Analysis of Hadoop's Performance under Failures

Florin Dinu T. S. Eugene Ng
Rice University

***Abstract*—Failures are common in today's data center environment and can significantly impact the performance of important jobs running on top of large scale computing frameworks. In this paper we analyze Hadoop's behavior under compute node and process failures. Surprisingly, we find that even a single failure can have a large detrimental effect on job running times. We uncover several important design decisions underlying this distressing behavior: the inefficiency of Hadoop's statistical speculative execution algorithm, the lack of sharing failure information and the overloading of TCP failure semantics. We hope that our study will add new dimensions to the pursuit of robust large scale computing framework designs.**

I. Introduction

The performance of large scale computing frameworks in the cloud directly impacts many important applications ranging from web-indexing, image and document processing to high-performance scientific computing. In this paper we focus on Hadoop [2], a widely used implementation of the popular MapReduce framework. Given the scale of cloud environments, failures are common occurrences. Studies show [6], [13], [1] tens of failures per day and multiple failures per average compute job. Despite the widespread prevalence of failures, little research work has been done on measuring and understanding Hadoop's performance under failures.

In this paper we analyze Hadoop's behavior under fail-stop failures of entire compute nodes and under fail-stop failures of the TaskTracker or DataNode Hadoop processes running on compute nodes. DataNode failures are important because they affect the availability of job input and output data and also delay read and write data operations which are central to Hadoop's performance. TaskTracker failures are equally important because they affect running tasks as well as the availability of intermediate data (i.e. map outputs). While we use Hadoop for our experiments we believe our findings are generally applicable and can guide the design and development of future large scale computing frameworks.

Our measurements point to a real need for improvement. Surprisingly, we discover that a single failure can lead to large and unpredictable variations in job completion time. For example, the running time of a job that takes 220s without failures can vary from 220s to as much as 1000s under TaskTracker failures and 700s under DataNode failures. Interestingly, in our experiments the failure detection and recovery time is significant and is oftentimes the predominant cause for both the large job running times and their variation. We identify several important underlying reasons for this sluggish behavior.

First, Hadoop's speculative execution (SE) algorithm can be negatively influenced by the presence of fast advancing tasks. DataNode failures are one cause of such fast tasks. These fast tasks can indirectly impact the moment when other tasks are speculated. We find that one fast task can severely delay or even preclude subsequent SEs. The reason lies with the statistical nature of Hadoop's SE algorithm. The algorithm deems a task slow by comparing its individual progress metric against aggregate progress metrics of similar tasks. Fast advancing tasks can skew these aggregate statistical measures.

Second, Hadoop tasks do not share failure information. For scalability and simplicity, each compute task performs failure detection and recovery on its own. The unfortunate effect is that multiple tasks could be left wasting time discovering a failure that has already been identified by another task. Moreover, a speculated task may have to re-discover the same failure that hindered the progress of the original task in the first place. Recent work on the efficiency of SE algorithms [4] pointed out the importance and benefits of making cause-aware decisions regarding outlier tasks. However, cause-aware SE decisions alone do not ensure good performance during failures. To ensure that a speculated task helps improve job running time, failure information needs to be effectively shared between tasks during the runtime of a job.

Third, Hadoop treats in a unified manner different adverse environmental conditions which on the surface have a similar effect on the network connections between Hadoop processes. Specifically, temporary overload conditions such as network congestion or excessive end-host load can lead to TCP connection failures. Permanent failures have the same effect. All these conditions are common in data centers [5], [6]. However, treating these different conditions in a unified manner conceals an important trade-off. Correct reaction to temporary overload conditions requires a conservative approach which is inefficient when dealing with permanent failures. We show that the efficiency of Hadoop's mechanisms varies widely with the timing of the failure and the number of tasks affected. We also identify an important side effect of coupling the handling of failures with that of temporary adverse conditions: a failure on a node can induce task failures in other healthy nodes.

In the existing literature, smart replication of intermediate data (e.g. map outputs) has been proposed to improve performance under TaskTracker failures [10], [4]. Replication minimizes the need for re-computation of intermediate data and allows for fast failover if one replica cannot be contacted as a result of a failure. Unfortunately, replication may not be always beneficial. It has been shown [10] that replicating intermediate data guards against certain failures at the cost of overhead during periods without failures. Moreover, replica-

tion can aggravate the severity of existing hot-spots. Therefore, complementing replication with an understanding of failure detection and recovery is equally important. Existing work on leveraging opportunistic environments for large distributed computations [11] can also benefit from this understanding as such environments exhibit behavior that is similar to failures.

The rest of the paper is organized as follows. In §II we review relevant Hadoop material. In §III and §IV we discuss design decision related to TT and DN failures and their implications on performance. §V presents detailed experimental result. We discuss lessons learned in §VI and conclude in §VII.

II. Background and Notation

A. Notation

In this paper, SE refers to *speculative execution*, *speculatively executed* or *speculatively execute*. The distinction should be clear from the context. We distinguish between the *initial* instance of a task and subsequent SE instances of the same task. We use WTO, RTO and CTO to signify write, read and connect timeouts.

B. Hadoop Components

A Hadoop [2] job has two types of tasks: mappers and reducers. A TaskTracker (TT) is a Hadoop process running on compute nodes which is responsible for starting and managing tasks locally. TTs are configured with a number of mapper and reducer slots, the same number for every TT. If a TT has two reduce slots this means that a maximum of two reducers can concurrently run on it. If a job requires more reducers (or mapper) than the number of reducer (mapper) slots in the system then the reducers (mapper) are said to run in multiple waves. A TT communicates regularly with a Job Tracker (JT), a centralized Hadoop component that decides when and where to start tasks.

Mappers read the job input data from a distributed file system (HDFS) and produce as their output key-value pairs. These map outputs are stored locally on compute nodes, they are not written to HDFS. Each reducer processes a particular key range. For this, it copies map outputs from the mappers which produced values within that key range (oftentimes all mappers). A reducer writes the job output data to HDFS.

Each task has a progress score which attempts to capture how close the task is to completion. The score is 0 at the task's start and 1 at completion. For a reducer, a score of 0.33 signifies the end of the copy (shuffle) phase. At 0.33 all map outputs have been read. A score of 0.66 signifies the end of the sort phase. Between 0.66 and 1 a reduce function is applied and the output data is written to HDFS. The progress rate of a task is the ratio of the progress score over the current task running time. For example, it can take a task 15s to reach a score of 0.45, for a progress rate of 0.03/s.

HDFS is composed of a centralized NameNode (NN) and of DataNode (DN) process running on compute nodes. DNs handle the read and write operations to the HDFS. The NN decides to what nodes data should be written to. HDFS write operations are pipelined. In a pipelined write an HDFS block is replicated at the same time on a number of nodes dictated by a configured replication factor. For example, if data is stored on node A and needs to be replicated on B and C, then, in a pipelined write, data flows from node A to B and from B to C. A WTO/RTO occurs when a HDFS write/read operation is interrupted by a DN failure. WTOs occur for reducers while RTOs occur for mappers. CTOs can occur for both mappers and reducers, when they cannot connect to a DN.

C. Speculative Execution

The JT runs a SE algorithm which attempts to improve job running time by duplicating under-performing tasks. The SE algorithm in Hadoop 0.21.0 (latest version at the time of writing) is a variant of the LATE SE algorithm [15]. Both algorithms rely on progress rates. Both select a set of candidate tasks for SE and then SE the candidate task that is estimated to finish farthest in the future. The difference lies in the method used to select the candidates. Hadoop takes a statistical approach. A candidate for SE is a task whose progress rate is slower than the average progress rate across all started tasks of the same kind (i.e. map or reduce) by one standard deviation. Let $Z(T_i)$ be the progress rate of a task T_i and T_{set} the set of all running or completed tasks of the same kind. A task T_{cur} can be SE if:

$$avg(Z(T_i)_{T_i \in T_{set}}) - std(Z(T_i)_{T_i \in T_{set}}) > Z(T_{cur}) \quad (1)$$

Intuitively Hadoop SEs an under-performing task only when large variations in progress rates occur. In contrast, LATE attempts to SE tasks as early as possible. For LATE, the candidates are the tasks with the progress rate below a SlowTaskThreshold, which is a percentile of all the progress rates for a specific task type. Both algorithms SE a task only after it has ran for at least 60s. To minimize wasted resources both algorithms cap the number of active SE task instances at 1.

III. Dealing with TaskTracker Failures

Next, we describe the mechanisms related to TT failure detection and recovery in Hadoop. As we examine the design decisions in detail, it shall become apparent that tolerating network congestion and compute node overload is a key driver of many aspects of Hadoop's design. It also seems that Hadoop attributes non-responsiveness primarily to congestion or overload rather than to failure, and has no effective way of differentiating the two cases. To highlight some findings:

- Hadoop is willing to wait for non-responsive nodes for a long time (on the order of 10 minutes). This conservative design allows Hadoop to tolerate non-responsiveness caused by network congestion or compute node overload.
- A *completed* map task whose output data is inaccessible is re-executed very conservatively. This makes sense if the inaccessibility of the data is rooted in congestion or overload. This design decision is in stark contrast to the much more aggressive speculative re-execution of straggler tasks that are *still running* [15].
- The health of a reducer is a function of the progress of the shuffle phase (i.e. the number of successfully copied

Var.	Description	Var.	Description	Var.	Description
P_j^R	Time from reducer R's start until it last made progress	K_j^R	Nr. of failed shuffle attempts by reducer R	T_j^R	Time since the reducer R last made progress
$N_j(M)$	Nr. of notifications that map M's output is unavailable.	D_j^R	Nr. of map outputs copied by reducer R	S_j^R	Nr. of maps reducer R failed to shuffle from
$F_j^R(M)$	Nr. of times reducer R failed to copy map M's output	A_j^R	Total nr. of shuffles attempted by reducer R	Q_j	Maximum running time among completed maps
M_j	Nr. of maps (input splits) for a job			R_j	Nr. of reducers currently running

TABLE I
VARIABLES FOR FAILURE HANDLING IN HADOOP. THE FORMAT IS $X_j^R(M)$. A SUBSCRIPT DENOTES THE VARIABLE IS PER JOB. A SUPERSCRIPT DENOTES THE VARIABLE IS PER REDUCER. THE PARENTHESIS DENOTES THAT THE VARIABLE APPLIES TO A MAP.

map outputs). However, Hadoop ignores the underlying cause of unsuccessful shuffles.

We identify Hadoop's mechanisms by performing source code analysis on Hadoop version 0.21.0 (released Aug 2010), the latest version available at the time of writing. Hadoop infers failures by comparing variables against tunable threshold values. Table I lists the variables used by Hadoop. These variables are constantly updated by Hadoop during the course of a job. For clarity, we omit the names of the thresholds and instead use their default numerical values.

A. Declaring a TaskTracker Dead

TTs send heartbeats to the JT every 3s. The JT detects TT failures by checking every 200s if any TTs have not sent heartbeats for at least 600s. If a TT is declared dead, the tasks running on it at failure time are restarted on other nodes. Map tasks that completed on the dead TT are also restarted if the job is still in progress and contains any reducers.

B. Declaring Map Outputs Lost

The loss of a TT makes all map outputs it stores inaccessible to reducers. Hadoop recomputes a map output early (i.e. does not wait for the TT to be declared dead) if the JT receives enough notifications that reducers are unable to obtain the map output. The output of map M is recomputed if:

$$N_j(M) > 0.5 * R_j \quad \text{and} \quad N_j(M) \geq 3.$$

Let L be the list of map outputs that a reducer R wants to copy from TT H. A notification is sent immediately if a read error occurs while R is copying the output of some map M1 in L. $F_j^R(M)$ is incremented only for M1 in this case. If on the other hand R cannot connect to H, $F_j^R(M)$ is increased by 1 for every map M in L. If, after several unsuccessful connection attempts $F_j^R(M)\ mod\ 10 = 0$ for some M, then the TT responsible for R sends a notification to the JT that R cannot copy M's output. A back-off mechanism is used to dictate how soon after a connection error a node can be contacted again for map outputs. After every failure, for every map M for which $F_j^R(M)$ is incremented, a penalty is computed for the node running M:

$$penalty = 10 * (1.3)^{F_j^R(M)}.$$

A timer is set to *penalty* seconds in the future. Whenever a timer fires another connection is attempted.

C. Declaring a Reducer Faulty

A TT considers a reducer running on it to be faulty if the reducer failed too many times to copy map outputs. Three conditions need to be simultaneously true for a reducer to be considered faulty. First,

$$K_j^R \geq 0.5 * A_j^R.$$

In other words at least 50% of all shuffles attempted by reducer R need to fail. Second, either

$$S_j^R \geq 5 \quad \text{or} \quad S_j^R = M_j - D_j^R.$$

Third, either the reducer has not progressed enough or it has been stalled for much of its expected lifetime.

$$D_j^R < 0.5 * M_j \quad \text{or} \quad T_j^R \geq 0.5 * max(P_j^R, Q_j).$$

IV. DEALING WITH DATANODE FAILURES

Hadoop guards against DN failures using connection errors and timeouts. If a timeout expires or an existing connection is broken then a new set of source or destination nodes is obtained from the NN and the HDFS operations continue. This deceptively simple process hides design decisions that become significant inefficiencies under DN failures. We next expand on these.

A. Not Sharing Failure Information

In Hadoop, information about failures is not shared among different tasks in a job nor even among different code-level objects belonging to the same task.

At the task level, when a failure is encountered, this information is not shared with the other tasks. Therefore, tasks may be impacted by a failure even if the same failure had already been encountered by other tasks. In particular, a SE task can encounter the same failure that previously affected the initial task. The reason for this lack of task-level information sharing is that HDFS is designed with scalability in mind. To avoid placing excessive burden on the NN, much of the functionality, including failure detection and recovery, is relegated to the compute nodes.

Inside a task, information about failures is not shared among the objects composing the task. Rather, failure information is stored and used on a per object basis. Oftentimes, if a task involves multiple HDFS requests (examples follow in (IV-C)) the requests are served by different code-level objects. Without

sharing of failure information, an object can encounter a CTO even though another object from the same task encountered a CTO because of the same DN failure.

B. Large Timeouts

The timeouts used by HDFS requests to recover from DN failures are conservatively chosen, likely in order to to accommodate transient congestion episodes which are known to be common to data centers [5]. Both an initial task and a SE task can suffer from these timeouts (§IV-A). RTOs and CTOs are on the order of 60s while the WTOs are on the order of 480s. Differences of 5s-15s in absolute timeout values exist and depend on the position of a DN in the HDFS write pipeline. For this paper's argument these minute differences are inconsequential.

C. Number of HDFS Accesses

The number of HDFS requests that a task performs is greater than what a high level Hadoop description may suggest. The more HDFS accesses a task (either initial or SE) performs the bigger the chance a DN failure will impact the task.

Each map task performs 3 or 5 HDFS block read operations for processing one single input split. The first access is for the job.split file which identifies the input split for the job. The second access reads the input data while the third access reads the start of the subsequent block because a map input split can span HDFS block boundaries. Two more accesses can appear in case the map input split is not at the beginning of the HDFS file. Each reduce task performs a number of HDFS pipelined writes equal to the number of output blocks.

Even before a job is started, the JT makes multiple HDFS write requests to replicate job-specific files. By default, 6 such files are written: job.jar (java classes for the job), job.split and job.splitmetainfo (description of map inputs), job.xml (job parameter values), jobToken (security permissions) and job.info. Any timeout delaying the writing of these files delays the start of the whole job. At the end of a job, the NN writes to HDFS a file with details about the run. A delay of this write request delays the delivery of the job results to the user even though all tasks, and therefore the computation are finished.

V. Experiments on the Efficiency of Failure Detection and Recovery in Hadoop

A. Methodology

We independently study DN and TT failures because in many Hadoop deployments DNs and TTs are collocated and therefore, under compute node failures it would be hard to single-out the effect of each contributing mechanism.

For our experiments we used 15 machines from 4 racks in the OpenCirrus testbed [3]. One node is reserved for the JT and NN, the rest of the nodes run DN and TT processes. Each node has 2 quad-core Intel Xeon E5420 2.50 Ghz CPUs. The network is 10 to 1 oversubscribed. We run Hadoop 0.21.0 with the default configuration We first analyzed TT failures. After these experiments one of the compute node crashed and this left us with one less compute node for the DN failure analysis. Fortunately, the two sets of results are independent, therefore this failure does not affect out findings.

The job we use for this paper sorts 10GB of random data using 2 map slots per node and 2 reduce slots per node. In the experiments we vary the number of reducers and the number of reducer waves. 200 runs are performed for each experiment. Without failures the job takes on average 220s to complete. In our experiments we look at how failures impact the job running time and the job startup time. We consider the job startup time to be the time between the job submission and the job start assuming no waiting for task slots and no job queueing delays. We consider the job running time to be the time between the job start and job end.

Our findings are independent of job running time. Our goal is not to quantitatively analyze Hadoop performance over many types of jobs and failures but rather to expose the mechanisms that react under failures and their interactions. Our findings are also relevant to multiple failure scenarios since the same Hadoop mechanisms are involved. Also, oftentimes the multiple failures are independent and their effect can be estimated as the sum of the effect of single failures.

We consider TT and DN processes failures as well as the failure of the entire node running these processes. The difference lies in the existence of TCP reset (RST) packets that are sent by the host OS when just a process is killed. RST packets may serve as an early failure signal. We induce the single DN (or TT) fail-stop failures by randomly killing one of the DNs (or TTs) at a random time after the job is started and before the 220s mark. At the end of each run we restart Hadoop. We simulate a fail-stop failure of the compute node running a DN (or TT) by filtering all RST packets sent after the failure if the source port of the RST packet corresponds to the ports used by the failed DN (or TT).

B. TaskTracker Failure Analysis

1) Detailed Analysis: Our first experiment details Hadoop's behavior under TT process failures. We chose a process failure because the presence of RST packets enables a more thorough analysis. In the absence of RST packets CTOs would slow down Hadoop's reaction considerably and would mask several important but subtle interactions between Hadoop's mechanisms. Figure 1 plots job running time against TT failure injection time. 14 reducers in 1 wave are used in this experiment. Out of 200 runs, 193 are plotted and 7 failed. Note the large variation in job running time. This is due to a large variation in the efficiency of Hadoop's failure detection and recovery mechanisms. To explain, we cluster the results into 8 groups based on the underlying causes. The first 7 groups are depicted in the figure. The 7 failed runs form group G8. The highlights that the reader may want to keep in mind are:

- When the failure affects few reducers, failure detection and recovery is exacerbated.
- Detection and recovery time in Hadoop is unpredictable – an undesirable property in a distributed system. The time it takes reducers to send notifications is variable and so is the time necessary to detect TT failures.

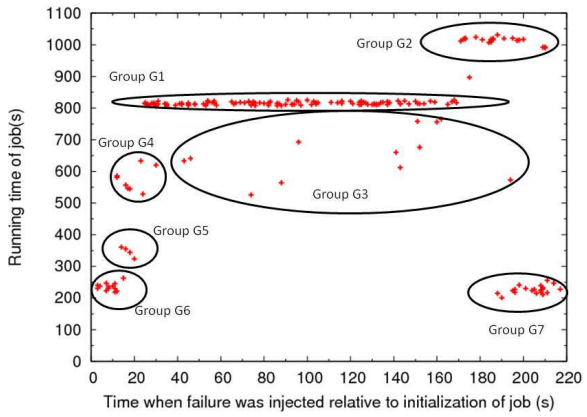

Fig. 1. Clusters of running times under failure. Without failure the average job running time is 220s

- The mechanisms used to detect lost map outputs and faulty reducers interact badly. Many reducers die unnecessarily as a result of attempting connections to a failed TT. This leads to unnecessary re-executions of reducers, thus exacerbating recovery.

Group G1. In G1, at least one map output on the failed TT was copied by all reducers before the failure. After the failure, the reducer on the failed TT is SE on another node and it will be unable to obtain the map outputs located on the failed TT. According to the penalty computation (§III-B) the SE reducer needs 10 failed connections attempts to the failed TT (416s in total) before a notification about the lost map outputs can be sent. For this one reducer to send 3 notifications and trigger the re-computation of a map, more than 1200s are typically necessary. The other reducers, even though still running, do not help send notifications because they already copied the lost map outputs. Thus, the TT timeout (§III-A) expires first. Only then are the maps on the failed TT restarted. This explains the large job running times in G1 and their constancy. G1 shows that the efficiency of failure detection and recovery in Hadoop is impacted when few reducers are affected and map outputs are lost.

Group G2. This group differs from G1 only in that the job running time is further increased by roughly 200s. This is caused by the mechanism Hadoop uses to check for failed TTs (§III-A). To explain, let D be the interval between checks, T_f the time of the failure, T_d the time the failure is detected, T_c the time the last check would be performed if no failures occurred. Also let $n * D$ be the time after which a TT is declared dead for not sending any heartbeats. For G1, $T_f < T_c$ and therefore $T_d = T_c + n * D$. However, for G2, $T_f > T_c$ and as a result $T_d = T_c + D + n * D$. In Hadoop, by default, $D = 200s$ and $n = 3$. The difference between T_d for the two groups is exactly the 200s that distinguish G2 from G1. In conclusion, the timing of the TT failure with respect to the JT checks can further increase job running time.

Group G3. In G3, the reducer on the failed TT is also SE but sends notifications considerably eariler than the usual 416s. We call these *early notifications*. 3 early notifications are sent and this causes the map outputs to be recomputed before the TT timeout expires (III-B). To explain early notifications consider the simplified example in Figure 2 where the penalty (III-B) is linear ($penalty = F_j^R(M)$) and the threshold for sending notifications is 5. A more detailed example is available in [7]. Reducer R needs to copy the output of two maps A and B located on the same node. Case a) shows regular notifications and occurs when connections to the node cannot be established.

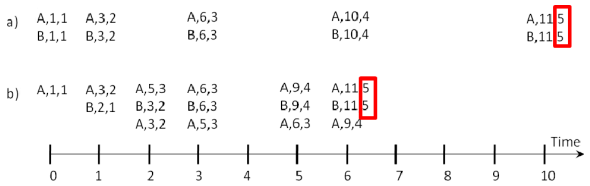

Fig. 2. Illustration of early notifications. The tuples have the format (map name, time the penalty expires, $F_j^R(M)$). The tuple values are taken immediately after the corresponding timestamp. Note that $F_j^R(A) = 5$ (i.e. notifications are sent) at different moments, shown by rectangles.

Case b) shows early notifications and can be caused by a read error during the copy of A's output. Due to the read error, only $F_j^R(A)$ is initially incremented. This de-synchronization between $F_j^R(A)$ and $F_j^R(B)$ causes the connections to the node to be attempted more frequently. As a result, failure counts increase faster and notifications are sent earlier.

Because the real function for calculating penalties in Hadoop is exponential (§III-B), a faster increase in the failure counts translates into large savings in time. As a result of early notifications, runs in G3 finish by as much as 300s faster than the runs in group G1.

Group G4. For G4, the failure occurs after the first map wave but before any of the map outputs from the first map wave is copied by all reducers. With multiple reducers still requiring the lost outputs, the JT receives enough notifications to start the map output re-computation §(III-B) before the TT timeout expires. The trait of the runs in G4 is that not enough early notifications are sent to trigger the re-computation of map outputs early.

Group G5. As opposed to G4, in G5, enough early notifications are sent to trigger map output re-computation earlier.

Group G6. The failure occurs during the first map wave, so no map outputs are lost. The maps on the failed TT are SE and this overlaps with subsequent maps waves. As a result, there is no noticeable impact on the job running time.

Group G7. This group contains runs where the TT was failed after all its tasks finished running correctly. As a result, the job running time is not affected.

Group G8. The failed jobs are caused by Hadoop's default behavior to abort a job if one of its tasks fails 4 times. A reduce task can fail 4 times because of the induced death problem described next.

2) Induced Reducer Death: In several groups we encounter the problem of induced reducer death. Even though the reducers run on healthy nodes, their death is caused by the repeated failure to connect to the failed TT. Such a reducer dies

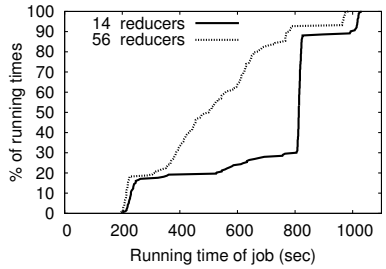

Fig. 3. Vary number of reducers

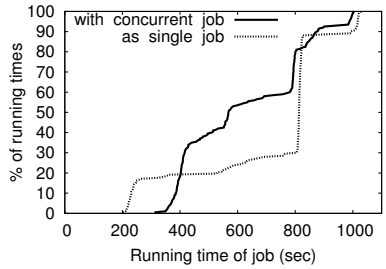

Fig. 4. Single job vs two concurrent jobs

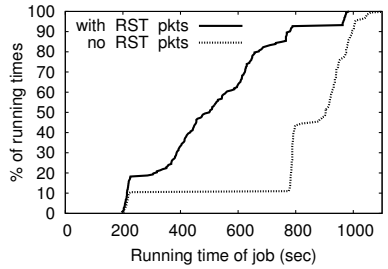

Fig. 5. Effect of RST packets

(possibly after sending notifications) because a large percent of its shuffles failed, it is stalled for too long and it copied all map output but the failed ones §(III-C). We also see reducers die within seconds of their start (without having sent notifications) because the conditions in §(III-C) become temporarily true when the failed node is chosen among the first nodes to connect to. In this case most of the shuffles fail and there is little progress made. Induced reducer death wastes time by causing task re-execution and wastes resources since shuffles need to be repeated.

3) Effect of Alternative Configurations: (§III) suggests failure detection is sensitive to the number of reducers. We increase the number of reducers to 56 and the number of reduce slots to 6 per node. Figure 3 shows the results. Considerably fewer runs rely on the expiration of the TT timeout compared to the 14 reducer case because more reducers means more chances to send enough notifications to trigger map output recomputation before the TT timeout expires. However, Hadoop still behaves unpredictably. The variation in job running time is more pronounced for 56 reducers because each reducer can behave differently: it can suffer from induced death or send notifications early. With a larger number of reducers these different behaviors yield many different outcomes.

Next, we run two concurrent instances of the 14 reducer job and analyze the effect the second scheduled job has on the running time of the first. Figure 4 shows the results for the first scheduled job compared to the case when it runs alone. Without failures, the first scheduled job finishes after a baseline time of roughly 400s. The increase from 220s to 400s is caused by the contention with the second job. The large variation in running times is still present. The second job does not directly help detect the failure faster because the counters in (§III) are defined per job. However, the presence of the second job indirectly influences the first job. Contention causes longer running time and in Hadoop this leads to increased SE of reducers. A larger percentage of jobs finish around the baseline time because sometimes the reducer on the failed TT is SE before the failure and copies the map outputs that will become lost. This increased SE also leads to more notifications so fewer jobs rely on the TT timeout expiration. Note also the running times around 850s. These jobs rely on the TT timeout expiration but suffer from the contention with the second job.

The next experiment mimics the failure of an entire node running a TT. Results are shown in Figure 5 for the 56 reducer job. The lack of RST packets means every connection attempt is subject to a 180s timeout. There is not enough time for reducers to send notifications so all jobs impacted by failure rely on the TT timeout expiration in order to continue. Moreover, reducers finish only after all their pending connections finish. If a pending connection is stuck waiting for the 180s timeout to expire, this stalls the whole reducer. This delay can also cause SE and therefore increased network contention. These factors are responsible for the variation in running time starting with 850s.

C. DataNode Failure Analysis

1) Delayed SE: To understand the DN failure results, we first take a deeper look at the interaction between DN failures and the SE algorithm. We show that these interactions can cause a detrimental effect which we deem delayed SE. This consists in one SE substantially delaying future SEs, or in the extreme case precluding any future SEs. The reason lies with the statistical nature of Hadoop's SE algorithm (§II-C). The DN experiments simulate the failure of entire compute nodes running DNs. Thus, RST packets do not appear. We do not present the effect of DN process failures since their impact is low. The results for different number of reducers and reducer waves are plotted in Figures 8, 9, 10 and 11. While a DN failure is expected to cause some job running time variation, the SE algorithm should eliminate substantial negative effects. Our results show the opposite. As an example consider Figure 8. In this experiment, the SE algorithm is largely ineffective after the map phase finishes (80s).

To explain delayed SE, consider the sample run in Figure 6 which plots the progress rates of two reducers alongside the value of the left side of equation (1) from (§II-C). We call this left side the *limit*. For this run, 13 reducers are started in total, all in 1 wave. In Hadoop, the progress rate for one task is the maximum rate over the progress rates of the initial tasks instance and the SE instance. When an instance of the task completes, the final progress rate for the task is that of the completed instance. Initially, the reducers need to wait for the map phase to end. Their progress rates are close to the average rate and the standard deviation is small. Hence, the limit is relatively high and close to the average rate. A DN failure occurs at time 176s and affects reducers R9 and R11. The failure interrupted the write phase of these reducers and therefore R9 and R11 are stuck in a WTO. At time 200s, R9 is SE. The progress rate for the SE R9 is very high because it does not need to wait for map outputs to be computed. The

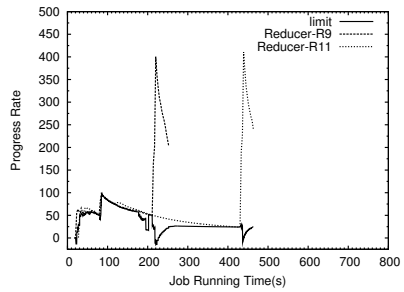

Fig. 6. Delayed speculative execution

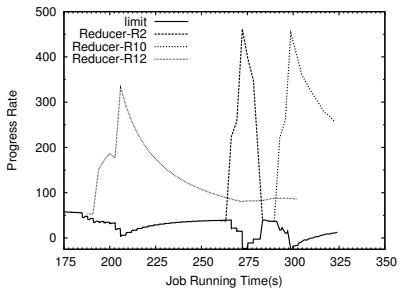

Fig. 7. Types of spikes

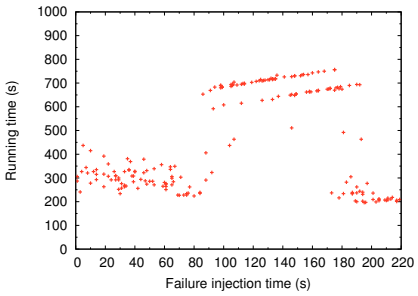

Fig. 8. 52 reducers - 1 wave

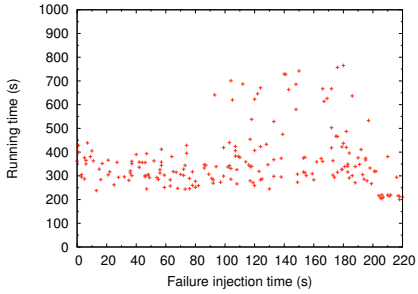

Fig. 9. 13 reducers - 1 wave

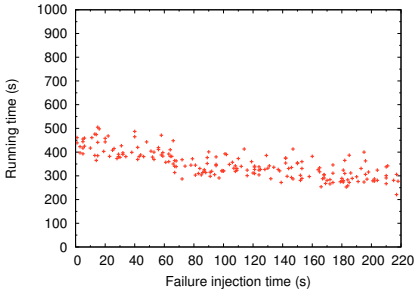

Fig. 10. 52 reducers - 4 waves

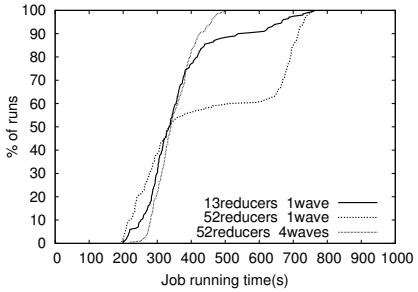

Fig. 11. CDF for the results in Figures 8, 9, 10

outputs are readily available for copying. The sort phase is also fast and this helps further increase the progress rate of SE R9. In Figure 6 this high rate is visible as a sudden large spike. As a result of the first spike, the average rate increases but the standard deviation increases more. Consequently, the limit decreases. Around 200s, the progress rate for R9 decreases because the SE R9 needs time to finish the write phase. Because of this progress rate decrease the limit increases but not to the point where it would allow R11 to be SE. R11 is SE only around 450s when its progress rate becomes lower than the limit due to the prolong stall in the WTO. In the extreme case, if the limit is lowered too much (can even become negative) then no further SE may be possible. To continue, all reducers stuck in an WTO would need to wait for the WTO to expire, because they cannot be SE. In the general case, spikes need not be isolated as in our example. Several reducers can have progress rate spikes at the same time.

The influence that a SE has on the limit and consequently on the start of subsequent SEs depends on the shape of the spike it creates. The ascending part of the spike decreases the limit. The severity of this ascending part depends on the amount of data the reducer needs to shuffle and on the speed of the network transfers. If little data is necessary or network transfers are very fast, then the reducer quickly finishes the shuffle and sort phases with a very high progress rate. The decreasing part of the spike influences how much the limit increases. In our runs we see three distinct decreasing shapes each of which influences the limit differently. We plot these decreasing shapes in Figure 7. A short decrease signals that the write phase proceeded normally (reducer R10). A longer decrease signals that the SE task also encountered a CTO because of the DN failure (reducer R12). A sharp decrease signals that the initial reducer finished shortly after the SE reducer finished the shuffle and sort phases (reducer R2).

2) Effects of Delayed SE on the Reduce Phase: Next, we explain in detail the results for DN failures injected during the reduce phase but after the map phase ends at roughly 80s.

For the 52-reducer, 1-wave case in Figure 8 the Hadoop SE algorithm is ineffective after the map phase ends ($\sim$80s). Notice the two parallel clusters of increasing job running time greater than 600s. The high job running times are caused by delayed SE. Due to delayed SE there is usually at least one reducer that cannot be SE and therefore has to wait for the WTO to expire before continuing. The reason why two clusters exist lies in a Hadoop code-level design choice where a reducer does not remember a failed DN if it caused a WTO. Thus, the same failed DN can cause the reducer to get stuck in a CTO after the WTO. On the other hand, after a CTO, the reducer remembers the failed DN and no further CTOs are caused by that failure. If the WTO occurs at the last block that the reducer needs to write, no CTOs can follow. Therefore, one cluster is formed by reducers suffering only from a WTO while the other cluster is comprised of reducers suffering from both a WTO and a CTO. The steady increase in job running time for each of the clusters is a function of how close to the end of the job the failure was injected.

For the 13-reducer, 1-wave case in Figure 9 the SE algorithm is more effective after the map phase end ($\sim$80s). Large job running times caused by delayed SE are still common but faster running times also exist. Compared to the 52-reducer case, each of the 13 reducers is responsible for writing 4 times more blocks and this considerably increases the chance that a CTO affects the write phase of a SE reducer. As a result of these CTOs the limit is increased more and SE becomes possible again thus resulting in some faster running times. Moreover, with only 13 reducers, less SEs are necessary

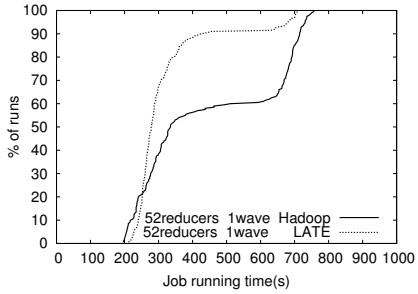

Fig. 12. LATE vs Hadoop CDF

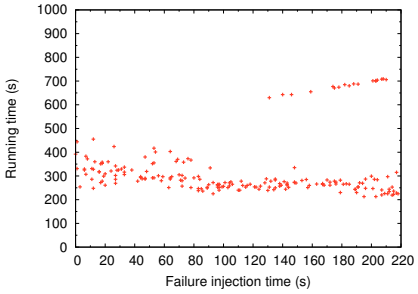

Fig. 13. LATE 52 reducers 1-wave

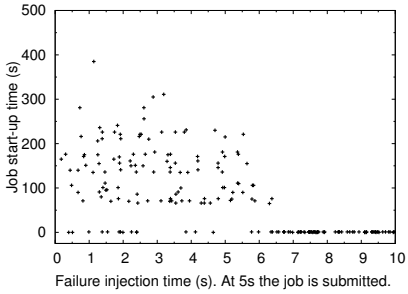

Fig. 14. Distribution of job start-up times

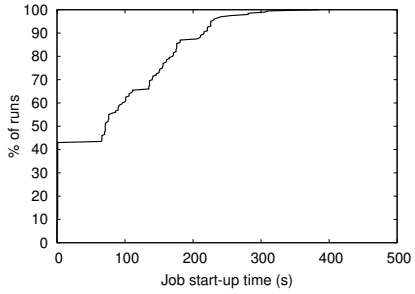

Fig. 15. CDF of job startup times

overall since fewer reducers are impacted by the failure. Sometimes only 1 or 2 SEs are necessary overall and if both are started at the same time (in the first spike) there is no other SE to be delayed.

For the 52-reducer, 4-wave case presented in Figure 10 the SE algorithm performs well. The reason is that the reducers in the last 3 waves all have very high progress rates initially since the map outputs are already available and the sort phase is fast. As a result, the limit becomes high and less influenced by subsequent spikes. Consequently, further SE is not impaired.

3) Using LATE as an Alternative SE: A SE algorithm that does not use aggregate statistical measures has its own set of drawbacks and does not necessarily remove inefficiencies during DN failures. This is best explained by analyzing the LATE SE algorithm on the 52-reducer, 1-wave case. As suggested in [15] we set the LATE SlowTaskThreshold to the 25th percentile. The results are plotted in Figures 12 and 13. Overall, LATE performs better than the Hadoop SE but running times larger than 600s are still present. These are not caused by delayed SE but rather by a different factor. Because of its more aggressive nature, LATE oftentimes SEs a task before that task has shown obvious signs of slowdown. In our experiment, this leads to tasks having both the initial and SE instance running before the failure injection. Recall from (§IV-A) that in Hadoop each task separately handles failure. The large job running times in this experiment are the runs in which both the initial task instance and its SE instance are stuck in a WTO because of the DN failure. Since there can be only one SE instance active at a time, the task can continue only after the WTOs expires for one of its instances. For the 52-reducer 4-wave and the 13-reducer 1-wave cases LATE did not produce large job running times. In these cases the problem described above is still possible but it is less probable since fewer reducers are active at the same time.

4) Effects of SE on the Map Phase: We next explain the results for the experiments in Figures 8, 9, 10, and 11 when the failure is injected before 80s. During the first 80s, failures overlap with the map phase and reducers are not yet in the write phase. We did not see cases of delayed SE for the map phase because maps did not have to wait for their input data to be available and the map progress rates were similar. In theory, delayed SE is also possible for the map phase when there is a large variation in progress rates among maps. This can happen in a topology with variable bandwidth. In these cases fast maps can skew the statistics.

Nevertheless, for the map phase we also identified inefficiencies of SE under DN failures. We encounter needless SE caused by the SE decision process not considering why a task is slow. For example, a map task can stall on a 60s CTO but the SE algorithm SEs a task only after the task has run for at least 60s. The SE can be needless here because it oftentimes occurs exactly when the CTO expires and the initial map task can continue and quickly finish.

When the failure occurs during the map phase, the job running times are smaller. However, job running time variation still exists and is caused by several factors most of which are common to all 3 experiments from Figures 8, 9, 10, and 11. For example, sometimes the NN encounters a CTO. Also, if one of the maps from the last map waves suffers from a CTO this impacts job running time more since the CTO cannot be overlapped with other map waves. The reducers are delayed until the map stuck in the CTO finishes. Specific to the 52-reducer, 4-wave case is the fact that timeouts are possibly encountered by reducers in every of the 4 waves. As a result, job running times are slightly larger for this scenario.

5) Delayed Job Start-up: We now analyze the effect of DN failures on the job start-up time. On each run, we fail one random DN at a random time starting 5s before the job submission time and ending 5s after. Job submission time is at 5s. The results are pictured in Figure 14 and 15.

Without DN failures, the job start-up time is roughly 1s, thus the JT finishes all write operations soon after 6s. This explains why failure occurring after 6s do not impact job start-up time. However, if the DN failure occurs during the first 6s most jobs are impacted. Recall that one failure can impact each of the 6 HDFS write operations started by the JT (§IV-C) because failure information is not shared among the different JT objects

responsible for the JT HDFS requests. Moreover, the HDFS write operation for the job.split file is more susceptible the DN failure influence because its pipeline can be larger than the default replication factor. In our runs, with 13 total DNs, we observed a 10 node pipeline for job.split when a replication factor of 3 was used. With 10 out of 13 DNs present in the job.split pipeline the chance that at least 1 of the 6 HDFS write operations was impacted by the failure was high. This explains why only few runs in Figure 14 were unaffected by a failure injected before 6s.

VI. Discussion

Delayed SE is a general problem. Delayed SE is a general concern for statistics-based SE algorithms such as Hadoop's. There are many ways to trigger delayed SE and failures are just one of them. The large HDFS timeouts are not a fundamental cause of delayed SE, although they can add to the overhead. Two common conditions are needed to trigger delayed SE's negative effects: the existence of slow tasks that would benefit from SE and conditions for tasks to suddenly speed up and create progress rate spikes. Slow tasks have many common causes including failures, timeouts, slow machines or slow network transfers. Progress rate spikes can be caused by varying input data availability (no more wait for map outputs after the map phase ends) or by small reducer input data size (small input size means fast progress). Varying network speeds can also cause progress rate spikes. This especially concerns recent proposals for circuit-augmented network topologies [14], [8] that inherently present large variations in bandwidth over different paths. While applying statistics in SE algorithms is in principle helpful, delayed SE highlights the difficulty in designing robust statistics-based SE algorithms because their underlying assumptions can easily be invalidated by various environmental conditions.

The need for adaptivity. Timeouts and thresholds in Hadoop are static. The disadvantage of static timeouts is that they cannot correctly handle all situations. Large timeouts are useful to cause a task's progress rate to slow down in order to be noticed by the SE algorithm. Conservative timeouts are also useful for protection against temporary network or compute node overload. However, short timeouts may allow failover to other DNs when data can be read from or written to multiple DNs. TT thresholds are also static and we have shown that this leads to poor performance when few reducers are impacted by a TT failure. Future work should consider adaptive timeouts that are set using system wide information about congestion, state of a job and availability of data.

Decoupling failure recovery from overload recovery. TCP connection failures are not only an indication of task failures but also of congestion. However, the two factors require different actions. It makes sense to restart a reducer placed disadvantageously in a network position susceptible to recurring congestion. However, it is inefficient to restart a reducer because it cannot connect to a failed TT. Unfortunately, the news of a connection failure does not by itself help Hadoop distinguish the underlying cause. This overloading of connection failure semantics ultimately leads to a more fragile system as exemplified by the induced reducer death problem. In the future, it can prove useful to decouple failure recovery from overload recovery entirely. For dealing with compute node load, solutions can leverage the history of a compute node's behavior which has been shown to be a good predictor of transient compute node load over short time scales [4]. For dealing with network congestion, the use of network protocols such as AQM/ECN [9], [12] that expose more information to the applications can be considered.

VII. Conclusion

In this paper we exposed and analyzed Hadoop's sluggish performance under compute node and process failures and identified several design decisions responsible: delayed SE, the lack of sharing of failure information and the overloading of connection failure semantics. We believe our findings are generally insightful beyond Hadoop and will enable the development of a class of large scale computing frameworks that is more predictable and more robust.

VIII. Addendum

This paper expands on [7] with a detailed analysis of DN failures. §III, §V, Figures 1-5 and Table I are related to their counterparts in [7]. We have improved the explanations and removed redundant or dull information. We believe presenting both TT and DN failures together helps paint a broader and more insightful picture on Hadoop's inefficiencies.

IX. Acknowledgements

This research was sponsored by NSF CAREER Award CNS-0448546, NeTS FIND CNS-0721990, NeTS CNS-1018807, by an Alfred P. Sloan Research Fellowship, an IBM Faculty Award, and by Microsoft Corp. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the Alfred P. Sloan Foundation, IBM Corp., Microsoft Corp., or the U.S. government.

References

[1] "Failure Rates in Google Data Centers," http://www.datacenterknowledge.com/archives/2008/05/30/ failure-rates-in-google-data-centers/.
[2] "Hadoop," http://hadoop.apache.org/.
[3] "Open Cirrus(TM)," https://opencirrus.org/.
[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *OSDI*, 2010.
[5] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding Data Center Traffic Characteristics," in *WREN*, 2009.
[6] J. Dean, "Experiences with MapReduce, an Abstraction for Large-Scale Computation," in *Keynote I: PACT*, 2006.
[7] F. Dinu and T. S. E. Ng, "Hadoop's Overload Tolerant Design Exacerbates Failure Detection and Recovery," in *NETDB*, 2011.
[8] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers," in *SIGCOMM*, 2010.
[9] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[10] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta, "Making Cloud Intermediate Data Fault-Tolerant," in *SOCC*, 2010.
[11] H. Lin, X. Ma, J. Archuleta, W. Feng, M. Gardner, and Z. Zhang, "MOON: MapReduce On Opportunistic eNvironments," in *HPDC*, 2010.
[12] K. Ramakrishnan, S. Floyd, and D. Black, *RFC 3168 - The Addition of Explicit Congestion Notification to IP*, 2001.
[13] K. Venkatesh and N. Nagappan, "Characterizing Cloud Computing Hardware Reliability," in *SOCC*, 2010.
[14] G. Wang, D. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan, "c-Through: Part-time Optics in Data Centers," in *SIGCOMM*, 2010.
[15] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *OSDI*, 2008.