Failure detector abstractions for MapReduce-based systems

Bunjamin Memishi [a,*], María S. Pérez [a], Gabriel Antoniu [b]

[a] OEG, Universidad Politécnica de Madrid, Campus de Montegancedo, Boadilla del Monte, Madrid 28660, Spain
[b] INRIA, Campus Universitaire de Beaulieu, Rennes, Brittany 35042, France

A B S T R A C T

Omission failures represent an important source of problems in data-intensive computing systems. In these frameworks, omission failures are caused by slow tasks, known as strag- glers, which can strongly jeopardize the workload performance. In the case of MapReduce- based systems, many state-of-the-art approaches have preferred to explore and extend speculative execution mechanisms. Other alternatives have based their contributions in doubling the computing resources for their tasks. Nevertheless, none of these approaches has addressed a fundamental aspect related to the detection and further solving of the omission failures, that is, the timeout service adjustment.

In this paper, we have studied the omission failures in MapReduce systems, formalizing their failure detector abstraction by means of three different algorithms for defining the timeout. The first abstraction, called High Relax Failure Detector (HR-FD), acts as a static alternative to the default timeout, which is able to estimate the completion time for the user workload. The second abstraction, called Medium Relax Failure Detector (MR-FD), dy- namically modifies the timeout, according to the progress score of each workload. Finally, taking into account that some of the user requests are strictly deadline-bounded, we have introduced the third abstraction, called Low Relax Failure Detector (LR-FD), which is able to merge the MapReduce dynamic timeout with an external monitoring system, in order to enforce more accurate failure detections.

Whereas HR-FD shows performance improvements for most of the user request (in partic- ular, small workloads), MR-FD and LR-FD enhance significantly the current timeout selec- tion, for any kind of scenario, regardless of the workload type and failure injection time.

1. Introduction

Omission failures happen when a process does not send (or receive) a message that is supposed to send (or receive). In distributed systems, in order to detect and marginalize the impact of omission failures, any framework needs to consider the failure detector abstraction [9]. Failure detectors are abstract devices that offer information about the operational status of processes in a distributed system. It is believed that the failure detector abstraction is fundamental and should be considered as a first-class citizen entity of any distributed computing framework [20].

Failures are often detected by using a static or dynamic timeout service, which is enabled by a heartbeat mechanism. In the static configuration, the timeout parameter is set up when the job starts and is not changed until the job execution

* Corresponding author.
E-mail addresses: bmemishi@fi.upm.es (B. Memishi), mperez@fi.upm.es (M.S. Pérez), gabriel.antoniu@inria.fr (G. Antoniu).

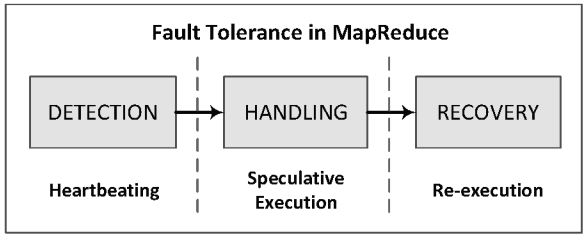


Fig. 1. Fault tolerance in MapReduce: The basic fault tolerance definitions (detection, handling and recovery) with their corresponding implementations.

finishes. It is well known that a static timeout value applicable to any application, infrastructure or networking environment does not exist [31]. This is due to its limitations: firstly, that value is not applicable to all the scenarios, and secondly, even if the timeout value would have been chosen well at the beginning, the application, infrastructure and the networking environment may suffer changes (e.g. failures, delays, etc.).

Failure detectors may be divided in perfect or eventual. Perfect detectors may report some process to have crashed, immediately with the first signs of unresponsiveness, while eventual detectors report a level of suspect. In failure detection, there are two metrics that provide the correctness of the mechanism [10]: (i) *Completeness*, which requires that a heartbeat-based detector eventually suspects every process (task) that actually crashes; and (ii) *Accuracy*, which restricts the mistakes that a heartbeat-based detector can make.

In the case of MapReduce [15], a relevant framework for Big Data processing, omission failures are due to stragglers. The concept of stragglers is very important in the MapReduce community, especially task stragglers, which can jeopardize the job completion time. Stragglers are common when MapReduce is deployed on a large-scale infrastructure, consisting of commodity hardware. Typically, the main causes of a MapReduce straggler task are the existence of a slow node, network overload or input data skew [7].

Foreseeing MapReduce usage in the next generation Internet [26], a particular concern is the aim of improving the MapReduce's reliability by providing better fault tolerance mechanisms. Whereas the handling and recovery in MapReduce fault-tolerance via data replication and task re-execution seem to work well even at large scale [3,23,37], there is relatively little work on detecting failures in MapReduce. Accurate detection of failures is as important as failures recovery, in order to improve applications' latencies and minimizing resource waste.

As far as we know, there is not a formalization of the failure detector abstraction in MapReduce-based systems. This is mainly due to the difficulty of modeling a framework for an environment with strictly dynamic requirements. In this paper we study the omission failures in MapReduce systems, formalizing their failure detector abstraction by means of three different algorithms, namely HR-FD (High Relax Failure Detector), MR-FD (Medium Relax Failure Detector) and LR-FD (Low Relax Failure Detector), according to the time heterogeneity, i.e., how much relaxation the system is capable to tolerate in order to detect an omission failure.

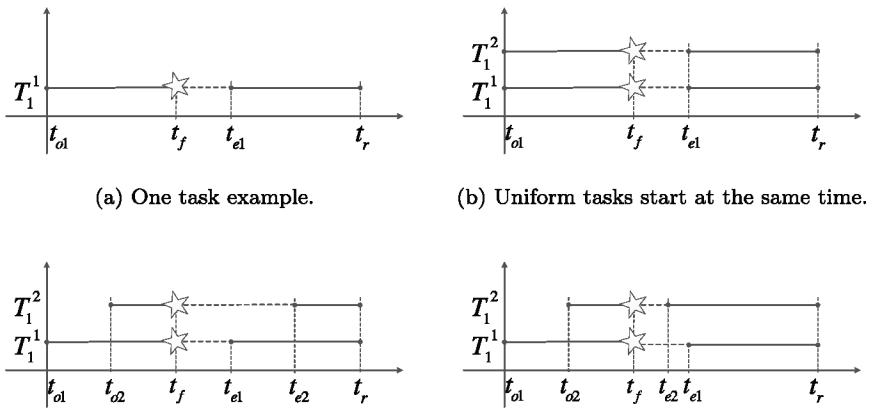
The rest of the paper is as follows. In Section 2, we specify the problem we aim to solve. After that, in Section 3 we define the system model in which the different modules (HR-FD, MR-FD and LR-FD) are based on. These modules are described and evaluated in Sections 4, 5 and 6. Section 7 describes the related work. We conclude the paper in Section 8.

2. Problem statement

The heartbeat is the concept in which the MapReduce failure detection mechanism is based, as shown in Fig. 1. Any kind of failure that is detected in MapReduce has to fulfill some preconditions, in this case to miss a certain number of heartbeats, so that the other entities in the system detect the failure. In addition to let the master know that a worker is alive, heartbeats also can be used as an additional channel for messages [33].

Currently, a static timeout based mechanism is used for detecting fail-stop failures by checking the expiry time of the last received heartbeat from a certain machine. In Hadoop (both Hadoop 1.0 and Hadoop YARN), each worker sends a heartbeat every 3 s and the master checks the expiry time of the last reported heartbeat every 200 s. If no heartbeat is received from a machine for 600 s, then this machine will be labeled as a failed machine and therefore the master will trigger the failure handling and recovery processes. However, some studies have reported that the current static timeout detector is not effective and may cause long and unpredictable latency [17,18]. Our studies in [24] report that, in the presence of a single machine failure, the applications' latencies vary not only in accordance to the occupancy time of the the failure, similar to [17], but also vary with the job length (short or long).

An additional important factor is the non-uniformity of straggler tasks. If we assume that a cloud infrastructure implements a shared Hadoop [2] cluster, it is possible that some users may be willing to run a heavy CPU job request. If its workload is large, it may need many virtualized tasks, spread among many machines. If other users are running different tasks on these machines, at some point in time, the cluster will have a large number of dynamic stragglers. To worsen the scenario, the percentage of stragglers could be the majority set of job tasks, that is, the stragglers number could be big-



(a) One task example. (b) Uniform tasks start at the same time.

(c) Uniform tasks start at different time. (d) Non-uniform tasks start at different time.

Fig. 2. Assumed timeout reaction to different task scenarios.

ger than or equal to the number of normal tasks. In this case, it is unlikely that a speculative execution could work well, because it would starve cloud resources.

Let us explain the failure timeout problem with an example, shown in Fig. 2.

Case 2a. We consider a single task T_1^1 (subscript is the task number, whereas superscript is the job number) of job J_1 to start on a worker m at time t_{o1}. If a worker crashes at time t_f, according to its progress score, the task T_1^1 is assumed to have finished around 70% of the load at time t_f. However, after the failure, a task will need to wait 10 min of the arranged timeout to finish in time t_r, and not in time t_{e1}. In this case, it is reasonable to assume that t_{e1} is proportional to the 30% left of the missing load, and obviously less than a 10 min timeout.

Case 2b. We consider two uniform tasks, T_1^1 and T_1^2 of different jobs, J_1 and J_2 respectively, to start on a single worker m at the same time t_{o1}. If a worker crashes at time t_f, both T_1^1 and T_1^2 will enforce their respective jobs to wait the 10 min timeout, and consequently, to extend their completion time equally, from t_e1 into t_r.

Case 2c. We consider two uniform tasks, T_1^1 and T_1^2 of different jobs, J_1 and J_2 respectively, to start on a single worker m, but at different times (t_{o1}, and t_{o2}, respectively). If a worker crashes at time t_f, both tasks will cause their respective jobs to respect the 10 min timeout, and both extend their completion time equally, from different t_{e1} and t_{e2} into t_r. In this case, it is clear that T_1^1 has started earlier than T_1^2. Therefore, the first one normally would have finished earlier at time t_{e1}. But the present timeout adjustment harms the T_1^1 equally as T_1^2, by giving advantage to the second task T_1^2, which is not fair.

Case 2d. We consider two non-uniform tasks, T_1^1 and T_1^2 of different jobs, J_1 and J_2 respectively, to start on a single worker m, and at different times (t_{o1}, and t_{o2}, respectively). If a worker crashes at time t_f, both tasks will cause their respective jobs to wait the 10 min timeout, and both extend their completion time equally, from t_{e1} and t_{e2} into t_r, as previously mentioned. In this case, it is clear that T_1^1 has started earlier (t_{o1}) than T_1^2 (t_{o2}). However, T_1^2 is shorter, and normally it should have finished earlier at time t_{e2}. But the present timeout adjustment harms the T_1^2 more than T_1^1, by giving advantage to the first task T_1^1, which is not fair again.

The static timeout implemented in MapReduce is not capable to address any of the cases above. An accurate timeout detector is important not only to improve application's latency but also to improve resource utilization, especially in the Cloud where we pay for the resources we use. Therefore, we state that a significant potential exists for performance improvement in applications, particularly MapReduce applications, when choosing the appropriate timeout failure detector. We believe that a new methodology to adaptively tune the timeout detector can significantly improve the overall performance of the applications, regardless of their execution environment.

3. System model

Our system model is an abstraction of a single MapReduce job in execution. We consider that each MapReduce request is composed of N_T limited number of identified processes (slave tasks, worker tasks, or tasks in general) to be run. One of these processes is the master process (leader process, master task, or master) T_M, which controls the other workers tasks T_W. During each MapReduce request, the framework's first duty is to initiate the master process. The master is the responsible for executing the failure detection on other slave processes. In this case, we consider that the master process is always alive, and correct. In a previous work [25], we addressed the failure handling of masters. However, in this work, we assume that

Table 1
The probable task intersection between normal and suspected set.

Normal	Suspected	Result
$\in$	$\in$	*suspected* $\setminus$ {*task*}
$\in$	$\notin$	✓
$\notin$	$\in$	✓
$\notin$	$\notin$	suspected $\cup$ {task}

Table 2
The probable task intersection between suspected and speculated set.

Suspected	Speculated	Result
$\in$	$\in$	✓
$\in$	$\notin$	speculated $\cup$ {task}
$\notin$	$\in$	–
$\notin$	$\notin$	✓

the master does not fail. Therefore, during the entire job execution, there is no leader election or any other algorithm that executes in the background to replace the master.

After master initiation from the application manager, the scheduler allocates $N_T - 1$ slave processes. At any time t, a master monitors and coordinates a set of D number of worker tasks ($D \subseteq N_T - 1$), by ensuring and enforcing the correct functioning of each worker task, until they finish the work partition that was assigned to them. During the job execution, if a slave has terminated its task, the scheduler decides whether to assign another task to the slave or terminate it. When all the slaves have finished, the master delivers the output to the application manager.

Unless explicitly stated, it is assumed that a cluster S consists of a limited amount of uniform computing machines n, which can execute a limited number of concurrent processes. We consider that each failure detector algorithm is aware and dependent on the timing assumptions, and not on the resource utilization. However, it is assumed that the more the algorithm relaxes its timeout, the less amount of resources will be requested.

For example, we expect that HR-FD uses less amount of resources. On the other side, the LR-FD algorithm is assumed to request more resources, because the timing assumptions are stricter and therefore, more speculative executions will be needed.

By default, we consider that none of the slave tasks is considered for speculation, without its exclusion from the set of normal tasks. In other words, the failure detector should rearrange the suspected task from the normal set into the suspected set. Only after this, a task may get into the queue of tasks for speculation. All these possible intersections are stated in Table 1 and Table 2. According to these results, the failure detector mechanisms should react to two scenarios from Table 1, because the other two scenarios belong to the normal functioning of the MapReduce framework. If a task is included in both normal and suspected sets, this task should be deleted from the suspected set. On the other hand, if a task is not included in any of these sets, this should be added to the suspected set. Table 2 shows that only one scenario requires a solution, namely, when a task belongs to the suspected set, but still has not been speculated. In this case, the task should be added to the speculated set. Two other scenarios (1 and 4) are completely correct, whereas the third scenario is not possible, because the failure detector mechanism does not allow a task to directly switch from the normal set to the speculated set, before being included into the set of suspected tasks.

We assume that our system model does not have network failures. This implies that if we have an operational task, this accomplishes the heartbeat mechanism, and if a task is non operational, the heartbeat is missed. As we use the heartbeat as basic mechanism for our model, the appearance of network failures would affect our accuracy. Although these failures are out of the scope of this contribution, an interesting research line related to them could be explored as future work.

In principle, if the network is not reliable, it is impossible to guarantee any Quality of Service (QoS) of any application, including MapReduce applications. MapReduce systems are capable of achieving a certain equilibrium in network usage through data locality and similar techniques [3,36]. Nevertheless, if the MapReduce framework needs to deploy any algorithm of the present contribution, apart from the network issues, it is necessary also to consider if these resources are coming from a single or multiple sites or clouds, if the clouds are private or public, and the network QoS that each of them could guarantee. In the simple case of a single master, managing worker tasks from two different clouds, the timeout adjustment can vary for both of these clouds. The information about these adjustment will be very relevant in order to adjust the timeout from the master side.

3.1. Simulation environment and experimental process

In the next sections, we describe the different modules of our failure detector abstraction. In order to validate our algorithms, we have performed a set of experiments that compare them with Hadoop MapReduce. These experiments have

been made by means of statistical-based simulations. In order to make this evaluation, we have considered 25 containers, which are able to simultaneously run 25 Map (with an input split of 128 MB) or Reduce tasks. This number of containers is sufficient to evaluate the approach, considering 25 containers per workload. We consider that every Map and Reduce container is the same, and can execute a particular portion (split) of the workload. The application master manages both Map and Reduce tasks. Commonly, the number of mappers is bigger than the number of reducers. Because both phases are run sequentially, we consider all the iterations to run the Map phase, except the last iteration, which runs the Reduce phase.

An important parameter to take into account is the workload size. In this case, we consider a workload size of 12 GB, which is an average workload size of important production clusters [8]. The workload is used as input of a *sort* application, which is a common MapReduce application for benchmarking the respective systems [33]. The estimated completion time of the workload is 5 min.

For the evaluation, we will mainly focus on the worst case task, since our goal is to measure the performance of our algorithms in terms of the workload completion time. In order to do this, we compare failure free workloads with single failure workloads. This failure is injected on a random task during the iterations.

4. High relax failure detector

In this module of the framework, called high relax failure detector (HR-FD), we extend the default functioning of the MapReduce failure detector mechanism. Particularly, since the default timeout of Hadoop MapReduce has a static based timeout mechanism of 10 min, we leave this value as it is, but only for large jobs, that is, those ones whose completion time is above this value. For other jobs, whose completion time is below the value of 10 min, the timeout should be adjusted according to the estimation of the job completion time.

In this way, the failure detector timeout will be fair to most of the user requests, regardless of the other parameters. This statement agrees with the state-of-the-art literature [5,8,17,19,23,38], where it is stated that most large-scale MapReduce clusters run small jobs.

As discussed in Table 1, any failure detector algorithm should have in mind that, a normal task which is suspected, needs to be removed from the normal set, and enter into the suspected set of tasks. According to the possible alternatives, we could derive the Algorithm 1.

Algorithm 1: A task evolution from normal to suspected and viceversa.

```
forall the task ∈ ⊓ do
    if (task ∉ normal) ∧ (task ∉ suspected) then
        suspected := suspected ∪ {task};
        trigger(task, SUSPECT);
    else if (task ∈ normal) ∧ (task ∈ suspected) then
        suspected := suspected \ {task};
        trigger(task, RESTORE);
    end
end
```

For the tasks that are in the suspected set, it is necessary to find alternatives for completing them. According to Table 2, a reasonable decision to make is speculating the suspected task, as a form of not jeopardizing the completion time of the overall job request. Algorithm 2 is used in this scenario. This algorithm is composed of a loop for all the suspected tasks. This algorithm also takes into account the resources available in the system. In particular, if a task is not speculated yet and there are resources available in different nodes to the node in which the task has been scheduled, the algorithm triggers this one as speculated.

Algorithm 2: Speculating the suspection.

```
forall the task ∈ suspected do
    if (task ∉ speculated) then
        if (availableResources) ∧ (availableResources ∉ worker_task ) then
            speculated := speculated ∪ {task};
            trigger(task, SPECULATE);
        end
    end
end
```

A relevant question to solve in this scenario is the maximum number of speculations for a single suspected task. This could be taken into account by the Algorithm 2, substituting the sentence **if**($task \notin speculated$) by **if**($speculated_{task} \geq$

max_*number_speculations*). This follows the guidelines provided for instance by [7], which suggests 2 as maximum number of speculations.

In addition, there may be nodes whose performance is causing general overhead on its tasks. In this case, a reasonable reaction would be to consider those nodes as harmful for future executions, and provide a solution to their suspected tasks. The procedure in Algorithm 3 is an example of this scenario. In this case, we are not allocating tasks to a node whose number of suspected tasks is equal or greater than 3.

Algorithm 3: Limiting the suspected task number in the same worker.

```
if (suspected_worker ≥ 3) then
    lost_workers := lost_workers ∪ {suspected_worker};
    trigger(suspected_worker, LOST);
end
```

From these algorithms, we have designed a complete one, called HR-FD, which implements an eventual failure detector algorithm on top of a synchronous system. As the other modules of our failure detector abstraction have similarities with this algorithm, we have designed a basic algorithm structure (Algorithm 4). This basic algorithm is composed of three steps:

Algorithm 4: Basic structure of a failure detector.

```
Implements: Timeout
Uses      : ProgressScore
upon event (Init) do
    normal := ⊓;
    suspected := ∅;
    speculated := ∅;
    starttimer(≪parameters≫);
end
upon event (Timeout) do
    ≪Update parameters≫
    forall the task ∈ ⊓ do
        if (task ∉ normal) ∧ (task ∉ suspected) then
            suspected := suspected ∪ {task};
            trigger(task, SUSPECT);
        else if (task ∈ normal) ∧ (task ∈ suspected) then
            suspected := suspected \ {task};
            trigger(task, RESTORE);
        end
    end
    forall the task ∈ suspected do
        if (task ∉ speculated) then
            if (availableResources) ∧ (availableResources ∉ worker_task ) then
                speculated := speculated ∪ {task};
                trigger(task, SPECULATE);
            end
        end
        if (suspected_worker ≥ 3) then
            trigger(suspected_worker, LOST);
        end
    end
    trigger(HeartbeatRequest, task, SEND);
    starttimer(≪parameters≫);
end
≪Address worker timeout≫
upon event (HeartbeatReply, task, DELIVER) do
    normal := normal ∪ task;
end
```

- Init (Initialization stage):

  - Initialize all sets from beginning
  - Estimate the workload finish time
- Timeout (Timeout adjustment)
- Heartbeat (Acknowledge the reception and change task set, if necessary)

The three main algorithms (HR-FD, MR-FD and LR-FD) follow this structure, although they have some differences, depending on the accuracy and level of flexibility of the failure detector. We will show the differential aspects for the three algorithms.

Algorithm 5 shows the differences of HR-FD with regards to the basic algorithm structure. Through the HR-FD algorithm, we establish time boundaries on omission failures. This is important, since the causes of the stragglers are not only crashes, and the system must react to these issues that harm users and resource providers. In other words, although it is not sure that a crash has happened, the system should decide whether to concurrently speculate the affected task or simply kill the straggler and re-execute it once again from the beginning.

Algorithm 5: The High relax failure detector definitions, from the basic structure.

starttimer(≪parameters≫) := starttimer(InitialEstimatedTime);
≪Update parameters≫ = ∅
≪Address worker timeout≫ = ∅

In the Algorithm 5, the master process maintains a list of normal, suspected and speculated tasks. In addition, it adjusts a timeout according to an estimated completion time increased with some probability margin of error.

Whenever the failure detector triggers a timeout, it will manage those tasks which do not belong to normal and suspected sets. Accordingly, it will place the tasks in the suspected set, and triggers a suspect event for the respective task. If the task belongs to both sets, then this task will be removed from the suspected set. It is very important to remove it, since it will not request other resources in the next step.

All the suspected tasks are eventually speculated, if new resources, which are independent from the worker, are available. In this condition, we have not implemented any speculated number for the suspected task, although this is possible.

The algorithm checks another condition in the same loop. Namely, it limits the number of speculated tasks within the same worker. This means, whenever the worker has a certain number of stragglers, it will be considered lost, in order to let the scheduler know that this node should not accept future tasks until the node recovers from this state.

4.1. Correctness

The *completeness* property is satisfied by the algorithm, because if a task is behaving as a straggler, it will not send a heartbeat to the master for a certain period, which is a condition of the respective master to place this straggler in the suspected set. Regarding the *accuracy* property, a timeout according to the initial estimation is believed to be sufficient for every task to deliver a heartbeat, informing the master about its liveness and progress score.

4.2. Performance

In order to detect a MapReduce straggler through the HR module, the initial timeout adjustment is crucial. As we have stated before, this algorithm should be capable to adjust a job specific timeout, according to the estimated job completion time, and a probable margin of error. This is particularly important for small jobs, whose estimated completion time is under 10 min. For longer lasting jobs, we have decided to leave the default timeout of the Hadoop MapReduce. By this statement, we have placed a higher boundary (that is, 10 min) of the timeout. Knowing that the majority of the production clusters run small jobs [5,8,17,19,23], this timeout is actually addressing the vast majority. We consider that there are also very tiny jobs whose estimated completion time is very small, in terms of seconds. Since a timeout of these margins would result harmful for the infrastructure (resource utilization), due to the possibility of many wrong suspicions, the algorithm should use a minimal timeout in these cases. This minimal timeout should have a lower boundary, that guarantees no eventual processing overhead. For small infrastructures, the administrator can decide this value. For larger infrastructures, the best choice would be to apply an autonomic approach [22,30].

We have run performance simulations in order to compare the Hadoop timeout with the HR-FD timeout. In Table 3, we have taken as a sample a workload with an estimated time completion of 5 min. This is an average value, which help us to see the evolution, since every iteration of the MapReduce tasks lasts approximately 1 min [37]. In the first column, there are different iterations of the same workload. According to the iteration, the second column indicates the finish time of the workload with no failures. The third column represents the failure injection time for each iteration. Right after the Hadoop finish time, there are 4 columns listing the HR-FD finish time, with different λ, which represents the margin of error. As we can notice, this static timeout, which is clearly better than the default timeout in Hadoop MapReduce, performs really well for most of the average production cluster jobs, whose completion time is not very long and neither very small. As is shown in the Table, the smaller margin of error is used, the more accurate the estimations are, since the system is more stable.

Table 3
A performance comparison of Hadoop MapReduce timeout and HR-FD timeout for a 5 min workload. Hadoop: Default Hadoop completion time without failure. Failure time: Failure injection time. FHadoop: Default Hadoop completion time with injected failure. $\lambda = 1.00$. HR-FD with the respective margin of error that is specified, that is, $\lambda = 1.00; 0.75; 0.50; 0.25$.

Iteration	Hadoop	Failure time	FHadoop	$\lambda = 1.00$	$\lambda = 0.75$	$\lambda = 0.50$	$\lambda = 0.25$
1	5	1	10	6.00	5.75	5.50	5.25
2	4	2	11	7.00	6.75	6.50	6.25
3	3	3	12	8.00	7.75	7.50	7.25
4	2	4	13	9.00	8.75	8.50	8.25
5	1	5	14	10.00	9.75	9.50	9.25

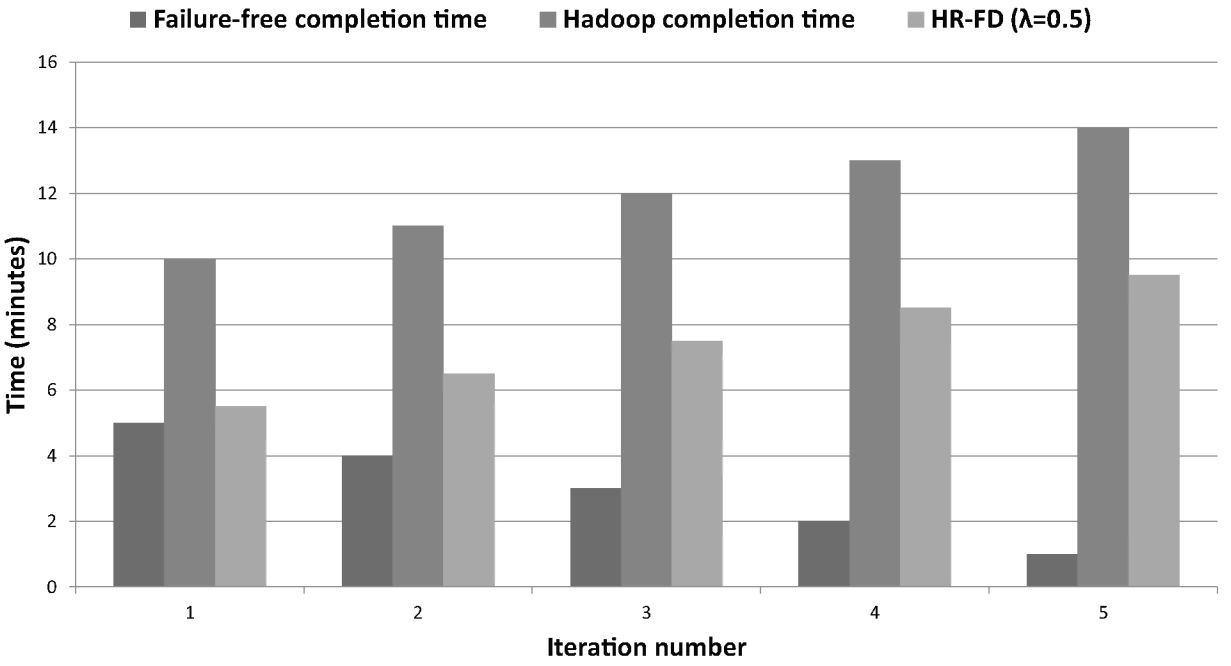


Fig. 3. A performance comparison of Hadoop MapReduce timeout and HR-FD timeout for a 5 min workload.

The best performance of HR-FD corresponds to the first iterations. This is due to the fact that the timeout triggers a threshold very early, and is capable of maintaining a moderate completion time upon failures, comparable to the normal case. For example, in case of $\lambda = 0.75$, if a failure is enforced during the first iteration, from a normal estimated completion time of $t_{e1} = 5$ min, the new completion time would be $t_r = 5.75$ min, instead of the Hadoop completion time of $t_H = 10$ min. In other words, HR-FD timeout exhibits only 15% of performance degradation, whereas Hadoop timeout exhibits 100% of performance degradation. Fig. 3 shows the behavior of HR-FD compared to default Hadoop and a failure-free scenario along the iterations. As the job progress score advances through iterations, the HR-FD timeout benefits decrease when compared to the Hadoop timeout, but despite this, they are still clearly much more favorable.

5. Medium relax failure detector

The previous algorithm (Algorithm 5) deploys a static timeout service. That is, the timeout provided by HR-FD is static, although adjusted at the beginning. Although the algorithm outperforms the default Hadoop mechanism, our aim is to extend the basic HR-FD algorithm by providing a dynamic timeout value.

The ideal choice would be to use an estimated progress score of the overall job, which is then divided in its phases and consequently, in individual tasks. A dynamic timeout can rely on the progress score, especially when is predictable in terms of tasks. This is an already built-in feature in MapReduce-based systems [1,2,21], and other contributions have given even more accurate results in this field [28,29,32].

The main novelty in this module is outlined in the lines of Algorithm 6, where the algorithm calculates the progress score, by adding a margin of error value (λ), that should carry the timeout of the next iteration. This involves that the new progress score is the main parameter of the newly chosen and calculated timeout. Upon each execution of the timeout event, the module sets up a new timeout with a new value.

Algorithm 6: Procedure to calculate the estimated progress score.

```
...
ProgressScore := ProgressScore + λ;
...
startTimer(ProgressScore);
...
```

The Algorithm 7 shows the adaptation of the basic structure (Algorithm 4) to the Medium relax failure detector (MR-FD), implementing an eventual failure detector algorithm on top of a synchronous system. MR-FD enforces stronger timing assumptions than HR-FD, but it still implements an eventual failure detector, presumably in a synchronous system. The main differences of the Algorithm 7 vs the Algorithm 5 is that for every task that gets out from the normal set, it is included in the suspected set of tasks, and in a certain moment, when the cluster provides additional resources, it is executed.

Algorithm 7: The Medium relax failure detector definitions, from the basic structure.

```
starttimer(≪parameters≫) := starttimer(ProgressScore);
≪Update parameters≫ = { ProgressScore := ProgressScore + λ; }
≪Address worker timeout≫ = ∅
```

5.1. Correctness

The failure detector properties are stronger than in the previous algorithm. This means that, both completeness and accuracy have equal or stronger assumptions than HR-FD. Considering this, the completeness property triggers a threshold as long as the minimum progress score is left, in order to change the set of a suspected task. On the other hand, the accuracy property is stricter to guarantee the liveness property of the monitored task.

5.2. Performance

Unlike the Algorithm 5, whose initial adjustment of the timeout is crucial, the Algorithm 7 does not really depend on the initial adjustment, except in the case of those workloads whose estimated completion time is really small, since in this case, the completion time is equal to the minimum possible timeout adjustment. This minimum value is the same as in the Algorithm 5, in order to enforce boundaries for future latency and heartbeat overhead.

The dynamic timeout service is provided at the expense of a higher resource utilization. Whereas the computing resources did not represent any real starvation risk, the Algorithm 7 is using clearly a higher amount of resources, since more speculative executions are needed.

In Table 4, we provide performance simulations, maintaining the same methodology than HR-FD (a workload sample with an estimated completion time of 5 min), by comparing the Hadoop timeout with the MR-FD timeout. Unlike the HR-FD timeout, whose reaction outcome was clearly noted in the first iterations due to its static parameter, MR-FD behaves clearly much better than the default timeout setup of Hadoop MapReduce.

For instance, for $\lambda = 0.75$, if a failure is injected during the first iteration, from a normal estimated completion time of $t_{e1} = 5$ min, the new completion time would be $t_r = 5.75$ min, instead of the Hadoop completion time of $t_H = 10$ min. In other words, MR-FD timeout exhibits only 15% of performance degradation, whereas Hadoop timeout exhibits 100% of performance degradation. And actually, as the job progress score advances through iterations, the MR-FD timeout maintains its performance when compared to the Hadoop timeout, except by the small influence of the accuracy margin values, which make the difference for all the iterations.

Table 4
A performance comparison of Hadoop MapReduce timeout and MR-FD timeout for a 5 min workload. Hadoop: Default Hadoop completion time without failure. Failure time: Failure injection time. FHadoop: Default Hadoop completion time with injected failure. $\lambda = 1.00$. MR-FD with the respective margin of error that is specified, that is, $\lambda = 1.00; 0.75; 0.50; 0.25$.

Iteration	Hadoop	Failure time	FHadoop	$\lambda = 1.00$	$\lambda = 0.75$	$\lambda = 0.50$	$\lambda = 0.25$
1	5	1	10	6	5.75	5.5	5.25
2	4	2	11	5	4.75	4.5	4.25
3	3	3	12	4	3.75	3.5	3.25
4	2	4	13	3	2.75	2.5	2.25
5	1	5	14	2	1.75	1.5	1.25

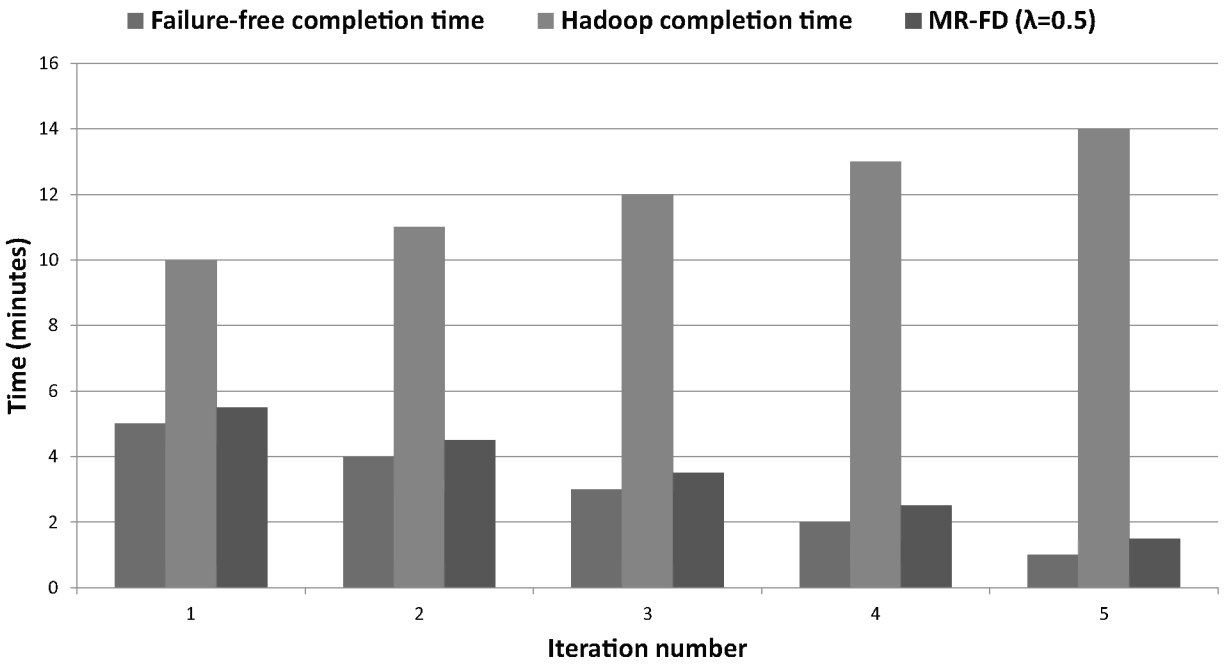


Fig. 4. A performance comparison of Hadoop MapReduce timeout and MR-FD timeout for a 5 min workload.

As in the previous section, Fig. 4 shows the behavior of MR-FD compared to default Hadoop and a failure-free scenario along the iterations. Unlike HR-FD, the MR-FD timeout benefits increase when compared to the Hadoop timeout as the iterations increase.

6. Low relax failure detector

As previously mentioned, the difference between Algorithm 5 and Algorithm 7 is the time reaction to failures. Whereas Algorithm 5 assumes static timeout predictions for suspicious tasks, the Algorithm 7 reacts dynamically to the same suspicions, by providing clear advantage in job completion time. However, the Algorithm 7 does not completely provide strictly bounded timings assumptions. This algorithm does not fit with systems whose results are strictly deadline-bounded, and where it would be possible to afford a bigger amount of resources in order to complete their tasks as fast as possible. These systems could be mission critical systems (such as military or air traffic control systems), or enterprise systems (such as auctioning systems), whose decision making is important and urgent.

In addition to the use of the heartbeat mechanism for providing a timeout service for tasks, it is also possible to monitor the machines (node, worker) metrics [27] and adjust specific thresholds in order to target or enforce the completion time of deadline-bounded workloads (requests).

For deadline-bounded workloads we consider the timeout for the workers as an additional parameter. As long as this timeout notices uncommon behavior after a certain established period, it will request from the application to trigger speculative execution for the ongoing tasks on that particular worker. If there are not tasks, then it will stop deploying future tasks, until the worker comes back to the normal set. For example, in a worker, whose monitoring system monitors parameter p, if this parameter does not appear during a number of times provided by a *threshold* function, the failure detector mechanism will suspect all of its tasks, and declare the worker as lost, as shown in Algorithm 8.

Algorithm 8: Worker parameters monitored with separate timeout.

forall the $worker \in \sqcap$ **do**
  if $(measures(worker, p) \leq threshold(p))$ **then**
    $suspected := suspected \cup \{task_{worker}\}$;
    trigger(task, SUSPECT);
    $lost_{workers} := lost_{workers} \cup \{suspected_{worker}\}$;
    trigger($suspected_{worker}$, LOST);
  end
end

The Algorithm 8 may even expand and maintain a history of the workers. By detecting a repeatable defect in any of them, it may decide to give priority to newer or more stable workers, as long as they respect a certain degree of data locality [36].

This approach is shown in Algorithm 9. This algorithm expands the Algorithm MR-FD, by considering the additional timeout of the workers. As soon as one of the timeouts shows an uncommon behavior, from either tasks or workers, the algorithm triggers speculations in different stable nodes.

Algorithm 9: The Low relax failure detector definitions, from the basic structure.

```
starttimer(≪parameters≫) := starttimer(ProgressScore, threshold(p));
≪Update parameters≫ = { ProgressScore := ProgressScore + λ; }
≪Address worker timeout≫ = {
upon event (Timeout_worker) do
    forall the worker ∈ ⊓ do
        if (measures(worker, p) ≤ threshold(p)) then
            suspected := suspected ∪ {task_worker};
            trigger(task, SUSPECT);
            lost_workers := lost_workers ∪ {suspected_worker};
            trigger(suspected_worker, LOST);
        end
    end
    startTimer(threshold(p));
end
}
```

6.1. Correctness

Let us consider the completeness property first. As long as a task or worker is behaving right, the algorithm does not act. However, if a task or worker does not deliver heartbeat signals for a certain period, these tasks or workers will be deleted from the normal set of tasks/workers, assigning them to the suspected set. The master process will suspect these tasks and workers until the job has finished. Therefore, this algorithm shows some differences in terms of the use of the sets with regards to the previous two failure detectors. Indeed, the change of the sets is for this algorithm more rigorous. If there is a timeout threshold, the suspected tasks will not be terminated. However, if a certain task moves to the suspected set, it will not be able to come back to the normal set again. Therefore, all these tasks of the suspected set have to be speculated in other workers.

Regarding the accuracy property, the master will suspect a task (worker), only if a task (worker) is not able to transmit a message within a specified interval. Otherwise, if any task or worker behave properly, it is assumed that it would be able to send delivery notifications to the master.

6.2. Performance

The above Algorithm 9 does not depend only on a single dynamic timeout. Indeed, it needs the participation of an external monitoring system. Unlike the two previous algorithms (Algorithm 5 and 7), whose initial and dynamic adjustment of the timeout were crucial, the Algorithm 9 goes beyond this. As previously mentioned, the first adjustment is only important for those workloads whose estimated completion time is really small, and the dynamic adjustment is only important for those workloads whose estimated completion time is endangered from common task problems. However, the suspicion probability takes bigger risks when there are issues related to the nodes where tasks are allocated. On the other hand, if the failure detector monitor depends on two stricter timeouts instead of one, the resource utilization factor increases, because the algorithm reacts sooner to suspicions and therefore, there would be more suspicions than in the other scenarios.

In Table 5, we provide performance simulations, maintaining the same methodology as in HR-FD and MR-FD (a workload sample with an estimated time completion of 5 min), by comparing the Hadoop timeout with the LR-FD timeout. Unlike the HR-FD timeout, the Algorithm 9 reacts very well for any production cluster scenario, and performs a slight improvement when comparing to the MR-FD timeout.

For example, in case of $\lambda = 0.15$, if a failure is enforced during the first iteration, from a normal estimated completion time of $t_{e1} = 5$ min, the new completion time would be $t_r = 5.15$ min, instead of the Hadoop completion time of $t_H = 10$ min. In other words, LR-FD timeout exhibits only 3% of performance degradation, whereas Hadoop timeout exhibits 100% of performance degradation.

Table 5
A performance comparison of Hadoop MapReduce timeout and LR-FD timeout for a 5 min workload. Hadoop: Default Hadoop completion time without failure. Failure time: Failure injection time. FHadoop: Default Hadoop completion time with injected failure. $\lambda = 1.00$. LR-FD with the respective margin of error that is specified, that is, $\lambda = 1.00; 0.75; 0.50; 0.25$.

Iteration	Hadoop	Failure time	FHadoop	$\lambda = 1.00$	$\lambda = 0.75$	$\lambda = 0.50$	$\lambda = 0.25$
1	5	1	10	5.25	5.15	5.10	5.05
2	4	2	11	4.25	4.15	4.10	4.05
3	3	3	12	3.25	3.15	3.10	3.05
4	2	4	13	2.25	2.15	2.10	2.05
5	1	5	14	1.25	1.15	1.10	1.05

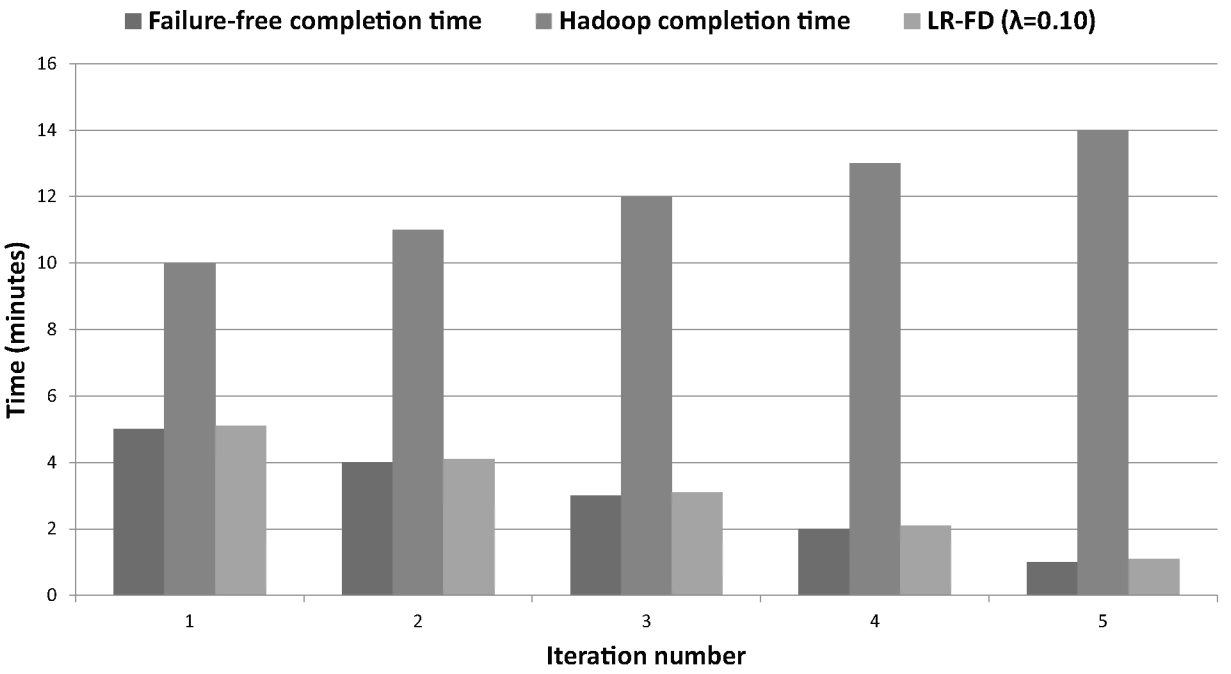


Fig. 5. A performance comparison of Hadoop MapReduce timeout and LR-FD timeout for a 5 min workload.

As in the previous cases, Fig. 5 shows the comparison between default Hadoop timeout and LR-FD timeout, representing also the failure-free completion time. This is clearly the approach with a higher performance, due to both timeouts (tasks and workers) and the lower margin of error exhibited by this algorithm.

Finally, Fig. 6 shows the comparison between all the algorithms, HR-FD, MR-FD and LR-FD and the default Hadoop MapReduce timeout. As we can notice, MR-FD and LR-FD behaves much better than the other alternatives. This is due to the dynamic adjustment of the timeout, which is applied by both approaches. LR-FD behavior is slightly better than MR-FD, demonstrating that the external timeout, that is, the timeout associated to the workers, is not so relevant as the task timeout.

7. Related work

Most of the state of the art in this direction has intended to improve the job execution time, by means of doubling the overall small jobs [4], or just by doubling the suspected tasks (stragglers) through different speculative execution optimizations [7,11,16,21,34,37].

In [37], authors have also proposed a new scheduling algorithm called Longest Approximate Time to End (LATE) to improve the performance of Hadoop in a heterogeneous environment, due to the variation of VM consolidation amongst different physical machines, by preventing the incorrect execution of speculative tasks. In this work, authors try to solve the issue of finding the real stragglers[1] among the MapReduce tasks, in order to speculatively execute them, giving them the deserved priority. As the node heterogeneity is common in the real-world infrastructures and particularly cloud infrastructures, the

[1] It is important to mention that, differently from [37] which considers tasks as stragglers, in the default paper of Google [16], a straggler is "a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation."

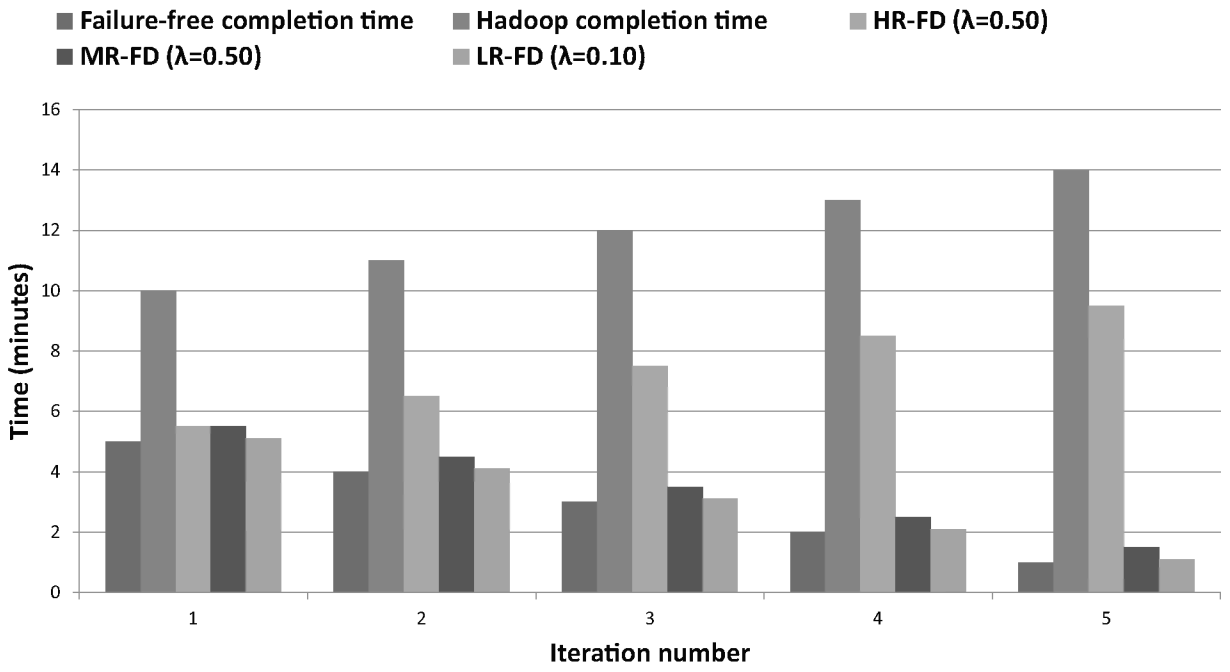


Fig. 6. A performance comparison of Hadoop MapReduce timeout, HR-FD, MR-FD, and LR-FD timeout for a 5 min workload.

speculative execution in the default Hadoop's MapReduce implementation is facing difficulties to give a good performance. The paper proposes an algorithm which should in some way improve the MapReduce performance in heterogeneous environments. It starts giving some assumptions made by Hadoop, and how they are broken down in practice. Later on, it proposes the LATE algorithm, which is based on three principles: prioritizing tasks to speculate, selecting fast nodes to run on, and capping speculative tasks to prevent thrashing. The paper has an extensive experimental evaluation, which proves the valuable idea implemented in LATE.

Mantri [7] is another important contribution related to omission failures, which are called outliers in this paper. The main aim of the contribution is to monitor and cull or relax the outliers, accordingly to their causes. Based on their research, outliers have many causes, but mainly are enforced by MapReduce data skew, crossrack traffic, and bad (or busy) machines. In order to detect these outliers, Mantri does not rely only on task duplication. A real time progress score is able to separate long tasks from real outliers. Whereas the former tasks are allowed to be run, the real outliers are only duplicated when new available resources arise. Since the state-of-the-art contributions are mostly based on duplicating tasks at the end of the job, Mantri is able to make smart decision even before this, in case the progress score of the task is heavily progressing. Apart from data locality, Mantri places task based on the current utilization of network links, in order to minimize the network load and avoid self-interference among loads. In addition, Mantri is also able to measure the importance of the task output, and according to a certain threshold, it decides whether to recompute task or replicate its output. In general, the real-time evaluations and trace-driven simulations show Mantri to improve the average completion time for about 32%.

In [38], authors have proposed two mechanisms to improve the failure detection in Hadoop via heartbeat, but only in the worker side, that is, the TaskTracker. While the adaptive interval mechanism adjusts the TaskTracker timeout according to the estimated job running time in a dynamic way, the reputation-based detector compares the number of fetch-errors reported when copying intermediate data from the mapper and when any of the TaskTrackers reaches a specific threshold that TaskTracker will be announced as a failed one. As authors explain, the adaptive interval is advantageous to small jobs while the reputation-based detector is mainly intended to longer jobs.

GRASS [6] is another novel optimization framework, which is oriented to trimming the stragglers for approximation jobs. Approximation jobs are very common in the last period, because many domains are willing to have partial data in a specific deadline or error margin, instead of processing the entire data in an unlimited time or with 0% error margin. After the introduction of the MapReduce programming model, which came with a simple solution of speculative execution of slow tasks (stragglers), the research community proposed more complex alternatives, such as LATE [37] or Mantri [7]. However, they were not meant to give near to optimal solution for the domain of approximation analytics. And this is the advantage of GRASS, which is basically formed of two algorithms:

1. Greedy Speculative Scheduling (GS). This algorithm is intended to greedily pick a task that will be scheduled next.
2. Resource Aware Speculative Scheduling (RAS). This algorithm is able to measure the cost of letting an old task run or scheduling a new task, according to some important parameters (e.g. time, resources, etc.)

GRASS is a combination of GS and RAS.

Depending on the cluster infrastructure size, but also on other parameters, the scheduler could impose different limitations per user or workload. Among others, it is common to place a limit on the number of concurrent running tasks. The overall set of these simultaneous tasks per each user (or workload) is known as wave. If a GRASS job requires many waves, then it starts with RAS and finally, in the last two waves uses GS. If the jobs are short, it may use only GS. This switching is mostly dependent on:

- Deadline-error bound.
- Cluster utilization.
- Estimation accuracy for two parameters, t_{rem} (remaining time for and old job), and t_{new} (an estimated time for a new job).

Evaluations show that GRASS improves Hadoop and Spark, regardless of the usage of LATE or Mantri, by 47% and 38% respectively, in production workloads of Facebook and Microsoft Bing. Apart from approximation analytics, the speculative execution of GRASS also shows to be better for exact computations.

In [11], authors propose an optimized speculative execution algorithm called Maximum Cost Performance (MCP) that is characterized by:

- Apart from the progress rate, it takes into consideration the process bandwidth in a phase, in order to detect the slow tasks.
- It uses Exponentially Weighted Moving Average (EWMA), whose duty is to predict the process speed and the task remaining time.
- It builds a cost-aware model that determines what task needs a backup based on the cluster load.

In addition, the MCP contribution is based on the disadvantages of previous contributions, which mainly rely on the task progress rate to predict stragglers, inappropriate reaction on input data skews scenarios, unstable cost comparison between the backup and ongoing straggler task, etc. Evaluation experiments on a small-cluster infrastructure show MCP to have 39% faster completion time and 44% improved throughput when compared to default Hadoop.

In [34], authors propose an optimized speculative execution algorithm that is oriented to solving a single-job problem in MapReduce. The advantage of this work is that takes into account two cluster scenarios, heavy and lightly loaded cases. For the lightly loaded cluster, authors introduce two different speculative execution policies, early cloning, and later speculative execution based on the task progress rate. During the stage of heavily loaded cluster, the intuition is to use a later backup task. In this case, an Enhanced Speculative Execution (ESE) algorithm is proposed, which basically extends the work of [7]. Same authors have also introduced an additional extended work that assumes to work for multiple MapReduce jobs [35].

An important project related to Hadoop's omission failures is presented in [12]. In this work, authors have tried to build separate fault tolerance thresholds in the UpRight library for omission and commission failures, because omission failures are likely to be more common than commission failures. As we have mentioned before, during omission failures, a process fails to send or receive messages specified by the protocol. Commission failures exclude omission failures, including the failures upon which a process sends a message not specified by the protocol. Therefore, in the case of omission failures, the library can be fine-tuned in order to provide the liveness property (meaning that the system is "up") in scenarios with these failures.

The TaskTracker omission failures have also been indirectly addresses in other works [13,14].

Unlike all these approaches, our proposal focuses on the timeout service adjustment, taking into account the timing assumption heterogeneity of MapReduce-based systems.

8. Conclusions and future work

This paper provides a formalization of a failure detection abstraction for MapReduce-based systems. As part of this formalization, we have defined three different algorithms. The first abstraction, called High Relax Failure Detector (HR-FD), is an alternative to the default timeout. The second abstraction, called Medium Relax Failure Detector (MR-FD), dynamically modifies the timeout, according to the progress score of each workload. Finally, the third abstraction, called Low Relax Failure Detector (LR-FD), is based on the intersection between the MR-FD timeout service and an external monitoring system timeout service, with the aim of achieving more efficient failure detections.

According to the performance evaluation, we have shown by simulation that these abstractions outperform the default timeout service of Hadoop. Furthermore, we have also demonstrated the correctness of the three algorithms.

As future work, we will instantiate these abstractions for different data-intensive computing systems, in order to enhance the behavior of these frameworks in terms of failure detection and its relation with the default timeout. This is particularly important in the case of production clouds. Indeed, many SLAs are dependent on the timing assumptions of the data-intensive computing systems. Any decision making process made in this scenario should be based on the knowledge of the accuracy boundaries of these systems. This constitutes an important challenge in order to achieve that the data-intensive computing systems constitute a fierce competitor in some fields where relational database systems have been ruling for many decades.

Acknowledgments

The research leading to these results has received funding from the H2020 project reference number 642963 in the call H2020-MSCA-ITN-2014.

References

[1] Apache Hadoop NextGen MapReduce (YARN), (2015). http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[2] The Apache Hadoop Project, (2015). http://hadoop.apache.org/.
[3] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, E. Harris, Scarlett: coping with skewed content popularity in mapreduce clusters, in: Proceedings of the Sixth Conference on Computer Systems, EuroSys '11, ACM, New York, NY, USA, 2011, pp. 287–300, doi:10.1145/1966445.1966472.
[4] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica, Effective straggler mitigation: attack of the clones, in: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, nsdi'13, USENIX Association, Berkeley, CA, USA, 2013, pp. 185–198. http://dl.acm.org/citation.cfm?id=2482626.2482645.
[5] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, I. Stoica, PACMan: coordinated memory caching for parallel jobs, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, USENIX Association, Berkeley, CA, USA, 2012. 20–20. http://dl.acm.org/citation.cfm?id=2228298.2228326.
[6] G. Ananthanarayanan, M.C.-C. Hung, X. Ren, I. Stoica, A. Wierman, M. Yu, GRASS: trimming stragglers in approximation analytics, in: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14, USENIX Association, Berkeley, CA, USA, 2014, pp. 289–302. http://dl.acm.org/citation.cfm?id=2616448.2616475.
[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris, Reining in the outliers in map-reduce clusters using mantri, in: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 1–16. http://dl.acm.org/citation.cfm?id=1924943.1924962.
[8] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, A. Rowstron, Scale-up vs scale-out for Hadoop: time to rethink? in: Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13, ACM, New York, NY, USA, 2013, pp. 20:1–20:13, doi:10.1145/2523616.2523629.
[9] C. Cachin, R. Guerraoui, L. Rodrigues, Introduction to Reliable and Secure Distributed Programming (2. ed.), Springer, 2011.
[10] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (1996) 225–267, doi:10.1145/226643.226647.
[11] Q. Chen, C. Liu, Z. Xiao, Improving mapreduce performance using smart speculative execution strategy, Comput. IEEE Trans. 63 (4) (2014) 954–967.
[12] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, T. Riche, Upright cluster services, in: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09, ACM, New York, NY, USA, 2009, pp. 277–290, doi:10.1145/1629575.1629602.
[13] T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein, K. Elmeleegy, R. Sears, Mapreduce online, in: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, NSDI'10, USENIX Association, Berkeley, CA, USA, 2010. 21–21. http://dl.acm.org/citation.cfm?id=1855711.1855732.
[14] P. Costa, M. Pasin, A. Bessani, M. Correia, Byzantine fault-tolerant mapreduce: Faults are not just crashes, in: Proceedings of the 3rd IEEE Second International Conference on Cloud Computing Technology and Science, CLOUDCOM '11, IEEE Computer Society, Washington, DC, USA, 2010, pp. 17–24, doi:10.1109/CloudCom.2010.25.
[15] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[16] J. Dean, S. Ghemawat, G. Inc, Mapreduce: simplified data processing on large clusters, in: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, OSDI'04, USENIX Association, 2004.
[17] F. Dinu, T.E. Ng, Understanding the effects and implications of compute node related failures in Hadoop, in: HPDC '12: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, ACM, New York, NY, USA, 2012, pp. 187–198.
[18] F. Dinu, T.S.E. Ng, Hadoop's overload tolerant design exacerbates failure detection and recovery, in: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, NetDB'11, ACM, New York, NY, USA, 2011, pp. 1–7.
[19] K. Elmeleegy, Piranha: optimizing short jobs in Hadoop, Proc. VLDB Endow. 6 (11) (2013) 985–996. http://dl.acm.org/citation.cfm?id=2536222.2536225.
[20] F.C. Freiling, R. Guerraoui, P. Kuznetsov, The failure detector abstraction, ACM Comput. Surv. 43 (2011) 9:1–9:40, doi:10.1145/1883612.1883616.
[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: Proceedings of the 2nd ACM SIGOPS/EuroSys 2007, EuroSys '07, ACM, New York, NY, USA, 2007, pp. 59–72, doi:10.1145/1272996.1273005.
[22] J. Kephart, D. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.
[23] S.Y. Ko, I. Hoque, B. Cho, I. Gupta, Making cloud intermediate data fault-tolerant, in: Proceedings of the 1st ACM symposium on Cloud computing, SoCC '10, ACM, New York, NY, USA, 2010, pp. 181–192, doi:10.1145/1807128.1807160.
[24] B. Memishi, S. Ibrahim, M.S. Pérez, G. Antoniu, On the dynamic shifting of the mapreduce timeout, in: R. Kannan, R.U. Rasool, H. Jin, S. Balasundaram (Eds.), Handbook of Research on Managing and Processing Big Data in Cloud Computing, IGI Global, Hershey, Pennsylvania (USA), 2016.
[25] B. Memishi, M.S. Pérez, G. Antoniu, Diarchy: an optimized management approach for mapreduce masters, Procedia Comput. Sci. 51 (2015) 9–18. International Conference On Computational Science, {ICCS} 2015 Computational Science at the Gates of Nature. http://www.sciencedirect.com/science/article/pii/S1877050915009874.
[26] G. Mone, Beyond hadoop, Commun. ACM 56 (1) (2013) 22–24, doi:10.1145/2398356.2398364.
[27] J. Montes, A. Sánchez, B. Memishi, M.S. Pérez, G. Antoniu, GMone: a complete approach to cloud monitoring, Future Gener. Comput. Syst. 29 (8) (2013) 2026–2040, doi:10.1016/j.future.2013.02.011.
[28] K. Morton, M. Balazinska, D. Grossman, ParaTimer: A Progress Indicator for MapReduce DAGs, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 507–518, doi:10.1145/1807167.1807223.
[29] K. Morton, A. Friesen, M. Balazinska, D. Grossman, Estimating the progress of mapreduce pipelines, in: Data Engineering (ICDE), 2010 IEEE 26th International Conference on, 2010, pp. 681–684.
[30] M. Nami, K. Bertels, A survey of autonomic computing systems, in: Autonomic and Autonomous Systems, 2007. ICAS07. Third International Conference on, 2007. 26–26.
[31] J.S. Plank, M. Allen, R. Wolski, The effect of timeout prediction and selection on wide area collective operations, in: Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01), NCA '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 320–329. http://dl.acm.org/citation.cfm?id=580585.883098.
[32] A. Sánchez, J. Montes, M.S. Pérez, T. Cortes, An autonomic framework for enhancing the quality of data grid services, Future Generation Comp. Syst. 28 (7) (2012) 1005–1016, doi:10.1016/j.future.2011.08.016.
[33] T. White, Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated), O'Reilly, 2012.
[34] H. Xu, W.C. Lau, Speculative execution for a single job in a mapreduce-like system, in: Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on, 2014, pp. 586–593.
[35] H. Xu, W.C. Lau, Optimization for speculative execution in a mapreduce-like cluster, in: 2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26 - May 1, 2015, 2015, pp. 1071–1079, doi:10.1109/INFOCOM.2015.7218480.
[36] M. Zaharia, D. Borthakur, J.S. Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling, in: Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, ACM, New York, NY, USA, 2010, pp. 265–278, doi:10.1145/1755913.1755940.

[37] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, I. Stoica, Improving mapreduce performance in heterogeneous environments, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 29–42. http://dl.acm.org/citation.cfm?id=1855741.1855744.
[38] H. Zhu, C. Haopeng, Adaptive failure detection via heartbeat under Hadoop, in: Proceedings of the 2011 IEEE Asia-Pacific Services Computing Conference, ApSCC'11, IEEE, New York, NY, USA, 2011, pp. 231–238.