Automated Diagnosis of Chronic Performance Problems in Production Systems

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Soila P. Kavulya

B.S., Computer Science, University of Nairobi
M.S., Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May 2013

To my parents, Fred and Anne Pertet, and my husband, Geoffrey Kavulya,
who supported me each step of the way.
To my teachers, who broadened my horizons. Asanteni sana.

Abstract

Large production systems are susceptible to chronic performance problems where the system still works, but with degraded performance. Chronic performance problems occur intermittently or affect a subset of end-users. Traditional approaches for diagnosis typically rely on a *bottom-up* approach that localizes problems by correlating low-level alarms (such as resource utilization indicators or network packet loss) across components in a production system. However, these alarm-correlation approaches fall short when diagnosing chronics because they fail to provide the necessary application-level visibility to detect chronics effectively. Due to the scale and complexity of production systems, there can be multiple unresolved chronics at any given time—their symptoms often overlap with each other, and they are sometimes triggered by complex corner cases.

This dissertation presents a *top-down* diagnostic framework for diagnosing chronic performance problems in production systems. The framework comprises of four components. First, an extensible log-analysis framework that extracts end-to-end causal flows using common white-box (*i.e.*, application) logs in the production system; these end-to-end flows capture the user's experience with the system. Second, anomaly-detection tools exploit heuristics and a peer-comparison approach to label each end-to-end flow as successful or failed. Third, a *top-down* statistical diagnostic tool combines white-box metrics with black-box metrics (*e.g.*, CPU usage) to localize the source of the problem by identifying attributes that are more correlated with failed flows than successful ones. Fourth, a visualization tool that uses peer-comparison to highlight anomalous nodes in a parallel-computing cluster.

The diagnostic framework has been used to localize real incidents at an academic cloud-computing cluster that runs the Hadoop parallel-processing framework, and a production Voice-over-IP system at a major Internet Services Provider. Our approach is not limited to these two systems and is applicable to systems such as Internet Services that serve users via independent interactions.

Kidole kimoja hakivunji chawa.

One finger cannot kill a louse.

(Teamwork is necessary when solving problems.)

Swahili Proverb

Acknowledgments

I stumbled on the field of computer science quite by accident when a team of instructors from Strathmore College in Kenya visited my high school and gave a fascinating talk on the subject. Prior to that, I had dreamed of becoming a zoologist because of fond memories of going on safari with my family. Since that first introduction to computer science, numerous people and experiences have shaped my academic career. This thesis would not have been possible without their support. While it would be impossible to provide a complete list of all the people who have influenced this dissertation, what follows is my best effort to express my sincere gratitude to those who have shaped this research.

First and foremost, I would like to express my gratitude to my parents, Fred and Anne Pertet, who have always supported me and sacrificed a lot to ensure that I got the best education. My loving husband, Geoffrey Kavulya, who has supported me every step of the way. My dear brothers, Kasaine and Emmanuel Ole Pertet, and my extended family, Lucy Pertet, Rosemary, Bernice and Ken Kavulya for their love and support.

My dissertation advisor, Priya Narasimhan, was instrumental in developing this thesis. Throughout my career as a graduate student, her constant support, enthusiasm for research, and depth of knowledge have been invaluable in shaping both my professional and personal life. Priya introduced me to fault-tolerant middleware systems, and wholeheartedly supported my decision to change my research topic to focus on problem diagnosis in distributed systems. She facilitated collaborations with industry partners so that I could better understand the challenges they faced during problem diagnosis. These collaborations helped me discover interesting problems and develop practical solutions to advance the state-of-the-art. My writing and presentation skills have also greatly improved under her guidance.

My committee members, Gregory R. Ganger, Christos Faloutsos, and Matti Hiltunen, provided me with invaluable feedback on scoping my thesis. Their careful reading and

thoughtful comments greatly helped to improve the overall quality of my thesis. Philip Koopman and Raj Rajkumar taught me about distributed systems, and were always available to provide sound advice. Roy Maxion taught me everything I know about experimental methods. My undergraduate advisors at the University of Nairobi, Peter W. Wagacha and Katherine Getao, introduced me to machine-learning and supervised my first research project.

My work on understanding complex systems would not have been possible without access to production systems, especially to the instrumentation data that they generate. I am indebted to Matti Hiltunen, Kaustubh Joshi, and Scott Daniels from AT&T Labs Inc., who collaborated with me to develop the algorithm for diagnosing chronic performance problems in large distributed systems. They provided me with access to logs from a production VoIP system that handled 10s of millions of calls each day, and championed the adoption of our diagnosis tool into the daily workflow of the operations team at the production system. I thank the system administrators of the Yahoo! M45 cluster and the OpenCloud computing cluster for providing me with access to Hadoop logs. I would especially like to thank Mitch Franzos, Michael Stroucken, Zisimos Economou, and Kai Ren for all the help they provided on the OpenCloud cluster.

I have been fortunate to collaborate with other brilliant researchers during my PhD. Jiaqi Tan, Xinghao Pan, Michael P. Kasick, Elmer Garduno, Eugene Marinelli, Nathan Mickulicz, and Rajeev Gandhi were instrumental in developing our peer-comparison approach for problem diagnosis and visualization. Arun Ganesany, Ben Gotow, James Mulhollandy, Sriram Ramasubramaniany, Mark Shuster, and Jason Campbell helped with the Hadoop user study. Raja Sambasivan, Ilari Shafer, Tudor A. Dumitras, Joseph G. Slember, and Rolando Martins were always available to participate in interesting discussions, and to provide feedback on my research.

My first internship at General Motors Research Lab taught me about developing dependable drive-by-wire systems. I would like to thank Thomas E. Furhman, Alan Baum, Sanjeev M. Naik, and Pradyumna K. Mishra from General Motors for giving me the opportunity to work on a fascinating research problem, and for my first joint patent with Sanjeev and P. K. I also thank Patrick E. Lanigan, Kunal Mankodiya and Utsav Drolia for our joint work on problem diagnosis in autonomous vehicles. My second internship at HP Labs with John Wilkes and Jay Wylie introduced me to service oriented architectures, which rely on

economic pressures to drive automated management decisions. John Wilkes also provided advise on my thesis topic.

Lynn Philibin, Elaine Lawrence, Karen Lindenfelser, Joan Digney, Samantha Goldstein, and Reenie Kirby provided the administrative support that ensured that my graduate studies went smoothly. They treated me with warmth and kindness, and helped me navigate through my studies at CMU.

My close friends, Alice Muiruri, Maggie Johnson, Katherine Gacharia, Rachel Syombua, and Mwiyeria Mucuha who have kept me grounded. The Kenyan community in Pittsburgh, particularly Isaac Kivuva and Catherine Mutunga, for providing me with a home away from home. Gary Denning for his advice and constant encouragment.

During my dissertation research, I received financial support from the NSF CAREER Award CCR-0238381, the DARPA PCES contract F33615-03-C-4110, the TRONE project CMUPT/RNQ/0015/2009, the General Motors-Carnegie Mellon Autonomous Driving Collaborative Research Lab, the Intel Science and Technology Center for Cloud Computing (ISTC-CC), as well as Carnegie Mellon's CyLab and Parallel Data Lab.

Contents

List of Figures

List of Tables

Whenever you can, count.

J. R. Newman (ed.), Commentary on Sir Francis Galton, 1956

Chapter 1

Introduction

THE use of large-scale distributed systems in production systems has become increasingly popular due to advances in cloud-computing and network technologies that have lowered the barrier to entry for businesses. Businesses often rely on these large-scale distributed systems to support Internet and telecommunication services such as e-Commerce, VoIP and business analytics. The evolution of these large distributed systems into entire *platforms* that provide dozens of distinct services to millions of users requires rethinking classic notions of availability as a binary property. Production systems are engineered for high-availability, and are rarely simply "up" or "down"; even when they are working for an overwhelming majority of users, there are almost always multiple ongoing problems of different types that affect small subsets of users or requests. Often, the symptoms of each individual problem are not big enough to trigger alarm thresholds, and thus they fly under the radar of operations teams that are geared towards major outages.

We call such problems *chronics*—performance degradations or failures that are perceivable by end-users, and that affect small subsets of end-users or requests. Chronic performance problems have also been referred to as *partial outages* or *brownouts* in Internet services [Kiciman, 2005]. Chronics can occur repeatedly but unpredictably for short durations of time, or persist for days or even weeks, affecting small subsets of users all the time. Although small individually, cumulatively, chronics contribute significantly to the degradation of the user experience. [ExtraHop.com, 2011] estimates that for every reported major outage, there are hundreds or even thousands of chronics. Our analysis of problems in Hadoop clusters corroborates the pervasiveness of chronics in production systems where chronics accounted for 78% of reported issues (see Table 3.6). Anecdotal evidence obtained

from a production Voice-over-IP (VoIP) platform at a major Internet Service Provider (ISP) also revealed that even in the worst month for major outages, chronics accounted for 43% of failed (*i.e.*, dropped or blocked) calls.

The discovery and diagnosis of never-before seen chronics in production systems comprising thousands of network, server, and user elements poses new challenges compared to the diagnosis of major system outages. Threshold-based techniques [Mahimkar et al., 2009; Cohen et al., 2005; Bodik et al., 2010] do not work well because lowering thresholds to detect chronics often increases the number of false positives. Long-running persistent chronics can get absorbed into a system's definition of "normal", thus posing problems for methods based on historical models [Kandula et al., 2009] or change-point detection [Agarwal et al., 2006]. Isolating individual problems is also more difficult—due to their persistent nature, lots of chronics are often present in a system at once, all starting and ending at different times, with larger problems hiding smaller ones. Furthermore, they occur even when the system works well for most users, and cannot be diagnosed by isolating the system's execution into periods of "good" and "bad" behavior [Sambasivan et al., 2011; Oliner et al., 2010]. Finally, chronics may involve some unexpected combination of corner-cases that impact only small subsets of users, *e.g.*, a configuration error that impacts only those users with a particular version of a software stack, or a performance degradation that occurs only when the load on a particular server temporarily increases beyond a certain threshold.

Therefore, production systems require diagnosis techniques that can scalably analyze a wide range of data sources from many elements of different types, from different vendors, often in the form of partially-structured logs with widely differing formats. To capture the user-visible symptoms of chronic problems, these diagnosis techniques need to adopt a *top-down* view of the system that captures the user's experience as their request traverses across the interconnected components within the service provider's network. Furthermore, any technique must be both accurate and timely. Anomaly detection techniques such as [Yemini et al., 1996; Oliner et al., 2010] that detect problems based on low-level events such as individual log alarms or resource utilization charts can suffer from a high false positive rates when diagnosing chronics because anomalies in low-level events do not always correlate to end-user issues. On the other hand, failing to detect problems until customers complain leads to lost or dissatisfied customers, damage to the company's reputation, and possible impact on the company's stock price.

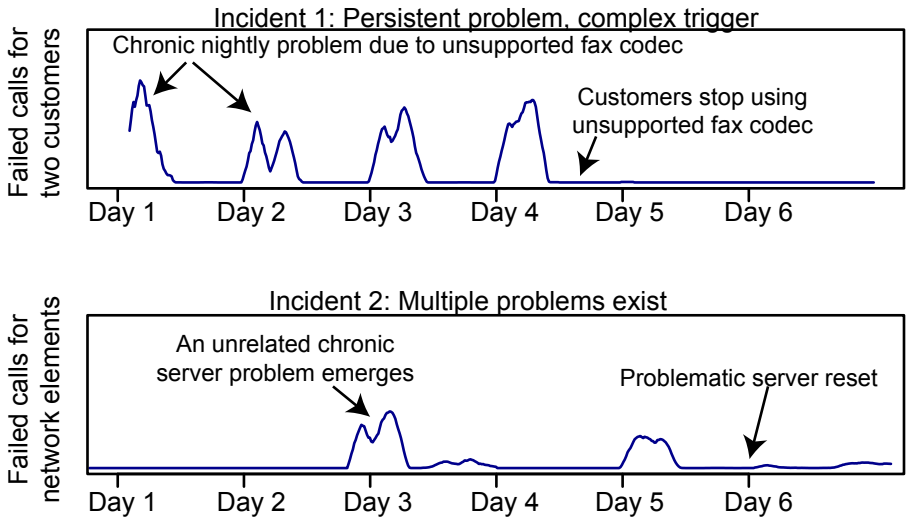


Figure 1.1. **Multiple ongoing problems at a production VoIP system.** These problems affected calls passing through the same network element at the production VoIP system.

1.1 Challenges in Diagnosing Chronics

Diagnosing chronic performance problems in production systems is challenging. Chronic performance problems can occur for a variety of reasons such as misconfigurations at the customer-site that affect some, but not all, requests made by the customer. These problems persist until the customer fixes their configuration. Increases in system workload can also cause regularly occurring chronics (*e.g.*, during peak business hours). Such problems may be due to under-provisioning of resources within the service provider's network, or customers exceeding their resource caps (*e.g.*, number of concurrent calls in a VoIP system). Equipment failures such as a bad row of memory or a bad disk can also cause intermittent failures. Operators could ignore these problems if they were one-off incidents. However, the recurrent nature of these problems negatively impacts customer satisfaction over time.

We illustrate the challenges faced when diagnosing chronics by using examples of real incidents experienced in a production Voice-over-IP system [Kavulya et al., 2012a]. Figure 1.1 shows actual instances of chronic problems in the service provider's logs that were discovered using our diagnosis approach. In the first incident, the recurrent increase in defects during night hours was traced to two different business customers, who were attempting to send faxes overseas using unsupported codecs during US night time. In the second incident, an independent problem with a specific network element arose and per-

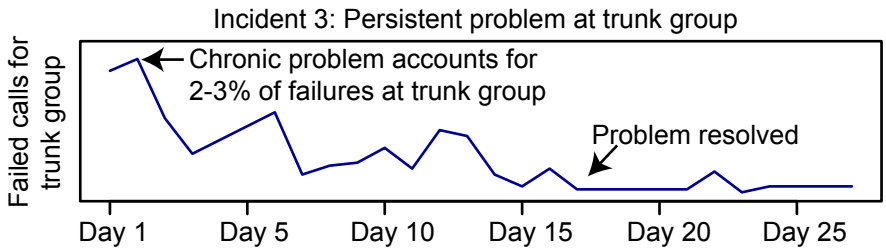


Figure 1.2. **Persistent chronic problem at a production VoIP system.** The chronic problem persisted for several weeks making it difficult to detect using change-points.

sisted until the network element was reset. Figure 1.2 shows a persistent chronic problem due to two blocked CICs (Circuit Identification Codes) on the trunk group that affected calls assigned to these blocked CICs in a round-robin manner. At peak, 2–3% of the calls passing this trunk group would fail. After those CICs were unblocked, the total defects associated with this error code were reduced by 80%.

These incidents highlight the challenges faced when diagnosing chronics namely:

1. **Chronics can fly under the radar.** Chronics occur sporadically, or affect a small subset customers, and thus may not trigger any threshold-based alarms. In the incidents shown in Figures 1.1 and 1.2, the defect rate observed by the customers was a fraction of one percent. Setting thresholds to detect these problems is notoriously difficult because lowering the thresholds to detect chronics would increase the number of spurious alarms.
2. **Performance variations.** Variance in production systems can occur due to legitimate sources (*e.g.*, hardware and load differences) or illegitimate sources (*e.g.* bugs, misconfigurations, resource contention, hardware failures). High-variance complicates the task of setting thresholds for problem detection—setting higher thresholds (or confidence levels) results in higher false negatives, while setting lower thresholds results in higher false positives [Sambasivan and Ganger, 2012a].
3. **Persistent problems.** Some problems, occur only for short durations of time, and could be discovered by change-detection algorithms. However, other problems persist for long periods of time as shown in Figure 1.2. Algorithms that rely on change-point detection methods [Agarwal et al., 2006] or those that rely on historical models [Kandula et al., 2009] would fail to detect these problems.

4. **Multiple independent problems.** Because chronics often persist for long periods of time before they are discovered, there are usually many of them ongoing at the same time. Figure 1.1 shows multiple ongoing problems in the VoIP system that affected calls passing through the same network element—one related to two different business customers, and one related to the network element.
5. **Complex triggers.** Chronics often involve only a small subset of user interactions because they are triggered by some unforeseen corner case arising due to a combination of factors. For example, certain chronics arise due to a conflict between the configuration at the customer's premises, and the ISP's server. To effectively debug these problems, operators need to know both the server configuration, and the subset of customers affected.
6. **Scale.** Production environments can have hundreds or thousands of nodes. Operators can be daunted by the prolific amount of monitoring available in these systems and they might not always know where to look when things go wrong.
7. **Labeled failure-data not always available.** Chronics can arise due to unforeseen problems (such as failed upgrades) that would not be addressed by techniques that rely on signatures of known problems [Cohen et al., 2005; Bodik et al., 2010].
8. **Desired level of instrumentation might not be possible.** Operators have to use as-is vendor instrumentation over which they have limited control. The cost of adding extra instrumentation might be high. In addition, the instrumentation in production systems might be diverse with different log formats. The diagnosis technique needs to cope with the diverse instrumentation sources available in production systems.

1.2 Thesis Statement

This dissertation explores the following hypothesis:

> *Diagnosis of chronic performance problems in production systems is possible through the analysis of common white-box logs to extract local behavior and system-wide dependencies, coupled with the analysis of common black-box metrics to identify the resource at fault.*

Specifically, this dissertation presents a holistic framework for diagnosing chronics in production systems that relies on a suite of statistical tools to detect user-visible symptoms

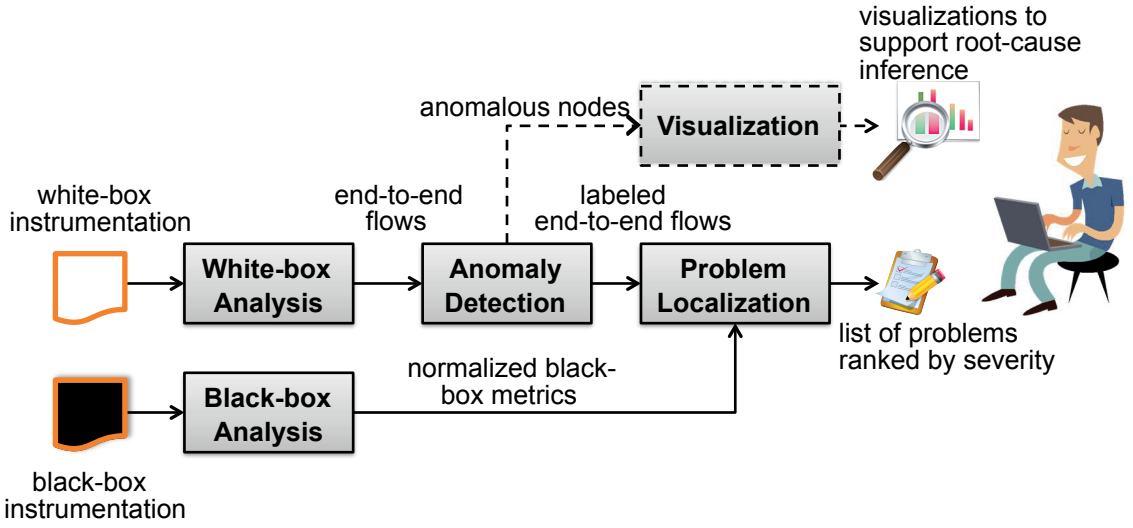


Figure 1.3. **Overview of diagnostic framework.** The diagnostic framework comprises of a suite of tools that analyze white- and black-box data to localize the source of chronic performance problems.

of problems (*e.g.*, slow requests), and drill-down on the source of the problems by analyzing unmodified white-box (*i.e.*, application-level) and black-box (*i.e.*, system-level) logs. The use of unmodified logs makes our framework amenable for use in production systems where we may not have the luxury of modifying existing instrumentation. Our diagnostic framework comprises of the five stages illustrated in Figure 1.3. First, an extensible log-analysis framework analyzes white-box logs to infer system dependencies, and construct end-to-end causal flows. These end-to-end flows capture the interactions between components, such as servers, that affect the user's experience with the system. Second, anomaly-detection tools detect user-visible problems using the end-to-end flows, and label each flow as successful or failed. The anomaly-detection tools combines *domain-specific heuristics* with a peer-comparison approach to detect performance degradations. Third, a *top-down* problem-localization approach identifies the source of the problem using the labeled end-to-end flows by identifying attributes (*e.g.*, nodes, customers, error codes) that are more correlated with failed flows than successful flows. Fourth, flows on indicted nodes are annotated with black-box data such as CPU and memory-usage. The problem-localization algorithm incorporates this black-box data to identify the resource at fault, and outputs a list of identified problems ranked by severity. Fifth, a visualization tool that relies peer-comparison highlights anomalous nodes in parallel-processing clusters. These visualizations complement our diagnosis algorithms by supporting root-cause inference.

Goals. The primary goals of this dissertation are:

- Diagnosis of chronic performance problems using common instrumentation available in production systems.
- Anomaly detection in the absence of labeled failure-data.
- Differentiation of workload changes from anomalies.

Non-goals. This dissertation does not address the following:

- Diagnosis of system-wide outages.
- Diagnosis of value faults and transient faults.
- Root-cause analysis at code-level, *i.e.*, finding faulty line of code.

Assumptions. This dissertation makes the following assumptions:

- Majority of the system is working correctly.
- Problems manifest as observable behavioral changes (*e.g.*, exceptions or performance degradations) that are visible to the end-user.
- White-box instrumentation supports inference of end-to-end flows by capturing local behavior, and dependencies with adjacent nodes.
- All instrumentation is locally time-stamped.
- Clocks are synchronized to enable system-wide correlation of data.
- Instrumentation faithfully captures system behavior.

1.3 Thesis Map

This dissertation explores the diagnosis of chronic performance problems in production systems. We apply our diagnostic approach to traces from two production systems namely: academic cloud-computing clusters running the Hadoop parallel processing framework [Apache Software Foundation, 2007], and a Voice-over-Internet-Protocol (VoIP) system at a major Internet Service Provider. Our approach is not limited to these two systems, but is also applicable to other platforms such as Internet services that serve users via independent requests, and support the generation of end-to-end causal flows.

The first hurdle we faced when developing our diagnostic approach was understanding the characteristics of chronics, and the limitations of existing research in addressing these problems. Chapter 2 discusses related work on problem diagnosis in distributed systems.

We group the related work into five major categories, and discuss the strengths and limitations of these approaches in the context of chronics. [Kavulya et al., 2012b] presents an extended version of this chapter. Chapter 3 provides insight on prevalence of chronics in the production Hadoop clusters and the VoIP system. We also present a comprehensive analysis of the performance characteristics of workloads in the Hadoop clusters [Kavulya et al., 2010]. This workload characterization informed our diagnostic approach.

Chapter 4 describes our extensible log-analysis framework for inferring end-to-end causal flows from the unmodified white-box logs. These end-to-end flows capture the user's experience with the system by tracing the control path and data demands of application requests as they are serviced across components and machines in the production system. The framework extends SALSA's state-machine abstraction [Tan et al., 2008] for log-analysis by supporting a configurable environment for describing log files, and expressing rules for generating end-to-end flows.

Chapter 5 presents our two-pronged anomaly-detection approach. Whenever available, we rely on domain-specific heuristics to detect anomalies—especially in the VoIP system which has well-established heuristics for detecting blocked or dropped calls. In the absence of heuristics, we developed a practical peer-comparison approach that detects performance degradations in the end-to-end flows in Hadoop clusters. The peer-comparison approach copes with heterogeneity at the application-level by exploiting regression to factor out legitimate sources of variance such as data skews.

Chapter 6 describes our problem-localization approach [Kavulya et al., 2012a,c], which uses a *top-down* statistical approach to identify sets of *problem signatures* that together explain the differences between the failed and successful end-to-end flows. This statistical approach is robust to noise introduced by the occasional mislabeling of flows. The problem-localization engine is scalable and domain-agnostic—supporting 10s of millions of flows.

Chapter 7 explores the effectiveness of our diagnostic approach in a controlled environment using fault-injection, and benchmarks our approach against existing approaches [Pan et al., 2009a; Kiciman and Fox, 2005; Sambasivan et al., 2011]. Chapter 8 presents case studies showing how we effectively diagnosed chronic problems in a production Hadoop cluster and a VoIP system at a major ISP. Chapter 9 describes how we used peer-comparison to visualize problems in Hadoop clusters [Garduno et al., 2012]. Chapter 10 summarizes the dissertation and presents our thoughts on future work.

Table 1.1. **Contributions of thesis.** Contributions explained in the context of two production systems.

	VoIP	**Hadoop**
Anomaly Detection	Heuristics-based	Peer-comparison (no labeled data)
Problem Localization	Localize to customer/network-element/resource/error-code	Localize to node/task/resource
Types of Chronics	Exceptions or performance problems due to single/multiple sources	Exceptions or performance problems due to single/multiple sources
Experimental Evaluation	Production VoIP system 1000s of network elements	OpenCloud cluster 64 nodes
Publications	[Kavulya et al., 2011, 2012a,c]	[Pan et al., 2009b,a; Tan et al., 2008, 2010a,b; Kavulya et al., 2010]

1.4 Contributions

This dissertation presents a suite of statistical tools for detecting and localizing chronics in production systems. The contributions of this dissertation are:

In the absence of labeled failure-data, peer-comparison is an effective approach for detecting anomalies in production systems. Some user-visible problems manifest as errors such as timeouts. While the user is aware that there is a problem, the operations team may not be made aware of the problem until it perturbs as significant number of requests or the customer complains. Our diagnostic framework detects these problems by extracting error codes from failed flows, or by applying domain-specific heuristics, *e.g.*, detecting calls with zero talk-time. On the other hand, performance problems can be harder to detect because there is no outright error other than the user's frustration with the progress of their request. We exploit the notion of *p*eers to detect performance problems by identifying system behaviors that can be considered equivalent ("peers") under normal conditions. Significant deviation from peers is regarded as anomalous. We present a hybrid peer-comparison approach that exploits domain-specific knowledge about the structure of Hadoop jobs to identify peers, and uses stepwise-regression to automatically factor out sources of variance due to application-level differences among these peers.

Knowledge of system dependencies facilitates the diagnosis of chronic performance problems. The end-to-end flows capture the dependencies between the components in the production system that are servicing a user's request. Knowledge of these depen-

dencies enables our problem-localization approach to narrow down the source of problem when errors propagate across components. Our problem-localization approach relies on a scalable Bayesian distribution learner and an information-theoretic measure of distance [Kullback and Leibler, 1951] to identify sets of problem signatures that together explain the differences between the failed and successful end-to-end flows. The statistical nature of our algorithm makes it robust to occasional mislabeling during anomaly detection. Our problem-localization approach also identifies the source of chronics triggered by a combination of factors by finding subsets of attributes that are more likely to appear in failed flows than successful ones. Our approach disambiguates between the sources of multiple ongoing problems by analyzing the impact of these problems on different sets of end-to-end flows.

Fusion of white-box and black-box data can provide more insight into the source of the problem. White-box data extracted from the application-level logs enables us to localize the source of the problem to customers, network-elements or nodes, job, tasks, and other categorical (or discrete) features available available in the logs (*e.g.*, error codes generated). The subsequent incorporation of black-box data such as CPU and memory usage can provide insight on the resource at fault.

Demonstration of the effectiveness of our approach on two production systems. We evaluated the effectiveness of our approach on an academic cloud-computing cluster running the Hadoop parallel processing framework, and a Voice-over-Internet-Protocol (VoIP) system at a major Internet Service Provider (see Table 1.1). This thesis provides empirical and anecdotal evidence showing that chronics are more prevalent than major outages in production systems. We demonstrate the effectiveness of our diagnostic approach on real incidents in these production systems. The operations team at the ISP has used an implementation of our diagnostic approach to diagnose real incidents in their production VoIP system since 2011.

1.5 Applying Approach to Different Systems

The problems we address, and our solutions, are not limited to VoIP and Hadoop. They are likely to be applicable to many other large platforms (*e.g.*, e-commerce, web-search, social networks) that serve users via independent interactions such as web requests. Our problem

localization approach is scalable and domain-agnostic (Chapter 6). Adapting our approach to other applications requires changes to the domain-specific steps namely: inferring the end-to-end flows from white-box logs (Chapter 4), and anomaly detection (Chapter 5). The changes needed to adapt our approach to other applications are:

White-box analysis. The end-to-end flows can be generated automatically using request-tracing tools [Fonseca et al., 2007; Barham et al., 2004; Sigelman et al., 2010] which monitor end-to-end behavior in distributed systems. If request-tracing is unavailable in the application, the end-to-end flows can be constructed using the steps described in Chapter 4. Domain-specific knowledge is needed to identify the attributes in the white-box logs that capture the control path, dependencies with adjacent nodes, and data demands of application requests. The attributes extracted also depend on desired granularity of problem localization. For example, extracting only hostnames results in coarse-grained problem localization that narrows the source of the problem to a given host. Finer-grained problem localization is possible by including attributes such as request types, error codes, and configuration-related information. For scalability reasons, the first step of our problem localization approach deals with categorical attributes (such as hostnames). Real-valued attributes such as CPU usage can be incorporated using the steps described in Section 6.4.

Anomaly detection. Anomalies in end-to-end flows can be detected by mining exceptions from the logs, or by using simple statistical techniques (*e.g.*, flagging anomalies if packet loss exceeds the 95th percentile of requests). Some requests exhibit high variance which complicates the task of setting thresholds for detecting problems—diagnosis algorithms either set higher thresholds resulting in higher false negatives, or set lower thresholds resulting in higher false positives [Sambasivan and Ganger, 2012a]. For example, job durations in Hadoop exhibit high variance (Section 3.2.2). This thesis proposes two techniques for reducing this variance thus improving the accuracy of anomaly detection. First, use domain-specific knowledge or clustering to identify groups with similar behavior, *i.e.*, peers, using the strategies discussed in Section 5.1.3. The behavior amongst peers may vary due to legitimate reasons, for example, tasks in Hadoop jobs which process more data take longer to complete. *Regression or normalization* can learn inter-relationships between explanatory variables to factor out sources of variability amongst peers (Section 5.1.4). Detect anomalies by identifying requests that deviate significantly from their peers.

1.6 Limitations

Our diagnostic approach provides a comprehensive solution for diagnosing chronic performance problems in production systems. However, we believe that no single approach will address all the types of problems encountered in production systems. Specifically, due to our fault-model, our diagnostic approach does not deal with problems that result in system-wide outages. Our approach also does address problems that do not result in user-visible problems, for example, failures that are masked by fault-tolerance mechanisms.

Our diagnostic approach provides a coarse-grained localization of the root-causes of the chronic problems. For example, we may localize the root-cause of the problem to a particular server or customer, but not to the exact line of source code or the configuration parameter causing the problem. The granularity of our diagnosis is limited by granularity of information available in the application-level logs that we analyzed. The strength of our approach lies in its ability to limit the scope of components that operators need to examine when identifying the fine-grained root-cause of chronics that arise due to complex system interactions. Our approach also helps operators to prioritize their troubleshooting efforts by ranking root-causes according to their impact on the end-users.

Our log-analysis framework relies on *domain-specific log-parsers* that use regular expressions to extract attributes of interest from the logs. One limitation of these domain-specific log parsers is their reliance of hard-coded regular expressions that vary based on the format of log messages. The parsers need to be manually updated if the underlying log format changes due to a software upgrade. Despite this limitation, we opted for this approach because we lacked visibility into the proprietary components in the production systems. The quality of end-to-end flows generated by our framework is also limited by the accuracy of information contained in the logs.

At present, our peer-comparison approaches for anomaly-detection and visualization rely on expert knowledge to identify peers. Identifying peers is easier in parallel-processing frameworks, such as Hadoop, which attempt to distribute load as evenly as possible across slave nodes in the cluster. Automatically identifying peers in heterogeneous systems such as VoIP is more challenging, and is not addressed in this dissertation. Instead, of peer-comparison, we rely on heuristics to identify anomalous flows in heterogeneous systems such as the production VoIP system.

If I have seen a little further it is by standing on the shoulders of Giants.

I. Newton, Letter to R. Hooke, 1676

Chapter 2

Related Work

The issue of diagnosing the underlying causes of hardware and software failures has existed for as long as computers have been around. Using the fault, error, and failure nomenclature of [Laprie, 1995], failure diagnosis is the process of identifying the fault that has led to an observed failure of a system or its constituent components. In any sufficiently large computing system, many types of faults are often not directly visible for a number of reasons—either due to the characteristics of the fault itself, due to fault-tolerance mechanisms built into the system that hide the expression of the fault, or as is most often the case, the lack of detailed monitoring functionalities that can detect and report on the occurrence of the fault directly. In some cases, monitoring systems may provide only an indication that a fault has occurred, but may not provide sufficient information to precisely locate it.

Failure diagnosis is a technically challenging endeavor because the relationship between faults, failures, and their observable symptoms is a complex one; single faults often produce multiple symptoms in different parts of a system, *e.g.*, a misconfiguration fault in a critical network component such as a Dynamic Host Configuration Protocol (DHCP) server can cause all client computers on the network to fail; conversely, similar symptoms may be caused by many different types of faults, *e.g.*, the failure of a networked computer to receive an IP address can have several causes including, but not limited to, packet loss in the physical network, a client misconfiguration, or a problem with the DHCP server. As operational systems become more mature, the failures they encounter often transition from easy to detect *hard failures* that cause a significant impairment to the system's primary function, to *soft failures* such as those due to performance bottlenecks, or transient faults that are much harder to detect. Therefore, the process of diagnosis often also includes the

identification of anomalous conditions that are symptoms of the occurrence of faults.

Due to the complexity of computing systems and difficulty of formalizing the scope of the diagnosis task itself, diagnosis has historically been a largely manual process requiring significant human input. However, techniques to automate as much of the process as possible have significantly grown in importance. In domains such as communication networks and Internet services, the sheer scale of modern systems and the high volumes of impairments they face drive such trends; while in domains such as embedded systems [Lanigan et al., 2011], the trends are driven by increasing complexity coupled with the need for autonomic operation (*i.e.*, self-healing) when human expertise is not available. Due to the diversity of the domains, a variety of failure diagnosis techniques drawing from diverse areas of computing and mathematics such as artificial intelligence, machine learning, statistics, stochastic modeling, Bayesian inference, rule-based inference, information theory, and graph theory have been studied in the literature. Finally, when automated techniques fail, approaches that assist humans perform diagnosis more efficiently via the use of visualization aids have also been widely deployed.

While a comprehensive survey of this broad topic can provide sufficient material for a book of its own, in this chapter, we provide a summary of the most important techniques. Table 2.1 provides a summary of the techniques described in this chapter. For each class of techniques, we describe different approaches proposed in the research literature, and conclude each discussion with a critique of the technique that highlights its strengths and limitations for diagnosing chronic problems. We compare the prior work against our statistical diagnosis approach that uses peer-comparison for anomaly detection, and a Bayesian algorithm for problem localization.

2.1 Rule-based Techniques

Rule-based techniques rely on expert knowledge expressed as a set of predefined directives, *i.e*. rules, to diagnose problems. The rules are typically formatted as a set of *if-then* statements where the *if-part* of the rule is called the *premise*, and the *then-part* of the rule is the conclusion. An example of a rule used for diagnosis is "*if* CPU utilization exceeds 90% *then* node is overloaded". Rule-based techniques for diagnosis typically rely on forward-chaining inference mechanisms [Steinder and Sethi, 2004] to synthesize results when mul-

Table 2.1. **Summary of Diagnosis Techniques.**

Technique	Critique
Rule-based techniques rely on expert knowledge expressed as a set of predefined rules to diagnose problems *(Section 2.1)*.	Rules are human-interpretable and extensible. However, they cannot diagnose unforeseen problems, and large knowledge bases are difficult to maintain.
Model-based techniques define a mathematical representation of a system, testing the observed state against the model to see if it conforms *(Section 2.2)*.	Model-based techniques are well suited for diagnosing application-level problems. However, building models requires a deep understanding of the system.
Statistical techniques summarize and interpret empirical data using techniques such as correlation, histogram comparison and probability theory, for diagnosis *(Section 2.3)*.	Statistical techniques require little expert knowledge or detailed models on system internals. However, they have difficulties distinguishing legitimate changes in behavior (*e.g.* workload changes) from illegitimate changes (*e.g.* performance problems).
Machine-learning techniques identify patterns in behavior using clustering, or use training data to determine if the likely cause of problems *(Section 2.4)*.	Machine-learning techniques automatically learn profiles of system behavior, but can suffer from the curse of dimensionality that reduces accuracy when the number of features is large.
Visualization techniques allow operators to visualize trends in data and spot anomalous behavior *(Section 2.5)*.	Visualization tools allow operators to explore different hypotheses on the root-cause of problems. However, they do not automatically identify they source of problems.

tiple rules fire. Forward inference processes events, such as high CPU and memory utilization, and uses the triggered rules to draw conclusions on the root-cause of the problem.

One approach for representing rules is *codebooks* [Yemini et al., 1996; EMC, 2009] which map each problem to a unique signature consisting of symptoms in both the faulty component where the problem occurs, and related components affected by the original problem. The codebook is instantiated as a dependency matrix where the columns represent the problems, and the rows represent the symptoms. Problems are uniquely diagnosable if all the columns are different. Codebooks diagnose the underlying problem by identifying the closest match to the observed symptoms. Other rule-based diagnosis tools, such as

Chopstix [Bhatia et al., 2008] and Vertical Profiling [Hauswirth et al., 2004] rely on a small collection of rules based on the semantics of the application, and the underlying behavior of the operating system to map changes in system performance on individual nodes to known problems. These tools provide an intuitive approach for diagnosing problems on individual nodes, however they currently do not correlate metrics across multiple nodes and do not address problems that can propagate across the network in distributed systems.

Diagnosis tools that analyze large sets of rules require more sophisticated techniques, such as expert systems that rely on forward inferencing to synthesize results and resolve conflicts when multiple rules fire. These expert systems allow administrators to cope with the deluge of alarms generated by large-scale distributed systems. JECTOR [Liu et al., 1999] presents a specification language for expressing rules that captures the timing relationship among correlated events. For example, alert operator if a link is down and no corresponding link up event occurs within 2 minutes. Commercial tools such as `HP Operations Manager` [Packard, 2010] use an optimized Rete algorithm [Forgy, 1982] to perform pattern matching on rules in a scalable manner that is independent of the number of rules.

Critique. Rule-based approaches are prevalent in commercial tools, such as `IBM Tivoli Enterprise Console` [IBM, 2010] and `HP Operations Manager` [Packard, 2010], as they offer an intuitive approach for expressing system behavior that allows users to augment the rule-base by developing new rules tailored to their unique operating environments. In addition, rule-based systems do not require profound understanding of the underlying system architectural and operational principles. However, rule-based systems suffer from the inability to learn from experience, and the inability to deal with problems not described within the rule-base. Rule-based systems are also difficult to maintain because the rules frequently contain hard-coded network configuration information [Steinder and Sethi, 2004]. Our diagnostic framework relies on rules to identify failed calls at the production VoIP system since telecommunications systems have well-established rules for detecting blocked or dropped calls. To address the limitations of rule-based approaches, we augmented the rules with a peer-comparison approach for detecting anomalies in the absence of labeled failure data.

2.2 Model-based Techniques

Model-based techniques define a mathematical representation of a system, and test the observed state of the system against the learned model to diagnose problems. Some models represent the normal operation of the system, and detect problems whenever the observed system behavior fails to conform to the learned model. Other techniques generate graphical models of how problems propagate through the system [Kompella et al., 2005; Bahl et al., 2007], and exploit this knowledge to infer the source of the problem. Alternatively, graphical models [Joshi et al., 2005; Rish et al., 2004; Tati et al., 2012] exploit probes to identify failures in the system. Probes are end-to-end test transactions which gather information about system components, for example, a dummy query which tests if the database is running. These graphical models then analyze patterns of probe failures and successes to infer the source of the problem. Lastly, graphical models may represent expected communication patterns within a system and flag problems whenever these patterns are violated.

Model-based techniques can be classified into: 1) *physical model based techniques* which use the physical laws that a system operates under to model constraints on system behavior; 2) *regression and queuing models* which model relationships between resource consumption and application behavior; and 3) *graph-theoretic models* which exploit knowledge on how errors or successes propagate in a system to localize problems.

Physical models use models of the physical world, such as the laws of mechanics, electromagnetics, or chemical kinetics to model system behavior and to determine when anomalous behavior is present. They typically model continuous cyber-physical systems in industrial, automotive and aerospace domains whose physics are well understood, *e.g*, powertrain [McCullough et al., 2007] and chassis systems [Huh et al., 2008] in cars. These systems run in a closed-loop, where sensors monitor the system output, then feed the data into a controller that signals actuators to adjust control as necessary to maintain the desired system output. Problems are diagnosed by executing the physical model alongside the actual system at run-time to detect when the system fails to conform to the model. The fault model typically associated with the control-theoretic approach includes sensor faults, actuator faults, and faults in the mechanical, electromechanical, or hydraulic plant being controlled [Lanigan et al., 2011].

Regression and queuing models are useful for workload characterization, capacity planning and detecting performance problems. These models represent relationships between resource consumption and application behavior, and detect anomalies whenever these relationships are violated. Some techniques model multi-tier Internet applications as queues, and use mean-value analysis [Liu et al., 2005; Urgaonkar et al., 2005] to predict transaction response times. These techniques use a network of queues to represent how the tiers in the multi-tier application cooperate to process requests. Mean-value analysis assumes closed queueing models in which the number of clients in the system remains constant. However, it is often difficult in practice to obtain the client session information required to calibrate closed models for real-world production applications [Stewart et al., 2007].

Production workloads are non-stationary, *i.e.*, the relative frequencies of transaction types changes over time. Queuing approaches which leverage regression to learn the relationship between resource consumption and application behavior can be used to predict response times for non-stationary workloads [Kelly, 2005; Stewart et al., 2007]. These models assume that the system contains a small number of types of transactions, and that transaction types strongly influence system resource demands. These models rely on open queues, where clients can join and leave the system model. Open models facilitate more thorough empirical validation in production systems than would be possible with closed models as they do not require client session information [Stewart et al., 2007].

In addition, using queuing theoretic approaches to model transaction mixes allows these systems to distinguish anomalies from workload changes. [Cherkasova et al., 2008] use queues to model the relationship between CPU usage and transaction response times for a transaction mix. They also exploit regression to define an application performance signature that allows them to detect software upgrades by monitoring changes in the application signature. [Stewart et al., 2007] model the relationship between multiple physical resources, namely CPU, disk and network, and response times for a transaction mix. Modellus [Desnoyers et al., 2012] uses queuing theory and stepwise-regression to automatically derive models that predict the resource usage of an application. Stepwise-regression allows Modellus to automatically identify features that best predict the observed resource usage or workload. However, these models need to be re-trained to cope with new transaction types. They also ignore interaction effects across transaction types and implicitly assume that queueing is the only manifestation of congestion.

Graph-theoretic models analyze communication patterns across nodes and processes to model the probability that errors, or successes, propagate through the system. The models may also monitor violations in expected communication patterns. Graph-theoretic models are useful for diagnosing both correctness and performance problems in distributed systems. They can be used to detect multiple independent problems—ranking them by likelihood of occurrence.

SCORE [Kompella et al., 2005] and Shrink [Kandula et al., 2005] localize problems in an IP network by modeling error propagation patterns in the wide-area networks. Both Shrink and SCORE model the system as a two-level graph between the IP layer and the underlying wide-area network. [Bahl et al., 2007; Khanna et al., 2007a] extend Shrink and SCORE to deal with multi-level dependencies and with more complex operators that capture load-balancing and failover mechanisms. These techniques infer the root-cause by computing the probability that errors propagate from a set of possible root-cause nodes to the observation nodes. They indict the root-cause nodes that best explain the symptoms at the observation nodes, and assume that there can only be a small number of concurrent problems in the system at a given time.

[Rish et al., 2004] propose an active probing approach that exploits a dependency-matrix to represent the failed components that each probe, *e.g.*, server ping, detects. Active probing allows probes to be selected and sent on-demand, in response to one's belief about the state of the system. At each step the most informative next probe is computed and sent. As probe results are received, belief about the system state is updated using probabilistic inference. [Joshi et al., 2005] use a Bayesian approach to diagnose problems in systems with different types of monitors, or probes, that have differing coverage and specificity characteristics. They use a dependency matrix to represent the probability that a monitor detects a failure in a component, and incrementally update their belief about the set of failed components based on the observed monitor output. [Khanna et al., 2007b] address diagnosis in distributed systems where errors can propagate across nodes. They track message exchanges between nodes and detect problems by comparing communication patterns against a rule-base of allowed state transitions. [Tati et al., 2012] identify faulty network elements despite incomplete symptoms during large-scale failures. They utilize a knowledge base of possible network paths and end-to-end symptoms (comprising of successful and failed probes) to output list of elements whose failures are consistent with the symptoms.

Critique. Model-based techniques are well-suited for diagnosing application-specific problems, such as chronics, because they encapsulate semantic knowledge on the expected behavior of the system. The incorporation of semantic knowledge can also help them distinguish legitimate changes in behavior, *e.g*. workload changes, from illegitimate changes due to failures [Kelly, 2005; Cherkasova et al., 2008; Stewart et al., 2007]. Our hybrid peer-comparison approach relies on regression to factor out legitimate sources of variance. As with Modellus [Desnoyers et al., 2012], we rely on stepwise-regression to automatically select the best features for our regression models, and detect anomalous end-to-end flows. Graph-theoretic techniques [Rish et al., 2004; Khanna et al., 2007b; Bahl et al., 2007] can be used to detect multiple independent problems—ranking them by likelihood of occurrence. However, these techniques do not address problems due to complex triggers as they assume that the root-cause of the problem stems from a single component. In addition, model-based techniques that rely on physical models require a deep understanding of system behavior to construct the models which is not always feasible in production systems. Even in cases where automatic model construction is feasible, there is often a tradeoff between the amount of semantic knowledge the model incorporates and the fidelity of the diagnosis. For example, graph-theoretic models [Bahl et al., 2007] that are automatically constructed by examining a system's communication patterns can localize a problem to a single node or a small neighborhood of nodes, but cannot tell what the deeper root cause is. Another disadvantage of model-based techniques is that they can fail to detect novel problems that were not considered in the model.

2.3 Statistical Techniques

Statistical techniques for diagnosis summarize and interpret empirical data using techniques such as correlation, histogram comparison and probability theory. These techniques are data-centric and require little expert knowledge or detailed models on system internals. Statistical techniques are either: 1) *parametric* techniques that assume data is drawn from a known distribution, *e.g*., normal distribution, or 2) *non-parametric* techniques that do not rely on data belonging to a particular distribution but rather estimate the underlying distribution, *e.g*., using histograms or kernel density estimation. Non-parametric methods make fewer assumptions than parametric methods, making them more robust and giv-

ing them wider applicability. However, there is a cost—larger sample sizes are required to draw conclusions with the same degree of confidence as parametric methods. Our diagnosis approach is a non-parametric statistical approach that relies on a Bayesian algorithm for localizing problems by comparing differences in the distributions of attributes, such as hostnames and customer IP addresses, in successful and failed user-interactions.

Statistical techniques are pervasive in problem diagnosis literature. Some model-based techniques discussed in Section 2.2 rely on statistical techniques, such as correlation and regression, in conjunction with deep knowledge of the application's behavior to diagnose problems. In contrast, the statistical techniques discussed in this section make fewer assumptions about the application's behavior.

Parametric techniques assume that data is drawn from a known distribution. Normal distributions are commonly used for anomaly detection and diagnosis because of their tractability, and because normality can sometimes be justified by the *central-limit theorem* which explains why many distributions tend to be close to the normal distribution. These techniques typically detect anomalous behavior by identifying significant deviations from the mean for performance counters, which they assume follow a normal distribution. However, hardware failure rates are better modeled using Weibull distributions which capture the increased failure rates of devices as they age [Schroeder and Gibson, 2006, 2007].

Agarwal et al [Agarwal et al., 2006] use change-point detection and problem signatures to detect performance problems in enterprise systems. They detect abrupt changes in system behavior by monitoring changes to the mean value of performance counters over consecutive windows of time. This technique does not scale well if the number of nodes and metrics is large. NetMedic [Kandula et al., 2009] diagnoses propagating problems in enterprise systems by analyzing dependencies between nodes, and correlations in state perturbations across processes to localize problems. NetMedic represents state for each system component as a vector that indicates whether each metric was anomalous or normal by assuming that each metric obeys a normal distribution and flagging anomalies based on deviation from the mean.

Non-parametric techniques assume that data is drawn from an unknown distribution. Non-parametric techniques estimate the underlying data distribution using histograms or

kernel density estimators, or make generalizations about the populations from which the samples were drawn, *e.g.*, using correlation.

Histogram-based techniques typically diagnose problems by comparing histograms (or distributions) of performance counters before and during an anomalous period to identify the metrics most likely to be associated with the problem. [Kasick et al., 2010; Tan et al., 2008; Pan et al., 2009a] diagnose problems in large clusters using histogram-comparison of performance counters to identify "odd-man-out" behavior. Peer-comparison allows these approaches to be robust to workload changes. However, propagating errors, *e.g.*, packet-loss that affects communication across multiple nodes, reduces their accuracy. [Shen et al., 2009] propose a reference-driven approach to diagnose performance problems due to configuration changes or upgrades. Their approach relies on histogram comparison to identify the collection of single-parameter changes that best explain the performance deviation observed. [Liu et al., 2006] uses a Bayesian approach to compare the distributions of failed and successful predicates when debugging software problems. Carat [Oliner et al., 2012] detects energy bugs in mobile devices by identifying applications running on the device whose energy-use distribution diverges significantly from similar devices running these applications. Our problem localization approach also compares distributions of successful and failed user interactions to localize problems in production systems, but with a more comprehensive fault model than these techniques.

Correlation-based techniques analyze historical data to automatically discover relationships between pairs of metrics that are stable over time [Jiang et al., 2009a,b]. Changes in these learned correlations may signal problems. Correlation can also be used to automatically discover causal relationships between metrics in distributed systems. Giza [Mahimkar et al., 2009] exploits knowledge of the system's topology to identify spatial correlations between events, and discover causal relationships between the observed symptoms and root-cause events. [Oliner et al., 2010] also use cross correlation to discover causal relationships between anomaly signals across components. The anomaly signals represent the changes in the behavior of components over time in terms of resource usage, message timing or semantics. Project5 [Aguilera et al., 2003] records packet traces at each node and uses message correlation algorithms to automatically extract end-to-end causal traces for requests, and detect high-latency paths. Correlation-based approaches can discover spurious relationships depending on the thresholds used to determine whether a correlation

is significant. In addition, correlation-based approaches do not scale well if the number of nodes and metrics is large.

Dimensionality-reduction techniques, *e.g.*, Principal Component Analysis, can reduce the number of metrics to compare when diagnosing problems by summarizing dominant trends. Intemon [Hoke et al., 2006] detects anomalies in large clusters by identifying sudden changes in system behavior, which are indicated by the change on the number of hidden variables. [Xu et al., 2009] use source-code analysis to apply structure to console logs and discover dominant historical trends in application state and message counts using principal component analysis. PeerWatch [Kang et al., 2010] uses peer-comparison to detect anomalies in heterogeneous clusters running different hardware. Their peer-comparison algorithm uses a dimensionality-reduction technique known as canonical correlation analysis to normalize performance differences due to different hardware, and discover correlations between peers.

Critique. Statistical techniques require little expert knowledge or detailed models of system internals. These diagnosis techniques can rely on well-established statistical theories to ground their algorithms, and test that their results are statistically significant, *i.e.*, unlikely to have occurred by chance alone. For example, hypothesis tests such as the t-test, allow us to reject the hypothesis that the observed system behavior is consistent with the expected system behavior with a degree of confidence. When building statistical profiles of behavior, care must be taken to include sufficient data samples and test assumptions on data distributions to ensure validity. Statistical approaches that rely on event correlation [Oliner et al., 2010; Mahimkar et al., 2009; Yemini et al., 1996] support diagnosis of multiple independent problems, and might be applicable in our system when there are resource-contention problems due to overload within the service provider's network. However, most of the chronics we have observed are due to customer-site problem such as misconfigurations, and operators at the ISP lack access to customer-site data other than names of the customer—therefore event-correlation might not be possible. Our approach localizes these chronics by analyzing data that is causally-related with each call rather than alarm signals across the network. Peer-comparison techniques [Kasick et al., 2010; Tan et al., 2008; Pan et al., 2009a] which compare behavior across node-groups are not well-suited for diagnosing chronics due to multiple independent faults. They also experience higher false-positives when problems cause errors to propagate across the system.

2.4 Machine Learning Techniques

Machine learning is a scientific discipline that is concerned with the design and development of algorithms that allow computers to evolve behaviors based on training data. Machine-learning techniques borrow heavily from statistical techniques, *e.g.*, data distributions and probability theory. Machine learning relies on training and cross-validation which involves partitioning a sample of data into complementary subsets, performing the analysis on one subset called the training set, and validating the analysis on the other subset called the validation set or testing set.

Diagnosis algorithms that rely on machine learning can be categorized into two broad categories namely: 1) unsupervised learning which identifies patterns in unlabeled data typically through clustering, and 2) supervised learning which infer a function that best classifies successful and failed states from labeled data.

Unsupervised learning identifies patterns in unlabeled data typically through clustering, and detects unexpected outlier data points that might be indicators of failures.

[Kiciman and Fox, 2005] use probabilistic context-free grammars to model the causal paths in the system. The grammar rules represent the probability that one component calls another. They identify anomalous causal paths by measuring the difference between the probability of the observed transition and the expected probability of the transitions that make up the causal path. Magpie [Barham et al., 2004] uses a string-edit-distance comparison to group together requests with similar behavior, from the perspective of request structure, synchronization points and resource consumption. The representative requests from each clusters allow them to construct concise workload models and detect outliers. [Thereska et al., 2010] exploits clustering of data from real-world deployments to generate performance signatures of application behavior that supports *what-if* analysis.

Supervised learning uses labeled data of successful and failed states to learn which metrics are most correlated with failed states, or to identify signatures of recurrent problems from a database of known problems.

Metric attribution approaches localize problems by identifying resource-usage metrics or components that are highly correlated with failed states. They allow operators to sift through the hundreds or thousands of metrics available in their system and narrow down the handful of metrics that yield insight to the cause of the problem, and its location. Once

the operators determine the root-cause, they can then annotate the output of metric attribution with the root-cause and build the database of known problems used by signature-based approaches.

Pinpoint [Chen et al., 2002] and MinEntropy [Chen et al., 2004] localize components highly correlated with failed requests using data clustering [Chen et al., 2002] or decision trees [Chen et al., 2004]. These approaches detect problems that result in changes in the causal flow of requests such as exceptions. More recently, Spectroscope [Sambasivan et al., 2011] categorizes requests based on functionality, *e.g.*, read or write requests, and applies data clustering to requests in each category to identify outliers due to changes in causal flows or request durations. These techniques have typically been used to diagnose infrastructural problems, such as database faults and software bugs (*e.g* infinite loops and exceptions) which lead to a marked perturbation of a subset of requests. In principle, techniques such as decision trees should fare well at diagnosing both major outages and chronics. However, decision trees did not fare well at diagnosing chronics when we applied them to our dataset (see Section 7.6). [Cohen et al., 2004] use tree augmented Bayesian networks to determine which resource-usage metrics are most correlated with the anomalous periods. They proposed an extension [Zhang et al., 2005] to their work that uses ensembles of Bayesian models to adapt to changing workloads and infrastructure.

Signature-based approaches allow system administrators to identify recurrent problems from a database of known problems. Signature-based approaches have wide applicability because studies have shown that typically half, and as much as 90% of software failures are due to recurrent problems [Duan and Babu, 2008]. Research has centered on how to represent and retrieve signatures of known problems from the database of known problems. However, these approaches do not fare well at automatically identifying problems that have not previously been diagnosed. [Yuan et al., 2006] learn signatures of known problems in standalone systems by analyzing sequences of system calls. They use multi-class Support Vector Machines to learn signatures of problems. However, their approach does not address distributed systems. [Cohen et al., 2005] and Bodik et al. [Bodik et al., 2010] generate signatures of recurrent problems in distributed systems by using the discrete feature vectors obtained through metric attribution. They found that they can leverage signatures learned at one geographical location to diagnose problems in data centers at a different location. [Duan and Babu, 2008] present an approach that can be used for

both known problems, and problems that have not previously been seen. They use a supervised approach (decision trees or signature databases) to identify recurrent problems. If the current failure does not match the annotated failures in the database, they compare it to the healthy data to identify features that are correlated with the failure. They then select multiple instances of the same failure which they can present to the system administrator to annotate.

Critique. Machine-learning techniques automatically learn profiles of system behavior, for example, using clustering to identify signatures of known problems. Machine-learning can also help localize problems by identifying resource-usage metrics or components that are highly correlated with failed states. However, these techniques can suffer from the curse of dimensionality that reduces accuracy when the number of features is large. Additionally, they are also susceptible to *overfitting*, a phenomenon in which the learner learns features of the evidence that are circumstantial rather than those that actually define the relationship between the faults and their effects. Over-fitted models generalize poorly, and can fail when presented with evidence that is only slightly different from the one on which the model was trained. Finally, because machine learning techniques learn a direct mapping between the symptoms and underlying root causes without an intermediate structural model of the system, lengthy retraining is required whenever the system behavior changes significantly. Furthermore, previously learned models often have to be thrown away during the period of retraining, leaving the system vulnerable to any problems. Therefore, machine learning techniques may not be appropriate for systems that are upgraded frequently.

2.5 Tracing and Visualization Techniques

Request-tracing tools. Request-tracing tools automate the monitoring of end-to-end behavior in distributed systems. The end-to-end flows generated by these tracing tools can serve as input into our diagnostic framework. Magpie [Barham et al., 2004], X-trace [Fonseca et al., 2007], and Dapper [Sigelman et al., 2010] are primarily tools for tracing causal request paths, but they also offer support for visualizing requests whose causal structure or duration is anomalous. Pip [Reynolds et al., 2006] detects application-specific problems in distributed systems by allowing programmers to embed expectations about application behavior in the source code. X-ray [Attariyan et al., 2012] diagnoses the root causes of

performance problems by instrumenting binaries as applications execute, and by using dynamic information flow tracking to estimate the likelihood that a block was executed due to each potential root cause. X-ray also highlights performance differences between two similar activities by differentially comparing their request execution paths.

Visualization tools. Visualization tools complement diagnosis tools by allowing operators to cope with these scenarios by: 1) summarizing data trends; 2) supporting interactive graphs that allow operators to explore different hypotheses on the root-cause of problems; and 3) integrating output from automated diagnosis tools.

[Ganglia, 2007; Splunk Inc., 2005; Sigelman et al., 2010] provide an array of simple graphs, *e.g*. line plots, barcharts, and histograms, to display trends in performance counters such as CPU utilization. They use simple statistical tests such as the deviation from the mean to flag outliers, and use color to highlight these outliers. LiveRAC [McLachlan et al., 2008] is a visualization system that supports the analysis of large collections of system management time-series data consisting of hundreds of parameters across thousands of network devices. LiveRAC provides high information density using a re-orderable matrix of charts, with semantic zooming that dynamically adapts different aspects of each chart based on available space. Artemis [Cretu-Ciocarlie et al., 2008] provides a pluggable framework for distributed log collection, data analysis, and visualization. Mochi [Tan et al., 2009] is a log-analysis based debugging tool that visualizes both the flow of data and the flow of control for a large-scale parallel processing framework known as Hadoop. NetClinic [Liu et al., 2010] visualizes data from computer networks using directed graphs, and presents suggested diagnostics for observed problems by incorporating output from an automated analytic reasoning engine [Kandula et al., 2009].

As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications.

D. Parnas, Communications of the ACM, 1990

Chapter 3

Workload Characterization

ANECDOTAL evidence suggests that for every major outage or blackout, there are hundreds or even thousands of chronics [Kiciman, 2005]. To gain insight on the prevalence of chronics and understand the performance characteristics of workloads in production systems, we analyzed traces from two production systems namely: two academic cloud-computing clusters running the Hadoop parallel processing framework [Apache Software Foundation, 2007], and a Voice-over-Internet-Protocol (VoIP) system at a major Internet Service Provider.

This chapter describes the production systems (Section 3.1), provides a detailed analysis of the performance characteristics (Section 3.2.2), and prevalence of problems in jobs executing on the Hadoop clusters (Section 3.3). We also present results of a Hadoop user study that highlights the shortcomings of existing troubleshooting tools (Section 3.2.1). Due to confidentiality concerns, we only present anecdotal evidence of chronics in the VoIP system. The anecdotal evidence provides insight on the prevalence of chronics (Section 3.4).

An understanding of the performance characteristics of workloads in production systems helped inform our diagnosis approach as follows: 1) the prevalence of chronics motivated our *top-down statistical approach* that drills down from user-visible symptoms of problems (*e.g.*, slow or failed user-interactions) to localize the source of the problem; 2) the presence of peer groups in the data (*e.g.*, tasks from the same job) motivated our use of peer-comparison to detect performance problems; 3) the variance in task durations motivated our use of regression to factor out legitimate sources of variance (*e.g.* differences in data input sizes) when identifying performance problems. Table 3.8 summarizes the implications of our workload characterization on diagnosis.

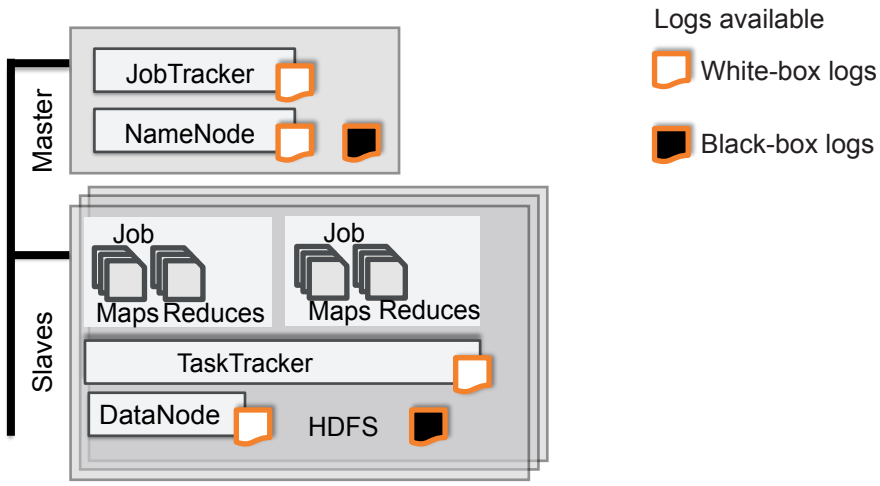


Figure 3.1. **Implementation of MapReduce in Hadoop.** Hadoop uses a master-slave architecture. The instrumentation sources are white-box logs from the Hadoop daemons, and black-box logs from the nodes.

3.1 Target Systems

This section provides a brief description of the production Hadoop clusters, and the Voice-over-IP (VoIP) system at a major Internet Service Provider—highlighting the similarities between these two production systems that make them amenable to our diagnosis approach in Section 3.1.3.

3.1.1 Hadoop Clusters

Large-scale data processing is becoming increasingly common, and has been facilitated by frameworks such as Google's MapReduce [Dean and Ghemawat, 2008], which parallelizes and distributes jobs across large clusters. In particular, Hadoop, the open-source implementation of MapReduce, has been widely used for large-scale data-intensive tasks such as click-log mining, web crawling, image processing, and data analysis. Hadoop is widely used at companies such as Yahoo!, Facebook, and Fox Interactive Media, as well as for academic research [Apache Software Foundation, 2012]. Hadoop clusters process large amounts of data—for example, Facebook's Hadoop Distributed File System (HDFS) consists of more than 100 PB of physical disk space [Ryan, 2012].

Hadoop decomposes a massive job into smaller (Map and Reduce) tasks, and a massive data-set into smaller partitions (blocks) such that each task processes a different partition in parallel, as shown in Figure 3.1. The Map task executes a user-defined Map function

for each key/value pair in its input, while the Reduce task fetches, merges, and sorts the outputs from completed map tasks. Hadoop shares data amongst the distributed tasks in the system through the Hadoop Distributed File System (HDFS), an implementation of the Google File System [Ghemawat et al., 2003]. HDFS splits and stores files as fixed-size blocks (except for the last block).

Hadoop uses a master-slave architecture with 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 (*i.e.*, the executing tasks). Each slave node runs two daemons: 1) the TaskTracker that launches tasks on its local node, and tracks the progress of each Map or Reduce task on its node; and 2) the DataNode that serves data blocks (on its local disk) to HDFS clients.

We analyzed the white-box logs generated by these daemons on two production Hadoop clusters namely: Yahoo!'s M45 [Yahoo!, 2009] supercomputing cluster, and the OpenCloud cluster for data-intensive research [Parallel Data Lab, 2012]. Each node in the system also generates black-box logs which monitor resource-usage on the node. Section 3.2.2 provides a comprehensive analysis of these traces.

3.1.2 Business Voice-over-IP (VoIP) system

Consumers are increasingly using interconnected VoIP services in lieu of traditional telephone service. As of December 2010, 31 percent of the more than 87 million residential telephone subscriptions in the United States were provided by interconnected VoIP providers. In addition, approximately 31 percent of residential wireline 9-1-1 calls were made using VoIP services, making the availability of VoIP infrastructure critical [FCC, 2012]. The FCC specifies reporting requirements for major outages which are defined as outages that last more than 30 minutes and potentially affect E-911 services, or outages that affect at least 900,000 user minutes of interconnected VoIP service. There is no FCC reporting requirement for chronics. However, VoIP providers are concerned about chronics because unresolved chronics significantly degrade users' satisfaction with the service.

We investigated the characteristics of chronics in the context of a business VoIP system at a major US-based Internet Service Provider (ISP). The ISP's VoIP network that we analyzed handles tens of millions of calls each day, contains thousands of network elements, and is

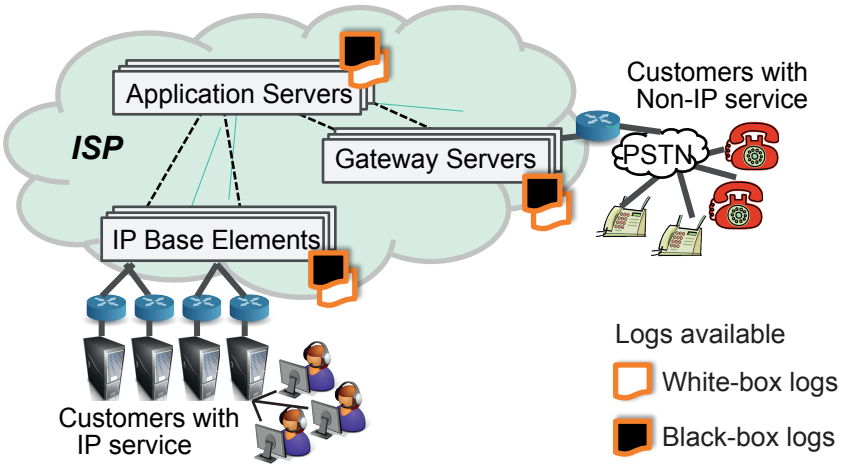


Figure 3.2. **An example of a VoIP system.** Customers connect to the network via the public-switched telephone network using phones or faxes, or using IP devices such as computers. A large number of network elements at the ISP route the calls from the source to their destination and execute application-specific logic.

layered on a large IP backbone. The network offers a portfolio of voice services including individual accounts, teleconferencing, self-managed solutions where users manage their own premise equipment, and wholesale users who buy network minutes in bulk and resell them. Each service has different call flow patterns, with calls going through combinations of network elements such as VoIP gateways (IPBEs), traditional phone gateways (GSXs), accounting servers, application servers (AS), voicemail servers, and policy servers (PSX). Many of these are built by different vendors and have different log file formats. Calls within the VoIP system may originate from, or be destined to circuit-switched networks (*e.g*, public switched telephone network (PSTN)) or packet-switched networks (*e.g*, VoIP services on personal computers) as shown in Figure 3.2.

Each network element in the VoIP system logs details of each call that passes through in white-box logs known as call-detail-records (CDRs). The network elements also track resource-usage in black-box logs. There are two types of CDRs namely: 1) `Success CDRs` generated when a call is answered, ring-no-answer, user-busy, or dialed-number-error; and, 2) `Defect CDRs` generated when a call fails during call setup (*i.e.*, `blocked call`), or when a call fails during data transfer after the connection is established (*i.e*, `cutoff calls`). Quality of Service in the VoIP system is measured in by the number of defects per million (DPM) calls.

Table 3.1. **Comparison of the target systems.** Summary of characteristics that make the target systems amenable to our diagnosis approach.

Property	**Hadoop**	**VoIP**
Workload	Batch jobs	Interactive calls
Scale	10s–100s of nodes	1000s of nodes
Labeled failure data	Partially available	Available
Application logs	Hadoop daemon logs	Call Detail Records (CDRs)
System logs	OS performance counters	OS performance counters
Causal flows	End-to-end task traces	End-to-end call traces
Peers	Tasks from same job	Calls from the same service

3.1.3 Comparison of Target Systems

Hadoop workloads are data-intensive and consist of batch jobs with diverse runtimes [Kavulya et al., 2010; Ren et al., 2012]. The scale of the two production Hadoop clusters we analyzed ranged from 10s–100s of nodes. Labeled failure data is partially available in the Hadoop clusters by mining the exceptions thrown whenever jobs or tasks abort. However, the diverse runtimes of Hadoop jobs makes it challenging to identify performance problems—necessitating the use of anomaly detection in our diagnosis approach. On the other hand, VoIP workloads consist of interactive calls. The production VoIP system that we analyzed comprised of 1000s of nodes. Labeled failure data was available in the VoIP system due to the existence well-established heuristics for identifying failed calls, *e.g.*, a user redialing the same number immediately after disconnection, or zero talk time might indicate failure.

Despite the differences in the Hadoop and VoIP systems, these systems share characteristics that make them amenable to our diagnosis approach as listed in Table 3.1, and elaborated below:

- Both systems are engineered for high-availability, thereby ensuring that they are working correctly for an overwhelming majority of users. However, due to the scale of these systems, there are almost always multiple ongoing problems of different types that affect small subsets of users or requests, *i.e.*, chronics.
- Multiple instrumentation sources are available in both systems. White-box logs, such as the daemon logs in Hadoop and the call-detail records (CDRs) in VoIP provide a

semantic-rich source of information. In addition, black-box logs, such as OS performance counters, provide insight on resource-usage.

- User requests are routed through the system using well-known patterns (*e.g.* MapReduce flows in Hadoop and call flow patterns in VoIP) that support the inference of end-to-end causal traces from the diverse white-box logs available in these production systems. These end-to-end traces capture the users' experience with the system by identifying successful and failed interactions. Our *t*op-down diagnosis approach exploits knowledge of these successful and failed interactions to localize the root-cause of the problem.
- Both systems support a notion of peers—making them amenable to a peer-comparison approach for detecting anomalous behavior in the absence of labeled failure data. In Hadoop, peers can be tasks from the same job, or slave nodes running similar workloads. In VoIP, peers can be calls from the same service, *e.g.*, conference calls and call-waiting.

3.2 Characterization of Hadoop Workloads

To gain insight on how to best to address the shortcomings of existing monitoring tools, we analyzed the characteristics of Hadoop jobs using traces collected from two production Hadoop clusters. The first set of traces comprised of 10-months of white- and black-box logs from the M45 [Yahoo!, 2009] supercomputing cluster, a production Hadoop system that Yahoo! administered and made freely available to select universities for academic research until August 2011. The M45 cluster had approximately 400 nodes, 4000 processors, 3 terabytes of memory, and 1.5 petabytes of disk space. The cluster ran Hadoop, and used Hadoop on Demand (HOD) to provision virtual Hadoop clusters over the large physical cluster. The second set of traces comprised of 8-months of white- and black-box logs from the OpenCloud cluster for data-intensive research. The OpenCloud cluster currently has 64 worker nodes, each with 8 cores, 16 GB DRAM, 4 1TB disks and 10GbE connectivity between nodes [Parallel Data Lab, 2012]. The compute cluster in OpenCloud, seen as a single computer, has over 1 tera-operations per second, over 1 TB memory, 256 TB of disk space, and over 40 Gbps bisection bandwidth.

This section provides background on Hadoop clusters by describing the users, existing monitoring tools, and the current diagnostic workflows of Hadoop users based on a user study conducted on the OpenCloud cluster (Section 3.2.1). This section also provides a comprehensive analysis of the performance characteristics of jobs and tasks executing in the Hadoop clusters (Section 3.2.2).

Users of Hadoop Clusters. The users of both clusters are researchers familiar with configuring Hadoop, writing and running MapReduce jobs, and analyzing the output generated by their jobs. These users run a diverse set of data-intensive workloads such as large-scale graph mining, text and web mining, natural language processing, machine translation problems, and data-intensive file system applications. Hadoop users troubleshoot their MapReduce jobs when they suspect that the job has been running longer than usual, or when their jobs fail due to exceptional conditions such as insufficient file privileges. Users also seek help from the system administrators when they are unable to solve problems on their own. The root-causes of these problems range from bugs in their MapReduce jobs (*e.g.* infinite loops), configuration problems such as allocating insufficient memory to complete their jobs, to the occasional disk failure as discussed in Section 3.3.

The system administrators are responsible for maintaining the overall health of each cluster by: 1) setting up hardware and software on the cluster; 2) troubleshooting infrastructural problems (*e.g.*, hardware failures, misconfigurations); 3) upgrading hardware and software; and 4) maintaining user accounts. The system administrators were less familiar with Hadoop, so the task of troubleshooting MapReduce problems was typically performed by Hadoop users.

Existing Monitoring Tools. Users primarily rely on three tools to diagnose problems in their jobs namely: 1) `Ganglia` [Ganglia, 2007], a scalable distributed system monitor tool for high-performance computing systems such as clusters and grids that allows users to remotely view live or historical statistics (such as CPU load averages or network utilization) as illustrated in Figure 3.3; 2) the Hadoop web interface [Apache Software Foundation, 2007] that allows users to browse white-box logs and monitor the progress of the map and reduce tasks that constitute their jobs, as illustrated in Figure 3.4; and 3) the operating system terminal that facilitates more powerful exploration such as querying the real-time resource usage (*e.g.*, CPU utilization), of nodes and processes, pinging remote nodes to

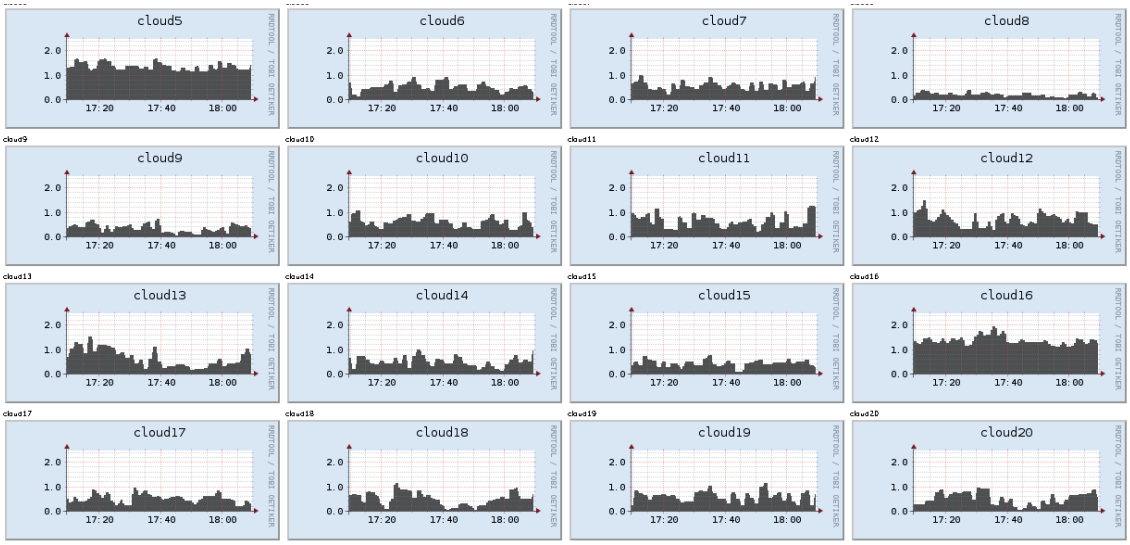


Figure 3.3. **Screenshot from Ganglia monitoring tool.** Each chart represents load on a node during a one hour period.

determine their status, `tail`-ing logs to track the real-time job progress, and dumping the process state of Hadoop jobs. In addition, system administrators use Nagios [Nagios Enterprises., 2008], an open-source application that monitors nodes and services, and alerts administrators when things go wrong. They also make use of an issue-tracking system (Request Tracker (RT) [Best Practical Solutions, 2013]) that allows them to keep track of both open and resolved problems.

3.2.1 Hadoop User Study

To better understand the diagnostic workflows of Hadoop users and identify shortcomings in the existing monitoring tools, we interviewed 3 Hadoop users and 2 system administrators of the OpenCloud cluster [Campbell et al., 2011]. The Hadoop users had varying levels of experience with Hadoop: one of the users was a novice user, while the other two were advanced users. Hadoop users try to diagnose problems on their own, by soliciting help from the cluster mailing list, or by escalating the problem to the system administrators. The users were interested in distinguishing between the following diagnostic scenarios:

1. **Bugs in their MapReduce jobs.** Users sometimes make mistakes when developing their MapReduce jobs, such as referencing incorrect files, assigning insufficient file

Job Name: random-writer
Job ID: job_20130504180930_0001
Status: Running
Started at: Sat May 4 09:30:17 EDT 2013
Running for: 9mins, 46sec

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
Map	65.00%	20	0	7	13	0	0 / 8
Reduce	0.00%	0	0	0	0	0	0 / 0

Figure 3.4. **Screenshot of the Hadoop web interface.** Screenshot shows progress of Map and Reduce tasks in a single job.

permissions, and software bugs such as infinite loops. Once they identify the bug, they can fix their code and re-run their jobs.

2. **Legitimately slow jobs.** Some tasks within their MapReduce jobs are legitimately slower than others because they are processing more data. Users can ignore these discrepancies in task durations, and allow their jobs to run to completion.
3. **Contention with other jobs.** As users share the cluster, sometimes buggy MapReduce jobs can degrade the performance of other jobs running in the cluster. For example, infinite loops in tasks or jobs which inadvertently fill up the temporary directories on nodes can interfere with other jobs in the system. Users can contact the owners of these buggy jobs or system administrators to resolve these issues.
4. **Infrastructural problems:** Users can escalate problems to the system administrators especially if they suspect an infrastructural problem, *e.g.*, disk failures, insufficient disk space or a cluster-wide misconfiguration.

Methodology of User Study

We used a human-centered design methodology [Beyer and Holtzblatt, 1997] to explore the diagnostic workflows of Hadoop users. Our aim was to identify common workflows in their day-to-day tasks, and identify breakdowns and opportunities for improving the existing diagnostic interfaces. To understand the diagnostic workflows of these users, we conducted contextual inquiries [Beyer and Holtzblatt, 1997]. Contextual inquiry is a human-centered research method in which researchers observe users performing their work in context, and

interrupt briefly to ask questions as they arise. We believe that *what the user says he does, and what he actually does can be very different things.* By observing users and their work flows, we gained a more concrete understanding of their everyday tasks and goals. Through our contextual inquiries, we identified common workflows of Hadoop users when diagnosing problems in their jobs. In what follows, we describe cases that illustrate important aspects of the diagnostic workflow, focusing on breakdowns because these are areas that require the most immediate attention.

Case 1: Misleading application bug. Bob[1] begins a Hadoop job from the terminal. He monitors the job progress from the terminal window. *"I like to monitor progress from the terminal. If there is a problem, I can see it faster."* After receiving sporadic feedback about job progress, Bob switches windows to the Hadoop web interface and refreshes the page. *"I think I'm competing with another job"*. He sees that another job is being run by a different user and restarts his job to test his theory. To do this, he must scroll through the job output on his terminal to find the kill command, then copy and paste the command in the terminal. *"If the job is very long, you have to go back several screens to get it."*

After modifying his code, Bob attempts the job again, but this time the job fails. The terminal tells him that the job failed, but it does not contain sufficient information to diagnose the problem. Bob returns to the web interface to gather more information and refreshes the job page again. He determines there is a problem by noticing that the number of failed tasks on his job is greater than zero. *"If this number is non-zero, basically there is something wrong with your program. If the number is low, two or three, the node may be unstable."* He then clicks on the number of failed tasks in the web interface to view the error messages, but he must navigate to another page to access a full error log. *"It's kind of painful to do this"*. From this he determines the cause to be due to a syntax error in his code.

Case 2: Overloaded node. Jeff[2] retrieves an email message posted to the cluster mailing list requesting assistance with troubleshooting a job that is failing. Jeff examines the failed job in the Hadoop web interface. He selects the job from the job list and subsequently clicks on the number of killed tasks from the job summary page. He observes the list of killed tasks and identifies the node on which the task was killed, *"ah, this was the machine"*, which is labeled as node 13, *"sometimes it's not that obvious"*. Jeff changes browser tabs to the Ganglia

[1]Not his real name.

[2]Not his real name.

web interface where the individual cluster nodes are monitored. He scrolls through the list of nodes to find node 13, which he identified as being troublesome in the Hadoop interface and selects it. He explains, *"Sometimes it's not just your job that caused the problem, sometimes someone else might cause you a problem"*. The resulting page displays hourly plots of all the metrics collected from node 13. He glances at the first few graphs, *"these are more likely to be important"*, he quickly scrolls the rest of the page, *"this is too much detail, there is too much data to look at"*. Back at the summarized node 13 graphs, he synthesizes the information from multiple graphs to determine that the node was under heavy load during the period over which his task ran.

Shortcomings in Existing Monitoring Tools

The contextual inquiries revealed several shortcomings in diagnostic interfaces of the monitoring tools available on the Hadoop cluster namely:

- **No single point for diagnosis-relevant information.** We observed that users spent an inordinate amount of time locating key pieces of data spread across multiple systems. Users often juggled between multiple browser tab views to obtain the desired information.
- **Lack of information prioritization.** In many cases, important data needed by the user is not displayed prominently and conveniently. The user often must go through several levels of navigation to find needed information.
- **Information overload.** Another major shortcoming we observed was the copious amount of information displayed to the user due to the large number of nodes in the cluster, and the approximately hundred different performance metrics collected from each node. Users were presented with stacks of graphs that commingled meaningful information with irrelevant data as shown in Figure 3.3.
- **Users unaware of the computational cost of their jobs.** The Hadoop users in the cluster had no way of knowing how effectively they were managing resources on the Hadoop cluster. They were unable to determine whether variations in task performance were due to legitimate issues (*e.g.*, a task processing a large chunk of data), due to application bugs, or due to infrastructural problems in the cluster.

Table 3.2. **Summary of M45 and Opencloud Hadoop job traces.** Over 90% of jobs completed successfully on both clusters. The job failure rate was slightly higher on OpenCloud than M45.

	M45			**OpenCloud**		
Log period	10 months			8 months		
Number of active users	33			57		
Job Status						
Successful jobs	160,278 (97.0%)			59,683 (94.3%)		
Failed jobs	4,874 (3.0%)			3,627(5.7%)		
Job statistics	*Mean*	*S. dev.*	*Max.*	*Mean*	*S. dev.*	*Max.*
Maps per job	121	528	38,927	303	2,134	143,949
Reduces per job	18	142	24,000	26	68	5,000
Nodes per job	25.6	22.4	299	30	22	64
Job duration	18mins	208mins	7days	7mins	107mins	13days
HDFS statistics	*Mean*	*S. dev.*	*Max.*	*Mean*	*S. dev.*	*Max.*
HDFS bytes read	2.8GB	28.7GB	2.4TB	12GB	476GB	85TB
HDFS bytes written	1.8GB	38.6GB	11.6TB	2GB	28GB	3TB

3.2.2 Characteristics of Hadoop Jobs

The first step towards understanding how to address the shortcomings in existing tools involved analyzing the performance characteristics of Hadoop jobs. To gain insight on these performance characteristics, we analyzed 10-months of white-box logs from the M45 [Yahoo!, 2009] supercomputing cluster, and 8-months of white-box logs from the Open-Cloud [Parallel Data Lab, 2012] cluster (see Table 3.2). The white-box logs contained information about the Map and Reduce tasks executed by each job, *e.g.*, task duration, status, and the volume of data read and written to the Hadoop Distributed File System (HDFS). The logs also recorded the completion status of each job. We categorized the completion status of jobs as:

- Successful jobs.
- Failed jobs which were aborted by the JobTracker due to unhandled exceptions, or aborted by the user.

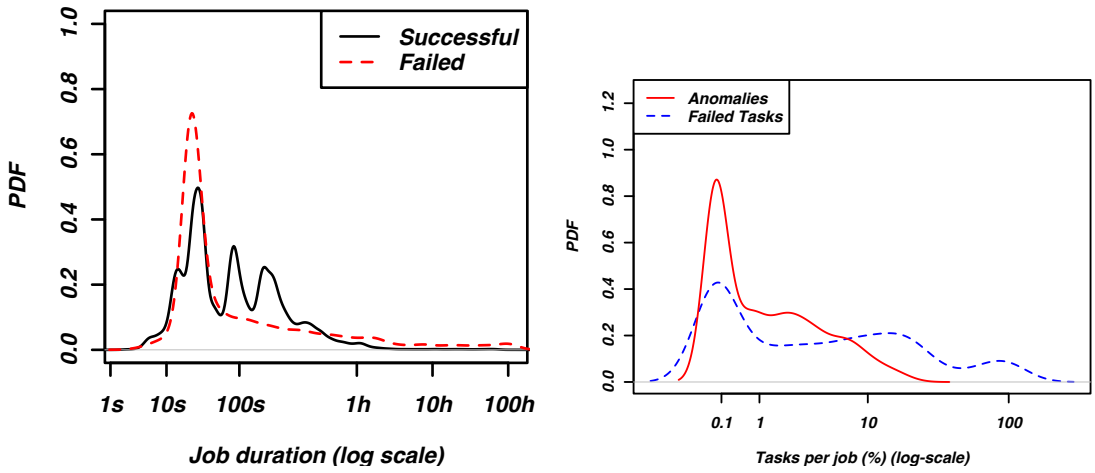
Over 90% of jobs completed successfully on both clusters as shown in Table 3.2. The percentage of failed jobs on OpenCloud is higher than that on M45—we hypothesized that this increased failure rate could be due to novice users and configuration problems on OpenCloud. Jobs on OpenCloud, on average, instantiated 303 Maps per job compared to 121 Maps per job on M45. The larger mean number of Maps per job on OpenCloud can be attributed to the larger mean number of bytes read from HDFS (12GB on OpenCloud compared to 2.8GB on M45). The differences in the mean number of Reduces per job and the data written by these Reduces to HDFS was less pronounced across both clusters.

Most jobs on both clusters utilized only a subset of available nodes. For example, on M45 the mean number of nodes utilized per job was 25.6 while the total number of available nodes was 400. Despite larger data input sizes, jobs completed faster on OpenCloud with a mean duration of 7 minutes than on M45 where the mean duration was 18 minutes as shown in Table 3.2. We hypothesize that the differences in the mean job duration could be due to the diversity of jobs running on the clusters, and higher resource contention on M45 which experienced a higher volume of jobs.

The performance characteristics of Hadoop jobs and tasks that we explore are:

1. **Variance in job durations.** We analyze the performance characteristics of Hadoop jobs by quantifying the variance in job durations.
2. **Variance in error latencies.** We analyze the amount of time it typically takes for Hadoop jobs to fail from the time the first exception is thrown. Long error latencies imply that Hadoop could benefit from better diagnosis techniques.
3. **Variance in task durations.** We analyze the performance characteristics of tasks within individual Hadoop jobs by quantifying the variance in the task durations.

An understanding of variance is important because research has shown that high variance in metrics can degrade the performance of diagnosis tools [Sambasivan and Ganger, 2012b]. Quantifying the amount of variance in Hadoop jobs and tasks guided our efforts to reduce variance during anomaly detection (Chapter 5). We estimated the degree of variance in Hadoop jobs and tasks by computing the mean, standard deviation, and the coefficient of variation (CV) for job and task durations. The coefficient of variation is the ratio of the standard deviation to the mean. Distributions with CV $<= 1$ have a low-variance, while those with CV > 1 have high-variance [Bendel et al., 1989].



(a) M45 cluster (b) OpenCloud cluster

Figure 3.5. **Probability density function of Hadoop job durations.** The job durations on Hadoop follow a multi-modal distribution—the median job duration was ≈1 minute, however some jobs lasted several days.

Table 3.3. **Variance in Hadoop job durations (in minutes).** SD - Standard deviation, CV - Coefficient of Variation, 99^{th} Perc. - 99^{th} percentile.

Cluster	Job status	Mean	SD	CV	Median	99^{th} Perc.
M45	Successful	13.6	159.7	11.7	1.2	89.5
	Failed	112.6	617.2	5.5	0.4	5,213.3
OpenCloud	Successful	5.2	48.8	9.3	1.4	39.5
	Failed	33.0	265.7	8.0	1.8	833.7

Problem 1: Quantifying variance in job durations. Given $d_1, d_2, ..., d_n$ durations of j_i jobs in each Hadoop cluster, quantify the variance in job durations. Figure 3.5 shows that the job durations on both clusters exhibit multi-modal behavior, and follow a heavy-tailed distribution where most jobs were short-lived with a median job duration of about 1 minute on both clusters (see Table 3.3). However, the longest job durations observed lasted several days. Heavy-tailed job durations have also been observed in other MapReduce clusters [Isard et al., 2009; Ren et al., 2012], prompting the development of fair job schedulers, such as the Hadoop capacity scheduler [Yahoo!, 2008] and the Quincy fair scheduler[Isard et al., 2009], which prevent large jobs or heavy users from monopolizing the cluster. We estimated the degree of variance in Hadoop jobs by computing the mean, standard deviation, and the

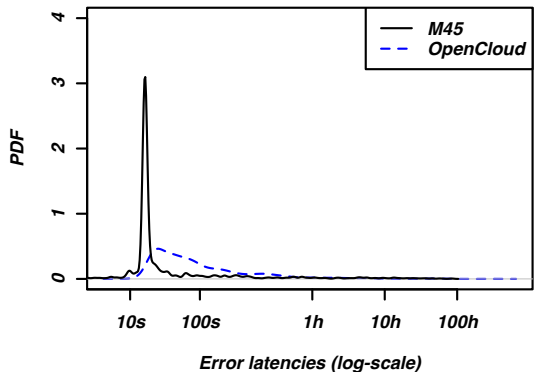


Figure 3.6. **Error latencies in Hadoop jobs.** Most jobs fail within 5 minutes after the first unrecoverable task aborts. However, we observed a maximum error latency of 3.62 days on OpenCloud.

Table 3.4. **Variance in Hadoop error latencies (in minutes).** SD - Standard deviation, CV - Coefficient of Variation, 99^{th} Perc. - 99^{th} percentile.

Cluster	Job status	Mean	SD	CV	Median	99^{th} Perc.
M45	Failed	43.3	293.5	6.8	0.3	1,241.3
OpenCloud	Failed	33.0	265.7	8.0	1.8	833.7

coefficient of variation (CV) for the job durations (see Table 3.3). The coefficient of variation for job durations on both clusters was greater than 1 indicating high variance.

Problem 2: Quantifying variance in error latencies. Given $l_1, l_2, ..., l_n$ error latencies of j_i failed jobs in each Hadoop cluster, quantify the variance in job durations.

We estimated the error latency for failed jobs *i.e.*, the time that elapsed from the first unrecoverable exception in a task to the time the JobTracker aborts the job. Figure 3.6 shows that most jobs fail within 2 minutes of the first aborted task. The error latencies on OpenCloud tended to be larger than those on M45 as shown by the median latencies in Table 3.4. The coefficient of variation in both clusters was greater than 1 indicating high variance. The maximum error latency observed was 3.62 days. Therefore, even jobs that eventually failed due to unrecoverable exceptions could benefit from better diagnosis techniques.

Problem 3: Quantifying variance in task durations. Given $t_1, t_2, ..., t_n$ task durations in each individual job, j_i, quantify the variance in task durations in each job.

Table 3.3 shows that Hadoop clusters experience high variance across jobs (*i.e.*, at a macro-level) because users run diverse jobs with different performance characteristics. We

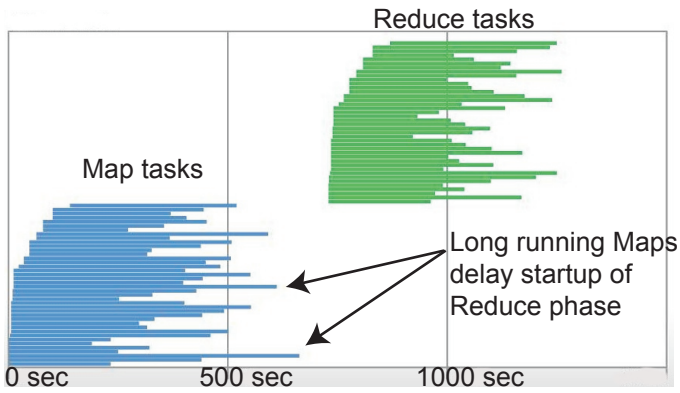


Figure 3.7. **Swimlane graph charting progress of tasks in Hadoop job.** The graph charts the start and end times, and durations of Map and Reduce tasks for a single job—highlighting the inherent structure of MapReduce jobs with Map tasks completing before Reduce tasks.

Table 3.5. **Variance in Hadoop task durations.** Prevalence of high variance is computed by the percentage of jobs with a coefficient of variation > 1 during the Map or Reduce phase.

Indicator of High Variance	**Percentage of jobs with high-variance**		
(Coefficient of Variation > 1)	**Successful jobs**	**Failed jobs**	**Canceled jobs**
M45			
Map phase	10.8%	6.7%	15.7%
Reduce phase	30.8%	15.4%	9.6%
OpenCloud			
Map phase	11.1%	13.7%	25.5%
Reduce phase	40.6%	33.8%	57.8%

investigated whether individual jobs experience variance in the durations of Map and Reduce tasks (*i.e.*, at the micro-level). A low amount of variance would indicate that *peer-comparison* would be a feasible strategy for anomaly detection in Hadoop. We term the property of low variance in task durations within the same job as the *equity of task durations*. The equity of task durations is important in optimizing the performance of parallel programs such as Hadoop. Intuitively, parallel programs perform optimally when the different threads (a term we use loosely to refer to independently executing parts) of the program executing in parallel complete in the same amount of time; otherwise, the runtime of the program can be reduced by redistributing work from threads taking longer to complete to work taking less time to complete. Alternatively, high variation in task durations within

a job can be seen as an indication of inefficiency or performance problems (if each task has the same amount of work), or as an indication that the job at hand intrinsically consists of barriers to its parallelization.

Visually, variance in task durations can be observed using the Swimlanes plots of MapReduce job behavior [Tan et al., 2009]. The Swimlanes visualization plots the executions of Map and Reduce tasks of one (or more) MapReduce job(s). Figure 3.7 shows a plot of a typical MapReduce job: the horizontal axis shows the time elapsed since the start of the job. For each (Map or Reduce) task, a horizontal line is plotted to indicate the duration of the job during which it was executing. The executions of Map and Reduce tasks are plotted using different colors. Tasks are plotted in the order in which they were executed during the job. *S*traggler Map or Reduce tasks, which take much more time than other tasks to complete, slow the completion of jobs, and are easily seen visually from our plots. These stragglers would also typically render the durations of tasks in the job less equitable, so studying the variance in task durations is also instructive in indirectly (but scalably, for large datasets) for identifying the possible existence of straggler tasks.

We measured the variance in task durations by computing the coefficient of variation for Map and Reduce tasks for each job in the Hadoop traces. Table 3.5 highlights the prevalence of high variance by computing the percentage of jobs with a coefficient of variation > 1 during the Map or Reduce phase. We observed that most successful jobs exhibited low variance for both Map and Reduce task durations—indicating that the durations of Map tasks (and Reduce tasks) were comparable in most jobs, with more than 90% of successful jobs with coefficients of variation $<= 1$ for Map durations, and with more than 60% of jobs with coefficients of variation $<= 1$ for Reduce durations. However, we also observed that there were jobs with high variance. The high variance could be due to legitimate issues (*e.g.*, tasks processing more data), or due to performance problems (*e.g*, resource contention). A diagnostic approach needs to factor out legitimate sources of variance when identifying anomalous tasks (see Chapter 5).

3.3 Prevalence of Chronics

This section presents empirical evidence of chronics in Hadoop clusters, and anecdotal evidence of chronics in VoIP systems. This section also explores the characteristics of chronics

Table 3.6. **Prevalence of problems in OpenCloud issue tracker.** Chronics (*i.e.*, problems that affect a subset of users or jobs) are more prevalent than major outages.

	Major Outage	Chronics
Total problems	30 (21.6%)	109 (78.4%)
Problem characteristics		
Single, independent	0 (0%)	59 (54.1%)
Multiple, independent	0 (0%)	15 (13.8%)
Correlated	30 (100%)	35 (32.1%)
Identified cause		
Hardware problem	4 (13.3%)	69 (63.3%)
Hard drive	0	59
Network	4	4
Power	0	4
CPU/memory	0	2
Configuration	15 (50.0%)	21 (19.3%)
Disk full	11	14
Out of memory	4	7
Software bug	1 (3.3%)	4 (3.7%)
Unknown	10 (33.3%)	15 (13.7%)

in production systems.

Evidence of Chronics in Hadoop

We estimated the prevalence of chronics in Hadoop clusters using one year's worth of data from OpenCloud's issue tracking system, and hardware replacement logs. OpenCloud relies on Request Tracker (RT) [Best Practical Solutions, 2013] to track issues and coordinate tasks on the cluster. Users submit problems to the issue tracker using the mailing list. The submitted problems are assigned to the system administrators who troubleshoot the problems by manually analyzing white-box and black-box logs. Once the system administrators identify the root-cause and resolve the problem, they record the resolution in the issue tracker. The system administrators also keep track of failed hardware components

Table 3.7. **Summary of M45 and Opencloud Hadoop job traces.**

Application-level bug	**1439 (53%)**
Command failed	371
Missing or mismatched class	216
Index out of bounds	202
Number format exception	185
Null pointer exception	136
Type mismatch	76
Other	253
Configuration	**910 (34%)**
Missing file	341
Permission problem	266
Out of memory	227
Disk full	76
Unknown	357 (13%)

and their replacements. We analyzed 139 resolved problems on the OpenCloud cluster by collating reported incidents in the issue tracker and hardware replacement logs. Our analysis did not include incidents due to scheduled downtime such as upgrading cluster software.

Table 3.6 shows the prevalence of problems in OpenCloud's Hadoop cluster. We categorized major outages as problems that impacted the majority of jobs running on the cluster. Major outages occurred due to failures in centralized Hadoop components namely: the JobTracker which schedules jobs on the cluster, the NameNode which manages the namespace for the Hadoop Distributed File System (HDFS), and the client which serves as the gateway into the cluster. Major outages on the cluster were primarily due to exhaustion of memory or disk space on these centralized nodes. We categorized chronics as problems that impacted a small subset of users or jobs. For example, a software bug in a user's MapReduce application that primarily affects jobs belonging that user, or a hardware problem such as a bad disk that degrades the performance of the subset of a job's tasks that are running on the affected node. In addition, Hadoop's fault-tolerance mechanisms attempt to mask

problems by rescheduling affected tasks on different nodes. We observed that chronics were more prevalent than major outages on the OpenCloud cluster—accounting for 78.4% of reported problems. Chronics in the cluster were mostly due to single, independent problems such as bad disks. Occasionally, the chronics were attributed to multiple, independent problems. For example, one incident was attributed to two faulty disks, and a faulty network interface card—each on a different node. This incident took several weeks to resolve. Correlated problems arose due to misconfigurations or software bugs in a user's job, *e.g.*, allocation of insufficient memory results in correlated task failures in a user's job.

We observed that only a small proportion of the problems reported to the system administrators involved software bugs. Most users attempted to first debug problems in their jobs before reporting issues to the system administrators via the mailing list. Table 3.7 highlights the root-causes of failed jobs on the OpenCloud cluster based on mining exceptions thrown when the JobTracker aborts a job due to an unrecoverable error. We identified the root-causes of the 2,706 failed OpenCloud jobs described in Table 3.2 by manually analyzing the classes of exceptions (*e.g.*, null pointer exceptions which indicated application-level bugs). Table 3.7 shows that 53% of jobs failed due to application-level bugs such as failed command-line scripts and mismatched classes. Misconfigurations and environmental problems such as referencing non-existent files or allocating insufficient memory accounted for 34% of failed jobs. We tried to ascribe the remaining 13% of job failures whose root-causes were unknown to hardware problems reported in the issue tracker and hardware replacement logs. However, due to the time lag between the job failure and the reporting of the incident to the issue tracker, we were only able manually ascribe 17 incidents to job failures. Therefore, we could not obtained a detailed breakdown of root-causes for 13% of failed jobs.

3.4 Anecdotal Evidence of Chronics in VoIP

Due to confidentiality concerns, we present anecdotal (rather than empirical) evidence of chronics in the production VoIP system. The VoIP system has real-time operations teams that ensure high-availability by monitoring both low-level alarms derived from the equipment, as well as end-to-end indicators such as user complaints and output from automated test call systems. These operations teams rely on codebook-based systems [Yemini et al.,

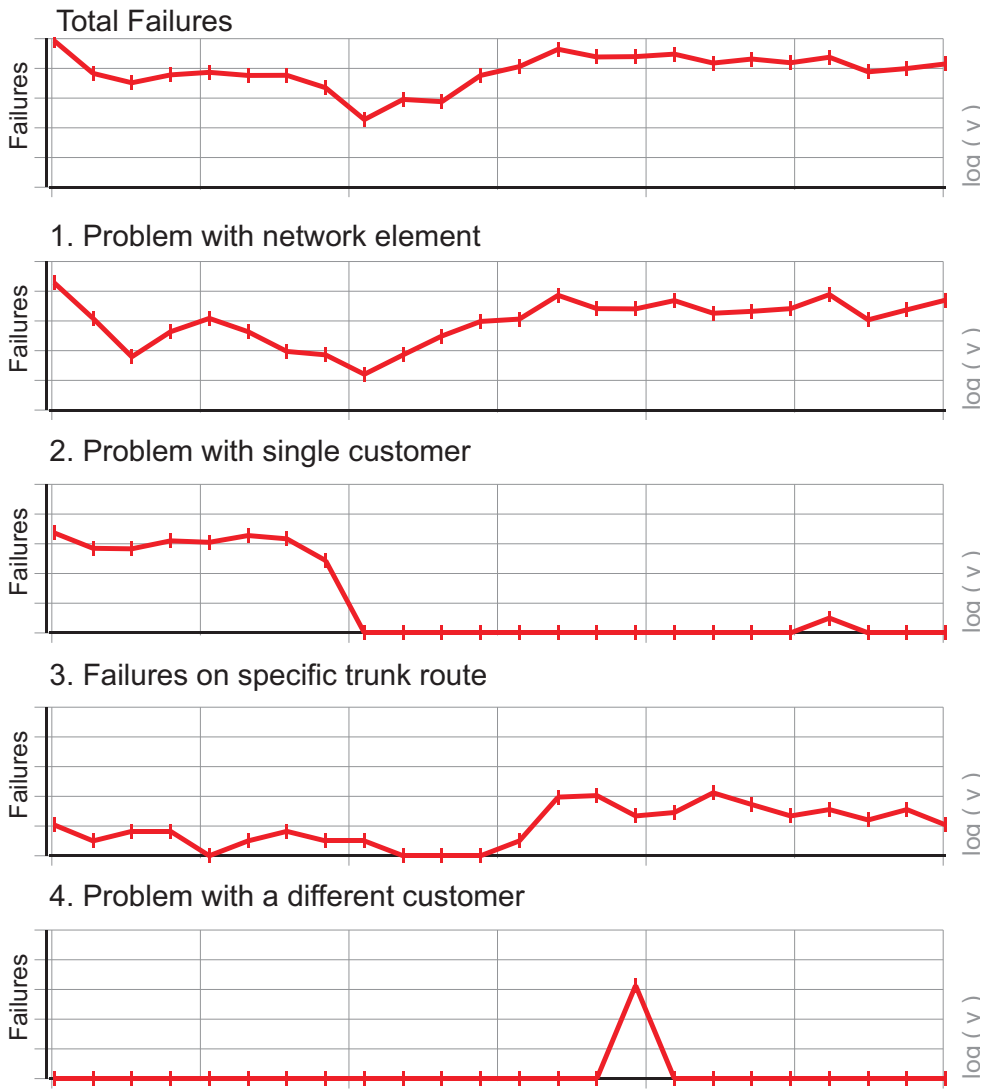


Figure 3.8. **Multiple chronics present in a single network element in VoIP system.** Defects associated with a network element may be due to many different causes.

1996] driven by signatures of known problems for diagnosis. Major outages often result in immediate impact on successful call volumes, alarms from many sources, and are usually detected and resolved quickly. For example, data we obtained from the Voice over IP (VoIP) platform of a major ISP revealed that even in the worst month for major outages, the number of calls affected (dropped or blocked) due to major outages was only 30% higher than the number of calls impacted by chronics.

Despite such robust operations support, the system always has a number of call defects occurring at any time of the day in the form of "background noise" which they refer to as chronics. The causes of these chronics were many, ranging from network elements that need

to be reset or rebooted, to protocol compatibility issues for corner cases, to configuration problems for individual customers. Measured in defects per million (DPM), they represent only a small fraction of the calls at any given time, but left unchecked, they can add up quickly over weeks and months. A separate operations team troubleshoots these defects, but diagnosis was still a largely manual process prior to the deployment of our diagnosis tool.

Figure 3.8 shows an actual example where the total number of defective calls passing through a network element (top graph) over a period of 3 days were in fact due to at-least four unique problems—one related to a network element, two related to two different customers (problems 2 and 4), and one a capacity problem with a trunk route (problem 3). The y-axis shows the number of failed calls due to each problem on a log scale as a function of time. The most dominant problem (problem 1) drives the shape of the overall failure graph, and hides the other problems. Some problems, such as problem 4 from Figure 3.8 occur only for short durations of time, and could be discovered by change detection algorithms. However, problems such as problem 1 persisted for long periods of time, thus making it difficult to detect them via change points.

Chronics may also be triggered by some unforeseen corner case requiring atypical conditions. An incident from the VoIP network illustrates this issue. Customers of a given service experienced difficulties making and receiving calls following a planned maintenance involving a configuration change to a server. The issue prevented customers whose phones used IP addresses instead of fully qualified domain names from registering with the network. To effectively debug this problem, operators needed to identify that the problem occurred only for customers of the specific service when certain types of phones were used with the misconfigured server. Identifying the combination of factors necessary to trigger the problem is challenging—motivating the need for automated diagnosis tools for localizing the source of chronics.

3.5 Summary

This chapter described the performance characteristics of Hadoop workloads, and investigated the prevalence of chronics in the context of two production systems namely: academic cloud-computing clusters running the Hadoop parallel processing framework

Table 3.8. Summary of Findings

Finding	Implications on diagnosis
Workload characterization	
I. Hadoop jobs exhibit multi-modal behavior, and large variance in job durations where short jobs are occasionally interspersed with long-running jobs. *(Figure 3.5 and Table 3.3)*	Anomaly detection can be challenging due to the diverse job durations.
II. Most Hadoop jobs exhibit low variance across Map and Reduce tasks. *(Table 3.5)*.	Peer-comparison is a feasible strategy for anomaly detection.
III. High task variance could be due to legitimate issues (*e.g.*, varying input), or due to performance problems (*e.g*, resource contention) *(Table 3.5)*.	Anomaly detection approach needs to factor out legitimate sources of variance when identifying anomalous tasks.
Prevalence of chronics	
I. Chronics are more prevalent than major outages in production systems *(Section 3.3)*.	*Top-down statistical approach* needed to drill down from user-visible symptoms of problems to identify source of problem.
II. Chronics arise due to various fault scenarios ranging from single, independent faults to complex component interactions *(Section 3.3)*.	Diagnostic approach needs to distinguish between these fault scenarios.

[Apache Software Foundation, 2007], and a Voice-over-Internet-Protocol (VoIP) system at a major Internet Service Provider. We summarized the implications of our workload characterization on anomaly-detection and problem localization in Table 3.8.

First, the comprehensive analysis of Hadoop workloads showed that Hadoop jobs experience high-variance at a macro-level (*i.e.*, across jobs) due to diverse runtimes. These diverse job runtimes make the detection of performance problems challenging because we cannot rely on time-based service-level objectives (SLOs) used in multi-tier web applications [Cohen et al., 2005; Chen et al., 2002; Aguilera et al., 2003] where there are well-accepted standards for defining slow web requests (*e.g.*, 99% of requests should complete within 4 seconds). However, at the micro-level (*i.e*, tasks within individual jobs), the majority of Hadoop jobs exhibit low-variance across Map and Reduce tasks. The low-variance within jobs suggests that peer-comparison is a feasible strategy for anomaly-detection. We also observed that some individual jobs experience high-variance in Map and Reduce du-

rations. The challenge is how to differentiate between legitimate sources of variance (*e.g.*, tasks processing more data) from illegitimate sources of variance (*e.g*, resource contention). An anomaly-detection approach that relies on peer-comparison would need to factor out legitimate sources of variance when identifying anomalous tasks.

Both empirical and anecdotal evidence revealed that chronics more prevalent than major outages in production systems. The root-causes of the chronics ranged from software bugs to resource-contention. Not all of these problems would trigger alarms at the node-level. Therefore, to detect these problems we adopted *top-down statistical approach* that drills down from user-visible symptoms of problems to localize the source of the problem. We also observed that due to the scale and distributed nature of production systems, complex interactions between components can result in complex failure modes, such as 1) correlated failures where problems whose root-cause is a single node manifest at multiple nodes due to internode communication; 2) complex triggers where a combination of factors have to go wrong to trigger the problem; and 3) multiple independent failures. A diagnostic approach needs to distinguish between these failure modes.

> It's no longer enough to monitor the up-versus-down status of thousands of discreet elements—you need a top-down view to monitor the performance of your applications.
>
> ExtraHop.com 2011

Chapter 4

White-box Analysis

TRADITIONAL approaches for diagnosis typically rely on a *bottom-up* approach [Yemini et al., 1996; Mahimkar et al., 2009; Oliner et al., 2010] that localizes problems by correlating low-level alarms (such as resource utilization indicators or network packet loss) across components in a production system. However, these alarm-correlation approaches fall short when diagnosing chronics because they fail to provide the necessary application-level visibility to detect chronics effectively. These tools are also prone to false positives, and tend to flood the operations team with alarms that do not correlate to end-user issues [ExtraHop.com, 2011]. Additionally, false negatives occur when the chronics are not be severe enough to trigger alarms—in these instances, the only indications of a problem are customer complaints which are often not specific and difficult to decipher.

We addressed the shortcomings in traditional alarm-correlation tools by adopting a *top-down* statistical approach that localizes the root-cause of chronics by drilling down from user-visible symptoms (*i.e.*, slow or failed user interactions) to the source of the problem. The first step of our approach involves inferring end-to-end causal flows from the white-box logs available in the production system. These end-to-end flows capture the control path and data demands of application requests as they are serviced across components and machines in the production system. Existing frameworks for automatically generating end-to-end flows either rely on source-code instrumentation, or black-box monitoring of messages exchanged between components. [Barham et al., 2004; Thereska et al., 2006; Sigelman et al., 2010] used source-code instrumentation in the application and middleware layers to track requests. However, production systems often comprise of proprietary components that do not support source-code instrumentation. Black-box monitoring of

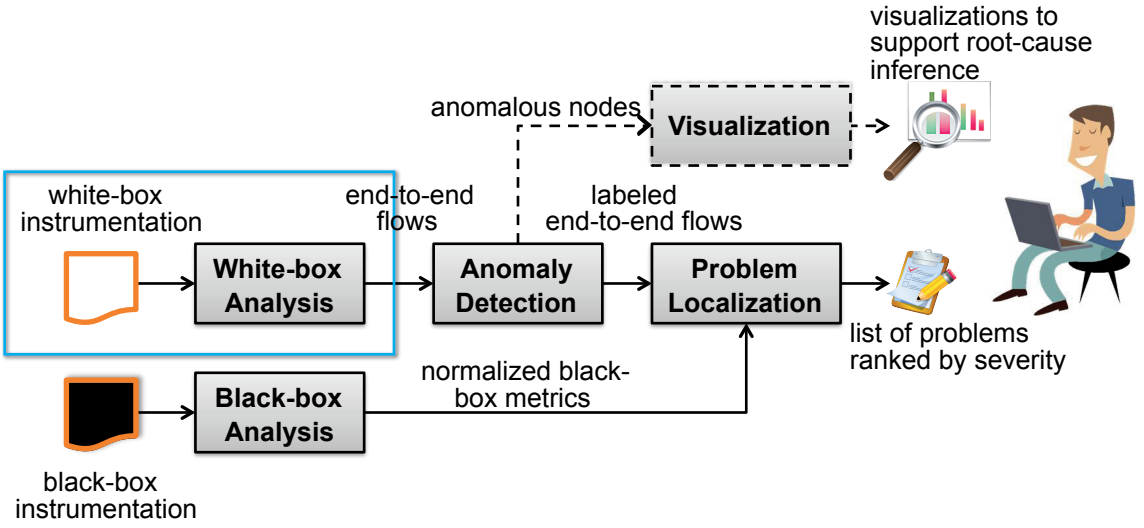


Figure 4.1. **Overview of white-box analysis.** During white-box analysis, the white-box (*i.e.*, application) logs are analyzed to infer end-to-end flows that capture system dependencies.

messages exchanged by components can be used to infer causal flows in proprietary components without relying on knowledge of node internals or message semantics [Aguilera et al., 2003]. But simply examining the black-box messages does not provide sufficient insight into the application-level behavior. [Attariyan et al., 2012] automatically generates request flows by instrumenting binaries as stand-alone applications execute—they do not support tracing across a distributed system.

We developed a log-analysis framework for use in production systems that infers end-to-end causal flows from the unmodified white-box (See Figure 4.1). The use of unmodified logs makes the framework better suited for production systems where we may not have the luxury of modifying existing instrumentation. The white-box logs provide our framework with semantic information on the runtime behavior of the application. The log-analysis framework infers the end-to-end causal flows by extracting control flows and data flows from the white-box logs. Control flows capture the sequence of events executed when servicing a user's request, while data flows capture the transfer of data between components.

Research questions The research questions that we asked when performing white-box analysis were:

- How do we extract local control-flow and data-flow information from the unmodified white-box logs?

- How do we infer dependencies with other components?
- How do we deal with missing dependency information in the logs?

Challenges. The challenges that we faced when performing white-box analysis were:

- **Diverse log formats.** Production systems comprise of different components, supplied by different vendors, each with different log formats—extracting meaningful information requires an understanding of these log formats.
- **Lack of global request identifiers.** The lack of global request identifiers that are propagated across the system complicates the inference of end-to-end flows. In VoIP systems, different components may use different formats to record key attributes (*e.g.*, some components log whole phone numbers while others log partial phone numbers). While in Hadoop, the TaskTracker logs do not record the unique identifiers of the data blocks transferred from the Hadoop Distributed File System (HDFS).
- **Scale.** Production systems can consist of hundreds (or thousands) of components. Each component can generate a large volume of monitoring data depending on the number of attributes tracked and the number of requests processed.

Design Overview. We addressed these challenges by developing an extensible log-analysis framework for inferring end-to-end causal flows. The framework examines the unmodified white-box logs to *trace control-flow and data-flow execution in a distributed system*, and to *derive state-machine-like views of the system's execution*. The framework extends SALSA's state-machine abstraction [Tan et al., 2008] for log-analysis by supporting a configurable environment for describing log files and expressing rules for generating end-to-end flows. User-defined configuration files specify how to stitch together the end-to-end flows based on the sequence of states extracted from locally-generated component logs. Since log data is only as accurate as the programmer who implemented the logging points in the system, we can only infer the state-machines that execute within the target system. We cannot (from the logs), and do not, attempt to verify whether our derived state-machines faithfully capture the actual ones executing within the system. Instead, we leverage these derived state-machines to support different kinds of analyses namely: to detect anomalous flows (Chapter 5), to localize problems (Chapter 6), and to visualize the system's execution (Chapter 9).

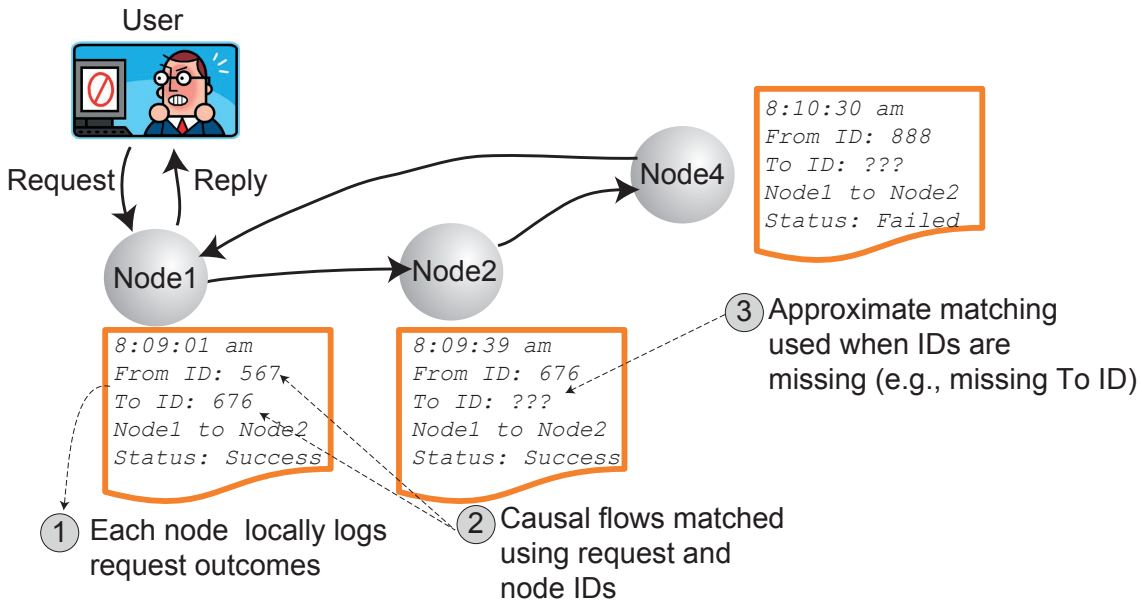


Figure 4.2. **Inferring end-to-end causal flows.** Our log-analysis framework infers end-to-end flows from locally-generated component logs based on approximate-matching of request and node IDs across components.

Figure 4.2 provides an overview of how we generate the end-to-end flows for a hypothetical request. Each component in the production system generates a log entry for each event processed. This log entry comprises of a timestamp, the type of event (*e.g.*, Map or Reduce task in Hadoop, or service type in VoIP), the IP address the sending and receiving nodes, and the event duration and status. Each event typically has a unique identifier associated with it, *e.g.* a task ID in Hadoop or the sender's and recipient's phone numbers in VoIP. Some log entries track the unique identifier for the next event in the sequence. For example, Hadoop logs explicitly specify the association between individual Map and Reduce tasks. Our log-analysis framework uses these associations to infer the state-machines that execute in the system. To cater for the log entries that do not explicitly state the identifiers for the next event in the sequence, we rely on *approximate-matching* based on domain-specific knowledge about relationships between events. For example, correlating block reads with Map tasks that occurred during the same time interval, and that accessed the same source and destination hosts. Each log entry may also contain additional descriptive attributes, such as the volume of data read or written during each event.

Our log-analysis framework relies on *domain-specific log-parsers* that use regular expressions to extract attributes of interest from the logs. One limitation of these domain-specific

log parsers is their reliance on hard-coded regular expressions that vary based on the format of log messages. The parsers need to be manually updated if the underlying log format changes due to a software upgrade. Despite this limitation, we opted for this approach because we lacked visibility into the proprietary components in the production systems.

Non-Goals. We do not seek to validate or improve the accuracy or the completeness of the logs, nor to validate our derived state-machines against the actual ones of the target system. Rather, our focus has been on the analyses that we can perform on the logs in their existing form. In addition, the current implementation of our log-analysis framework does support online analysis.

Assumptions. We assume that the logs faithfully capture events and their causality in the system's execution. For instance, if the log declares that event X happened before event Y, we assume that is indeed the case, as the system executes. We assume that the logs record each event's timestamp with integrity, and as close in time (as possible) to when the event actually occurred in the sequence of the system's execution. Again, we recognize that, in practice, the preemption of the system's execution might cause a delay in the occurrence of an event X and the corresponding log message (and timestamp generation) for entry into the log. We do not expect the occurrence of an event and the recording of its timestamp/log-entry to be atomic. However, we do assume that clocks are loosely synchronized across hosts for correlating events across logs from different hosts.

This chapter presents SALSA's state-machine abstraction, and describes the extensible log-analysis for inferring end-to-end flows from white-box logs. We illustrate our approach using Hadoop logs. The operations team at the VoIP system uses a similar approach to generate the end-to-end flows where call-flow patterns represent the state-machine abstraction, and approximate-matching on key attributes infers end-to-end flows (see Section 4.3).

4.1 State-Machine Abstraction for Log Analysis

SALSA [Tan et al., 2008] analyzes the production system's logs to derive the control-flow on each node, the data-flow across nodes, and the state-machine execution of the system. When parsing the logs, SALSA also extracts key attributes (state durations, bytes read/written, etc.) of interest. SALSA does not require any modification of the hosted applications, middleware or operating system. To describe SALSA's high-level operation, consider a dis-

tributed system with many producers, $P1, P2, ...$, and many consumers, $C1, C2, ...$ Many producers and consumers can be running on any host at any point in time. Consider one execution trace of two tasks, $P1$ and $C1$ on a host X (and task $P2$ on host Y) as captured by a sequence of time-stamped log entries at host X:

```
[t1] Begin Task P1
[t2] Begin Task C1
[t3] Task P1 does some work
[t4] Task C1 waits for data from P1 and P2
[t5] Task P1 produces data
[t6] Task C1 consumes data from P1 on host X
[t7] Task P1 ends
[t8] Task C1 consumes data from P2 on host Y
[t9] Task C1 ends
:
```

From the log, it is clear that the executions (control-flows) of $P1$ and $C1$ interleave on host X. It is also clear that the log captures a data-flow for $C1$ with $P1$ and $P2$.

SALSA interprets this log of events/activities as a sequence of *states*. For example, SALSA considers the period $[t1, t6]$ to represent the duration of state $P1$ (where a state has well-defined entry and exit points corresponding to the start and the end, respectively, of task $P1$). Other states that can be derived from this log include the state $C1$, the data-consume state for $C1$ (the period during which $C1$ is consuming data from its producers, $P1$ and $P2$), *etc*. Based on these derived state-machines (in this case, one for $P1$ and another for $C1$), SALSA can derive interesting attributes, such as the durations of states.

SALSA can then compare these attributes and the sequences of states across hosts in the system. In addition, SALSA can extract data-flow models, e.g., the fact that $P1$ depends on data from its local host, X, as well as a remote host, Y. We explain how to infer end-to-end flows using the state-machine abstraction in a Hadoop cluster.

4.1.1 Hadoop's Logging Framework

Hadoop uses the Java-based `log4j` logging utility to capture logs of Hadoop's execution on every host. `log4j` is a commonly used mechanism that allows developers to generate log entries by inserting statements into the code at various points of execution. By default, Hadoop's `log4j` configuration generates a separate log for each of the daemons—the Job-

Hadoop source-code

```
LOG.info("LaunchTaskAction (registerTask): " + t.getTaskID();
ClientTraceLog.info(String.format(MR_CLIENTTRACE_FORMAT,
        request.getLocalAddr() + ":" + request.getLocalPort(),
        request.getRemoteAddr() + ":" + request.getRemotePort(),
        totalRead, "MAPRED_SHUFFLE", mapId, reduceId, endTime-startTime));
```

⇓ TaskTracker log

```
2011-10-12 00:01:53,789 INFO
        org.apache.hadoop.mapred.TaskTracker:
        LaunchTaskAction (registerTask): attempt_201106031747_9581_m_005404_2
2011-10-12 00:01:54,050 INFO
        org.apache.hadoop.mapred.TaskTracker.clienttrace:
        src: 10.0.0.23:40060, dest: 10.0.0.14:43094, bytes: 6256992,
        op: MAPRED_SHUFFLE, cliID: attempt_201106031747_9581_m_000109_0
```

Figure 4.3. **`log4j`-generated TaskTracker log entries.** Dependencies on task execution on local and remote hosts are captured by the TaskTracker log.

Tracker, NameNode, TaskTracker and DataNode. Each log is stored on the local file-system of the executing daemon (typically, 2 logs on each slave host and 2 logs on the master host).

Typically, logs (such as syslogs) record events in the system, as well as error messages and exceptions. Hadoop's logging framework is somewhat different since it also checkpoints execution because it captures the execution status (*e.g.*, what percentage of a Map or a Reduce has been completed so far) of all Hadoop jobs and tasks on every host. Hadoop's default `log4j` configuration generates time-stamped log entries with a specific format. Figure 4.3 shows a snippet of a TaskTracker log, and Figures 4.4 and 4.5 show snippets of the DataNode logs.

4.1.2 Control Flows in Hadoop

Figure 4.7 illustrates how we infer end-to-end flows in Hadoop. Each TaskTracker log records events related to the TaskTracker's execution of Map and Reduce tasks on its local host, as well as any dependencies between locally executing Reduces and Map outputs from other hosts. On the other hand, each DataNode log records events related to the reading or writing (by both local and remote Map and Reduce tasks) of HDFS data-blocks that are located on the local disk (see Figures 4.4 and 4.5). The dependencies between the HDFS

Hadoop source-code

```
ClientTraceLog.info(String.format(DN_CLIENTTRACE_FORMAT,
      receiver.inAddr, receiver.myAddr, block.getNumBytes(),
      "HDFS_WRITE", receiver.clientName,
      datanode.dnRegistration.getStorageID(), block, endTime - startTime));
```

⇓ DataNode log

```
2011-10-12 00:00:06,358 INFO
        org.apache.hadoop.hdfs.server.datanode.DataNode.clienttrace:
        src: /10.0.0.44:52866, dest: /10.0.0.23:40010, bytes: 67108864,
        op: HDFS_WRITE, cliID: DFSClient_attempt_201106031747_9581_r_000021_0,
        srvID: DS-1453395396-10.0.0.23-40010-1267417661277,
        blockid: blk_3458061035615104394_40996376
```

Figure 4.4. **`log4j`-generated DataNode log with task dependencies.** Local and remote data dependencies are captured. The log also captures dependencies between block IDs and task IDs through the $cliID$ token.

DataNode log

```
2009-02-24 06:04:07,771 INFO
        org.apache.hadoop.dfs.DataNode: 10.251.103.32:50010
        Starting thread to transfer block blk_1338452498666794354 to 10.251.162.80:50010
2009-02-24 06:04:07,829 INFO
        org.apache.hadoop.dfs.DataNode: 10.251.103.32:50010:
        Transmitted block blk_1338452498666794354 to /10.251.162.80:50010
```

Figure 4.5. **`log4j`-generated DataNode log without task dependencies.** Earlier versions of the Hadoop logs did not specify dependencies between block IDs and tasks IDs. Our framework infers these missing dependencies using approximate matching.

data blocks, and the Maps and Reduces are explicitly specified in the logs from Hadoop release 0.20.0 (Figure 4.4). Earlier versions of the DataNode logs did not record these dependencies (Figure 4.5). We infer dependencies by using exact matches on task IDs or block IDs specified in the Hadoop daemon logs. In the event of missing dependencies in the logs, we infer dependencies using approximate-matching by identifying events that occur within a given time window, and that share the same source and destination IP address.

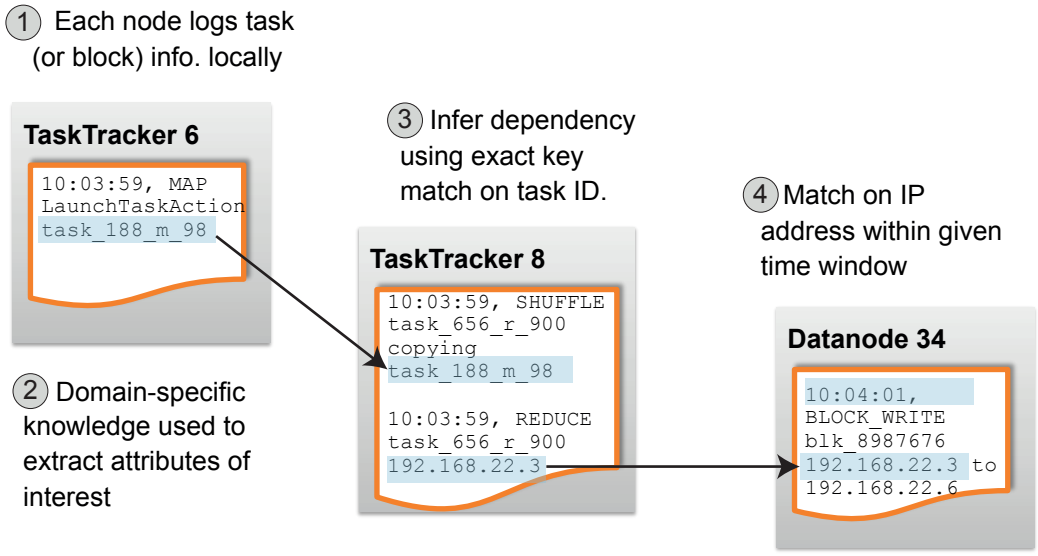


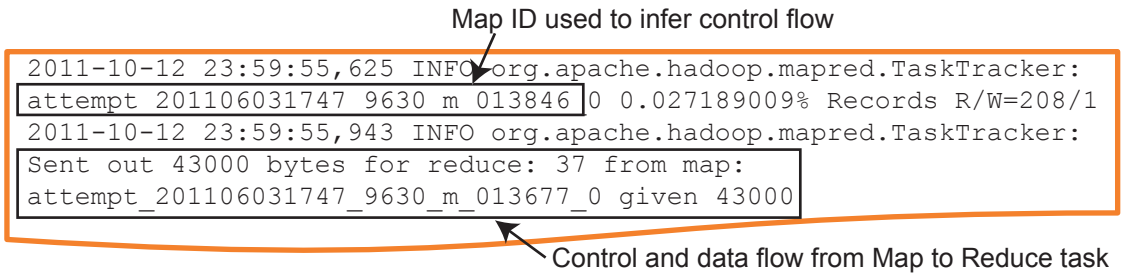
Figure 4.6. **Deriving control-flows from Hadoop's white-box logs.**

4.1.3 Data-Flows in Hadoop

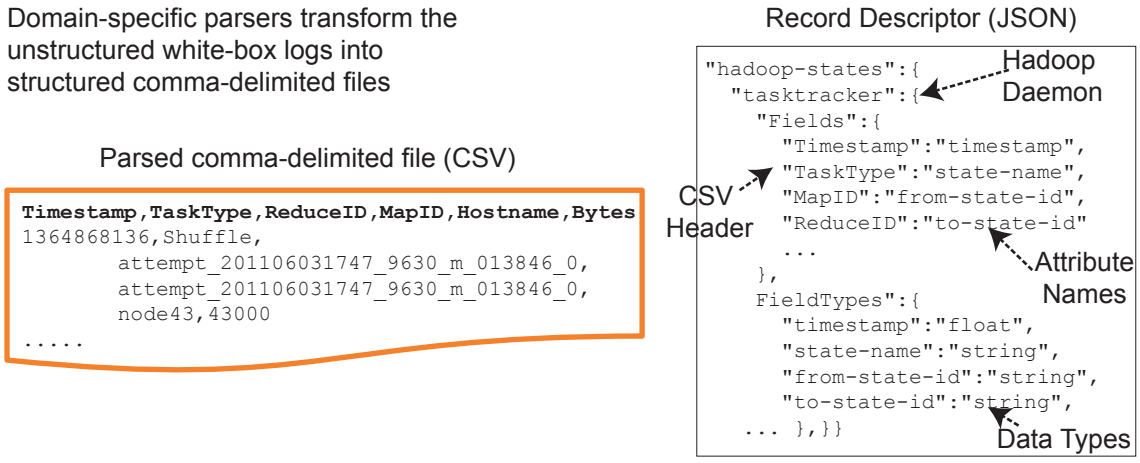
A data-flow dependency exists between two nodes when an activity on one node requires transferring data to/from another node. The DataNode daemon acts as a server—receiving blocks from clients that write to its disk, and sending blocks to clients that read from its disk. Thus, data-flow dependencies exist between each DataNode and each of its clients for each of the ReadBlock and WriteBlock states. We identify the data-flow dependencies on a per-DataNode basis by parsing the hostnames jointly with the log-messages in the DataNode log. Data exchanges occur to transfer outputs of completed Maps to their associated Reduces during the Shuffle phase. This dependency is captured, along with the hostnames of the source and destination hosts involved in the Map-output transfer. Tasks also act as clients of the DataNode in reading Map inputs and writing Reduce outputs to HDFS.

4.2 Extensible Log-analysis Framework

We implemented an extensible log-analysis framework for inferring end-to-end causal flows from the Hadoop logs based on the state-machine abstraction described in Section 4.1. This framework comprises of domain-specific log-parsers and configuration files, and a domain-agnostic inference engine that generates the end-to-end causal flows. The infer-



(a) Snippet of the Hadoop logs showing tokens extracted during a shuffle task.



(b) Snippet of CSV file generated by log parsers, and the corresponding JSON record descriptor.

Figure 4.7. **Extracting attributes of interest from Hadoop logs.** The extraction of attributes is domain-specific and depends on the format of the log.

ence engine stores the data generated in the MongoDB NoSQL database [Plugge et al., 2010]. NoSQL databases are key-value stores that can provide higher scalability and availability than traditional relational databases.

Domain-specific Log-parsers. The domain-specific elements of the log-analysis framework include: 1) log-parsers that extract states of interest from the white-box logs; and 2) configuration files that describe the extracted states, and specify the sequence of expected states for each end-to-end flow. In Hadoop, the log-parsers process the logs generated by the TaskTracker and DataNode daemons on the slave nodes, and the JobTracker and NameNode daemons on the master nodes. Data extracted from the master nodes is merged with data from the slave nodes to provide additional context on the Map, Shuffle, and Reduce states in the TaskTracker, and the ReadBlock and WriteBlock states in the DataNode.

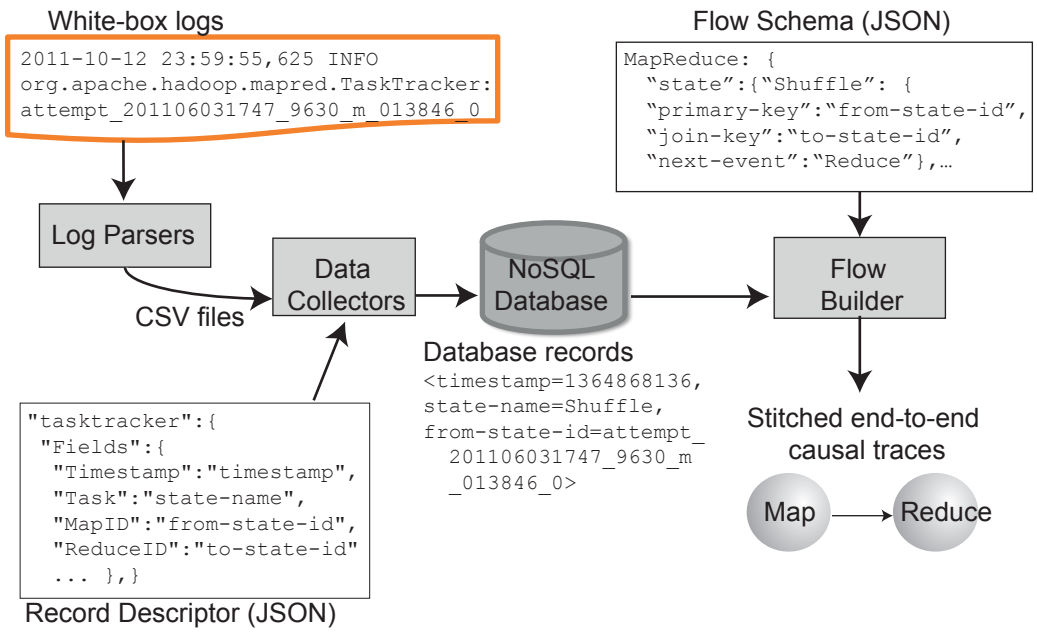


Figure 4.8. **Derived Control-Flow for Hadoop's execution.**

For example, the JobTracker, which schedules jobs and tasks on the cluster, records whether a Map task is accessing data from a local or a remote DataNode—the inference engine exploits this information to infer a causal relationship between Map tasks and data blocks in the Hadoop Distributed File System (HDFS).

Figure 4.7(a) shows the attributes extracted by the log-parsers for a Shuffle task. Each log-parser outputs a CSV (comma-separated-values) file where each record, *i.e.*, line, in the file consists of fields that describe the state, *e.g.*, Shuffles. The first line in the file is the header listing the names of the fields. Each CSV file has a corresponding record descriptor written in JSON (JavaScript Object Notation); JSON [Crockford, 2006] is a text-based open standard designed for human-readable data interchange. The record descriptor maps the header field-names in the CSV file to the key attribute-names in the NoSQL database, as shown in Figure 4.7(b). The record descriptor also describes the data types for each field.

Flow Inference Engine. Figure 4.8 illustrates the domain-agnostic inference engine for generating the end-to-end causal flows. The inference engine consists of a set of data collectors which read the CSV files generated for each component by the domain-specific log-parsers, and stores the attributes of interest in the NoSQL key-value store. The naming conventions and data formats for attributes in the key-value store are governed by the JSON record-descriptor. A separate JSON file describes the schema of each end-to-end flow by

listing the sequence of states that make up each flow. The flow schema specifies the join keys needed to match adjacent states. For example, generating a Shuffle flow based on the Map and corresponding Reduce identifiers stored in the shuffle record in the NoSQL database. If a record in the database does not explicitly specify the unique identifier for the next state in the flow, we rely on approximate-matching to infer the dependency. For example, in Hadoop, the flow schema indicates that the ReadBlock state and the Map state are causally-related. However, the DataNode logs to not track the mapping between Block IDs and Map IDs in earlier versions of Hadoop. Therefore, the inference engine infers dependencies between states using time-based correlation, *e.g.*, by correlating ReadBlocks with Maps that occur during the same time interval, and that access the same source and destination hosts. The inference engine consists of about 3,000 lines of Python code. This implementation of the inference engine, running on a single node, takes about 2-hours to synthesize end-to-end traces from 18GB of raw log data. The performance of the inference engine could be improved by distributing the MongoDB database on multiple nodes.

The inference engine is also responsible for outputting the flows generated into the standard text format supported by our problem localization tool (described in Chapter 6). This output format is shared by both the Hadoop and VoIP systems. Each line of the file represents an end-to-end flow with its associated attributes. The problem localization tool does not require preservation of the ordering between states in an end-to-end flow—rather, it treats the attributes for each flow as a bag of words. Each attribute in the file has a prefix which indicates the category to which it belongs. For example, the category code `+01` indicates a status code, such as `SUCCESS`. The prefix can also include the hostname when attributes are associated with a given node, *e.g.*, `+06:node40 attemptm768` indicates a map whose unique identifier is `attemptm768` was executed on `node40`. Figure 4.9 shows examples of the end-to-end flows generated by the log-analysis framework, along with a snippet of the text files used during problem localization. Our framework can generate flows that capture all the dependencies from the block read during the Map to the block writes during the Reduce as described in [Tan et al., 2009]. However, we opted to generate shorter sub-flows comprising of the Map, Shuffle, and Reduce flows because the cross-product that occurs during the Shuffle can result in very large flows. For example, we observed Hadoop jobs with 140,000 maps and 200 reduces yielding a cross-product of 28 million dependencies during the shuffle.

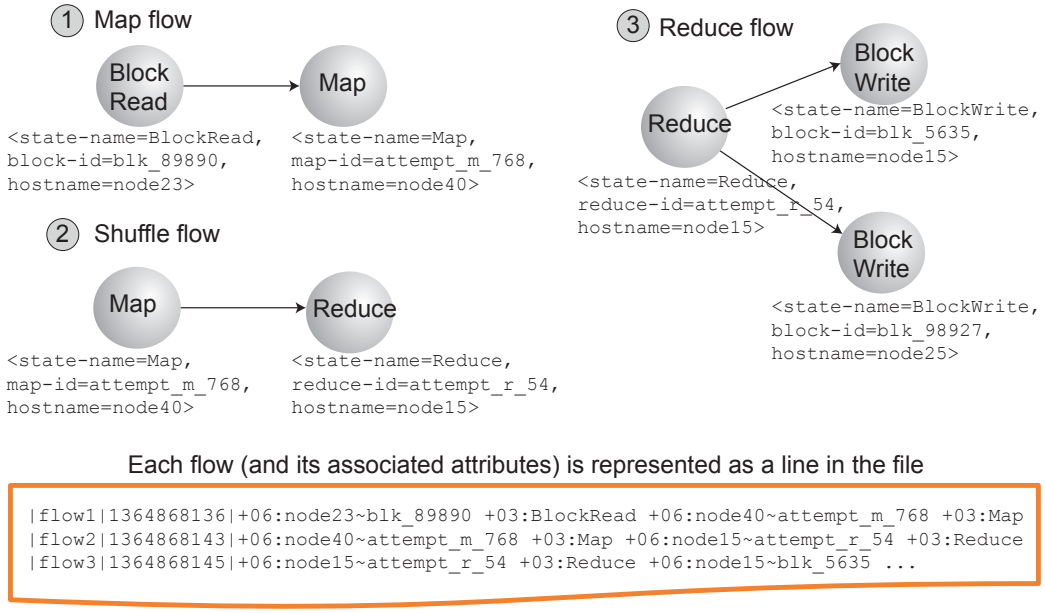


Figure 4.9. **Examples of end-to-end Hadoop flows generated by log-analysis framework.** The framework generates Map, Shuffle, and Reduce flows, and outputs a text file listing the flows and their associated attributes.

4.3 End-to-end Flows in VoIP

The operations team at the large ISP uses a similar approach to generate the end-to-end flows for the VoIP system. The call-flow patterns represent the state-machine abstraction, while approximate-matching on key attributes helps construct the end-to-end flows. These end-to-end flows represent user-level events (*i.e.*, phone calls). The operations team uses these flows for network quality accounting, and defect analysis. In the VoIP network, each network element locally records information about each call that passes through it in the call detail record (CDR) logs (see Figure 4.10). These white-box logs often contain hundreds of attributes that specify details of the call such as the caller and callee information as shown in Table 4.1. The structure and semantics of these records are vendor-specific, and require domain-specific log-parsers to extract attributes of interest. The logs tend to be large—the average size of the raw CDR logs is 30GB/day. Even after significant consolidation to eliminate irrelevant data fields, the average size is 2.4GB/day, and each log contains between 5000–10000 unique call attributes pertinent to diagnosis, *i.e.*, attributes that appear in defective calls. In addition to CDR logs, each network element generates black-box logs at 5-15 minute intervals which store OS performance counters.

The operations team consolidates data extracted from the raw CDRs and constructs a

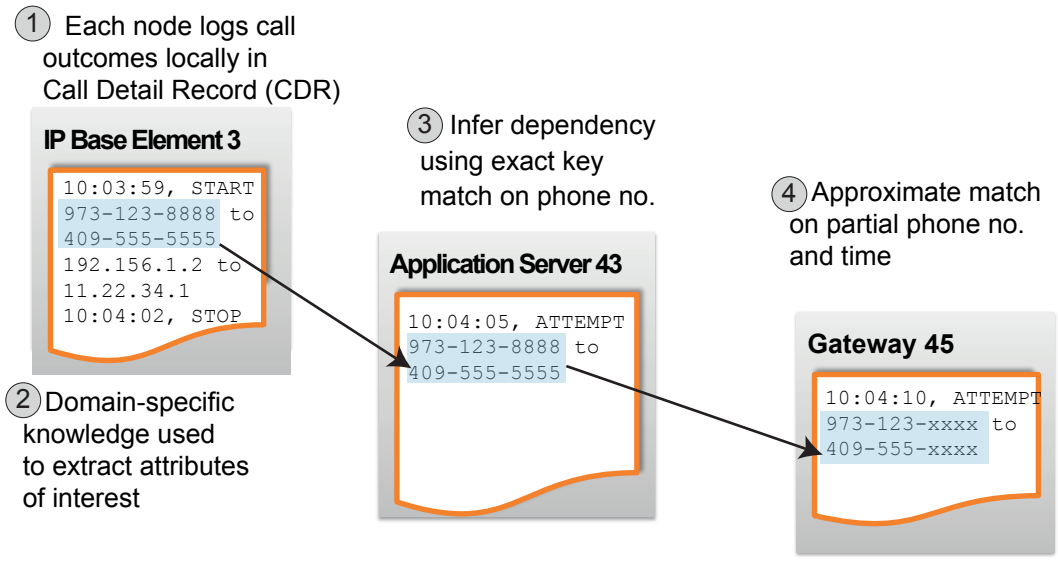


Figure 4.10. **Fix caption: Derived Control-Flow for Hadoop's execution.**

Table 4.1. **A Generic Call Detail Record (CDR) in the VoIP system.**

Attribute	**Description**
Timestamps	Call start and end times
Service	Type of service
Caller/callee info	Phone number and IP address
Network Element	Name of network element, *e.g.*, gateway X
Error code	Problem encountered, *e.g.*, server timeout

master record that represents the consolidated end-to-end flows. Each raw CDR contains common keys such as timestamps, phone numbers, and IP addresses that can be used to infer the end-to-end flows for each phone call. However, the matches need not be exact, and domain-specific matching rules can be used. For example, some vendor log partial phone numbers. To cater for these instances, we infer dependencies by approximate-matches on the partial phone numbers within a small window of time.

We post-process the end-to-end flows generated by the operations team to incorporate synthetic attributes that increase the scope of problems diagnosed as described below:

- **Wild-card attributes.** We use wild-cards to create synthetic attributes that can detect problems that affect all the servers in a high-availability cluster. For example, the synthetic attribute, $svr * loc1$, would represent the high-availability servers at $location1$, comprising of the primary server, $svr1loc1$, and the standby server, $svr2loc1$.

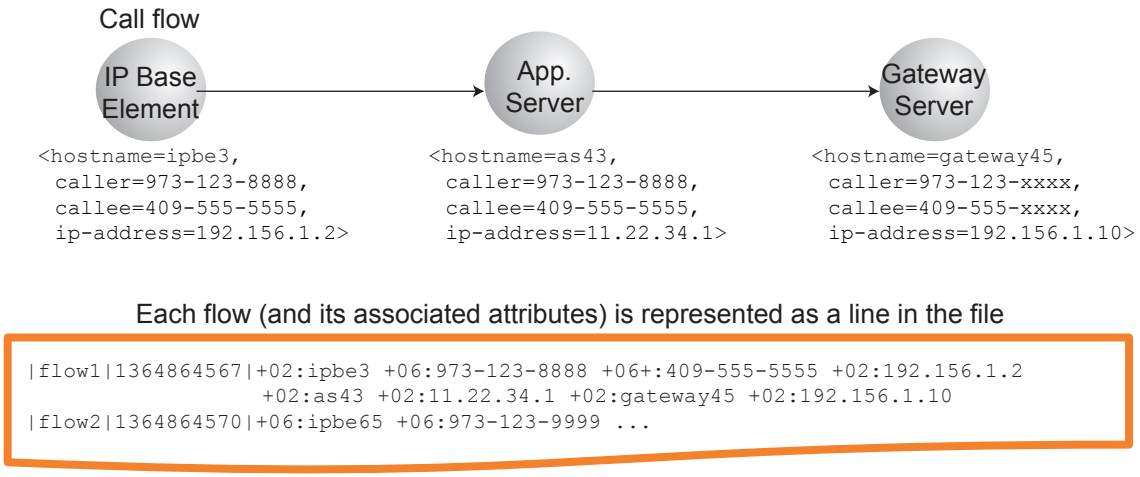


Figure 4.11. **Derived Control-Flow for VoIP call flows.**

- **Ingress and egress fields.** The call detail records contain ingress and egress fields that identify the preceding and the next network element in the call path. By extracting the hostnames from these fields, we can detect problems due to timeouts where the call detail record of the faulty network element is missing.
- **Attribute categories.** We categorize attributes in the call detail records into defect codes, network elements, telephone numbers, and VoIP services. Operators can use these categories to filter diagnostic output.

The end-to-end flows for the VoIP system serve as input to our problem localization tool described in Chapter 6. The log-format for the VoIP end-to-end traces is the same as the Hadoop traces shown in Figure 4.9.

4.4 Summary

This section described our extensible log-analysis framework that infers end-to-end causal flows using the unmodified white-box logs available in production systems. These end-to-end causal flows capture the control path and data demands of application requests as they are serviced across components and machines in the production system. Each end-to-end flow embodies a user's interaction with the system through Map and Reduce flows in Hadoop, and phone calls in VoIP. The log-analysis framework comprises of domain-specific log-parsers and configuration files, and a domain-agnostic inference engine that generates the end-to-end causal flows. The inference engine generates end-to-end flows

by coupling SALSA's state-machine abstraction [Tan et al., 2008] with configuration files that specify the sequence of expected states for each flow. When the application logs do not explicitly record the identifiers for the next state in the sequence, our framework relies on *approximate matching* based on domain-specific knowledge about relationships between states and time-based correlations.

We applied our framework to generate end-to-end flows in Hadoop. These end-to-end flows are useful for a variety of purposes, such as anomaly detection (Chapter 5), problem localization (Chapter 6), and visualization (Chapter 9). A limitation of our log-analysis approach is that the end-to-end flows might not be accurate since state-machines are inferred purely from the log data. Our framework has no way of verifying what the system is actually doing. Thus, the inferences are undoubtedly affected by the quality of the log data. Another limitation of using the unmodified logs is that the log-parsers might need to be upgraded for every new version of the production system, if system's log messages or its logging points are modified by the developers.

Essentially, all models are wrong; some models are useful.

G. Box, Empirical Model-Building and Response Surfaces, 1987

Chapter 5

Anomaly Detection

Major outages cause considerable disruptions to service delivery in production systems. These disruptions can be quickly detected by alarms in the service provider's network, or by violations of the service-level objectives (SLOs) which guarantee high-throughput or low-latency. Chronics, on the other hand, are more elusive for the operations team to detect because they may not be severe enough to trigger low-level alarms within the service provider's network. Chronics may manifest to the end-user as performance degradations, error messages, or incorrect results. To effectively detect chronics, the service provider needs to monitor the end-user's experience with the service. Figure 5.1 provides an overview of how we detect anomalies in the end-to-end flows, described in Chapter 4. Anomaly-detection determines whether an end-user is experiencing problems with the system, and labels each flow as successful or failed.

Research questions. The research questions that we asked during anomaly detection are:

- How do we detect performance problems in the absence of labeled data?
- How do we distinguish between performance differences due to legitimate application-behavior from differences due to performance problems?

Design overview. We detect anomalous end-to-end flows using a two-pronged approach. Some user-visible problems manifest as errors such as timeouts. While the user is aware that there is a problem, the operations team may not be made aware of the problem unless it perturbs a significant number of requests, or the customer complains. We detect these problems by extracting error codes from failed flows, or by applying *domain-specific heuristics*. VoIP systems have well-established heuristics for detecting anomalous

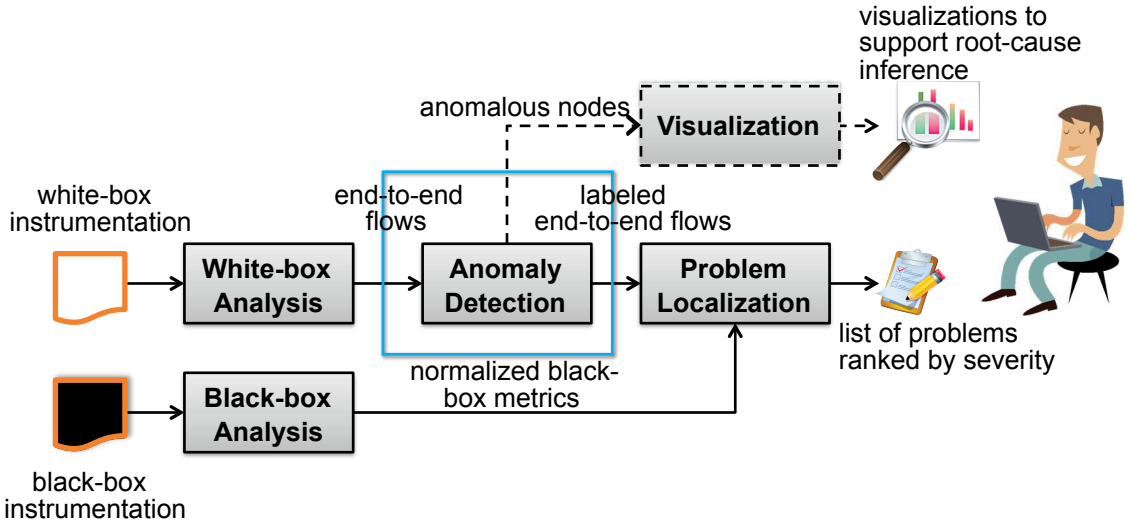
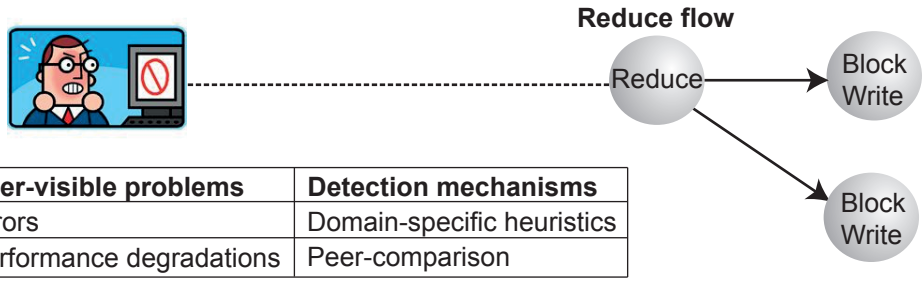


Figure 5.1. **Overview of anomaly detection.** Anomaly detection identifies user-visible errors and performance degradations in end-to-end flows.

user interactions. For example, a user redialing the same number immediately after disconnection, zero talk time, or server reported error code can be used to indicate a problem in the VoIP system. In Hadoop, uncaught task exceptions serve as indicators of problems. Similar heuristics can also be applied to Internet Services, *e.g.*, a user repeatedly refreshing a web page might indicate issues with the website.

On the other hand, performance problems are harder to detect because there is no outright error other than the user's frustration with the progress of their request. We exploit the notion of *peers* to detect performance problems. Peers are system behaviors that can be considered equivalent under normal conditions, *e.g.*, tasks within the same job exhibit similar performance. Any significant deviation from the *peers* is regarded anomalous. Peer-comparison assumes that the majority of the system is working correctly—a reasonable assumption when targeting chronics. Peer-comparison is an attractive option for anomaly-detection because: 1) it does not rely heavily on domain experts to model system behavior; 2) it is relatively robust to workload changes as peers execute similar workloads in a given period of time; and 3) operators do not need to explicitly demarcate non-problem and problem time-periods which might be difficult to identify when problems persist for long periods of time. However, for peer-comparison to be practical for anomaly detection, it needs to distinguish legitimate sources of variance, such as I/O differences, from variance induced by problems. This chapter describes our peer-comparison approach.



User-visible problems	Detection mechanisms
Errors	Domain-specific heuristics
Performance degradations	Peer-comparison

Figure 5.2. **Strategies for anomaly-detection.** Anomaly detection uses heuristics and peer-comparison to identify user-visible problems in end-to-end flows.

5.1 Peer-comparison for Anomaly Detection

Peer-groups in production systems comprise of sets of machines, components or tasks that perform similar work in a given time-interval. These peer-groupings can arise due to the need for fault-tolerance, load-balancing, or parallel-processing. A peer-comparison approach detects anomalies based on the notion that peers exhibit similar behavior under fault-free conditions, and diverge when a problem occurs.

Challenges. The two main challenges faced when implementing a peer-comparison approach are: 1) how to identify peers; and 2) how to factor out legitimate sources of variance. Researchers have typically addressed these challenges by using two strategies namely: 1) *segmentation* where they use domain-specific knowledge or clustering to identify groups with similar behavior, *i.e.*, peers; and 2) *regression or normalization* where they learn inter-relationships between explanatory variables to factor out sources of variability amongst peers. We describe these strategies in detail below.

- **Identifying peers through segmentation.** Some peer-comparison approaches rely on domain-specific knowledge to identify peers. [Kavulya et al., 2008; Pan et al., 2009a; Kasick et al., 2010] exploited knowledge about the structure of parallel distributed systems to identify peers and pinpoint anomalous nodes. [Barham et al., 2004; Sambasivan et al., 2011] automatically identify peers by clustering requests, while [Thereska et al., 2010] uses nearest-neighbor searches to automatically find peers with similar configurations. These clustering techniques are better suited for handling static attributes such as CPU type rather than dynamic attributes such as CPU and memory usage which vary over time.

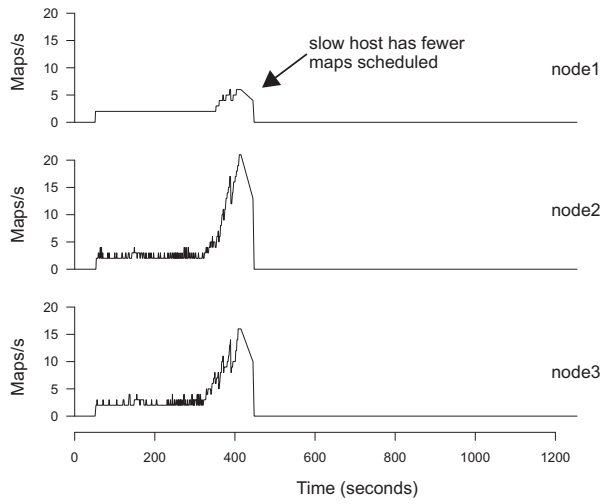


Figure 5.3. **Example of peer-comparison of Map tasks scheduled across M45 hosts.** Performance of a slow host in a user's job differs significantly from its peers.

- **Coping with variance.** High-variance complicates the task of setting thresholds for detecting problems [Sambasivan and Ganger, 2012a]. Regression can be used to reduce some sources of variance such as load variations and application differences [Kelly, 2005; Cherkasova et al., 2008; Stewart et al., 2007]. Instance-based learning approaches [Smith, 2007; Kapadia et al., 1999; Kavulya et al., 2010] combine clustering to identify sets of similar jobs in the recent past, and regression to predict the completion time based on the set of similar jobs. Instance-based approaches flag jobs as anomalous if they exhibit high residuals.

5.1.1 Background

Our initial exploration of the feasibility of peer-comparison for problem diagnosis began when we were investigating the effect of different group communication protocols on problem diagnosis in replicated systems [Kavulya et al., 2008]. Our research demonstrated that a peer-comparison approach was less susceptible to false positives due to workload fluctuations than an approach that relied on historical data to identify abrupt changes in metrics on individual nodes. Next, we applied our peer-comparison approach to parallel-processing systems such as MapReduce clusters [Tan et al., 2008; Pan et al., 2009a; Tan et al., 2010a]. We developed algorithms that compared the distribution of black-box and white-box metrics across the different nodes of a MapReduce cluster, and indicted the culprit-node by iden-

tifying *odd-man-out* behavior. We observed that in the absence of a performance problem, the slave nodes in a Hadoop cluster tended to exhibit similar behavior, as measured in any number of ways, *e.g.*, CPU usage, network traffic, and task durations. This peer-similarity arose because the JobTracker tried to distribute workload in the cluster as evenly as possible. However, when we injected a performance problem (or when we observed some problems in the field), we observed further that the slave node on which the problem originated (the *culprit* node) deviated significantly from the other slave nodes in its behavior, as illustrated in Figure 5.3. These earlier implementations of our peer-comparison algorithms assumed that the hardware on the cluster was homogeneous, and that the workload was evenly distributed across all the slave nodes. We also assumed that performance problems arose due to a single, independent fault.

5.1.2 Peer-comparison of End-to-end Flows

We developed a practical peer-comparison approach for detecting anomalous behavior in production systems. Our peer-comparison approach was inspired by the concept of Rika—a Swahili word for people in the same age-group, and is a core-concept in Kenyan culture as peers undergo rites of passage like initiation and marriage at similar times. Our peer-comparison approach defines peers as tasks belonging to the same job, and detects performance degradations in the end-to-end Hadoop flows by searching for flows whose performance profiles differ significantly from their peers. The approach combines domain-specific knowledge about the structure of Hadoop jobs to identify peers, with stepwise-regression to automatically factor out variance due to application-level differences such as differences in input sizes.

Assumptions. We relax the assumption of a homogeneous workload in [Tan et al., 2008; Pan et al., 2009a; Tan et al., 2010a], and cater for variance due to application-level differences. We also relax the single, independent fault assumption—coping with more complex failure modes like multiple, independent faults when coupled with our problem-localization approach. We can cope with heterogeneous hardware in a cluster if the hardware type of each component is specified as a categorical attribute—the inclusion of hardware type allows us to form homogeneous sub-clusters by grouping data based on hardware type. Studies have shown that even homogeneous systems, *e.g.*, disks of identical make and model [Krevat et al., 2011], experience variations in performance. We treat any

Table 5.1. **Peer-comparable properties.** Examples of peer-comparable properties for Hadoop.

Property	Description	Examples
Temporal	Timestamps	Start and end times
Spatial	Nodes running same workload	Nodes running same job
Phase	Tasks with same structure	Maps with local reads vs. remote reads
Context	Application or hardware characteristics	Job ID, user, service

large variations in hardware performance in homogeneous systems as anomalies. We also assume that the majority of the system is working correctly most of the time, and that clocks are loosely synchronized across all nodes to facilitate log correlation.

5.1.3 Identifying Peers using Domain-specific Knowledge

Peers can be compared across different dimensions. We identify peer-comparable properties in production systems based on the concept of age-sets (or Rika) in African culture. Belonging to an age-set implies *temporal*, *spatial*, *phase*, and *contextual* similarities between members of the age-set. *Temporal* similarity arises because age-sets comprise of people born around the same time. *Spatial* similarity arises because members of the same-group typically reside within the same geographical region. Members of an age-set also pass through a series of age-related statuses together imposing a *phase* similarity between members. For example, among the Maasai people of East Africa, young men undergo an initiation ceremony which marks their transition from childhood to junior warriors. After several years, they participate in another ceremony to transition to senior warriors. Lastly, the structure of the age-set and the series of age-related transitions are dictated by *context*, *e.g.*, tribe and gender.

The concepts of *temporal*, *spatial*, *phase*, and *contextual* similarity can also be applied to identifying peers in a distributed systems like Hadoop. *Temporal* similarities occur between events in the same time window, for example, distributed systems often have daily variations in behavior with peak load during the day and non-peak load at night. *Spatial* similarities can arise due to load-balancing of workloads across similar nodes. *Phase* similarities relate to the path that causal flows take through the system. For example, the Map, Shuffle, and Reduce phases in MapReduce jobs, or the distinct paths taken by calls involved in a teleconference when compared to call-waiting. *Contextual* similarities arise due to shared

Table 5.2. **Application-level attributes that influence task durations in Hadoop.**

Application-level Attribute	**Type**
Job ID, Job name, User name	Categorical
Task Type *(Map, Shuffle, Reduce)*	Categorical
Task status *(Success, Failed, Incomplete)*	Categorical
Hostname	Categorical
Data-locality *(Data-local, Rack-local, Non-local)*	Categorical
Timestamp, Duration	Continuous
Task input/output records	Continous
Combiner input/output records	Continuous
HDFS bytes read/written	Continuous
Local bytes read/written	Continuous
Records spilled to disk	Continuous
Shuffle bytes read	Continuous
Reduce input groups	Continuous

application or hardware characteristics such as tasks belonging to the same job or user, or tasks running on the same hardware platform.

We rely on domain-specific knowledge to identify peers. The analysis of performance characteristics in Hadoop job in Chapter 3 showed us that the majority of Hadoop jobs exhibited low-variance across Map and Reduce tasks—suggesting that peer-comparison of tasks belonging to the same job is a feasible strategy for anomaly-detection. Table 5.1 describes peer-comparable properties and provides examples of their use in Hadoop. Although we focus on Hadoop, these peer-comparable properties can be applied to other systems.

5.1.4 Coping with Variance using Linear Regression

In Chapter 3, we also observed that some Hadoop jobs experience high-variance in Map and Reduce durations. The challenge is how to differentiate between legitimate sources of variance (*e.g.*, tasks processing more data) from illegitimate sources of variance (*e.g*, resource contention). Table 5.2 lists the application-level attributes extracted from the Hadoop logs which influence task durations. These attributes are either categorical or continuous in type. Categorical attributes take on a limited number of possible values such as data-

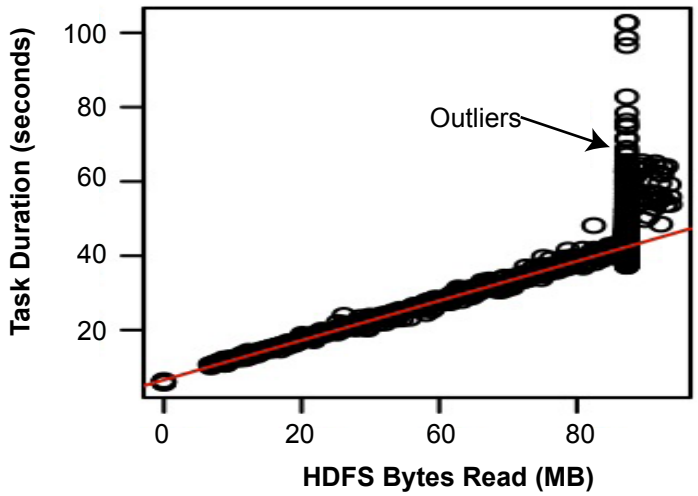


Figure 5.4. **Example of variance in the durations of tasks in a Hadoop job.** The task durations in this job are primarily influenced by the amount of data read from HDFS, and the amount of bytes written to local disk. The task durations also exhibit unexplained variance that might be due to a problem on the cluster.

locality which indicates whether a task reads data from a local disk, a shared rack, or a remote node of a different rack (*i.e.*, non-local). While continuous attributes, such us the number of local bytes read from disk, are numeric variables that can have an infinite number of values within a certain range. Figure 5.4 provides an example of a Hadoop job whose task durations were primarily influenced by the amount of data read from HDFS, and written to the local file-system. The job also exhibits unexplained variance that might be due to a problem on the cluster. We describe how to localize these problems in Chapter 6.

Linear-Regression Approach

Our anomaly-detection approach uses stepwise linear-regression to automatically identify the application-level features that influence the durations of end-to-end flows, and to detect anomalous flows. In Hadoop, these flows are the Map, Shuffle and Reduce tasks belonging to the same job. Each type of task represents a peer-group. The linear-regression model predicts expected task durations using a multi-variate linear function of the features (or covariates) listed in Table 5.2. Equation 5.1 expresses the multi-variate linear function where y is the response variable, $x_1, x_2, ..., x_p$ are the covariates or independent variables, $\beta_1, \beta_2, ..., \beta_p$ are the regression coefficients, and ε is the error term or residual.

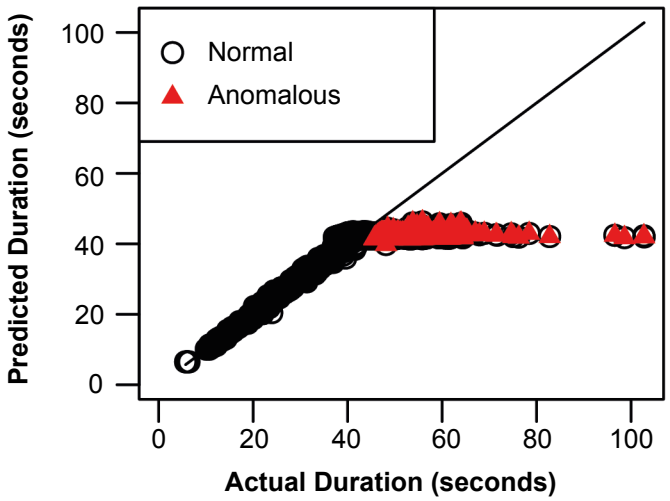


Figure 5.5. **Detecting anomalous tasks using linear-regression.** Tasks with high-residuals are flagged as anomalous (indicated by red triangles).

$$y = \beta_1 * x_1 + \beta_2 * x_2 + ... + \beta_p * x_p + \varepsilon \tag{5.1}$$

We measured the goodness of fit of our linear models using the adjusted coefficient of determination, R^2, which explains the percentage of variation that can be explained by the variables in our model. The adjusted R^2 adjusts for the number of parameters in the linear model and ranges from 0 to 1. A value of 1 indicates that the model perfectly explains the observed data. If we were unable to compute the adjusted R^2 value due to non-linear relationships in the data, we detected outliers using the median absolute deviation (MAD). MAD is a robust measure of the variability in quantitative data. The median absolute deviation for a univariate dataset, $X_1, X_2, X_3, ...X_n$, is the median of the absolute deviations from the data's median expressed as: $MAD = median_i(|X_i - median_j(X_j)|)$. The threshold value is calculated as a constant, $K * MAD$.

We automatically identified the relevant features that influence task performance using stepwise-regression. Stepwise-regression sequentially adds and drops features in the linear model, and attempts to return the smallest linear model that minimizes the Akaike Information Criterion (AIC). The AIC is a measure of the tradeoffs between the precision and complexity of statistical models. We observed that some input features were linearly dependent, *i.e.*, collinear, in certain jobs. For example, some jobs exhibited a strong correla-

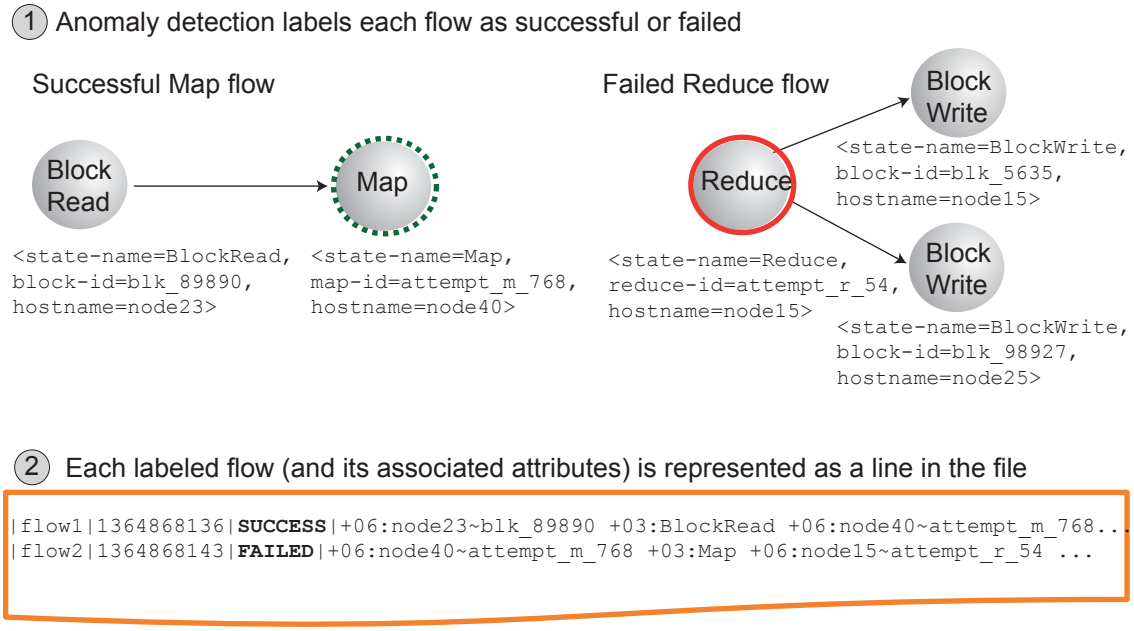


Figure 5.6. **Labeled end-to-end flows generated by anomaly detection.**

tion between the number of bytes read from HDFS and the number of map input records. Linear regression in the presence of collinearity leads to unstable estimates, *e.g.*, negative or very large estimates for the task durations. We detected collinearity by calculating the variance inflation factor[Stewart, 1987] and sequentially dropping collinear variables. The variance inflation factor, $\frac{1}{1-R^2}$, is derived from the coefficient of determination, R^2, which is computed by performing a linear regression analysis of each independent variable using the remaining independent variables. Collinearity exists if the variance inflation factor exceeds 10. We used the median absolute deviation to flag tasks whose residuals exceeded the threshold, $(K * MAD)$, as anomalous. Figure 5.5 provides an example of how we detect anomalous Map tasks in a Hadoop job using linear-regression. The outputs of anomaly detection are the labeled end-to-end flows illustrated in Figure 5.6.

5.2 Summary

This chapter presents our two-pronged approach for detecting chronics end-to-end flows in production systems. The approach relies on domain-specific heuristics and a peer-comparison approach to detect anomalies by identifying end-to-end flows whose behav-

ioral profiles differ significantly from their peers. The two main challenges faced when performing peer-comparison are: 1) how to identify peers; and 2) how to cope with legitimate source of variance due to application-level and load differences. Our anomaly-detection approach addresses these challenges by using domain-specific knowledge to manually identify peer-groups, and by using stepwise-regression to automatically factor out variance induced by application-level differences. We use the linear-regression models to identify end-to-end flows which exhibit high-residuals—implying that the performance profile of these flows deviates significantly from the peer-group. We describe how to apply our approach in parallel-processing frameworks such as Hadoop. For the VoIP system, we relied primarily on heuristics to identify anomalous calls. The use of heuristics was feasible because the telecommunications industry has well-established rules for detecting failed calls.

It is a mistake to think you can solve any major problems just with potatoes.

D. Adams, Life, the Universe and Everything, 1982

Chapter 6

Problem Localization

CHRONIC problems can occur due to a variety of different underlying problems at the service provider's network or at the customer site, *e.g.*, resource contention, misconfigurations, failed software upgrades, and hardware issues. Localizing the root-cause of the chronics detected using the end-to end flows is challenging because a single fault can produce multiple symptoms in different parts of a system; conversely, different faults can produce the same symptom, *e.g.*, a generic timeout error could be due to network congestion or a hang in a server process. Due to the scale of production systems, there are often multiple ongoing problems—some of which may be triggered by an unexpected combination of corner-cases. Surprisingly, we have observed that even simple problems, like running out of disk space, can confuse users (especially novices) because the system generates spurious error messages that overwhelm the user. Sifting through these cryptic symptoms requires automated problem-localization tools that disambiguate between chronics caused by a single independent fault, by multiple independent faults, or by a combination of interacting factors.

Research questions. The research questions that we asked during problem localization were:

- How do we identify problems due to a combination of factors?
- How do we distinguish between the causes of multiple ongoing problems?
- How do we determine the resource-usage metric (*e.g.*, CPU and memory) associated with the problem?
- How do we handle noise due to flawed anomaly detection?

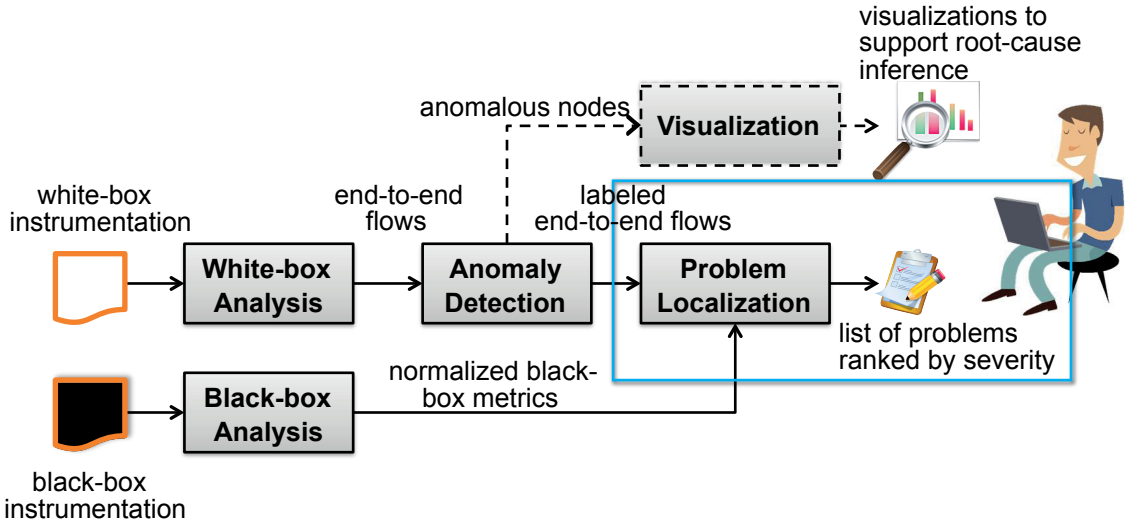


Figure 6.1. **Overview of problem localization.** Problem localization drills down on the source of the problem, and generates a list of problems identified ranked by severity.

Design overview. Our problem-localization tool performs statistical diagnosis of chronics on large systems by combining data logs from different sources, and by diagnosing multiple ongoing problems—each identified by complex signatures across multiple dimensions. The tool handles both discrete data extracted from the end-to-end flows described in Chapters 4 and 5, and real-valued OS performance counters stored in the black-box logs as shown in Figure 6.1. To enable discovery of problems that have never been seen before, or those that have persisted in the background for a long time, our tool does not rely on historical data. We minimize false positives by using a *top-down* approach that relies on a scalable Bayesian distribution learner and an information-theoretic measure of divergence (Kullback-Leibler divergence) [Kullback and Leibler, 1951] to identify sets of *problem signatures* that together explain the differences between the failed and successful user interactions. Our statistical algorithm is also robust to occasional mislabeling in the end-to-end flows by ranking identified problems based on the number of flows affected—any spurious attributes introduced by noise receive lower ranks.

Our problem-localization tool is scalable and domain-agnostic—requiring only changes to some well-isolated data parsers to be adapted to other applications. It is currently in production use by the operations team of a major US-based provider's VoIP platform that handles tens of millions calls per day. The tool has also been used to localize problems in a production Hadoop cluster. Problem-localization proceeds in three steps. First, we com-

pute an anomaly score for each attribute using a standard information-theoretic metric that represents the *difference* between the success and failure attribute occurrence probability distributions (Section 6.1). Second, we use a scalable ranking function to identify groups of attributes that best discriminate between the success and failure labels (Section 6.2). Third, we examine the black-box logs of any network elements indicted during the second step, and apply a similar ranking function to identify anomalous black-box metrics, such as high CPU or memory usage (Section 6.4). We describe each step of the approach in more detail below.

6.1 Scalable Anomaly Score Computation

Figure 6.2 shows a log snippet that represents the end-to-end flows generated using the white-box logs. Each end-to-end flow, *i.e*, phone call in VoIP or task in Hadoop, is labeled as successful or failed based on the anomaly-detection results. Each line in the log represents an end-to-flow, and stores an unordered list of attributes associated with each flow. These attributes include phone numbers or task identifiers, timestamps, duration, hostnames and IP addresses of components used to process each flow, and any error and success codes generated by these components. Our tool transforms these logs into a truth table representation where the rows represent the end-to-end flows, and the columns represent the attributes. The end result is a sparse table that can support 10s of thousands of attributes and 10s of millions of flows.

We then identify the group of attributes that are the most highly indicative of failures by ranking the attributes using a scoring function that quantifies the ability of the group to discriminate between successful and failed user-interactions. To do so, we use an iterative Bayesian approach to learn a simple Bernoulli (*i.e.*, "coin toss") model of successes and failures. The idea is to model an attribute a as occurring in an end-to-end flow with a fixed, but unknown probability p^a which depends on multiple factors such as routing paths and load distribution. This attribute occurrence probability is p_f^a for failed flows, and p_s^a for successful flows. The model estimates these unknown probabilities using attribute counts computed from the end-to-end traces. However, rather than learning a single value, we can estimate the entire probability *distribution* of these unknown attribute occurrence probabilities, *i.e.*, $F_f^a(x) = P[p_f^a \leq x]$, and $F_s^a(x) = P[p_s^a \leq x]$. We start with an initial estimate for

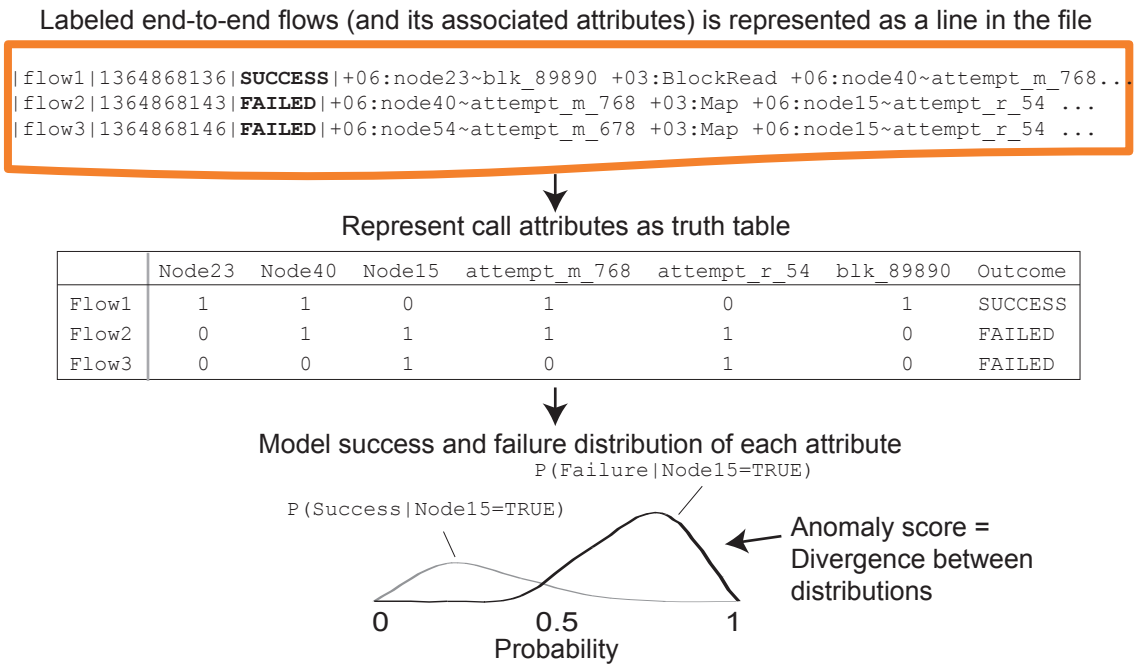


Figure 6.2. **Computing anomaly score for individual attributes.** We first extract attributes from the labeled end-to-end traces and represents them as truth table. Next, we compute an anomaly score as the distance between the successful and failed distribution for each attribute.

F_f^a and F_s^a, and Bayes rule is used to update this estimate as each new flow in the dataset is processed, depending on whether it is a successful and failed flow, and whether it contains the attribute a or not. For example, in Figure 6.2, we estimate the conditional probability distribution that a flow succeeds or fails given that $Node15 = True$. The x-axis represents our estimate of the probability, while the y-axis represents our degree of belief that this estimate is true. The degree of belief can be any number greater than zero that results in the area under the curve adding up to one; a broader curve represents more uncertainty in our estimates of the underlying distribution, while a narrower curve indicates greater confidence. Once these distributions are learned, the score is simply the Kullback–Leibler divergence [Kullback and Leibler, 1951], a standard information theoretic metric of the "difference" between two distributions, computed between these success and failure attribute occurrence probability distributions.

We can compute the score for large numbers of attribute groups and over large volumes of flows efficiently because the KL divergence can be reduced to a closed form equation due to two textbook results. The first result is that Beta distributions are *conjugate priors* for Bernoulli models, *i.e.*, if a Beta distribution $Beta(x, y)$ is used as an initial estimate for

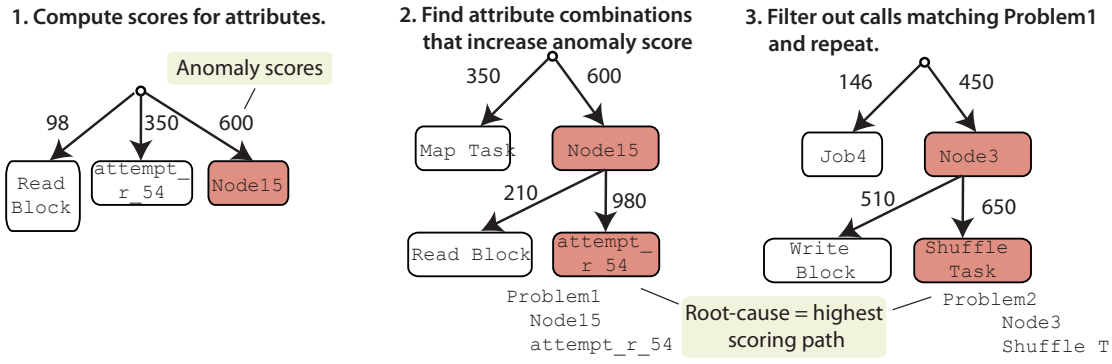


Figure 6.3. **Ranking combinations of attributes correlated with problems.** We use an iterative Bayesian approach to rank combinations of attributes most correlated with the problem.

distribution F_f^a (or F_s^a), and the forward probability $P[a \text{ appears in a failed flow}|F_f^a]$ (and similarly for successful flows) is given by a Bernoulli distribution, then the new estimate for F_f^a after applying Bayes rule is also a Beta distribution $Beta(x+a, y+b)$, where a and b are the number of flows with and without attribute a, respectively. The second result is that the KL divergence between two Beta distributed random variables, $X \sim Beta(a,b)$ and $Y \sim Beta(c,d)$ is given by the Equation

$$KL(Y||X) = \ln \frac{B(a,b)}{B(c,d)} - (a-c)\psi(c) - (b-d)\psi(d) + (a-c+b-d)\psi(c+d) \quad (6.1)$$

where B is the Beta function and ψ is the digamma function. Therefore, if one starts with the initial assumption that the failure and successful flow attribute occurrence probabilities p_f and p_s are uniformly distributed (which is a special case of the Beta distribution), then setting $a/b = 1+$#successful flows with/without attribute a, and $c/d = 1+$#failed flows with/without attribute a yields the desired score, *i.e.*, $KL(Y||X) = KL(P[Failure|attribute\ a]||P[Success|attribute\ a])$ in Equation 6.1. A similar observation is used to compute KL divergences between two Bernoulli models in [Liu et al., 2006].

Figure 6.2 shows how the scoring works in terms of the density functions for the success and failure attribute occurrence probability distributions. Intuitively, it scores higher those attribute groups that are more likely to occur in failed flows than in successful flows, but it does so while taking into account the volume of data observed. This allows us to increase confidence as we observe more flows. For example, the score is higher after observing an attribute in 50 out of 100 failed flows as compared to observing it in 1 out of 2 failed flows, even though both scenarios have the same underlying probability p_f of 0.5.

Illustrative Example. We illustrate the computation of the anomaly score using a simulated example. Suppose a configuration change in *nodeR* caused some flows that passed through it to fail. Assume that the number of failed flows that passed through *nodeR* is 3000, while the number of successful flows is 25000. Due to other problems that might exist in the system, the total number of failed and successful flows is 5000 and 1000000 respectively. To compute the anomaly score $KL(P[Failure|nodeR]||P[Success|nodeR])$, we calculate the attribute counts as shown in Equation 6.2:

$$
\begin{aligned}
&a = 1 + 25000 \text{ (a=1 + successful flows with attribute)} \\
&b = 1 + (1000000 - 25000) \text{ (b=1 + successful flows without attribute)} \\
&c = 1 + 3000 \text{ (c=1 + failed flows with attribute)} \\
&d = 1 + (5000 - 3000) \text{ (d=1 + failed flows without attribute)} \\
&KL(P[Failure|nodeR]||P[Success|nodeR]) = 789295
\end{aligned} \tag{6.2}
$$

6.2 Attribute Group Generation

Chronics can arise due to complex triggers involving a combination of factors such as conflicting software versions on different network elements. These complex conditions are represented as conjunctions of groups of attributes, e.g., `Node15 and ReduceTask and attemptr54`. Our tool identifies these groups using a search tree as shown in Figure 6.3. The root node of the tree represents the null set, and each branch represents a single attribute. Each non-root node of the tree represents a unique attribute group as specified by the path from the root to that node, and the weight of the node is the anomaly score for node's attribute group. Starting with the direct children of the root (representing a single attribute each), we expand the tree to a depth of d to consider all groups containing up to d attributes. Expanding along a branch of the tree involves a filtering operation that retains only those successful and failure events in which the attributes represented by that branch were present. The filtering is required to get the success and failure counts needed for the anomaly score computation. These filtering operations dominate the algorithm's running time and dictate the data structures used in our design, as described in Section 6.3. The

node with the highest weight is picked as the dominant problem signature in that iteration.

For this process to be practical, there are two additional complications that must be handled. The first is to find any attributes that are synonyms of each other. For example, in VoIP, attributes such as a particular customer's IP address and name, or a customer's IP address and a dedicated IPBE server assigned to that customer, may appear together in all flows. Such overlapping attributes are indistinguishable from a statistical point of view but may be meaningful to an operator from a semantic standpoint (*e.g.*, an operator may know how to investigate an IPBE server but not know how to investigate the customer's IP). Therefore, at each node of the tree, we identify all its equivalent attributes and represent the entire set by a single canonical attribute when expanding the tree. The threshold used to mark two, or more, attributes as synonyms is referred to as the *overlap probability* and is a user-configurable parameter that is typically set to a high value such as 0.99. However, when presenting the problem signatures to the operator, we show all the synonyms associated with the attributes identified in the signature.

The second complication is one of scalability. Because tens of thousands of attributes can be present in the dataset, a brute-force approach that expands the entire tree up to depth d is infeasible. To explore attribute groups optimally, we use a branch-and-bound algorithm [Land and Doig, 1960] to dynamically determine the maximum breadth of the tree to explore. Specifically, for each unexplored node of the tree n, we compute an upper bound for the maximum anomaly score that can be achieved by any child node n. If this upper bound is lower than the maximum anomaly score seen so far, then exploration of n is guaranteed to be fruitless, and it is discarded without further exploration. The upper bound of the anomaly score for a subtree, *e.g.*, *customer*1 in Figure 6.3, can be shown to be attained by assuming that there is a branch of that subtree that explains all the failed flows in the subtree, and has zero successes as computed by Equation 6.3.

$$\begin{aligned} KL^b(Y||X) &= \ln \frac{B(1, a+b-1)}{B(c,d)} - (1-c)\psi(c) - (a+b-1-d)\psi(d) \\ &\quad +(a+b-c-d)\psi(c+d) \end{aligned} \tag{6.3}$$

For example, if the attribute *node*15 was associated with 100 failed flows and 10000 successful flows, then the maximum possible anomaly score for the subtree anchored at *node*15

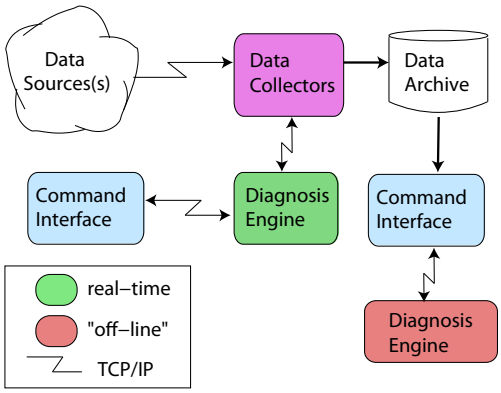


Figure 6.4. **Architecture of problem-localization engine.** The flexible architecture supports multiple data sources, and the problem-localization engines can run in either real-time or offline mode.

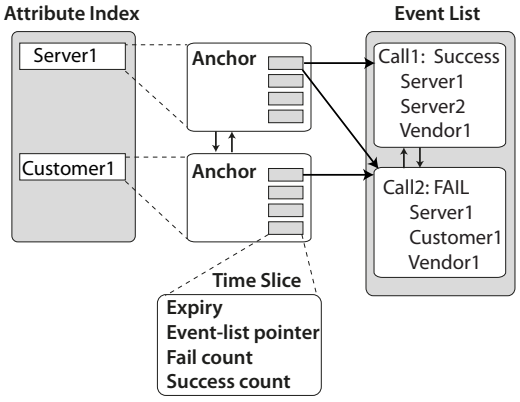


Figure 6.5. **Data structures that support problem-localization's scalable design.** We achieve high performance by maintaining in-memory indices of attribute and event data.

would be a branch with 100 failed flows and zero successes. We iteratively apply this algorithm in a greedy fashion to identify multiple concurrent problems by removing all flows (both success and failures) that match this identified problem signature from the dataset, and repeat the process. Doing so removes the impact of the first diagnosed problem and allows us to ask what explains the remaining failures. In this manner, we can identify separate independent failure causes (see Steps 2 and 3 in Figure 6.3). The average complexity of our algorithm is $M * N * D^r$, where M is the number of attributes, N is the number of flows, D is the average depth of the tree, and r is the average degree of nodes in the tree. The magnitudes of D and r are determined dynamically by the branch-and-bound algorithm.

6.3 Architecture and Design of Diagnosis Engine

We have implemented a prototype of the problem-localization approach, written in *C*, which is comprised of data collectors that process consolidated end-to-end flows to extract attributes of interest, and a diagnosis engine that outputs a ranked list of problems identified (see Figure 6.4). For the past two years, this prototype has been in active daily use by the operations team at the large ISP to analyze the production VoIP platform. The prototype is flexible since it can be easily extended to incorporate additional sources of information, such as software versions and Quality of Service (QoS) data. In addition, the prototype is scalable and capable of handling tens of millions of flows in real-time even when running on a single server.

The data collectors extract attributes from application-level logs, and archive the processed logs. Each data collector supports one or more data formats specified using configuration files, which increases the flexibility of our prototype. The data collectors also send data to the diagnosis engine, which implements the algorithms described above in Sections 6.1 and 6.2. The diagnosis engine can receive data from concurrent input sources (*i.e.*, multiple collectors) to reduce the amount of time needed to load data. The diagnosis engine can also be run in an offline mode by reading processed logs from the data archive.

The diagnosis engine considers each end-to-end flows as an event. The engine collects and manages events over a user-controlled time window of length T seconds (the operations team typically uses a window size of a whole day). Timestamp information in the event data is used to determine the bounds of the window; as new data is received, the window progresses forward and old events are aged off.

Performance was a primary concern while architecting the diagnosis engine as it is necessary to manage thousands of attributes from the production system in real-time. The filtering operations involved in the exploration of the search tree and in filtering out data that can be explained by a newly discovered problem signature, as described in Section 6.2, are the most expensive operations of each analysis. This is because each filtering operation must operate on the entire dataset consisting of both successful and failed events, which, despite reductions due to sampling, can still be very large. To construct appropriate data structures for this process, we use the observation that if each event is treated as a "document" that contains words corresponding to each attribute, then computation of the

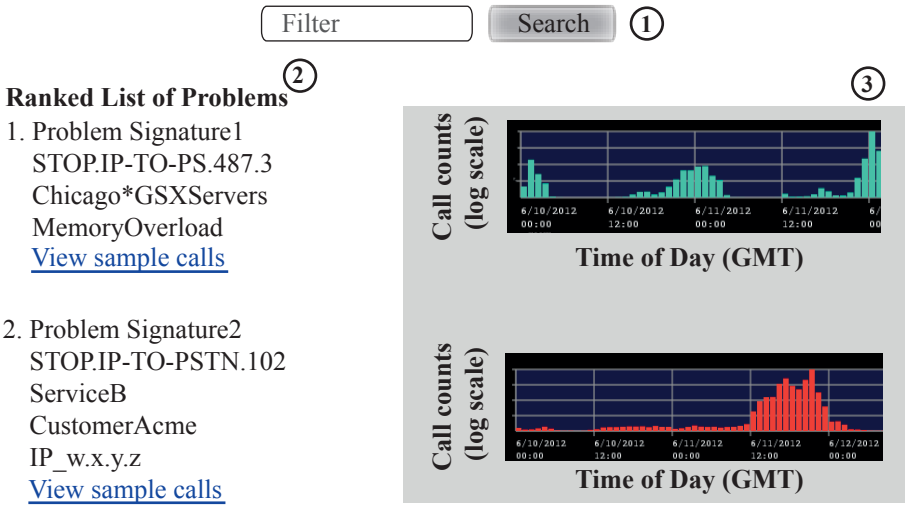


Figure 6.6. **Screenshot of problem-localization user interface.** The dashboard allows operators to: 1) specify a search criteria such as problems on given date; 2) view a ranked list of chronics diagnosed; and 3) identify recurrent problems using plots of affected calls.

anomaly score for an attribute group involves "searching" both the success and failure document sets for that group of attribute keywords, and counting the matches. Therefore, as shown in Figure 6.5, problem-localization's core data structures are constructed similarly to search engines—using an inverted hash table to index attributes.

The inverted index maps each attribute back to a linked list of events that contains the attribute. These mappings allow linear time computation of set intersections so that success and failure counts can be quickly constructed for any conjunction of attributes and negated attributes (to support exclusion of events that match previously discovered signatures). For each attribute, a series of success and failure counts are maintained based on the time slices. Managing the counts by time allows them to be adjusted as the time window rolls forward without the need to recount across all unexpired events.

6.3.1 Success Event Sampling

Due to the nature of chronics, the datasets processed usually have a disproportionately larger number of successful events as compared to failures. Sampling successful events as they are read by the diagnosis engine yields a significant reduction in overall memory utilization, and also significantly reduces the time to perform an analysis. To sample, we bin each successful event based on its time slice, and keep 1 out of every Nth successful event

Table 6.1. Examples of metrics extracted from black-box logs.

user	% CPU time in user-space
system	% CPU time in kernel-space
iowait	% CPU time waiting for I/O job
ctxt	Context switches per second
eth-rxbyt	Network bytes received per second
eth-txbyt	Network bytes transmitted per second
bread	Total bytes read from disk per second
bwrtn	Total bytes written to disk per second

in each bin. Unbiased random sampling preserves the correctness of the Bayesian estimation of success and failure distributions as described in Section 6.1, and thus preserves the correctness of the anomaly score of Equation 6.1. Its only impact is to reduce the number of success events, and thus the uncertainty of the success distribution. The results from our production runs, discussed in Chapter 8, shows that sampling does not appreciably impact accuracy, but does increase its speed by more than two orders of magnitude.

6.3.2 Visualization

Operators access the prototype via an interactive web-based user interface. Figure 6.6 illustrates how the web-interface facilitates the operator's workflow.

1. The operator searches for the date and the types of problems they are interested in analyzing. For example, operators can restrict the analysis to calls for a specified VoIP service on a given date.
2. Next, operators are directed to an interactive web-interface interface that ranks the top-20 problems diagnosed by our tool that match their filter, sorted by decreasing severity. Operators can gain more insight on the nature of the problem by viewing samples of calls affected via a drop-down option. The call samples display additional information from the call detail records, such as telephone numbers and call durations, that might not be captured by the problem signature.
3. A plot showing the frequency of the problem is displayed on the right, providing insight on the duration and severity of the problem.

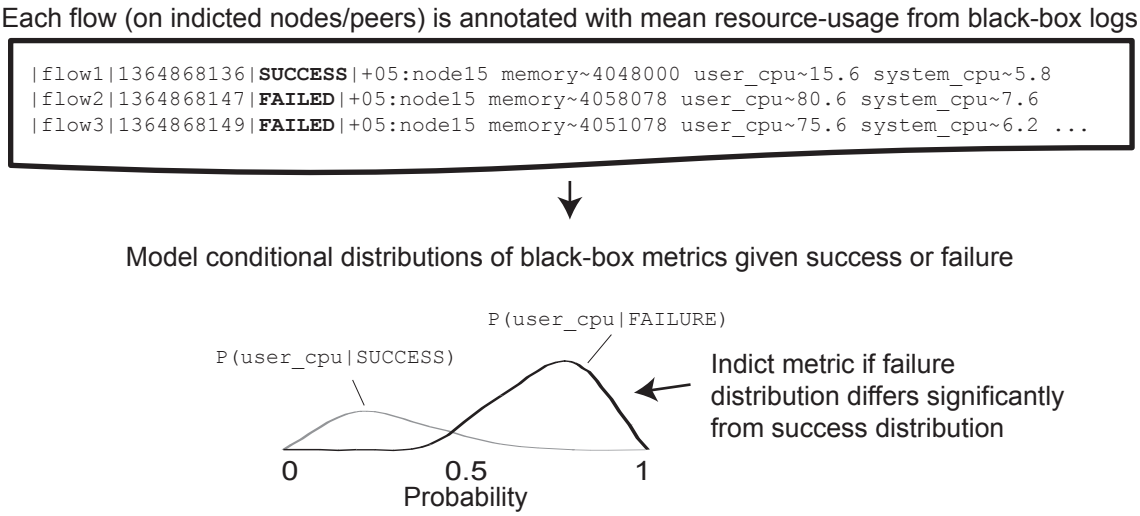


Figure 6.7. **Identifying black-box metrics most correlated with failures.**

6.4 Fusing Black-box Metrics

The Bayesian approach presented in Section 6.1 is scalable enough to be directly applied to the large numbers of discrete attributes present in our data sets. However, it is difficult to construct such numerically cheap comparison techniques to compare between success and failure distributions of real-valued data. To overcome this limitation, we analyze only a subset of the real-valued data that are linked to server/node attributes implicated by the Bayesian analysis.

Specifically, the real-valued data in the black-box logs includes performance logs of any servers within the service provider's network. These black-box logs include periodic measurements (at 1 second–15 minute intervals) of CPU and memory utilization, network traffic, disk I/O, and other OS-level metrics, as shown in Table 6.1. For each problem signature identified in Section 6.2, the tool considers only those performance measurements associated with servers present in the signature. We identify the resource-usage metrics that are highly correlated with the problem by annotating each flow that matches the problem signature with the mean resource-usage metrics gathered during the same time interval, as illustrated in the log snippet in Figure 6.7.

Next, we use the Wilcoxon rank-sum test [Mann and Whitney, 1947] to determine whether the conditional distribution of each metric in failed flows differs significantly from the conditional distribution of each metric in successful flows. The Wilcoxon rank-sum test

is a non-parametric test that does not assume that the data is drawn from any particular distribution, and assesses whether one of two samples of independent observations tends to have larger values than the other. We indict a black-box metric if we reject the null hypothesis that the observations were drawn from the same distribution. Comparing the distribution of metrics between successful and failed flows within the same time interval makes our tool more robust to seasonal variations in load (*e.g*, night- vs. day-time).

6.5 Why does it work?

A number of the characteristics of our problem-localization approach allow it to deal well with the challenges introduced by chronic problems.

1. Separation of concurrent root causes. Our iterative analysis allows us to separate and localize the different causes of many concurrent chronics that persist at any given time, even if they share attributes (*e.g.*, pass through the same network element). By repeatedly filtering out end-to-end flows that match dominant problems as they are detected, we expose increasingly smaller problems that may earlier have been hidden in the noise.

2. Identification of novel problems. An undiagnosed chronic problem is often novel, *i.e.*, something that operators have not seen before. For example, it may involve a new network element, or a new customer (with potentially novel configuration issues). Since the problem-localization algorithm does not rely on signatures of known defects, our approach can be used for problems that have never been seen before.

3. Identification of low-grade problems. Alarm-based approaches often miss chronics because of their low numbers might not cause significant perturbations within the service provider's network. Our tool ranks the identified causes of chronics using an anomaly score that compares the distribution of successful and failed flows, and identifies attributes discriminative of failures. The Bayesian inference we use can update the success and failure distributions with very few flows. Therefore, our approach can detect problems with very small numbers of failures.

4. Identification of complex triggers. Because our approach evaluates many groups of attributes against the scoring function, it can identify problems that occur only when multiple conditions (encoded by attributes) are satisfied at once.

5. Robustness to noise. Our approach is robust to noise introduced by occasional mislabeling during anomaly detection. Since we rank the problems identified by severity according to the number of flows affected, any spurious attributes introduced by noise typically receive a lower ranking. Ranking the problems by severity allows the operations team to effectively prioritize their troubleshooting efforts.

6.6 Summary

This chapter describes our problem-localization tool that relies on a scalable Bayesian distribution learner and an information-theoretic measure of distance to identify sets of problem signatures that together explain the differences between the failed and successful user interactions. Chronic problems are notoriously difficult to diagnose: 1) their small size makes setting alarm thresholds tricky; 2) there are many of them active concurrently even when the system as a whole is mostly functional; 3) their symptoms often overlap with each other; 4) they are triggered by complex corner cases involving multiple conditions; and 5) they they persist for lengthy periods and can get absorbed into the system's definition of what is normal.

We address these issues through a variety of techniques: 1) using top-down statistical diagnosis starting with abnormal user interactions to localize the root-cause of problems rather than relying on bottom-up alarms based on server logs; 2) statistically identifying root-causes by comparing bad interactions with good ones *from the same interval of time* rather than relying on historical data from good intervals of time; 3) a branch-and-bound procedure to identify complex triggers comprised of conjunctions of multiple attributes; 4) greedy filtering of failures explained by already identified problems to discover additional concurrent problems; and 5) fusion of black-box metrics to identify resource-usage metrics that are correlated with failures. Our tool also tolerates noise due to occasional mislabeling. Our tool also supports sampling of successful user interactions, which yields a significant reduction in overall memory utilization, and also significantly reduces the time to perform an analysis.

Chapter 7

Experimental Evaluation

This chapter describes the results of benchmarking our anomaly detection and problem localization algorithms (*i.e.*, diagnosis algorithm) in a controlled environment using fault-injection. Fault-injection is an invaluable technique for testing the effectiveness of a diagnosis algorithm under a variety of precisely controlled synthetic faults so that ground truth is known. We performed our fault-injection experiments on both Hadoop and VoIP, as summarized in Table 7.1. In Hadoop, we injected faults by emulating resource-hogs (*e.g.*, by running a disk-intensive process on a node) and task hangs. In VoIP, we injected faults by simulating server and customer problems using one-week's worth of call logs obtained from the production VoIP system at a major ISP.

Research questions. The research questions that we asked during our experimental evaluation were:

- Does knowledge of system dependencies improve diagnosis (Section 7.1)?
- Does fusion of white-box and black-box metrics provide insight on root-cause (Section 7.2)?
- Can we effectively diagnose low-probability faults (Section 7.3)?
- Can we effectively diagnose multiple ongoing problems (Section 7.4)?
- What is the impact of noise introduced by occasional mislabeling during anomaly detection (Section 7.5)?
- How does our approach compare to existing diagnosis algorithms that rely on machine-learning [Kiciman and Fox, 2005; Sambasivan et al., 2011] (Section 7.6)?

Table 7.1. **Summary of benchmarking approaches for Hadoop and VoIP.**

	Hadoop	**VoIP**
Workload	Gridmix cluster benchmark	One-week of ISP's call logs
Technique	Fault emulation	Fault simulation
Injected faults	Resource hogs/Task hangs 10 iterations per fault	Server/Customer problems 1000 simulated faults in total
Experimental setup	10-node EC2 cluster 4 1.2GHz cores, 7.5GB RAM	1 simulation node 8 2.4GHz cores, 24GB RAM

7.1 Impact of Knowledge of Dependencies

To assess the impact of the knowledge of system dependencies on diagnosis, we benchmarked our diagnostic approach, which detects and localizes problems using end-to-end flows, against an earlier implementation of our peer-comparison algorithm that relied on node-level comparisons to indict the faulty-node [Pan et al., 2009a]. The end-to-end flows captured dependencies across components in the distributed system, while the node-level peer-comparison approach did not capture dependencies across components.

7.1.1 Experiment Methodology

We conducted fault-injection on a 10-node Hadoop cluster running on the Amazon's EC2 cloud-computing system; fault-injection supplies a controlled environment where the ground truth is known. Each EC2 node had the equivalent of 7.5 GB of RAM and 4 1.2 GHz CPU cores, 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 of 5 different job types, ranging from an interactive workload that samples a large dataset, to a large sort of uncompressed data that accesses the entire dataset.

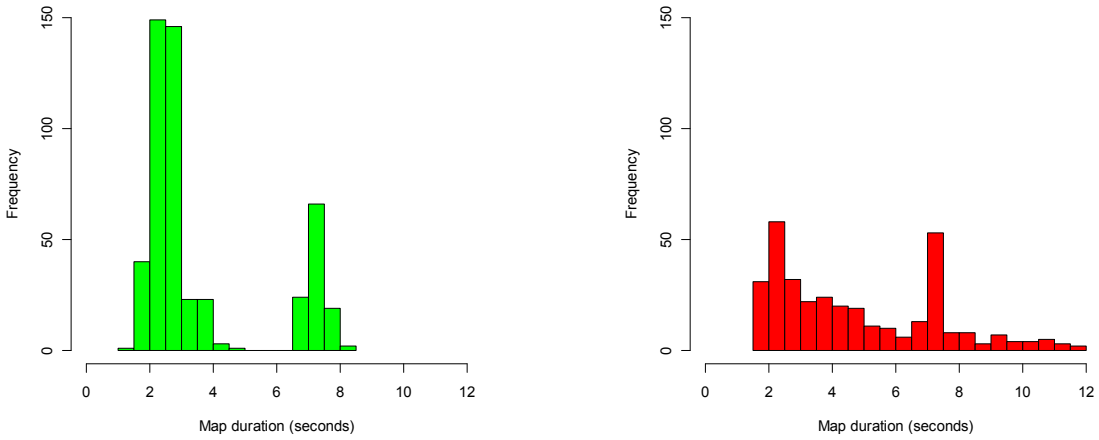
Table 7.2. **Injected faults in Hadoop, and the reported failures that they simulate.** HADOOP-xxxx represents a Hadoop JIRA entry.

Fault Type	[Source] Reported Failure	[Fault Name] Fault Injected
Resource contention	[Hadoop mailing list, Sep 26 2007] Excessive messages logged to file.	[DiskHog] Sequential disk workload wrote 20GB of data to filesystem.
	[HADOOP-2956] Degraded network connectivity between DataNodes results in long block transfer times.	[PacketLoss] Induce 50% packet loss.
Application bugs	[HADOOP-1036] Infinite loop at slave node due to an unhandled exception from a Hadoop subtask that terminates unexpectedly.	[HANG-1036] Manually revert to older version of Hadoop and trigger bug by throwing NullPointerException.
	[HADOOP-1152] Reduce tasks fail while copying map output due to an attempt to rename a deleted file.	[HANG-1152] Manually revert to older version of Hadoop and trigger bug by deleting file.

We injected a single fault on one node in each cluster to validate the effectiveness of our algorithms at diagnosing each fault. The faults emulate various classes of representative real-world Hadoop problems as reported by Hadoop users and developers in: 1) the Hadoop issue tracker [Apache Software Foundation, 2006] from October 1, 2006 to December 1, 2007, and 2) 40 postings from the Hadoop users' mailing list from September to November 2007. We describe our results for the injection of the four specific faults listed in Table 7.2. We ran 10 iterations of each fault.

Node-level Peer-comparison Approach

We benchmarked our diagnostic approach against an earlier implementation of our peer-comparison approach which relied on node-level peer-comparison to localize problems to the faulty node(s) [Pan et al., 2009a]. The node-level peer-comparison analyzed the distribution of white-box and black-box metrics on Hadoop slave nodes, and indicted a node as faulty if the distribution of metrics on the node differed significantly from its peers over a window of time. The white-box metrics comprised of Map and Reduce durations (we omitted heartbeats from our analysis because heartbeats are not typically logged in production environments due to the increased overhead). The black-box metrics comprised of 14 met-



(a) Histogram at majority of nodes.

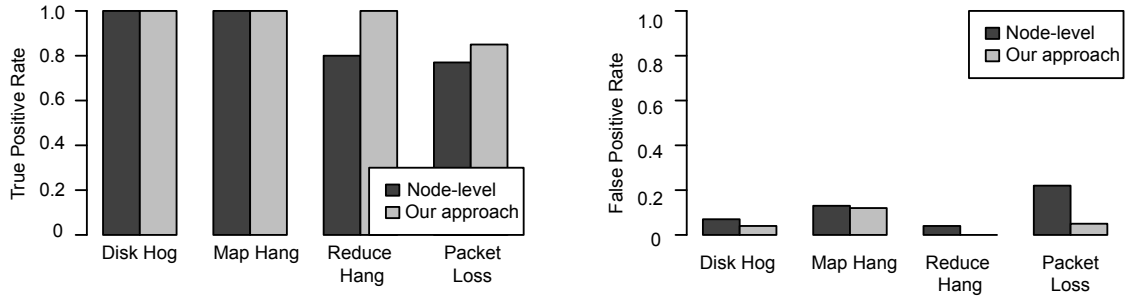
(b) Faulty node is "odd-man-out".

Figure 7.1. Histograms of Map durations at successful and faulty nodes.

rics from the pseudo-filesystem on the operating system (Table 6.1), and 2 TCP-related metrics from `netstat` monitored at 1-second intervals.

For each white-box metric and vector of black-box metrics, for a given period of time, we captured the behavior of the majority of nodes by generating a global/aggregate histogram using samples of metrics from all nodes. Next, for each node, we compared its histogram for that metric against the global histogram, resulting in O(n) comparisons. Figure 7.1 illustrates the global and per-task histograms for durations of Map tasks when a disk-intensive process ran on the faulty node.

Peer-comparison was done by computing a statistical measure (the Jensen-Shannon divergence [Kullback and Leibler, 1951]) which captured the distance between histograms. If a given node's histogram differed significantly (by more than a threshold value) from those of a majority of the other nodes, then, an alarm was raised for the slave node. An alarm is treated merely as a suspicion; repeated alarms are needed for indicting a node. Thus, an exponentially weighted alarm-count is maintained for each slave node. The slave node is then indicted when its exponentially weighted alarm-count exceeds a predefined value.



(a) True Positive rate.

(b) False Positive Rate.

Figure 7.2. **Benchmarking effectiveness of peer-comparison approaches].** Our anomaly detection algorithm achieved higher true-positive rates and lower false-positive rates than our earlier node-level peer-comparison approach.

7.1.2 Results

Figure 7.2 shows that the knowledge of system dependencies captured by our white-box approach improves the effectiveness of diagnosis when compared to the node-level peer-comparison approach. We measured the effectiveness of diagnosis by evaluating the outcome of each experiment as follows: an indictment of the fault-injected node was a true-positive, an indictment of any other node was a false-positive (FP), and a failure to indict the fault-injected node was a false-negative (FN). The true-positive (TP) and false-positive (FP) rates $\in [0.0, 1.0]$, with $TP = 1.0, FP = 0.0$ representing a perfect diagnosis. For the node-level approach, we reported the results of the Reduce, Map, or black-box analysis that yielded the highest true-positive rate in [Pan et al., 2009a]. The performance of both approaches was comparable for the disk hog and the hang during the Map phase (*Hang*1036). However, our approach performed better at diagnosing the hang in the Reduce phase (*Hang*1152) and the packet-loss fault. The hang in the Reduce phase did not perturb the system as much as the hang in the Map phase—resulting in lower true-positive rates for the node-level approach. Our diagnostic approach performed better because we detected problems using user-visible symptoms in end-to-end flows where the hang was more evident. The node-level approach experienced high false-positive rates for the packet-loss fault because this fault resulted in correlated problem manifestations across multiple nodes. Since the end-to-end flows captured dependencies between components, our approach was better placed to disambiguate the root-cause of the propagating problem.

Table 7.3. **Impact of fusion of white-box and black-box metrics on diagnosis.** The fusion of metrics can provide insight on the root-cause of the problem.

	Top Metrics Indicted	
Fault injected	**White-box**	**Black-box**
DiskHog	Maps	Disk
PacketLoss	Shuffles	-
Hang-1036 (Map hang)	Maps	-
Hang-1152 (Reduce hang)	Reduces	-

7.2 Impact of Fusion of White- and Black-box Metrics

We investigated whether the fusion of white-box and black-box metrics by our diagnosis algorithm provided insight on the root-cause of the problem.

7.2.1 Experiment Methodology

We evaluated the impact of fusing white-box and black-box metrics on our diagnostic approach by using the 10-node `GridMix` dataset, described in Section 7.1. We detected anomalies using the end-to-end flows generated from the white-box logs, and localized the problem to the faulty node(s). We then annotated flows on the indicted nodes with black-box metrics, as described in Section 6.4, and identified any black-box metrics that were correlated with failures.

7.2.2 Results

Table 7.3 shows the top white-box and black-box metrics indicted by our diagnostic approach. The fusion these metrics can provide operators with insight into the root-cause of a problem. For example, the slowdown experienced by Maps during the *DiskHog* fault is correlated with significant changes in disk-related black-box metrics. In addition, the Map and Reduce hangs perturbed the Map and Reduce metrics—pointing to an application-level problem. However, for some problems, such as the *PacketLoss* fault, our approach indicted the Shuffles but did not provide a clear indication on the root-cause of the problem using the black-box metrics. The operations team might be able to deduce that the performance degradation was due to a network problem using white-box metrics because it affected the

Shuffle phase, which is network-intensive. There were no obvious indicators of the problem in the black-box metrics because the *PacketLoss* fault caused drops in multiple metrics, such as CPU-, disk-, and network-usage since the faulty node received less work. In instances where the root-cause of the problem is not obvious from the indicted metrics, a technique that augments our black-box analysis approach with a signature-matching approach [Cohen et al., 2005; Bodik et al., 2010] could help operators diagnose recurrent problems.

7.3 Impact of Fault Probability

Since chronic performance problems affect a subset of users or requests, we investigated the impact of varying fault probability on the effectiveness our diagnosis algorithm. We varied the fault probability by simulating a variety of precisely controlled synthetic faults using one-week's worth of call logs from a major ISP. Fault simulation provided us with greater control over the range of faults that we could inject when analyzing the sensitivity of our algorithms to faults with different characteristics.

7.3.1 Experiment Methodology

We simulated faults using actual call-detail records (CDRs) of successful calls from the VoIP production system. We divided the CDRs into 1-hour intervals to yield 500 hourly traces. We injected faults by changing the labels of successful calls, which contained attributes of interest, to failed calls. The attributes of interest were individual network elements, customer sites, links (routes), and their combinations. These attributes were selected because they were the most common features tracked by the operations team at the large ISP. Faults were generated by picking these attributes randomly. Therefore, we can claim qualitative representativeness (what attributes), but not statistical representativeness (distribution of attributes) compared to the ground truth failure distribution, which is unknown, even to the operations team.

We randomly varied the combination of attributes associated with each fault from 1 to 3, and ensured that these attributes were not synonyms of each other. Each injected fault recurred during two 15-minutes in each hourly trace. We also varied the number of independent faults in each hourly trace from 1 to 3. The probability of each fault injected ranged from 1% to 10% of calls containing the chosen attributes. We evaluated the effectiveness of

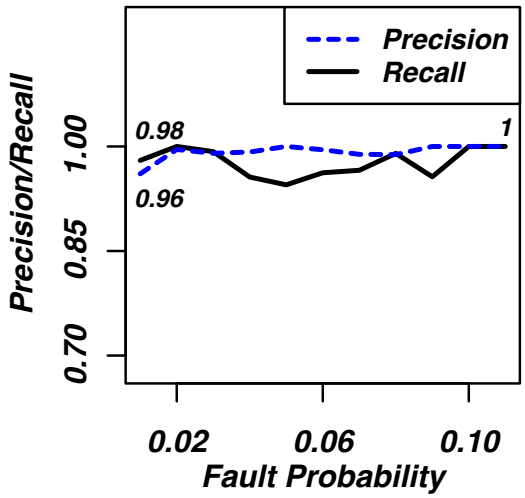


Figure 7.3. **Effect of varying fault probability on diagnosis.** Precision and recall remained relatively constant despite variations in fault probability.

problem localization based on the rank of the correct root-cause in the diagnostic output, and computed *recall* and *mean-precision*. Recall is the fraction of injected faults that were correctly identified in the top-20 root-causes (Equation 7.1). Mean-precision is a measure of the false positive rate that we used to analyze the quality of the ranked search results. We computed the precision of each problem signature as the fraction of indicted attributes that are relevant to the injected fault (Equation 7.2). The mean-precision is the arithmetic mean of the precision scores for the set of experiments (see Equation 7.4). A high precision indicates low false positive rates.

$$Recall = \frac{TruePositives}{TruePositives + FalsePositives} \tag{7.1}$$

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{7.2}$$

$$MeanPrecision = \frac{\sum_{i=1}^{N} \frac{\sum_{j=1}^{R}(Precision(j))}{R}}{N} \tag{7.3}$$

where, R=Number of ranked root-causes per experiment

N = Number of experiments

7.3.2 Results

We successfully diagnosed 97% of faults injected, with lower than 4% false positives in the fault-simulation dataset. All the false negatives occurred when we injected two faults that were logically independent, but happened to share a large intersection of attributes correlated with the faults. In these cases, we typically reported a single root-cause that listed the shared attributes. Figure 7.3 shows that we correctly identified the root-cause of injected faults despite variations in the fault probability from 1% to 10% of calls containing the attributes of interest. Our precision and recall remained relatively constant at $>$96% and $>$94% respectively, even when faults affected only 1% of calls with a given set of attributes.

7.4 Impact of Multiple Ongoing Problems

We used fault simulation to investigate whether our diagnostic approach could diagnose complex failures involving multiple attributes, and could discriminate between the root-causes of multiple ongoing problems.

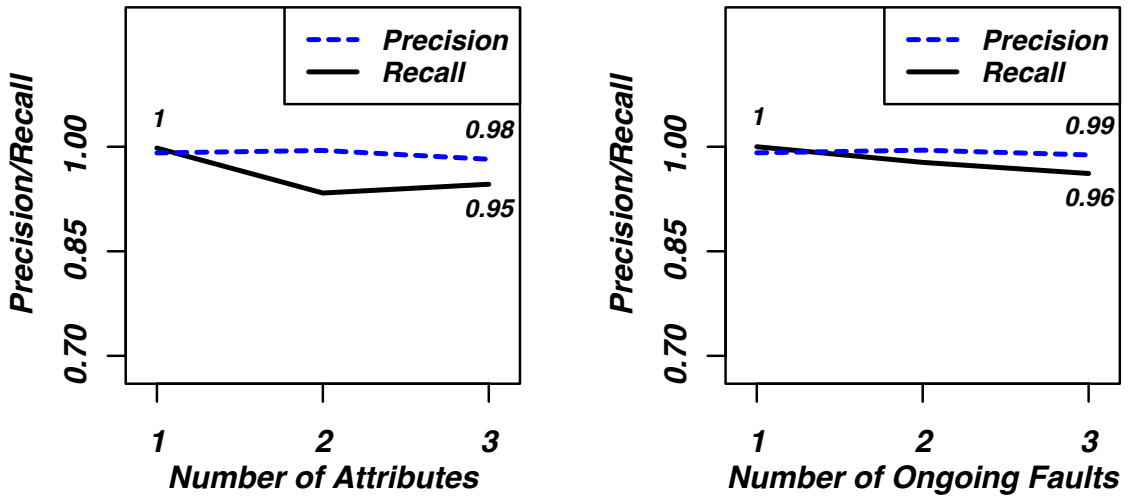
7.4.1 Experiment Methodology

We evaluated the impact of varying the combination of attributes associated with a fault, and the number of ongoing faults using the fault-simulation dataset generated from one-week's worth of call logs from a major ISP (described in Section 7.3). The combination of attributes associated with each fault in the dataset randomly varied from 1 to 3, while the number of ongoing faults in each hourly trace varied from 1 to 3. We measured the effectiveness of our diagnostic approach by computing recall and mean-precision.

7.4.2 Results

Figure 7.4(a) shows that we correctly diagnosed chronics triggered by the interaction of two or more attributes. Precision and recall were slightly degraded from 99% to 98%, and 99% to 95% respectively for chronics involving multiple attributes. As explained above, this drop in recall was due to the presence of faults that were not truly independent rather than the number of attributes associated with each fault.

Figure 7.4(b) shows that performance was relatively unaffected when multiple ongoing faults were injected—recall dropped slightly from 1 to 0.96 when three concurrent faults



(a) We correctly identified complex problems involving a combination of attributes.

(b) We effectively diagnosed multiple ongoing faults.

Figure 7.4. **Effect of varying combination, and number of faults on diagnosis.**

were present. Overall, we correctly ranked 97% of the relevant root-causes within the top-3 likely causes of chronics. This high ranking of likely root-causes allows operators to quickly focus their attention on the most pressing issues impacting end-users.

7.5 Impact of Noise

We used fault simulation to investigate the impact of noise due to occasional mislabeling during anomaly detection. Noise can arise due to the thresholds used to detect anomalous user-interactions—higher thresholds increase the number of false negatives, while lower thresholds increase the number of false positives.

7.5.1 Experiment Methodology

We evaluated the impact of noise using the fault-simulation dataset generated from one-week's worth of call logs from a major ISP (described in Section 7.3). We simulated noise by incorrectly labeling 5–20% of failed calls as successful, and randomly labeling an equivalent number of successful calls as failed. We measured the effectiveness of our diagnostic approach by computing recall and mean-precision.

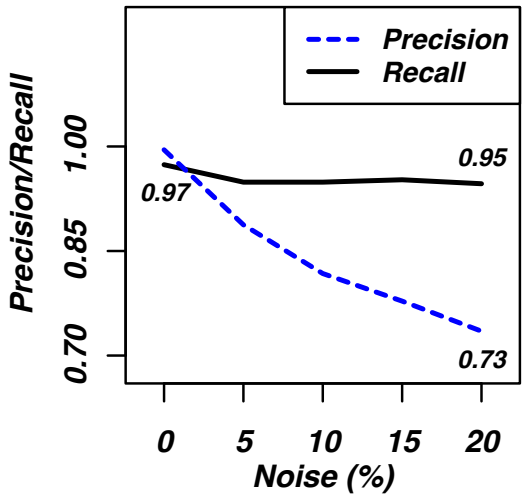


Figure 7.5. **Effect of noise on problem localization.** Recall is robust to noise, but its precision is degraded in proportion to noise.

7.5.2 Results

Figure 7.5 shows that recall is robust to noise, and that precision is degraded in proportion to noise. The drop in precision is due spurious attributes introduced by the incorrect labels. Our ranking of likely causes remained robust to noise—even when 20% of failed calls were mislabeled, we correctly identified >94% of injected faults within the top-3 likely causes.

7.6 Benchmarking Against Existing Algorithms

We benchmarked our approach against Pinpoint [Kiciman and Fox, 2005], and a modified version of Spectroscope [Sambasivan et al., 2011]. These diagnosis algorithms are most similar to our approach as they rely on truth tables, and decision trees that use information-theoretic splitting functions to identify attributes most indicative of failures.

7.6.1 Experiment Methodology

We implemented Pinpoint [Kiciman and Fox, 2005], and a modified version of Spectroscope [Sambasivan et al., 2011] using decision trees generated by *See*5 [RuleQuest Research Data Mining Tools, 2011]—an open-source implementation of the C5.0 algorithm written in C++. The C5.0 algorithm is an extension of the C4.5 decision tree algorithm, which also

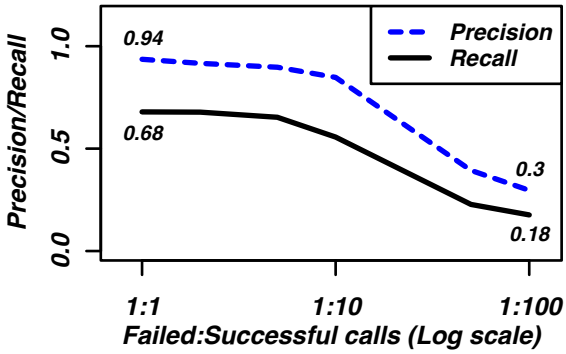


Figure 7.6. **Influence of ratio of failed to successful call on decision-tree performance.** Performance of the decision tree degrades significantly when successful calls greatly outnumber failed calls in the dataset.

uses information gain to split the tree. C5.0 uses memory more efficiently, and supports additional classification features such as boosting.

Pinpoint

We implemented Pinpoint [Kiciman and Fox, 2005] by training a decision tree using the labeled failed and successful calls. We then diagnosed problems by examining each branch in the decision tree whose leaf node classified failed calls, and ranked the branches based on the number of failed calls. We observed that precision and recall were primarily influenced by the ratio of failed calls to successful calls in the dataset, as shown in Figure 7.6. We varied this ratio by randomly sampling successful calls, while leaving the number of failed calls unmodified. The best performance was achieved when the ratio of failed to successful calls was similar. Weiss and Provost [Weiss and Provost, 2003] explain that the performance of decision tree algorithms is degraded when class distributions are imbalanced—these imbalances are commonplace when diagnosing chronics as the number of successes significantly exceeds the number of failures. An example of this degraded performance is shown in Figure 7.6 where recall dropped to 18% when the number of successful calls outweighed the number of failed calls by a factor of 100. In this case, often the best predictor was a decision tree with no branches that always predicted success.

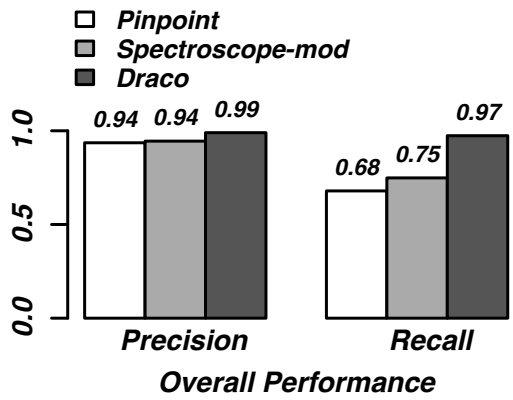
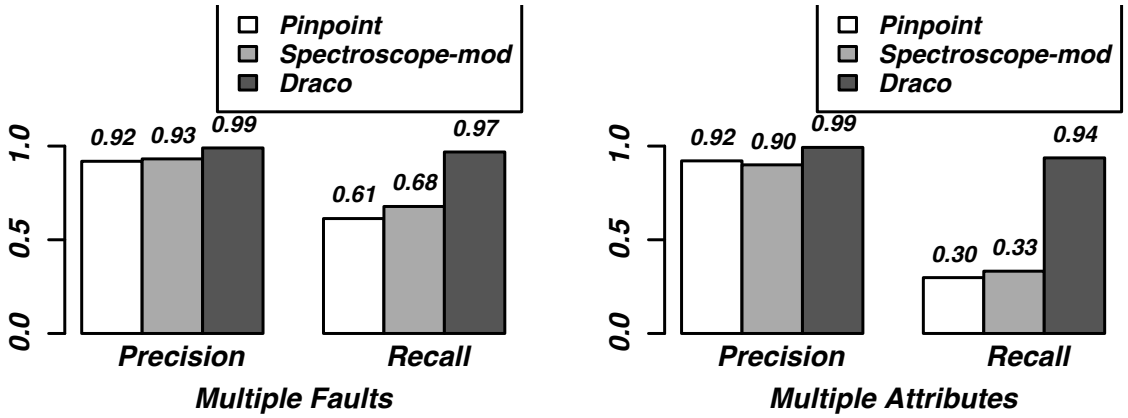


Figure 7.7. **Benchmarking our approach against Pinpoint and Spectroscope-mod.** Overall, we performed better than both Pinpoint and Spectroscope-mod at diagnosing chronics.

Spectroscope

Spectroscope [Sambasivan et al., 2011] localizes the source of performance degradations between two periods or executions of a system to just a few relevant components. It does so by leveraging the insight that such changes often manifest as changes or *mutations* in the structure of individual requests (*e.g.*, the components visited, the functions executed, *etc.*) or in their per-component latencies. Spectroscope identifies mutated request flows from the problem period, and localizes the problem by showing how they differ from their *precursors*—the way they were serviced in the non-problem period. Additional localization is performed by using a decision tree to identify low-level parameters (*e.g.*, function calls) that best differentiate a mutation from its precursor.

The fault models for Spectroscope and our approach are different—Spectroscope targets problems that result in significant performance degradations, whereas our approach targets chronics. Therefore, we implemented a modified version of Spectroscope-mod where successful calls represent the non-problem period, and failed calls represent the problem period. We investigated whether sampling successful calls using the notion of *precursors* (*i.e*, successful calls that were similar, but not identical to failed calls), yielded better results than the random sampling we employed for Pinpoint. We identified precursors by sampling successful calls whose string-edit distance from failed calls was below a predefined threshold. We then localized the root-cause of the problem using decision trees.



(a) Our tool's recall was higher by up to 20% for traces containing multiple independent faults; Spectroscope-mod performed 6% better than Pinpoint for these traces.

(b) We outperformed both Pinpoint and Spectroscope-mod, with a recall of up to 56% better for complex chronics triggered by a combination of 2 or more attributes.

Figure 7.8. **Benchmarking effect of complex failure modes on Pinpoint and Spectroscope-mod.**

7.6.2 Results

Figure 7.7 summarizes the overall mean-precision and recall of Pinpoint[1], Spectroscope-mod, and our approach when diagnosing injected faults, in the absence of noise. We performed better than both Pinpoint and Spectroscope-mod by identifying 97% of injected faults with an average precision of 99%. The precision of Pinpoint and Spectroscope-mod were comparable at 94%. Spectroscope-mod's recall was 7% higher than Pinpoint's demonstrating that strategic sampling of success data can improve performance.

The differences in performance between our approach and the decision tree approaches were more pronounced when we limited our analysis to fault-injection traces that either contained multiple independent faults, or chronics triggered by complex corner cases involving a combination of two or more attributes. We correctly diagnosed up to 36% more injected faults for traces containing multiple independent faults, as illustrated in Figure 7.8(a). We significantly outperformed the decision tree approaches for chronics triggered by a combination of two or more attributes, achieving a recall of up to 64% higher as shown in Figure 7.8(b).

[1]For Pinpoint and Spectroscope-mod, we sampled successful calls to yield a 1:1 ratio of failed to successful calls, which provided the best performance.

The reasons for the degraded precision and recall for Pinpoint and Spectroscope-mod are outlined below:

1. The decision tree performed poorly at diagnosing faults injected with very-low probabilities, particularly in traces containing multiple concurrent faults. In these instances, the decision tree algorithm would split the tree to classify faults occurring at higher probabilities, thereby masking faults with lower probabilities.
2. The performance of the decision tree is degraded when chronic problems arise due to a combination of attributes because it identifies some, but not all relevant root-causes. We took great care to ensure that the combination of attributes associated with each injected fault were not synonyms of each other to eliminate this as contributing factor to the poor performance. We investigated the effect on performance of decision trees of considering partial matches where at least one of the affected attributes is identified. In this case, the recall of both Pinpoint are Spectroscope-mod improved to 83% suggesting that the decision tree was pruning relevant features.

7.7 Summary

This chapter explores the effectiveness of our diagnostic approach in a controlled environment using fault-injection. Our experimental evaluation shows that the knowledge of system dependencies captured by the end-to-end flows improves the effectiveness of diagnosis—particularly when the symptoms of the problem are not severe enough to trigger low-level alarms, and when faults result in correlated problem manifestations across multiple nodes in a distributed system. We also observed that the fusion of white-box and black-box metrics can provide operators with additional insight on the root-cause of the problem. Our fault-simulation experiments showed that our diagnosis approach can provide coverage levels as high as 97% with false positives as low as 4%, and outperformed state-of-the-art diagnostic techniques by up to 64% for chronics triggered by a combination of interacting factors. Our diagnosis approach effectively localized the root-causes of multiple ongoing problems, and is robust to noise due to occasional mislabeling during anomaly detection.

Our empirical evaluation also highlighted some limitations with our approach. False negatives occurred when we injected two or more faults that were logically independent,

but happened to share a large intersection of attributes correlated with the faults. In these cases, we typically reported a single root-cause that listed the shared attributes. In addition, our diagnostic approach provided a coarse-grained localization of the root-causes of the chronic problems. For example, we may localize the root-cause of the problem to a particular server or customer, but not to the exact line of source code or the configuration parameter causing the problem. The granularity of our diagnosis is limited by granularity of information available in the logs that we analyzed. The symptoms of some faults, such as packet-loss, may not always provide an obvious pointer to the root-cause. In these instances, augmenting our diagnostic approach with a signature-matching approach [Cohen et al., 2005; Bodik et al., 2010] could help operators diagnose recurrent problems.

Chapter 8

Case Studies

In this chapter, we describe how we used our diagnosis approach to detect and localize real-incidents problems in the OpenCloud cluster (Section 8.1), and the production VoIP system at a major ISP (Section 8.2). We have deployed our problem-localization tool on a portion of wireline VoIP services provided by the ISP. Over the past two years, our tool has assisted operators at the ISP in performing chronics analysis of dropped and blocked calls on the production system.

8.1 Hadoop Case Studies

We evaluated the effectiveness of our diagnosis approach by performing a post-mortem analysis of real incidents in the OpenCloud Hadoop cluster. These incidents were drawn from OpenCloud's issue tracking system, and hardware replacement logs described in Section 3.3. Troubleshooting chronic problems in Hadoop can be daunting. We have observed instances where the Hadoop job threw cryptic exceptions which had no direct mapping to the underlying problem. Due to the large-scale of Hadoop jobs, a single job can throw hundreds or even thousands of exceptions which can overwhelm the user. Conversely, a small number of straggler-tasks waiting to access remote data from HDFS (Hadoop Distributed File System) can slow down the progress of the entire job. The dependencies between HDFS and the MapReduce tasks are not always clear because Hadoop maintains separate user-interfaces for MapReduce jobs and the HDFS instances. We have also observed that multiple ongoing problems are common in the cluster. Table 3.6 shows that 14% of reported incidences involved multiple ongoing problems. This number serves as a

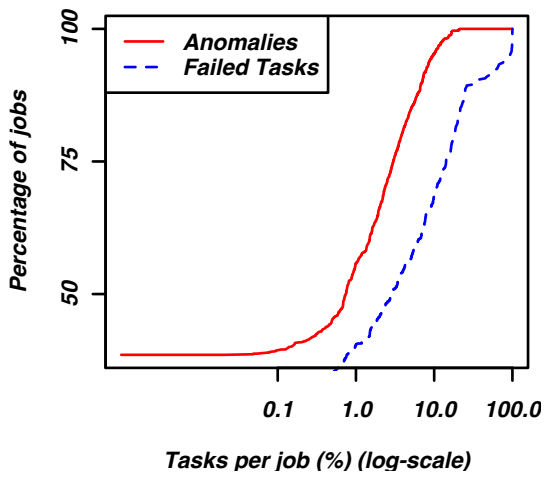


Figure 8.1. **Percentage of failed and anomalous Hadoop tasks over a 2-week period.**

conservative estimate because it does not include problems due to bugs in the MapReduce application that mostly go unreported.

We detected chronic problems in Hadoop by mining task exceptions from the application-level logs, and by applying our anomaly-detection algorithm described in Section 5.1. To gain insight on the quality of the models generated by our stepwise-regression approach, we computed the percentage of tasks flagged as anomalous or failed in each Hadoop job over a two-week period. There were 890 jobs executed during this time-period. On average, we flagged 2.25% of tasks as anomalous. The maximum percentage of anomalies flagged in a single job was 21%. The mean percentage of tasks aborted per job by the TaskTracker was 13.27%, with a maximum of 99.91%. We then labeled the end-to-end flows associated with each task as successful or failed. Our problem-localization approach used these labeled traces to localize chronic problems. We provide examples of real incidents in the Hadoop cluster that were diagnosed by us.

8.1.1 Examples of Chronics in Hadoop

Incident 1. A user's job was constantly failing with the cryptic errors, such as `Task process exit with nonzero status of 1`. The administrators initially suspected that the job was running out of memory because there were a few `out-of-memory` exceptions in the logs. The job failures were eventually traced back to multiple problems in the

cluster. The first problem was due to a bug in the network driver on a node. The second issue was a bad disk on a separate node. We successfully identified both these nodes.

Incident 2. We localized a recurrent application-level problem in a user's job. The user submitted four successive jobs which failed due to an error configuring an object. These jobs threw hundreds of exceptions which might have overwhelmed the user. We ranked these jobs as the top-4 problems during that day, and also identified the exception that was most correlated with the failed Map tasks in one of these jobs.

Incident 3. We manually inspected some of the jobs with high-rates of anomalies to determine whether the linear regression models for anomaly-detection were a poor fit, or whether there was an unreported problem in the cluster. We discovered two nodes whose that were repeatedly indicted by us. We suspect that these nodes might have been experiencing a hardware problem that went unnoticed.

Incident 4. We were unable to localize a disk problem on a node because Hadoop blacklisted the node. Hadoop has built-in fault-tolerance mechanisms that migrate tasks from faulty nodes, and blacklists them if their performance is significantly poorer than their peers. Since blacklisted nodes do service user jobs, our diagnostic approach is unable to localize problems on these nodes because they do not impact user performance. Node-level peer-comparison approaches [Tan et al., 2010a; Kasick et al., 2010] can be used to complement our approach in these cases.

8.2 VoIP Case Studies

We evaluated the effectiveness of our approach using a diverse set of real incidents from a production telecommunication system, listed in Table 8.1. Some of these chronics recurred several times a day, while others occurred intermittently over several days or weeks. The ranking of the individual chronics varied from day-to-day depending on the severity of the problem. For example, at its peak, the trunk-group problem due to blocked circuit-identification codes (incident 3) only affected 2-3% of the calls passing through this trunk group. The root-causes of the chronics included configuration problems at the customer premises, resource contention, software problems, and an intermittent power-outage. We correctly localized the network element or customer associated with the chronic problem in

Table 8.1. **Examples of chronics at the production VoIP system.** We correctly diagnosed 8 out of 10 incidents and ranked them among the top-20 problems identified. We also identified anomalous resource-usage metrics whenever performance logs were available.

Examples of problems	Type	Diagnosed	Resource anomalies
1. Customers use wrong codec to send faxes abroad.	Configuration	✓	-
2. Customer problem causes recurrent blocked calls at IPBE.	Configuration	✓	-
3. Blocked circuit identification codes on trunk group.	Configuration	✓	-
4. Problem with customer equipment leads to poor QOS.	Configuration	✓	-
5. Congestion at gateway servers due to high call volumes.	Contention	✓	CPU/Concurrent sessions
6. Performance problem at application server.	Contention	✓	CPU/Memory
7. Debug tracing overloads servers during peak traffic.	Contention	✓	CPU
8. Software problem at control server causes blocked calls	Software bug	✓	-
9. Policy server not responding to invites from application servers.	Software bug		Low responses at app. server
10. Power outage and unsuccessful failover causes brief outages.	Power		-

8 out of these 10 incidents. Once a problem is localized, operators can promptly liaise with customers, or query logs outside our scope to diagnose the problem in more detail. The two incidents in which we did not implicate the correct element were a software problem in a policy server (incident 9), and a power outage that resulted in intermittent problems during failover (incident 10). In both incidents, the network element that was the root-cause of the problem was not present in our input data so we indicted the network elements adjacent to the root-cause.

In addition to localizing network elements associated with the chronic problem, we analyzed the performance logs of the identified network element whenever they were avail-

able. We flagged a resource metric as anomalous if the distribution of the metric in failed calls was significantly different from that in successful calls. We used the Mann-Whitney rank test to reject the null hypothesis that the real-valued metrics associated with failed and successful calls were drawn from the same distribution with a significance-level of 1%. The test helped to localize problems due to resource-contention at a network element.

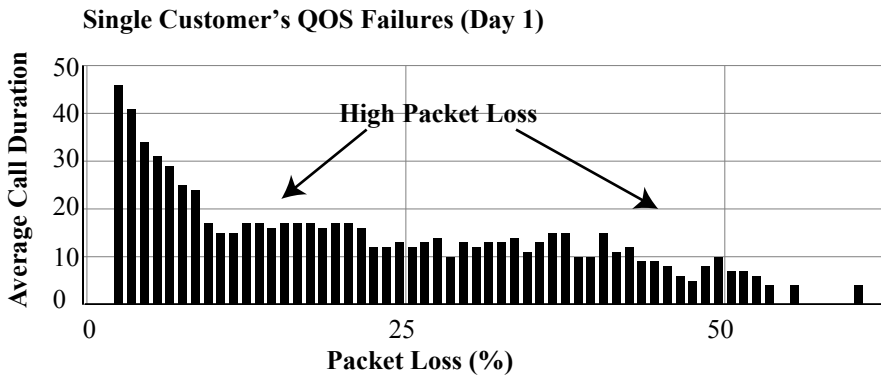
8.2.1 Examples of Chronics in VoIP

We highlight six case studies from Table 8.1, to illustrate how our tool has been used by the chronics team quickly to identify several new problems.

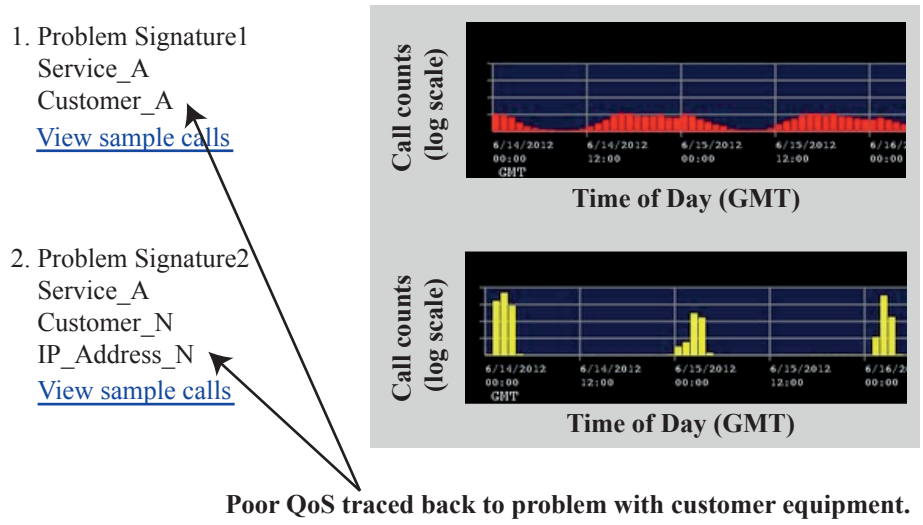
Incident 1. A repeating increase in the number of defects during night hours was observed associated with a given defect code illustrated in Figure 1.1. Our analysis identified two different (business) customers as being associated with the bulk of the defects. While these customers accounted for large share of total defects, the defect rate observed by the customers were a fraction of one percent. The operations team determined that these two customers were attempting to send faxes overseas using unsupported codecs during US night time. Shortly after the date the customers were notified of the problem, the daily defect count associated with this defect code decreased by 56%.

Incident 2. Our analysis identified an independent problem with a specific network element that occurred concurrently with incident 1 (see Figure 1.1), and accounted for over 50% of the remaining defects when failures due to Incident 1 were excluded. Again, overall only a fraction of one percent of the calls passing through this element were failing making the problem harder to identify. After the operations team reset the element, the total number of daily defects associated with this defect code was reduced by 76%, and this element was no longer implicated by our analysis.

Incident 3. An increase in failure rate during business hours was observed for a single defect code (see Figure 1.2). Our analysis identified a trunk group as being associated with up to 80% of these defects. At peak, 2-3% of the calls passing this trunk group would fail. Analysis by the operations team revealed two blocked CICs (Circuit Identification Codes) on the trunk group and as a result the problem would only affect calls assigned to these blocked CICs (in a round robin manner). After those CICs were unblocked, the total defects associated with this code were reduced by 80%.



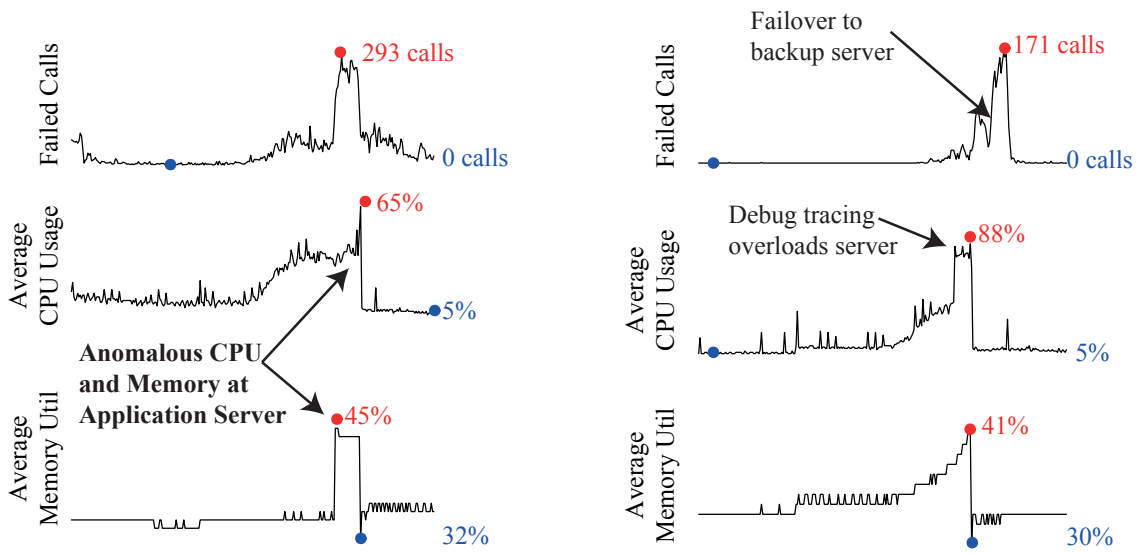
(a) A customer was experiencing poor Quality of Service (QOS) due to high packet-loss rates.



(b) We correctly localized the QOS problem to misconfigured customer equipment.

Figure 8.2. **Diagnosis of Quality of Service (QOS) violation in VoIP system**

Incident 4. Poor call quality (due to packet delay, jitter, and packet loss) is a chronic problem that is difficult to detect because the call is neither blocked nor dropped and thus appears as a successful event from the system's point of view. The quality of a VoIP call is determined by factors such as packet delay, jitter, and packet loss. To diagnose poor call quality, the gateway servers were configured to log the message-loss percentage for each call. In addition, the data collector was modified to ignore failed (dropped and blocked) calls and treat the set of calls with poor quality (loss $>$ threshold) as the new set of failed calls. The resulting data can then be analyzed normally by the diagnosis engine. The tool indicated that the top quality of service issue (approximately 48% of all poor quality calls)



(a) Intermittent performance problem at application servers perturbs memory and CPU usage.

(b) Debug tracing overloads server during peak traffic.

Figure 8.3. **Localizing resource-usage problems in VoIP network.**

was related to a single business customer. Further, we did not implicate any network elements indicating that the root-cause of the problem was likely with the customer equipment and not a problem with the ISP's hardware and/or network. A quality investigation conducted by the ISP, independent of our analysis, identified the same customer and also concluded that the problem was being caused by customer equipment thus confirming our analysis. When the customer was notified, and the problem corrected, the overall number of quality of service failures was reduced as expected.

Incident 6. An intermittent performance problem with two application servers led to an increase in call defects persisting for several days. This problem affected 0.1% of all calls passing through these application servers (see Figure 8.3(a)). We identified both servers affected by the problem. After the operations team failed over traffic to a backup server, the number of defects was reduced by 85%. We analyzed the CPU, memory and network-related metrics on the application servers and observed that these failures occurred during periods of heavy load and high CPU usage.

Incident 7. Customers experienced call failures following an autonomous failover of the control servers in the service provider's network. Although the failover completed successfully, some customers using the Bridge Line Appearance (BLA) feature lost synchroniza-

Table 8.2. **Performance of problem-localization tool.** Average data load time, average number of nodes in a diagnosis tree, and mean analysis time to generate the top 20 diagnoses for more than 30 million calls.

Mode			Load Time	Nodes	Analysis Time
Branch & Bound	Sampling	Restricted			
NO	NO	YES	374 ± 29sec	429 ± 208	524 ± 128sec
YES	NO	YES	374 ± 29sec	12 ± 5	128 ± 53sec
YES	NO	NO	374 ± 29sec	36 ± 20	880 ± 124sec
YES	YES	NO	120 ± 7sec	40 ± 30	16 ± 6sec

tion with the platform. Post outage analysis determined that failover was caused by a CPU spike, which was likely triggered by increased logging levels implemented to help isolate an intermittent BLA issue (see Figure 8.3(b)).

Incident 9. A chronic problem arose when a policy server in the VoIP network stopped responding to invites from application servers, and affected 0.4% of calls passing through the application server. Since records for the policy server were not present in the master CDRs that we analyzed, we implicated the application servers that were sending invites to the policy server. An analysis of the performance logs at the application server indicated that low response rates were an additional symptom of the problem. Although in this instance, our tool did not identify the root-cause, our analysis provided useful clues to operators to help localize the problem. The incorporation of more server logs, such as policy server and router logs, would improve our ability to localize problems. Our flexible architecture allows us to incorporate additional data sources as they become available.

8.3 Performance of Problem Localization

We ran our experiments on a 8-core Xeon HT (@2.4GHz) with 24GB of memory. Depending on the operating mode, our tool takes 2 to 6 minutes to load input data, and from 16 seconds to over 10 minutes to analyze input data comprised of more than 30 million calls; see Table 8.2 for details.

In the initial implementation, the tree size was limited in order to achieve acceptable analysis times. Enabling the branch-and-bound algorithm described in Section 6.2, while

continuing to limit the tree size, resulted in more than a 50% performance improvement in the analysis time. However, the branch-and-bound algorithm alone does not provide enough of a performance gain to allow the restrictions on tree size to be removed; doing so caused the analysis times to exceed those of the initial implementation. Sampling (at the rate of 1/200 of successful calls), when used in combination with the branch-and-bound algorithm, does allow the restrictions on tree size to be lifted while reducing analysis times to a near interactive level. The reduction of data load time by more than 60% is another benefit to the use of sampling.

The problem signatures generated when sampling have a 97% match rate when compared to those generated when all success data is used. Specifically, the analysis of several days' data yielded 220 problem signatures, but only 214 matching signatures were produced by the analysis using sampled input. Of the six unmatched signatures, all but one were ranked either 19^{th} or 20^{th} (out of 20); the exception was ranked 13^{th}.

8.4 Summary

This chapter provides examples of real chronics that our diagnosis approach detected and localized in a Hadoop cluster and a production VoIP system. We successfully identified different classes of chronic problems ranging from configuration problems at the customer-site to resource-contention within the service provider's network. Our tool has been deployed on a major VoIP platform serving millions of users and handling tens of millions of calls a day, and is being successfully used by its operations team. The top-down statistical approach that we adopted allowed us to incrementally incorporate instrumentation from different layers of the system. The granularity of the diagnosis that we provided is dictated by the instrumentation data available.

The greatest value of a picture is when it forces us to notice what we never expected to see.

J. W. Tukey, Exploratory Data Analysis, 1977

Chapter 9

Problem Visualization

Automated diagnosis tools help narrow down the possible root-causes of problems in production systems. However, the use of automated diagnosis techniques in isolation is not always sufficient to localize problems at the level of granularity desired by users. Visualization tools help bridge this gap by providing interactive interfaces that allow users to explore their data, and formulate their own hypothesis about the root-cause of problems. Our problem-localization tool supports visualization of time-series data by displaying the number of calls affected by chronic problems over time. This visualization helps operators spot recurring chronic problems that are affecting customers (see Figure 6.6).

A number of instrumentation frameworks also support visualization of time-series data. [Ganglia, 2007; McLachlan et al., 2008; Ren et al., 2011] use plugins to visualize time series of performance metrics across a cluster. Artemis [Cretu-Ciocarlie et al., 2008] provides a pluggable framework for distributed log collection, data analysis, and visualization. Request-tracing infrastructures [Dai et al., 2011; Fonseca et al., 2007; Sigelman et al., 2010] support for visualizing requests whose causal structure or duration are anomalous. [Tan et al., 2010b] describes how we visualized the causal structure of Hadoop jobs to support performance debugging. We developed a visualization tool that highlights infrastructural and application-level problems in a Hadoop cluster, as shown in Figure 9.1.

Research questions. The research questions that we asked during problem visualization were:

- How to develop compact visualizations of problems in large production systems?
- Can these visualizations help us diagnose different classes of problems?

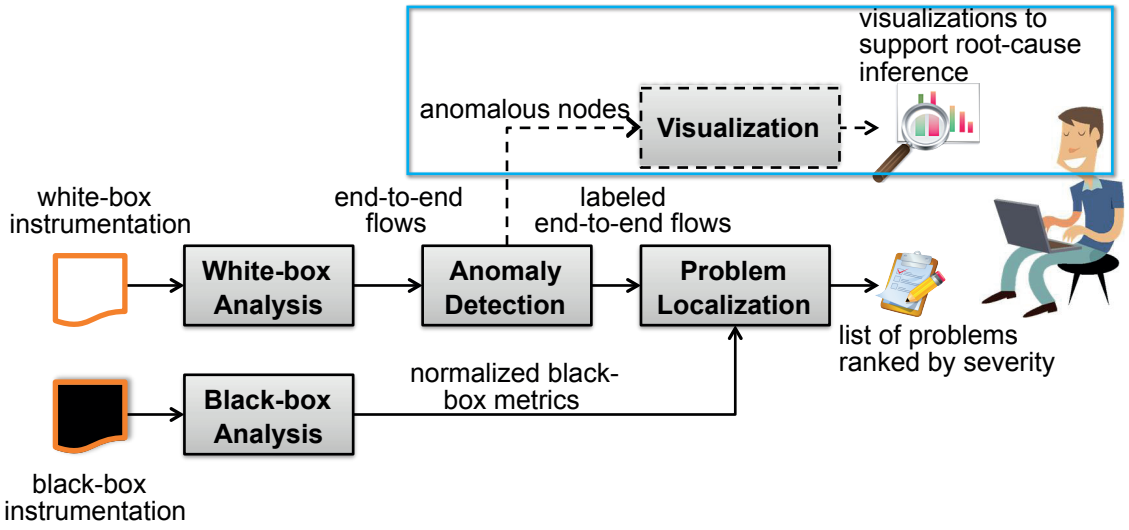


Figure 9.1. **Overview of problem visualization.** Problem visualization highlights anomalous nodes in the cluster, and supports root-cause inference.

Design overview. In this chapter, we explore how to use visualizations to distinguish between different classes of problems. We developed a visualization tool that analyzes application-level logs in a Hadoop cluster, and generates visual signatures of each job's performance as shown in Figure 9.1. These visual signatures provide compact representations of task durations, task status, and data consumption by jobs. Our tool leverages application-specific semantics about the structure of the Hadoop programming model to generate high-density, interactive visualizations of job performance that scale to support current industry deployments. Our study of users at a production Hadoop cluster [Campbell et al., 2011] highlighted users' need to differentiate application-level problems (*e.g.*, software bugs, workload imbalances) from infrastructural problems (*e.g.*, contention problems, hardware problems). We have developed *visual signatures* that allow users to easily spot performance problems due to application-level and infrastructural issues. We use peer-comparison to detect anomalous behavior in a production Hadoop cluster. As described in Chapter 5, we define peers as nodes executing the same Hadoop job. We chose to visualize the performance of Hadoop jobs by summarizing behavior at the node-level instead of visualizing individual tasks because the node-level representation was more compact—the number of tasks in a Hadoop jobs can be several orders of magnitude larger than the number of nodes in the cluster. For example, we have observed jobs with over 120,000 tasks running on a 64-node cluster. Our current visualizations can apply to other

Table 9.1. **Heuristics for developing visual signatures of problems in Hadoop.** Heuristics help distinguish between application-level and infrastructural problems in a Hadoop cluster.

Dimension	**Visual Signatures**		
	Application problem	**Workload imbalance**	**Infrastructural problem**
Time	*Single* user or job over time	*Single* user or job over time	*Multiple* users and jobs over time
Space	Span *multiple* nodes	Span *multiple* nodes	Typically affect *single* node, but correlated failures also occur
Value	Performance degradations and *task exceptions*	Performance degradation and *data-skews*	Performance degradations and *task exceptions*

parallel-computing frameworks such as parallel file systems [Kasick et al., 2010]. However, more research is needed to develop visualizations for heterogeneous distributed systems such as VoIP.

Our visualization tool supports three different types of visualizations: one at the cluster-level that represents the performance of jobs across nodes over time, and two others at the job-level that summarize task performance across nodes in terms of task duration, task status and volume of data processed. We evaluated these visualizations using real problems experienced by Hadoop users at the OpenCloud cluster for data-intensive research [Parallel Data Lab, 2012] cluster. Our visualizations correctly identified 192 out of 204 problems that we observed during a one-month period.

The rest of the chapter is organized as follows. Section 9.1 describes the design and implementation of our visualization tool. Section 9.2 describes our visualizations using case studies drawn from real problems experienced by Hadoop users in the cluster. Finally, Section 9.3 analyzes the effectiveness of our visualization at classifying problems in the Hadoop cluster, while Section 9.4 summarizes our results.

9.1 Visual Signatures for Hadoop

We developed a visualization tool that characterizes the (mis-)behavior of large Hadoop clusters as a series of visual signatures, and facilitates troubleshooting of performance prob-

lems on the cluster. We targeted performance problems due to hardware failures or data-skews, and failed jobs due to software bugs. Our current implementation does not address performance problems due to misconfigurations. The key requirements were: 1) an interactive interface that supports data exploration thereby enabling users to formulate a hypothesis on the root-cause of problems; and 2) compact representations that can support Hadoop clusters consisting of up to thousands of nodes.

We extracted data from the job-history logs generated by Hadoop's JobTracker using the domain-specific parsers presented in Chapter 4. We stored this information in a relational database, and generated visualizations in the web browser using the Data-Driven Documents (D3) framework [Bostock et al., 2011]. D3 provides a JavaScript library for generating interactive visualizations by binding data to a Document Object Model (DOM), and then applying data-driven transformations to the document.

We developed visual signatures that allow users to spot of patterns (or signatures) of misbehavior in job execution by identifying visual patterns across the time, space, and value domain. Table 9.1 summarizes the heuristics that we used to develop visual signatures that distinguish between application-level problems, workload imbalances between tasks from the same job, and infrastructural problems. We developed the visualizations iteratively by manually identifying jobs which we knew had failed, and consulting with the system administrators to learn what incidents had occurred in the cluster. Next, we developed the visualizations using a subset of these incidents, and iterated through different designs to select the visualizations that best displayed the problem. We then manually verified that the visualizations matched up with the heuristics for distinguishing between different problems.

These heuristics are explained below:

1. **Time dimension.** Different problems manifest in different ways over time. For example, application-level problems and workload imbalances are specific to an application; therefore, the manifestation of a problem is restricted to a single user or job over time. On the other hand, infrastructural problems, such as hardware failures, affect multiple users and jobs running on the affected nodes over time.
2. **Space dimension.** The space dimension captures the manifestation of the problem across multiple nodes. Application-level problems and workload imbalances associated with a single job manifest across multiple nodes running the buggy or mis-

configured code. Infrastructural problems are typically limited to a single node in the cluster. A study of a globally distributed storage system [Ford et al., 2010] shows that correlated failures are not rare, and were responsible for approximately 37% of failures. Therefore, infrastructural problems can also span multiple nodes.

3. **Value dimension.** We quantify anomalies in the value domain by capturing the extent of performance degradation, data-skew, and task exceptions experienced by a single job. Application-level and infrastructural problems manifest as either performance degradations or task exceptions. Workload imbalances in Hadoop clusters can stem from skewed data distributions that lead to performance degradations.

9.1.1 Detecting Anomalies using Peer-comparison

To generate the visual signatures of problems, we quantify the anomalies experienced during job execution using a small number of metrics. We visualize *Map* and *Reduce* tasks separately because these tasks have very different semantics. We detect anomalies using as *peer-comparison* approach by first assuming that under fault-free conditions, the workload in a Hadoop cluster is relatively well-balanced across nodes executing the same job—therefore, these nodes are peers and should exhibit similar behavior [Pan et al., 2009a]. Next, we identify nodes whose task executions differ markedly from their peers and flag them as anomalous. Aggregating task behavior on a per-node basis allows us to build compact signatures of job behavior because the number of nodes in the cluster can be several orders of magnitudes smaller than the maximum number of tasks in a job. We flag anomalous nodes based on the following metrics:

1. **Task duration.** Task duration refers to the span of a task execution for a given job on a single node. Performance degradations are detected by identifying nodes whose task durations significantly exceed those of its peers.
2. **Data volumes.** The volume of data processed is the total number of bytes read or written from the local filesystem, and the Hadoop Distributed Filesystem (HDFS). We flag anomalies when nodes process significantly more data than their peers (indicating a workload imbalance) or significantly less data (possibly indicating performance problem at the node).
3. **Failure ratios.** The failure ratio is computed by aggregating tasks on a per-node basis, and calculating the ratio of failed to successful tasks. In our visualizations,

we distinguish *failed* tasks from *killed* tasks which arise when speculatively-executed tasks are terminated by the task scheduler.

To compute the anomaly score we assume that metrics follow a normal distribution, and use the z-score—a dimensionless quantity that indicates how much each value deviates from the mean in term of standard deviations. The z-score is computed using the following formula: $z = \frac{x-\mu}{\sigma}$, where μ is the mean of the values, and σ is the corresponding standard deviation. For the cluster-level visualization, we estimate the severity of problems by using a single anomaly score that flags nodes as anomalous if the geometric mean of the absolute value of the z-scores is high, *i.e.*, $AnomalyScore = (|z_{TaskDuration}| * |z_{DataVolume}| * |z_{FailureRatio}|)^{(1/3)}$.

9.1.2 Visualizing Anomalies

To generate visualizations that are meaningful in clusters with hundreds or thousands of nodes, we take advantage of the human brain's deduction and perception capabilities [Kalawsky, 2009]. Human perception is determined by two kinds of processes: bottom-up, driven by the visual information in the pattern of light falling on the retina, and top-down, driven by the demands of attention, which in turn are determined by the needs of the tasks [Ware, 2008]. In our case, the top-down task is to find those nodes, tasks, or jobs that are exhibiting anomalous behavior. We have generated three visualizations that represent different aspects of a job's execution at varying levels of granularity.

All of our visualizations are designed with the following guidelines: 1) they display jobs and nodes in an order that preserves contextual information, *e.g.*, sorting nodes by the amount of data they process; 2) they clearly distinguish between attributes that have different semantics, *e.g.*, distinguishing between Map or Reduce tasks, and failed and killed tasks; and 3) they preserve the structure of the information across the different visualizations. We use a square as the unit of representation for the different metrics of the tasks executing on a node. Table 9.2 shows the different metrics represented in our visualizations.

We also made the following design decisions: 1) present as much information as is understandable in a single viewport by using color and size to signal the relevant information—this allows the brain's visual query mechanism to process large chunks of information; 2) postpone the display of non-relevant attributes and take advantage of the interactive nature of web-browsers—all of our visualizations provide access to additional

Table 9.2. **Metrics used for Hadoop visualizations.**

Metric	Type
Duration	Scalar/Anomaly
Information volume	Scalar/Anomaly
Successful tasks	Count
Killed tasks	Count
Failed tasks	Count
data-skew	Percentage
Job name & job id	Text
Date & time	Text

information by using the mouseover gesture, and allowing users to drill-down to a more detailed view of the data by clicking on the relevant interface elements.

9.1.3 Scalability

A Hadoop cluster might be composed of hundreds or thousands of nodes—leading to challenges in problem diagnosis brought about by the scale of the system. Research on human perception has shown that the brain can manage to distinguish features at a very high resolution, *e.g.*, differentiating up to 250 features per linear inch [Tufte, 2001]. The following formula can be used to calculate the data density of a visualization:

$$\text{Data density} = \frac{\text{Number of data entries or features}}{\text{Area of data display}}$$

For scalability, we leveraged *high-resolution data graphics* to display the relevant information of each node in the cluster. Our heatmap visualization is capable of representing between 1,500 and 2,900 features per square inch, depending on the resolution of the display; this contrasts with the approximately 10 features per square inch shown in a typical publication [Tufte, 2001].

9.2 Visualizations and Case Studies

We developed three different visualizations that facilitate problem diagnosis. We describe their design considerations, and use cases in this section. The first visualization is the

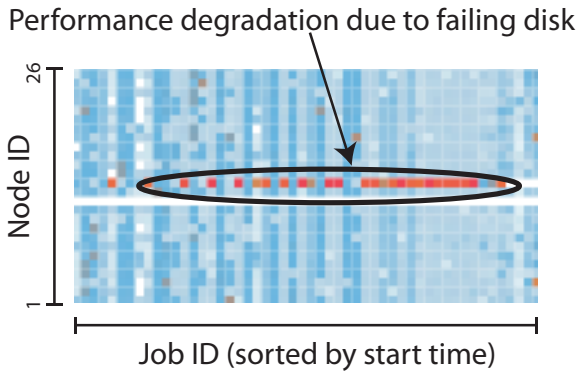


Figure 9.2. **Visual signature of an infrastructural problem using anomaly heatmap.** The anomaly heatmap shows succession of anomalous jobs (darker color) due to a failing disk controller on a node.

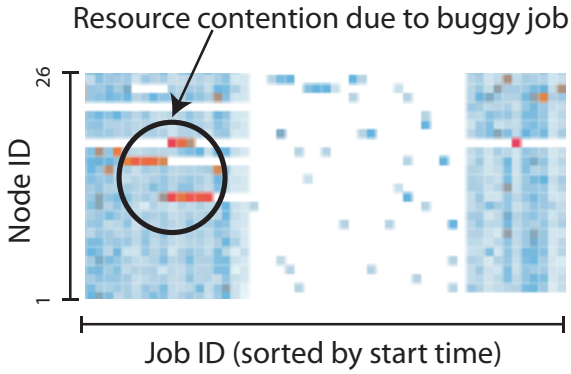


Figure 9.3. **Visual signature of an application-level problem using anomaly heatmap.** In this case, the problem was caused by a single user submitting a resource-intensive job to the cluster repeatedly causing degraded performance on multiple nodes which were eventually blacklisted.

`anomaly heatmap` which summarizes job behavior at the cluster-level; the other two visualizations are at the job-level. The first job-level visualization, referred to as the `job-execution stream`, allows users to scroll through jobs sequentially thus preserving the time context. The second job-level visualization, referred to as the `job-execution detail`, provides a more detailed view of task execution over time on each node in terms of task duration and amount of data processed.

9.2.1 Anomaly Heatmap

A heatmap is a high-density representation of a matrix, that we use to provide users with a high-level overview of jobs execution at the cluster-level. This visualization is formulated

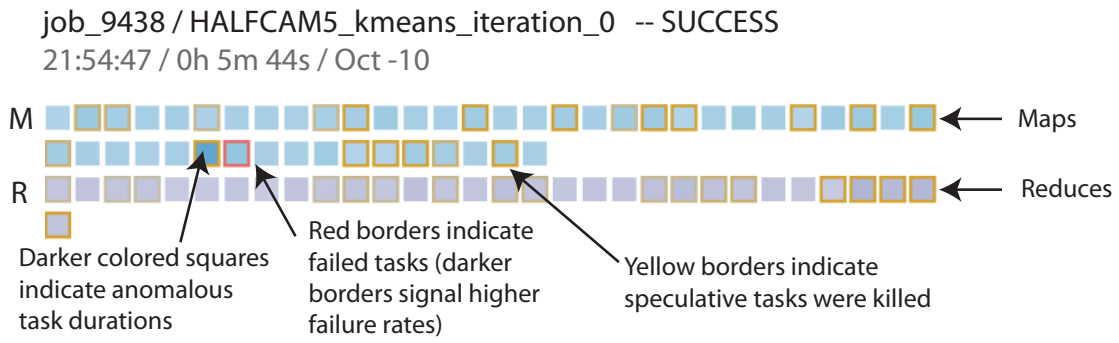


Figure 9.4. **The job-execution stream visualization compactly displays information about a job's execution.** The header lists the job ID, name, status, time, duration and date. The visualization also highlights anomalies in task duration by using darker colors, and task status by using yellow borders for killed tasks and red borders for failed tasks. The nodes are sorted by decreasing amount of I/O processed.

over a grid that shows nodes on the rows and jobs on the columns, as shown in Figure 9.2. The darkness of an intersection on the grid indicates a higher degree of anomaly on that node for that job. By using this visualization, anomalies due to application-level and infrastructural problems can be easily spotted as bursts of color that contrast with non-faulty nodes and jobs in the background.

Figure 9.2 displays the visual signature of an *infrastructural problem* identified by a succession of anomalous jobs (darker color) due to a failing disk controller on a node. Figure 9.3 illustrates the visual signature of an *application-level problem* caused by a single user repeatedly submitting a resource-intensive job to the cluster. The resource contention led to degraded performance across multiple nodes which were eventually blacklisted. It is easy to identify when a node has gone offline or has been blacklisted by spotting a sudden sequence of horizontal white-space; vertical white-space indicates periods of time when the cluster was idle. The data density of the `anomaly heatmap` is around 2,900 features per square inch on a 109 ppi [1] display, using 2x2 pixels per job/node. This gives us a capacity to show approximately $54 \times 54 \approx 2900$ features per square inch [2]; which is equivalent to fit 1200 jobs x 700 nodes on a 27″ display.

[1] ppi: pixels per inch

[2] (109*109ppi)/(2*2pixels) = 2970.25 features per square inch

Figure 9.5. **Visual signature of bugs in the Map phase.** Failures spread across all Map nodes (solid red border) typically signals a bug in the Map phase.

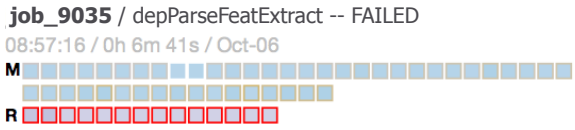


Figure 9.6. **Visual signature of bugs in the Reduce phase.** Failures spread across all Reduce nodes (solid red border) typically signals a bug in the Reduce phase.

9.2.2 Job-execution Stream

The `job-execution stream`, shown in Figure 9.4, provides a more detailed view of jobs while preserving information about the context by showing a scrollable stream of jobs sorted by start time. In addition to displaying general information about the job (job ID, job name, start date and time, job duration) in the header, this visualization has a more complex grammar that allows the representation of an extended set of signatures.

Since the application-semantics of Map and Reduce tasks are very different, we divided nodes into two sets: the Map set and the Reduce set. Each of these sets represent nodes that executed Map or Reduce tasks for a given job sorted by decreasing I/O. The intuition behind the sort order is that nodes that process significantly less data than their peers might be experiencing performance problems. We enhanced the representation of each node with a colored border that varies in intensity depending the ratio of failed tasks to successful tasks, or the ratio of killed tasks to successful tasks; killed tasks arise when the task scheduler terminates speculative tasks that are still running after the fastest copy of the task completed. Killed tasks are represented using a yellow border, which is overloaded by a red border if there are any failed tasks.

The `job-execution stream` visualization allows us to generate signatures for the following scenarios:

1. *Application-level problems.* Bugs in either the Map or Reduce phase manifest as a large number of failed tasks across all nodes in either the Map or Reduce set (see

Figure 9.7. **Visual signature of data-skew.** A node with anomalous task durations (darker color) and high volume of I/O (first nodes in the list) can signal data-skew.

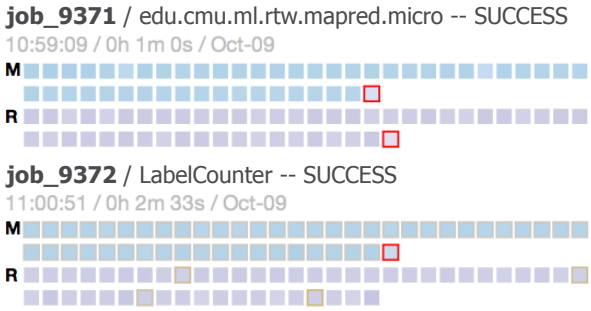


Figure 9.8. **Visual signature of infrastructural problem affecting several jobs.** A node with failures (red border) spread across multiple jobs can signal problems with that node.

Figures 9.5 and 9.6).

2. *Workload imbalance or data-skew.* A *data-skew* can be identified by spotting nodes with anomalous task durations on the first positions of the node list. For example, see the dark left-most node in Figure 9.7.
3. *Infrastructural problems. Infrastructural problems* can be detected if a node recurrently fails across multiple jobs, as shown in Figure 9.8.

9.2.3 Job-execution Detail

The `job-execution detail` visualization provides a more detailed view of task execution and is less compact than the `job-execution stream`. The `job-execution detail` visually highlights both the progress of tasks over time, and the volume of data processed as shown in Figure 9.9. Nodes are still represented as two sets of squares for Map and Reduce tasks; however, given that there is additional space available since we are only visualizing one job at a time, we use the available area on each of the squares to represent two additional variables: 1) anomalies in task durations over time by dividing the area of each node into five vertical stripes, each corresponding to a fifth of the total time spent executing tasks on that node—darker colors indicate the severity of the anomaly while white

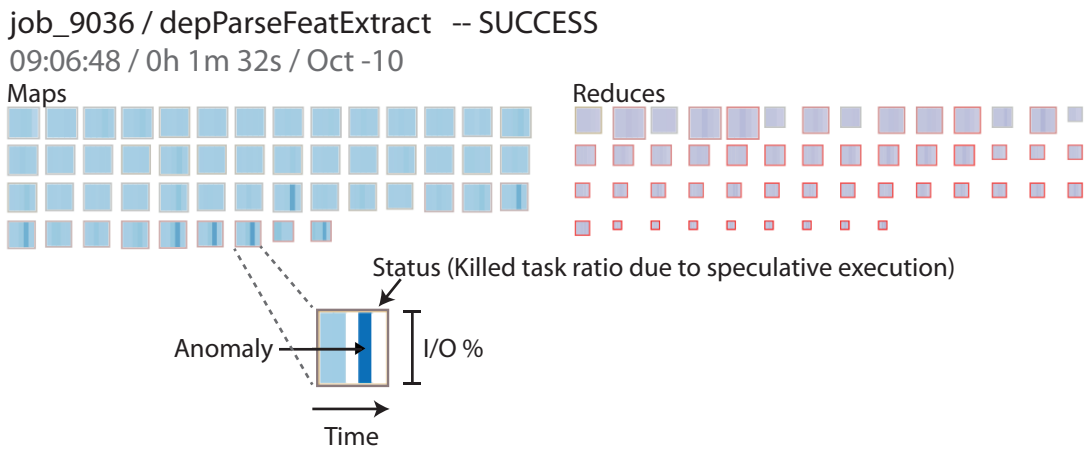


Figure 9.9. **Visualization highlighting both the progress of tasks over time, and the volume of data processed. Job execution detail visualization highlighting both the progress of tasks over time, and the volume of data processed.** Each node is divided on five stripes that represent the degree of anomalies in tasks executing during the corresponding time slot; the size of the square represents the proportion of I/O processed by that node.

stripes represent slots of time where no information was processed; 2) percentage of total I/O processed by that node, *i.e.*, reads and writes to both the local filesystem and the Hadoop distributed filesystem (HDFS)—larger squares indicate higher volumes of data.

Figure 9.10 shows the visual signature of a data skew where a subset of nodes with anomalous task durations (darker color) and high amounts of I/O (first nodes in the list, large square size) indicate data-skew. In this visualization, the data-skew is more obvious to the user when compared to the same problem visualized using the `job-execution stream` in Figure 9.7. Infrastructural problems as shown in Figure 9.11 can be identified as a single node with high task durations (darker color) or failed tasks (red border), coupled with a low volume of I/O (small square size) which might indicate a performance degradation. The data density of this visualization, using 24x24 squares and 7 features per square (5 time slots + I/O percentage + status ratio) on a 109 ppi screen, is about 112 numbers per square inch—this allows us to display up to 4800 nodes on a 27″ display (2400 nodes for Maps + 2400 nodes for Reduces).

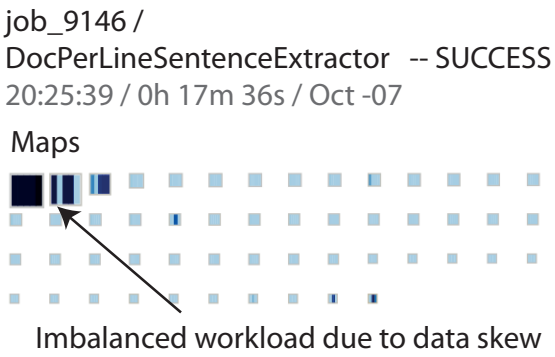


Figure 9.10. **Alternative visual signature of data-skew.** The `job-execution detail` visualization shows a single node with high duration anomaly (darker color) and high amount of I/O (first nodes in the list, large square size) can signal data-skew.

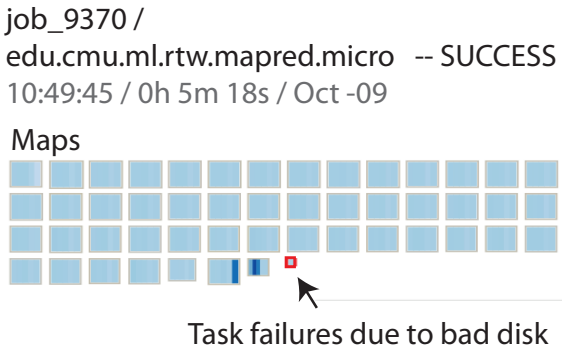


Figure 9.11. **Alternative Visual signature of an infrastructural problem.** The `job-execution detail` visualization shows a single node with high task durations (darker color) or failed tasks (red border), and a low volume of I/O (small square size) can indicate a hardware failure.

9.2.4 Interactive User Interface

Our visualization tool is implemented as an interactive web interface supporting a top-down data exploration strategy that allows users to form, and confirm their hypotheses on the root-cause of the problems. Users can navigate from the cluster-level visualization to the job-level visualizations by clicking on the relevant interface elements. Our tool takes 1–2 seconds to change between views. All of our visualizations provide access to additional information by using the mouseover gesture. Figure 9.12 uses the `job-execution detail` visualization to show how the performance of a job was degraded by a failed NIC (Network Interface Controller) at a node highlighted using a dark blue square. By hovering

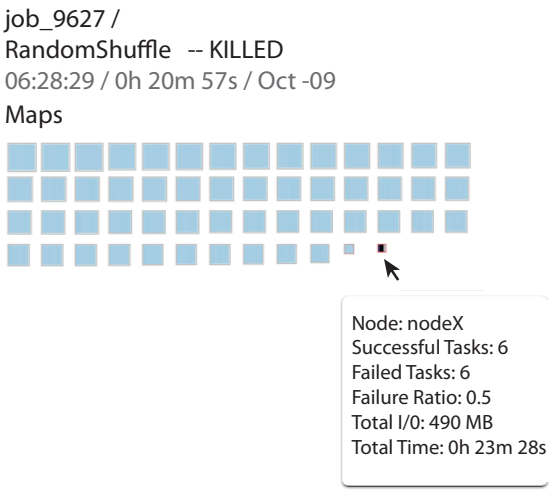


Figure 9.12. **Interactive User Interface.** This job-execution detail visualization shows degraded job performance due to a NIC failure at a node. Hovering over the node provides the user with additional information about the behavior of tasks executed on that node.

Table 9.3. **Problems diagnosed by cluster-level and job-level visualizations.** The infrastructural problems consisted of 42 disk controller failures, 2 full hard drives, and 1 network interface controller (NIC) failure. The infrastructural problems diagnosed by the job-execution stream were a subset of those identified by the anomaly heatmap.

Type	Total problems	Diagnosed by heatmap	Diagnosed by job-execution stream
Application-level problem	157	0	157
Data-skew	2	2	2
Infrastructural problem	45	33	10

over the failed node, a user can obtain additional information about the behavior of tasks executed on that node. For example, a user can observe that 50% of the tasks executed on this node failed.

9.3 Visualization Results

We generated our visualizations using one month's worth of logs generated by Hadoop's JobTracker on the OpenCloud cluster. During this period, 1,373 jobs where submitted, comprising of a total of approximately 1.85 million tasks. From these 1,373 jobs, we manually identified 157 failures due to application-level problems, and 2 incidents of data-skew. We

also identified infrastructural problems by analyzing a report of events generated by the Nagios tool installed on the cluster. During the evaluation period, Nagios reported 68 messages, that were associated with 45 different incidents namely: 42 disk controller failures, 2 full hard drives, and 1 network interface controller (NIC) failure.

We evaluated the performance of our tool by manually verifying that the visualizations generated matched up with the heuristics for distinguishing between different problems described in Table 9.1. Table 9.3 shows that we successfully identified all the application-level problems and data-skews using the `job-execution stream` (similar results are obtained using the `job-execution detail`). In addition, the `anomaly heatmap` was able to identify 33 of the 45 infrastructural problems (the problems identified by the `job-execution stream` are a subset of those identified by the heatmap). We were unable to detect 4 of the infrastructural problems because the nodes had been blacklisted. We hypothesize that the heatmap was unable to detect the remaining 8 infrastructural problems because they occurred when the cluster was idle.

9.4 Summary

This chapter presents our visualization tool that exploits application-specific semantics about the structure of Hadoop jobs to generate high-density, interactive visualizations of job performance that scale to support current industry deployments. Visualization tools complement diagnosis tools by providing interactive interfaces that allow users to explore their data, and formulate their own hypothesis about the root-cause of problems.

Our visualization tool relies on peer-comparison to detect anomalous behavior in a Hadoop cluster. We then applies heuristics to generate visual signatures of problems that allow users to distinguish application-level problems (e.g., software bugs, workload imbalances) from infrastructural problems (e.g., contention problems, hardware problems). Our tool supports three types of visualization namely: a cluster-level visualization that displays the performance of jobs across nodes over time; and two job-level visualizations that summarize task performance across nodes in terms of task duration, task status and volume of data processed. We have evaluated our visualizations using real problems experienced by Hadoop users at a production cluster over a one-month period. Our visualizations correctly identified 192 out of 204 problems that we observed.

Chapter 10

Conclusion

This dissertation introduces *chronics*—performance degradations or failures that are perceivable by end-users, and that affect small subsets of end-users or requests. We developed a *top-down* statistical approach for detecting user-visible problems and localizing the root-cause of chronics in production systems using unmodified white- and black-box logs. Our approach relies on a suite of tools to diagnose problems namely: 1) an extensible log-analysis framework that uses a state-machine abstraction to infer end-to-end flows from the unmodified white-box logs; 2) an anomaly-detection approach that uses heuristics and peer-comparison to label each end-to-end flow as successful or failed; 3) a problem-localization approach that identifies the root-cause of the problem by identifying the groups of attributes that best discriminate between the success and failure labels; and 4) a visualization tool that exploits peer-comparison to generate visual signatures of problems in parallel-computing frameworks. We demonstrated the effectiveness of our approach on real-incidents in two production systems namely: an academic cloud-computing cluster that runs Hadoop, and a VoIP system at a major ISP.

Using both empirical and anecdotal evidence of problems experienced in production systems, we demonstrated that chronic problems are more prevalent than major outages. We also demonstrated that chronics can be notoriously difficult to diagnose: 1) their small size makes setting alarm thresholds tricky; 2) there are many of them active concurrently even when the system as a whole is mostly functional; 3) their symptoms often overlap with each other; 4) they are triggered by complex corner cases involving multiple conditions; and 5) they persist for lengthy periods and can get absorbed into the system's definition of what is normal.

The diagnostic framework presented in this thesis addresses the challenges faced when detecting, localizing and visualizing chronics in production systems as follows:

Inferring knowledge of system dependencies from unmodified white-box logs. Our white-box log-analysis framework infers end-to-end flows from unmodified white-box logs in production systems by extracting data-flow and control-flow information from the logs, and by deriving state-machine like views of the system's execution. The knowledge of system dependencies captured by the end-to-end flows facilitates the diagnosis of chronic performance problems. For example, knowledge of system dependencies enables our problem-localization approach to narrow down the source of a problem when errors propagate across components.

Anomaly-detection. Whenever available, our approach relies on heuristics to detect problems in end-to-end flows. In the absence of labeled failure-data, we rely on peer-comparison to detect user-visible problems by flagging *odd-man-out* behavior in the end-to-end flows. Our peer-comparison approach combines domain-specific knowledge about the structure of Hadoop jobs to identify peers, with stepwise-regression to automatically factor out variance due to application-level differences, such as differences in data input sizes. Any significant variance not explained by the regression models is labeled as anomalous. Our anomaly-detection approach exploits a top-down strategy that focuses on the end-to-end flows, rather than relying on bottom-up alarms based on server logs—as the bottom-up approach might fail to detect chronics that do not significantly perturb server behavior.

Problem localization. We rely on a top-down problem-localization approach to drill-down on the root-cause of chronics. Our problem-localization approach uses the labeled end-to-end flows generated during anomaly-detection to localize the root-causes of problems by comparing bad interactions with good ones *from the same interval of time* rather than relying on historical data from good intervals of time, which might fail to detects chronics that have persisted for long periods of time. We use a greedy, iterative search to determine whether a problem is due to a combination of factors, or due to multiple ongoing problems. We also incorporate black-box metrics, such as CPU-usage, to gain insight on whether the problem was due to resource contention.

Visualization. Our visualization approach uses peer-comparison, coupled with heuristics, to generate visual signatures of classes of chronic problems in parallel-computing

frameworks such as Hadoop. At present, our visualizations distinguish between three broad classes of problems namely: application-level problems such as software bugs, infrastructural problems such as bad disks, and data-skews due to imbalanced workloads.

Our problem-localization tool has been deployed on a major VoIP platform serving millions of users and handling tens of millions of calls a day, and is being successfully used by its operations team. We provided examples of real chronics that our tool helped identify, and demonstrated through fault-injection that our tool can provide coverage levels as high as 97% with false positives as low as 4%. Our tool provides near-interactive performance of < 1 second per chronic, all while running on a single server machine with middle of the range hardware.

10.1 Open Questions and Future Work

The work presented in this dissertation raises a number of questions about the scientific foundations, and the engineering choices that are required to diagnose chronics in production systems. These open questions revolve around: 1) automatic identification of peers in heterogeneous systems; 2) support for online monitoring and diagnosis in large production systems.

Automatic Identification of Peers in Heterogeneous Systems. Our peer-comparison based approaches for anomaly-detection and visualization rely on expert knowledge to identify peers. For example, in Hadoop, peers can be defined as tasks belonging to the same job or nodes executing the same set of jobs. Through our use of step-wise regression during anomaly-detection, we automatically factored our variance due to application-level characteristics, such as variations in input and output sizes in parallel-computing frameworks such as Hadoop. [Kang et al., 2010; Ananthanarayanan et al., 2010] also relied on expert knowledge and regression to identify peers in parallel-computing clusters and load-balanced web servers. The notion of what constitutes a *peer-group* is less clear-cut in highly-heterogeneous systems, such as VoIP, which comprise of many different types of components with different performance characteristics. Researchers have automatically identified peers in heterogeneous systems by clustering requests [Barham et al., 2004; Sambasivan et al., 2011], or by using nearest-neighbor searches to find similar configurations [Thereska et al., 2010]. However, these clustering techniques are better suited for handling static at-

tributes (*e.g.*, CPU type) rather than dynamic attributes (*e.g.*, CPU usage) which vary over time. Instance-based learning approaches [Smith, 2007; Kapadia et al., 1999; Kavulya et al., 2010] combine clustering to find a set of similar jobs, with regression models to automatically identify the attributes most correlated with job performance. These instance-based approaches can cope with both static and dynamic attributes. The open question is whether we can extend these instance-based approaches to cope with variance in both the white- and black-box metrics in highly-heterogeneous production environments. One approach could be to create *virtual peer-groups* by strategically factoring out sources of variance in heterogeneous systems. These virtual peer-groups could then be used to explore performance characteristics, and flag outliers during anomaly-detection and problem visualization. For example, a virtual peer-group could be created by normalizing for load differences across servers in heterogeneous environments—thereby making peer-comparison possible despite the existence of heterogeneity. The notion of a virtual peer-group is also helpful when membership in the group varies over time due to configuration changes. For example, in virtualized environments, peer-groups may vary due to the migration of virtual machines. Access to configuration-related information using tools such as vQuery [Ilari Shafer and Snorri Gylfason and Gregory R. Ganger, 2012] would facilitate dynamic updates to the definition of peer-groups whenever the configuration changes.

Online Monitoring and Diagnosis Our problem localization tool supports online diagnosis, however our white-box and black-box log-analysis tools for inferring end-to-end flows were implemented offline. The open questions when developing online monitoring and diagnosis frameworks are: 1) how to develop a scalable, streaming implementation of the monitoring framework that captures both end-to-end dependencies and programmers' expections; 2) how to sample end-to-end flows and still detect rare events such as chronics; and 3) how to diagnose problems across multiple administrative domains in real-time. There has been considerable research in developing robust monitoring frameworks that automatically generate end-to-end flows by examining messages exchanged by components [Barham et al., 2004; Fonseca et al., 2007; Sigelman et al., 2010]. However, these frameworks do not completely capture the expectations of programmers. [Reynolds et al., 2006] allows programmers to manually embed expectations about application behavior in the source code—these expectations are extracted by Pip when generating end-to-end flows in a distributed system. X-ray [Attariyan et al., 2012] automatically instruments binaries of

stand-alone applications, and uses dynamic information flow tracking to narrow down the root-cause of the problem. Through the instrumentation of binaries, X-ray captures a programmer's expectations, but it does not support distributed environments. More research is needed to develop frameworks that automatically infer system dependencies across a distributed system, and capture the expectation of programmers.

Scalability and the incorporation of new data sources, such as those proffered by virtualized environments, will continue to be a challenge for monitoring. Large-scale distributed systems exert pressure on online diagnosis systems to analyze massive amounts of data, and produce a diagnosis outcome within minutes. Online monitoring and diagnosis frameworks exploit sampling to cope with the large volume of data. Chopstix [Bhatia et al., 2008] uses sketches—a probabilistic data structure that allows the approximate tracking of a large number of events at the same cost as deterministically tracking significantly fewer events. Sketches are an alternative to uniform sampling, which has the disadvantage of drawing most of its samples from events with large populations. Research is needed to determine the best sampling strategies for detecting chronic performance problems. For example, whether sketches are effective for detecting chronics, or whether alternative sampling strategies are needed.

In many domains such as Internet services and telecommunications, large systems are increasingly built as a composition of multiple horizontal *technology layers* and vertical *administrative domains*. For example, consider a typical Internet application constructed using the Java runtime and its libraries, hosted in a Tomcat application server running on a Linux OS inside a virtual machine at a particular data center of a cloud provider. In addition, this application uses the Bing mapping service from Microsoft, obtains analytics support from Google Analytics, and uses PayPal as a payment service. Each of these services also run on very similar infrastructure layers, and depending on which cloud provider the application users, some of these services may also share a data-center and/or a network provider with the application.

In such a highly layered and highly silo'ed setup, faults can occur in each of the technology layers, or at the third-party providers that the service uses. Seemingly independent third party providers may have common dependencies (*e.g.*, using the same cloud provider) resulting in correlated failures. No single layer or administrative domain may have sufficient information to completely determine the root cause of a fault occurring in

the system. These complications make diagnosis a challenging task. Our diagnostic framework analyzes data across two technology layers in a single administrative domain namely: the application-level logs, and the OS performance counters. [Kompella et al., 2005; Oliner et al., 2010; Mahimkar et al., 2009] have done preliminary work on combining information across technology layers within a single administrative domain. However, more research is needed to develop comprehensive online-diagnosis algorithms that can capture a wholistic view of the system across multiple domains and technology layers.

Bibliography

M. K. Agarwal, M. Gupta, V. Mann, N. Sachindran, N. Anerousis, and L. B. Mummert, "Problem determination in enterprise middleware systems using change point correlation of time series data," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Vancouver, Canada, April 2006, pp. 471–482. 2, 4, 21

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 (SOSP)*, Bolton Landing, NY, Oct 2003, pp. 74–89. 22, 50, 53

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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, Canada, October 2010, pp. 1–16. 135

Apache Software Foundation, "Hadoop," 2007, http://hadoop.apache.org/core. 7, 28, 34, 50

——, "Powered By Hadoop," 2012, http://wiki.apache.org/hadoop/PoweredBy. 29

——, "Apache's JIRA issue tracker," 2006, https://issues.apache.org/jira. 95

M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012. 26, 53, 136

P. Bahl, R. Chandra, A. G. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Kyoto, Japan, August 2007, pp. 13–24. 17, 19, 20

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 (OSDI)*, San Francisco, CA, December 2004, pp. 259–272. 11, 24, 26, 52, 70, 135, 136

R. Bendel, S. Higgins, J. Teberg, and D. Pyke, "Comparison of skewness coefficient, coefficient of variation, and gini coefficient as inequality measures within populations," *Oecologia*, vol. 78, no. 3, pp. 394–400, March 1989. 40

Best Practical Solutions, "RT: Request Tracker - Issue tracking system," 2013, http://bestpractical.com/rt/. 35, 45

H. Beyer and K. Holtzblatt, *Contextual Design: Defining Customer-Centered Systems*, 1st ed. San Francisco, CA: Morgan Kaufmann, September 1997. 36

S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. L. Peterson, "Lightweight, high-resolution monitoring for troubleshooting production systems," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, December 2008, pp. 103–116. 16, 137

P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises," in *European conference on Computer systems (EuroSys)*, Paris, France, April 2010, pp. 111–124. 2, 5, 25, 99, 108

M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-driven documents," *IEEE Transactions on Visualization & Computer Graphics (Proc. InfoVis)*, 2011. [Online]. Available: http://vis.stanford.edu/papers/d3 121

J. D. Campbell, A. B. Ganesan, B. Gotow, S. P. Kavulya, J. Mulholland, P. Narasimhan, S. Ramasubramanian, M. Shuster, and J. Tan, "Understanding and improving the diagnostic workflow of MapReduce users," in *ACM Symposium on Computer Human Interaction for Management of Information Technology CHIMIT*, Boston, MA, December 2011. 35, 119

M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *IEEE International Conference on Automatic Computing (ICAC)*, New York, NY, May 2004, pp. 36–43. 25

M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *IEEE Conference on Dependable Systems and Networks (DSN)*, Bethesda, MD, Jun 2002. 25, 50

L. Cherkasova, K. M. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *IEEE Conference on Dependable Systems and Networks (DSN)*, Anchorage, Alaska, June 2008, pp. 452–461. 18, 20, 71

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 (SOSP)*, Brighton, United Kingdom, October 2005, pp. 105–118. 2, 5, 25, 50, 99, 108

I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating instrumentation data to system states: A building block for automated diagnosis and control," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004, pp. 231–244. 25

G. F. Cretu-Ciocarlie, M. Budiu, and M. Goldszmidt, "Hunting for problems with artemis," in *USENIX Workshop on Analysis of System Logs*, San Diego, CA, December 2008. 27, 118

D. Crockford, "JavaScript Object Notation (JSON)," 2006, http://www.json.org/. 62

J. Dai, J. Huang, S. Huang, B. Huang, and Y. Liu, "Hitune: dataflow-based performance analysis for big data cloud," in *USENIX Annual Technical Conference (ATC)*, Portland, OR, Jun. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002181.2002188 118

J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters." *Communications of the ACM*, vol. 51, pp. 107–113, 2008. 29

P. Desnoyers, T. Wood, P. J. Shenoy, R. Singh, S. Patil, and H. M. Vin, "Modellus: Automated modeling of complex internet data center applications," *ACM Transactions on the Web TWEB*, vol. 6, no. 2, p. 8, May 2012. 18, 20

S. Duan and S. Babu, "Guided problem diagnosis through active learning," in *IEEE International Conference on Automatic Computing (ICAC)*, Chicago, IL, June 2008, pp. 45–54. 25

EMC, "Automating root cause analysis: Emc ionix codebook correlation technology vs. rule-based analysis," EMC, Tech. Rep. h5964, Nov 2009. 15

ExtraHop.com, "Blackouts, Brownouts, and Application Aware Network Performance Management (AANPM)," January 2011, http://www.extrahop.com/post/blog/extrahop-analysis/application-performance-management-blackouts-brownouts/. 1, 52

FCC, "The Proposed Extension of Part 4 of the Commission's Rules Regarding Outage Reporting To Interconnected Voice Over Internet Protocol Service Providers and Broadband Internet Service Providers," Federal Communications Commission, Tech. Rep. PS Docket No. 11-82, FCC 12-22, February 2012. 30

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 (NSDI)*, Cambridge, MA, Apr 2007. 11, 26, 118, 136

D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, CA, October 2010, pp. 61–74. 122

C. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, no. 4, pp. 17–37, 1982. 16

Ganglia, "Ganglia monitoring system," 2007, http://ganglia.info. 27, 34, 118

E. Garduno, S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "Theia: Visual signatures for problem diagnosis in large hadoop clusters," in *USENIX Large Installation System Administration Conference (LISA)*, San Diego, CA, December 2012. 8

S. Ghemawat, H. Gobioff, and S. Leung, "The Google File System." in *ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct 2003, pp. 29 – 43. 30

M. Hauswirth, A. Diwan, P. Sweeney, and M. Hind, "Vertical profiling: Understanding the behavior of object-oriented applications." in *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, BC, Canada, Oct. 2004, pp. 251 – 269. 16

E. Hoke, J. Sun, and C. Faloutsos, "Intemon: Intelligent system monitoring on large clusters," in *ACM Conference on Very Large Data Bases (VLDB)*, Seoul, Korea, Sep. 2006, pp. 1239–1242. 23

K. Huh, K. Han, D. Hong, J. Kim, H. Kang, and P. Yoon, "A model-based fault diagnosis system for electro-hydraulic brake," SAE International, Warrendale, PA, USA, SAE Technical Paper Series 2008-01-1225, April 2008. 17

IBM, "Tivoli enterprise console," 2010, http://www.ibm.com/software/tivoli/products/enterprise-console. 16

Ilari Shafer and Snorri Gylfason and Gregory R. Ganger, "vQuery: A platform for connecting configuration and performance," VMware Technical Journal, Tech. Rep. VMware Technical Journal, Vol. 1, No. 2, December 2012. 136

M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, Montana, Oct. 2009, pp. 261–276. 41

G. Jiang, H. Chen, K. Yoshihira, and A. Saxena, "Ranking the importance of alerts for problem determination in large computer systems," in *IEEE International Conference on Automatic Computing (ICAC)*, Barcelona, Spain, June 2009a, pp. 3–12. 22

M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward, "System monitoring with metric-correlation models: problems and solutions," in *IEEE International Conference on Automatic Computing (ICAC)*, Barcelona, Spain, June 2009b, pp. 13–22. 22

K. R. Joshi, W. H. Sanders, M. A. Hiltunen, and R. D. Schlichting, "Automatic model-driven recovery in distributed systems," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, Orlando, Florida, October 2005, pp. 25–38. 17, 19

R. S. Kalawsky, "Gaining greater insight through interactive visualization: A human factors perspective," in *Trends in Interactive Visualization*, ser. Advanced Information and Knowledge Processing, R. Liere, T. Adriaansen, and E. Zudilova-Seinstra, Eds. Springer London, 2009. 123

S. Kandula, D. Katabi, and J.-P. Vasseur, "Shrink: A Tool for Failure Diagnosis in IP Networks," in *ACM SIGCOMM Workshop on mining network data (MineNet-05)*, Philadelphia, PA, August 2005. 19

S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," in *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Barcelona, Spain, August 2009, pp. 243–254. 2, 4, 21, 27

H. Kang, H. Chen, and G. Jiang, "Peerwatch: a fault detection and diagnosis tool for virtualized consolidation systems," in *IEEE International Conference on Automatic Computing (ICAC)*, Washington, DC, June 2010. 23, 135

N. H. Kapadia, J. A. Fortes, and C. E. Brodley, "Predictive application-performance modeling in a computational grid environment," in *International Symposium on High-Performance Distributed Computing*, Redondo Beach, CA, Aug. 1999, p. 6. 71, 136

M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan, "Black-box problem diagnosis in parallel file systems," in *USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2010, pp. 43–56. 22, 23, 70, 111, 120

S. Kavulya, R. Gandhi, and P. Narasimhan, "Gumshoe: Diagnosing performance problems in replicated file-systems," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, Naples, Italy, October 2008, pp. 137–146. 70, 71

S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, Melbourne, Australia, May 2010, pp. 94–103. 8, 9, 32, 71, 136

S. Kavulya, K. R. Joshi, M. A. Hiltunen, S. Daniels, R. Gandhi, and P. Narasimhan, "Practical experiences with chronics discovery in large telecommunications systems," *Operating Systems Review*, vol. 45, no. 3, pp. 23–30, 2012a. 3, 8, 9

S. P. Kavulya, K. Joshi, M. Hiltunen, S. Daniels, R. Gandhi, and P. Narasimhan, "Practical experiences with chronics discovery in large telecommunications systems," in *ACM Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML)*, Cascais, Portugal, October 2011. 9

S. P. Kavulya, S. Daniels, K. R. Joshi, M. A. Hiltunen, R. Gandhi, and P. Narasimhan, "Draco: Statistical diagnosis of chronic problems in large distributed systems," in *IEEE Conference on Dependable Systems and Networks (DSN)*, Boston, MA, June 2012c, pp. 1–12. 8, 9

S. P. Kavulya, K. Joshi, F. Di Giandomenico, and P. Narasimhan, "Failure diagnosis of complex systems," in *Resilience Assessment and Evaluation of Computing Systems*. Springer Berlin Heidelberg, 2012b, pp. 239–261. 8

T. Kelly, "Detecting performance anomalies in global applications," in *USENIX Workshop on Real Large Distributed Systems (WORLDS)*, San Francisco, CA, December 2005. 18, 20, 71

G. Khanna, M. Y. Cheng, P. Varadharajan, S. Bagchi, M. P. Correia, and P. Verissimo, "Automated rule-based diagnosis through a distributed monitor system," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 266–279, 2007b. 19, 20

G. Khanna, I. Laguna, F. A. Arshad, and S. Bagchi, "Distributed diagnosis of failures in a three tier e-commerce system," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, Beijing, China, October 2007a, pp. 185–198. 19

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*, vol. 16, no. 5, pp. 1027– 1041, September 2005. 8, 24, 93, 103, 104

E. Kiciman, "Using statistical monitoring to detect failures in internet services," Ph.D. dissertation, Stanford University, Sep. 2005. 1, 28

R. R. Kompella, J. Yates, A. G. Greenberg, and A. C. Snoeren, "Ip fault localization via risk modeling," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005. 17, 19, 138

E. Krevat, J. Tucek, and G. R. Ganger, "Disks are like snowflakes: No two are alike," in *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Ana Pueblo, NM, May 2011, pp. 1–5. 72

S. Kullback and R. A. Leibler, "On information and sufficiency," *The Annals of Mathematical Statistics*, vol. 22, pp. 79–86, March 1951. 10, 80, 82, 96

A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960. 85

P. E. Lanigan, S. Kavulya, T. E. Fuhrman, P. Narasimhan, and M. A. Salman, "Diagnosis in automotive systems: A survey," Carnegie Mellon University PDL, Tech. Rep. CMU-PDL-11-110, May 2011. 14, 17

J. C. Laprie, "Dependable computing: Concepts, limits, challenges," in *IEEE International Symposium on Fault-Tolerant Computing: Special Issue*, 1995, pp. 42–54. 13

C. Liu, Z. Lian, and J. Han, "How Bayesians debug," in *IEEE International Conference on Data Mining (ICDM)*, Hong Kong, China, December 2006, pp. 382–393. 22, 83

G. Liu, A. Mok, and E. Yang, "Composite events for network event correlation," in *International Symposium on Integrated Network Management*, Boston, MA, May 1999, pp. 247 –260. 16

X. Liu, J. Heo, and L. Sha, "Modeling 3-tiered web applications," in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Atlanta, GA, September 2005, pp. 307–310. 18

Z. Liu, B. Lee, S. Kandula, and R. Mahajan, "Netclinic: Interactive visualization to enhance automated fault diagnosis in enterprise networks," in *IEEE Conference on Visual Analytics Science and Technology*, Salt Lake City, UT, October 2010, pp. 131–138. 27

A. A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao, "Towards automated performance diagnosis in a large IPTV network," in *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Barcelona, Spain, August 2009, pp. 231–242. 2, 22, 23, 52, 138

H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. 90

G. McCullough, N. McDowell, and G. Irwin, "Fault diagnostics for internal combustion engines – current and future technologies," SAE International, SAE Technical Paper Series 2007-01-1603, April 2007. 17

P. McLachlan, T. Munzner, E. Koutsofios, and S. C. North, "Liverac: interactive visual exploration of system management time-series data," in *Conference on Human Factors in Computing Systems, CHI*, Florence, Italy, April 2008, pp. 1483–1492. 27, 118

Nagios Enterprises., "Nagios," 2008, http://www.nagios.org. 35

A. Oliner, A. P. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stoica, "Carat: Collaborative energy debugging for mobile devices," in *USENIX Workshop on Hot Topics in Dependable Systems (HotDep)*, Hollywood, CA, Oct. 2012. 22

A. J. Oliner, A. V. Kulkarni, and A. Aiken, "Using correlated surprise to infer shared influence," in *IEEE Conference on Dependable Systems and Networks (DSN)*, Chicago, IL, July 2010, pp. 191–200. 2, 22, 23, 52, 138

H. Packard, "HP operations manager," 2010, http://www.managementsoftware.hp.com. 16

X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Ganesha: Black-Box Diagnosis of MapReduce Systems," in *Workshop on Hot Topics in Measurement and Modeling of Computer Systems (HotMetrics)*, Seattle, WA, Jun. 2009b. 9

——, "Blind Men and the Elephant: Piecing together Hadoop for diagnosis," in *International Symposium on Software Reliability Engineering (ISSRE)*, Mysuru, India, Nov. 2009a. 8, 9, 22, 23, 70, 71, 72, 94, 95, 97, 122

Parallel Data Lab, "OpenCloud cluster," 2012, http://wiki.pdl.cmu.edu/opencloudwiki/Main/WebHome. 30, 33, 39, 120

E. Plugge, T. Hawkins, and P. Membrey, *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*, 1st ed. Berkely, CA, USA: Apress, 2010. 61

K. Ren, J. López, and G. Gibson, "Otus: resource attribution in data-intensive clusters," in *Workshop on MapReduce and its applications (MapReduce)*, San Jose, CA, June 2011. 118

K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence: A comparative workload analysis from three research clusters," Carnegie Mellon University, Tech. Rep. CMU-PDL-12-106, June 2012. 32, 41

P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006. 26, 136

I. Rish, M. Brodie, N. Odintsova, S. Ma, and G. Grabarnik, "Real-time problem determination in distributed systems using active probing," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Seoul, South Korea, April 2004, pp. 133–146. 17, 19, 20

RuleQuest Research Data Mining Tools, "See5/C5.0," 2011, http://www.rulequest.com/. 103

A. Ryan, "Under the Hood: Hadoop Distributed Filesystem reliability with Namenode and Avatarnode," June 2012, http://www.facebook.com/notes/facebook-engineering/under-the-hood-hadoop-distributed-filesystem-reliability-with-namenode-and-avata/10150888759153920. 29

R. R. Sambasivan and G. R. Ganger, "Automated diagnosis without predictability is a recipe for failure," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Boston, MA, Jun. 2012a, pp. 1–6. 4, 11, 71

——, "Automated diagnosis without predictability is a recipe for failure," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Boston, MA, Jun. 2012b. 40

R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, March 2011, pp. 43–56. 2, 8, 25, 70, 93, 103, 105, 135

B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," in *Dependendable Systems and Networks*, Philadelphia, PA, Jun. 2006. 21

B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you?" in *USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2007, pp. 1–16. 21

K. Shen, C. Stewart, C. Li, and X. Li, "Reference-driven performance anomaly identification," in *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Seattle, WA, June 2009, pp. 85–96. 22

B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhagy, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Tech. Rep. dapper-2010-1, April 2010. 11, 26, 27, 52, 118, 136

W. Smith, "Prediction services for distributed computing," in *International Parallel and Distributed Processing Symposium*, Long Beach, CA, March 2007, pp. 1–10. 71, 136

Splunk Inc., "Splunk: The it search company," 2005, http://www.splunk.com. 27

M. Steinder and A. S. Sethi, "A survey of fault localization techniques in computer networks," *Science of Computer Programming*, vol. 53, no. 2, pp. 165–194, July 2004. 14, 16

C. Stewart, T. Kelly, and A. Zhang, "Exploiting nonstationarity for performance prediction," in *European conference on Computer systems (EuroSys)*, Lisbon, Portugal, March 2007, pp. 31–44. 18, 20, 71

G. W. Stewart, "Collinearity and least squares regression," *Statistical Science*, vol. 2, no. 1, pp. 68–84, 1987. [Online]. Available: http://www.jstor.org/stable/2245615 77

J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, San Diego, CA, Jun. 2009. 27, 44, 63

J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "SALSA: Analyzing Logs as StAte Machines," in *USENIX Workshop on Analysis of System Logs*, ser. WASL'08. San Diego, California: USENIX Association, 2008. 8, 9, 22, 23, 54, 56, 67, 71, 72

J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Visual, log-based causal tracing for performance debugging of mapreduce systems," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, Genova, Italy, June 2010b, pp. 795–806. 9, 118

J. Tan, X. Pan, E. Marinelli, S. Kavulya, R. Gandhi, and P. Narasimhan, "Kahuna: Problem diagnosis for mapreduce-based cloud computing environments," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Osaka, Japan, Apr. 2010a, pp. 112–119. 9, 71, 72, 111

S. Tati, B.-J. Ko, G. Cao, A. Swami, and T. F. L. Porta, "Adaptive algorithms for diagnosing large-scale failures in computer networks," in *IEEE Conference on Dependable Systems and Networks (DSN)*, Boston, MA, Jun. 2012, pp. 1–12. 17, 19

E. Thereska, B. Salmon, J. D. Strunk, M. Wachs, M. Abd-El-Malek, J. López, and G. R. Ganger, "Stardust: tracking activity in a distributed storage system," in *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Saint Malo, France, Jun. 2006, pp. 3–14. 52

E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel, "Practical performance models for complex, popular applications," in *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, New York, NY, Jun. 2010, pp. 1–12. 24, 70, 135

E. R. Tufte, *The Visual Display of Quantitative Information*, 2nd ed. Graphics Pr, May 2001. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0961392142 124

B. Urgaonkar, G. Pacifici, P. J. Shenoy, M. Spreitzer, and A. N. Tantawi, "An analytical model for multi-tier internet services and its applications," in *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Banff, Alberta, Canada, June 2005, pp. 291–302. 18

C. Ware, *Visual Thinking: for Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. 123

G. M. Weiss and F. J. Provost, "Learning when training data are costly: The effect of class distribution on tree induction," *Journal of Artificial Intelligence Research (JAIR)*, vol. 19, pp. 315–354, 2003. 104

W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009, pp. 117–132. 23

Yahoo!, "Hadoop capacity scheduler," 2008, https://issues.apache.org/jira/browse/HADOOP-3445. 41

——, "M45 supercomputing project," 2009, http://research.yahoo.com/node/1884. 30, 33, 39

S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie, "High speed and robust event correlation," *Communications Magazine, IEEE*, vol. 34, no. 5, pp. 82–90, May 1996. 2, 15, 23, 47, 52

C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma, "Automated known problem diagnosis with event traces," in *European conference on Computer systems (EuroSys)*, 2006, pp. 375–388. 25

S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox, "Ensembles of models for automated diagnosis of system performance problems," in *IEEE Conference on Dependable Systems and Networks (DSN)*, Yokohoma, Japan, July 2005, pp. 644–653. 25