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 other, 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 to distant nodes is becoming the bottleneck owing to an increased network traffic causing performance degradation.

Therefore to construct a distributed system capable to provide its services even in the presence of failures is become so critical; consequently, to provide a *fault tolerant* cloud application represents an important goal in distributed-systems design.

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

In these paper we will describe our *Arbitrary Fault-Tolerant Locality-Aware* (AFTLA) MapReduce system capable to resolve all issues described above. We have implemented an AFTLA MapReduce system, capable to perform a word-count of a given text input, from scratch using Go programming language, adopting Apache ZooKeeper™ for maintaining configuration information and system coordination [1]; all libraries used in our implementation are fully described in table 1. Our prototype 's source code is fully available on GitHub[2], famous web-based hosting service for version control using `git`; we remind that LaTeX source code of this paper is available too[3].

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; in our prototype, every process run on his own Amazon EC2

[1] See https://zookeeper.apache.org
[2] See https://github.com/AndreaG93/SDCC-Project
[3] See https://github.com/AndreaG93/SDCC-Project-Report

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

server hosted in a same region (`us-east-1`). All processes interact with each other using RPC

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. All system 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 Hadoop implementation. framework.

3 SYSTEM'S PROCESSES

Three kind of process are used in our system:

Client Processes Which send word-count requests to our system sending directly an input text file.

Worker Process (WP) They execute *map* or *reduce* scheduled by LPP.

Primary Process (PP) It is similar to *JobTracker* process used in Apache Hadoop which duty is to satisfy clients requests scheduling *map* or *reduce* tasks.

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, 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 a *single point of failure*.

Therefore, to ensure an high system availability, adopting the same design used in Apache Mesos™ [4], our system architecture is based on multiple PP backup copies run on different machines; using any leader election algorithm, is possible to elect one of these copies as *Leader Primary Process* (LPP) while all the others remain on stand-by. Using this approach, is possible to manage PP crash failure: when current leader fails, a backup copy is promoted to become the new .

LPP represents our system coordinator since only it can can interact with WP scheduling 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.

The algorithm

To make our system arbitrary fault-tolerant we have followed the approach according to which each task is executed more than once by different processes running on different host. This design requires to organize several identical processes into a group through which became possible to mask one or more faulty processes.

As known, MapReduce framework splits input file in several splits to each of which a map task is associated; in other words, if an input file is split, for instance, into n elements, primary process has to schedule n map tasks. In order to achieve arbitrary fault-tolerance, each map task has to be executed by a groups of identical process performing given task. According to this design, in order to manage n map

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

tasks, n different groups of processes are needed; these group are called Worker Groups. A worker group is a set of equal worker processes, each of which execute the same commands using same input data in the same order. In a group all worker processes run independently on different host and they do not interact with each other in any way. Current Leader Primary Process can interact with groups members using a push-based approach in order to schedule map or reduce tasks. This is the reason according to which this design is more expensive than using the original MapReduce runtime or Hadoop.

To be more precise, suppose to have n worker groups and to want tolerate at most k faulty processes for each group. To achieve arbitrary fault-tolerance, we can apply, for instance, a consensus algorithm like Paxos or Raft in order to reach consensus among all processes belonging to a given worker group. However that solution is too expensive because it requires $3f + 1$ replicas; moreover if we have, for instance, n worker groups, we need at least $n(3k + 1)$ processes.

This is the reason according to which we have adopt an An important issue with this design is how much replication is needed. As known, if processes exhibit arbitrary failures, continuing to run when faulty and sending out erroneous or random replies, a minimum of $2k + 1$ processes are needed to achieve k-fault tolerance. In the worst case, the k failing processes could accidentally generate the same reply. However, the remaining $k + 1$ will also produce the same answer, so the primary process just believe the majority. Notice that, for instance, applying consensus algorithm to reach consensus among k group members may suffer from fail-arbitrary failures requires 3f + 1 replicas to tolerate at most f faulty replicas.

we have discarded this option considering it too expensive through which

,

In replicated-write protocols, an update is forwarded to several replicas at the same time.

we have a group of 3k+1 processes. Our goal is to show that we can establish a solution in which k group members may suffer from fail-arbitrary failures, yet the remaining nonfaulty processes will still reach consensus

All worker processes are running are logically split into several *Groups*, that is sets of equal worker processes, each of which execute the same commands using same input data in the same order. In a group all worker processes run independently on different host and they do not interact with each other in any way. Current Leader Primary Process can interact with groups members using a push-based approach in order to schedule map or reduce tasks. Although, for performance reasons, not always happen, when a task is sent to the group itself, all members of the group receive it.

Digest outputs

As explained above, in order to consider a task correct, tolerating at most f faulty processes, we need $f + 1$ matching outputs 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 primary process fetches and compares outputs *digests* from all workers within a group; this design allows us to increase system's performance.

Crash failure detection

To manage and detect workers processes crash faults we have use some features of Apache Zookeeper.

As known, Apache ZooKeeper has the notion of *ephemeral nodes*, that is 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).

Notice that session is kept alive by requests sent by client, therefore is critical that client will send PING request to keep the session alive. This PING request not only allows the ZooKeeper server to know that the client is still active, but it also allows the client to verify that its connection to the ZooKeeper server is still active.

Using this design, is very easy to keep check the status of worker processes.

Deferred execution

We believe that there is no point in always executing $2f + 1$ replicas for each task to usually obtain the same result.

To minimize both the number of copies of tasks executed and the overhead due to network traffic, saving also some energy, we have adopted a design called *Deferred Execution*: according to that solution current primary starts only $f + 1$ replicas of the same task, checking if they all return the same result. If a time-out elapses, or some returned results do not match, more replicas (up to f) are started, until there are $f + 1$ matching replies.

Supposing that Byzantine faults are uncommon, this design reduce the overhead introduced by the basic scheme.

Data locality awareness

Since moving data repeatedly among nodes is bottleneck, to improve MapReduce performance, we have adopt a design

aware of data locations and sizes in order to mitigate network traffic.

Our solution is based on a basic principles which states that "*moving computation towards data is cheaper than moving data towards computation*". When the map phase is fully done, all the network locations of the feeding nodes of every reducer will be known

Moving data repeatedly to distant nodes is becoming the bottleneck [23]. In this paper we rethink reduce task scheduling in Hadoop and suggest making Hadoop's reduce task scheduler aware of partitions' network locations and sizes in order to mitigate network traffic.

A key question is how to schedule reduce tasks at Task Trackers so as to diminish shuffled data and improve MapReduce performance. One of Hadoop's basic principles is: moving computation towards data is cheaper than moving data towards computation. Such a principle is employed by Hadoop when scheduling map tasks but bypassed when scheduling reduce tasks. MapReduce is aware of the network locations of splits (inputstomappers)andleveragessuchinformationtoschedule mappers nearby splits. In contrast, MapReduce is oblivious to the network locations of partitions (inputs to reducers) and does not schedule reducers nearby partitions. Thus, similar to map task scheduling, we suggest making MapReduce aware of partitions network locations in order to apply locality to reduce task scheduling

As we will be able to explain later, this design allows us to exploit data locality to increase system performance; in fact, when

From an implementation point of view, we used services offered by Apache ZooKeeper to implement our leader election mechanism.

by a set of distributed processes:

Client Process the clients that request the execution of jobs composed by map and reduce tasks

Leader Primary Process It manages the execution of word-count jobs received from clients coordinating Worker Nodes

Backup Primary Process It manages the execution of word-count jobs received from clients coordinating Worker Nodes

Worker Process A Worker Process executes map and reduce task scheduleted by current Leader Primary Process. In order to achieve fault tolerance, any Worker Process must be run indepently on different host. In our implementation, each process run on independent Amazon EC2 server

The Mesos master stores information about the active tasks and registered frameworks in memory: it does not persist it to disk or attempt to ensure that this information is preserved after a master failover. This helps the Mesos master scale to large clusters with many tasks and frameworks. A downside of this design is that after a failure, more work is required to recover the lost in-memory master state.

Worker Group All system's nodes in which worker process are running are logically split into several *Groups*, that is sets of equal worker processes, each of which execute the same commands using same input data in the same order. In a group all worker processes run independently on different host and they do not interact with each other in any way. Current Leader Primary Process can interact with groups members using a push-based approach in order to schedule map or reduce tasks. Although, for performance reasons, not always happen, when a task is sent to the group itself, all members of the group receive it.

The key property that all groups have is that when a message is sent to the group itself, all members of the group receive it.

primary coordinates all write operations

In other words, we can replicate processes and organize them into a group to replace a single (vulnerable) process with a (fault tolerant) group.

When a task is for work is generated, either by an external client or by one of the workers, it is sent to the coordinator.

as a set of Task- Trackers that execute tasks

The algorithm

In order to achieve A simplistic solution to make MapReduce Byzantine fault-tolerant given the system model would be the following. First, the JobTracker starts 2f + 1 replicas of each map task in different servers and TaskTrackers. Second, the JobTracker starts also 2f + 1 replicas of each reduce task. Each reduce task fetches the output from all map replicas, picks the most voted results, processes them and stores its output in HDFS. In the end, either the client or a special task must make the vote of the outputs to pick the most voted. An even more simplistic solution would be to run a consensus, or Byzantine agreement between each set of map task replicas and reduce task replicas. This would involve even more replicas (typically 3f + 1) and more messages exchanged.

Crash failure detection

Deferred execution

As known, arbitrary faults are very hard to detect and manage

Deferred execution. Crash faults are detected by the previously existing Hadoop mechanisms, and arbitrary faults are uncommon, so there is no point in always executing $2f + 1$ replicas to usually obtain the same result.

By default, current leader primary process starts only $f+1$ replicas of the same task, then wait results checking if they all return the same result. If a timeout elapses, or some returned results do not match, more replicas (up to f) are started, until there are $f + 1$ matching replies.

In the best case, without Byzantine faults, only f + 1 replicas are started. If arbitrary faults are uncommon, we have a < f + 1 replica started reducing the overhead

4 ARBITRARY FAULT-TOLERANT LOCALITY-AWARE MAPREDUCE

Arbitrary fault tolerance

As known academic literature describes many type of failure, like *crash failures*; however the most serious are known as *arbitrary failures* or *Byzantine failures*, according to which a server may produce arbitrary responses at arbitrary times which cannot be detected as being incorrect.

Redundancy represents the key technique used to manage these kind of failure, according to which,

The key approach to tolerating a faulty process is to organize several identical processes into a group.

Our BFT MapReduce follows the approach of executing each task more than once, similarly to the works mentioned above. The chal-

Process groups are part of the solution for building fault-tolerant systems. In particular, having a group of identical processes allows us to mask one or more faulty processes in that group. In other words, we can replicate processes and organize them into a group to replace a single (vulnerable) process with a (fault tolerant) group.

An important issue with using process groups to tolerate faults is how much replication is needed. To simplify our discussion, let us consider only replicated-write systems. A system is said to be k-fault tolerant if it can survive faults in k components and still meet its specifications. If the components, say processes, fail silently, then having k + 1 of them is enough to provide k-fault tolerance. If k of them simply stop, then the answer from the other one can be used.

On the other hand, if processes exhibit arbitrary failures, continuing to run when faulty and sending out erroneous or random replies, a minimum of 2k + 1 processes are needed to achieve k-fault tolerance. In the worst case, the k failing processes could accidentally (or even intentionally) generate the same reply. However, the remaining k + 1 will also produce the same answer, so the client or voter can just believe the majority.

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.*