The Blind Men and the Elephant: Piecing Together Hadoop for Diagnosis

Xinghao Pan

CMU-CS-09-135
May 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Professor Priya Narasimhan, Chair
Professor Christos Faloutstos

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

Copyright © 2009 Xinghao Pan

This research was partially funded by the Defence Science & Technology Agency, Singapore, via the DSTA Overseas Scholarship.

Keywords: MapReduce, Hadoop, Failure Diagnosis

Abstract

Google's MapReduce framework enables distributed, data-intensive, parallel applications by decomposing a massive job into smaller (Map and Reduce) tasks and a massive data-set into smaller partitions, such that each task processes a different partition in parallel. However, performance problems in a distributed MapReduce system can be hard to diagnose and to localize to a specific node or a set of nodes. On the other hand, the structure of large number of nodes performing similar tasks naturally affords us opportunities for observing the system from multiple viewpoints.

We present a "Blind Men and the Elephant" (BliMeE) framework in which we exploit this structure, and demonstrate how problems in a MapReduce system can be diagnose by corroborating the multiple viewpoints. More specifically, we present algorithms within the BliMeE framework based on OS-level performance counters, on white-box metrics extracted from logs, and on application-level heartbeats. We show that our BliMeE algorithms are able to capture a variety of faults including resource hogs and application hangs, and to localize the fault to subsets of slave nodes in the MapReduce system.

In addition, we discuss how the diagnostic algorithms' outcomes can be further synthesized in a repeated application of the BliMeE approach. We present a simple supervised learning technique which allows us to identify a fault if it has been previously observed.

Acknowledgments

I would like to acknowledge all those who provided me with invaluable advice, assistance and support in the course of my research.

Firstly, I would like to express my gratitude to my advisor, Prof. Priya Narasimhan, for her guidance, infectious enthusiasm, and patience in the face of my often ridiculous ideas.

I would like to thank Keith Bare, Mike Kasick, and Eugene Marinelli, members of our research group with whom I embarked on the quest to automate diagnosis for Hadoop. Special thanks goes to Soila Kavulya and Rajeev Gandhi who persevered and provided priceless assistance towards the completion of this thesis. I am especially indebted to Jiaqi Tan for years of friendship and for being a trustworthy sounding board for my ideas.

Thanks also goes to Julio Lopez, U Kang and Christos Faloutsos, whose invaluable insights on Hadoop have led me down the path to the BliMeE framework. I also want to thank other members of the PDL community who gave me a chance to voice my ideas and get exposure for my work.

I would also like to thank my parents, Phoon Chong Cheng and Lau Kwai Hoi, who brought me up to be who I am today, and who always believed in me even when I have sometimes doubted myself.

Last but not least, I would like to thank my beloved girlfriend, Rui Jue, for constant encouragement and support during this especially challenging period. Without her, this research work would not have been at all possible.

Contents

1 Introduction **1**

2 Problem statement **3**

2.1 Motivation: Bug survey . . . 3

2.2 Motivation: Web console . . . 4

2.3 Thesis Statement . . . 5

2.3.1 Hypotheses . . . 5

2.3.2 Goals . . . 6

2.3.3 Non-goals . . . 6

2.3.4 Assumptions . . . 7

3 Overview **9**

3.1 Background: MapReduce and Hadoop . . . 9

3.2 Synopsis of BliMeE's Approach . . . 10

4 Diagnostic Approach **13**

4.1 High-Level Intuition . . . 13

4.2 Instrumentation Sources . . . 14

4.3 Component Algorithms . . . 16

4.3.1 Black-box Diagnosis . . . 17

4.3.2 White-box Diagnosis . . . 20

4.3.3 Heartbeat-based Diagnosis . . . 22

5 Synthesizing Views: “Reconstructing the Elephant” 25

6 Evaluation and Experimentation 27

6.1 Testbed and Workload 27

6.2 Injected Faults 27

7 Results 29

7.1 Diagnostic algorithms 29

7.1.1 Slave node faults 29

7.1.2 Master node faults 32

7.2 Synthesizing outcomes of diagnostic algorithms 34

8 Discussions 39

8.1 Data skew and unbalanced workload 39

8.2 Heterogenous hardware 39

8.3 Scalability of sampling for black-box algorithm 40

8.4 Raw data versus secondary diagnostic outcomes 40

9 Related Work 41

9.1 Diagnosing Failures and Performance Problems 41

9.2 Diagnosing MapReduce Systems 42

9.3 Instrumentation Tools 42

10 Conclusions 45

10.1 Conclusion 45

10.2 Future work 45

Publications

X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Black-Box Diagnosis of MapReduce Systems. Technical report, Carnegie Mellon University PDL, CMU-PDL-08-112, Sept 2008

X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Black-Box Diagnosis of MapReduce Systems. Poster at Third Workshop on Tackling Computer Systems Problems with Machine Learning Techniques, San Diego, CA, Dec 2008

X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Black-Box Diagnosis of MapReduce Systems. To appear at Hot Topics in Measurement & Modeling of Computer Systems, Seattle, WA, Jun 2009

J.Tan, **X. Pan**, S. Kavulya, R. Gandhi, P. Narasimhan. Salsa: Analyzing logs as state machines. In Workshop on Analysis of System Logs, San Diego, CA, Dec 2008

J. Tan, **X. Pan**, S. Kavulya, R. Gandhi, P. Narasimhan. Mochi: Visualizing Log-Anlaysis Based Tools for Debugging Hadoop. Under submission to Workshop on Hot Topics in Cloud Computing, May 2009.

Prior Work

Inevitably, as with a large systems research project, this work builds on existing/previous work of some of my collaborators at Carnegie Mellon University.

SALSA [24] was developed by Jiaqi Tan for extracting white-box states by analyzing appliction logs. In particular, his work extracted Map and Reduce tasks' durations by applying SALSA to the application logs generated by Hadoop (see Section 4.2).

Soila Kavulya conducted the bug survey mentioned in Section 2.1. In addition, she reproduced the faults injected in our experiments, and was the key person for collecting the data used in this thesis from the experiments conducted on Amazon's EC2.

List of Figures

2.1 Manifestation of 415 Hadoop bugs in survey. 4

2.2 Screenshot of Hadoop's web console. A single node experienced a 50% packet-loss, which caused all reducers across the system to stall. 5

3.1 Architecture of Hadoop, showing our instrumentation points. 9

3.2 The BliMeE approach: piecing together Hadoop (the elephant) by corroborating and synthesizing multiple viewpoints (different instrumentation sources from different nodes) across the entire Hadoop system. 11

4.1 Heartbeat rates of 4 slave nodes throughout an experiment with no fault injected. 23

4.2 Residual heartbeat propagation delay of 4 slave nodes throughout an experiment with injected hang2051. 24

7.1 True positive and false positive rates for faults on slave nodes, on 10 slaves cluster. 30

7.2 True positive and false positive rates for faults on slave nodes, on 50 slaves cluster. 31

7.3 Alarm rates for faults on master nodes, on 10 slaves cluster. 32

7.4 Alarm rates for faults on master nodes, on 50 slaves cluster. 33

7.5 Decision tree, classifies faults by the outcomes of the diagnostic algorithms. 34

7.6 Confusion matrix of fault classification. Each row reprsents a class of faults, and each column represents the classification given. Lighter shades indicate the actual fault was likely to be given a particular classification. . 36

List of Tables

4.1 Gathered black-box metrics (`sadc`-vector). 15

6.1 Injected faults, and the reported failures that they simulate. HADOOP-xxxx represents a Hadoop bug database entry. 28

7.1 Fault localization for various faults 37

Chapter 1

Introduction

Problem diagnosis is the science of automatically discovering if, what, when, where, why and how problems occur in systems and programs. More concretely, we are interested in knowing whether there is a problem, the type of problem that occurred, at what time the problem occurred, the location in the system at which it occurred, the causes of the problem, and its manifestations. In general, however, answering these questions may not be easy. At times, it is not entirely clear what constitues a problem or even if a problem exists. As systems today become increasingly large and complex, programmers and sysadmins have more trouble reasoning about their systems. The vast amounts of data can also easily overwhelm a human debugger.

MapReduce (MR) [8] is a programming framework and implementation introduced by Google for data-intensive cloud computing on commodity clusters. Hadoop [15], an open-source Java implementation of MapReduce, is used by Yahoo! and Facebook. Debugging the performance of Hadoop programs is difficult because of their scale and distributed nature. For example, Yahoo! Search Webmap is a large production Hadoop application that runs on a 10,000+ core Linux cluster and produces data that is used in Yahoo! Web search queries; the cluster's raw disk is 5+ petabytes in size [20]. Facebook has deployed multiple Hadoop clusters, with the biggest (as of June 2008) having 250 CPU cores and 1 petabyte of disk space [11]. These Facebook Hadoop clusters are used for a variety of jobs ranging from generating statistics about site usage, to fighting spam and determining application quality. Due to the increasing popularity of Hadoop, Amazon supports its usage for large data-crunching jobs through a new offering, Elastic MapReduce [2], that is supported on Amazon's pay-as-you-use EC2 cloud computing infrastructure. Hadoop can be certainly debugged by examining the local (node-specific) logs of its execution. These logs can be overwhelmingly large to analyze manually, e.g., a fairly simple *Sort* workload running for

850 seconds on a 5-node Hadoop cluster generates logs at each node, with a representative node's log being 6.9MB in size and containing 42,487 lines of logged statements. Furthermore, to reason about system-wide, cross-node problems, the logs from distinct nodes must be collectively analyzed, again manually. Hadoop provides a `LocalJobRunner` that runs jobs in isolation for debugging, but this does not support debugging of large jobs across many nodes. Hadoop also provides a simple web-based user interface that reveals key statistics about job execution (e.g., task-completion times). However, the user interface can be cumbersome (i.e., multiple clicks to retrieve information) to navigate when debugging a performance problem in a large MapReduce system, not to mention the fact that some kinds of problems might completely escape (i.e., not be visible in) this interface.

In this thesis, we propose to perform problem diagnosis for Hadoop systems by corroborating and synthesizing multiple distinct viewpoints of Hadoop's behavior. Hadoop, as a large distributed system, provides us with multiple sources (e.g. OS-level performance counters, tasks' durations as inferred from application logs) and multiple locations (the master node and large number of slave nodes) at which the instrumentation may be performed. We corroborate the instrumentated data from the different locations, and further synthesize this corroboration to piece together a picture of Hadoop's behavior for the purpose of problem diagnosis.

More concretely, the contributions of this thesis are:

- One set of diagnostic algorithms, each of which corroborates the views from different instrumentation points (blind men, in this case) in the system. For every node in the Hadoop cluster, each of these algorithms generates an intermediate diagnostic outcome that reflects the algorithm's confidence that the node is faulty. By thresholding the outcome, we obtain a binary diagnostic describing whether or not the node is faulty.

- The diagnostic algorithms then provide a secondary perspective into the MapReduce system. Treating the different intermediate diagnostic algorithms, in turn, as blind men, we synthesize the algorithms' outcomes to identify the kind of fault that has occurred in the system.

To the best of our knowledge, this is the first research result that aims to synthesize a variety of instrumentation sources and intermediate diagnostic outcomes to produce a holistic picture of Hadoop's behavior that then enables improved problem diagnosis.

Chapter 2

Problem statement

By corroborating and synthesizing multiple distinct viewpoints of Hadoop's behavior, we can provide fault diagnosis for Hadoop systems.

2.1 Motivation: Bug survey

We conducted a survey of 415 closed bugs in Hadoop's MapReduce and HDFS components over a two-year period from February 2007 to February 2009. We observed that performance problems and resource leaks accounted for 40% of reported bugs (see Figure 2.1). The average resolution time for the bug reports was 23 days. We also observed that 40% of the bugs were attributed to the masters (i.e., `jobtracker` and `namenode`), while 43% of the bugs were attributed to the slaves (i.e., `tasktracker` and `datanode`). The remainder of the bugs were attributed to the clients and shared libraries. The goal of our research is to study the effect of resource contention and performance problems in the Hadoop master and slave nodes, and to automate the debugging of performance problems.

BliMeE alleviates the debugging task by corroborating views from instrumentation points to indict nodes, and synthesizes the intermediate outcomes of the diagnosis algorithms (as a secondary view) to identify the fault. We synthesize the different views because analysing each individual metric in isolation might not provide sufficient insight for localizing the problem. For example, high `iowait`, `user`, and `system` CPU utilization might be perfectly legitimate in an application that checkpoints a lot of data to disk. However, a bad disk as described in HADOOP-5547 might manifest high `iowait`, and low `user`, and `system` CPU utilization. If we did not analyze the correlations between

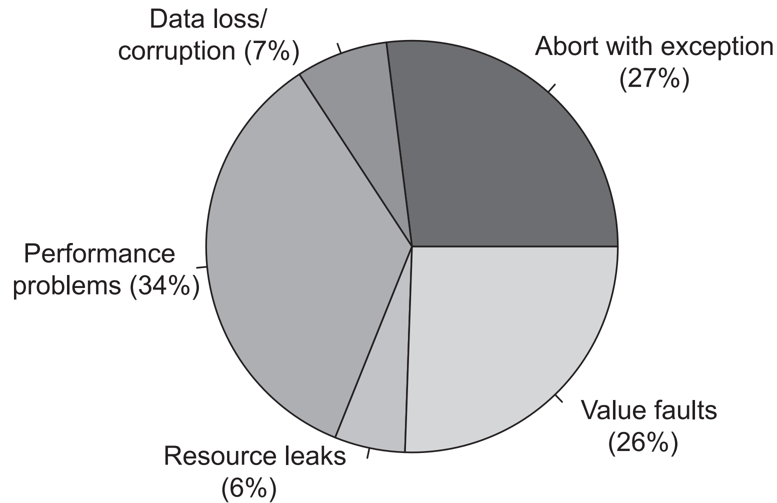

Figure 2.1: Manifestation of 415 Hadoop bugs in survey.

the different metrics, we might mistake a legitimate workload for a performance problem.

2.2 Motivation: Web console

To the best of our knowlege, the only monitoring or debugging tool currently available to Hadoop users is the built-in web console. The web console reports the Map and Reduce tasks' durations to the user in both textual and visual form, as seen in Figure 2.2. System logs captured from `stdout` and `stderr` on each node are also available via the web console.

However, the web console alone does not suffice as a debugging aid. First, it presents a static snapshot of the system. For a user or sysadmin to understand the dynamic evolution of the Hadoop system, he/she must continuously monitor the web console. Second, the web console presents raw data to the user without highlighting where potential problems may lie. The user has to manually investigate for indicators and possible manifestations of problems. To get to the relevant information for each node often requires multiple clicks, which simply is not scalable.

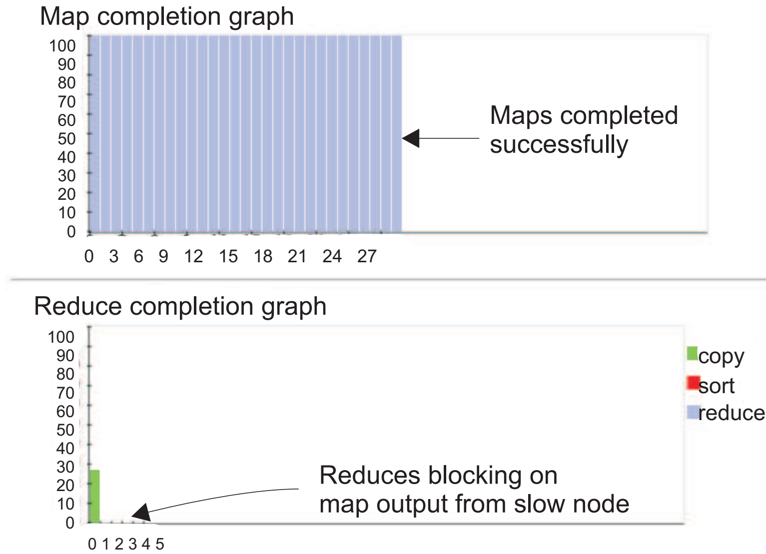

Figure 2.2: Screenshot of Hadoop's web console. A single node experienced a 50% packet-loss, which caused all reducers across the system to stall.

2.3 Thesis Statement

By corroborating and synthesizing multiple distinct viewpoints of Hadoop's behavior, we can provide fault diagnosis for Hadoop systems.

In other words, we will identify multiple distinct viewpoints from which to observe the Hadoop systems; we will corroborate (or correlate) these viewpoints across the entire Hadoop system; we will synthesize this corroboration (or correlation) to piece together a picture of Hadoop's behavior for the sake of performing fault diagnosis.

This will be realized in our Blind Men and the Elephant (BliMeE) framework.

2.3.1 Hypotheses

We hypothesize that large parallel distributed systems like MapReduce have multiple instumentation points from which one can observe the system, and that corroborating the multiple instrumentation points allows one to gain greater insight into the system.

We also hypothesize that different faults manifest in different ways on different met-

rics. While diagnostic algorithms based on subsets of the metrics may individually fail to identify the fault, we hypothesize that synthesizing the outcomes from multiple diagnostic algorithms can result in better diagnosis of the problem.

2.3.2 Goals

There are two high-level goals in this thesis:

- We seek to indict faulty slave nodes for a variety for faults. Hadoop offers multiple locations at which we can instrument and observe the system. Corroborating the instrumentated data in the BliMeEframework will allow us to indict faulty slave nodes.

- We show that by synthesizing the outcomes of diagnostic algorithms, we can improve the localization of the fault, and furthermore identify the fault if it were previously seen.

Furthermore, in our diagnosis, we primarily target performance problems that result in a Hadoop job taking longer to complete than expected for a variety of reasons, including external/environmental factors on the node (e.g., a non-Hadoop process consuming system resources to the detriment of the Hadoop job), reasons not specific to user MapReduce code (e.g., bugs in Hadoop), or interactions between user MapReduce code and the Hadoop infrastructure (e.g., bugs in Hadoop that are triggered by user code). We intentionally do not target faults due to bugs in user-written MapReduce application code. We seek to have our diagnosis approach work in production environments, requiring no modifications to existing MapReduce-application code or existing Hadoop infrastructure.

2.3.3 Non-goals

We do not aim to have fine-grained diagnosis, that is, our diagnosis will not identify the root-cause, nor pinpoint exactly the offending line of code at which the fault originated. We also do not aim to have complete coverage of either faults or all possible instrumentation sources. While we keep these ultimate goals of problem diagnosis in mind, they are not the primary focus of this thesis.

2.3.4 Assumptions

We assume that MapReduce applications and infrastructure are the dominant sources of activity on every node. We assume that a majority of the MapReduce nodes are problem-free and homogeneous in hardware.

Chapter 3

Overview

3.1 Background: MapReduce and Hadoop

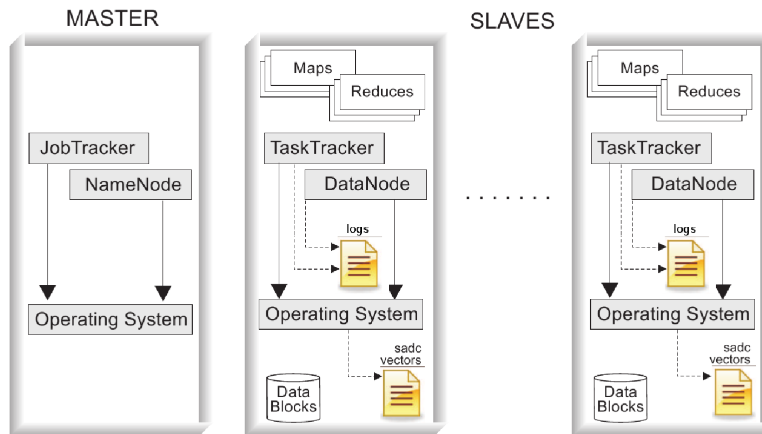

Figure 3.1: Architecture of Hadoop, showing our instrumentation points.

Hadoop [15] is an open-source implementation of Google's MapReduce [8] framework that enables distributed, data-intensive, parallel applications by decomposing a massive job into smaller (Map and Reduce) tasks and a massive data-set into smaller partitions, such that each task processes a different partition in parallel. A Hadoop job consists of a group of Map and Reduce tasks performing some data-intensive computation. Hadoop uses the Hadoop Distributed File System (HDFS), an implementation of the Google Filesystem [16], to share data amongst the distributed tasks in the system. HDFS splits and stores files

as fixed-size blocks (except for the last block). Hadoop uses a master-slave architecture, as shown in Figure 3.1, with a Hadoop cluster having a unique master node and multiple slave nodes.

The master node typically runs two daemons: (1) the JobTracker that schedules and manages all of the tasks belonging to a running job; and (2) the NameNode that manages the HDFS namespace by providing a filename-to-block mapping, and regulates access to files by clients (the executing tasks). Each slave node runs two daemons: (1) the TaskTracker that launches tasks locally on its host, as directed by the JobTracker, and then tracks the progress of each of these tasks; and (2) the DataNode that serves data blocks (on its local disk) to HDFS clients. Hadoop provides fault-tolerance by using periodic keep-alive heartbeats from slave daemons (from TaskTrackers to the JobTracker and DataNodes to the NameNode). Each Hadoop daemon generates logs that record the local execution of the daemons as well as MapReduce application tasks and local accesses to data.

3.2 Synopsis of BliMeE's Approach

There can be multiple perspectives of Hadoop's or a MapReduce application's behavior, e.g., from the operating-system's viewpoint, from the application's viewpoint, from the network's viewpoint, etc. In the mythological story of *The Blind Men and the Elephant* , each blind man arrives at a different conclusion based on his limited perspective of the elephant. It is only by corroborating all the blind men's perspectives that one can reconstruct a complete picture of the elephant. Along the same vein. MapReduce, as a large distributed system, affords us many opportunities or instrumentation points to observe the system's behavior. Each view can be thought to correspond to a "blind man", and the MapReduce system itself to the elephant. By mediating across and synthesizing the different views, our approach, BliMeE, acts as the wise man and is able to diagnose problems in MapReduce. In particular, we apply the BliMeE approach at two different levels: instrumentation points and diagnostic algorithms.

- We present a number of diagnostic algorithms, each of which corroborates the views from different instrumentation points (the blind men, in this case) in the system. For every node in the Hadoop cluster, each of these algorithms generates a diagnostic statistic that reflects the algorithm's confidence that the node is faulty. By thresholding the statistic, we obtain a binary diagnostic describing whether or not the node is faulty.

- The diagnostic algorithms then provide a secondary perspective into the MapReduce

system. Treating the different diagnostic algorithms, in turn, as the blind men, we synthesize the algorithms' outcomes to identify the kind of fault that has occurred in the system.

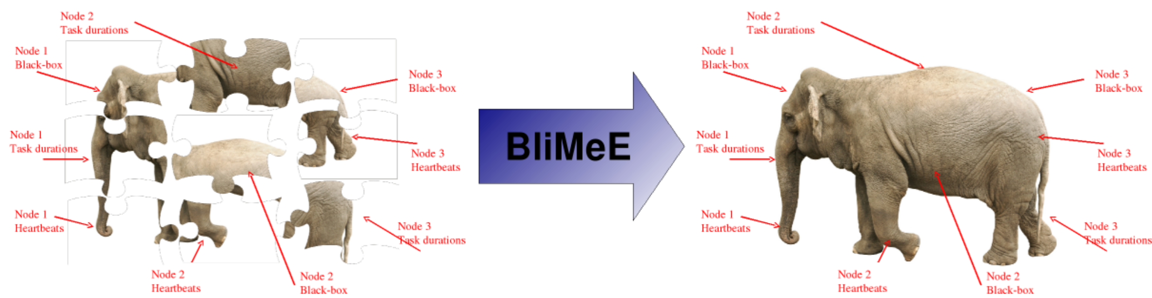

Figure 3.2: The BliMeE approach: piecing together Hadoop (the elephant) by corroborating and synthesizing multiple viewpoints (different instrumentation sources from different nodes) across the entire Hadoop system.

We produce two levels of diagnostic outcomes. First, a collection of diagnostic algorithms (Section 4.3) each produces a set of slave nodes in the cluster that caused the job to experience an increased runtime (this set could be empty, from which it can be inferred that the algorithms did not diagnose any problems). Second, the synthesis of the outputs of these algorithms (Section 5), given prior labelled training data, produces for each node, the most likely fault from a class of previously seen faults in the labelled training data (possibly no fault) present in the cluster, and whether that node suffers from the identifed fault.

[Blind Men #1] Views from instrumentation points. Large distributed systems have different instrumentation points from which the behavior and properties of the system can be simultaneously observed. Often, these instrumentation points can serve as somewhat redudnant or corroborating (albeit distinct) views of the system. We differentiate between white-box metrics (instrumented data gathered through pre-existing logging statements within the Hadoop or MapReduce-application code) and black-box metrics (instrumented data gathered from the operating system, without Hadoop's or the application's knowledge). Both types of metrics can be currently gathered without any modifications to Hadoop or the MapReduce application itself. We exploit the parallel distributed nature of MapReduce systems by applying the BliMeE framework to three distinct types of instrumentation views of the system: (white-box) heartbeat-related metrics, extracted from the Hadoop logs; (white-box) execution-state metrics, extracted from the Hadoop logs; and

(black-box) performance and resource-usage metrics, extracted from the operating system. We describe the details of these instrumentation sources, along with the algorithms used to analyze the collected metrics, in Section 4.

[Blind Men #2] Views from diagnostic algorithms. Faults in a distributed system can manifest on different sets of metrics in different ways. A given fault might manifest only on a specific subset of metrics or, alternatively, might manifest in a system-wide correlated manner, i.e., the fault originates on one node but also manifests on on some metrics on other (otherwise untainted) nodes, due to the inherent communication/coupling across nodes. Thus, diagnosis algorithms that focus on or analyze a selective set of metrics will likely miss faults that do not manifest on that specific set of metrics; worse still, as in the case of cascading fault-manifestations, diagnosis algorithms might wrongly indict nodes (that are truly innocent and not the original root-cause of the fault) that exhibit any correlated manifestation of the fault. By further synthesizing the outcomes of our diagnostic algorithms, BliMeE gains greater insight into the distributed nature of the fault, allowing it to identify the kind of fault as well as the true culprit node, despite any correlated fault-manifestations.

Chapter 4

Diagnostic Approach

This section describes the internals of the BliMeE diagnotic approach, including the instrumentation sources and the diagnosis algorithms that analyze the corroborating instrumentation-based views of the system.

4.1 High-Level Intuition

First, Hadoop uses heartbeats as a keep-alive mechanism. The heartbeats are periodically sent from the slave nodes (TaskTrackers and DataNodes) to the master node (JobTracker and NameNode). Upon receipt of the hearbeat, the master node sends a *heartbeatResponse* to the slave node, indicating that it has received the heartbeat. Both the receipt of heartbeats at the JobTracker and of the heartbeatResponses at the TaskTrackers are recorded in the respective daemon's logs, together with the heartbeat's unique id. (While the same heartbeat mechanism exists between the NameNode and DataNodes, the exchange is not recorded in the logs.) We corroborate these log messages in one of our diagnosis algorithms. In addition, control messages are embedded in the heartbeats and heartbeatResponses sent between the TaskTracker and JobTracker: the JobTracker uses the heartbeatResponse to assign new tasks, and upon completion of tasks, the TaskTrackers *proactively* send heartbeats to the JobTracker indicating the task completion. As such, the rate at which a TaskTracker sends heartbeats is indicative the workload it is experiencing. In another of our diagnosis algorithms, we corroborate the rate of heartbeats across the slave nodes.

Secondly, a job consists of multiple copies of Map and Reduce tasks, each running the same piece of code, albeit operating on different portions of the dataset. We expect that the the Map tasks exhibit similar behavior with other Map tasks, and that Reduce tasks

exhibit similar behavior with other Reduce tasks. More abstractly, since each Map task M_i is an instance from the global set $\mathscr{S}_{map}$ of all Map tasks, any property $P(M_i)$ of the Map task M_i can be treated as a single sample from a global distribution of the property P over all Map tasks. The same is true for Reduce tasks, as well as other white-box states that we can extract from the application logs. In particular, we are most interested in the completion times of Map and Reduce tasks. Since each TaskTracker executes a subset of the Map and Reduce tasks, each TaskTracker observes a sampling of the global distribution of completion times of tasks. Then, the local distribution of task completion times on each node is a data source, and in the absence of faults, these times should corroborate across all TaskTrackers.

Finally, we apply a similar principle to the observations on each slave node's OS-level performance counters. Since each slave node executes a subset of the global set of tasks, and the OS-level performance counters are dependent on the slave node's workload, its OS-level performance counters can be thought of as a sampling of a global distribution of OS-level performance counters too.

4.2 Instrumentation Sources

Black-box: OS-level performance metrics. On each node in the Hadoop cluster, we gather and analyze black-box (i.e., OS-level) performance metrics, without requiring any modifications to Hadoop, the MapReduce applications or the OS to collect these metrics. For black-box data collection, we use `sysstat`'s `sadc` program [17] and a custom script that samples TCP-related and `netstat` metrics to collect 16 metrics (listed in Table 4.1) from */proc*, once every second. We use the term `sadc`-vector to denote a vector containing samples of these 16 metrics, all extracted at the same instant of time. We then use these `sadc`-vectors as our (black-box) metrics for diagnosis.

White-box: Execution-state metrics. We collect the system logs generated by Hadoop's own native logging code from the TaskTracker and DataNode daemons on each slave node. We then use our Hadoop-log analysis tool (called SALSA [24] and, its successor, Mochi [25]) to extract inferred state-machine views of the execution of each daemon. This log-analysis treats each entry in the log as an event, and uses particular Hadoop-specific events of interest to identify states in the execution flow of each daemon, e.g., the states extracted from the TaskTracker include the `Map` and `Reduce` states, and the states extracted from the DataNode include the `ReadBlock` and `WriteBlock` states. The output of the log-analysis consists of sequences of state executions, the data flows between these states, and the duration of each of these states, for every node in the Hadoop cluster. We then examine

Metric	Description
user	% CPU time in user-space
system	% CPU time in kernel-space
iowait	% CPU time waiting for I/O
ctxt	Context switches per second
runq-sz	# processes waiting to run
plist-sz	Total # of processes and threads
ldavg-1	system load average for the last minute
bread	Total bytes read from disk /s
bwrtn	Total bytes written to disk /s
eth-rxbyt	Network bytes received /s
eth-txbyt	Network bytes transmitted /s
pgpgin	KBytes paged in from disk /s
pgpgout	KBytes paged out to disk /s
fault	Page faults (major+minor) /s
TCPAbortOnData	# of TCP connections aborted with data in queue
rto-max	Maximum TCP retransmission timeout

Table 4.1: Gathered black-box metrics (`sadc`-vector).

the durations of these execution states as the metrics in our (white-box) diagnosis.

White-box: Heartbeat metrics. Heartbeat events are also recorded in Hadoop's native logs, and we extract these from the master-node (JobTracker, NameNode) and the slave-node (TaskTracker, DataNode) logs. Although both are derived from white-box instrumentation sources, these heartbeat metrics are orthogonal to the previously described execution-state metrics.

For each heartbeat event between the master node and a given slave node, a log entry is recorded in both the master-node's log and that specific slave-node's logs, along with with a matching monotonically increasing heartbeat sequence-number. Each (master node, slave node) pair has an independent, unique space of hearbeat sequence-numbers. Each message is timestamped with a millisecond-resolution timestamp. The master-node's log first records a message as it receives the heartbeat from the slave node, and the slave node's log then records a message as it receives the master node's acknowledgment/response for the same heartbeat. Hadoop has an interesting implementation artefact where the master node logs the slave node's heartbeat message, and then performs additional processing within the same thread, before it acknowledges the slave. Analogously, the acknowledgement/response is first processed by the slave node before it is finally logged. This artefact is exploited, as we explain in the next section.

4.3 Component Algorithms

The algorithms we present cater to different classes of data, none of which is alike the other. Our black-box metrics are collected at a per-second (or per-unit time) sampling rate. More importantly, the black-box metrics are closely related with one another and together represent a single, (near) instantaneous state of the physical machine. Thus, black-box metrics lend themselves to a combined treatment as a single state. In contrast, white-box state durations are themselves a measure of time. Instead of a sample per-unit time, we obtain one sample for each instance of a state (e.g. per Map task, per Reduce task) resulting in a variable number of samples at different machines and times. While they represent the execution state of the application, the representation is not instantaneous but long-lived. Furthermore, the relative lack of tight coupling between multiple white-box metrics renders it meaningless to treat them in a fashion similar to that of the black-box metrics. Heartbeats are unlike both the black and white box metrics because they are periodic and are nearly stateless. The property of interest, in this case, is the rate at which the heartbeats are sent. We also exploit the fact that a heartbeat is a uniquely identified communication message that is logged at both the source and the destination. This property is absent and irrelevant in the former two cases since the both the black and white box metrics reflect local node behavior.

We argue that the algorithms are not restricted to only the data that we have collected, but to more general classes of data. For example, in contrast to previous work on black-box metrics, we have extended our black-box model to include TCP data, which is closely related to the OS-level performance counters that we previously collected. One can also imagine building additional models based on other types of closely related data. Our white-box algorithm can be applied to any type of long-lived states for which we can measure durations. In fact, in this paper, we show the same algorithm apply to both Map and Reduce tasks' durations. We present two heartbeat-based algorithms, the first of which depends on the event-like nature of heartbeats, and the second which depends on the message-like nature of heartbeats. The former is amenable to other types of events, and the latter can be applied to any class of uniquely identified messages that is logged on both ends of the communication. For instance, we were initially interested in DataNode-NameNode heartbeats in addition to TaskTracker-JobTracker heartbeats, but realized that these were not logged by default. Communication between TaskTrackers, or between TaskTrackers and DataNodes, are also potential targets that we can analyze.

The diagnostic algorithms that we present below work by corroborating the redundant views of Hadoop; essentially, the instrumentation sources on multiple nodes correspond to the "blind-men" in the BliMeE framework. The algorithms each have an implicit notion

of what constitutes correct, normal, or more generally, good behavior. For each node in a Hadoop cluster, each algorithm then generates diagnostic statistic that reflects the degree to which the algorithm thinks the node is behaving correctly. A binary diagnostic of whether or not a node is faulty is then produced for each node, for each algorithm, by thresholding the diagnostic statistic against a predefined value.

In the next section, we will synthesize the outcomes of our diagnostic algorithms, in a repeated application of the BliMeE concept, treating each algorithm as a "blind-man", and synthesizing the different perspectives provided by our algorithms.

4.3.1 Black-box Diagnosis

Intuition. Each slave node in the MapReduce system executes a subset of the global set of Map and Reduce tasks. We note that all MapReduce jobs follow the same temporal ordering: Map tasks are assigned, and begin by reading input data from DataNodes; upon completion, the MapOutput data is Shuffled to the Reduce tasks; eventually the job terminates after the Reduce tasks write their outputs to the DataNodes. Since each slave ndoe executes a subset of the global set of Map and Reduce tasks, this temporal ordering is reflected on the slave nodes as well. Hence, we expect that within reasonably large windows of time, slaves nodes encounter similar workloads that are reflective of the global workload of the MapReduce system. In the language of BliMeE, each slave node is a "blind-man" who has a limited view of the entire system.

The workload on each slave node at every instant of time is represented by the black-box metrics that we collect on the slave node. More abstractly, we can represent the global workload of the MapReduce system as a global distribution of black-box metrics. The observed black-box metrics on each slave node is then a sampling of the global distribution at the time of collection. Our black-box diagnosis algorithm, then, corroborates the black-box views on slave nodes. A slave node whose black-box view differs significantly from the that of the other slave nodes indicted. We describe below how we perform the comparison in practice.

Algorithm. Our black-box algorithm consists of three parts: collection, sampling and corroboration.

In the collection part, we collect 14 metrics from */proc* and 2 TCP-related metrics from `netstat` 4.1. This is done for every slave node at a fixed time interval of 1 second. We denote the metrics collected on slave node i at time t as a 16-dimensional `sadc`-vector $m_{i,t}$. Each slave node maintains a window of the W most recently collected `sadc`- vectors. We

chose to maintain a window of $W = 120$ seconds of `sadc`-vectors on each slave node.

A naive pair-wise comparison of each slave node's `sadc`-vectors with every other slave node's `sadc`-vectors would require $O(n^2)$ comparisons. Instead, to maintain scalability, we only corroborate each slave node's black- box view with an approximate global distribution of black-box metrics. To accomplish this, we note that the `sadc`-vectors $m_{i,t}$ are samples of the global distribution of black-box metrics, and thus, a random subset of these samples is itself a sampling of the global distribution. Furthermore, the random subset of samples is not biased towards any particular slave node. Therefore, we maintain, for each slave node i, a set of W `sadc`-vectors $m_{j,t}$ collected on other nodes $j \neq i$ within the same window of W seconds. At every instant t when we collect the `sadc`-vectors, we also compute a random permutation π_t, and add the `sadc`-vector $m_{i,t}$ to the collection of `sadc`-vectors at slave node $node_{\pi_t(i)}$. We also discard the oldest sample in the collection. The use of the random permutation ensures that each node also receives exactly one new `sadc`-vector even as it discards the oldest sample, thus maintaining exactly W `sadc`-vectors collected on other nodes at all times. Further, the permutation also ensures each node only needs to send its most recently collected `sadc`-vector to exactly one other node at each instant. This would be important in an online diagnosis system to prevent overloading the network.

Thus, at each slave node i, we have two sets of `sadc`-vectors: those collected on the slave node itself, and a subset of the `sadc`-vectors collected on other slave nodes, representing the global distribution of the black-box metrics. Both sets have the same number W of `sadc`-vectors. There are a number of ways to corroborate the two sets of `sadc`-vectors. For instance, one can peform standard statistical tests like the chi-square test, or directly compute the KL-divergence or Earth-mover's distance between the two sets. However, given the high-dimensionality of the data, we chose not to use these techniques. Instead, we performed a medoidshift [23] clustering over the union of the two sets. This allows us to bin the `sadc`-vectors according to the resultant cluster to which they belong, effectively reducing the high- dimensional data to a single discrete dimension[1]. Our choice of medoidshift clustering as a method of dimensionality reduction is driven by a number of its properties as claimed in [23]: not restricted to spherical or ellipsoidal clusters, no pre-specified number of clusters, and a potential for incremental clustering. The first two properties make our diagnosis algorithm extensible to other metrics whose nature we cannot know *a priori*. Although we do not perform the clustering in an incremental fashion, the potential to do so allows us to use overlapping windows of collected metrics with low

[1]While we have chosen one particular method of reducing dimensionality, we do not exclude other techniques like Singular Value Decomposition or Multidimensional Scaling. We leave this as a choice to the user, but explain our design choice.

overhead. Having reduced the data with medoidshift clustering, we can compute a discrete distribution for each of the two sets of `sadc`-vectors. Finally, we compute the square root of the Jensen-Shannon divergence [10] between the two distributions. We chose this as our measure of distance as it has been shown to be metric [2]. An alarm for a slave node is raised whenever the computed distance of the slave node exceeds a threshold. An alarm is treated merely as a suspicion; repeated alarms are needed for indicting a node. Thus, we maintain an exponentially weighted alarm-count for each slave node. The slave node is then indicted when its exponentially weighted alarm- count exceeds a predefined value. We present the black-box algorithm for raising alarms below.

[2]A distance between two objects is "metric" if it has the properties of symmetry, triangular inequality, and non-negativity.

Algorithm 1 Black-Box Algorithm. Note: $MShift(S)$ is the medoidshift clustering that returns a labeling h for each element in S. $JSD(h_1, h_2)$ is the Jensen-Shannon divergence between two histograms h_1 and h_2

```
 1: procedure BLACK-BOX(W,threshold)
 2:     for all node i do
 3:         Initialize self_i with W sadc-vectors
 4:         Initialize other_i with W sadc-vectors
 5:     end for
 6:     while job in progress do
 7:         π_t ← random permutation
 8:         for all node i do
 9:             Collect new sadc-vector m_{i,t}
10:             Add m_{i,t} to self_i
11:             Send m_{i,t} to node π_t(i)
12:             Receive m_{j,t} where π_t(j) = i
13:             Add m_{j,t} to other_i
14:             Remove m_{i,t−W} from self_i
15:             Remove m_{j,t−W} from other_i
16:             h ← MShift(self_i ∪ other_i)
17:             dist ← sqrt(JSD(h(self_i), h(other_i)))
18:             if dist > threshold then
19:                 Raise alarm for node i
20:             end if
21:         end for
22:     end while
23: end procedure
```

Notice that since we can maintain the two sets of `sadc`-vectors at each node itself, the diagnosis can be performed in a distributed fashion. The work done by each slave node scales at a constant $O(1)$ with the number of slave nodes.

4.3.2 White-box Diagnosis

Intuition. From our log-extracted state-machine views on each node, we consider the durations of maps and reduces. For each of these states of interest, we can compute the histogram of the durations of that state on the given node. As mentioned in Section 4.1, the durations for the state on a given node is a sample of the global distribution of the durations

for that state across all nodes. The local distribution of durations is hence an estimate of the global distribution. According to our BliMeE framework, the local distribution is a limited view of the global distribution, which is a property of the MapReduce system. We corroborate each local distribution against a global distribution, indicting nodes with local distributions that are dissimilar from the global distribution as being faulty. The intuition, as described earlier, is that the tasks on each node (for a given job are multiple copies of the same code, and hence should complete in comparable durations[3].

Algorithm 2 White-Box Algorithm. Note: $JSD(distrib_i, distrib_G)$ is the Jensen-Shannon divergence between the local distributions of the states' durations at node i and the global distribution. $KDE(states_i)$ is the kernel density estimate of the distribution of durations of $states_i$.

```
1:  procedure WHITE-BOX(W,threshold)
2:      for all node i do
3:          states_i ← {s : s completed in last W sec}
4:          distrib_i ← KDE(states_i)
5:      end for
6:      distrib_G ← N^{-1} ∑ distrib_i
7:      for all node i do
8:          dist_i ← sqrt(JSD(distrib_i, distrib_G))
9:          if dist_i > threshold then
10:             raise alarm at node i
11:         end if
12:     end for
13: end procedure
```

Algorithm. First, for a given state on each node, probability density functions (PDFs) of the distributions of durations are estimated from their histograms using a kernel density estimation with a Gaussian kernel [26] to smooth the discrete boundaries in histograms. Then, an estimate of global distribution is built by summing across all local histograms. Next, the difference between these distributions from the global distribution is computed as the pair-wise distance between their estimated PDFs. We again use the square root of the Jensen-Shannon divergence as our distance measure. We repeat this analysis over each window of time. As with the black-box algorithm , we raise an alarm for a node when its distance to the global distribution exceeds a set threshold, and indict it when

[3]Note that we would also indict nodes with tasks that take longer than others due to data skews, and this diagnosis is to be interpreted as indicating an inefficient imbalance in the workload distribution of the job.

the exponentially weighted alarm-count exceeds a predefined value. The pseudo-code for raising alarms is presented above.

4.3.3 Heartbeat-based Diagnosis

Heartbeat-rate Corroboration In a Hadoop cluster, each slave node sends heartbeats to the master node at the same periodic interval across the cluster (this interval is adaptively increased across all TaskTracker nodes as cluster size increases). Hence, in the absence of faulty conditions, the same heartbeat rate (number of heartbeat messages logged per unit time) should be observed across all slave nodes (see Section 4.1). The heartbeat-rate is computed by smoothing over the discrete event series of heartbeats into a continuous time-series using a Gaussian kernel. Figure 4.1 shows the heartbeat rates of 4 slave nodes through an experiment with no fault injected. These heartbeat rates are then compared across slave nodes, by computing the difference between the rates and the median rate.

Heartbeat Propogation Delay The Heartbeat Propagation Delay is the difference between the time at which a received heartbeat is logged at the JobTracker, and at which the received acknowledgement is logged at the TaskTracker for the same heartbeat. This delay includes both the network propagation delay, and the delay caused by computation occurring in the same thread as that for handling the heartbeat at both the JobTracker and TaskTracker. This difference in timestamps, however, is subject to clock synchronization and clock drift. Differences in clock synchronization offsets the timestamps by a constant value, whereas clock drift results in a linear divergence in clock times. We can model this as:

$$\text{timestamp diff} = \alpha t + \text{ hearbeat propagation delay } + \beta$$

where t is either the timestamp of the log message at the JobTracker or at the TaskTracker, α accounts for the rate of clock drift, and β accounts for the difference in clock synchronization. As we are interested in the heartbeat propagation delay rather than the timestamp difference, we perform a local linear regression on the timestamp difference against time. (The linear regression has to done locally, as periodic synchronizations of local clocks with a central clock will reset the parameters.) If the true propagation delay is almost constant, the residuals of our local linear regression would be almost zero. On the other hand, if a heartbeat has a large residual heartbeat propagation delay, then either the heartbeat is anomalous compared to other heartbeats from the same TaskTracker, or there is a large variation in the true heartbeat propagation delay. Both cases are indicative of problems in the MapReduce system. Thus, we indict nodes for which there is a large average residual heartbeat propagation delay.

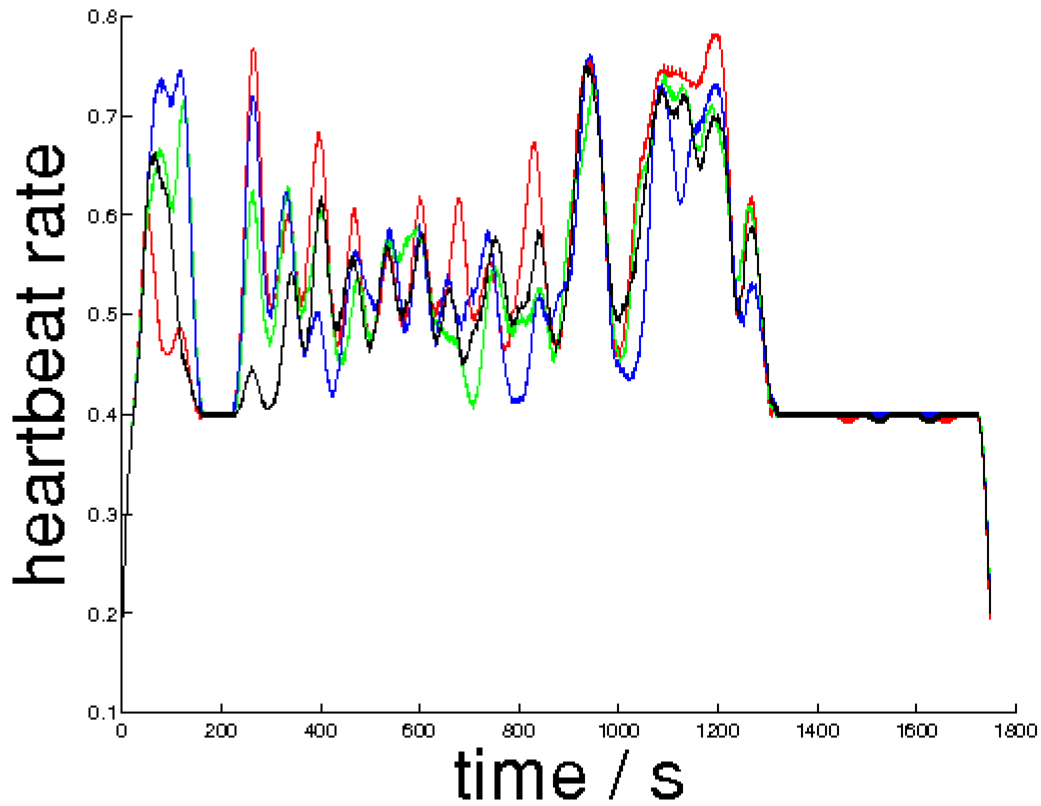

Figure 4.1: Heartbeat rates of 4 slave nodes throughout an experiment with no fault injected.

Figure 4.2 shows the residuals obtained from the local linear regression on timestamp difference against log message time. In this particular experiment, we injected hang2051, a JobTracker hang (see Table 6.1). We observe that before the fault is triggered, the residuals are mostly less than 100ms. After the fault was triggered, the residuals increased to about ± 1500.

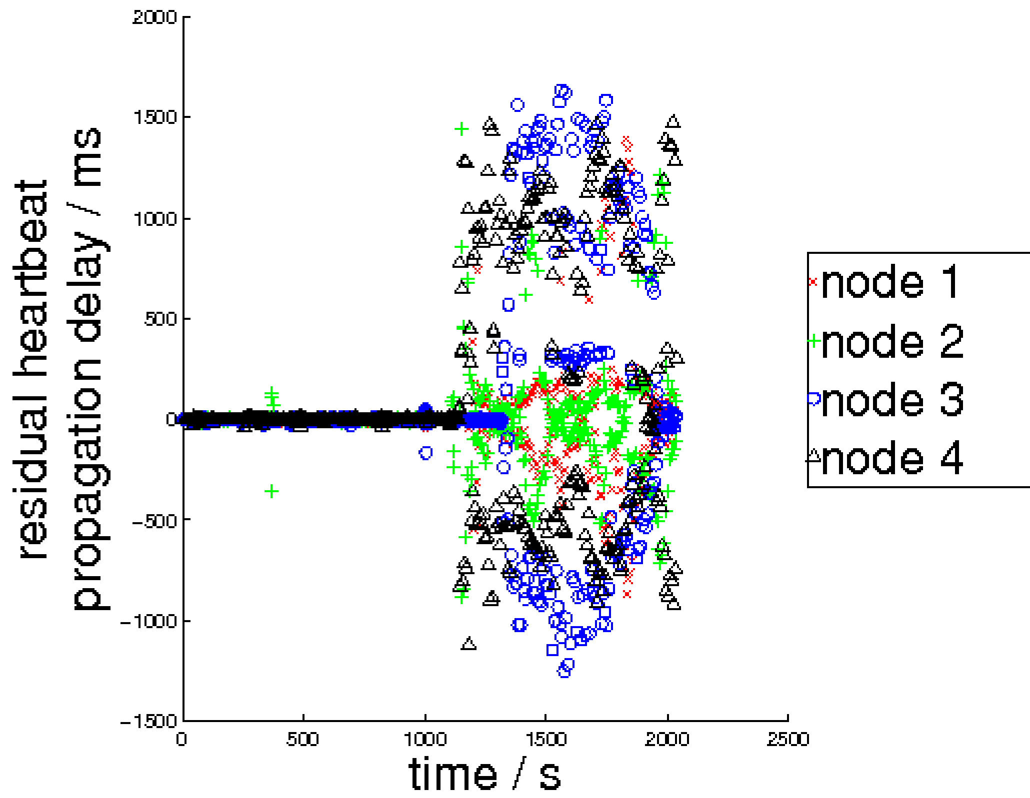

Figure 4.2: Residual heartbeat propagation delay of 4 slave nodes throughout an experiment with injected hang2051.

Chapter 5

Synthesizing Views: "Reconstructing the Elephant"

Different faults manifest differently on different metrics, resulting in different outcomes from our diagnostic algorithms. A particular fault may or may not manifest on a particular metric, and the manifestation may be correlated to varying degrees. Each of our diagnostic algorithms thus acts as a "blind-man" to give us a different perspective into the fault's effect on the MapReduce system. By synthesizing these perspectives, it is possible to identify the particular fault. More specifically, given a cluster, we would like to know, for each node, if it is faulty, and if so, which of the previously known faults it most closely resembles.

To this end, we represent each node by the diagnostic statistics that are generated by the algorithms. The diagnostic statistic for both our black-box and white-box algorithms is the exponentially weighted alarm-count, for the heartbeat rate corroboration algorithm it is the difference between the node's heartbeat rate and the median rate, and for the heartbeat propagation delay algorithm it is the sum of residuals. For each node, we construct a vector consisting of the diagnostic statistics for each algorithm, and also the average of the diagnostic statistics across all other nodes in the cluster for each algorithm. The former captures the ability of the diagnostic algorithms to indict the faulty node, whereas the latter captures the degrees to which each fault manifests in a correlated manner on the diagnostic algorithms.

Using this representation, we are able to build classifiers for the faults. In particular we chose to use decision trees as our classifiers. Note, however, that it is possible to use other types of classifiers, and while decision trees tend not to have the best prediction errors, they have the added advantage of being easily understood, and reflect the natural manner

in which human operators identify problems. Each interior node in our decision trees corresponds to the degree to which a particular diagnostic algorithm believes: (1) the node in concern is faulty, (2) other nodes in the cluster are faulty. The leaves of the decision tree then corresponds to whether the node in concern is faulty, and which previously seen fault is the most likely candidate fault to have occurred in the cluster.

Chapter 6

Evaluation and Experimentation

6.1 Testbed and Workload

We analyzed system metrics from Hadoop 0.18.3 running on 10- and 50-node clusters on Large instances on Amazon's EC2. Each node had the equivalent of 7.5 GB of RAM and two dual-core CPUs, running amd64 Debian/GNU Linux 4.0. Each experiment consisted of one run of the `GridMix` workload, a well-accepted, multi-workload Hadoop benchmark. `GridMix` models the mixture of jobs seen on a typical shared Hadoop cluster by generating random input data and submitting MapReduce jobs in a manner that mimics observed data-access patterns in actual user jobs in enterprise deployments. The `GridMix` workload has been used in the real-world to validate performance across different clusters and Hadoop versions. `GridMix` comprises 5 different job types, ranging from an interactive workload that samples a large dataset, to a large sort of uncompressed data that access an entire dataset. We scaled down the size of the dataset to 2MB of compressed data for our 10-node clusters and 200MB for our 50-nod clusters to ensure timely completion of experiments.

6.2 Injected Faults

We injected one fault on one node in each cluster to validate the ability of our algorithms at diagnosing each fault. The faults cover various classes of representative real-world Hadoop problems as reported by Hadoop users and developers in: (i) the Hadoop issue tracker [14] from October 1, 2006 to December 1, 2007, and (ii) 40 postings from the

Hadoop users' mailing list from September to November 2007. We describe our results for the injection of the seven specific faults listed in Table 6.1.

[Source] Reported Failure	[Fault Name] Fault Injected
[Hadoop users' mailing list, Sep 13 2007] CPU bottleneck resulted from running master and slave daemons on same machine	[CPUHog] Emulate a CPU-intensive task that consumes 70% CPU utilization
[Hadoop users' mailing list, Sep 26 2007] Excessive messages logged to file during startup	[DiskHog] Sequential disk workload wrote 20GB of data to filesystem
[HADOOP-2956] Degraded network connectivity between DataNodes results in long block transfer times	[PacketLoss5/50] $5\%, 50\%$ packet losses by dropping all incoming/outcoming packets with probabilities of $0.01, 0.05, 0.5$
[HADOOP-1036] Hang at TaskTracker due to an unhandled exception from a task terminating unexpectedly. The offending TaskTracker sends heartbeats although the task has terminated.	[HANG-1036] Revert to older version and trigger bug by throwing NullPointerException
[HADOOP-1152] Reduces at TaskTrackers hang due to a race condition when a file is deleted between a rename and an attempt to call getLength() on it.	[HANG-1152] Simulated the race by flagging a renamed file as being flushed to disk and throwing exceptions in the filesystem code
[HADOOP-2080] Reduces at TaskTrackers hang due to a miscalculated checksum.	[HANG-2080] Simulated by miscomputing checksum to trigger a hang at reducer
[HADOOP-2051] Hang at JobTracker due to an unhandled exception while processing completed tasks.	[HANG-2051] Revert to older version and trigger bug by throwing NullPointerException

Table 6.1: Injected faults, and the reported failures that they simulate. HADOOP-xxxx represents a Hadoop bug database entry.

Chapter 7

Results

We first present the results for the diagnostic algorithms that corroborate views from instrumentation points , and then present the decision tree generated by synthesizing the outcomes of the diagnostic algorithms.

7.1 Diagnostic algorithms

Since our diagnostic algorithms only generate diagnostic results for slave nodes, we will present true positive and false positive rates for faults on slave nodes, and alarm rates for faults on master nodes, with true positive, false positive and alarm rates defined below.

7.1.1 Slave node faults

We evaluated our diagnostic algorithms' performance at detecting faults by using true positive and false positive rates [12] across all runs for each fault injected on a slave node, and for clusters of sizes of 10 and 50 slave nodes. A slave node with an injected fault that is correctly indicted is a true positive, while a slave node without an injected fault that is incorrectly indicted is a false positive. Thus, the true positive (TP) and false positive (FP) rates are computed as:

$$TP = \frac{\text{\# faulty nodes correctly indicted}}{\text{\# nodes with injected faults}}$$

$$FP = \frac{\text{\# nodes without faults incorrectly indicted}}{\text{\# nodes without injected faults}}$$

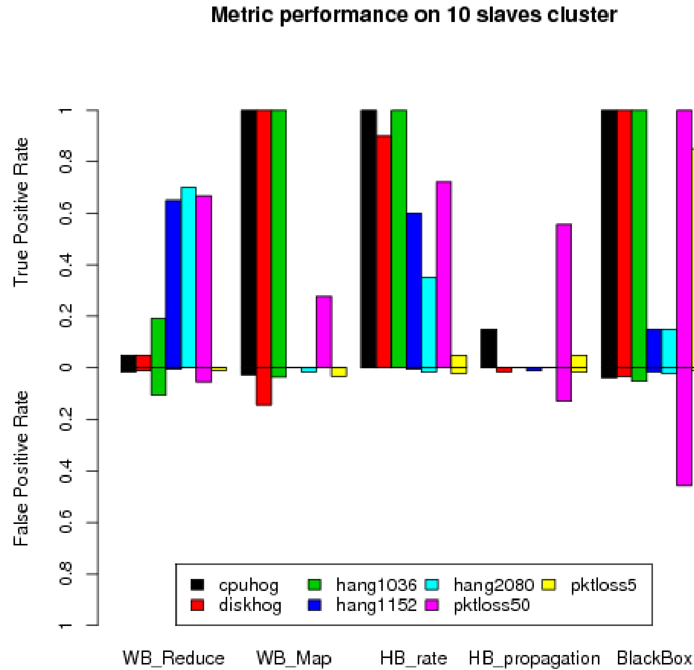

Figure 7.1: True positive and false positive rates for faults on slave nodes, on 10 slaves cluster.

Figures 7.1 and 7.2 show the TP and FP rates of the algorithms for a 10 and 50 slave node cluster respectively.The bars above the zero line represent the TP rates, and the bars below the zero line respresent the FP rates for each fault. Each group of 12 bars (6 above, 6 below zero line) show the TP and FP rates for a particular algorithm and instrumentation source. "WB_Reduce" and "WB_Map" are the diagnostic algorithms that corroborate Reudce and Map tasks' durations across the slave nodes, respectively. "HB_rate" and "HB_propagation" refer to the two hearbeat-based diagnostic algorithms that corroborate heartbeat rates across TaskTrackers, and heartbeat propagation delay between TaskTrackers and JobTrackers. The "BlackBox" algorithm, as previously described, corroborates OS-level performance counters across physical slave nodes.

From Fig 7.1 and 7.2, we observe that every fault is detected (with TP > 0.65) by at least one algorithm. In the case of resource- related faults (cpuhog, diskhog, pktloss5, pktloss5), our black-box algorithm has high TP rates of at least 0.83. BlackBox is also able to detect hang1036, but not hang1152 and hang2080. This is because hang1036 is

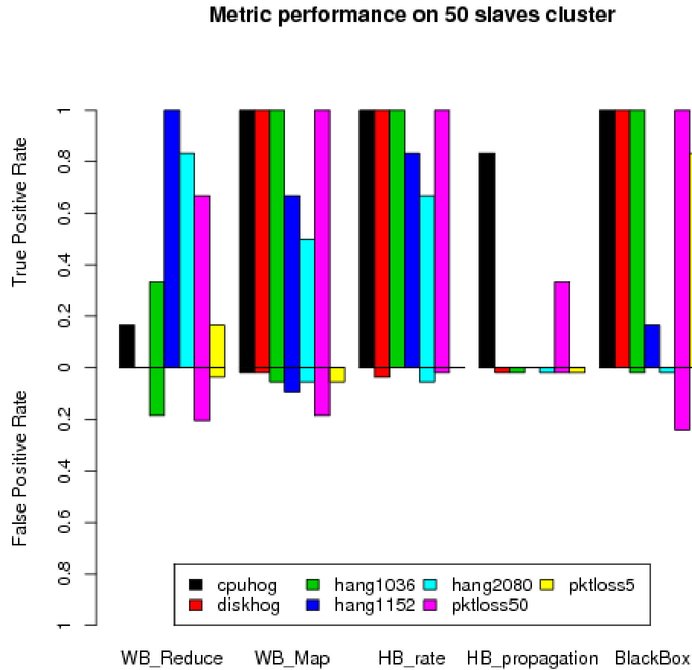

Figure 7.2: True positive and false positive rates for faults on slave nodes, on 50 slaves cluster.

a hang in the Map task, and results in a idle period where the Map tasks have hung and the Reduce tasks block on waiting for output from the Map tasks. On the other hand, for hang1152 and hang2080 in the Reduce tasks, the slave node is able to continue consuming resources for execution of Map tasks, masking the hangs from the black-box point of view. Not suprisingly, the white-box algorithm WB_Map based on Map tasks' durations capture the hang (hang1036) in the Map phase, and the algorithm WB_Reduce based on Reduce tasks' durations capture both hangs (hang1152 and hang 2080) in the Reduce phase. The algorithm HB_rate detects most faults, except pktloss5, and the algorithm HB_propagation is most effective at detecting resource-related faults. Since heartbeat rate is a reflection of workload, we expect HB_rate to detect any fault that may adversely affect workload. On the other hand, HB_propagation targets a specific operation in the application: the sending of a heartbeat response from the JobTracker to the TaskTracker. The application hangs do not adversely affect this operation and are thus not detected, whereas the resource-related faults affect almost all operations in the system are are thus detected by HB_propagation.

We notice that pktloss5 is not sufficient severe and can be eventually overcome by TCP's retransmissions. Thus, it fails to be detected by almost all algorithms (except BlackBox which explicitly tracks TCP-related metrics, and HB_Propagation, which targets a network-dependent operation).On the other hand, pktloss50 is sufficient severe that it affects the slave node's abiltiy to communicate and operate normally. All our algorithms detect, to varying TP rates, pktloss50. The severeness of pktloss50 also affect other slave nodes that block on reading or sending data to the faulty slave node, explaining the generally higher FP rates for pktloss50.

In addition, we also observe that different faults have different TP and FP rates across different algorithms. This supports our hypothesis that different faults manifest differently on different metrics. We will exploit this variation in a later section to synthesize the diagnostic algorithms' outputs to generate a decision tree.

7.1.2 Master node faults

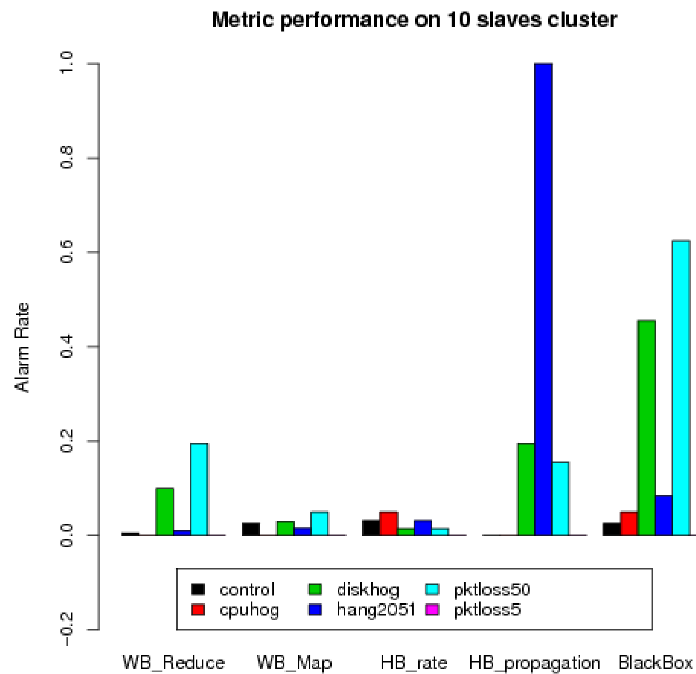

Figure 7.3: Alarm rates for faults on master nodes, on 10 slaves cluster.

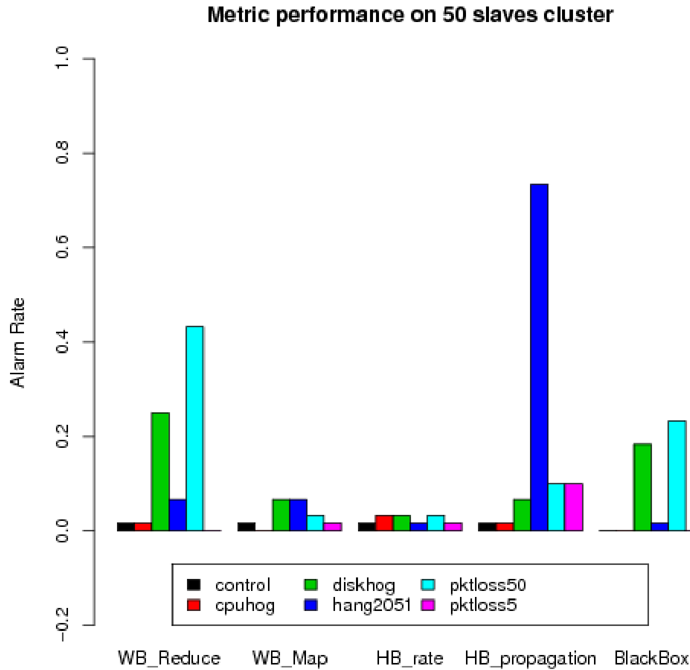

Figure 7.4: Alarm rates for faults on master nodes, on 50 slaves cluster.

As our diagnostic algorithms never explicitly indict the master node, it would be meaningless to discuss TP and FP rates for master node failures. Instead, we compute the alarm rate, that is, the proportion of slave nodes that were indicted by the algorithm:

$$alarm = \frac{\text{\# of indicted slave nodes}}{\text{\# of slave nodes}}$$

Figure 7.3 and 7.4 show the alarm rates for the 10 slave cluster and the 50 slave cluster respectively. Note that the fault "control" is not actually a fault; it refers to control experiments where we did *not* inject *any fault* into the system. We also used the control set to determine our thresholds. Specifically, we set the thresholds for each algorithm such that the alarm rates in the control sets would be 3% or less.

In the case of master node faults, the alarm rates only serve to give a notion of the effect of the master node fault on the slave nodes. An alarm rate significantly higher than 3% would indicate that the master node fault has a significant effect on the slave nodes. Note, however, that the diagnostic algorithms only indict slave nodes. As such, none of

the algorithms localize the fault correctly, much less identify it. We fix this problem using the decision tree classification, as shown in the following section.

Nevertheless, we observe that the alarm rates vary between algorithms and faults. In particular, the alarm rate for hang2051 for HB_propagation is 1.0 on the 10-slave cluster, and 0.73 on the 50-slave cluster. This is because hang2051 is a master node hang, and HB_propagation is our only algorithm that explicitly accounts for the master node. All other algorithms corroborate views from multiple slave nodes. This demonstrates the usefulness of multiple types of corroboration.

7.2 Synthesizing outcomes of diagnostic algorithms

We generated a decision tree by using the `rpart` package of the statistical software `R`. The decision tree generated is shown in Fig 7.5.

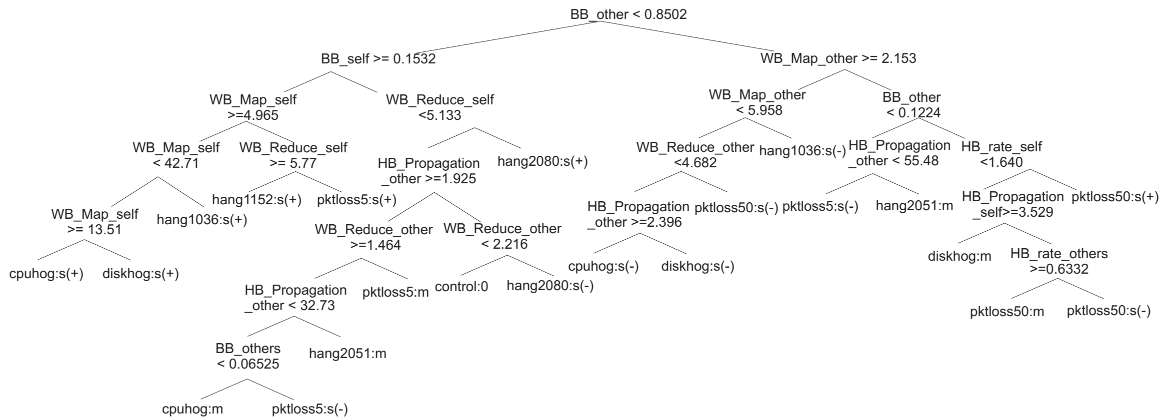

Figure 7.5: Decision tree, classifies faults by the outcomes of the diagnostic algorithms.

The interior nodes (and the root) of the decision tree is labeled with an inequality of the form $X < t$ or $X >= t$, where X is a component of the representation (see Section 5) and t is a threshold. X is of the form *algorithm_location*, where *algorithm* can be any of BB, WB_Map, WB_Reduce, HB_rate and HB_propagation (representing our black-box algorithm, white-box algorithm corroborating Map durations, white-box algorithm corroborating Reduce durations, heartbeat-based algorithm corroborating heartbeat rates, and heartbeat-based algorithm corroborating heartbeat propagation delays). *location* can be either self or other, with the former representing the diagnostic statistic of the algorithm

for the node in concern, and the latter representing the mean of the diagnostic statistics of other nodes that were indicted by the algorithm.

Labels on the leaves have the form of *fault:suffix*, where *fault* indicates the most likely fault that occurred in the system; and a suffix of m indicates the fault occurred at the master node, s(+) indicates that the fault occurred on the slave node in concern, and s(-) indicates that the fault occurred on some slave node, but not on the node in concern.

Using the decision tree to classify the fault on a node would involve traversing the tree from root to leaf, following the left branch whenever the inequality at an interior node (or the root) evaluates to true, and the right branch otherwise. The labels at the leaves indicate whether the node in concern was faulty, and the fault that was most likely, among the known faults, to have occurred in the system.

For example, if there are no faults in the system, then all our algorithms would likely generate low diagnostic statistics for every node in the cluster. For any individual node, its diagnostic statistics would be small in value, as would the average diagnostic statistics of all other nodes. In other words, the value of *algorithm_location* is small for every *algorithm*, for both *location*=*self* and *other*. To classify this node, we would evaluate *BB_other<0.08502* to true, following the left branch; *BB_self>=0.1532* to false, following the right branch; *WB_Reduce_self<5.133* to true, following the left branch; *HB_propagation_other>=1.925* to false, following the right branch; *WB_Reduce_other <2.216* to true, following the left branch; and correctly conclude that the node should be labeled *control*, i.e. it is fault-free, and there are no faults elsewhere in the cluster.

The above example shows how decision trees lend themselves to interpretation: when none of the algorithms detect any problems in the cluster, there is likely to be no faults in the cluster. We also observe that labels of leaves in the right-subtree correspond to faults on other nodes in the cluster, or to faults that are more correlated. In both cases, the BlackBox algorithm is likely to produce large diagnostic statistics for for other nodes in the system, and to indict those other nodes.

In addition to merely visualizing the decision tree, we also evaluated the ability of our classification technique by using a *N*-fold cross- validation method. We randomly partitioned our experiments into *N* subsets, and classified the data in each subset using a decision tree trained on the remaining $N-1$ subsets. We chose $N = 296$ for our evaluation, as we had 296 experiments.

Fig 7.6 shows a confusion matrix of our classification results. Each row reprsents an actual class of faults, and each column represents the classification given by the decision tree. A cell in row i, column j would hold the proportion of nodes that were actually of class i, and were given the classification j by the decision tree. Lighter (less red) shades

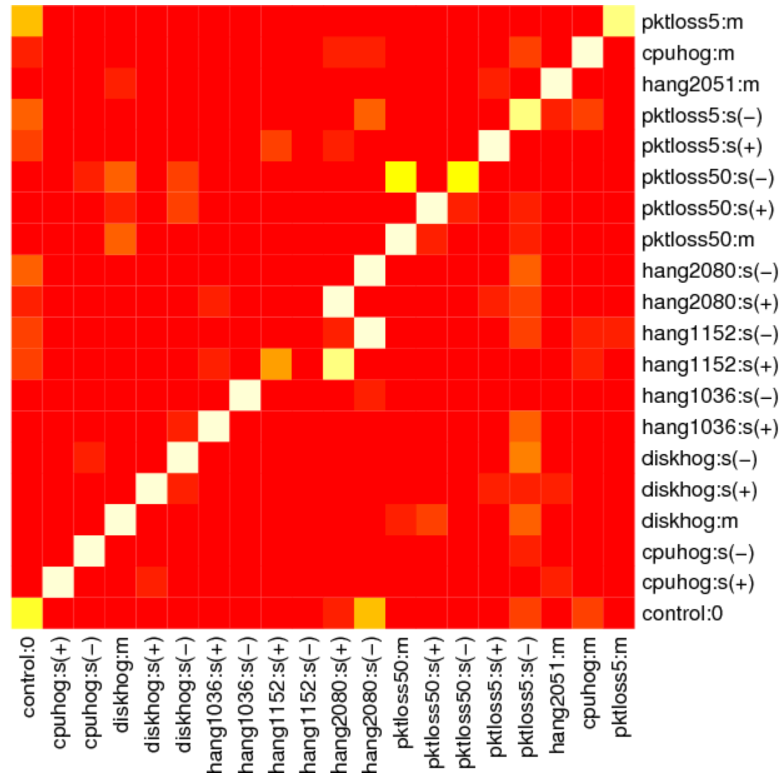

Figure 7.6: Confusion matrix of fault classification. Each row reprsents a class of faults, and each column represents the classification given. Lighter shades indicate the actual fault was likely to be given a particular classification.

correspond to higher values. A perfect confusion matrix would have a lightly shaded diagonal and dark shades at all other non-diagonal cells.

Our confusion matrix shows that for most of the classes, we are able to achieve high classification accuracy, with some notable exceptions: hang1152 is confused as hang2080, pktloss50:s(-) as pktloss50:m, control as hang2080:s(-), and pktloss5:m as control. In the first case, we note that hang1152 and hang2080 are both application hangs in the Reduce task. Hence, the two faults have similar manifestations and thus syntheszing the diagnostic algorithms' outcome does not aid differentiation between the two faults. In the second case, pktloss50:s(-) and pktloss50:m are classifications of network faults that have occurred in the system. We note that in these two cases, the wrong classification still provides valuable assistance to the user or sysadmin. Classification of hang1152 as hang2080 directs the user to look at the Reduce tasks. Since we do not confuse hang1152:s(-) for hang2080:s(+) nor hang1152:s(+) for hang2080:s(-), the user is able to localize the fault to

the faulty node. Classifying pktloss50:s(-) as pktloss50:m indicates to the sysadmin that a severe network loss is occurring in the system, and furthermore that it is not at the node for which we have wrongly classified the fault.

In the case where control is confused as hang2080:s(-), our diagnosis system has a false positive. But if the system is truly fault-free, it would be unlikely that any slave node is classified as hang2080:s(+); conversely if the system was indeed suffering from hang2080, then with high likelihood the faulty node would be classified as hang2080:s(+). Thus, the absence of any node that is classified as hang2080:s(+) gives us confidence that the system is in fact fault-free. Finally in the case where pktloss5:m is confused as control, we reiterate that pktloss5 is a less severe fault from which TCP is able to recover. As the network utilization by the master node is low, pktloss5 on master has minimal effect on the MapReduce system's operation.

		Localized to			
Fault	Not detected	Master node	Node in concern	Other node	Correct localization
control:0	0.43	0.1	0.08	0.39	0.43
cpuhog:m	0.04	0.71	0.06	0.19	0.71
diskhog:m	0	0.65	0.13	0.22	0.65
pktloss50:m	0	0.79	0.1	0.11	0.79
pktloss5:m	0.36	0.64	0	0	0.64
hang2051:m	0	0.9	0.08	0.02	0.9
cpuhog:s(+)	0	0.04	0.96	0	0.96
cpuhog:s(-)	0	0	0	1	1
diskhog:s(+)	0	0.04	0.88	0.08	0.88
diskhog:s(-)	0	0	0	0.99	0.99
pktloss50:s(+)	0	0.08	0.67	0.25	0.67
pktloss50:s(-)	0	0.48	0.01	0.5	0.5
pktloss5:s(+)	0.12	0	0.88	0	0.88
pktloss5:s(-)	0.14	0.18	0.01	0.67	0.67
hang1036:s(+)	0	0	0.74	0.26	0.74
hang1036:s(-)	0	0	0.01	0.99	0.99
hang1152:s(+)	0.12	0.12	0.73	0.04	0.73
hang1152:s(-)	0.14	0.13	0.07	0.66	0.66
hang2080:s(+)	0.08	0	0.81	0.12	0.81
hang2080:s(-)	0.18	0.04	0.04	0.74	0.74

Table 7.1: Fault localization for various faults

In general, we are able to localize the fault to the correct node, even if the identity of the fault is confused. To investigate the ability of our technique in localizing the fault, we considered, for each fault class, the proportion of time when the fault was localized to the master node, the node in concern, or other nodes in the system, or completely not detected

(i.e. classified as control). Table 7.1 shows these results. In particular, the final column of Table 7.1 shows that we are able to correctly localize most fault classes.

Given the large number of classes, we believe that our decision tree has produced reasonable classification errors. We also suspect that other classifiers may produce lower classification errors, but few offer the interpretability of decision trees.

Chapter 8

Discussions

In this section, we discuss some of the concerns that readers may have with our approach.

8.1 Data skew and unbalanced workload

Three of the four algorithms presented in this paper have an underlying assumption that Map and Reduce tasks are sufficiently similar for comparison. However, while the tasks may execute the same code, they operate on different portions of the input data. One may contend that the assumption is easily broken in the case of a data skew or a heavily data-dependent load. We argue, though, that in the case of data skews and increased data-dependent load, if the job has maps or reduces that take on extra load, then the job can be optimized by spreading the load more evenly, assuming homogeneous node capabilities. Thus, the alarms raised and indictments made by our algorithms do in fact indicate a performance problem and a potential for optimizing the distribution of load.

8.2 Heterogenous hardware

Another of our assumptions is that the hardware is homogeneous in the MapReduce cluster. While this may appear overly restrictive, we argue that it is not unrealistic. For instance, clusters operated on virtualized services like EC2 can be easily configured to have the same (virtualized) hardware. Nodes in physical clusters are often upgraded and replaced in batch. Furthermore, we do not insist that the entire MapReduce cluster be homogeneous. In the case where subsets of nodes have homogeneous hardware, our algorithms

can be trivially adapted to work with the islands of homogeneous nodes.

8.3 Scalability of sampling for black-box algorithm

In order to attain scalability of computation with our black-box algorithm, we have resorted to sampling a constant number (one) of the other slave nodes at each instant of time. The astute reader may have realized that this may have come at a price of lower accuracy as the sample becomes less representative with larger number of slave nodes. We do not deny that this may indeed be the case. However, we suspect that a sampling rate greater than $O(1)$ but less than $O(n)$ may suffice, since one never has to poll too many people to find out what the nation thinks. A more rigorous and formal statistical analysis, however, is beyond the scope of our current work.

8.4 Raw data versus secondary diagnostic outcomes

In generating the decision tree, we consumed the secondary diagnostic statistics generated by our algorithms as the representation for the nodes. An alternative would be to simply use the raw data (black-box metrics, white-box state durations, heartbeat rates and residual propagation delays) directly. This may not be feasible simply because of the large amount and high-dimensionality of the raw data. More importantly, it is often unclear what constitutes a fault from the raw data alone. For instance, an idle period from the black-box point-of-view may indicate a fault at a node if all other slave nodes are experiencing heavy workload, but the same idleness is legitimate behavior when the cluster is not processing any MapReduce jobs. Our diagnosis algorithms capture the notion of normal versus abnormal behavior within the BliMeE framework. The secondary diagnostic statistics is thus semantically meaningful. One of the purposes of generating a decision tree is to help the user, researcher or sysadmin understand the fault pathology better. The use of semantically meaningful data achieves this aim; using raw data might not.

Chapter 9

Related Work

Recent work on diagnosing failures in distributed systems have focused on multi-tier Internet service systems which process large numbers of short-lived requests [1, 18, 7, 5]. A few key features distinguish these systems from MapReduce, which call for different techniques for diagnosing MapReduce systems. We highlight these differences next, and describe current work in diagnosing MapReduce systems, as well as work in instrumentation tools that can be added to the BliMeE framework.

9.1 Diagnosing Failures and Performance Problems

[7] is the most similar to our work; Cohen *et al.* build signatures of the state of a running system by summarizing system metrics. These signatures are similar to our characterization of known performance problems, however our characterizations are based on the intermediate outputs of component diagnosis algorithms, and they serve to synthesize algorithms, while the signatures in [7] are built directly on observed system metrics. [9] built signatures of standalone applications using detailed system call information, and would need to be augmented with significant network tracing mechanisms and causal correlation across nodes to work with a distributed system such as MapReduce. [18, 1] use path-based techniques to diagnose failures; [18] detects anomalously shaped paths (e.g. missing or additional elements) while [1] focused on accurately extracting causal paths. Both techniques were demonstrated with multi-tier Internet service systems, where paths for different requests can take on different shapes, so that path shape differences can highlight problems. However, shapes of processing paths in MapReduce are generally homogeneous, due to the parallel nature of MapReduce applications. As a result, traditional path-based techniques

which highlight anomalously-shaped paths will not be effective at diagnosing performance problems in MapReduce. [22] enables programmers to specify "expectations", a form of distributed assertions which it then propagates and tracks through the system to identify when these are violated; [22] is similar to our work in that they, too, corroborate multiple information sources, in corroborating programmer-specified program behavior with actual program behavior to detect problems; however, they require programmers to specify *a priori* expectations of program behavior, while all of the points of our corroboration are actual observed data from the system, increasing the robustness of our diagnosis. [6] presented statistical methods to investigate and understand the performance of multi-tier Internet services (the LAMP, or Linux, Apache, MySQL, PHP stack), and is similar to our use of statistical methods; however, we have used our statistical insights to design algorithms and shown them to catch various classes of problems in MapReduce systems.

9.2 Diagnosing MapReduce Systems

Current techniques for diagnosing problems in MapReduce systems have examined only individual sources of instrumentation, while we have taken a holistic approach to utilize multiple information sources. [19] examined detailed causal network trace data generated using custom instrumentation and identified simple faults such as a slow disk by examining latencies along processing paths; these are similar to our white-box instrumentation, although our approach does not require the invasive instrumentation that [19] used. [27] considered Hadoop's logs as well, although they focused on only DataNode logs and only considered error events as opposed to the processing events which we considered. Consequently, we have been able to diagnose a larger range of faults than either work. [24, 21] constitute our prior work. In particular, [24] provides the component white-box algorithm for our overarching BliMeE framework. The black-box approach presented in this paper differs from [21] in that we have abandoned the training phase in favor of clustering at every window of time.

9.3 Instrumentation Tools

Magpie [4] enables causal tracing of request flows with attributed resource usage for each request, by using minimal programmer-specified program structure and inserted instrumentation. Unlike Magpie, our approach does not require any modification to the target application, namely, Hadoop. Also, Magpie demonstrated anomaly detection by clus-

tering request paths; however, such clustering techniques would not be able to identify emergent behavior that is anomalous, for instance, MapReduce systems regularly execute new MapReduce programs which have user maps and reduces which are essentially arbitrary programs, and Magpie would require sufficient amounts of training data while our techniques do not as we corroborate, dynamically, multiple viewpoints of the system's execution behavior.

[13] provides causal network request tracing via inserted instrumentation into the various communications layers; this represents an additional data source which can be included in our BliMeE framework to enhance the robustness and fault coverage of our diagnosis.

Chapter 10

Conclusions

10.1 Conclusion

We have presented the "Blind Men and Elephant" framework, and described how this approach is useful for fault diagnosis in a large parallel, distributed system like MapReduce. In particular, we have presented black-box, white- box, and heartbeat-based diagnostic algorithms within the BliMeE framework, and demonstrated that by corroborating multiple instrumentation points of the system, one can identify suspect slave nodes. Further, in a repeated application of the BliMeE approach, we show that the synthesis of the diagnostic algorithms' outcomes can aid the identify of the fault and localize the fault to the correct master or slave node.

10.2 Future work

An ongoing research aims to move the techniques presented in this thssis online. This requires us to provide real-time tools to debug a live system, and this will be done using the ASDF framework [3]. It also requires that we can run our techniques in an incremental fashion. This can be easily done for the diagnostic algorithms by using finite windows or exponential weights.

To further increase the value of the tool to sysadmins, we need to present visualizations of the raw instrumentation data as well as the output from our algorithms, which respectively represent the primary and secondary viewpoints of Hadoop's behavior. Oftentimes it is easily to understand a visual, rather than textual, representation.

We are also looking to increase our coverage of instrumentation sources. We would like to incoperate X-trace or other path-based instrumentation. Corroboration of different instrumentation sources could possibly lead to other insights and algorithms in the BliMeE framework.

The current synthesis of diagnostic outcomes is performed in a supervised learning setting. In a production environment, however, not all faults are known *a priori*. Ideally we would want the BliMeE framework to automatically discover new faults as they happen. We are exploring the use of unsupervised, semi-supervised and human-aided learning for this purpose.

Finally, we would like to improve the granularity of our diagnosis. Specifically, we want to perform root-cause and code-level analysis. The obvious value of doing so is to further assist the sysadmin and programmer in understanding, debugging and fixing problems by pinpointing the problem.

Bibliography

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed system of black boxes. In *ACM Symposium on Operating Systems Principles*, pages 74–89, Bolton Landing, NY, Oct 2003. 9, 9.1

[2] Amazon.com. Amazon web services launches amazon elastic mapreduce - a web service for processing vast amounts of data, Apr 2009. `http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=1272550&highlight=`. 1

[3] K. Bare, M. Kasick, S. Kavulya, E. Marinelli, X. Pan, J. Tan, R. Gandhi, and P. Narasimhan. ASDF: Automated online fingerpointing for Hadoop. Technical Report CMU-PDL-08-104, Carnegie Mellon University PDL, May 2008. 10.2

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec 2004. 9.3

[5] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing*, pages 36–43, New York, NY, May 2004. 9

[6] M. Chiarini and A. Couch. Dynamic dependencies and performance improvement. In *LISA'08: Proceedings of the 22nd conference on Large installation system administration conference*, pages 9–21, Berkeley, CA, USA, 2008. USENIX Association. 9.1

[7] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *ACM Symposium on Operating Systems Principles*, pages 105–118, Brighton, United Kingdom, Oct 2005. 9, 9.1

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, San Francisco, CA, Dec 2004. 1, 3.1

[9] X. Ding, H. Huang, Y. Ruan, A. Shaikh, and X. Zhang. Automatic software fault diagnosis by exploiting application signatures. In *LISA'08: Proceedings of the 22nd conference on Large installation system administration conference*, pages 23–39, Berkeley, CA, USA, 2008. USENIX Association. 9.1

[10] D. M. Endres and J. E. Schindelin. A new metric for probability distributions. *Information Theory, IEEE Transactions on*, 49(7):1858–1860, 2003. 4.3.1

[11] Facebook. Engineering @ facebook's notes: Hadoop, Jun 2008. `http://www.facebook.com/note.php?note_id=16121578919`. 1

[12] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27:861–874, 2006. 7.1.1

[13] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *USENIX Symposium on Networked Systems Design and Implementation*, Cambridge, MA, Apr 2007. 9.3

[14] The Apache Software Foundation. Apache's JIRA issue tracker, 2006. `https://issues.apache.org/jira`. 6.2

[15] The Apache Software Foundation. Hadoop, 2007. `http://hadoop.apache.org/core`. 1, 3.1

[16] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, pages 29 – 43, Lake George, NY, Oct 2003. 3.1

[17] S. Godard. SYSSTAT, 2008. `http://pagesperso-orange.fr/sebastien.godard`. 4.2

[18] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Trans. on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, 16(5):1027– 1041, Sep 2005. 9, 9.1

[19] A. Konwinski, M. Zaharia, R. Katz, and I. Stoica. X-tracing Hadoop. *Hadoop Summit*, Mar 2008. 9.2

[20] Yahoo! Developer Network. Yahoo! launches world's largest hadoop production application (hadoop and distributed computing at yahoo!), Feb 2008. `http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html`. 1

[21] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Black-Box Diagnosis of MapReduce Systems. Technical report, Carnegie Mellon University PDL, CMU-PDL-08-112 (Also Under Submission to Workshop on Hot Topics in Measurement and Modeling of Computer Systems), September 2008. 9.2

[22] P. Reynolds, C. Killian, J. Wiener, J. Mogul, M. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, May 2006. 9.1

[23] Y. A. Sheikh, E. Khan, and T. Kanade. Mode-seeking by medoidshifts. In *Eleventh IEEE International Conference on Computer Vision (ICCV 2007)*, October 2007. 4.3.1

[24] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Salsa: Analyzing logs as state machines. In *Workshop on Analysis of System Logs*, San Diego, CA, Dec 2008. (document), 4.2, 9.2

[25] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop. In *(Under Submission to Workshop on Hot Topics in Cloud Computing)*, May 2009. 4.2

[26] L. Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 1st edition, Sep 2004. 4.3.2

[27] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Mining console logs for large-scale system problem detection. In *Workshop on Tackling Systems Problems using Machine Learning*, Dec 2008. 9.2