Exploring QoS in MapReduce Task Scheduling

Pedro Álvarez-Tabío Togores
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
palvare3@hawk.iit.edu

Javier Villa
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
jvilla2@hawk.iit.edu

Jesús Hernández Martín
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
jherna22@hawk.iit.edu

***Abstract*— in a market with an increasing number of companies requiring vast amounts of data processing and computation, the necessity of a system which properly distinguishes between the requirements of all these companies is becoming essential. Current intensive data processing providers rely on scheduling techniques which do not take into account their clients' demands in terms of service guarantee. In this paper we explore a possible solution to address this situation by taking advantage of existing techniques in the field of network packet switching. First we explore the existing techniques for achieving an effective task scheduling in the Hadoop open-source framework. Then, considering the main concepts involving the two most widespread techniques, Fair and Capacity Scheduler, we propose an alternative approach based on concepts taken from computer networks, such as Quality of Service (QoS). Finally, we implement a QoS MapReduce task scheduling simulator and test its and behavior and performance.**

Keywords: mapreduce, distributed, data, cloud, computing, qos, scheduling, quality

I. Introduction

Data intensive cluster computing has raised a lot of interest since Google introduced MapReduce back in 2004. MapReduce is a data-driven programming model which is especially well suited for applications with high needs of data processing. A large number of companies and institutions which provide services involving intensive data processing (i.e. data mining, distributed data storage, etc.) are relying in this technology, in which distributed and parallel computing are applied among large clusters of commodity nodes where jobs get split into small tasks. Those small tasks can be processed by the nodes in a parallel fashion.

MapReduce must be seen as a model which focuses on the parallelization of data rather than operations applied to data. The computation in MapReduce may be expressed by means of two functions, both written by the user: map and reduce. The first one takes a series of key-value pairs as input and produces an output consisting of a set of intermediate key-value pairs. The MapReduce library processes all these intermediate key-value pairs and groups together all the values associated with the same intermediate key. The resulting group of key-value pairs is then passed to the reduce function. Therefore, this function has a set of values associated to the same key, which is then computed to form a possible smaller set of values, hence producing the final results.

Regarding the implementation, each map operation may be understood as a task, which is, in turn, the schedulable entity within MapReduce. This fact makes task scheduling one of the most important processes within a distributed data computation system. Nevertheless, relatively little work has been made to address the problems raised by current task scheduling algorithms: data locality and fairness between jobs.

Precisely, the scheduling of jobs and tasks is the main area of interest in our project. Thus, after the analysis presented above, we intend to gain insight in the general overall impact job scheduling has on the performance of a system, distributed or not, by doing practical research work on that field, with an emphasis on the quantitative aspect.

II. Preliminary work

Over the last decade, large-scale distributed computing systems have become the alternative to the traditional, tightly coupled high-performance computing systems for executing scientific tasks. While such distributed systems can be very cost-effective and easily scalable, due to resource heterogeneity and to the lack of accurate resource information, scheduling jobs in such systems can be a challenge.

Currently, there are two big approaches which deal with this large scale distributed computing systems. In both of them we can find different mechanisms to cope with task-scheduling.

1) MapReduce, which is already discussed above. One of the well-known implementations of MapReduce is Hadoop, which is an open source framework which enables applications to run on large distributed networks. In terms of job scheduling, Hadoop has one through different "solutions". In its early days, Hadoop only made use of its nowadays known as "default" scheduler: the FIFO scheduler. As its name says, this scheduler chooses which task to run merely based on the time the task was submitted to the scheduler. The Fair Scheduler works in a different manner. This one assigns resources to jobs such that all jobs get, on average, an equal share of resources over time. There is no job queue in this one. Finally, the Capacity Scheduler chooses tasks from different queues. Each of these queues has an allocated capacity of the system, to which the jobs in each queue will have access. These queues also support job prioritizing.

2) dryad: Dryad is a general-purpose distributed execution engine for coarse-grain data-parallel applications. A Dryad application combines computational "vertices" with communication "channels" to form a dataflow graph. Dryad runs the application by executing the vertices of this graph on a set of available computers, communicating as appropriate through files, TCP pipes, and shared-memory FIFOs.

3) DataSynapse, Inc. holds a patent which details speculative execution for scheduling in a distributed computing platform. The patent proposes using mean speed, normalized mean, standard deviation, and fraction of waiting versus pending tasks associated with each active job to detect slow tasks. However, detecting slow tasks eventually is not sufficient for a good response time.

4) In [8], the authors presents Purlieus, a resource allocation system for MapReduce in a cloud. The main aspect concerning their work is that they describe how existing data and virtual machine placement techniques lead to longer job execution times and large amounts of network traffic in the data center. One important aspect is that the authors identify data locality as the key principle which if exploited can alleviate these problems and develop a unique coupled data and VM placement technique that achieves high data locality.

5) The benefit of data locality has also been explored in [9]. The authors propose DARE, a distributed adaptive data replication algorithm that aids the scheduler to achieve better data locality. DARE solves two problems, how many replicas to allocate for each file and where to place them, using probabilistic sampling and a competitive aging algorithm independently at each node.

III. RELATED WORK

The study of scheduling algorithms for parallel computing resources has a long history; the problem of assigning jobs to machines can in general be cast as a job-shop scheduling task, however theoretical results for job-shop problems tend to be quite general and not directly applicable to specific implementations, such as parallel task scheduling, scheduling in a distributed computing cluster, or fair-share scheduling.

While assigning and scheduling jobs in distributed and parallel systems is a well-established area, scheduling in MapReduce framework is relatively less studied. Quincy and delay scheduling are two recent schemes focused on fair scheduling in MapReduce-like systems. Both these scheduling algorithms model many aspects of real systems, for example, to preserve locality of computation and data as much as possible, and ensure fairness across jobs. Their goal, however, is not so much finding an optimal solution. Our work is to build upon these principles and established algorithms in search of a better performing design and/or implementation (fair, capacity scheduling as well as FIFO scheduling as a starting point and reference for measurements).

IV. MOTIVATION

Our work is inspired by the idea that existing job scheduling techniques are merely based on addressing data locality issues and computation power sharing between jobs. However, we think that something really important is being left behind: the quality of service assurance.

Nowadays, although there are a big number of companies requiring data intensive computing, not all of them require the same amount of processing, nor they all have the same economic means to implement such an expensive data distributed system. Instead, what the latter do is to outsource the computation to other companies (Cloudera, for example).

These service providers, in turn, may have different clients, with different needs. It is at this point where some kind of service guarantee should be provided. This would allow to properly distinguishing between the jobs submitted by each client and dynamically assign resources to them based on their owner's quality of service needs.

V. BACKGROUND

The studied job scheduling techniques belong to Hadoop, which is one of the well-known implementations of MapReduce, as said before. A complete description of these scheduling mechanisms can be seen below:

A. the FIFO scheduler

This is the default scheduler included in Hadoop. The scheduler has five priority levels. When the scheduler receives a heartbeat indicating that a map or reduce slot is free, it scans through jobs in order of priority and submit time to find one with a task of the required type. However, this scheduler does not seem to be fair, since a high priority small job could stay blocked waiting for its turn, due to a lower priority long job that was started before the small job was scheduled.

Thus, the basic problem in Hadoop default task scheduler for MapReduce is that, because of its implementation as a FIFO queue, short jobs get stuck behind long ones. In conclusion, we can state three well-recognized drawbacks of the default FIFO scheduler in Hadoop:

1. Some users may be running more jobs than others; we want fair sharing at the level of users, not jobs.
2. Users want control over the scheduling of their own jobs. For example, a user who submits several batch jobs may want them to run in FIFO order to get their results sequentially.
3. Production jobs need to perform predictably even if the cluster is loaded with many long user tasks.

B. the fair scheduler

There is a conflict between fairness in scheduling and data locality (placing tasks on nodes that contain their input data).

To multiplex clusters efficiently, a scheduler must take into account both fairness and data locality. We have shown that strictly enforcing fairness leads to a loss of locality. However, it is possible to achieve nearly 100% locality by relaxing fairness slightly, using a simple algorithm called delay scheduling.

The authors in [6] address these problems in designing the Fair Scheduler. This uses a two-level scheduling hierarchy. At the top level, allocates task slots across pools using weighted fair sharing, allowing one pool per user by default. At the second level, each pool allocates its slots among jobs in the pool, using either FIFO with priorities or a second level of fair sharing. Figure 4 shows an example pool hierarchy. Pools 1 and 3 have minimum shares of 60 and 10 slots. Because Pool 3 is not using its share, its slots are given to Pool 2. Each pool's internal scheduling policy (FIFO or fair sharing) splits up its slots among its jobs.

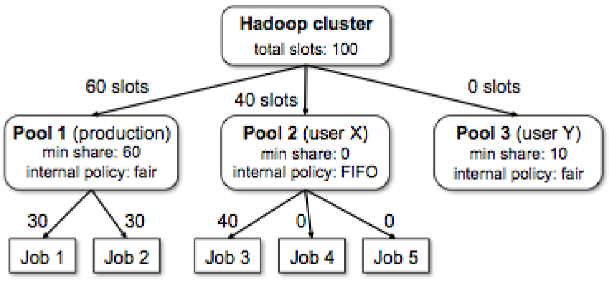

Figure 1. The fair scheduler

As shown in the example, each pool can be given a minimum share, representing a minimum number of slots that the pool is guaranteed to be given as long as it contains jobs, even if the pool's fair share is less than this amount (e.g. because many users are running jobs). Fair Scheduling always prioritizes meeting minimum shares over fair shares, and may kill tasks to meet minimum shares. Administrators are expected to set minimum shares for production jobs based on the number of slots a job needs to meet performance requirements. If the sum of all pools' minimum shares exceeds the number of slots in the cluster, HFS logs a warning and scales down the minimum shares equally until their sum is less than the total number of slots.

Finally, although HFS uses waiting to reassign resources most of the time, it also supports task killing. We added this support to prevent a buggy job with long tasks, or a greedy user, from holding onto a large share of the cluster. HFS uses two task killing timeouts. First, each pool has a minimum share timeout, Tmin. If the pool does not receive its minimum share within Tmin seconds of a job being submitted to it, we kill tasks to meet the pool's share. Second, there is a global fair share timeout, Tfair, used to kill tasks if a pool is being starved of its fair share. We expect administrators to set Tmin for each production pool based on its SLO, and to set a larger value for Tfair based on the level of delay users can tolerate. When selecting tasks to kill, we pick the most recently launched tasks in pools that are above their fair share to minimize wasted work.

C. *the capacity scheduler*

The operation of this scheduler is based on arranging the jobs into different queues. Each of these queues has an allocated fraction of the cluster capacity, which means that a certain capacity of its resources will be at their disposal, so all the jobs in that queue will have access to that capacity.

Within the queues, the ordering of jobs can be performed either by a FIFO policy or by prioritizing jobs. The latter means that jobs with higher priority will have access to the queues' allocated resources before jobs with lower priority. However, it has to be taken into consideration that a running job cannot be preempted for a higher priority job, though new tasks from the higher priority job will be preferentially scheduled.

RUNNING TASKS IN CAPACITY SCHEDULER

When a client application submits a job to the system, this job is handled by the JobTracker, which, after querying the NameNode to locate the needed data, looks for a free TaskTracker node near the data. At this point, the Capacity Scheduler picks a queue which has most free space. Once a queue is selected, the Scheduler picks a job in the queue. Jobs are sorted as described previously (FIFO and priorities). A job is selected if its user is not already using queue resources above his/her limit. The Scheduler also makes sure that there is enough free memory in the chosen TaskTracker to run the job's task.

Once a job is selected, the Scheduler picks a task to run and submits it to the selected TastTracker node, where the work is done.

VI. OUR APPROACH

QOS

As said before, current job/task scheduling approaches aim to specify fairness among jobs but neither of them says anything about guaranteed job/task processing.

Our objective is to include a quality of service specification in a job/task scheduler, so as to allow the scheduler to prioritize jobs based on the QoS required by the client who submitted them. To achieve this, we have defined several client profiles for different performance requirements. This implementation permits the fine tuning of the performance in each client, allowing the service provider to easily calculate the needed scale of the system in advance.

UNDERLYING CONCEPTS

To deal with the different QoS specifications, we have used algorithms from traffic control in computer networks and applied them to the special case of task scheduling. The description of these algorithms is included below:

A. *the tocken bucket*

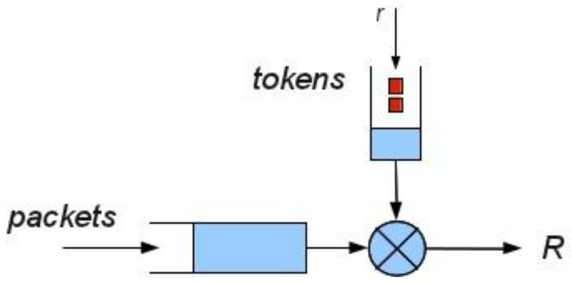

Figure 2. Token bucket in computer networks

This algorithm[6] is used in computer networks to check if data transmissions are conformant to limits in bandwidth and burstiness.

Each bucket contains tokens. In packet switched networks, these tokens may represent an amount of bytes or a single packet of predetermined size. When a packet has to be checked if it is conformant to the bandwidth defined limits, the bucket is inspected to see if it contains enough tokens at that time. This amount of tokens must be equal to that specified by the incoming packet. If the bucket contains that number of tokens, these are removed from it and the packet is allowed to pass for transmission. If there are not enough tokens in the bucket to transmit the packet, the packet is marked as “non-conformant” and no tokens are removed from the bucket. When a packet is marked as “non-conformant”, the following can happen:

- it may be dropped
- it may be queued for transmission when the bucket has enough tokens
- it may be transmitted, but without any assurance that it will not be dropped.

To allow further transmissions, tokens are added to the bucket at a given constant rate.

B. *Weighted Fair Queue*

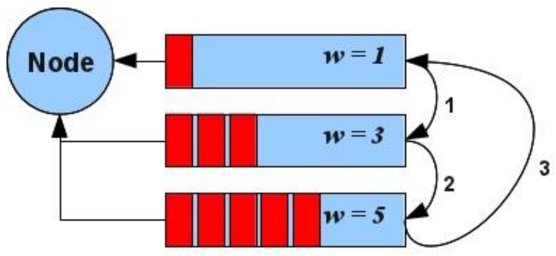

Figure 3. Weighted fair queue

Packet scheduling technique [10] which allows different scheduling priorities (weights) to statistically multiplexed data flows. It can be implemented as a weighted round robin.

VII. SYSTEM ARCHITECTURE

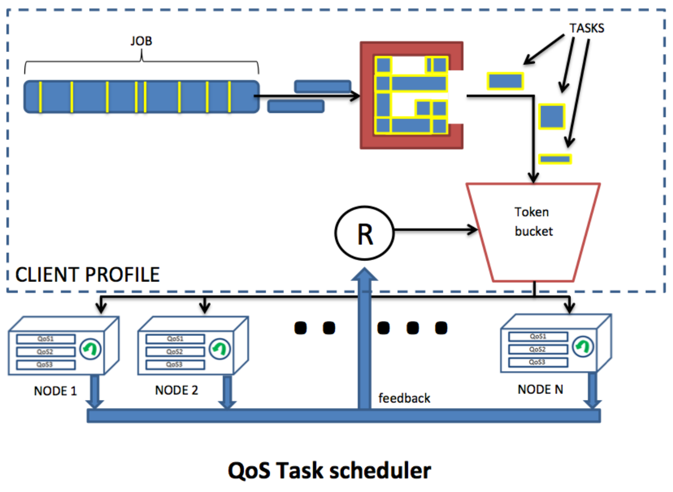

Figure 4. QoS Task scheduler overview

THE CLIENT

Clients in our system are defined by a quality of service specification, a queued token bucket and a group of jobs. Each job, in turn, is divided into a number of tasks.
Regarding the QoS specifications, each class of service is defined by the following parameters:

1) the capacity of the token bucket (in tokens)
2) the token insertion rate (in tokens/second)
3) the priority for the queues in the nodes (WFQ)

In our project, we have defined three different classes of service:

- *Premium*: highest available token insertion rate and bucket capacity, as well as queue priority. Suitable for clients who submit heavy jobs.
  800 tokens/sec | capacity: 10000 tokens | priority: 10
- *Advanced*: suitable for clients who are not very demanding on computation needs but require a better service than basic clients.
  300 tokens/sec | capacity: 150 tokens | priority: 3
- *Basic*: tasks of the clients with this class of service are given the lowest priority, as well as the lowest token insertion rate and capacity. This limits to a great extent the number of tasks to be submitted to the distributed cluster.
  100 token/sec | capacity: 125 tokens | Priority: 1

QUEUED TOKEN BUCKET

The token bucket itself is included in the client side and it is defined by its capacity and a constant token insertion rate. Its operation is similar to those used in computer networks.

Every task in our system has a priority given by the quality specification assigned to the client who submits the jobs. In addition to this, each task requires a number of tokens in the bucket to be forwarded to the nodes. We define a token as the abstract equivalent of 1 millisecond of processing in a node of the system. In this first design, we assume the nodes are in a cluster and have equal performance and bandwidth; we do not care about possible performance problems or asymmetries in the performance of the nodes. Each time a task is posted to the token bucket, the latter checks if it has enough tokens to forward it. If it does, the task is forwarded to a random node in the system, where it is processed. The bucket, in turn, removes the number of tokens used by the forwarded task.

If there aren't sufficient tokens in the bucket, the task is either:

- *Discarded*: if the current priority of the task is that of a Basic User. This is an important limitation of a Basic User: as the service provided has barely more guarantees than a best-effort approach, if the user is greedy it will not be able to forward a job.

- *Degraded*: if the current priority of the task is that of an Advanced or Premium User. Degrading a task means decreasing its priority to the immediate lower level, which will involve a longer processing time in the node. For example, in the case of a Premium User, its tasks will be degraded to the equivalent of an Advanced User.

To penalize greedy clients who submit heavy tasks, so they do not cause a performance drop in the other users, we have implemented an adaptive feedback system in which the nodes provide the token bucket with custom feedback. This feedback merely consists of a number which tells the processing time of the last task in the node.

Let α be the needed number of tokens (1 token represents 1 millisecond of computing) to forward the next task and μ the time in milliseconds provided by the feedback. The bucket computes the following:

$$\alpha = (\alpha + \mu)/2$$

The result of this operation (α) is the number of tokens needed to forward the next incoming task. If this time results to be greater than the bucket's capacity, the next task posted to the bucket is automatically penalized (either discarded or degraded, depending on the user's QoS specification)

With this approach, we can reassure fairness among users with the same quality of service specification and make the system responsive to every type of MapReduce job; no matter if the job has only a few heavy tasks or it has many lightweight processing.

THE NODES

Nodes are assigned tasks randomly, after being sent by a client who has enough tokens to forward a task; a node may thus receive tasks from various clients, each with its own flow control mechanism.

First, a node implements a set of queues, one for each defined QoS category. In our work, each node will typically have three queues since we have setup that number of QoS categories. Upon arrival, tasks are pushed to the corresponding queue.

The node controller constantly polls the queues for pending tasks using a round robin schema. To account for the QoS effect, each QoS has an associated weight, so that 'weight' tasks are extracted from the associated queue in a given iteration. This means that on any iteration over the set of queues, the number of tasks processed will be at most the sum of all the weights of the QoS categories implemented.

Upon completion of a task, the feedback mechanism takes place so that the client that originated the task can adjust its output rate accordingly. The mechanism works as follows:

- A token represents 1 millisecond of computing at any given node.
- Each Client, via its associated Token Bucket, requires a given amount of tokens to forward a task.
- This value is adjusted based on the duration of the last received task, by computing the average with the time duration with the amount of tokens needed (effectively amount of 1ms slots) to forward a task.

VIII. SIMULATION

A. Input: parameters of the simulation

The parameters in which our simulation will be based are two: the mean duration of the computation of a task in a node, and the number of tasks for each job.

In the first case, the mean duration of the computation of a task represents the heaviness of the different tasks that compose a job. For clients that submit heavy tasks, as we have explained before, the adaptive feedback system will penalize the local token bucket, thus making more difficult to submit heavy tasks. If the client has selected a cheap, low performance-guaranteed Class of Service, it will reach a point in which the tasks are so heavy that it would be submitted with a very low rate, or even would be impossible to submit.

In the second case, the average number of tasks per job gives an idea of the complexity of Map/Reduce tasks that have to be made to complete the job. In this case, and in contrast with the mean duration of a task, the adaptive feedback will not trigger the degradation algorithm provided the capacity of the token buckets is sufficient. Hence, by incrementing the average number of tasks in a job we predict that the processing time will increase a bit more than if we, instead of incrementing the duration of a single task, increase the number of tasks. However, the tasks will not be as heavy, so a lesser number will be dropped or degraded and we can get a better picture of the throughput for each QoS without the "noise" of the degradation.

B. Output: metrics for different service classes

In order to test the validity of our approach, it is necessary to simulate in a real environment the function of the simulation. The fundamental thing which we want to find out is the average job processing time. This metric represents how much time does in average spend the system to return a response (the results of the MapReduce computation). We are going to obtain it for all the different Classes of Service. As it is primarily function of the parameters given in the previous section, we are going to make two plots, one for each parameter, to see the tendency in the performance of the system.

Another very important metric is the ratio of job dropping. This can increase dramatically while incrementing the values of the first parameter we described, the mean duration of a MapReduce task. As we already explained, the adaptive feedback can cause the job to be dropped if the jobs are heavy.

C. Simulator design

In our simulator, we follow a pseudo-distributed approach in one computer. Each client is represented by a thread which periodically submits tasks to its own token bucket. Respectively, each node is a thread as well, which will operate the queues assigned to it.
The way of getting traces for the analyzer is centralized. Every client sends a short trace in all of these cases:

- When a client starts a job, it will send a timestamp, as well as its QoS and jobId
- When a client receives the last task of a certain job, it will again send the receiving time stamp, QoS and jobId.
- When a client submits a task to the token bucket, it will send its taskId and QoS
- Only when the token bucket of a client receives a non-conformant signal, it will send again that taskId and QoS

When all of these traces are collected, our data curation scripts will compute all the data retrieved and return the previously specified metrics for each QoS. For the average job processing time, we will compute all timestamp differences belonging to every job in a certain QoS, and return the average. Conversely, for the ratio of job dropping we will just return the percentage of non-conformant signals we received for every task in each QoS.

D. Simulation results

These simulation results are given by our system running under the following conditions:

- number of nodes: 100
- number of clients: 1000
- QoS randomly selected for each client
- average number of jobs: 10
- average tasks per job by default: 20
- average task duration by default: 50 milliseconds

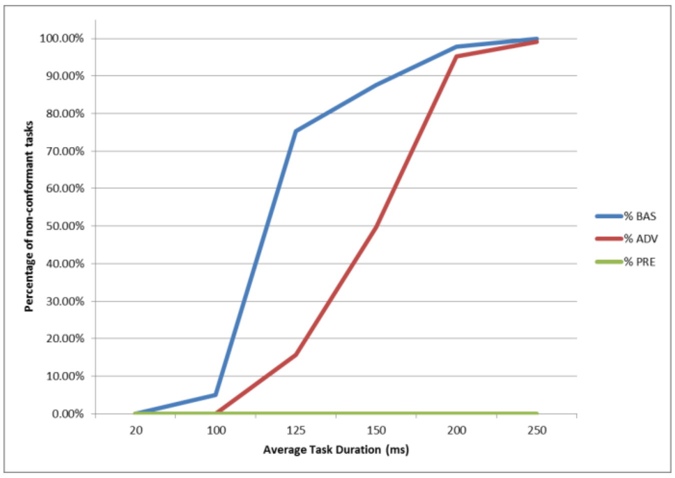

Figure 5. Average task duration vs. Percentage of non-conformant tasks

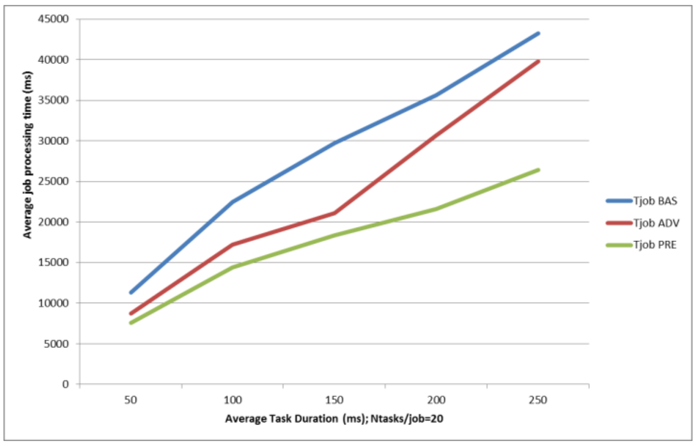

Figure 6. Average task duration vs. Average job processing time

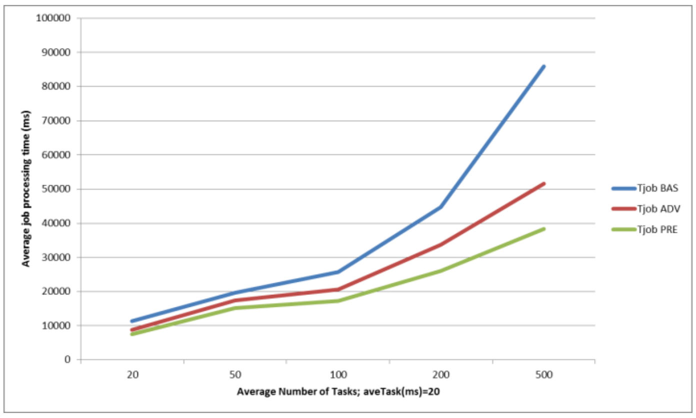

Figure 7. Average number of tasks vs. Average job processing time

We can see in figure 5, the percentage of non-conformant tasks for each QoS. As Advanced users have a slightly more capacity in their token buckets, the huge rise in the percentage will appear with longer tasks than in Basic Users. Premium users, as we have defined them, do not degrade packages in these ranges.

As we see in figure 6, the average job completion time behaves well with up to 250 ms/task. For Basic Users, the percentage of failing jobs is so high that the data only represents a few jobs that could be submitted, as the rest were dropped. Similarly, for Advanced Users the failing jobs are also high, so the tasks get degraded to Basic priority in the nodes, thus behaving similarly to a Basic QoS. Premium users do not experience any performance drop.

In figure 7, we compare the average job processing time with the number of tasks per job. The behavior of the system is different than that in figure 6. Here, as we have set a low task duration, no tasks get degraded or dropped, so the function of the system is according to the node queue priority for each QoS.

IX. CONCLUSIONS

With the rise of Big Data applications in the last few years, applications such as data mining with ever growing amounts of data that need even more computation. However, nowadays the scalability is dealt a posteriori. We propose dealing with these scalability requirements by simply looking at the MapReduce task scheduling. If we define Quality of Service metrics, we will be able to accurately predict the needs of the entire system in terms of computing power. As we can predict it, we can allocate resources in advance and give an adequate service to all users, without scalability issues. What is more, a system implementing our approach can easily segment its customer base among different requirements, that giving obvious advantages.

X. FUTURE WORK

Our design represents an initial attempt to make MapReduce "as a service" more flexible, scalable and cheap, both for the user and the service provider. Our future work is to enhance the integration with Hadoop, perhaps by defining our QoS scheduler as a layer on top of MapReduce. This will surely ease the implementation and will save us from dealing with other aspects out of our scope, such as load balancing or data locality in HDFS.

REFERENCES

[1] "Quincy: Fair Scheduling for Distributed Computing Clusters". Isard, Prabhakaran, Currey, Wieder, Talwar, Goldberg. Microsoft Research Silicon Valley.

[2] "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling". Zaharia, Borthakur, Sarma, Elmeleegy, Shenker, Stoica.

[3] "Automatic Task Re-organization in MapReduce". Guo, Pierce, Fox, Zhou. 2011 IEEE International Conference on Cluster Computing.

[4] "MapReduce Optimization Using Regulated Dynamic Prioritization". Sandholm, Lai. Social Computing Laboratory, Hewlett-Packard Laboratories

[5] "Performance-Driven Task Co-Scheduling for MapReduce Environments". Polo, Carrera, Becerra, Torres, Ayguadé, Stender, Whalley. Barcelona Supercomputing Center and IBM.

[6] Andrew S. Tanenbaum, Computer Networks, Fourth Edition, Prentice Hall PTR, 2003 , page 401..

[7] Hadoop documentation, http://hadoop.apache.org/

[8] "Purlieus: Locality-aware Resource Allocation for MapReduce in a Cloud"

[9] "DARE: Adaptive Data Replication for Efficient Cluster Scheduling"

[10] Stiliadis, D. and Varma, A. (1998). "Latency-rate servers: a general model for analysis of traffic scheduling algorithms". IEEE/ACM Transactions on Networking (TON).