Adaptive Failure Detection via Heartbeat under Hadoop

Hao Zhu
School of Software
Shanghai Jiao Tong University
Shanghai, P.R. China
ainilife@sjtu.edu.cn

Haopeng Chen
School of Software
Shanghai Jiao Tong University
Shanghai, P.R. China
chen-hp@sjtu.edu.cn

Abstract—**Hadoop has become one popular framework to process massive data sets in a large scale cluster. However, it is observed that the detection of the failed worker is delayed, which may result in a significant increase in the completion time of jobs with different workload. To cope with it, we present two mechanisms: Adaptive interval and Reputation-based Detector that support Hadoop to detect the failed worker in the shortest time. The Adaptive interval is trying to dynamically configure the expiration time which is adaptive to the job size. The Reputation-based Detector is trying to evaluate the reputation of each worker. Once the reputation of a worker is lower than a threshold, then the worker will be considered as a failed worker. In our experiments, we demonstrate that both of these strategies have achieved great improvement in the detection of the failed worker. Specifically, the Adaptive interval has a relatively better performance with small jobs, while the Reputation-based Detector is more suitable for large jobs.**

I. INTRODUCTION

Apache Hadoop [1], an open-source implementation of Google's MapReduce [2], is a widely-accepted technology today for massive data processing on large scale clusters. Hadoop is a composition of MapReduce and the Hadoop distributed file system (HDFS) [19] which implements many features of the Google's DFS [10]. There are four major components in Hadoop: JobTracker, TaskTracker for MapReduce, NameNode and DataNode for HDFS. JobTracker is responsible for scheduling jobs and tasks onto TaskTracker, and NameNode is in charge of maintaining the index of all the massive files. The advantage of Hadoop is that it allows programmers to focus on designing their application data flows but hide messy details of parallelization, fault-tolerance, data distribution and load balancing in a library.

Due to the popularity of Hadoop, many researchers seek to extend the functionality of MapReduce or optimize its performance [8] [9] [12] [18] [20]. One of the most interesting research hotspots is how to enhance the fault tolerance of Hadoop. As the volume of the data stored in the cluster grows dramtically, the cluster will experience more failures. According to Jeff Dean [3] [4], 1-5% of the disk drivers will die and servers will crash at least twice (2-4% failure rate). A typical new cluster will have more than 1000 machine failures in the first year. Therefore, dealing with the fault tolerance problem is a non-trivial task.

To handle with the failures, both software and hardware failures, MapReduce use a re-execution strategy which will take the following actions [2]: if it is on worker failure, then MapReduce framework will (1) detect failure via periodic heartbeats, (2) re-execute completed and in-progress map tasks on that worker, and (3) re-execute in progress reduce tasks. If it is on master failure, state can be checkpointed into HDFS so as for a new master to recover and continue from the last checkpoint. This re-execution mechanism can achieve good fault tolerance. However, it is observed from our experiments that the detection of the failed worker in Hadoop has a certain delay. This delayed detection of the failure will have several bad consequences:

- Execution time for a small job increases significantly. A small job, which will finish in half a minute under no failure, will probably finish beyond 10 minutes if there is a failure occurring at the runtime. This is because Hadoop set 10 minutes by default for the expiry time of each TaskTracker. Therefore, for a small job, the default expiry interval is too large to find out the lost worker quickly, consequently increasing the completion time of small jobs significantly.
- A Healthy node will possibly be added into blacklist by mistake. In Hadoop the blacklist mechanism is used to mark unhealthy nodes in the cluster which has more than four tasks failed on the node. The node in the blacklist will never be assigned with any tasks on it for a period of time. If failure happens in the reduce phase of a job, some tasks on the healthy node may fail because they cannot correctly fetch the map data from the failed worker. As a result, these healthy nodes will be possibly added into the blacklist by mistake, which will further increase the execution time.
- Too many unnecessary backup tasks scheduled. This is because Hadoop will consider those tasks on the failed worker as slow tasks, so that they assign backup tasks to re-execute them.

This paper focuses on how to fast detect the failed worker in a Hadoop cluster. It should be noted that by failure, it refers to the worker failure. In this paper, we propose two mechanisms to achieve the goal of fast failure detection in Hadoop via heartbeat. Firstly, we introduce the Adaptive interval which will dynamically configure the expiry time adapted to the various sizes of jobs. As the experiments show, the Adaptive interval is advantageous to the small jobs. To handle with large jobs, the Reputation-based Detector was developed. In the Reputation-Based Detector, JobTracker is

978-0-7695-4624-7/11 $26.00 © 2011 IEEE
DOI 10.1109/APSCC.2011.46

in charge of evaluating the reputation of each worker according the reports of the failed fetch-errors from each worker. Once the reputation of one worker is lower than a bound, the master will mark this worker as a failed tracker. As the experiments show, these two mechanisms can achieve a better performance in reducing the time to detect the worker failure than the original Hadoop, thus reducing the job completion time.

The roadmap for the rest of the paper is as follows. In Section II, we discuss some related work about the fault tolerance optimizations. In Section III to Section V, we will show the architecture design and explain how the Adaptive interval and Reputation-based Detector work respectively. We further demonstrate how the experiments are carried out and compare the performance of the two proposed methods with the original Hadoop in Section V.

II. Related Work

Recently there are lots of efforts trying to enhance the fault tolerance of Hadoop. Speculative execution is an effective way to guarantee the fault tolerance and response time. Speculative execution is to re-execute slow tasks which have relatively low progress scores. According to Zaharia et al [5], the speculative execution can achieve good performance in a homogeneous environment but not in a heterogeneous environment. It is because the original job progress score estimating algorithm cannot find the real slow tasks in a heterogeneous environment. Therefore, they proposed a new algorithm called Longest Approximate Time to End (LATE), aiming to find the real slow tasks. LATE first estimates the remaining time for each tasks, then assigns the speculative tasks for those tasks with the longest remaining time to end.

Another attempt is trying to improve the availability of Hadoop cluster by avoiding single point of failure (SPOF). Hadoop Distributed File System (HDFS) and Hadoop MapReduce both adopt master-slave architecture, so that SPOF can occur in the NameNode in HDFS and JobTracker in MapReduce. Either of their failure will make the entire cluster un-available. Feng Wang et al. proposed a metadata replication-based solution which involves three major phases [6]: in initialization phase, each standby/slave node is registered to active/primary node and its initial metadata (such as version file and file system image) are caught up with those of active/primary node; in replication phase, the runtime metadata (such as edit logs and lease states) for failover in future are replicated; in failover phase, standby/new elected primary node takes over all communications.

Other studies are trying to protect the intermediate data since intermediate key-value pairs are the most important data transferred between the nodes. Steven Y. Ko et al. develop the Intermediate Storage System (ISS) [7] to keep the intermediate data safe. ISS use an asynchronous rack-level selective replication mechanism, which minimizes the effect of run-time server failures on the availability of intermediate data. Another effort to keep intermediate data safe is from Bicer [15], et al. In their solution, the reduction object after processing of a certain number of elements on each node is copied to another location periodically. Therefore, if one worker fails, its reduction values exist on another node. Their system design can be viewed as a checkpoint-based approach.

Regarding the study of the worker expiry time in Hadoop, MOON [17] introduces a new state for each worker called *hibernate* state and a new parameter into Hadoop called *NodeHibernateInterval* which will work together with *NodeExpiryInterval.* The worker enters into the *hibernate* state if no heartbeats are received for more than the *NodeHibernateInterval*. *NodeHibernateInterval* is much shorter than the *NodeExpiryInterval*. If a worker is in *hibernate* state, it will not service any request to avoid unnecessary access attempts from clients. Compared with it, our study focuses on variable expiry intervals which are adaptive to the job sizes rather than a fixed or static interval.

Overall, our study differs from the work mentioned above in the time to handle with failure. All the above studies except the MOON are attempting to improve recovery mechanisms which are effective when the failure is detected and located. However, our work is much like a pro-active way, trying to detect the lost workers as quickly as possible. If the cluster is not aware of the failure in shortest short time, the above work cannot get the expected performance. To best of our knowledge, it is the first attempt to guarantee the fault tolerance of Hadoop by adaptively detecting the failed workers.

III. Architecture Design

Original failure detection in Hadoop is based on the expiry thread which monitors the heartbeats from TaskTracker and periodically checks whether the time between the current and last heartbeat expires. As is shown in Figure 1, TaskTracker will send its heartbeat every 3 seconds by default and the module called Heartbeat Processor will process these heartbeats and reply a new heartbeat back to TaskTracker. Then the expiry thread will utilize these heartbeats received on JobTracker to periodically check the workers whether they are dead or alive. If the expiry thread does not receive the heartbeats from one worker within a period of time, which is 10 minutes by default, then the worker will be considered as dead.

To cope with the delayed failure detection, we develop two modules for JobTracker which are marked with shadow in the Figure 1. One is Adaptive Interval, which will configure the heartbeat interval and node expiry interval time based on that estimating time. It needs Job Estimator to estimate the job execution time for Adaptive Interval. Another mechanism is called Reputation-based Detector, which is responsible for collecting the heartbeats from each TaskTracker and extracting fetch-error information from these heartbeats. The Reputation-based Detector will use these fetch-errors to evaluate the reputation for each TaskTracker. Basically, these two mechanisms proposed in this paper are based on the heartbeat because they depend on the heartbeat to either check expiry or deliver messages. Specifically, as the figure 1 shows, the new processing flow is as following:

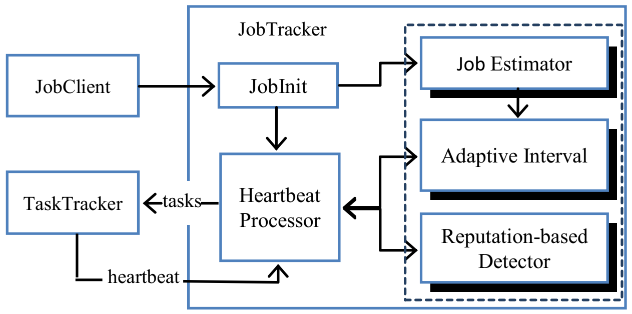

Figure 1. Architecture Design of the Adaptive Failure Detction

(1) JobClient first receives a request from users, and then submits this request to JobTracker.

(2) JobInit prepares the necessary data for the job, including the data locations, executing jars and etc.

(3) After initializing the job, general information about the job flows into the Job Estimator, which will analyze this information to estimate the execution time of the job.

(4) Then, the Adaptive interval will use the estimated time to calculate the new adaptive expiry time for each job and configure it into expiry thread and the heartbeat interval at the runtime. The detection method for the Adaptive interval is exactly same as the original Hadoop but differs in its expiry time interval.

(5) At the same time, the Reputation-based Detector is collecting the fetch-errors extracted from the heartbeat and evaluating the reputation for each TaskTracker. If the reputation of one TaskTracker is lower than the lowest bound, JobTracker will mark that TaskTracker as a failed TaskTracker.

In Sections IV and V, the details about how these two mechanisms work will be explained in details.

IV. Adaptive Interval

The design goal of the adaptive interval is to enable Hadoop to dynamically configure its expiry interval which is adaptive to the job size, thus detecting the failed worker in the shortest time. To achieve that goal, the relationship between the heartbeat interval and the job execution time will be discuss firstly. And then a job estimating model will be built to estimate the execution time. Finally, we will introduce the Adaptive Interval which will use the estimating model to calculate the expiry interval and achieve the design goal.

A. Adaptive Heartbeat Interval

The adaptive heartbeat interval means that the heartbeat interval is adaptive to the job size. Why do Hadoop need adaptive heartbeat interval? First, having a longer heartbeat interval means there is a time delay in the time when TaskTracker reports its available slots to JobTracker. In contrast, a low heartbeat interval value means that TaskTracker can report its status about its available slots and failure information to JobTracker without any delay, but the relatively frequent interval will result in the overloading of the master [16]. It is confirmed in the experiments as showed in Figure 2. Overall, the job execution time with the default 3 seconds interval has the largest execution time. As the interval time decreases, the execution time has decreased. In contrast, to finish a job, the average number of heartbeats received by JobTracker increases significantly as the heartbeat interval becomes shorter. Therefore, it is important to give a right value to the heartbeat interval to balance the overload of JobTracker and the execution time.

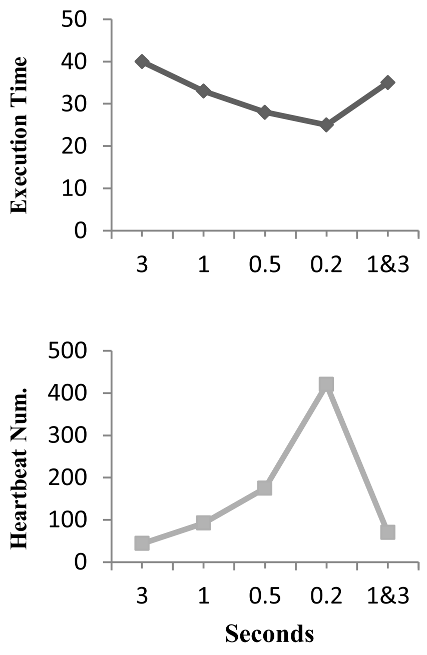

Figure 2. Job Execution Time and Heartbeat Num over various Heartbeat Intervals for the sort program

We observe that the reduce tasks spend more time than the map tasks. Therefore, it is unnecessary to use a uniform interval for both map and reduce tasks. We develop a *two-phase interval*: map-phase interval and reduce-phase interval, to adaptively set the heartbeat interval. When the job runs in the map phase, JobTracker and TaskTracker adopts the map-phase interval. When the job runs in the reduce phase, the reduce-phase interval is adopted, which is relatively longer than the map-phase interval. In the Figure 2, the last point in both line graphs shows the results of the *two-phase interval*. In this specific experiment, the map phase interval is 1 second and the reduce phase interval is 3 seconds. The execution time for the job is 12.5 percent less than the default 3 seconds. Compared with other heartbeat intervals except the default 3s, the two-phase heartbeat interval has much smaller heartbeat number which is manifest to ease the burden of the master. It should be noted that the two intervals are adaptive to the map and reduce task execution time respectively. The execution time for both map and reduce task is estimated in the following sector Job Estimator.

B. Job Estimator

There are already several efforts to estimate Hadoop job execution time [13] [14], which are quite similar with ours.

The estimating model is based on the following assumptions: (1) the cluster consists of homogeneous nodes, so that the time of processing a unit of data is equal; (2) distribution of the input data is uniform, so that the both map tasks and reduce tasks have the same amount of data to process. Several notations are specified as described as below:

- α: Total input data for map tasks.
- t_m and t_r: Time to process unit map and reduce data respectively.
- n_m and n_r: Average number of running slots for map and reduce tasks respectively.
- t_s: Time to transfer unit data in the shuffle phase.
- ρ: The ratio of the amount of map input data and map output data, which depends on the characteristic of the different jobs. Different workload with different application logic may have different ρ.
- β: Total map output data where $\beta = \rho * \alpha$.

Hadoop jobs have three major phases: map, shuffle and reduce. In map phase, each task keeps all the intermediate results into the local file system. Before reduce phase, reduce tasks have to prepare the input data, which is often called shuffle phase. In shuffle phase, reduce tasks need to fetch data from the remote workers. We assume that the amount of the data to shuffle is the same amount as the map output data. But in practice, the data to shuffle is relatively smaller than the amount of the map output data. After shuffling, reduce tasks can perform reduce task and keep all of its results into HDFS. Therefore, the estimating execution time (EET) should consist of three components: time to process map data, time to shuffle the data and time to process data in the reduce phase, which is demonstrated in (1).

$$EET = \alpha * \frac{t_m}{n_m} + \beta * t_s + \beta * \frac{t_r}{n_r} \tag{1}$$

C. *Adaptive Expiry Interval*

The expiry interval is closely related to the heartbeat interval. To obtain the same level of the fault tolerance for the various jobs, the expiry interval should be adaptive to the job size.

Hadoop has a TaskTracker-level tuning parameter: *mapred.tasktracker.expiry.interval*, the default value of which in Hadoop is 10 minutes, which is fixed and disadvantageous to small jobs. Our proposal use (1) to estimate the execution time for each job at the runtime and configure that parameter with the new expiry interval time. For large jobs whose execution time is beyond an upper bound value, the upper bound value will be used instead to forbid those large jobs to use its execution time as expiry interval. This upper bound value can be set based on the cluster size and workload characteristic. In the experiments, this upper bound value is 10 minutes. To conclude, the final expiry interval algorithm is defined in (2), where TET denotes the *TaskTrackerExpiryTime*.

$$\text{TET} = \begin{cases} EET & if \text{ EET} < 10 \\ 10 & if \text{ EET} \geq 10 \end{cases} \tag{2}$$

After choosing the expiry interval time, the expiry thread in Heartbeat Processor will use this value to monitor the entire cluster. If JobTracker has not received a heartbeat from one worker within that interval, JobTracker will consider that worker as a failed worker.

V. Reputation-Based Detector

The Reputation-Based Detector aims to evaluate the reputation of each worker based on the fetch-errors information monitored by JobTracker, thus reducing the time to detect the failed worker.

To evaluate the reputation, we first need to explain how TaskTracker can lose its reputation. It is observed that the fetch-errors is one of the most common exceptions caught by TaskTracker when its reduce task cannot finish its copy from the remote worker, primary because the remote worker has failed and its intermediate data have lost. In the proposed solution, the worker who has caught these fetch-errors will report them to JobTracker via heartbeat because it suspects that the remote worker is probably failed. By referring to the suspected information, we call them gossips later on because some of the suspect information is true while others may be not. After receiving these gossips across the workers, JobTracker will subtract corresponding penalty for the reputation to the suspicious worker. When the reputation is lower than a threshold, the suspicious worker will be marked as a failed worker. Before explaining how the reputation-based detector functions, we specify several important objects used in this mechanism as described as below:

- *gossip*< A_{from} , B_{to} , *Time, Penalty*>: Each Time JobTracker receives heartbeat from TaskTracker, it will extract the gossips from heartbeat. Each gossip means the worker A_{from} meets a fetch-error from the worker B_{to}, thus suspecting that the worker B_{to} has a failure at the *Time*. The field *Penalty* means how much reputation should be subtracted from the current reputation of the worker B_{to} , which is calculated based on the past gossips.
- *gossipQueues<TrackerId,Queue<Gossip>>*: JobTracker maintains a gossip queue for each worker. It is where all the gossips are kept on JobTracker. Each time JobTracker receives a gossip in the heartbeat, it will put the gossip into the corresponding queue. Each gossip queue has an expiry time. If the time, between the newly gossip and the latest gossip in the same queue, is longer than the expiry time, all the expired gossips will be removed from the queue.
- *taskTrackerToReputation<TrackerId, Value>*: It is a map which contains all the reputation value to each TaskTracker. If the tracker id is not found in this map, a new object will be created into it with an initial reputation. If the reputation of one tracker is lower than a threshold value, the corresponding tracker object will be deleted from the map.

It is interesting to note that the sematic information contained in these gossips is extremely different. We explore the temporal and spatial characteristics among these gossips.

1) Temporal: The gossips tend to be gathered within a short period of time. The more gossips received within that time, the more penalties will be given to that TaskTracker. For example, three gossips collected at t_1, t_2 and t_3 in Figure 3 (a) are much more believable than the situation where there is only one gossip reporting during the same period. If the worker A_{from} suspects another misbehavior of the worker B_{to}, it needs to re-caculate the reputation value with what have already been stored in its gossip queue regarding the worker B_{to}. For each gossip received, a penalty value will be given to the current reputation of the suspicious worker. The more gossips received, the higher penalty value will be given.

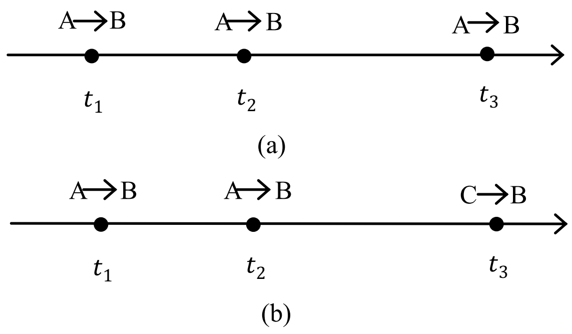

Figure 3. Example of the gossip queue for the worker B

We introduce a tuning parameter called *incremental penalty ratio* to give a penalty to each gossip. Therefore, the function is defined in (3) to calculate the gossip penalty in the Figure 2 where *incremental penalty ratio* is denoted as ρ. $P_t(A,B)$ and $P_{t-1}(A,B)$ denote the penalty calculated for the current gossip and previous gossip from A to B, respectively.

$$P_t(A,B) = \rho * P_{t-1}(A,B) \qquad (3)$$

2) Spatial: The more different workers involved to report, the more JobTracker is convinced that the particular TaskTracker has a failure. For example, even though the queues in the Figure 3 (a) and (b) have exactly the same temporal characteristic, the queue in the figure 2 will be more believable. It is because that the Figure 3 (b) has more workers involved to report the worker B. Therefore, higher penalty weight will be assigned to the gossips from the worker C.

Let $\Phi(\mathrm{B}) = \{\alpha_1, \alpha_2, \alpha_3 \cdots\cdots \alpha_m\}$ be the set of the different workers reporting the suspicion of B. If there is a gossip from a new worker, the following function will be used to calculate the new penalty value,

$$P_t(\alpha_i, B) = \lambda * P_1(\alpha_i, B) \quad \text{if } \alpha_i \notin \Phi(\mathrm{B}) \qquad (4)$$

where λ and $P_1(\alpha_i, B)$ represent the size of $\Phi(\mathrm{B})$ and the initial penalty repectively. After calculating the penalty, this worker will be added into the set $\Phi(\mathrm{B})$. This set will be updated periodically because some gossips will be out of date in the queue, and some workers will be removed out of the set. Combined with (3), we define the final function:

$$P_t(\alpha_i, B) = \begin{cases} \rho * P_{t-1}(\alpha_i, B) & \text{if } \alpha_i \in \Phi(\mathrm{B}) \\ \lambda * P_1(\alpha_i, B) & \text{if } \alpha_i \notin \Phi(\mathrm{B}) \end{cases} \qquad (5)$$

When the reputation of one worker is lower than a threshold, JobTracker will believe that worker failed and is lost. This threshold value can be set based on the size of the cluster and the historical data.

However, how can a worker gain its reputation? A worker can gain its reputation via heartbeat as well. Each time when JobTracker receives a heartbeat, the worker sending this heartbeat will gain an increase on its reputation. Even if it is possible to report gossips by mistake, the worker still can gain its reputation by sending its heartbeats to JobTracker. The reputation-based detector has a upper-bound for the maximum repuation, which means that if the reputation of one TaskTracker is equal to the upper-bound, it cannot gain reputation any more. Figure 4 shows our pseudo code of the reputation-based detector implemented in the prototype system.

Algorithm Reputation-based Detector algorithm

```
1 procedure RBD
2     input: hb
      output: failure signal
      variable definition:
          1.  hb: Represent the received heartbeat
          2.  repFrom: Represent the reputation of the worker
              who has transmitted the heartbeat;
          3.  repTo: Represent the reputation of the
              suspicious worker.
          4.  gp: short for the gossip.

3   repFrom = taskTrackerToReputation.get(hb.from)
4   if repFrom < Maximum Reputation
5      repFrom = repFrom + 1   //gain reputation
6   endif
7   if finding gossips in this heartbeat
8      foreach gp in gossips
9         gossipQueue = gossipQueues.get(gp.to)
10        Remove the expiry gossips in gossipQueue
11        penalty = Use (5) to calculate the new penalty
12        gp.penalty = penalty
13        repTo= taskTrackerToReputation.get(gp.to)-penalty
14        taskTrackerToReputation.put(gp.to, repTo)
15        gossipQueue.add(gp)
16        if (repTo < Minimum Reputation)
17            LostTracker(gp.to)
18     end foreach
19   endif
20 end procedure
```

Figure 4. Pseudo code of the Reputation-based Detector

V. EXPERIMENTS

A. *Configuration*

In this sector, we will explain how the experiments were carried out to validate the proposed mechanisms. We deploy a local cluster which contains 6 Dell Inspiron^{TM} 580s-468 nodes connected by gigabit Ethernet. Each node has Intel Core i3 550, 4GB memory. The Hadoop cluster is build on Hadoop 0.20.2, two nodes of which service as JobTracker and NameNode respectively. All of the six nodes install both TaskTracker and DataNode services.

We use GridMix2 as our default workload. The main purpose of GridMix2 [11] is to model the production loads of the real world and provide it to the developers as a performance beachmark. It supports to generate the massive data for the jobs and to automatically run a mix of synthetic workload, which can be specified in a profile file. There are three versions of the GridMix tool, from which GridMix2 is prefered as the experimental workload, not only because it can simulate the real enviroment, but also because it configure various jobs with variable size. It is convient for us to custeromize the workload to validate our prototype.

The experiments utilise two representative MapReduce applications, i.e., sort and word count, which can be configured in the GridMix2 workload. The configuratioins of the two programs are given in Table I. For both programs, the input data is randomly generated using the tools given by the GridMix2. From the Table I, it can be seen that the sort program is a small job while the word count is relatively large. We choose the jobs with different size because the two mechanisms have their own size preferences, which will be explained in the Results section.

TABLE I. APPLICATION CONFIGURATIONS

Programs	*Input Size*	*#Maps*	*#Reduces*
sort	45M	10	15
word count	545M	60	170

B. *Results*

We use job execution time as the major performance metric. As Figure 5 shows, the native Hadoop can finish the sort and word count programs within 30 seconds and 2.22 minutes respectively, if there are no failures. We inject the worker failure by simply shutdown the TaskTracker service of the worker manually.

While under one worker failure, it is manifest that the performance of the native Hadoop siginificantly decreases since the execution time for the both programs dramtically rises, which are 13 and 14 minutes respectively. The reason for this is that the native Hadoop uses the fixed expiry interval 10 minutes for each worker, which results in the delayed detection of the failed worker.

We first evaluate the performance of the adaptive interval. After estimating the job execution time, the adaptive expiry interval for the sort and word count programs are decided, which are 27 seconds and 2.2 minutes respectively. From Figure 5, the finish time for both programs is relatively longer than the time without any node failures, but is significantly shorter than the time when there are failures of the native Hadoop. It shows that it has 90% and 80% improvement in execution time for sort and word count respectively.

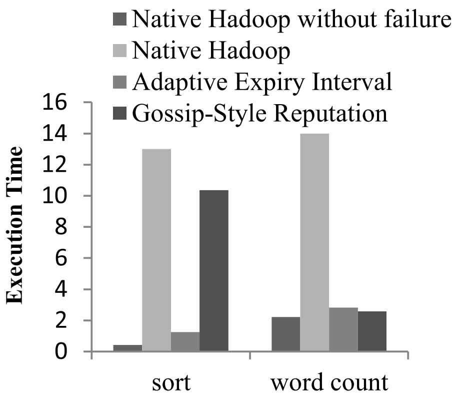

Figure 5. Average Execution Time of different detection mechnisam with one node failure

Furthermore, we evaluate the reputation-based detector mechanism. In the experiments, the configuration of some inportant parameters is as described as below:

- REPUTATIOIN_INITIAL: Initial reputation value for each worker, which is 0 for the cluster.
- REPUTATIOIN_MAX: The maximum reputation value allowed by JobTracker, which are 30 in the experiment.
- REPUTATION_MIN: The minimum reputation value allowed by JobTracker, below which JobTracker will believe that worker as a lost tracker, which is -10.
- INCREAMENTAL_PENALTY_RATIO: Represent the ρ in the reputation-based detector algorithm, which is 2 by default.
- PENALTY_INITIAL: Initial penalty value for each worker, which is 2 by default.

All these parameters can be configured in the configure file: *mapred.xml*. According to the Figure 5, the result for the program sort is slightly shorter than the native Hadoop, but not as good as the result for the program word count, which has achieved 82% execution time decreasing than the native Hadoop, approximately 2.57 minutes.

Why does the performance of the reputation-based detector differ greatly on different jobs? As is shown in the Figure 6 sort (a), when the reputation-based detector was used, the execution time for the sort program is not always fluctuating between 10 and 13 minutes. There are some separated points lower than 3 minutes. While in Figure 6 sort (b), the results are relatively stable and stay off the level of 1.3 minutes. Therefore, the adaptive interval functions stable and better than the reputation-based detector on the

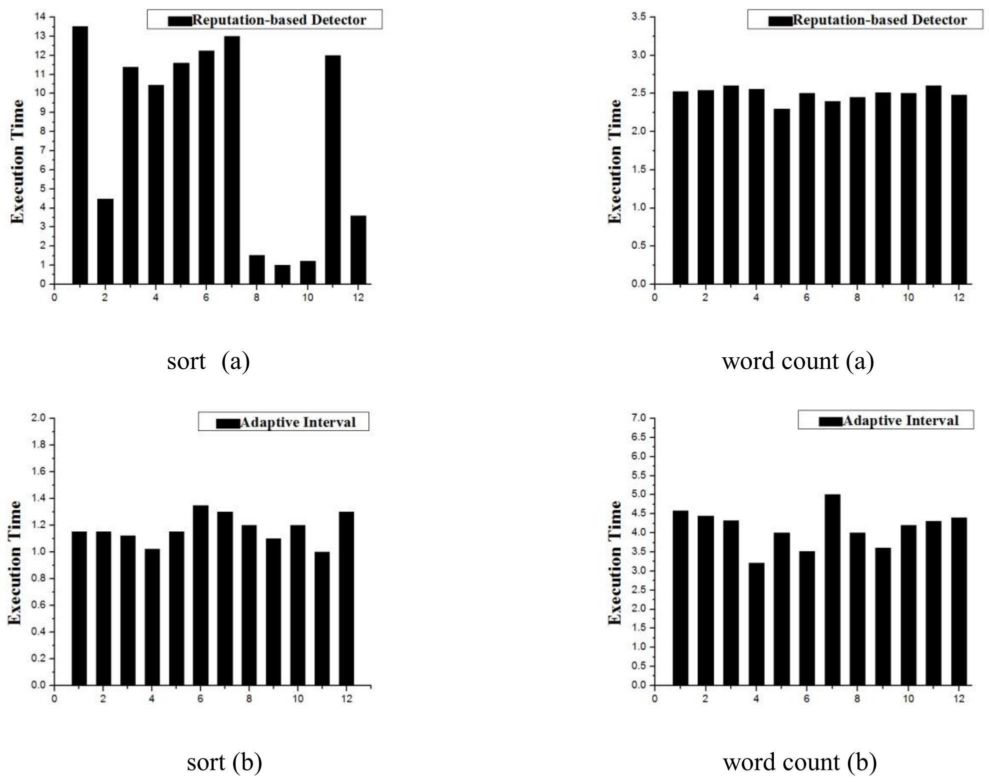

Figure 6. Comparison of the Adaptive Interval and the Reputatioin-based Detector.

sort program. The reason for the unstable of the reputation-based detector is that the sort program is a relatively small job, which will result in that there are probably no tasks running on the ready-to-fail worker, thus decreasing the number of the failed-fetch reports. If there are not enough failed-fetch reports to support JobTracker to decay the reputation for the failed TaskTracker, the Reputation-based detector is exactly the same as the native Hadoop. It can be seen from Figure 6 sort (a) that the average execution time for those points fluctuating around 12 minutes is the same as the native Hadoop.

As the size of the job increases, the performance of the reputation-based detector becomes stable, as is shown in Figure 6 word count (a). Compared with the Figure 6 word count (b), the result shows that the average execution time of the reputation-based detector is better than the adaptive interval. To finish the word count with one failure, the adaptive interval spends at around 3.5 minutes while the reputation-based detector spends only at around 2.5 minutes. The reason for this is that the expiry time for the adaptive interval increases as the size of the job increases, which results in the decrease of the performance. The adaptive expriy interval has a maximum value, which means if the value is larger than the default value, the default 10 minutes will be used. In this case, the adaptive interval mechanism is no much different with the native Hadoop.

Finally, we can conclude that the two mechanisms have achieved the goal of fast detecting the failure workers, thus reducing the execution time significantly. The major difference between the two mechanisms is that the adaptive interval prefers to the small jobs and the reputation-based detector is advantageous to the large jobs. Therefore, they can work even better if they can work together.

VI. Conclusion and Future Work

In this paper, we present the adaptive interval and the reputation-based detector that support Hadoop to detect the lost trackers in the shortest time, thus reducing the job execution time eventually. In particular, we demonstrate the benefit of the two mechanisms to greatly improve the response time. We have learned several things from this work. First, heartbeat interval is important to the job execution time just as the experimental results show. Second, the reputation-based detector is not suitable for the small jobs because there are probably no or less tasks scheduling on the failure node, which results in fewer fetch-errors to report to JobTracker.

Due to the limitations of the reputation-based detector, the reputation-based are not suitable for the small jobs. Therefore, utilizing more kinds of exceptions besides the fetch-errors exceptions is a future improvement.

References

[1] Hadoop. http://hadoop.apache.org/core/

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", presented at Commun. ACM, 2008, pp.107-113.

[3] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool", presented at Commun. ACM, 2010, pp.72-77.

[4] J. Dean, “Design Lessons and Advice from Building Large Scale Distributed Systems”, keynote talk at LADIS 2009.

[5] M. Zaharia, A. Konwinski, A.D. Joseph, R.H. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments", in Proc. OSDI, 2008, pp.29-42.

[6] F. Wang, J. Qiu, J. Yang, B. Dong, X.H. Li, and Y. Li, "Hadoop high availability through metadata replication", in Proc. CloudDB, 2009, pp.37-44.

[7] S.Y. Ko, I. Hoque, B. Cho, and I. Gupta, "Making cloud intermediate data fault-tolerant", in Proc. SoCC, 2010, pp.181-192.

[8] M. Zaharia, D. Borthakur, J.S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling", in Proc. EuroSys, 2010, pp.265-278.

[9] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. xChakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - A Warehousing Solution Over a Map-Reduce Framework", presented at PVLDB, 2009, pp.1626-1629.

[10] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system", in Proc. SOSP, 2003, pp.29-43.

[11] GridMix. http://hadoop.apache.org/mapreduce/docs/r0.21.0/gridmix.pdf

[12] C. Chambers, A. Raniwala, F. Perry, S. Adams, R.R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: easy, efficient data-parallel pipelines", in Proc. PLDI, 2010, pp.363-375.

[13] C. Tian, H. Zhou, Y. He, and L. Zha, "A Dynamic MapReduce Scheduler for Heterogeneous Workloads", in Proc. GCC, 2009, pp.218-224.

[14] Kc K., Anyanwu K., "Scheduling Hadoop Jobs to Meet Deadlines", Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on , vol., no., pp.388-392, Nov. 30 2010-Dec. 3 2010 doi: 10.1109/CloudCom.2010.97.

[15] T. Bicer, Wei Jiang, G. Agrawal, "Supporting fault tolerance in a data-intensive computing middleware", Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on , vol., no., pp.1-12, 19-23 April 2010 doi: 10.1109/IPDPS.2010.5470462.

[16] Linh T.X. Phan, Zhuoyao Zhang, Boon Thau Loo, Insup Lee, "Real-time MapReduce Scheduling", Technical Report No. MS-CIS-10-32, University of Pennsylvania, 2010.

[17] H. Lin, X. Ma, J.S. Archuleta, W. Feng, M.K. Gardner, and Z. Zhang, "MOON: MapReduce On Opportunistic eNvironments", in Proc. HPDC, 2010, pp.95-106.

[18] A. Abouzeid, K. Bajda-Pawlikowski, D.J. Abadi, A. Rasin, and A. Silberschatz, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads", presented at PVLDB, 2009, pp.922-933.

[19] HDFS. http://hadoop.apache.org/common/docs/r0.20.0/hdfs_design.pdf

[20] I. Michael, P. Vijayan, C. Jon, W. Udi, T. Kunal, G. Andrew, "Quincy: Fair scheduling for distributed computing clusters", in Proc. SOSP, 2009, p 261-276.