return the same result. If a time-out elapses or some return results do not match, more 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[10], 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 because our scheduler gives priority to *nearest* WPs. When an AFT-Map Task/AFT-Reduce Task starts $f+1$ replica on WPs, the latter are not chosen at random but according to their network location; in fact, the tasks are scheduled firstly to the $f + 1$ nearest nodes. 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.

We have designed our task scheduler to be also *state-aware* since priority is given to those WPs which have minimum CPU utilization which we compute using `mpstat` tool. WPs states are acquired indirectly by LPP through their replies.

[10] 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.

Speculative Execution

Although the advantages of the deferred execution mechanism describes above, we believe that waiting for $f + 1$ matching digest for all AFT Map-Tasks before starting the Reduce-Phase can worsen the time for the job completion in some situations. Assuming that we have n WPGs, since we need at least $f + 1$ matching reply from each WPG, we have to wait for at least $n(f + 1)$ replies to fully complete Map-Phase, which may require long time.

A way to increase system's performance reducing its response time is the following: when all n scheduled AFT Map-Tasks have received the *first* replica's result, LPP starts Reduce-Phase immediately without waiting for each AFT Map-Tasks receive exactly $f + 1$ matching results; in other words, Reduce-Phase starts after one replica of all AFT Map-Tasks finishes. Essentially, our system make a bet according to which arbitrary faults are highly unlikely and therefore first Map-Task replies can be considered as correct. This mechanism is known in literature *Speculative Execution*, an optimization technique applied frequently in computer technology.

Obviously validate the correctness of first Map-Task replies is critical otherwise job's output can be incorrect. To do that, all scheduled 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 and Shuffle

So far we have not yet mentioned the shuffle task according to which WP redistribute data (produced by the map function), such that all data belonging to one key is located on the same WP's node. Unlike Hadoop implementation, shuffle task is difficult and expensive due to processes replication. Clearly data transfers among servers have to be replicated, generating an huge amount of network traffic and 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 capable to reduce the overall amount of transferred data: this approach is called *Data-Locality-Aware Reduce Scheduling*. That solution is based on a basic principles which states that "*moving computation towards data is cheaper than moving data towards computation*".

Using the same Hadoop terminology, map task output, produces by every WPs, are called *partition* and, contrary to splits, they are of variable sizes: so why not just shuffle smaller partitions instead big one? Then, why not schedule an AFT-Reduce Task to the WPG which nodes has bigger partition? In this way we can avoid transferring huge amount of data increasing overall system performances.

To explain how our scheduler works, let's illustrate it with an example: assume for simplicity 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. 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. In this context LPP schedules R_1 to g_1 and R_2 to g_2 because in this way is able to reduce transferred data among WPGs by 50%.

Crash failure detection

To guarantee the correct functioning of the system, is critical to be able to detect WP and PP crash failure.

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 we have exploit this Zookeeper feature design our system in such a way any system's process, except for client process, must to "sign up" creating his own ephemeral node in Zookeeper Cluster before to start to perform its task [11]. To be more precise, every WPs, whose ID is `i` and belonging to WPG `g`, creates a znode named `/membership/i_g`. Then, when any WP crashes, LPP, which watch for changes upon `/membership` znode path, is notified by Zookeeper

Election algorithm

As we said previously, our system is based on multiple PP backup copies which run independently on different machines. Since we need a leader, let's now describe in details how we have managed leader election problem which we have resolved, once again, 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

[11]To be precise that ephemeral znodes corresponding to PPs and WPs are stored in different path to not interfere with leader election algorithms

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.

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.
```

5 SYSTEM'S WORK-FLOW

To In this last section we will sum up how our system works in several steps to better understand some aspect of our solution.

Data upload phase

In the first place, any client uploads its test input file into Amazon S3 using its digest, obtained making use of SHA-512 hash algorithm, as key. Remember that clients, to perform upload operation, have to require a pre-signed URL to LPP, which network location will be obtained querying Zookeeper cluster.

Acceptance phase

When data upload is complete, client sends a word-count request to current LPP which execute several initialization operations in order to register incoming request. To be more precise, LPP creates two new znodes into Apache Zookeeper cluster: one of them is used to store request related information (request's status information, text input data digest etc.) necessary to restore LPP state in case of crash; the other is used instead to store the digest of job's output.

If all these znodes are successfully created, received request is considered as "accepted" and clients start to wait for change the output's znode while LPP can start Map-Phase. If any error occurs during this phase, client have to re-forward its request to LPP until it will receive acceptance confirmation; re-forwarding requests is fully safe since LPP can filter duplicated requests. If LPP crashes, clients will wait for a new leader before to re-forwarding its request.

Map-Phase

During this phase, LPP fetches input text file from Amazon S3 using the key provided by client; then, it splits it into n parts where n is the number of WPGs currently registered on Zookeeper cluster. Afterwards LPP schedules n AFT-Map Tasks assigning them to each WPG. Fault tolerance mechanism is been already described previously but note that our prototype is designed to check at runtime the number of currently available WPs into a WPG calculating automatically reachable 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).

Normally, when all AFT-Map Tasks finish successfully, LPP performs a checkpoint of request's state writing into Zookeeper cluster all data required to recover from a LPP crash (these data include digest outputs from all WPGs and information about owned partitions size). Since we have adopt speculative execution mechanism, Reduce-Phase will be stared immediately after one replica replies of all AFT Map-Tasks; in this case there is no point on doing a checkpoint because we cannot trust on Map-Phase output used. Therefore, we will make the Map-Phase checkpoint only if speculative execution fails.

Note that if LPP crashes, the new leader is able to recover all pending requests from last checkpoint restarting them.

Shuffle and Reduce Phase

We have already described how we have manage Shuffle and Reduce Phase and, anyhow, there are no relevant concepts to be explored. Just note that another request status checkpoint will be made after the end of Reduce Phase. If too many arbitrary errors occur, it's enough to restart current phase.

Retrieve and Collect phase

When reducer phase is fully done, LPP starts to retrieve all AFT-Reduce TASK outputs from each WPG sending a specific request to the nearest and least loaded WP belonging to them. When all outputs are finally retrieved and concatenated in an single output file, LPP stores output file into Amazon S3 using output file digest as key which will be written into Zookeeper cluster. Client process, informed of the completion of his request, can fetch the output's digest from Zookeeper cluster and use it to download 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.*