A Deadline Scheduler for Jobs in Distributed Systems

Quentin Perret[1], Gabriel Charlemagne[1], Stelios Sotiriadis[2], Nik Bessis[2]
[1] Génie Électrique et Informatique INSA, Toulouse, France
[2] School of Computing and Maths, University of Derby, Derby, United Kingdom
[1] (quentin.perret, gabriel.charlemagne)@gmail.com, [2] (s.sotiriadis, n.bessis)@derby.ac.uk

Abstract — **This study presents a soft deadline scheduler for distributed systems that aims of exploring data locality management. In Hadoop, neither the Fair Scheduler nor the Capacity Scheduler takes care about deadlines defined by the user for a job. Our algorithm, named as Cloud Least Laxity First (CLLF), minimizes the extra-cost implied from tasks that are executed over a cloud setting by ordering each of which using its laxity and locality. By using our deadline scheduling algorithm, we demonstrate prosperous performance, as the number of available nodes needed in a cluster in order to meet all the deadlines is minimized while the total execution time of the job remains in acceptable levels. To achieve this, we compare the ability of our algorithm to meet deadlines with the Time Shared and the Space Shared scheduling algorithms. At last we implement our solution in the CloudSim simulation framework for producing the experimental analysis.**

Index Terms — Cloud computing, Cluster computing, Soft deadline scheduling, Hadoop

I. Introduction

The Future Internet is a notion respresenting the needs and the solutions for the coming applications over intenet. The big-data processing is one of the major study theme related to the Future Internet e.g. integrated vision so of Internet resrouces [18]. Recently, Google presented MapReduce [1] as a programming model for efficiently processing big data sets; produced by large scale systems e.g. grids and inter-clouds [14], [19]. This was one of the first paradigms for processing massive data in distributed, cluster and cloud computing. This is beacuse the computation power required to process and analyse big data is high thus such decision making management remains a problem.To this extend, a particular useful solution has been proven to be the specification of deadlines for each job for calculating the maximum amount of resources which will be used to compute the job. An overpassed deadline in this case can be seen as a compromisation yet in some instances with a non-manageable cost.

With this in mind, herein we develop a soft deadline scheduler for distributed systems with data locality management named as CLLF. The algorithm is based on the popular Least Laxity First (LLF) algorithm and has been designed for considering the practical issues related to distributed applications. Our solution shows that LLF is not applicable out of the box on a distributed system because of its preemptive behaviour which implies a null transfert cost from one node to an other.

So, we firstly present the deadline-oriented scheduling work existing in the literature (Section II), and we focus on the problems related to the implementation of these algorithms on distributed systems. By this evaluation, we show that in application such has Hadoop [2] the data locality, the predictability of the execution time of a task, and the preemption are important issues to be able to implement any deadline scheduling algorithm in a cloud. Further to this, section III describes our proposal algorithm and its requirements and assumptions. The rest of the paper is organized as follows, section IV, presents the experiment scenario and we analyze the performances of CLLF by comparing the ability of our algorithm to meet deadlines to the Time Shared and the Space Shared scheduling algorithms implemented in CloudSim [3]. We show that the deadline-meeting approach of CLLF allows to minimize the extra-cost implied by the lateness of a task. We also demonstrate the importance of the data allotment in a cloud and we propose an algorithm which determines a data distribution compatible with CLLF in order to avoid situations where some workers are resource bottlenecks. Finally, section V illustrates the concluding remarks and the further research directions of our work.

II. Background

This work aims on exploring time-oriented scheduling for large scale infrastructures e.g. HPC, grids [20] and clouds by focusing on their local resource management system [15]. Specifically, meta-scheduling decision making process is based on random services request from a user or a set of users that are clients of a datacentre that could be extended to an inter-cloud system [14]. The inter-cloud facility distributes the request for service and encloses services into VMs (a procedure that called sandboxing).

A scheduling algorithm with a deadline-meeting approach aims of catching the deadlines of every task that are about to be executed prior to its burst time. Based on that, it becomes possible to specify tasks which must be processed within a short delay that means that some answers must be given within a short delay as well. The notion of deadline meeting has been defined in many different ways, but the common point for every definition is that the aim of such scheduler is not only to guarantee the answer quality, but to do it within time specification. In this concept, we assuemt that our solution is capable for scheduling in a large scale setting that could incorporate meta-computing characteristics e.g. meta-brokers [13] that are currently are considered a future integration step.

In the following sections, we discuss the different types of deadlines and we present three popular monoprocessor deadline-aware scheduling algorithms which are directly related with our work. We then present the Hadoop schedulers used to process tasks over a cloud without any form of deadline management. And we finaly highlight issues to implement a deadline scheduler in a distributed environment such as Hadoop.

A. Deadline scheduling literature

A task τ has two main characteristics: its worst execution time T_τ and a deadline D_τ. We can then classify deadline based tasks in three categories :

- τ is a *hard deadline* task: in any case, $T_\tau < D_\tau$. The deadline miss is not allowed.
- τ is a *soft deadline* task: if $T_\tau > D_\tau$, the task has a penalty in function of its lateness $L_\tau = T_\tau - D_\tau$.
- τ is a *firm deadline* task: the task τ gains reward if $T_\tau < D_\tau$. The reward of the task is function its aheadness $A_\tau = D_\tau - T_\tau$.

1) Rate Monotonic (RM)

The RM algorithm is probably the most popular and most used scheduling algorithm in practice on mono-processor systems. However, it is based on strong assumptions. For example, the set of tasks which will be proceeded has to be known a priori. Moreover, each task of the set must:

- be periodic,
- be preemptable,
- be independent,
- have a period equal to its deadline.

The RM algorithm defines each task's priority by its respective duration. Thus, the shortest task will be given the highest priority.

2) Earlier Deadline First (EDF)

EDF is an optimal mono-processor scheduling algorithm with sporadic tasks support. The EDF algorithm gives the highest priority to the task which has the closest deadline.

The algorithm is preemptive. It means that some tasks may be interrupted in order to process other tasks with a higher priority. So, unlike the static priority of the RM algorithm, the EDF algorithm is driven by "*dynamic priority in the sense that the priority of a request is assigned as the request arrives*" [5].

The assumptions made for the RM algorithm are the same for the EDF algorithm except the periodicity of tasks. So the only assumptions remaining are that each task must:

- be preemptable,
- be independent (no sequential relation between the tasks)

3) Least Laxity First (LLF)

The LLF algorithm is another optimal scheduling algorithm driven by dynamic priorities. It is based on the notion of laxity. The laxity of a task is defined as the deadline minus the remaining computation time needed to complete the task [5]. So the laxity is the maximum time that a task can wait before it becomes not possible to meet its deadline. LLF gives the highest priority to the process which has the lowest laxity. LLF is also preemptive and the assumptions are the same as those for EDF. LLF also have a better support than EDF of non-periodic tasks [5].

B. Scheduling with Hadoop

Hadoop [2] is a free Java framework implementing the MapReduce paradigm presented by Google in 2004 [1]. Thus, Hadoop allows programmers to create distributed applications with a high level of abstraction from the technical issues related to distributed systems. A Hadoop cluster is composed of one master node, and many workers. The jobs are always submitted by an user to the master which split them in Map and Reduce tasks. Then the master uses a scheduling policy depending on the needs of the user to submit each task to the workers. In this section, we present the two existing scheduling algorithms which already exist for Hadoop in order to compare them to our algorithm further in this paper.

1) Hadoop Fair Scheduler (HFS)

"*Fair scheduling is a method of assigning resources to jobs such that all jobs get, on average, an equal share of resources over time*" [6]. This method implies two problems: to be able to define how fairly a job has been proceeded, and to choose which job to run when a task slot becomes available.

The fairness can be measured by the calculation of a deficit which is "*the difference between the amount of compute time it should have gotten on an ideal scheduler, and the amount of time it actually got*" [6]. So, when a task has a large deficit, it means that it has been proceeded during less time that it should have been. This task has been treated unfairly. Thanks to the deficit of each job, the master can sort tasks by fairness and give the highest priority to the one which has been proceeded the most unfairly. The actual goal of the HFS is to minimize the deficit of each application. If the deficit equals zero for all tasks, it means that the resources have been shared in perfect proportions.

In practice, several users may need the cluster at the same time, and it is easy to imagine that the jobs will not have the same priority. So, this scheduler also implements a system of *pools* which groups some jobs together. Then, the scheduler will try to be fair between the pools. For example, let's say that each user using the cluster has got his own pool (which is the actual default configuration for HFS), the resource will be fairly shared between pools, and thus each user will dispose of a fair part of the resources to run his jobs without slowing down the other users. It's also important to note that HFS allows to give weights to pools or to jobs. The job weights can be based on a given priority or on their sizes. The scheduler fairly shares the resources between the pools, but the jobs within each pool must be scheduled as well. To solve this issue, HFS uses the exact same scheduling algorithm locally inside the pools. Thus, the resources are fairly shared between pools, and the running time of each job is fairly shared in each pool. This management allows several users to use the cluster at the same time and provides them a draft of multitasking behaviour. This algorithm is non preemptive.

2) Hadoop Capacity Scheduler (HCS)

The Hadoop Capacity Scheduler is based on the idea of a priori resource sharing. As explained in [7], the HCS implements a multiple queues support. A fraction of the available resources is allocated to each queue a priori. So, all jobs are submitted to queues, and each job in each queue is proceeded by the dedicated fraction of the resources that has been allocated to the queue. If there's unallocated resources available, it can be used by any queue beyond its guaranteed capacity.

Jobs are sorted in queues by submission time (FIFO) and optionally by job priority. The algorithm does not support preemption once a job is running.

C. Deadline scheduling implementation issues on distributed systems

Hadoop schedulers have been implemented in order to satisfy the needs of different companies (Amazon, Facebook, etc). The Hadoop Fair Scheduler provides short response times to small jobs in a shared Hadoop cluster. It also improves utilization over private clusters or Hadoop On Demand [8]. The features of the Capacity Scheduler are close with the Fair Scheduler, but the implementation is different.

None of these schedulers take care of some time constraints for the jobs. We have shown that HFS aims to share the resource fairly between the jobs while HCS works with pre-defined resource allocation, but none of these schedulers allows the user to specify a deadline that must be met. So, we have to implement a new scheduler in order to use this feature in Hadoop. But the three deadline schedulers presented in section II**Error! Reference source not found.** are not directly applicable to a distributed environment. If we take the example of the LLF algorithm, we can see that there are many important assumptions that are not true in our case.

We define the following scenario to show how a classic LLF algorithm would work in a Hadoop environment: we have a cloud composed of a number $n_{VM}$of virtual machines (VM), each VM has one processor, and we have a number n_C of cloud jobs (cloudlets) to run. Each cloudlet C_i has a deadline D_i. Our aim is to meet the deadlines of each cloudlet. Assuming that $n_C > n_{VM}$, if we use the LLF algorithm to schedule the cloudlets, while each VM will be running one cloudlet, the other cloudlets will wait in queue sorted by laxity (at any time). Then, two events are possible :

- *Event 1*: One of the running cloudlet finishes. Then one VM becomes idle, so the scheduling algorithm will start the first cloudlet of the waiting queue into the newly idle VM.
- *Event 2* : One of the waiting cloudlets (named C_A) has a lower laxity than one of the running cloudlets (named C_B currently running on VM_1). So, the scheduling algorithm will pause C_B and run C_A on VM_1 instead. Then C_B will be put into the waiting queue. Assuming that C_B is the first element of the queue, if an *event 1* happens, then C_B will have to run on a VM which might be different than VM_1.

In this scenario, both events are unrealistic in a cloud. Indeed, during the *event 1*, the scheduling algorithm starts the task with the least laxity on any idle VM. But this behaviour does not take care of the data locality. As explained in [9], the data management on a Hadoop cluster is done thanks to a distributed file system. Each chunk is duplicated several times on different hosts (three times by default on HDFS). So, assuming that a cloudlet is actually an operation on a specific chunk, it cannot be executed on any other host. The cloudlet must be ran on a machine which has a local copy of the data, or the chunk must be transferred to an idle VM.

During the *event 2*, the cloudlet C_B starts its execution on one VM, is then preempted, and finally finishes on another VM. Once again, this procedure is unrealistic. The problem of the data locality put in evidence on the *event 1* remains true in this case : C_B cannot be resumed on any VM. Moreover, although it is possible to implement a pause/resume task feature on a different VM by using the procedure of task migration (or VM) [16]**,** this action has an important cost while LLF is clearly defined for a null cost of task migration.

So, the first conclusion is that LLF is obviously inappropriate to schedule tasks within a cloud. Arguably enough, all other algorithms presented earlier will also encounter the same issues.

A list of other problem related to deadline scheduling in Hadoop is in [9]. The main problem highlighted in [9] is about *predictability* of the execution time needed for a task

The number of slots that have been defined on the worker is very important for the completion time of task. Let's imagine a worker with n_{PE} available Processor Elements (PE), and n_s available slots. The *slot-to-core* ratio will be $\frac{n_S}{n_{PE}}$. It means that the local operating system of the worker will have to schedule the execution of the n_S tasks on n_{PE} PE. Consequently, if $n_S > n_{PE}$, the execution times of the tasks on the workers are closely linked to the OS scheduling algorithm (which might be different between workers). Moreover, a large number of slots on one worker can increase the number of disk access and makes the execution time evaluation even harder. Once again, it depends on the local OS management. The influence of the Slot-to-Core ratio and the Multiple Concurrent Jobs have been studied in [9].

III. The Cloud Least Laxity First (CLLF) Proposal

In this section, we present our scheduling algorithm which solves the issues highlighted above. We firstly discuss the assumptions made during the design of the algorithm and we then present the scheduling policies on the master and on each worker.

A. Assumptions

CloudLLF is a non-optimal distributed deadline scheduler for soft deadline tasks in the sense of the definition given in section II.

We consider a cloud composed of n_{VM} virtual machines. Each VM has a number of Processor Elements (PE) n_{PE} which are all defined by a number of millions of instructions that can be executed during one second (MIPS). Moreover, each VM is also defined by an amount of memory (RAM), a bandwidth (BW) and a capacity of storage (HDD). Each VM runs on a single host. As our proposed algorithm tries to provide a deadline meeting behaviour to the cloud, the predictability of the total execution time of each task remains one of the most important assumptions we have made. Running several VMs on one single host implies a local scheduling by the host's OS which considerably increases the hardness of the prediction of the completion time of a task. We then decided to consider one VM by host with the exact same hardware characteristics.

Thanks to the conclusions related to the Slot-to-Core Ratio made in section II.C we decided to allow a number of concurrent tasks on a VM equal to n_{PE}.

Finally, as described in [10], we consider that each data chunk is duplicated n_{CHK} times on the hosts.

B. Local scheduling on nodes

The local scheduling on each worker is really simple. It is a First-In-First-Out (FIFO) queue as described by the Algorithm 1. The node notifies the master when one or more PE is idle and if the node receives a new task while processing the maximum number of concurrent tasks, the new task waits in the FIFO queue for one PE to become idle. It is important to notice that during a normal utilisation, the waiting queue size should always equals zero thanks to the global scheduling algorithm.

```
Data :
    - $c_{LIST}[\,]$ : Local cloudlets queue (in form of a list).
    - $v_{PE}[\,]$ : The state of each PE (idle/busy) on the VM (in form of an array).
    - $x$ : Iteration variable (Integer)

Procedure :
1   For $x < length\ of\ v_{PE}, x \in \mathbb{N}$ do
2   .   If $v_{PE}[x]\ is\ idle$ then
3   .   .   Run $c_{LIST}[0]$;
4   .   .   Remove $c_{LIST}[0]$ from $c_{LIST}$;
5   .   .   Set $v_{PE}[x]$ as busy;
6   .   End if
7   End for
```

Algorithm 1: Node FIFO scheduling algorithm

C. Global scheduling on the master

On the master node the global scheduling algorithm is hosted. We choose a derivation of the LLF algorithm presented in II.A.3). The main difference between LLF and our algorithm is about preemption. We decided to implement a non-preemptive algorithm because on a multiprocessor algorithm, one of the assumptions is the following. If a task is paused on one processor, it can be resumed on an other processor with a negligable cost (as explained in [11]). This assumption cannot be transposed to a distributed system that uses data locality management. If a task is paused on a node, this task can only be resumed on the same node. Indeed, there's only few nodes having the task data locally, and among them, the only one to know the current progression of the task is the one which started it. If the resuming is done on an other node, the task will have to restart from the beginning, so, the assumption made earlier cannot be assumed any more. That's why we decided to forbid the preemption in our algorithm.

The general idea of the algorithm is to sort the cloudlets by laxities (the first has the lowest one). Giving this sorted list, the algorithm takes the first element of this list and looks for a host that locally have the data of the cloudlet and which also have at least one free slot. If one matching host is found, the task is ran on it, otherwise, the algorithm restart the same procedure using the second element of the list. The implementation is described by the pseudo-code of the Algorithm 2.

```
Data :
    - $c_{LIST}[\,]$ : Remaining cloudlets (in form of a list)
    - $c_{LAX}$ : The current laxity of the cloudlet
    - $c_{LOC}[\,]$ : The ids of the VMs which locally have the cloudlet data (in form of an array)
    - $v_{LIST}[\,]$ : The available VMs (in form of an array)
    - $v_{PE}[\,]$ : The state of each PE (idle/busy) on the VM $v$ (in form of an array).
    - $v$ : Iteration variable (VM)
    - $x$ : Iteration variable (Integer)

Procedure :
1   Update $c_{LAX}$ of $c_{LIST}$ with the current time ;
2   Sort $c_{LIST}$ by $c_{LAX}$ ;
3   For $v \in v_{LIST}$ do
4   .   For $x < length\ of\ c_{LIST}\ \&\ v_{PE}\ has\ one\ idle\ PE$ do
5   .   .   If $v.id \in c_{LIST}[x]_{LOC}$ then
6   .   .   .   Run $c_{LIST}[x]$ on $v$ if $v_{PE}$;
7   .   .   .   Remove $c_{LIST}[x]$ from $c_{LIST}$ ;
8   .   .   .   Set one more PE used on $v_{PE}$;
9   .   .   End if
10  .   End for
11  End for
```

Algorithm 2 : Master CLLF scheduling algorithm

Algorithm 2 is derivated from the classic mono-processor LLF algorithm. It is important to say that this algorithm must be executed periodically, or on events (completion of a task, arrival of a new task). In the fisrt case, the study in [10] of the influence of the Slave-to-Master Heartbeat interval remains true.

The differences between our algorithm and a classic LLF algorithm are the data locality management and the non-preemptive behaviour of our algorithm. Those differences allow CLLF to avoid the issues highlighted in section II.C. We have shown in the previous algorithms that once a task started on a node, it can't be stopped until its completion, and each task must be ran on a worker which owns a local copy of the data chunk corresponding to the task. Unfortunately, those characteristics make CLLF loose its theorical optimality, but the performances observed in practice are motivating.

We remark to give the algorithm a behaviour that handles both static and dynamic priorities is quiet trivial. Indeed, assuming that each task has been defined with a given static priority $p \in \mathbb{N}$, where $p = 1$ is the least, the tasks might be sorted not only using their current laxities, but also using their static priorities.

Thus, we can define a rank R_i for each task τ_i with a laxity L_i and a priority P_i as in equation (1).

$$R_i = \frac{L_i}{P_i} \quad (1)$$

So, the algorithm remains the same except that the tasks are now sorted by rank (the lowest first) instead of laxity.

D. Global characterization

Figure 1 illustrates a low-level global architecture of the system. It shows the path followed by the cloudlets. All cloudlets known before the execution of the algorithm are sorted by laxity in a queue and are then sent to idle VMs. An online cloudlet is directly inserted to the laxity-sorted queue at the right place and is then sent to a VM as well. Figure 1 also demonstrate the implementation to avoid the Slot-to-Core ratio issue discussed in II.C.

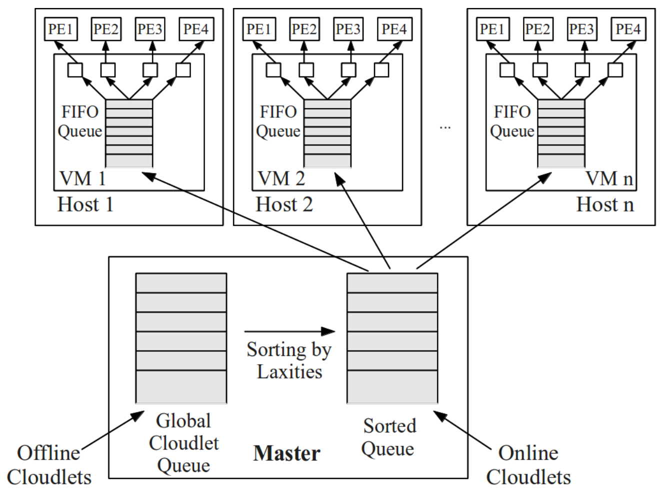

Figure 1: Global representation - N hosts - 4 PEs by host

TABLE I. shows the main differences between the Hadoop schedulers, the CloudSim schedulers and our scheduler.

It appears that the Fair Scheduler and the Capacity Scheduler have a respectively a close behaviour to Time Shared and Space Shared. The main difference is about the data locality management. It is also important to notice that none of the algorithms is preemptive. Moreover, we can observe that our algorithm is the only one which has been designed with a deadline-meeting approach. We will explain in section IV.D that this behaviour allows this algorithm to minimize the cost overhead implied when a task misses its deadline.

TABLE I. COMPARISON OF THE SCHEDULERS

	Hadoop		**CloudSim**		**CLLF**
	Fair Scheduler	***Capacity Scheduler***	***Time Shared***	***Space Scheduler***	
Pre-reserved Ressources	-	×	-	×	-
Fair on exec. time	×	-	×	-	-
Data locality management	×	×	-	-	×
Static priorities management	×	×	-	-	optional
Deadline aware	-	-	-	-	×
Preemptive	-	-	-	-	-

IV. Performances

In this section, we analyse the practical capabilities of CLFF. We firstly present the experiment scenario and we then compare the perfornances (deadline meeting, execution time, cost, data locality sensibility) of CLLF to the Space Shared and Time Shared scheduling algorithm of Cloudsim.

A. Scenario

The performance measurement has been realized using the CloudSim [4] framework.

The aim of this experiment is to show the ability of CLLF to meet deadlines in situations where the other algorithms (Time Shared and Space Shared) cannot. To achieve this, we propose the following scenario. We define an homogenous datacenter composed of n_{HOST} hosts. Each host has the following hardware specification :

- A number $n_{PE} = 4$ of processing element. Each PE has a speed of 800 mips.
- A Random Access Memory (RAM) of 2048Mo.
- An available bandwith of 10Gbit/s.
- A storage capacity of 1To.

We process 2,524 cloudlets on the previously defined datacentre using three schedulers : CLLF, Time Shared and Space Shared. The cloudlets lengths are not equal and each cloudlet is given a deadline proportionnaly to its length. The data chunks are duplicated on three different hosts and we assume that the data placement has been done before the beginning of the experiment.

B. Deadline meeting

Time Shared and Space Shared are the two cloudlet scheduling policies given out of the box with CloudSim. Their accurate description are available in [12]. They can be considered quiet close from the Fair Scheduler and the Capacity Scheduler of Hadoop as we can see on the previous table. However, Space Shared and Time Shared do not handle any form of data locality management. It's important to notice that during the experiment, our algorithm did handled the data locality problem and so, the comparison of the results is not really fair for our algorithm regarding that it had to solve a harder problem than the other algorithms. Despite this unfairness, we show that the behaviour of our algorithm remains more efficient to meet the deadline of each cloudlet.

We compare the number of deadline missed by each scheduler for the given set of cloudlets and a variable number of hosts.

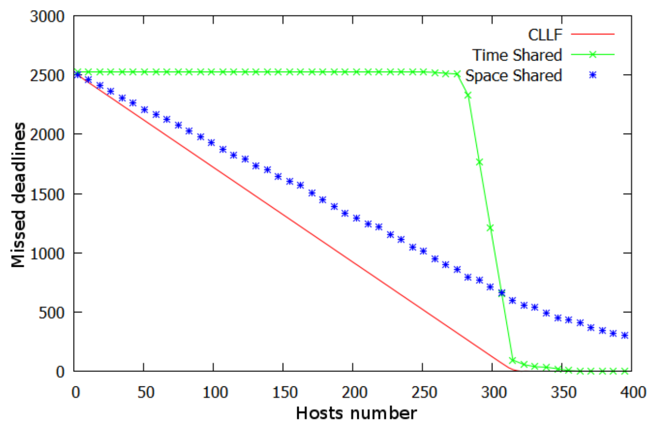

Figure 2 : Comparison of the missed deadlines

On the Figure 2 is shown the result of this experiment. On the x axis is the number of hosts used (3 to 400) and on the y axis is the number of deadlines missed. According to these results, we can clearly see that the Time Shared algorithm becomes totally inefficient under one certain number of hosts. In TABLE II. are the results extracted from the experiment that uses 280 hosts.

TABLE II. RESULT FOR 280 HOSTS

	Number of missed deadlines	Missed deadlines ratio
Time Shared	2512	99.5%
Space Shared	827	32.8%
CLLF	301	11.9%

We can notice in this example that Time Shared misses almost all the deadlines (which is the worst possible result) while Space Shared is a little bit more efficient but still misses more than two times more deadlines than our algorithm (which is disadvantaged because of the data locality problem). The second important result is the minimum number of host to meet all the deadlines, and once again, our algorithm remains the best. In the TABLE III. are the accurate results for the experiment using 321 hosts. We can observe that our algorithm meets all the deadlines while Time Shared has a pretty good performance but remains not perfect and Space Shared still misses the deadline of more than one cloudlet out of five.

TABLE III. RESULT FOR 321 HOSTS

	Number of missed deadlines	Missed deadlines ratio
Time Shared	48	1.9%
Space Shared	566	22.4%
CLLF	0	0%

C. Execution Time

We can observe on the Figure 3 the result of an other experiment. In this case, we tried to measure the total execution time of the job (all the cloudlets processed) for a variable number of host. On Figure 3, the x axis is the number of hosts used (3 to 400) and the y axis represents the total execution time.

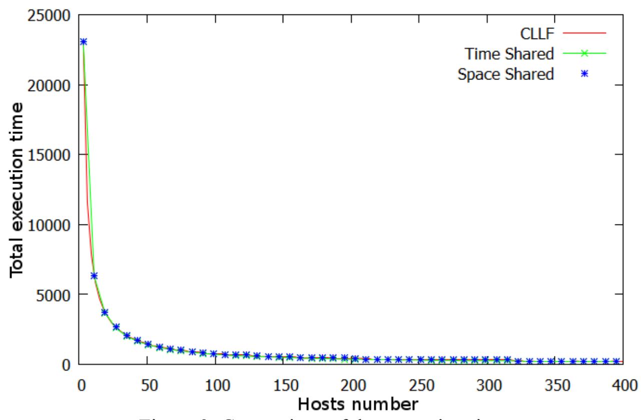

Figure 3: Comparison of the execution time

In this case, it appears clearly that the execution times of all the three algorithms seems to be quiet identical. Once again, it is important to remember that only our algorithm dealt with the data locality problem which may imply an overhead on the total completion time. Despite this, we observe performances which are between Space Shared and Time Shared.

D. Costs

CLLF is a scheduler able to process soft deadline tasks. Such a task has penalties when it misses its deadline. This penalty is a function of the lateness of the task. So, it is interesting to analyse the percentage of the computational power used to process tasks after their deadlines in order to measure the performance of CLLF to minimize the lateness of the tasks. We call the percentage of the computation time used to process a task after its deadline the *cost* of the task.

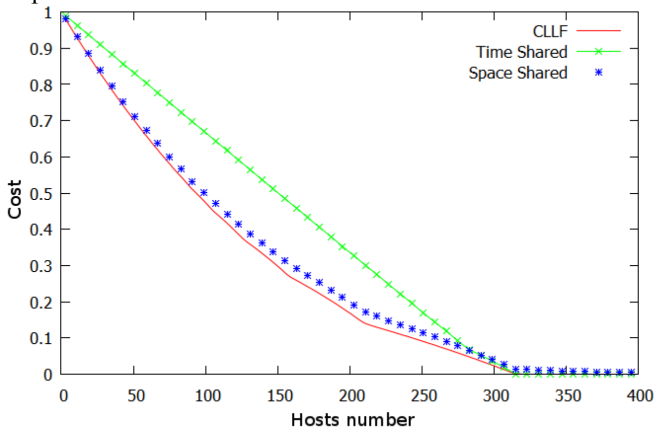

Figure 4 : Comparison of the costs

For example, a task τ, with a deadline D_τ, has been processed during T seconds. We can distinguish two cases :

- $T \leq D_\tau$: the task has been finished on time, the cost is null.
- $T > D_\tau$: the task has overpassed its deadline, so the lateness of the task is $L_\tau = T - D_\tau$. The penalty

received by τ is then $P_\tau = f(L_\tau)$ with $f(x)$ being the penalty function. So finally, the cost is $C_\tau = \frac{L_\tau}{T}$.

On the Figure 4 is shown the sum of the costs of each cloudlet implied by each algorithm in function of the number of hosts. We can see that the best algorithm to minimize the costs is CLLF. The integrals of the three curves on the Figure 4 are shown on the TABLE IV.

TABLE IV. INTEGRALS OF THE COST'S CURVES

	Cost Integral	**Normalized cost**
Time Shared	165.56	1.29
Space Shared	134.37	1.05
CLLF	127.9	1

E. Data Locality

One of the most important characteristics of our algorithm is that a cloudlet cannot run on a host which do not locally have the corresponding data. The data move to an idle host is forbidden in order to solve issues described as follows.

The main reason of this decision is related to the predictability of the transfer time. Indeed, the move of a cloudlet to a special host implies an upload time. But, this duration is closely bounded to the network state at the moment of the transfer. Moreover, in practice, the network bandwidth might be used by another application. So, it becomes almost impossible to predict accurately the uploading time and regardless of this, the deadline of the moving cloudlet will not change. In this case, the laxity is described by equation (2).

$$\boldsymbol{Laxity} = \boldsymbol{Deadline} - (\boldsymbol{Execution\ time} + \boldsymbol{Transfer\ time}) \quad \textbf{(2)}$$

Since the transfer time is unpredictable, the laxity becomes undefinable. This vagueness is not acceptable in our case, and this is why we decided to avoid it. It seems obvious that the data placement has an important role in the algorithm's efficiency. Indeed, if a host has locally only very long cloudlets, there is a lot of chance that it will be the last host to finish its tasks, and so it will increase the total duration time of the job. So, it appears essential to make a decision when choosing the machines on which a dataset will be hosted regarding on which datasets are already located on those hosts in order to avoid overloaded hosts.

For our simulation, we implemented an intuitive simple algorithm which provides acceptable results. The pseudo-code which describe it is in the Algorithm 3. The algorithm is used to choose on which VMs should be duplicated the data chunks. For each chunck, with n_D duplication required, the algorithm sums the already present cloudlets lengths on each VM, and looks for the n_D VMs with the minimum sum (charge).

We can clearly see the influence of the data locality on the global performances of the algorithm. The pseudo periodicity of the Figure 5 has a period of 3 hosts, which is also the number of duplication of the data chunks for this experiment.

Data :

- $c_{LIST}[\,]$: List of cloudlets (in form of an array).
- $c_{LOC}[\,]$: The VMs's ids which locally have the cloudlet (in form of an array).
- c_{LENGTH} : The length of the cloudlet.
- v_{CHARGE} : The charge of the VM (the sum of its local cloudlets's lengths).
- n_D : The number of duplication of each data (3 by default).
- x, y : Iteration variables (Integers)
- v_{CUR} : Temporary variable (VM) representing the currently selected VM.

Procedure :

```
1   For $x < length\ of\ c_{LIST},\ x \in \mathbb{N}$ do
2   .   For $y < n_D,\ y \in \mathbb{N}$ do
3   .   .   Mark as $v_{CUR}$ the VM with the $v_{CHARGE}$ min ;
4   .   .   $c_{LIST}[x]_{LOC}[y] \leftarrow v_{CUR}$ ;
5   .   .   $v_{CUR_{CHARGE}} \leftarrow v_{CUR_{CHARGE}} + c_{LIST}[x]_{LENGTH}$ ;
6   .   End for
7   End for
```

Algorithm 3 : Data placement decision

After several experiments, we remark that the best performances depend on the criteria of equation (3).

$$n_{HOST} = \alpha \times n_{CHK}, \alpha \in \mathbb{N} \quad (3)$$

The α in the equation (3) is an integer. It means that the best performances are met for a number of hosts which is a multiple of the number of data chunks duplications. On the Figure 6 is a 3D representation of the experiment result. On the first axis is the number of data chunk duplication, on the second axis is the number of hosts in the datacenter and on the third axis is the number of missed deadlines.

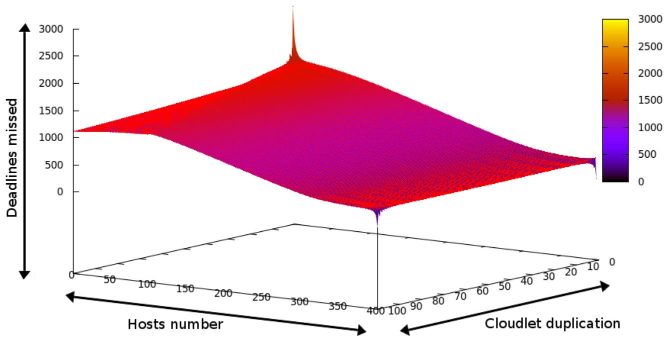

Figure 6 : Influence of the data chunk duplication

We can see that the maximum number of missed deadlines happens for a minimum number of hosts and duplications. And the best performances are reached for a maximum number of hosts and duplications. As CLLF runs the tasks on VMs which have a local copy of the tasks data, the more duplication of chunks there is, the more solution CLLF has. So, an important diponibilty of the data allows CLLF to choose an efficient VM for each task, thus, the performances are increased.

V. Conclusion

This study presents a soft real-time scheduling algorithm for distributed systems with data locality management. The algorithm called CLLF is based on the popular LLF algorithm and has been designed to take into account the practical issues related to distributed applications. We demonstrate that LLF is not applicable out of the box on a distributed system because of its preemptive behaviour which implies a null transfert cost from one node to an other, and we notice numerous other issues about real-time scheduling on distributed system. However, we show that a deadline-meeting approach to schedule tasks over a cloud allows to minimize the extra-cost of each task while the execution time of the job remains acceptable. We demonstrate the importance of the data placement within a cloud to avoid situations where one node becomes a resource bottleneck, and we propose an algorithm for the data allotment over a distributed file system such as HDFS. The future step of our research includes integration of more advance scheduling menagemetn to include resource discovery means [17] for demonstrating the effectiveness of our deadline algorithm in dynamic node formation. In addition, we aim of applying the solution into large-scale virtualized grids [19] and inter-cloud [14] scenarios to explore the efficiency of the algorithm in higly dynamic and large-scale cases.

References

[1] Dean, J. and Ghemawat, S. (2004) “MapReduce: Simplified Data Processing on Large Clusters,” OSDI'04 Sixth Symposium on Operating System Design and Implementation, December 2004.

[2] Apache, “Hadoop Wiki Page,” [Online]. Available: http://wiki.apache.org/hadoop/, Accessed 14/09/2012.

[3] CLOUDS Laboratory (2012), “CloudSim: A Framework For Modeling And Simulation Of Cloud Computing Infrastructures And Services,” January 2012. [Online]. Available: http://www.cloudbus.org/cloudsim/, Accessed 14/09/2012.

[4] Mohammadi, A. and Akl, G. S. (2005) “Scheduling Algorithms for Real-Time Systems,” July 2005.

[5] Apache, “Capacity Scheduler Guide,” February 2010. [Online]. Available: http://hadoop.apache.org/common/docs/r0.20.2/fair_scheduler.html, Accessed 14/09/2012.

[6] Apache, “Capacity Scheduler Guide,” February 2010. [Online]. Available: http://hadoop.apache.org/common/docs/r0.20.2/capacity_scheduler.html, Accessed 14/09/2012..

[7] Zaharia, M. (2009) “Job Scheduling with the Fair and Capacity Schedulers,” in Hadoop Summit 2009.

[8] “HDFS Federation,” May 2012. [Online]. Available: http://hadoop.apache.org/common/docs/current/hadoop-yarn/hadoop-yarn-site/Federation.html, Accessed 14/09/2012.

[9] Phan, T. L., Zhang, Z., Loo, T. B., and Lee, I. (2010) “Real-Time MapReduce Scheduling,” University of Pennsylvania Department of Computer and Information Science Technical Report, Vols. MS-CIS-10-32, October 2010.

[10] Ghemawat, S., Gobioff, H. and Leung, T. S., (2004) “The Google File System,” Google Research Publicactions.

[11] Cho, H., Ravindran, B. and Jensen, D. E. (2006) "An Optimal Real-Time Scheduling Algorithm for Multiprocessors", Proceedings of the 27th IEEE International Real-Time Systems Symposium.

[12] Apache, “Hadoop Wiki - PoweredBy,” [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy/, Accessed 14/09/2012.

[13] Sotiriadis, S., Bessis, N. and Antonopoulos, N. (2012). Decentralized Meta-brokers for Inter-Cloud: Modeling Brokering Coordinators for Interoperable Resource Management, 9th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD'12), May 29-31, Chongqing, May 29 – 31 2012, pp. 2475-2481.

[14] Sotiriadis, S., Bessis, N. And Antonopoulos, N. (2011). Towards inter-Cloud Schedulers: A Survey of meta-Scheduling Approaches, 6th IEEE International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2011), Barcelona, Spain, October 26-30, 2011, ISBN: 978-0-7695-4531-8, pp. 59-66.

[15] Bessis, N., Sotiriadis, S., Xhafa, F., Pop, F. and Cristea, V. (2012). Meta-scheduling Issues in Interoperable HPCs, Grids and Clouds, International Journal of Web and Grid Services, Volume 8, Issue 2, Inderscience, pp. 153-172.

[16] Sotiriadis, S., Bessis, N., Xhafa, F., and Antonopoulos, N.. 2012. Cloud Virtual Machine Scheduling: Modelling the Cloud Virtual Machine Instantiation. In Proceedings of the 2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS) (CISIS '12). IEEE Computer Society, Washington, DC, USA, 233-240.

[17] Sotiriadis, S., Bessis, N., Huang, Y., Sant, P. And Maple, C. (2010). Towards to Decentralized Grid Agent Models for Continuous Resource Discovery of Interoperable Grid Virtual Organizations, International Workshop on Distributed Information and Applied Collaborative Technologies (DIACT-2010), in conjunction with the 3rd International Conference on the Applications of Digital Information and Web Technologies (ICADIWT-2010), 12th -14th July 2010, Istanbul, pp. 170-175.

[18] Huang, Y., Bessis, N., Sotiriadis, S., Brocco, A., Courant, M., Kuonen, P., And Hirsbrunner, B. (2009). Towards an integrated vision across inter-cooperative grid virtual organizations. in: Proceedings of the 1st International Conference on Future Generation Information Technology (FGIT 2009), 2nd Intl. Conference on Grid and Distributed Computing (GDC 2009), pp.120-128, Springer LNCS, Jeju island, Korea, December, 2009. pp. 120-128.

[19] Sotiriadis, S., Bessis, N., Huang, Y., Sant, P. And Maple, C. (2010). Defining Minimum Requirements of Inter-collaborated Nodes by Measuring the Weight of Node Interactions, 4th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS-2010), 15th-18th February, Krakow, pp. 291-298.

[20] Sotiriadis, S., Bessis, N., Xhafa, F., and Antonopoulos, N. 2012. From Meta-computing to Interoperable Infrastructures: A Review of Meta-schedulers for HPC, Grid and Cloud. In *Proceedings of the 2012 IEEE 26th International Conference on Advanced Information Networking and Applications* (AINA '12). IEEE Computer Society, Washington, DC, USA, pp. 874-883.