Hadoop's Overload Tolerant Design Exacerbates Failure Detection and Recovery*

Florin Dinu T. S. Eugene Ng
Department of Computer Science, Rice University

Abstract

Data processing frameworks like Hadoop need to efficiently address failures, which are common occurrences in today's large-scale data center environments. Failures have a detrimental effect on the interactions between the framework's processes. Unfortunately, certain adverse but temporary conditions such as network or machine overload can have a similar effect. Treating this effect oblivious to the real underlying cause can lead to sluggish response to failures. We show that this is the case with Hadoop, which couples failure detection and recovery with overload handling into a conservative design with conservative parameter choices. As a result, Hadoop is oftentimes slow in reacting to failures and also exhibits large variations in response time under failure. These findings point to opportunities for future research on cross-layer data processing framework design.

General Terms

Performance, Measurement, Reliability

Keywords

Failure Recovery, Failure Detection, Hadoop

1. INTRODUCTION

Distributed data processing frameworks such as MapReduce [9] are increasingly being used by the database community for large scale data management tasks in the data center [7, 14]. In today's data center environment where commodity hardware and software are leveraged at scale, failures are the norm rather than the exception [1, 8, 16]. Consequently, large scale data processing frameworks need to automatically mask failures. Efficient handling of failures is important to minimize resource waste and user experience degradation. In this paper we analyze failure detection and recovery in Hadoop [2], a widely used implementation of MapReduce. Specifically, we explore fail-stop Task Tracker (TT) process failures and fail-stop failures of nodes running TTs. While we use Hadoop as our test case, we believe the insights drawn from this paper are informative for anyone building a framework with functionality similar to that of Hadoop's.

We find that although Hadoop runs most[1] jobs to completion under failures, from a performance standpoint failures are not masked. We discover that a single failure can lead to surprisingly large variations in job completion time. For example, the running time of a job that takes 220s with no failure can vary from 220s to as much as 1000s under failure. Interestingly, in our experiments the failure detection time is significant and is oftentimes the predominant cause for both the large job running times and their variation.

The fundamental reason behind this sluggish and unstable behavior is that the same functionality in Hadoop treats several adverse environmental conditions which have a similar effect on the network connections between Hadoop's processes. Temporary overload conditions such as network congestion or excessive end-host load can lead to TCP connection failures. TT permanent failures have the same effect. All these conditions are common in data centers [5, 8]. 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. Hadoop uses such a unified and conservative approach. It uses large, static threshold values and relies on TCP connection failures as an indication of task failure. We show that the efficiency of these 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

*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.

NetDB'11, 12-JUN-2011, Athens, Greece Copyright 2011 ACM 978-1-4503-0654-6/11/06 $10.00.

[1]A small number of jobs fail. The reasons are explained in §3.

on a node can induce task failures in other healthy nodes. These findings point to opportunities for future research on cross-layer data processing framework design. We expand on this in Section 5.

In the existing literature, smart replication of intermediate data (e.g. map output) has been proposed to improve performance under failure [12, 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 [12] that replicating intermediate data guards against certain failures at the cost of overhead during periods without failures. Moreover, replication can aggravate the severity of existing hot-spots. Therefore, complementing replication with an understanding of failure detection and recovery is equally important. Also complementary is the use of speculative execution [17, 4] which deals with the handling of under-performing outlier tasks. The state-of-the-art proposal in outlier mitigation [4] argues for cause-aware handling of outliers. Understanding the failure detection and recovery mechanism helps to enable such cause-aware decisions since failures are an important cause of outliers [4]. Existing work on leveraging opportunistic environments for large distributed computation [13] can also benefit from this understanding as such environments exhibit behavior that is similar to failures.

In §3 we present the mechanisms used by Hadoop for failure detection and recovery. §4 quantifies the performance of the mechanisms using experimental results. We conclude in §5 with a discussion on avenues for future work.

2. OVERVIEW

We briefly describe Hadoop background relevant to our paper. A Hadoop job has two types of tasks: mappers and reducers. Mappers read the job input data from a distributed file system (HDFS) and produce key-value pairs. These map outputs are stored locally on compute nodes, they are not replicated using HDFS. Each reducer processes a particular key range. For this, it copies map outputs from the mappers which produced values with that key (oftentimes all mappers). A reducer writes job output data to HDFS. A TaskTracker (TT) is a Hadoop process running on compute nodes which is responsible for starting and managing tasks locally. A TT has a number of mapper and reducer slots which determine task concurrency. For example, two reduce slots means a maximum of two reducers can concurrently run on a TT. A TT communicates regularly with a Job Tracker (JT), a centralized Hadoop component that decides when and where to start tasks. The JT also runs a speculative execution algorithm which attempts to improve job running time by duplicating under-performing tasks.

3. DEALING WITH FAILURES IN HADOOP

In this section, 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* [17].

- Our experiments in Section 4 shows that Hadoop's failure detection and recovery time is very unpredictable – an undesirable property in a distributed system. The mechanisms to detect lost map output and faulty reducers also interact badly, causing many unnecessary re-executions of reducers, thus exacerbating recovery. We call this the "Induced Reducer Death" problem.

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 1 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.

3.1 Declaring a Task Tracker Dead

TTs send heartbeats to the JT every 3s. The JT detects TT failure 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 the time of the failure are restarted on other nodes. Map tasks that completed on the dead TT and are part of a job still in progress are also restarted if the job contains any reducers.

3.2 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:

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

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 1: 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.

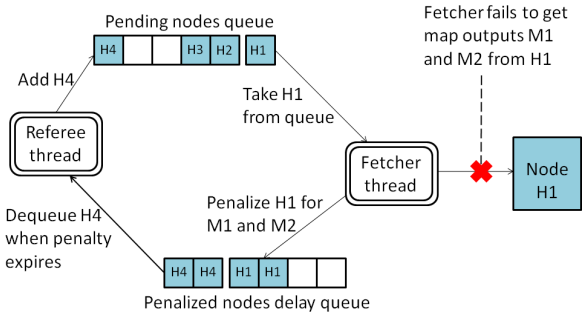

Figure 1: Penalizing hosts on failures

Sending notifications: Each reducer R has a number of Fetcher threads, a queue of pending nodes, and a delay queue. A node is placed in the pending queue when it has available map outputs. The life of a Fetcher consists of removing one node from the pending queue and copying its available map outputs sequentially. On error, a Fetcher temporarily penalizes the node by adding it to the delay queue, marks the not yet copied map outputs to be tried later and moves on to another node in the pending queue. Different actions are taken for different types of Fetcher errors. Let L be this list of map outputs a Fetcher is to copy from node H. If the Fetcher fails to connect to H, $F_j^R(M)$ is increased by 1 for every map M in L. If, after several unsuccessful attempts to copy map M's output, $F_j^R(M)\ mod\ 10 = 0$, the TT responsible for R notifies the JT that R cannot copy M's output. If the Fetcher successfully connects to H but a read error occurs while copying the output of some map M1, a notification for M1 is sent immediately to the JT. $F_j^R(M)$ is incremented only for M1.

Penalizing nodes: A back-off mechanism is used to dictate how soon after a connection error a node can be contacted again for map outputs. Hadoop's implementation of this mechanism is depicted in Figure 1. For every map M for which $F_j^R(M)$ is incremented on failure, the node running M is assigned a penalty and is added to a delay queue. In Figure 1, the Fetcher cannot establish the connection to H1 and therefore it adds H1 twice (once for M1 and once for M2) to the delay queue. While in the delay queue, a node is not serviced by Fetchers. When the penalty expires, a Referee thread dequeues each instance of a node from the delay queue and adds the node to the pending queue only if it is not already present. On failure, for every map M for which $F_j^R(M)$ is incremented, the penalty for the node running M is calculated as

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

3.3 Declaring a Reducer Faulty

A reducer is considered faulty if it failed too many times to copy map outputs. This decision is made at the TT. 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.

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

4. EFFICIENCY OF FAILURE DETECTION AND RECOVERY IN HADOOP

We use a fail-stop TT process failure to understand the behavior of Hadoop's failure detection and recovery mechanisms. We run Hadoop 0.21.0 with the default configuration on a 15 node, 4 rack cluster in the Open Cirrus testbed [3, 6]. One node is reserved for the JT. The other nodes are compute nodes and are distributed 3,4,4,3 in the racks. Each node has 2 quad-core Intel Xeon E5420 2.50GHz CPUs. The network is 10 to 1 oversubscribed. The job we use sorts 10GB of data using 160 maps and 14 reducers (1 per node), 2 maps slots and 2 reduce slots per node. Without failures, on average, our job takes roughly 220s to complete. On each run we randomly kill one of the TTs at a random time between 1s and 220s. At the end of each run we restart Hadoop. Our findings are independent of job running time. Our goal is to expose the mechanisms that react under failures and their interactions. Our findings are also relevant for multiple failure scenarios because each of those failures independently affects the job in a manner similar to a single failure.

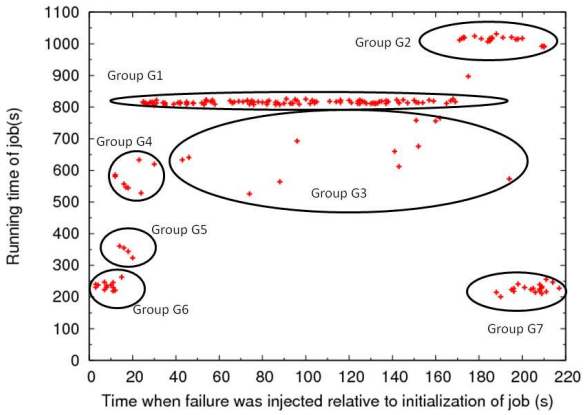

Figure 2: Clusters of running times under failure. Without failure the average job running time is 220s

Killing only the TT process and not the whole node causes the host OS to send TCP reset (RST) packets on connections attempted on the TT port. RST packets may serve as an early failure signal. This would not be the case if the entire node failed. The presence of RST packets allows us to better analyze Hadoop's failure detection and recovery behavior since otherwise connection timeouts would slow down Hadoop's reaction considerably. §4 presents an experiment without RST packets.

Figure 2 plots the job running time as a function of the time the failure was injected. Out of 200 runs, 193 are plotted and 7 failed. Note the large variation in job running time. The cause is a large variation in the efficiency of Hadoop's failure detection and recovery mechanisms. To explain the causes for these behaviors, we cluster the experiments into 8 groups based on similarity in the job running time. The first 7 groups are depicted in the figure. The 7 failed runs form group G8. Each group of experiments is analyzed in detail in the next section. These are the highlights that the reader may want to keep in mind:

- When the impact of the failure is restricted to a small number of reducers, failure detection and recovery is exacerbated.
- The time it takes to detect a TT failure depends on the relative timing of the TT failure with respect to the checks performed at the JT.
- The time it takes reducers to send notifications is variable and is caused by both design decisions as well as the timing of a reducer's shuffle attempts.
- Many reducers die unnecessarily as a result of attempting connections to a failed TT.

4.1 Detailed Analysis

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 speculatively executed on another node and it will be unable to obtain the map outputs located on the failed TT. According to the penalty computation (§3.2) 416s and 10 failed connections attempts are necessary for the reducer before $F_j^R(M)$ for any map M on the lost TT reaches the value 10 and one notification 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 in sending notifications because they already finished copying the lost map outputs. As a result, the TT timeout (§3.1) 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 (§3.1). 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 separate G2 and G1. In conclusion, the timing of the failure with respect to the checks can further increase job running time.

Group G3 In G3, the reducer on the failed TT is also speculatively executed but sends notifications considerably faster than the usual 416s. We call such notifications *early notifications*. 3 early notifications are sent and this causes the map outputs to be recomputed before the TT timeout expires. These early notification are explained by Hadoop's implementation of the penalty mechanism. For illustration purposes consider the simplified example in Figure 3 where the penalty is linear ($penalty = F_j^R(M)$) and the threshold for sending notifications is 5. Reducer R needs to copy the output of two maps A and B located on the same node. There are three distinct cases. Case a) occurs when connections to the node cannot be established.

Case b) can be caused by a read error during the copy of A's output. Because of the read error, only $F_j^R(A)$ is 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. A race condition between a Fetcher and the thread that adds map output availability events to a per-node data structure can also cause this behavior. The second thread may need to add several events for node H, but a Fetcher may connect to H before all events are added.

Case c) is caused by a race condition in Hadoop's implementation of the penalty mechanism. Consider again Fig-

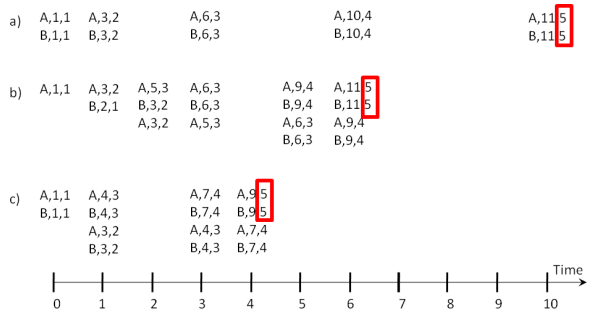

Figure 3: Effect of penalty computation in Hadoop. The values represent the contents of the reducer's penalized nodes queue immediately after the corresponding timestamp. The tuples have the format (map name, time the penalty expires, $F_j^R(M)$). Note that $F_j^R(A) = 5$ (i.e. notifications are sent) at different moments

ure 1. The Referee thread needs to dequeue H4 twice at time T. Usually this is done without interruption. First, H4 is dequeued and added to the pending nodes queue. Next it is again dequeued but it is not added to the queue because it is already present. If a Fetcher interrupts the operation and takes H4 after the first dequeue operation, the Referee will add H4 to the pending queue again. As a result, at time T, two connections will be attempted to node H4. This also results in early notifications failure counts increasing faster.

Because the real function for calculating penalties in Hadoop is exponential (§3.2) 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 §(3.2) before the TT timeout expires. The trait of the runs in G4 is that early notifications are not enough to trigger re-computation of map outputs. At least one of the necessary notifications is sent after the full 416s.

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

Group G6 The failure occurs during the first map wave, so no map outputs are lost. The maps on the failed TT are speculatively executed 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 Failed jobs are caused by Hadoop's default behavior to abort a job if the same task fails 4 times. A reduce task can fail 4 times because of the induced death problem described next.

4.2 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 (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 §(3.3). We also see reducers die within seconds of their start because the conditions in §(3.3) 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. Because they die quickly these reducers do not have time to send notifications. Induced reducer death wastes time waiting for re-execution and wastes resources since shuffles need to be performed again.

4.3 Effect of Alternative Configurations

The equations in (§3) show 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 4 shows the results. Considerably fewer runs rely on the expiration of the TT timeout compared to the 14 reducer case. This is because more reducers means more chances to send enough notifications to trigger map output re-computation 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 instances of our 14 reducer job concurrently and analyze the effect the second job has on the running time of the first scheduled job. 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. Results are shown in Figure 5. The large variation in running times is still present. The second job does not directly help detect the failure faster because the counters in (§3) 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 speculative execution of reducers. A larger percentage of jobs finish around the baseline time because sometimes the reducer on the failed TT is speculatively executed before the failure and copies the map outputs that will become lost. This increased speculative execution also leads to more notifications and therefore fewer jobs rely on the TT timeout expiration. Note also the running times around 850s. These jobs rely on the TT timeout expiration but also suffer from the contention with the second job.

The next experiment mimics the failure of a entire node running a TT by filtering all TCP RST packets sent from the TT port after the process failure is injected. Results are shown in Figure 6 for the 56 reducer job. No RST packets means every connection attempt is subject to a 180s timeout.

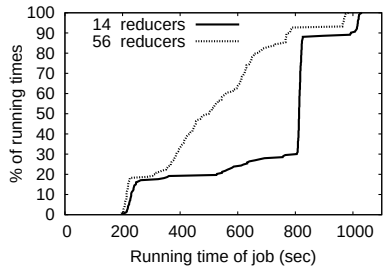

Figure 4: Vary number of reducers

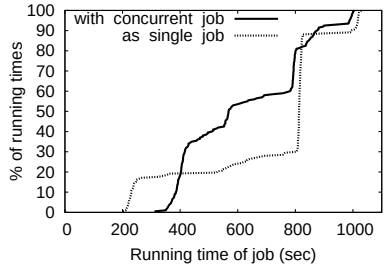

Figure 5: Single job vs two concurrent jobs

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 the shuffle phase only after all Fetchers finish. If a Fetcher is stuck waiting for the 180s timeout to expire, the whole reducer stalls until the Fetcher finishes. Also, waiting for Fetchers to finish can cause speculative execution and therefore increased network contention. These factors are responsible for the variation in running time starting with 850s.

5. DISCUSSION AND FUTURE WORK

Our analysis shows three basic principles behind Hadoop's failure detection and recovery mechanisms. First, Hadoop uses static, conservatively chosen thresholds to guard against unnecessary task restarts caused by transient network hot-spots or transient compute-node load. Second, Hadoop uses TCP connection failures as indication of task failures. Third, Hadoop uses the progress of the shuffle phase to identify bad reducers (§3.3).

These failure detection and recovery mechanisms are not without merit. Given a job with a single reducer wave and at least 4 reducers, the mechanisms should theoretically recover quickly from a failure occurring while the map phase is ongoing. This is because when reducers and maps run in parallel, the reducers tend to copy the same map output at roughly the same time. Therefore, reducers theoretically either all get the data or are all interrupted during data copy in which case read errors occur and notifications are sent.

In practice, reducers are not synchronized because the Hadoop scheduler can dictate different reducer starting times and because map output copy time can vary with network location or map output size. Also, Hadoop's mechanisms

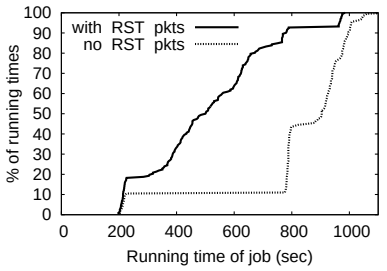

Figure 6: Effect of RST packets

cannot deal with failures occurring after the end of the map phase without the delays introduced by the penalty mechanism. Static thresholds cannot properly handle all situations. They have different efficiency depending on the progress of the job and the time of the failure. 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 reducers not progressing in the shuffle phase because of other failed tasks suffer from the induced reducer death problem.

For future work, adaptivity can leveraged when setting threshold values in order to take into account the current state of the network and that of a job. It can also 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 related problems over short time scales [4]. An interesting question is who should be responsible for gathering and providing this historical information. Should this be the responsibility of each application or can this functionality be offered as a common service to all applications? For dealing with network congestion, the use of network protocols that expose more information to the distributed applications can be considered. For example, leveraging AQM/ECN [11, 15] functionality on top of TCP can allow some information about network congestion to be available at compute nodes [10]. For a more radical solution, one can consider a cross-layer design that blurs the division of functionality that exists today and allows more direct communication between the distributed applications and the infrastructure. The network may cease to be a black-box to applications and instead can send direct information about its hot-spots to applications. This allows the applications to make more intelligent decisions regarding speculative execution and failure handling. Conversely, the distributed applications can inform the network about expected large transfers which allows for improved load balancing algorithms.

6. 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] R. Campbell, I. Gupta, M. Heath, S. Y. Ko, M. Kozuch, M. Kunze, T. Kwan, K. Lai, H. Y. Lee, M. Lyons, D. Milojicic, D. O'Hallaron, and Y. C. Soh. Open Cirrus Cloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research. In *Hotcloud*, 2009.

[7] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. Bradski, and A. Y. Ng. Map-Reduce for Machine Learning on Multicore. In *NIPS*, 2006.

[8] J. Dean. Experiences with MapReduce, an Abstraction for Large-Scale Computation. In *Keynote I: PACT*, 2006.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[10] F. Dinu and T. S.Eugene Ng. Gleaning network-wide congestion information from packet markings. In *Technical Report TR 10-08, Rice University*, July 2010. http://compsci.rice.edu/TR/TR_Download.cfm?SDID=277.

[11] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[12] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making Cloud Intermediate Data Fault-Tolerant. In *SOCC*, 2010.

[13] H. Lin, X. Ma, J. Archuleta, W. Feng, M. Gardner, and Z. Zhang. MOON: MapReduce On Opportunistic eNvironments. In *HPDC*, 2010.

[14] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. SIGMOD '09, 2009.

[15] K. Ramakrishnan, S. Floyd, and D. Black. *RFC 3168 - The Addition of Explicit Congestion Notification to IP*, 2001.

[16] K. Venkatesh and N. Nagappan. Characterizing Cloud Computing Hardware Reliability. In *SOCC*, 2010.

[17] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.