USING STATISTICAL MONITORING TO DETECT FAILURES IN INTERNET SERVICES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Emre Kıcıman
September 2005

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Armando Fox
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Andrew Y. Ng

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jeffrey Mogul

Approved for the University Committee on Graduate Studies.

Abstract

Since the Internet's popular emergence in the mid-1990's, Internet services such as e-mail and messaging systems, search engines, e-commerce, news and financial sites, have become an important and often mission-critical part of our society. Unfortunately, managing these systems and keeping them running is a significant challenge. Their rapid rate of change as well as their size and complexity mean that the developers and operators of these services usually have only an incomplete idea of how the system works and even what it is supposed to do. This results in poor fault management, as operators have a hard time diagnosing faults and an even harder time detecting them.

This dissertation argues that statistical monitoring—the use of statistical analysis and machine learning techniques to analyze live observations of a system's behavior—can be an important tool in improving the manageability of Internet services. Statistical monitoring has several important features that are well suited to managing Internet services. First, the dynamic analysis of a system's behavior in statistical monitoring means that there is no dependency on specifications or descriptions that might be stale or incorrect. Second, monitoring a live, deployed system gives insight into system behavior that cannot be achieved in QA or testing environments. Third, automatic analysis through statistical monitoring can better cope with larger and more complex systems, aiding human operators as well as automating parts of the system management process.

The first half of this thesis focuses on a methodology to detect failures in Internet services, including high-level application failures, by monitoring structural behaviors that reflect the high-level functionality of the service. We implemented prototype fault monitors for a testbed Internet service and a clustered hashtable system. We

also present encouraging early results from applying these techniques to two real, large Internet services.

In the second half of this thesis, we apply statistical monitoring techniques to two other problems related to fault detection: automatically inferring undocumented system structure and invariants and localizing the potential cause of a failure given its symptoms. We apply the former to the Windows Registry, a large, poorly documented and error-prone configuration database used by the Windows operating system and Windows-based applications. We describe and evaluate the latter in the context of our testbed Internet service.

Our experiences provide strong support for statistical monitoring, and suggest that it may prove to be an important tool in improving the manageability and reliability of Internet services.

Acknowledgments

First and foremost, I would like to thank my Ph.D. advisor, Armando Fox. Throughout my career as a graduate student, our lively discussions, combined with his careful advice, have been invaluable in providing guidance and support. Armando has been an exemplary model, showing how to choose important research topics grounded in real-world problems and how to attack them; I clearly would not be where I am without the time and effort he devoted to his mentorship. I look forward to continuing our discussions in the future.

I would also like to thank my committee members. Jeff Mogul's careful reading and thoughtful comments greatly helped improve the overall quality of my thesis. My discussions with Andrew Ng on the topic of machine learning provided good grounding as I ventured outside my systems background. Thank you as well to my orals committee members, Robert Tibshirani and Mendel Rosenblum: their questions and feedback during the oral exam provided a helpful perspective as I wrote my dissertation.

As part of the Recovery-Oriented Computing (ROC) project, I have also had the good fortune of working closely with David Patterson. His perspective and opinions on research topics and methodologies, as well as his career guidance, was much appreciated. Earlier in my career at Stanford, I benefited from collaborating and interacting with both Mary Baker and Terry Winograd.

I appreciate all the help of the people at Amazon.com, Jon Ingalls, Sanjeev Kumar, Ajit Banerjee, George Borle, Robert Noble, and Shivaraj Tenginakai. Their help in understanding the issues of managing Internet services and providing logs was instrumental to validating this research. I would also like to thank the people I interacted with at the anonymous large Internet service #2.

I am very grateful to all my friends and colleagues at Stanford, George Candea, Jamie Cutler, Andy Huang, Ben Ling, Petros Maniatis, Laurence Melloul, Shankar Ponnekanti and Mema Roussopoulos,with whom I spent many hours talking about a wide variety of academic and non-academic topics. I also had many informative discussions with colleagues at U.C. Berkeley, including Aaron Brown, Mike Chen, David Oppenheimer, and Lakshminaryanan Subramanian. I would like to thank all my collaborators, including Yi-min Wang, Eugene Fratkin, Shinichi Kawamoto, Steve Zhang, Anthony Accardi, Brad Johanson, Pedram Keyani, Caesar Sengupta, and Edward Swierk. In particular, my discussions and collaborations with George Candea and Mike Chen greatly influenced the substance of my dissertation work.

I would like to thank the staff at the Stanford University Computer Science department, including Sarah Weden, Peche Turner, Jam Kiattinant, and all the other administrators who keep things in the Gates building running smoothly.

Before beginning my graduate studies at Stanford University, I worked with many wonderful researchers as an undergraduate at U.C. Berkeley. The advice and guidance of Anthony Joseph, Randy Katz and Eric Brewer, as well as interactions with Yatin Chawathe, Steve Czerwinski, Steve Gribble, Todd Hodes, Barbara Hohlt, Matt Welsh, Ben Zhao, and many others, formed an enlightening introduction to research in computer science, and helped convince me to pursue my doctorate.

I would like to especially thank my parents, Nilgün and Ömer, and my brother, Erdem. I am grateful to my parents for their unconditional love, support, and advice. I am grateful to my brother for his cheerfulness, his can-do attitude and his thoughtfulness. Obviously, without my family's influence, I would not be the person I am today.

Most of all, I want to express my gratitude to Johanna, my wife, my confidante and my best friend. She gives me strength and helps me in countless ways. Above all, I am thankful for how she fills my life with happiness and joy.

Through my career as a Ph.D. student, I have received generous support from a National Science Foundation fellowship, a Stanford Graduate Fellowship, a Stanford Networking Research Center (SNRC) Doctoral FMA Fellowship funded by Sony, and an IBM Ph.D. Fellowship. Part of my thesis work was performed during an internship at Microsoft Research.

Contents

List of Tables

List of Figures

Chapter 1

Overview and motivation

Since the Internet's popular emergence in the mid-1990's, Internet services such as e-mail and messaging systems, search engines, e-commerce, news, and financial sites have become an important and often mission-critical part of our daily lives. Unfortunately, managing Internet services and keeping them running is a significant challenge: the scale and complexity of these systems, as well as their rapid evolution and their open-ended workload, mean that the people running these systems have at best an incomplete view (and at worst a wrong view) of what their systems are doing, what their systems are supposed to be doing, and how they are doing it. The result is systems that are hard to manage and unreliable.

1.1 Problem: failures in Internet services

While many Internet services have successfully scaled their systems to serve hundreds of millions of users, these systems still suffer from poor reliability. As of this writing, the last 12 months of news headlines have reported significant Internet service failures at Walmart.com, Blackberry, Check Free, Google, Hotmail, MSN Messenger, Amazon.com, major political web sites, Paypal, Vonage, and Microsoft Money. The causes of these failures included misconfigurations, problems during software upgrades, difficulties recovering from power outages and hardware failures, and many causes that remain unreported. All but one of these reported failures lasted for more than an

hour, while half lasted for more than a day [27, 28, 37, 51, 52, 64, 67–69, 90, 91, 115]. [1]

However, most problems at Internet services do not make it to the news headlines. Keynote, an Internet service monitor, states that the top 40 sites on the Internet have a typical availability of 97%, or about 10 days of downtime each per year [81].[2] Compare this to the phone network's gold standard of 99.999%, or about 5 minutes of downtime per year [88].

There are three fundamental challenges to managing or operating an Internet service:[3]

1. **Large scale and complex systems** exhibit poorly understood behaviors [55]. While good design goes a long way towards improving system predictability, even the best-engineered systems will exhibit unanticipated behaviors at a large scale. Not only does it become harder for any one person to understand the system at the level of detail necessary to predict its behavior, but the scale of the system increases the likelihood of unexpected dependencies, poorly understood interactions, and emergent behavior.

2. Because of market pressures and the (perhaps deceptive) ease of deployment, Internet services are **rapidly changing systems**. As a recent example, on July 25, 2005, Microsoft Network unveiled a location-based search tool that combined aerial imagery with overlaid street maps and yellow page data. Within hours, Google responded with similar features on their location-based search service [99]. While this is pace is extreme, the feature sets, software, hardware, and even architectures generally do evolve at such a rapid pace as to preclude the use of traditional software quality assurance techniques that rely on extended

[1] More details of these failures are discussed later, in Section 2.3

[2] Keynote Systems is in the business of externally monitoring Internet service performance and declares itself the "Internet Performance Authority." The top 40 sites referred to here are the "Keynote Business 40," measuring "the download performance of the 40 most highly traveled, well-connected sites in the United States from 50 cities around the world."

[3] Throughout this thesis we use the terms *management* and *operations* synonymously to refer to the process of maintaining a running system after its initial deployment. We use the term *development* to refer to the system design and software programming that occurs before a service or feature's initial deployment. Of course, the management and development are also intertwined, where lessons being learned from an existing deployment are helping drive future development plans. See Chapter 2 for more background.

design cycles and pre-deployment testing. Moreover, this fast rate of change also exacerbates the difficulty of understanding a large scale system, and introduces significant maintenance cost as *ad hoc* management tools and procedures have to be kept up-to-date with the current version of the system.

3. The **open-ended workload** presented by the Internet means that testing Internet services with a complete range of workloads it will face once deployed is practically impossible. Once an Internet service is deployed, it is subject not only to previously experienced workloads, which may be replicated or reasonably approximated, but also to new workloads as user behaviors change, new kinds of hacks, denial-of-service attacks, and unanticipated flash crowds, giving peak loads many times normal demand [76]. Moreover, the variety of inputs users, much less malicious hackers, can send to a service exacerbate this unpredictability. Combining this new workload with the large scale of the production system means that reproducing problems in controlled conditions (for quality assurance testing and debugging) is simply infeasible.

These challenges affect many aspects of systems management, from fault management to capacity planning, and software development to testing strategies.

Traditional techniques for detecting and managing failures, such as software reliability engineering [103], rely fundamentally on operators and developers having a deep understanding of the behavior and operation of the system as well as the environment in which it will be deployed. In practice, this places limits on the system structure, behavior, and development process in ways that are unrealistic. Whereas these techniques work well when developers can reason about a system's behavior and explore its operational profiles, that opportunity is not available in a dynamic Internet service. Their size and complexity, their rate of change, and their unpredictable open-ended workload simply preclude the application of the techniques that have worked so well in, for example, NASA's software systems [117].

Realizing that something is wrong with a system is the first step toward fixing it, and quickly detecting failures can be the largest bottleneck in improving the availability of an Internet service.[4] Our conversations with Internet service operators confirm

[4]Other major steps are understanding what the problem is, and repairing it. The full fault

that detecting these failures is a significant problem: TellMe Networks estimates that 75% of the time they spend recovering from application-level failures is spent just detecting them [31]. Other sites we have spoken with agreed that application-level failures can sometimes take days to detect, though they are repaired quickly once found. This situation has a serious effect on the overall reliability of Internet services: a study of three sites found that earlier detection might have mitigated or avoided 65% of user-visible failures [107]. Fast detection of these failures is therefore a key problem in improving Internet service availability. In the rest of this section, we are going to focus on how fault detection is affected by the challenges of complexity, size and rapid rate of change.

While realizing that a system is not working may seem simple, many of the failures that affect Internet services are subtle *application-level failures* that change the user-visible behavior of a service without causing obvious lower-level problems that a service operator would notice. Additionally, many failures do not disable the whole service, but cause "only" *brown-outs*, where part of a site is disabled or only some users are unable to access the site.[5]

Existing techniques for fault detection take one of two approaches to monitor systems. They either check that the system's behavior and outputs conform to a specification describing correct behavior or they look for known signs of failure in the system's behavior. In the context of Internet services, however, the problem is that in neither case do we understand the system well enough to we know what to look for.

Checking that a system is conforming to a specification of its correct behavior is challenging because of the simple fact that, in practice, an accurate specification of the correct behavior of large software systems is not available. The size of such systems and the scope of their functionality make writing an unambiguous specification practically impossible. In addition, most of today's Internet services provide a service directly to end users, meaning that their critical functionality is defined only at the

management cycle is much more complicated, and can include steps for temporary avoidance of fault triggers, permanent repairs, investigation into root-causes, and methods for avoiding repetition of the same problems in the future.

[5]See Section 2.3 for more details on failures at Internet services, including examples of actual application-level failures and reports of recent outages.

human level, not as easily verifiable machine-level semantics. This makes it difficult to write a specification that can be used to automatically check for correct operation.

If, instead of specifying what correct behavior is, we attempt to monitor for the possible incorrect behaviors, we run into similar issues. Because of the size, complexity and interdependencies within an Internet service, anticipating everything (or even most) of what can go wrong is simply not practical. Even if we could specify or capture the correct and/or incorrect behaviors of an Internet system, the rate of change of the Internet service's hardware and software environments would make any specification obsolete almost as soon as it was completed, and maintenance of the fault detector would be a nightmare.

The problem of fault detection is representative of a class of important problems in managing Internet services and other large scale systems. Current best practices for these management tasks require operators to understand the "big picture" of how a system is put together and operates. This leads to the problem that, as in fault detection, we do not have a "big picture" understanding of how these large-scale systems work. Whether because of scale, complexity, or rate of change, the operators and developers of Internet services today can see and reason about only the low-level behaviors of individual components. At best, operators have a learned intuition of how these low-level behaviors might relate to each other and to the big picture operation of the whole system. But relying on the intuition of a small number of people simply does not scale, and as Frederick Brooks argues, throwing more people at the software problems only makes them worse [15].

We argue that systems developers and operators must accept that none of us fully understand large scale systems today; and we must look for ways to manage these systems without depending on *a priori* knowledge of their inner workings. The next section introduces our approach of using statistical monitoring to address management tasks such as fault detection without requiring *a priori* specifications or understanding of Internet services.

1.2 Approach: statistical monitoring

To build and maintain robust Internet services, we must find new techniques to bridge the gap between what we can easily observe and verify and the "big picture" understanding that we need to manage a system. These new techniques must scale to the size, complexity and rapid rate of change of large scale systems.

In this dissertation, we argue that applying statistical analysis and machine learning techniques to analyze live observations of a system's behavior (statistical monitoring) can be a powerful tool in improving the manageability and reliability of Internet services.[6] Statistical monitoring combines:

- The **live observation** of system behavior. By definition, monitoring the behavior of a deployed, on-line system lets us see how it behaves in a realistic environment. Capturing a large volume of behavior allows us to take advantage of the "law of large numbers," and apply statistical techniques with reasonable confidence.

  In the context of fault detection, we observe the structural behavior of a system—how the internal components of the system interact with each other. We monitor this class of behaviors because, as we show, these interactions are likely to reflect high-level application functionality in Internet services, allowing us to treat these observations as a proxy for the semantic functionality provided by an Internet service.

- **Weak assumptions** about a system guide our choice of what behaviors to observe and how to model the relationship between these behaviors and the end-goal of our task. By requiring only weak assumptions about an Internet service, our approach is easier to apply to systems that may be poorly understood. In contrast, techniques that require strong assumptions on a system tend toward fragility and high-maintenance costs as they require detailed and up-to-date application-specific knowledge to ensure that their strong assumptions are valid.

[6] Throughout the rest of this thesis, we will use the term statistical techniques to refer to both standard statistical analysis as well as machine learning techniques grounded in statistical methods.

The primary weak assumption we make in applying statistical monitoring to detect faults is to assume that most of the system is working correctly most of the time. This means that we can use the majority of our observations to help us learn what the likely correct behavior of the system is.

- **Statistical techniques.** We use statistical techniques to extract patterns and correlations in our observations, and make inferences that guide our management of a system. In particular, we look for statistical techniques that are interpretable, such as decision trees, as opposed to those whose workings are more difficult to understand, such as neural networks. Improvements in the performance and applicability of off-the-shelf statistical analysis techniques in a variety of domains has made their application to systems management a compelling option.

  We use statistical techniques to translate our observations of structural behavior into probabilistic models of a system's likely correct behavior. Once we have built these models, we can use them to detect anomalous behaviors that are likely to indicate some failure in the high-level functionality being provided by the service.

The primary advantage of this statistical monitoring approach is that it learns what the Internet service should be doing through dynamic observation, and requires no *a priori* application-specific knowledge, other than the weak assumption that most of the time, most of the system is working correctly. Not only does this make deployment easy, but it minimizes maintenance of the monitor as well—whenever the Internet service undergoes major changes, the statistical fault monitor can simply relearn what the system's correct behavior is. While in the past, these techniques may have been considered too computationally expensive to use for on-line monitoring of a system, advances in both processing power and statistical techniques have made these statistical analysis of large quantities of data more than feasible.

Moreover, these techniques have the potential to avoid a common trap: *automation irony*, the fact that taking humans out of the systems-management loop reduces their understanding of the system's operation, making it harder for them to recognize

problems and step in during extreme circumstances.[7] Statistical monitoring, however, is a technique that can help people improve their understanding of system behavior by highlighting patterns, correlations and otherwise explaining complex behavior.

Applying statistical monitoring to a specific systems management problem requires several steps. Given a particular system and a specific problem, we must:

1. Determine what underlying property or behavior of the application will most likely help us solve our problem. We should note that if this property is known *a priori* or easily observable, then statistical monitoring is not necessary. If, on the other hand, the property is unknown, hidden, or costly to determine precisely, then statistical monitoring may be more appropriate.

2. Determine what easily observable and collectible data might reasonably be assumed to have a relationship to the property of interest. The aim is to find a proxy that can be analyzed in lieu of the property we actually are interested in.

3. Determine what class of algorithms (*e.g.,* classification, correlation, or clustering algorithms) and data models are most appropriate for analyzing the data we have chosen. This choice will be determined jointly by the problem domain—namely, how we want to use the result of our analysis—and the nature of the relationship between our chosen proxy data and the actual property we are interested in.

4. If necessary, modify the system to instrument and gather the data chosen in step 2. At the same time, we must choose a specific instance of the class of algorithms determined in step 3. Several issues affect both our instrumentation of the system as well as the algorithm we will choose to analyze the data, including the quality of data we can collect and the algorithm's tolerance of noise; and the scalability of both the instrumentation framework and the algorithm to the volume of data, number of dimensions in the data. Also, there is an interplay between how we sample the data, the kinds of noise that can affect the data (e.g., dropped packets and time skews), and the algorithm's tolerance of that noise.

[7] According to [111], this phrase was first coined in 1987 by Lisanne Bainbridge as "the ironies of automation" in [7].

Finally, the algorithm itself must have appropriately tunable sensitivities and thresholds, and make acceptable assumptions on the underlying distributions of the data being analyzed.

5. Determine how the result of the analysis will be interpreted. In interpreting results, one must be careful to avoid over-interpretation: for example, anomaly detection algorithms may tell us that the behavior of a system has changed, but not whether or not it has failed, and correlation algorithms may tell us that two events occur together, but not that one event causes another.

Finally, we must decide how to act upon the result of the analysis. While the interpretation of the analysis may be more easily generalizable across a problem domain, how to act upon the result is likely to be a system-specific policy decision. Some of the issues that must be considered include the likelihood of the result being correct, as well as the costs and benefits of acting or not acting if the algorithm turns out to be correct or incorrect.

Of course, the process of mapping statistical techniques to a specific systems management problem is not as straightforward as a bullet-pointed process. Above, we only highlight the relationship between the data being observed and the algorithm being instrumented. However, there is a significant interplay between all the choices we make, and limitations imposed on us in one part of the system or analysis process can often be creatively worked around by modifying another part.

1.3 Contributions and thesis map

This dissertation presents statistical monitoring—the use of statistical analysis and machine learning techniques to analyze live observations of a system's behavior—to help improve the manageability of Internet services. The goal of statistical monitoring is to enable people to monitor and analyze a large-scale system without requiring them to know very much about the system in the first place.

The first half of this thesis applies statistical monitoring to the problem of fault detection and reducing the time to notice a failure in an Internet service. The second

half of this thesis applies statistical monitoring techniques to two other problems related to fault detection: automatically inferring undocumented system structure and invariants and localizing the potential cause of a failure given its symptoms.

This dissertation has three main contributions:

- Identification of two easily monitored structural behaviors, and demonstration in a testbed environment that a broad range of failures are visible as changes in these behaviors. These structural behaviors reflect high-level application functionality, allowing us to use them as a proxy for high-level semantics that would otherwise be too difficult to capture.

- Application of statistical analysis and machine learning techniques to detect changes in these structural behaviors. This enables a class of low-maintenance and application-generic fault monitors that can detect problems without requiring *a priori* knowledge of correct or incorrect system behavior. We have prototyped and evaluated these techniques using a testbed Internet service environment, a clustered hashtable system, and data from real-world Internet service environments.

- Demonstration of applying statistical monitoring techniques to two additional problems in the management of complex systems: inference of undocumented system structure and fault localization. We have prototyped and evaluated our techniques for automatically inferring system structure in the context of the Windows Registry, and prototyped and evaluated our fault localization techniques in the context of our testbed environment.

It should be noted that our it is not a contribution of this thesis to create or explore new statistical techniques. Our focus is on mapping existing and well-known statistical techniques to the systems problems of fault management. We explore how systems may be modified to better accommodate statistical monitoring, what kinds of issues arise in applying statistical monitoring techniques to these systems problems, and the results these applications.

Chapter 2 presents background information relevant to this thesis. First, we present background on Internet services, including common architectures and development practices. We review in detail failures in Internet services, as well as current best practices for monitoring for failures. Finally, we briefly overview relevant statistical analysis and machine learning algorithms that are used in this dissertation.

Chapter 3 discusses related work in applying statistical techniques and non-statistical techniques to both Internet service management and fault management more generally.

The concept of the structural behavior of an Internet service and how we apply statistical monitoring to these behaviors to detect failures is presented in Chapter 4. Chapter 5, describes the implementation of our prototype fault detector, our evaluation testbed and our experimental results.

Chapter 6 applies statistical monitoring to the problem of extracting otherwise hidden structure from observations of a system. We first analyze snapshots of Windows configuration registries to extract complex data type descriptions from the simple types of the existing configuration values. We show how these data type descriptions are useful for building likely correctness constraints to detect configuration errors in a Windows system.

Chapter 7 applies statistical monitoring to the problem of localizing the potential causes of a failure. We evaluate two algorithms, data clustering and decision trees, in the context of our testbed environment. We then develop a general model of the fault localization problem which abstracts fault localization across a surprisingly wide variety of systems, including Internet services, BGP networks and others.

The integration of statistical monitoring into the broader fault management process is discussed in Chapter 8, including results from a prototype system for application-generic autonomous recovery which combines statistical monitoring for fault detection and localization with a generic, reboot-based recovery technique.

Finally, Chapters 9.5 and 10 discuss future work and conclude.

1.4 Bibliographic notes

The majority of the work reported in this dissertation has been previously published [24, 25, 31–33, 82–84, 93]. Most of chapters 4 and 5 appeared in [82]; portions were joint work with Ben Ling and appeared in [93]. Parts of Chapter 6 are drawn from [84]. Parts of the root-cause localization work presented in Chapter 7 were drawn from [82]; other parts are joint work with Mike Chen and appeared in [31–33]. The abstract model for root cause localization discussed in the future work Section 9.5.3 is joint work with Lakshminarayanan Subramanian [83]. The autonomous recovery prototype discussed in Chapter 8 is joint work with George Candea and appeared in [24, 25].

Chapter 2

Background

This chapter presents background information relevant to Internet services, including common architectural patterns, failures, and existing fault detection mechanisms. In the latter part of this chapter, we briefly review a number of statistical analysis and machine learning algorithms, such as decision trees and data clustering, that we will take advantage of in later chapters of the thesis.

2.1 Internet services

At the beginning of the Internet's entrance into popular consciousness, about ten years ago, Internet services were simple, single machine web servers providing mostly static content to their users. Today, the largest Internet services span over a hundred thousand machines, geographically distributed around the world. With the original static content of the web replaced by dynamic content, personalization and growing functionality, Internet services often use hundreds or more machines, working together, to generate a reply to a single request.

Internet services run a wide variety of applications. Some of the more commonly used sites include e-commerce sites, search engines, e-mail and messaging systems, photo and document sharing sites, financial services, and news and information services. While admittedly these applications cover a broad range of requirements and systems, this range of services still have much in common that set them apart from

other large computer systems. In the rest of this section, we describe their major commonalities: their tiered and cluster architectures, their use of middleware platforms, their rapid rate of change, and workload characteristics.

2.1.1 Clusters of commodity hardware

Most medium- to large-scale services are built atop clusters of machines with anywhere from tens of nodes to hundreds of thousands of nodes [14]. The hardware of the cluster is generally made of commodity machines, chosen based on their cost-effectiveness, not purely on performance or reliability considerations. The result is that hardware is often several years behind the state-of-the-art, and fails. The reasoning behind this decision is driven by economics. With thousands of machines, it is inevitable that there will be some failures regardless of the reliability of the machines, and once you're spending money to deal with failures, the issue of how many failures you are willing to tolerate becomes a calculable trade-off.

Clustered architectures have several key advantages for Internet services [14, 48]. First, they take advantage of the parallelizable workloads that almost all Internet services face, to provide improved scalability: adding more hardware nodes can often lead directly to an increase in capacity. Secondly, upgrade management becomes simplified: new versions of software can be installed in a rolling fashion across the nodes of the cluster, without requiring the whole system to be shutdown and upgraded in lock step. Fault management is simplified as well: suspect nodes can be turned off or rebooted with only a $\frac{1}{n}$ reduction in capacity.

As a caveat, we should note that some classes of Internet services, such as financial services and airline systems, are developed as a hybrid between a purely clustered system and a purely mainframe architecture. In these services, it is often the case that an existing mainframe system handles core functionality, while a clustered architecture is wrapped around it to handle the web-oriented functionality.

2.1.2 Tiered architecture

A common design pattern for clustered Internet applications is the tiered architecture: the system is divided into multiple tiers of functionality, with each tier depending on the functionality of a lower tier. Each tier consists of many nodes, plus a load balancer that directs incoming traffic to one or more nodes within the tier. A user's request will usually enter the Internet service and first go to a load balancer to be directed to a node in the top tier of the system. In processing the request, this node may make calls to lower tiers; each time the request passes to a lower tier, a load balancer will direct the request in attempt to provide improved performance and availability.

A typical pattern is to use a three-tier architecture: a presentation tier consists of stateless Web servers that are responsible for the presentation-related processing, such as HTML formatting, of responses to user requests; the application tier runs the application or business logic of the Internet service and is responsible for the core service; and finally, the storage tier is responsible for managing persistent data in one or more databases.

2.1.3 Middleware platforms: J2EE

Most Internet services today are built atop some form of middleware platform that handles generic concerns, such as scalability, load-balancing, security, and allows the higher-level application code to focus on providing the functionality of the service. Many older Internet services use home-grown middleware platforms, while newer services, as well as services from non-technology companies, use more recently standardized middleware platforms, such as Sun Microsystems' J2EE standard, or Microsoft's .NET platform.

Much of the Internet service-related work presented in this dissertation was prototyped and tested in the context of J2EE, and we spend the rest of this section describing some of the more salient details of the J2EE platform.

J2EE is a Java-based standard middleware championed by Sun Microsystems [98]. The goal of J2EE is to simplify the development, deployment and management of multi-tier server-centric applications, such as Internet services.

At the core of the J2EE middleware architecture are Enterprise Java Beans (EJBs) and EJB management containers. The application code is implemented as an EJB; and the middleware provides a management container for the EJB. The EJB is responsible for application-level functionality, while the management container handles the generic concerns of replication, thread management, lifecycle management, security and access control, transaction management and resource pooling, etc. An EJB and its management container are similar to event handlers, in that they do not constitute a separate locus of control—a single Java thread shepherds a user request through multiple EJBs, from the point it enters a J2EE server process until it finishes processing and returns.

There are several flavors of EJBs. While all EJBs run within the application tier, *entity beans* are the interface for interacting with persistent storage, and represent the data and tables kept in the storage tier. *Session beans* are responsible for managing processes or tasks related to a user request. As a rule of thumb, entity beans model nouns, while session beans model verbs [101].

J2EE also supports a number of additional types of components related to the presentation tier. Java Scripting Pages (JSP) is a simple markup-language used to dynamically build HTML pages. *JSP tags* are components that extend the functionality available within a JSP. A JSP interacts with EJBs through JSP tags. In addition, Java Servlets are components similar to CGI scripts.

End users interact with a J2EE application through a Web interface, usually built using JSP scripts and custom JSP tags in the application's presentation tier, hosted within a web server. These components in turn invoke methods on EJBs in the application tier, using a remote method invocation (RMI) protocol. These invoked EJBs can call on other EJBs, interact with backend databases, invoke other web services, etc.

While J2EE is often used to build user-facing Internet service, it can also be used to build web service meant to be used programmatically via a web API, such as SOAP, or a Java API, such as RMI.

There are many independent implementations of the J2EE standard. Commercial implementations include Sun's own Java System Application Server Platform,

IBM's WebSphere, and BEA's WebLogic. Open source implementations include JBoss Group's JBoss server, and ObjectWeb's JOnAS application server.

2.1.4 Rapid rate of change

The hardware and software environment at Internet services is almost constantly changing, with minor changes occurring every day or two, and major changes occurring every 4-6 weeks. This occurs for several reasons. First, the mostly centralized administration of Internet services makes software upgrades and feature roll-outs much simpler than distributing new versions of shrink-wrapped software or upgrading a decentralized system. Secondly, the rapid growth of Internet computing over the last decade has led to both competitive pressure to improve one's service, as well as increasing demand that requires constant improvements in the scale and scalability of the service. Finally, the size of large Internet services means that hardware (machines, racks, networks) are failing, and need to be replaced over time. The result of these frequent changes and rapid evolution is that the workings of Internet services are often very poorly documented and understood.

2.1.5 Workload characteristics and requirements

All Internet services are open to an uncontrolled external workload from the wide-area Internet. While workload is often very predictable, sites often experience unanticipated flash crowds (the "Slashdot" effect), denial-of-service attacks, spiders and robots unintentionally making intensive requests, worms, malicious hackers, and simply unexpected combinations of requests from regular users. The key point is that Internet services are subject to a poorly understood and poorly controlled workload that can cause a myriad of problems.

While Internet services have a large workload, most incoming requests come from different users, and can be processed independently of each other. This means that the work performed by most Internet services is almost trivially parallelized. Even when there is a potential for interaction between two independent requests, they can often be serviced with a lower level of consistency without doing harm.

In addition to these common features of the workload, many larger Internet services today are also subject to 24x7 availability requirements. Except for a few services with geographically constrained service boundaries, Internet services serve people from around the world. The result is that there is little or no time for scheduled downtime, and failures at any point in time can have serious consequences.

2.2 Fault-tolerance terminology

In this dissertation, we use the *fault*, *error*, *failure* terminology of [4]. A failure occurs when a service deviates from its correct behavior, for some definition of correctness. The definition of a service failure does not necessarily rely on a well-defined specification of its behavior—for example, a broken specification can cause a failure—but on a more abstract, though loosely defined, notion of what a service's correct behavior ought to be. An error is the corrupt system state that directly caused the failure. A fault is the underlying cause of this system corruption. It is worth noting that not all faults cause errors. A fault is called an active fault if it does produce an error, and is dormant otherwise.

It should be noted that problems occur across many layers of a system and what is considered a failure in one layer might be considered the fault that causes an error in a higher layer. For example, a hard disk might crash (failure) because the disk head was given a wrong instruction (error) by a bug in the hard disk's firmware (fault). At a higher level, this hard disk crash might be considered a fault that causes a software program to lose access to important data (error) and cease providing functionality to end-users (failure).

2.3 Failures in Internet services

Despite their growing mission-critical nature, failures, even very serious ones, are quite common in today's Internet services. There are many challenges to providing highly dependable Internet services. The fast software release cycles and the growth rate of Internet services violate one of the traditional dependable computing principles

of minimizing change. In practice, the fast release cycles mean that Internet service software is far from perfect. In addition, frequent software and hardware updates increase the occurrence of operator errors. Worse still, Internet services are exposed to unpredictable workloads [76]. Table 2.1 summarizes outages publicized in the last year. There are likely many more under-reported cases where only capacity or partial functionality was affected.

Recent surveys show that problems are common in Internet services, even otherwise well-managed ones. A series of studies by Business Internet Group of San Francisco (BIG-SF) found that many web sites suffered from undetected application failures [18–20]. Of the 40 top-performing web sites (as identified by KeyNote Systems [79]), BIG-SF found that 72% had had user-visible failures in common functionality at the time of the survey [20]. These user-visible failures included issues such as items not being added to a shopping cart or an error message being displayed. These failures do not necessarily disable the whole site, but often cause brown-outs, where part of a site's functionality is disabled or only some users are unable to access the site.

While brown-outs at Internet services are pervasive, more serious outages also occur frequently, as noted in Table 2.1. The costs of a site outage, where a site's main functionality is essentially disabled or inaccessible, can be substantial. Table 2.2 shows the estimated cost as of the year 2000 of a single hour of downtime at various types of Internet services:

While there have been several academic studies on failures and their causes in other large systems such as mainframes, networked workstations, enterprise servers, and the Internet itself [53, 54, 77, 89, 121, 124], there is only one that we know of that reports directly on causes of failures in Internet services [107]. In that report, Oppenheimer *et al.* study over a hundred post-mortem reports on user-visible failures at three different Internet services, and find that improved detection of application-level failures could have mitigated or avoided 65% of reported user-visible failures.

In addition to [107], in [113], Reimer *et al.* present code analysis techniques to discover several high impact coding errors. To inform and motivate their research, they discuss their three years of experience with source code errors in IBM's source code

(Date) Site	Problem description	Cause	Length
(7/14/05) Walmart.com	Online store "temporarily closed" [115]	Unreported	6-12 hrs
(6/22/05) Blackberry	Disrupted service to customers nationwide [27]	Hardware failure triggered backup server, operating at unexpectedly low capacity	<1 day
(6/15/05) Check Free	Web site and online bill payment unavailable [52]	After power outage, neither master nor backup systems came up.	1 day
(5/08/05) Google	Web site inaccessible [37]	DNS problem	15 min.
(2/23/05) Hotmail	Intermittent access problems [51]	Faulty server	≈2 weeks
(2/08/05) MSN Messenger	Intermittent outage for many customers [68]	"Datacenter issue"	1 day
(12/06/04) Amazon.com	"Service unavailable" [67]	Unreported	5 hrs
(10/20/04) Political sites	Unresponsive site [91]	Unknown	8 hrs
(10/13/04) Paypal	Customers locked out [64]	Software code update	≈5 days
(10/11/04) MSN Messenger	Customers locked out [90]	"Technical glitch"	3 days
(8/02/04) Vonage	Outbound calls broken for some users [28]	Unreported	1.5 hrs
(7/29/04) Microsoft Money	Some users denied access [69]	Unreported	4 days

Table 2.1: **Recent major outages at Internet services.** The descriptions, causes and length of failure are extracted from news reports during the last 12 months.

Site	Cost/hour
Brokerage operations	$6,450,000
Ebay (1 outage, 22 hours)	$225,000
Amazon.com	$180,000
Home shopping channel	$113,000

Table 2.2: **Cost of Internet service downtime per hour.** This information is from an April 2000 InternetWeek and Contingency Planning Research survey,as reported in [109].

production systems. They note that many kinds of failures are not easily discovered during development and testing, only manifesting under the stresses of a production environment. Some of the categories of coding errors they note deal with resource management, concurrency, persistent data management, and violations of informal interface contracts. Based on their experience, Reimer *et al.* propose a simple rule-based approach for detecting these common coding mistakes. In [113], they report that they have written over 400 rules to detect these mistakes.

While failures manifest in many ways, one particularly pernicious class of failures is the *application-level failure* whose only obvious symptoms are changes in the semantic functionality of the system. As further elucidation, let us model a simple system as a layered stack of software, where the lowest layers are the hardware and the operating system, and the highest layer is the application, with various other layers in between (*e.g.,* libraries, middleware software, standard protocols). In this model, an application-level failure manifests solely in the application layer, though the cause of the failure might be in another layer. In particular, an application-level failure is not fail-stop, as this would generally cause several layers of the software stack to stop.

Sometimes failures are caught internally and users see an obvious error message, as in Figure 2.1. We hope that in these cases, the error is being logged and details are being forwarded to operators or developers for diagnosis and repair. Other failures, however, are not obviously caught internally. Figures 2.2 and 2.3 show two examples of application-level failures. While the author is not privy to the detailed cause of these failures, it appears that no symptoms are manifest below the application-layer: the sites respond to pings, HTTP requests, and return valid HTML. Even the performance of the sites appear to be unaffected. Without knowing the cause of the

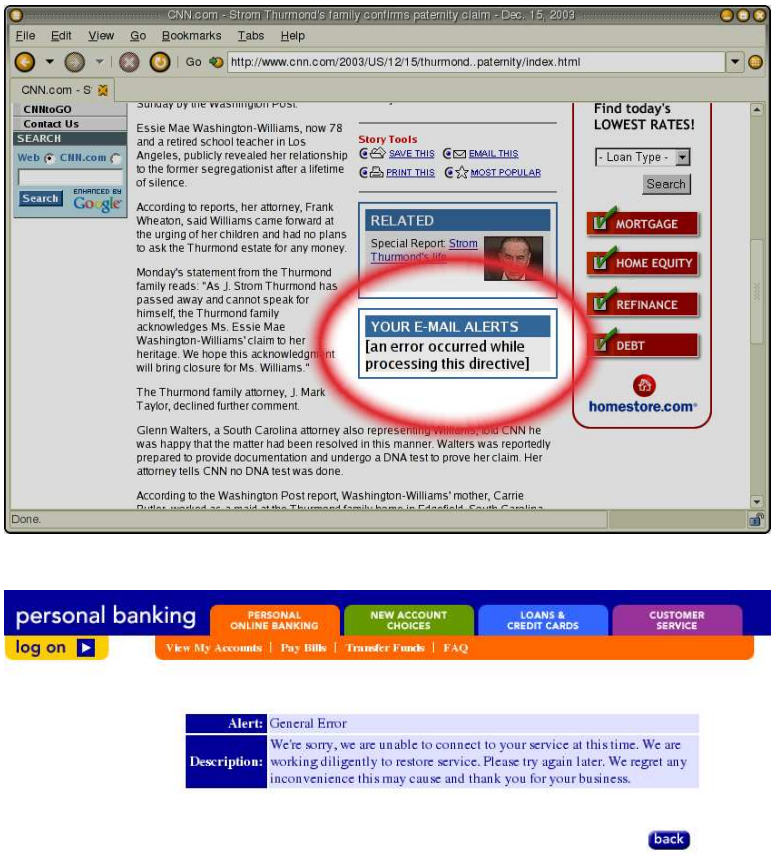


Figure 2.1: **Error messages at two web services.** These screenshots show a site displaying obvious error message instead of providing the correct functionality or attempting to hide the failure.

failure, we cannot be sure that our statistical monitoring approach would detect these specific failures. However, they are still useful as examples of the kind of application-level failures that are easily noticed by users, but difficult for automated monitors to detect without significant application-specific tailoring.

2.4 Existing fault monitors

In [96], Marcus and Stern discuss some of the challenges of building reliable fault detectors. First, any monitor has to be sure it is monitoring all critical components of

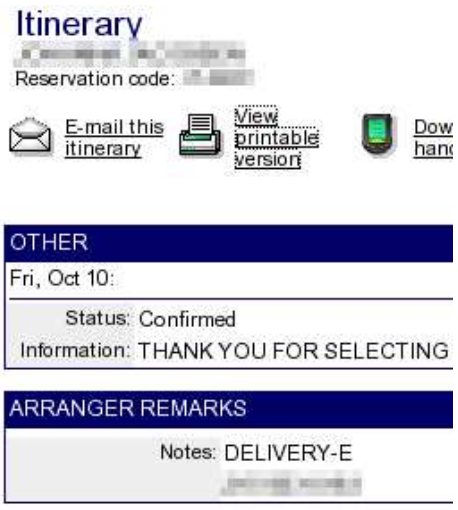
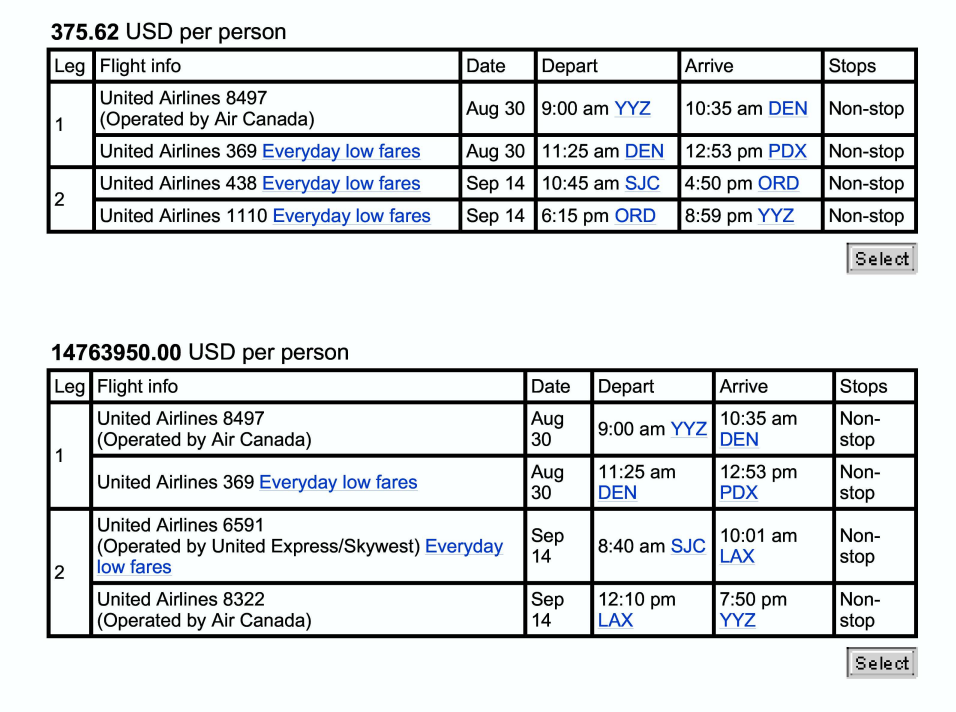


Figure 2.2: **An application-level failure at a web service.** While this page should be displaying a complete flight itinerary, it show no flight details at all. Instead, it shows only an incorrect confirmation date.



375.62 USD per person

Leg	Flight info	Date	Depart	Arrive	Stops
1	United Airlines 8497 (Operated by Air Canada)	Aug 30	9:00 am YYZ	10:35 am DEN	Non-stop
	United Airlines 369 Everyday low fares	Aug 30	11:25 am DEN	12:53 pm PDX	Non-stop
2	United Airlines 438 Everyday low fares	Sep 14	10:45 am SJC	4:50 pm ORD	Non-stop
	United Airlines 1110 Everyday low fares	Sep 14	6:15 pm ORD	8:59 pm YYZ	Non-stop

Select

14763950.00 USD per person

Leg	Flight info	Date	Depart	Arrive	Stops
1	United Airlines 8497 (Operated by Air Canada)	Aug 30	9:00 am YYZ	10:35 am DEN	Non-stop
	United Airlines 369 Everyday low fares	Aug 30	11:25 am DEN	12:53 pm PDX	Non-stop
2	United Airlines 6591 (Operated by United Express/Skywest) Everyday low fares	Sep 14	8:40 am SJC	10:01 am LAX	Non-stop
	United Airlines 8322 (Operated by Air Canada)	Sep 14	12:10 pm LAX	7:50 pm YYZ	Non-stop

Select

Figure 2.3: **Another application-level failure at a web service.** The price quote on the second ticket is likely in error.

the system. The challenge is that the components and their criticality are application- and system-specific. Other issues include avoiding obtrusive monitors that may actually cause harm during normal operation, exacerbate failures, or place excessive load on the system being monitored.

From our discussions with Internet service operators, we find that existing deployed detection methods fall into three categories:

- *Low-level monitors* are machine and protocol tests, such as heartbeats, pings and HTTP error code monitors. They are easily deployed and require few modifications as the service develops; but these low-level monitors miss high-level failures, such as broken application logic or interface problems.

- *Application-specific monitors*, such as automatic test suites, can catch high-level failures in tested functionality. However, these monitors usually cannot exercise all interesting combinations of functionality (consider, for example, all the kinds of coupons, sales and other discounts at a typical e-commerce site). More importantly, these monitors must be custom-built and kept up-to-date as the application changes, otherwise the monitor may both miss real failures and cause false alarms. For these reasons, neither the sites that we have spoken with, nor those studied in [107] make extensive use of these monitors.

- *Business-metric monitors* watch simple statistics about the gross state of high-level metrics relevant to the core business of the Internet service. For example, such a monitor might track the searches per second at a search engine, or the orders per minute at an e-commerce site. These monitors are generally easy to deploy and maintain, and with a site with many users, can detect a broad range of failures that affect the business's well-being. However, these monitors do not often give much more than an indication that something might have gone wrong, and are also often lagging indicators, as it may take minutes or more before a failure trickles through the system to affect a business metric. Also, since these monitors are essentially watching user behavior, they can generate false alarms due to external events, such as holidays or disasters.

Section	Used in ...
2.5.1 Anomaly detection	Ch 4 Monitoring system structure
2.5.2 Probabilistic context free grammars	Section 4.4 Monitoring path shapes
2.5.3 Data clustering	Ch 6 Extracting system structure
2.5.4 Decision trees	Ch 7 Correlating faults to causes

Table 2.3: **What parts of Section 2.5 are used where in the thesis.**

Of course, not all fault monitoring techniques fall neatly into these categories. For example, at many sites, system operators are in the habit of occasionally manually checking system functionality. Finally, the catch-all fault detector is customer service complaints. Unfortunately, poor or slow communication between customer service centers and operation centers, *e.g.,* when a corporation out sources its call centers, can make customer service complaints a very slow indicator of failure.

To be most useful in a real Internet service, a monitoring technique should have the following properties:

High accuracy and coverage: A monitoring service should correctly detect and localize a broad range of failures. Ideally, it would catch never-before-seen failures anywhere in the system. In addition, a monitor should be able to report what part of the system is causing the failure.

Few false alarms: The benefit provided by early detection of true failures should be greater than the effort and cost to respond to false alarms.

Deployable and maintainable: A monitoring system must be easy to develop and maintain, even as the monitored application evolves.

2.5 Statistical analysis and machine learning

Throughout this thesis, we use several different statistical analysis and machine learning algorithms as part of our statistical monitoring process. This section provides a brief background on these algorithms and references to further reading on each. Table 2.3 lists where in the thesis each of these algorithms described is used.

2.5.1 Anomaly detection

Anomaly detection is the general process of analyzing a collection of data points to detect deviations from a normal or common order. With relatively minor differences in application or technique, anomaly detection is also called outlier detection, one-class classification, or abrupt change detection.

Anomaly detection is a tool in broad use. Anomaly detection is used to monitor for failures in mechanical and industrial processes [122], intrusion detection in computer systems [3], fraud detection in financial networks and phone systems [47], and even detection of the signs of disease outbreak [130]. Outlier detection is usually used to refer to the process of detecting anomalies for the purpose of regularizing or "cleaning" data sets and prevent chance impurities from unduly affecting experimental results. Abrupt change detection refers to anomaly detection in the context of time series analysis, where the goal is to detect when the trend in a time series suddenly changes.

In this dissertation, we use statistical techniques to build a model of a believed normal or common behavior of a monitored system. We then use this model to calculate how well subsequent behaviors conform to this model. Anomalies appear as behaviors that do not conform to our learned model. The specific modeling technique we use depends on the type of data being modeled. One technique we use is based on probabilistic context free grammars, presented in the next section. A second test we use is the χ^2 test of goodness of fit.

A test of goodness of fit compares a sample of data to a population with a specific distribution, and computes the likelihood that the sample came from the population. While some tests make assumptions on the underlying distribution of data within the set, others are distribution-free. One such distribution-free test is the χ^2 test, which works on categorical data:

$$T = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i} \tag{2.1}$$

where O_i is the observed occurrences of the i^{th} category of data and E_i is the estimated occurrence of the i^th category of data according to our model. If T, our test statistic, is small, then there is a good fit between the observed data and our

estimated model, whereas if T is large, the fit is poor. We can determine the statistical significance of a fit for a given significance α by comparing T to the χ^2 distribution with $k-1$ degrees of freedom. If T is greater than the $1-\alpha$ quantile from the χ^2 distribution, then we have a high confidence that our observed data does not fit our model's distribution, and was likely generated by a different underlying process.

For more details on nonparametric statistics, including other tests of goodness of fit, see [40].

2.5.2 Probabilistic context-free grammars

A probabilistic context-free grammar (PCFG), is a structure used in natural language to calculate the probabilities of different parses of a sentence. A PCFG consists of a set of grammar rules, $R_{ij} : N^i \rightarrow \zeta^j$, where N^i is a symbol in the grammar and ζ^j is a sequence of zero or more symbols in the grammar. Each grammar rule is annotated with a probability $\mathrm{P}(R_{ij})$, such that $\forall i \Sigma_j R_{ij} = 1$.

The probability of any given parse tree is given by the product of the probabilities of all the grammar rules involved in the production of the parse. The probability of a sentence occurring in the language represented by that grammar is the sum of the probabilities of all the legal parsings of that sentence.

For more details on probabilistic context free grammars, see [95].

2.5.3 Data clustering

Data clustering is a form of machine learning or data mining used for descriptive modeling. Data clustering attempts to take a set of unlabeled data (that is, a set of vectors that are not organized or sorted), and organize the data such that similar elements are grouped together and dissimilar elements are not grouped together. The definition of similarity is domain-specific and can be any distance metric.

There are several forms of data clustering algorithms: hierarchical clustering methods; partition-based clustering, such as k-means; as well as a third recent addition, correlation clustering [8, 41]. While all of these methods have their advantages and disadvantages, we focus our description here on hierarchical clustering.

Hierarchical, bottom-up clustering methods begin by initially placing each element into its own singleton cluster. Then, the two clusters that appear to be the most similar to each other are merged together to form a new cluster. This process is repeated until either all clusters are merged together into one, or all remaining clusters are more than some dissimilarity threshold apart.

There are two similarity (or distance) functions that need to be specified for hierarchical clustering. The first is the similarity between two objects, and is usually a domain-specific function. The second function defines the distance function between two clusters. Here, there are several standard options: single linkage or nearest neighbor distance defines the distance between two clusters to be the distance between the two closest elements in the different clusters; complete linkage or furthest neighbor distance defines the distance between two clusters to be the distance between the two elements furthest apart in the different clusters; group average linkage defines the distance between two clusters to be the average distance over all pairs of elements across the two clusters.

For more details on data clustering algorithms, see one of [44, 62].

2.5.4 Decision trees

A decision tree is a data structure that represents a discrete-valued function, where each branch of the tree is a test of some attribute of an input, and where the leaves of the tree hold the result of the function. Decision tree learning is the process of building a decision tree to approximate some unknown discrete valued function. While our description of decision trees focuses on boolean valued functions, decision trees can be generalized to any discrete valued functions.

During the process of decision tree learning, a learner is provided with a set of training data. This data consists of a set of inputs, and how these inputs have been labeled by the function that we want to approximate with a decision tree. Oftentimes, only part of the available training data is used for learning a decision tree, while the rest of the data is set aside to validate the result.

One of the simpler decision tree learning algorithms is the ID3 algorithm [110]. ID3 constructs a decision tree top-down, beginning at the root of an empty tree. To

determine what test to use at any given node in the decision tree, ID3 calculates the entropy (or homogeneity) of the data. Given a collection of data containing both true and false labels, the entropy of the set can be calculated by:

$$Entropy(S) = -p(t)\log p(t) - p(f)\log p(f) \tag{2.2}$$

where $p(t)$ is the proportion of true examples, and $p(f)$ is the proportion of false examples.

The general approach of ID3 is to calculate the entropy of the data at each node of the tree, and choose a test that will split the data in a way that minimizes the entropy of the child nodes. If the entropy of the data at a node is below a threshold, then ID3 creates a leaf node and labels it with the majority value of the data at the node. Further refinements of ID3 include algorithms that post-prune trees to avoid overfitting, converting trees to a series of if-then rules, and efficiently learning trees online from a stream of incoming data.

For a more in-depth introduction and more details on decision tree algorithms, see any one of [44, 62, 100].

Chapter 3

Related work

Related work to this thesis can be roughly divided along two axes. The first axis splits related work by the problem domain, either fault fault management in Internet services and other large-scale systems, or fault management in the context of other systems. The second axis splits related work by the approach, either statistical techniques such as our statistical monitoring, or non-statistical techniques. Together, this places related work into three loosely-defined, broad categories, which we explore in this chapter.[1]

3.1 Statistical analysis for Internet service management

The category of related work closest to our own is that of using statistical techniques to help reason about or manage Internet services. This is a relatively young area of research, having begun in the last several years. However, there are several significant projects.

At Hewlett Packard Labs, Cohen *et al.* developed *metric attribution*, correlating SLA violations in Internet services with monitored metrics within the system [38,133]. In effect, when something is going wrong, they are able to give a list of metrics, such

[1] A fourth category in this taxonomy of applying non-statistical techniques to problems outside Internet services is only distantly related work, and omitted from this chapter.

a high load on the database or drop in network traffic along a link, which are highly correlated with the an externally visible SLA violation. This work is most related to our fault localization work, except that they apply correlation techniques to find metrics related to the underlying problem, but without requiring any tracking of dependencies between components and end-user requests. At the same time, they do assume the existence of a simple fault detector, such as a check for SLA violations, and might have a more difficult time correlating metrics for rare or minor failures, such when a fault affects only a small number of end-user requests.

In [39], Cohen *et al.* extend this work to use the results of metric attribution as a signature for failures. That is, he suggests identifying failures using the list of metrics correlated with the failure. He evaluates several interesting uses of these signatures, including searching for related problems in the past or in other systems, and tying together failures over time to identify recurring problems.

Also at Hewlett Packard Labs, Project5 uses signal-processing techniques to reconstruct likely runtime paths that flow through a distributed system [1]. Using only observations about timing information of communication between individual components, Project5 uses two algorithms, a convolution algorithm and a nesting algorithm, to compute causal connections between input and output messages to a component, and thus piece together runtime paths, including average timing information. Unlike our runtime path instrumentation, these techniques work in black-box environments where there is no access to the software of the system. While the convolution algorithm only provides performance information about the majority behavior of the system, and cannot deduce information about specific requests or request behaviors in the minority, the nesting algorithm can identify these anomalous paths. While Project5 does not itself attempt to detect or rank anomalies in the system, through its nesting algorithm, Project5 can provide the observational data required for an anomaly detector, including the fault detectors we describe in this dissertation.

In [12], Bodík *et al.* detect failures in Internet services by looking for changes in user behavior. They use a combination of statistical analysis to highlight potential anomalies and clever visualization to improve operator's confidence in their techniques. Analysis of HTTP logs from a real site, they were able to detect several

failures, sometimes significantly before existing processes noticed the problem. By analyzing user behavior, they avoid the problem of instrumenting the Internet service itself. In its goal of improving fault detection, this work is the closest to ours. The primary differentiator between our goals is that [12] develops a system primarily for use by operators, whereas we develop a system that can provide information to either operators or further stages of automatic fault localization and recovery.

The Magpie project, developed at Microsoft Research Cambridge aims to extract representative models of a system's resource usage and behavior for the purpose of performance modeling and debugging [9,10]. First, Barham *et al.* use fine-grained instrumentation, such as interrupts, thread context switches and network packet transfers, within the operating system to track extremely detailed resource usage characteristics for requests, and then use a middleware- or application-specific schema to translate their observations into the equivalent of a very detailed runtime path. Once these runtime paths have been captured, Magpie uses data clustering techniques to discover representative runtime paths that express a compact model of the system's overall behavior. Like our own approach, Magpie's instrumentation is deterministic, while the analysis of their data is statistical. The major difference between our work is that, while the goal of our work is to detect and localize failures, including application-level problems, Magpie concentrates on characterizing system behavior, with a specific focus on performance modeling.

At U.C. Berkeley and EBay, Chen, Zheng *et al.* use a decision tree learning approach for fault localization,very similar to ours [30]. While Chen *et al.* assume the existence of pre-labeled data (*i.e.,* failed or successful requests) and do not attempt fault detection before localizing a problem, we believe that their exploration of decision trees is complementary to our own work, and could easily be adapted to use the output of our fault detectors as the labels on their data.

At NEC Labs, Jiang *et al.* have taken our own Pinpoint prototype, and replaced our PCFG-based path-shape analysis with one based on multi-resolution learned automata [75]. We, as systems researchers, are particularly excited that researchers in the machine learning community are taking our systems problems and looking for better algorithms to solve them. When we began to cast our problems into the

machine learning domain, this is exactly what we hoped would happen.

Applying control theory to dynamically adapt system behavior is a topic of considerable interest, especially in the context of autonomic computing, IBM's initiative to build self-managing and self-configuring systems. In [108], Parekh *et al.* demonstrate how to apply classical control theory to managing the performance of a system, and empirically validate their approach by applying their controller to a Lotus Notes groupware server.

3.2 Other techniques in Internet service management

There are many problems in Internet service management, including fault detection and localization that can benefit from improved architectures and more complete and careful observation and analysis of system behavior. In this section, we review some of these non-statistical techniques for improving Internet service management.

ARMOR is a software-based fault tolerance environment for managing failures in the context of critical systems such as financial, health and telecommunications systems [6,129]. The goal of ARMOR is to detect failures and repair them in componentized systems built to the Chameleon ARMOR architecture. ARMOR takes a layered approach to fault detection, with many fault monitors watching each component of an application. The layers are categorized by the degree of application-integration and application-specific knowledge used by the detector. The first level of detectors are assertions and livelock checks that are built into a component at design time. The second level of detection is done at the local machine level. Here, process exit status and heartbeats are used to determine whether a failure has occurred. Level three and four fault detection protocols include cross-machine failure detection, such as signature checking to avoid data corruption and byzantine agreement protocols. While the ARMOR system can detect a wide range of failures, detecting application-level failures does require an *a priori* understanding of the application functionality at each node, and the insertion of assertion checks within components.

There has been extensive literature on event correlation systems [13, 116], mostly

in the context of network management. There are also many commercial service management systems that aid problem determination, such as HP's OpenView [63], IBM's Tivoli [71], and Altaworks' Panorama [2]. These systems mainly use two approaches. The first approach uses expert systems with rules (or filters) input by humans or obtained through machine learning techniques. The second approach uses dependency models [36, 58, 132]. However, these systems do not consider how the required dependency models are obtained. Usually, they require system operators to input the dependency models in either *a priori* , or as required during system monitoring.

To address the issue of creating dependency models, Brown *et al.* use active perturbation of the system to identify dependencies and use statistical modeling of the system to compute dependency strengths [16]. The dependency strengths can be used to order the potential root causes, but they do not uniquely identify the root cause of the problem, whereas our approach uniquely identifies the root case, and is limited only by the coverage of the workload. The intrusive nature of their active approach also limits its applicability in production systems. In addition, their approach requires components and inputs to be identified before the dependencies can be generated, which is not required in our approach.

3.3 Statistical techniques for management of other systems

Recently, statistical analysis and machine learning techniques have begun to see significant use in the general context of fault and problem management, across wide set of domains.

Many approaches to intrusion detection rely on statistical techniques to search for anomalies that might be signs of intruders [3]. On the surface, fault detection in Internet services and intrusion detection share many similarities. However, seemingly minor differences in the domain make significant differences. Primarily, intrusion detection has a very low tolerance for false positives because the prudent responses to a potential break-in are generally expensive—shutting down the system. In contrast,

Internet services have cheaper responses available, such as quickly rebooting misbehaving nodes. Other differences include the high workload on Internet services that allows statistical techniques to build more complete models of acceptable behavior. Together these differences make Internet services a more suitable environment for applying statistical techniques to fault detection.

In the area of bug discovery, Engler *et al.* use pattern matching techniques to infer correctness rules upon the source code of a large system. These rules can include, for example, function-call ordering constraints, and requirements for locking critical sections. Once these correctness rules are discovered, Engler *et al.* use them to search for deviations from these rules within the source code [45]. Kremenek *et al.* continue this work in [86, 87], using statistical analysis techniques to reduce the false positive rate when searching for bugs in systems source code.

Anomaly detection has also been used to detect failures in many other kinds of systems, outside the computer domain, such as in mechanical systems [122], nuclear power plants [126], and cellular phone fraud detection [47].

To diagnose configuration errors in the Windows Registry, the Strider project uses dynamic tracing of registry accesses by errant applications together information about recent registry changes and typical change frequencies to help narrow down suspect configuration settings and fix a problem [128]. A related project, PeerPressure, uses a peer-comparison method to isolate configuration errors in individual machines within a larger population [127].

Recently, Liblit *et al.* have proposed an approach they call statistical debugging [92], where they advocate constant sampling of the code-level behaviors of end-user software during normal execution. These behaviors include the results of conditional tests, the return values of functions, etc. By discovering which of these behaviors are most correlated with symptoms of a Heisenbug, statistical debugging helps programmers understand and discover a bugs true cause.

In [85], Kompella *et al.* present an approach to fault localization in IP networks using a model of shared risk. They model the potential fault points in the network and what observable symptoms, such as alarms at particular routers, a fault at each of these points might cause. Then, given a set of alarms observed over a short window

of time, the problem of determining what fault point(s) likely caused the observed failure symptoms can be reduced to a graph cover problem.

In [119], David Sullivan uses Bayesian networks to reason about software configuration and performance tuning in a database system. His technique uses a combination of an expert-designed Bayes network with observations of system behavior to build a controller that automatically adapts a database's configuration to current workload characteristics. In their experiments, the controller does quite well, exploiting the configuration knobs of the system to improve performance in ways that even the database's own designers had not anticipated.

In the context of file systems, Mesnier *et al.* explore the problem of setting file management policies to improve performance. The difficulty of this problem lies in the number of policy settings, such as cache write-back times, file placement settings, etc. based on assumptions of how a file will be accessed in the future. In [97], Mesnier *et al.* apply decision trees to classify files and select appropriate storage policies based on easily observable file attributes. They find that their prediction accuracies often exceed 90% and improve file system performance without requiring administrators to understand the expected file system workload.

Statistical inference techniques are slowly gaining acceptance in the systems community, and are being applied to a wider variety of tasks, not only in systems management, but in other aspects of systems design. For example, in [42], Deshpande *et al.* apply statistical modeling to improve the performance and efficiency of data-acquisition in wireless sensor network. They do so by using a learned statistical model of the environment and environmental data to tailor a query plan to balance the efficiency of sensor readings against the desired accuracy of the result. As statistical techniques continue to be applied in more varied domains, we hope that the general experience and lessons presented in this dissertation may be of broader use.

Chapter 4

Monitoring dynamic system behaviors

Despite operators' best efforts at monitoring Internet services for signs of failure, the time it takes to notice that a system is failing is often the largest component in the overall time to recover from the failure. The difficulty is that many faults often exhibit symptoms only at the application-level, and do not show symptoms, such as machine crashes, that operators can notice easily. What this means is that while end-users notice these high-level problems quickly, it can take minutes or hours after a failure first occurs before operators notice. To help us detect failures faster, we would like to know when the application-level functionality of the system has changed. Unfortunately, measuring or monitoring this directly is impractical.

A key insight is that the software components within Internet services are usually defined at a granularity tied to application functionality. While this is not true for all software systems, it is common in Internet services because of software engineering and scalability concerns. For example, in an e-commerce site, one software component (or small group of software components) might be responsible for managing users' shopping carts, while another might be responsible for the product catalog. Because of this, how the internal components of the system interact with each other to service end-user requests reflects the application-level functionality that the whole system is providing. Thus, we can use this internal structure as a proxy for application

functionality, and look for changes in it, in lieu of directly monitoring application functionality, and without requiring *a priori* knowledge about the application.

Consider the example e-commerce site we described in Chapter 2. If the shopping cart component in this site is communicating with the product catalog component, there's likely to be a semantic reason for the communication. By observing these interactions in a live system, we can develop a model of likely correct functionality. Because these behaviors reflect the semantics of the service, when these patterns change, we can have a high confidence that the service's high-level functionality has also changed, indicating a possible failure. Noticing changes and outliers in these patterns is a natural fit for anomaly detection algorithms. Note that in using anomaly detection to analyze the structural behavior of a system, we are not deriving any truth about the actual application-level functionality of the system, and effectively side-stepping the thorny problem of representing and reasoning about the semantics of an application.

There are many different kinds of structural behaviors that one can analyze to give insight into application functionality and exploit to improve the management of an Internet service. In this chapter, we first describe the general approach of monitoring structural behavior. Then, we focus on two specific structural behaviors, component interactions and path shapes, and explore how modeling and analyzing them can help us detect failures in application functionality. Also, recognizing that not all systems have the complex structure to allow analyzing structural behavior to be insightful and useful, an additional section describes our experience detecting failures through statistical monitoring of non-structural behaviors of components.

4.1 Monitoring and analysis procedure

Our approach for monitoring a system for anomalies (likely failures) can be divided into a straight-forward three-stage process:

1. **Observation:** We capture the *runtime path* of each request served by the system: This path is the ordered set of coarse-grained components, resources, and control-flow used to service a client's request. From these paths, we extract two

specific low-level behaviors likely to reflect high-level functionality: component interactions and path shapes.

2. **Learning:** We build a reference model of the fault-free behavior of an application with respect to component interactions and path shapes, under the assumption that most of the system is working correctly most of the time.[1]

3. **Detection:** We analyze the current behavior of the system and search for anomalies with respect to our learned reference model.

During the observation phase, we capture the runtime paths of requests by instrumenting the middleware framework used to build the Internet service. As these middleware frameworks wrap all the application's components and manage their invocations, instrumenting the middleware gives us the visibility we require. In addition, by instrumenting a standard middleware, such as J2EE or .NET, we have the ability to observe any application built atop it. Section 5.3.1 describes the instrumentation used in our own testbed systems.

Before analyzing these observations, we "bin" our runtime paths by their request type. By analyzing each type of request separately, we aim to improve the resilience against changes in the workload mix presented to the Internet service. The degree of resilience is determined by the quality of the binning function. In our testbed, we use the URL of a request; a more sophisticated classifier might also use URL arguments, cookies, etc. Of course, resilience against changes in workload comes at the cost of hiding failures whose only symptom is a change in user behavior. This trade-off can be explicitly managed by using a combination of binned and non-binned models to monitor system behavior. We believe that to catch these failures, however, it is best to use complementary techniques, such as [12], to explicitly monitor user behavior.

We learn a *historical* reference model to look for anomalies in components relative to their past behavior, and a *peer* reference model to look for anomalies relative to the current behaviors of a component's replicated peers. As shown in Table 4.1, these two models complement each other: a historical analysis can detect acute failures, but

[1]We refer to the *fault-free* behavior of a system instead of *normal* or *correct* behavior. We wish to avoid confusion with the statistical definition of normal distributions; as well as the implication of a formal proved correct behavior.

	Acute failure	**Existing failure**
Partial system failure	Historical and peer	Only peer
Whole system failure	Historical only	Neither

Table 4.1: **Categories of failures detected by historical or peer reference models.** We can categorize failures along two axes: (1) whether they occur acutely or have always existed in the system; and (2) whether they affect the part of the system or the whole system. This table shows which of these categories are detectable by historical or peer reference models.

not those that have always existed in the system; peer analysis, which only works for components that are replicated, is resilient to external variations that affect all peers equally (such as workload changes), but a correlated failure that affects all peers equally will be missed. Steady-state failure conditions affecting the whole system would not be detected by either type of analysis.

To detect failures, we compare the current behavior of each component to the learned reference model. The details of model representation and anomaly detection are specific to the individual behaviors we monitor. We discuss these in more detail in sections 4.3 and 4.4.

Once a failure has been detected, a separate policy agent is responsible for deciding how to respond to discovered failures. While a full analysis of how to react to failures is outside the scope of this thesis, one such policy agent is discussed and evaluated in Chapter 8.2.

4.2 Basic system model and assumptions

We abstractly model an Internet service as a set of components interacting with each other to service user requests. While the precise definition of components is specific to an environment and system, the components of our model generally refer to the hardware and software nodes within the Internet service. We use a slightly abstracted concept of a request to represent the primary unit of work, as well as the unit of success and failure, in an Internet service. These requests may be end-user requests, or they may be work triggered internally in a service. In our model, the structure of a request is essentially the ordered tree of components used to service

the request.

In our model, all interactions between components are made in the context of (and on behalf of) a specific request. This implies that if we are able to observe the behavior of all requests in the system, we are also observing the entire behavior of the system. The goal of the request abstraction is to encapsulate the smallest observable unit of application failure or success.

An instance of component, such as a specific software process running on a specific machine, is a *physical component*. Every physical component is also associated with a component type or *logical component*. Thus, while we directly observe the physical structure and behavior of a service, we can also reason about its logical structure and behavior. We simply merge all physical instances of a component class into a single logical component, and discard details about the physical components from the request's associated component tree.

To apply our statistical monitoring approach, we make several general assumptions about Internet services and their behavior. First, our analysis of component-level interactions fundamentally assumes that the behaviors we are observing are effective surrogates of application-level functionality, and that a fault that causes the application to seriously change its behavior will also cause noticeable changes in how the system services requests.

A second general assumption allows us to easily build a model of the likely correct behavior of an Internet service: most of the time, most of the system is working correctly. This allows us to use the majority of our observations to build a reference model of likely correct behavior. While this assumption does not require the system to be completely fault-free, it does mean that if we are only able to observe a system while most of its components are failing, our reference models will represent incorrect behavior, and could even lead to misclassifying correct behavior as faulty! Of course, while Internet services are nowhere near perfect, we would hope that given a reasonably administered system and observations over a long enough period of time, that our weak assumption would hold true.

We also make the following secondary assumptions about the system under observation and its workload:

1. **Component-based:** the software is composed of interconnected modules (components) with well-defined narrow interfaces. These may be software objects, subsystems (*e.g.,* , a relational database may be considered a single, large black-box component), or physical node boundaries (*e.g.,* , a single machine running one of the web server front-ends to an Internet service).

2. **Request-reply:** a single interaction with the system is relatively short-lived, and its processing can be characterized through its runtime path.

3. **High volume of largely independent requests** (*e.g.,* , from different users): combining these allows us to appeal to "law of large numbers" arguments justifying the application of statistical techniques. The high request volume ensures that most of the system's common code paths are exercised in a relatively short time, allowing us to quickly and dynamically build accurate and mostly-complete models of system behavior. This assumption does not hold for some other applications of anomaly detection, such as intrusion detection in multi-purpose or lightly-used servers, in which it is not reasonable to assume that we can observe a large volume of independent requests.

In a typical large Internet service, (1) arises from the service being written using one of several standard component frameworks, such as .NET or J2EE, and from the clustered and/or tiered architecture [14, 72] of many such services. (2) arises from the combination of using a component-based framework and HTTP's request-reply nature. (3) arises because of the combination of large numbers of (presumably independent) end users and high-concurrency design within the servers themselves.

4.3 Structure #1: Component interactions

The first low-level behavior that we analyze is component interactions. An interaction between components can be any method call, message passing, or call return that crosses a component boundary and triggers some software execution. Specifically, we model the interactions between a physical component and each component class in the system.

The intuition behind looking for anomalies in component interactions lies in the fact that components in Internet services are defined at a granularity that is tied to application functionality. This means that if we observe an interaction between two components, we can assume that it is providing some functionality at the application-level. If we observe this same interaction occur many times, we may believe that the provided functionality is stable and perhaps an important part of the application. If this interaction then disappears or changes, we can surmise that the application-level functionality of the service must also have changed, and that this might indicate a failure.

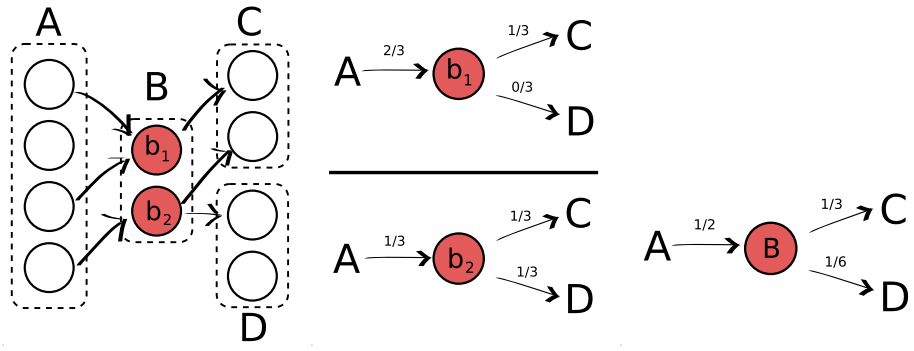
This component interaction analysis can allow us to notice gross failures in the behavior of a component instance. For example, if a shopping cart component completely ceases to make method calls to a database, it is likely that the shopping cart component is failing. However, more subtle failures in a component, such as the mishandling of a single request, are unlikely to provide statistically significant evidence of a component failure. Nor will component interaction analysis point out individual requests that have failed. Some of these shortcomings are addressed through the complementary analysis of path shapes, discussed in Section 4.4.

4.3.1 Modeling component interactions

We represent the interactions of a component instance as a set of weighted links, where each link represents the interaction between a component instance and a class of components, and is weighted by the proportion of runtime paths that enter or leave a component through each interaction.

As a more formal example, assume a system is constituted of four classes of components, A, B, C and D, shown in Figure 4.1(a). If b_i is an instance of B, we will model b_i's component interactions with three links: $b_i \rightarrow A$, representing how often b_i interacts with any instance of A; $b_i \rightarrow C$, representing how often b_i interacts with any instance of C; and $b_i \rightarrow D$, representing how often b_i interactions with any instance of D. Figure 4.1(b,c) show the models that result from the interactions.

We do not analyze interactions between two individual instances because in many systems, this level of interaction is not identical across instances of a component.



(a) Component interactions

(b) Component instance models

(c) Component class model

Figure 4.1: **Component interaction model.** (a) A diagram of the interactions between components b_1, b_2 and other classes in the system. (b,c) Component interaction models, based on the interactions in (a), for the component instances b_1 and b_2, as well as the whole class of components B.

For example, some systems use a notion of affinity, where a component a_1 will only communicate with b_1, a_2 with b_2, etc.

Because our analysis only considers the interactions between classes of components, and not the interactions between physical instances, our analysis will not directly detect affinity-related failures. For example, our analysis would not detect lowered system performance due to an affinity relationship not being respected. However, our analysis would detect a problem if an affinity error caused a failure of application-level functionality. For example, if a request fails because a user's current data is not available on a node then we would expect to be able to detect the changes in component interactions caused by this failure.

More generally, for a system with N component classes, the model of component interactions consists of a set of N weighted links for each component in the system, subject to the following constraints. First, $|c_i \rightarrow C_j| <= 1$, for all component instances c_i and component classes C_j. Secondly, if $c_i \in C_j$, then $|c_i \rightarrow C_j| = 0$. Finally, $\forall c_i$, $\sum_{j=1}^{N} |c_i \rightarrow C_j| = 1$.

We generate our historical reference models of component interactions by averaging the weights of links through them over time. Our peer reference model is

generated by averaging the current behaviors of replicated peers in the system.

4.3.2 Detecting anomalies in component interactions

We detect anomalies by measuring the deviation between a single component's current behavior and our reference model using the χ^2 test of goodness-of-fit:

$$Q = \sum_{j=1}^{k} \frac{(N_j - w_j)^2}{w_j} \tag{4.1}$$

where N_j is the number of times link j is traversed in our component instance's behavior; and w_j is the expected number of traversals of the link according to the weights in our reference model. We calculate w_j as $|j|N$, where N is the total number of link traversals observed involving the observed component. If the observed component behavior matches our reference model exactly, then $N_j = w_j$.

Q is our confidence that the fault-free behavior and observed behavior are based on the same underlying probability distribution, regardless of what that distribution may be. The higher the value of Q, the less likely it is that the same process generated both the fault-free behavior and the component instance's behavior.

We use the χ^2 distribution with $k - 1$ degrees of freedom, where k is the number of links in and out of a component, and we compare Q to an anomaly threshold based on our desired level of significance α, where higher values of α are more sensitive to failures but also more prone to false positives.

One of the primary advantages of using the χ^2 test of goodness-of-fit is the fact that it is a non-parametric and distribution-free test. This allows us to apply it to our observations of component interactions without assuming that this data follows, for example, a normal distribution. However, the χ^2 test does not take into account any *a priori* margin for error. That is, we cannot ask whether our observations match our model plus or minus some percentage error, but only whether or not our observations might match our model exactly. This means that as we gather more observations, even the smallest variations in system behavior will eventually become statistically significant. In some ways, this is a positive feature, allowing us to detect even small changes in system behavior, but ideally, we would have the ability to control this

behavior, to test explicitly whether our observations show a statistically significant deviation beyond some acceptable error ϵ as compared to our model.

The Kolmogorov-Smirnov (KS) and the Anderson-Darling (AD) tests are also non-parametric and distribution-free tests of goodness-of-fit. However, unlike the χ^2 test, KS and AD operate on non-binned data, and would be a poor fit to our naturally binned counts of component interactions. Another alternative is to use a simple similarity or distance measures, such as the Jaccard similarity coefficient. A function such as Jaccard can be used to measure the similarity or dissimilarity between current observations and fault-free behavior. However, these distance measures generally do not have a statistical grounding, and in our initial experience, have not proved as useful as χ^2.

4.4 Structure #2: Path shapes

A *path* is the set of components, resources, and control flow associated with the processing of a user request. While earlier conceptualizations of paths in Scout [102] and Ninja [57] primarily defined paths as a static data flow, we extend paths to include runtime properties as well. In this context, a path is the ordered set of physical components (or component instances) and resources dynamically chosen to satisfy a request.[2]

A path's *shape* provides the same dynamic information about a request, but disregards information about component instances and specific physical resources; including instead information about the component classes and types of resources used to satisfy a request.

We use the term "request" in a broad sense to mean a unit of work. This includes both requests that require responses, such as HTTP requests from end-users and those that do not, such as one-way messages. Our definition of paths is trivially extended to include requests triggered internally by a system, as opposed to being triggered by an external workload.

[2]Scout defines a path as "a logical channel through a multi-layered system over which I/O data flows within a single host." Ninja defines a path as "a flow of typed data through multiple services across the wide area."

There are two requirements for the monitoring of runtime paths to be possible and useful. First, it must be possible to associate a unique path with each distinct request. For example, if the same request is handled by many components across different, possibly distributed, processes, we must be able to associate the work done by these many components with the triggering request.

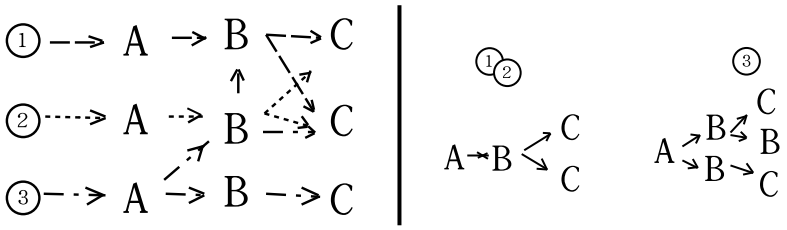
Second, to be a useful unit of analysis, a path must represent a relatively coarse-grained (and complete) unit of work. This is the case in most Internet services, where the relatively high latencies of a wide-area network and the simple interaction model of web browsing precludes extremely fine-grained units of work. It is not the case, however, for all systems, such as remote procedure call (RPC) servers where more complicated client-side software may initiate many requests of a server to complete a single unit of work.

Path shapes capture a different aspect of system behavior than component interactions. While component interactions represent how a single component behaves as it processes many requests, path shapes capture how a single request is processed by many components. This also makes it likely that different kinds of faults can be detected by analyzing these two structural behaviors.

One kind of failure that can be detected via path shape analysis, but not by component interactions, would be an occasional error that affects only a few requests out of many. Since few requests are affected, the fault would not significantly change the weights of the links in a component interaction model. Path shape analysis, however, could detect anomalies in the individual paths. As a converse example, consider that it is normal for a password-verification component to occasionally reject login attempts. Path shape analysis of a request that ended in a login failure would therefore not be considered anomalous in its own right, even if the login was rejected erroneously. However, if the password-verification component was rejecting too many login requests, then a component interaction analysis would detect an anomaly.

4.4.1 Modeling path shapes

We represent the shape of a path in a call-tree-like structure, except that each node in the tree is a component rather than a call site. *i.e.,* calls that do not cross component



$R_{1,1}$:	$\mathbf{S} \rightarrow A$	$p = 1.0$	$R_{3,2}$:	$B \rightarrow C$	$p = 0.2$
$R_{2,1}$:	$A \rightarrow B$	$p = 0.66$	$R_{3,3}$:	$B \rightarrow CB$	$p = 0.2$
$R_{2,2}$:	$A \rightarrow BB$	$p = 0.33$	$R_{3,4}$:	$B \rightarrow \$$	$p = 0.2$
$R_{3,1}$:	$B \rightarrow CC$	$p = 0.4$	$R_{4,1}$:	$C \rightarrow \$$	$p = 1.0$

Figure 4.2: **Path shape model.** Top left: a set of three inter-component call paths through a system consisting of three component types (A, B, C). Top right: Call paths 1 and 2 have the same shape, while 3 is different. Bottom: PCFG corresponding to this collection of paths. **S** is the start symbol, **$** is the end symbol, and A, B, C are the symbols of the grammar.

boundaries are hidden.

We model a set of path shapes as a probabilistic context-free grammar (PCFG) [95], a structure used in natural language processing to calculate the probabilities of different parses of a sentence. A PCFG consists of a set of grammar rules, $R_{ij} : N^i \rightarrow \zeta^j$, where N^i is a symbol in the grammar and ζ^j is a sequence of zero or more symbols in the grammar. Each grammar rule is annotated with a probability $P(R_{ij})$, such that $\forall i \Sigma_j R_{ij} = 1$. The probability of a sentence occurring in the language represented by that grammar is the sum of the probabilities of all the legal parsings of that sentence.

In our analysis, we treat each path shape as the parse tree of a sentence in a hypothetical grammar, using the component calls made in the path shapes to assign the probabilities to each production rule in the PCFG. Figure 4.2 shows an example of a trivial set of path shapes and the corresponding PCFG.

To build a historical reference model, we build a PCFG based on a set of path shapes observed in the past, and our peer reference model from the path shapes observed in the last N minutes. In both cases, we want to be sure that our models are based on enough observations that we capture most of the different behaviors in the system.

4.4.2 Detecting anomalous path shapes

Once we have built a reference model, we use it to calculate an anomaly score for subsequently observed path shapes. To calculate this score, we start at the root of the tree of component calls in a path shape and compare each transition in the tree to its corresponding rule R_{ij} in our PCFG:

$$\sum_{\forall R_{ij} \in t} \min(0, P(R_{ij}) - 1/n_i) \tag{4.2}$$

where t is the path shape being tested, and n_i is the number of production rules in our PCFG where the left-hand side is N^i. The simple intuition behind this scoring function is that we are measuring the difference between the probability of the observed transition, and the expected probability of the transition at this point. We use this difference as the basis for our score because in these systems, we have found that low probability transitions are not necessarily anomalous. Consider a PCFG with 100 equally probable rules beginning with N^i and probability 0.005 each, and 1 rule with probability 0.5. Rather than penalize the low-probability 0.005 transitions, this scoring mechanism will calculate that they deviate very little from the expected probability of 1/101. Figure 4.3 shows how this scoring function separates fault-free paths from faulty paths in one experiment.

Because rare requests can result in rare (anomalous) path shapes, it is expected that in a large system there may always exist a small number of requests that are assigned relatively high anomaly scores. To decide whether there may exist an actual failure requires an additional step of processing to look at the rate of anomalies across many requests.

After scoring our path shapes, if more than αn paths score above the $(1-n)^{th}$ percentile of our reference model's distribution, we mark these paths as being indicative of a true failure in the system. For example, any path with a score higher than any we have seen before (*i.e.,* above the 100^{th} percentile) will be marked as likely faulty requests. Similarly, if $\alpha = 5$ and 1% of paths suddenly score higher than our historical 99.9^{th} percentile, we will mark these 1% of paths as likely faulty, since $0.01 > 5 * (1 - 0.999)$.

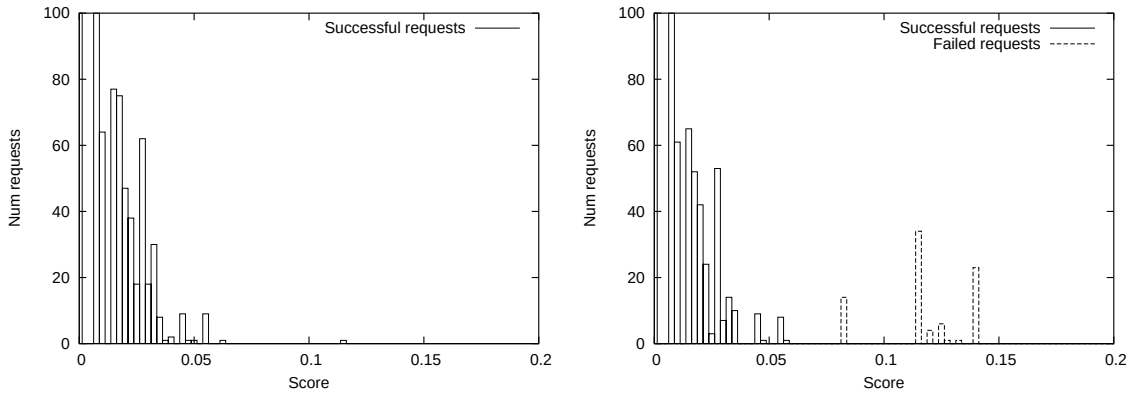


Figure 4.3: **Histogram of path shape scores.** The left graph is a histogram of path shape scores during fault-free operation. The right shows the scores when we cause part of the system to misbehave. Some requests fail as a result, and are clearly separated from the successful requests by our scoring algorithm. This data is taken from our experiments with the Petstore 1.3, described in Chapter 5.

Here, the α coefficient allows us some degree of control over the ratio of false positives to true-positives. All other things being equal, we would expect to have less than $\frac{1}{\alpha}$ of our anomalous paths to be false positives when a failure occurs.

Alternatives

Before settling on our PCFG scoring function (Eq. 4.2), we also tried and discarded two scoring functions used in natural language processing: first, taking the product of the probabilities of all the transition rules and, second, taking their geometric mean. While the former had an unacceptable bias against long paths, the latter masked problems when only a small number of improbable transitions were exercised.

Together with Peter Bodík, we have used one-class support vector machines (SVM) to detect anomalies in the path shapes and component interactions we captured in our experiments. While SVM worked as well as our χ^2 test of goodness of fit for detecting anomalies in component behaviors, standard SVM techniques did not work well when analyzing path-shape behaviors. A core step in an SVM analysis is to compute the similarity score (a dot-product-like function called a kernel method) between two paths. However, in the case of path-shapes, a good anomaly detector must not only detect differences in a path, but also estimate how significant those

differences are. While our PCFG scoring method does this with its calculation of the deviation from expectation, the standard SVM tree-comparison techniques do not. An SVM analysis that allowed for feature-weighting might improve the SVM analysis of path shapes.

Formal derivation of an anomaly score

While the anomaly score (Eq. 4.2) is not well-grounded in a strict mathematical sense, a more formal derivation of an anomaly score rooted in decision theory (Eq. 4.3) gives us a similar (though not identical) scoring function.

$$\lambda = \frac{P(x|\text{SUCCESS})}{P(x|\text{FAILURE})} \tag{4.3}$$

To decide whether an observed path x is anomalous or not, we compare the probability of our observations occurring in the context of a successful request and in the context of a failing request. We set our decision-boundary at $\lambda = 1$, where we choose to believe the request succeeded if $\lambda < 1$, and we choose to believe the request failed if $\lambda > 1$.

Taking the log of Equation 4.3, we have:

$$\log \lambda = \log P(x|\text{FAILURE}) - \log P(x|\text{SUCCESS}) \tag{4.4}$$

We can directly calculate $\log P(x|\text{SUCCESS})$ from our probabilistic context free grammar. For convenience, we also replace $\log \lambda$ with λ', setting our decision boundary at $\lambda' = 0$. This gives us:

$$\lambda' = \log P(x|\text{FAILURE}) - \log PCFG(x) \tag{4.5}$$

What remains is to calculate $P(x|\text{FAILURE})$. Since we cannot make any statements about likely behavior, it seems reasonable to make an assumption of simple uniform behavior, such that:

$$P(x|\text{FAILURE}) = \sum_{\forall t_i \in x} \frac{1}{n_i} \tag{4.6}$$

where t_i ranges over all transitions in the path shape x being tested, and n_i is the number of possible behaviors at this transition point—or equivalently, n_i is the number of production rules in our PCFG where the left-hand side N^i equals the left-hand side of the transition t_i.

We can further refine this by smoothing the calculation of $P(x|\text{FAILURE})$ to account for the possibility of previously unseen behaviors occurring during a failure:

$$P(x|\text{FAILURE}) = \sum_{\forall t_i \in x} \frac{1}{n_i + 1} \tag{4.7}$$

This leads to the anomaly score:

$$\lambda' = \log \sum_{\forall t_i \in x} \frac{1}{n_i + 1} - \log PCFG(x) \tag{4.8}$$

Empirically, using the PCFG function to calculate the probability of a path has a bias against long paths. However, this anomaly score removes that bias by using as a comparison point the probability of a faulty long path occurring.

4.5 Monitoring non-structural observations

While most Internet services have a significant amount of observable structure that is ties to the functionality they provide, not all distributed systems share this property. When a system does not have an observable structure, its structural behavior is not a reflection of the functionality it provides, or its structure is just too simple or constant to provide useful insights, then we must monitor alternative behaviors to help detect failures in the system and its components.

One such system is a clustered hash table (CHT) [56, 70, 93]. A CHT acts as a single-key lookup data store, where keys and data are distributed across a cluster of machines. Clients of CHTs generally access data within the CHT by sending a single key query. While there are many implementations of CHTs, generally, one or more nodes of the CHT have part or all of the data matching the query. Nodes of the CHT can respond independently or coordinate their actions to serve the client request. Writes can be slightly more complicated, but generally involve either locating

all existing copies of a key and overwriting their values completely, or writing new copies of a key across the CHT and invalidating older versions.

These CHTs generally have a very simple structure and few, if any, complicated interactions between nodes to reflect high-level functionality. This is no different in the specific systems that we wish to monitor, session-state management store (SSM) and DStore. In these two systems, the primary interactions occur between a thin library residing at the client and individual nodes of the CHT system. In particular, no multi-hop interactions occur while serving a client request, and it is normal behavior for a CHT node to either send a message back to a client (if it has data) or to silently ignore the message (if it is too busy or does not have data). This means that monitoring these interactions for changes is not likely to lead to a good failure detector.

In this section, we look instead to monitoring for alternative symptoms of failure. We describe statistics that are easy to collect, and likely to be early indicators of the kinds of failures that we might expect will affect the primary purpose of a CHT, such as disk usage, memory usage, and message response activity.

While we present in this section the simple statistical techniques we use to analyze these non-structural observations, we should note that more sophisticated techniques do exist and may be more suitable in many applications. We discuss these simple techniques here and evaluate them in Section 5.9 as a proof-of-concept that statistical monitoring of non-structural behaviors can aid fault detection at times when structural behaviors may not.

4.5.1 Activity Statistics

Activity statistics, *e.g.,* the number of processed writes, represent the rate at which a node is performing some activity. Activity statistics are primarily a function of current workload. This means that, in a system with high load and well-functioning load balancers, activity statistics should be comparable across peer nodes. A node whose activity statistics are significantly different from its peers is likely to be functioning differently, and possibly failing.

If the workload on the system is known to be stable or only slowly-changing over time, we can also compare current activity statistics to short-term or long-term

historical activity rates.

Since clustered hash tables can be deployed on a relatively small number of nodes, we calculate the median absolute deviation of the activity statistics. This metric is robust to outliers even in small populations, and lets us identify deviant activity statistics with a low-false positive rate.

4.5.2 State Statistics

State statistics represent the size of some state, such as the length of a message queue, or the amount of memory being used by a node. These statistics can be more complex than activity rates, varying in periodic patterns as a result of past state, current workload, and periodically scheduled tasks within the node. For example, if the software on a node is written in Java, the amount of memory it is using tends to grow until the garbage collector is triggered to free memory; then, the pattern repeats. Unfortunately, we do not know *a priori* the period of this pattern—in fact, we cannot even assume a regular period.

To discover the patterns in the behavior of state statistics, we use the Tarzan algorithm for analyzing time series [78]. For each state statistic of a node, we keep an N-length history or time-series of the statistic. We discretize this time-series into a binary string. To discover anomalies, Tarzan counts the relative frequencies of all substrings shorter than k within these binary strings. If a node's discretized time-series has a surprisingly high or low frequency of some substring as compared to the other node's time series, we say that this statistic implies that the node is potentially faulty. This algorithm can be implemented in linear time and linear space, though even our simpler non-linear implementation provides sufficient performance to monitor a small number of nodes.

4.5.3 Combining statistics

Since we monitor several different statistics, we have the equivalent of several anomaly detectors: one anomaly detector for each monitored statistic. This leads to the obvious question of how to combine the results from these monitors. A simple strategy

is to decide that a node may be faulty if more than some threshold N of its statistics appear anomalous. If $N = 1$, then our monitor combination is very sensitive, marking a node as faulty if any of its statistics appear anomalous. This makes the combination very aggressive, but also prone to false positives. At the other extreme, setting $N = |total\#monitors|$ makes our combination very conservative, requiring all statistics to be anomalous before we will believe that the node may have failed. This setting makes our combined monitor more resilient to false positives, but will likely detect fewer failures.

In our experiments, described in Chapter 5, we empirically found that setting $N = 3$ provided a good balance between aggressive fault detection and protection from spurious alarms. However, this setting was not very sensitive. For example, we found that false positives rarely, if ever, caused more than 2 statistics to appear anomalous at any time; and that true failures generally caused 6 or more statistics to appear anomalous at once. This suggests that any setting between $N = 3$ and $N = 6$ would have provided similar performance.

More complex combination schemes may also be considered, such as the summing individual anomaly scores instead of their boolean output, weighting the influence of individual statistics, and attempting to filter out duplicate or correlated statistics from consideration.

4.6 Summary

This chapter described the application of statistical monitoring to detect failures in Internet services. In the context of complex Internet service environments, with their large workloads and rapidly changing systems, statistical monitoring has several important features:

- Operators and developers are not required to write either a model of correct behavior, or explicitly test changing application functionality. A model of probably correct behaviors is learned automatically through on-line observation of the Internet service. Learning these models automatically depends on the weak assumption that most of the system is working correctly most of the time,

and eases deployability and reduces maintenance requirements when monitoring rapidly changing, poorly understood systems.

- Under certain assumptions, structural behaviors can act as proxies for application functionality, allowing us to monitor for changes in functionality without understanding the application's semantics. The two behaviors we discuss in this chapter, component interactions and path shapes, are defined under the assumptions that component boundaries are defined at the granularity of application functionality, and that interactions between components occur to provide some high-level function.

- We can also extend statistical monitoring to non-structural behaviors in systems without significant structure. In these cases, we look for monitorable metrics that are likely to change in anticipated failure modes.

- Monitoring the online behavior of a system with a significant workload can help us substantially avoid the test coverage problem presented by high-level application test suites. The need for artificially generated and executed tests of functionality is removed by monitoring system behavior under the load of many users. These users are constantly exercising the important pieces of the service, and by tracking how their requests are serviced, we monitor exactly the components and interactions of the application that the users are depending upon. Moreover, our monitoring occurs in the most realistic environment possible, the real system itself, and allows us to notice changes in system behavior caused by the open-ended workload.

Chapter 5

Evaluating fault detection

In this chapter, we test our statistical monitoring approach to failure detection. We present experiments that test failure detection capabilities, failure detection time, and susceptibility to false alarms during normal day-to-day operations.

To test fault detection rates, we use our prototype statistical monitor, Pinpoint, to monitor the structural behaviors within a widely-used middleware framework (J2EE) for building Internet services [98]. We inject various faults and errors into applications running on top of this middleware, and evaluate how well Pinpoint detects and localizes the resultant failures.

We also connect Pinpoint to a clustered hashtable, and monitor its non-structural behaviors as we inject a number of failures. Finally, we have obtained log data captured during several serious failures at a large Internet service, and apply statistical monitoring techniques to validate their applicability to real-world failures.

5.1 Metrics

To measure how well our approach detects the failure of a system, we use the metrics of *recall* and *precision*, two metrics borrowed from information retrieval research (Figure 5.1). When searching for items belonging to some target population, recall measures the proportion of the target population that is correctly returned, and precision measures the proportion of returned items that actually match the target

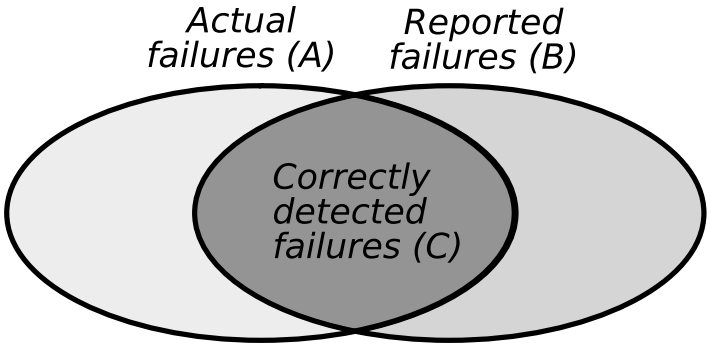


Figure 5.1: **Metrics of recall and precision.** Given a set of actual faults (A), and a set faults detected by some monitor (B), the correctly detected faults (C) will be the intersection of sets A and B. Recall can be calculated as $\frac{|C|}{|A|}$. Precision can be calculated as $\frac{|C|}{|B|}$.

population. In our context, perfect recall (recall=1) means that all faults are detected, and perfect precision (precision=1) means that no false alarms were raised.

Sometimes, we also refer to recall as the *fault detection rate*. We use the term *miss rate* to refer to the proportion of undetected faults, *e.g.,* $1 - \frac{|C|}{|A|}$.

Another common metric of fault detector efficacy is the false positive rate or false alarm rate. This metric measures how often a monitor mistakenly raises an alarm when no fault is actually occurring. In the context of Figure 5.1, the false alarm rate can be calculated as $\frac{|B|-|C|}{t_{period}}$, where t_{period} is the period over which we have observed the set of actual failures and reported failures. The related metric of the false negative rate, or miss rate, can be calculated as $\frac{|A|-|C|}{t_{period}}$. As measurements of false positive and false negative rates are highly dependent on the rates of failures, system changes and other events in a specific environment, we do not attempt to measure them. Instead, we focus on how well our approach detect different kinds of faults, irrespective of their rate of occurrence.

A final important metric is the time to detect a failure. Failure detection time is simply defined as the time between the first occurrence of a fault and the first report of its detection by our monitor. However, in an uncontrolled environment, this can be difficult to measure, since we often know only when we detected a fault, not when the

fault first occurred. In our experiments, we are able to measure fault detection time by using our first injection or triggering of a failure as the base point for calculating the time taken to detect the problem.

Later, in Chapter 7, we describe how to use the metrics of recall and precision in the context of fault localization.

5.2 Prototype

The core of Pinpoint is a plugin-based analysis engine with a simple dataflow oriented architecture. Data-flow architectures are a natural fit for analyzing streaming data, such as dynamic observations of system behavior. However, building efficient data-flow infrastructures that are well-suited for statistical analysis and machine learning algorithms is an area of on-going research [131]. We do not claim significant novelty in the architecture of our prototype, and simply describe our architecture for completeness.

In our system, we use *record collections* to represent sets of data, such as observations, intermediary results of an analysis, or the final output of analysis. Each record in a record collection wraps around an opaque Java object, allowing plugins to associate arbitrary metadata with each object.

Plugins are active code modules that (generally) operate on record collections. Plugins can read data from record collections, as well as write data to and remove data from record collections. Plugins can be triggered to run whenever a record collection changes, or can simply be run on a periodic basis.

While plugins and record collections can be dynamically created, the initial configuration of the analysis engine is defined using a simple XML-based configuration file. We have found that an important benefit of using a plugin-based analysis engine is the ease of experimenting with new analysis techniques and modifying implementation details by simply copying and modifying existing plugins and configuration. This allows us to experiment with new analysis code while still enabling easy code reuse and leaving original plugins and configurations untouched and in working condition.

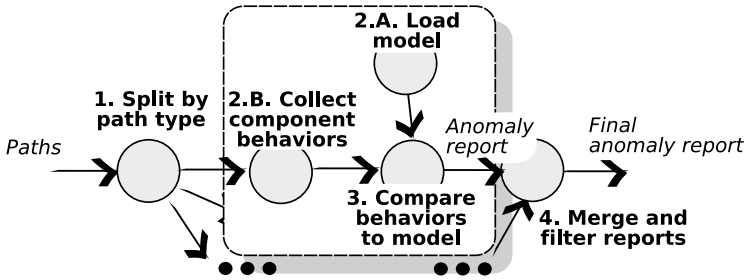


Figure 5.2: **Analysis pipeline for anomaly detection in component interactions.**

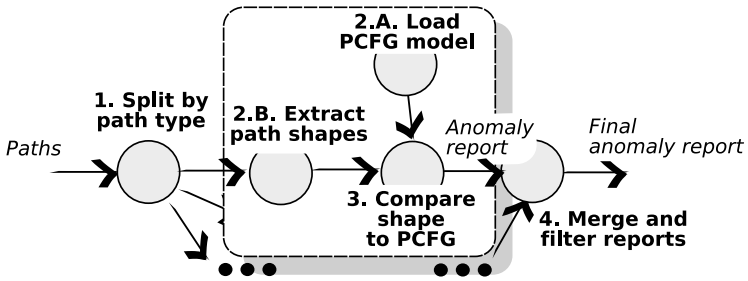


Figure 5.3: **Analysis pipeline for anomaly detection in path shapes.**

The analysis pipelines for generating models of likely correct behavior and detecting failures in component interactions is shown in Figure 5.3. Figure 5.2 shows the analysis pipeline for path shapes. For simplicity of presentation, some minor details are removed,such as helper plugins for debugging and various stages of reconstructing runtime paths from observations.

In both analysis pipelines, we see the same basic stages of a) binning the incoming runtime paths by type; b) loading a model for each bin; c) extracting a type of structural behavior from the paths; d) comparing the extracted behavior to each model to generate an anomaly report per bin; and e) merging the anomaly reports from all bins together into a single final report.

Most of these stages can be parallelized for scalability. When analyzing path shapes, the processing of incoming paths, extraction of path shapes and comparison of a path shape to a model are easily parallelizable as each incoming request can be analyzed separately. To parallelize the analysis of component interaction models, we

have to separate the extraction of component interactions from the collection of these interactions into a component behavior model. The analysis step that is most difficult to parallelize is the building of our path shape and component interaction models, since building a model fundamentally centralizes observations from across our system into a single model. However, we can still parallelize the building of the different models for each of our behaviors and bins of runtime paths. Also, since our models are based on the majority behavior of the site, our model-building should tolerate heavy sampling, if necessary, to aid in the scalability of our model generation.

5.3 Testbed Internet service

In this section, we describe the setup of our testbed platform. We have instrumented a popular middleware platform to gather the behaviors we observe, deployed several applications atop our platform, and injected a variety of faults and errors to test the detection capability of Pinpoint. Though our testbed is not perfect, notably because of its small size, we have attempted to make it as realistic as possible.

5.3.1 Instrumentation

We have instrumented the JBoss version 3.2.1 open-source implementation of the J2EE middleware standard which provides a standard runtime environment for three-tier enterprise applications [74, 98], as described in Section 2.1.3. JBoss is widely used: it has been downloaded from SourceForge several million times, was awarded the JavaWorld 2002 Editor's Choice Award over several commercial competitors, and according to one study [11], offers better performance than several competitors as well. More than 100 corporations, including WorldCom and Dow Jones, are using JBoss as an important piece of their computing infrastructure. According to a 2004 survey, JBoss is in wider commercial deployments than competing application servers [21].

In the *presentation* or web server tier, our instrumentation captures the URL and other details of an incoming HTTP request, and also collects the invocations and returns for used JSP pages, JSP tags, and servlets. In the *application* tier, which manages and runs the Enterprise Java Bean modules that make up the core of the

application, we capture calls to the naming directory (used by components to find each other) and the invocation and return data for each call to an EJB. Finally, we capture all the SQL queries sent to the *database* tier by instrumenting JBoss's Java Database Connection (JDBC) wrappers.

When a client request first arrives at the service, the request tracing subsystem is responsible for assigning the request to a unique ID and tracking it as it travels through the system. To avoid forcing extra complexity and excessive load on the components being traced, the tracing subsystem simply reports whenever an important event occurs, such as a request's entry into or exit from a component. These events are later collated into a complete history of the request's path through the system.

We track unique request IDs with a requests computational processing by storing the ID in a thread-specific local variable during local computations. Whenever a request is about to cross a component or machine boundary, *e.g.,* via a remote method invocation (RMI), we retrieve the current unique ID from this thread-specific variable and pass it along with the request. With the assumption that components do not spawn new threads, and that all the component entry and exit points are instrumented, this scheme allows us to correctly associate a single unique ID with all the processing that occurs on its behalf. In a system where components spawned their own threads, we would likely have had to instrument the thread creation classes as well.

Whenever we observe an event, we capture six pieces of information: (1) a unique ID identifying the end-user request that triggered this action; (2) an event number, used to order events within a request; (3) whether the observed event is a component call or return; (4) a description of the component being used in the event (*e.g.,* software name, IP address, etc.); (5) timestamp; (6) any event-specific details, such as the SQL query string. These observations are reported asynchronously across the service, and gathered at a central logging and analysis machine. We use the unique request IDs and event numbers to recreate the path that each request took through the system. A sample observation record is shown in Figure 5.5.

Minor refinements significantly improve the performance of this instrumentation

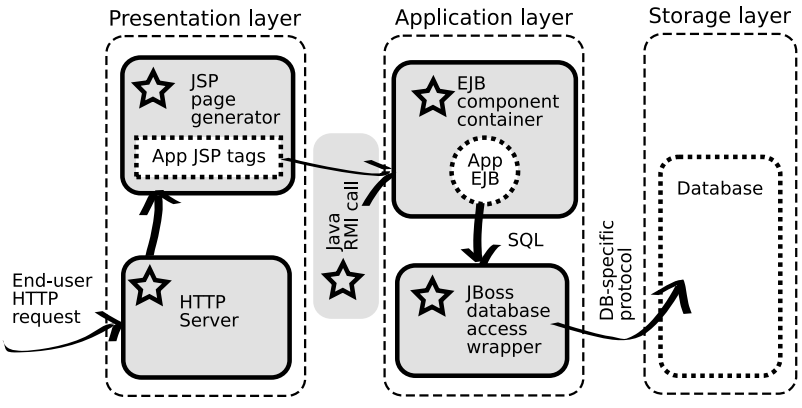


Figure 5.4: **JBoss instrumentation points.** This figure shows the typical flow of a user request through the pieces of our instrumented JBoss testbed. Stars and gray backgrounds mark middleware components and protocols that we have modified for our instrumentation framework. Our instrumentation does not require any changes to applications, including presentation layer JSP tags and application-layer EJB components, or the storage layer database. Not shown in the figure are our instrumentation points for the Java messaging service (JMS) and the Java naming server (JNDI).

scheme. First, we modify our instrumentation to only periodically report static information, including component descriptions and some event-specific details, and modify the logging machine to cache this static information. In addition, minor modifications would be necessary to correctly handle reporting of asynchronous requests.[1].

Our instrumentation is spread across several sites within the JBoss code. Each site instruments one type of component (EJB, servlet, etc.) and required changes to only 1–3 files. Each instrumentation site required 1–2 days of graduate student time to implement and debug. In our instrumentation, we make observations asynchronously, and drop observations rather than hurt the performance of the system. Table 5.1 summarizes the code changes we made to JBoss.

Using asynchronous reporting of observations on a single-node system, our unoptimized implementation can maintain full instrumentation without dropping observations at a steady rate of 75 clients, and can tolerate bursts of up to 300 clients

[1]Instead of reporting a ⟨*unique ID, counter*⟩ tuple to identify each observation, we would have to report a triple of ⟨*unique ID, asynchronous branch ID, counter*⟩ Whenever an asynchronous message is sent a new, unique asynchronous ID would have to be created. Without reporting this triple of information, it would be difficult or impossible to order observations of an asynchronous request.

```
[Observation: [
       eventType = component use
       requestId = 1062313694989_1532882574
       sequenceNum = 0
       originTimestamp = 1062313694989
       collectedTimestamp = 1062313718254
       originInfo = {java.vm.vendor=Sun Microsystems Inc.,
                     os.name=Linux,
                     type=servlet,
                     program.name=run.sh,
                     os.version=2.4.20-perfctr,
                     java.class.version=48.0,
                     java.vm.info=mixed mode,
                     jboss.server.name=petstore-backend-1,
                     os.arch=i386,
                     java.vm.version=1.4.0-b92,
                     ipaddress=169.229.50.206,
                     java.specification.version=1.4,
                     java.vm.specification.version=1.0,
                     hostname=x6.Millennium.Berkeley.EDU,
                     name=class org.mortbay.jetty.servlet.Default}
       rawDetails = {}
       attributes = {observationLocation=org.mortbay...ServletHolder,
                     stage=METHODCALLBEGIN}
```

Figure 5.5: **A sample observation for a call into the JBoss ServletHolder container for J2EE servlets.** Servlet holders only have a single invoke() method, so this observation point does not bother to report the method name. This observation has already been merged with the cached static details about the component and its environment.

Subsystem	**Description**	**Modified files**	**Added Lines of code**	**Notes**
HTTP server	Assigns unique IDs to incoming requests	1	54	Also logs IDs and HTTP status for visibility
Servlets	Records servlet entry and exit	1	23	
Database wrapper	Reports SQL statements as they are executed	3	243	
Naming server	Reports name registrations, directory lookups and results	2	135	
Remote method invocation	Transmits request ID along with remote call	3	88	
Java messaging service	Reports use of asynchronous messaging service and assigns unique IDs to asynchronous requests	3	42	
EJB components	Reports EJB entry and exit	1	255	Implemented as a plugin interceptor to JBoss
JSP tags	Reports JSP tag usage	3	66	

Table 5.1: **JBoss instrumentation details.** Details of the minimal impact of our JBoss instrumentation. In addition to our instrumentation points, we also wrote a 1200 line helper library used across most instrumentation points. All together, instrumenting the 56k-line JBoss system required only 2100 additional lines of code, much of it error checking and braces.

for minutes at a time (with subsequent queuing delays in reporting observations). Using synchronous reporting, our observations are not subject to queuing delays, but does increase request latency by $2 - 40$ ms depending on the length of the path, and degrades system throughput by 17%, from 25 to 29 requests/sec on our baseline testbed.

The majority of our performance costs come from our use of Java serialization as the basis of our network transfer protocol. However, the deployment of commercial instrumentation packages such as Integritea on large sites such as Priceline.com suggests that this relatively coarse-granularity of instrumentation is practical if some engineering effort is focused on the implementation [123].

5.3.2 Applications and workload

We deployed three different applications in our testbed platform:

- **Petstore 1.1** is Sun's sample J2EE application, which simulates an e-commerce web site (storefront, shopping cart, purchase tracking, etc.). It consists of 12 application components (EJBs and servlets), 233 Java files, approximately 11k lines of code, and stores its data in a Cloudscape database. The primary advantage of this application is that we have been able to modify the original application to distribute its presentation and business logic across a cluster of machines.

- **Petstore 1.3** is a significantly rearchitected version of Sun's initial application. This version includes a suite of applications for order-processing and supply chain management. It consists of 47 application components, 310 files, 10k lines of code, and also stores its data in a Cloudscape database. Because of its new architecture, we were unable to cluster Petstore 1.3, though we do run the application, database and analysis engine on separate machines. However, its increased functionality makes it an interesting application in our testbed.

- **RUBiS** is an auction web site, developed at Rice University for experimenting with different design patterns for J2EE [26]. The complete RUBiS package

contains over 500 Java files and over 25k lines of code. More relevant for our purposes, RUBiS has 21 EJB components and several servlets. RUBiS stores its product catalog, user database and transaction information in a MySQL database. The RUBiS application is not easily clustered, though, like Petstore 1.3, we run the application, database and analysis engine on separate machines.

RUBiS comes with its own load generator that simulates a user browsing the auction web site with a state transition matrix. This transition matrix represents the probability of moving from any one interaction (*e.g.,* searching for an item) to another (*e.g.,* purchasing an item). Each load generator can simulate a variable number of clients, with each client running in its own thread within the load generator. Between requests, clients pause for a randomized *think time*, generated from a negative exponential distribution borrowed from the TPC-W benchmark [125]. The load placed on the testbed by RUBiS's load generator is controlled by varying the number of simulated clients.

We built our own HTTP load generator for our Petstore applications. The Petstore workloads presented by our load generator simulates traces of several parallel, distinct user sessions, with each session running in its own client thread. A session consists of a user entering a site, performing various operations such as browsing or purchasing items, and then leaving the site. We choose session traces such that the overall workload fully exercises the components and functionality of the site. As in RUBiS's load generator, each client pauses for a think time between requests, and load is controlled by varying the number of simulated clients.

In our load generator, if a client thread detects an HTTP error, it retries the request. If the request continues to return errors, the client quits the trace and begins the session again. Our load generator will also reset its session if it finds that the next URL being requested is not reachable from the current page; this can happen, for example, if an error causes part of the service to fail. Our recorded traces are designed to take different routes through the web service, such that a failure in a single part of the service will not artificially block all the sessions early in their life cycle.

Note that the errors that affect the playback of traces are an easily detectable subset of the failures that we are trying to detect with our fault monitor. Many failures will cause the load generator to stop a trace several requests after the failure originally occurs. Also, many failures are only noticeable by the load generator because it has knowledge through the session traces of the URLs which should be appearing in the server's returned HTML. Of course, changing the server application would likely require updating our load generator session traces as well.

While our synthetic workload is very regular, our discussions with multiple large Internet services indicate that this regularity is realistic. One site reported that their aggregate user behavior at any time is generally within 1-2% of the behavior at the same time in the previous week, with the exception of major events such as holidays or disasters. Also, while we would like to validate that the scale of the workload in our testbed environment is comparable to the workload per machine in a typical Internet services, we have found that such request rate information is considered to be highly confidential, as it is related to the customer base of an Internet service, and are unfortunately unable to publish a comparison.

Unless otherwise indicated, when we deploy these applications, we run our observation collector, the application, the database and the load generator each on separate machines. Our clustered version of Petstore 1.1 runs with one front-end node and three middle-tier nodes.

5.3.3 Fault and error load

The stated goal of Pinpoint is to detect application-level failures, as described in Section 2.3. Since we detect failures by looking for changes in application behavior, we have tried to choose fault and error loads which will cause a variety of different reactions in an application.[2] We believe one of the primary factors that determines how an application will react to a problem is whether the system was designed to

[2]As presented in Section 2.2, by generally accepted definition [4], *failures* occur when a service deviates from its correct behavior, for some definition of correctness. An *error* is the corrupt system state that directly caused the failure. A *fault* is the underlying cause of the system corruption. In our experiments, we both inject faults (such as source code bugs) and errors (such as directly through Java exceptions). In the rest of this thesis, we use the term "fault injection" to include both fault and error injection.

handle the failure or not. Thus, our injected faults and errors include those that a programmer building a system should expect, might expect, and likely would not expect.

To inform our choice in fault and error loads, we surveyed the studies of system failures discussed in Chapter 2, as well as the faults injected by other researchers in their experiments [34,66,104]. While some experiments focus on a range of byzantine faults, we found that most the faults injected concentrated on problems that caused program crashes and other obvious failures, as opposed to triggering only application-level failures.

The faults and errors we inject are:

Java exceptions: Because Java coerces many different kinds of failures, from I/O errors to programmer errors, to manifest as exceptions, injecting exceptions is an appropriate method of testing an application's response to real faults. To test an application's response to both anticipated and possibly unexpected faults, we inject both exceptions that are declared in component interfaces and undeclared runtime exceptions. Note that both kinds of exceptions can sometimes be normal and other times be signs of serious failures.

Naming directory corruption: To simulate some kinds of configuration errors, such as mislabeled components in a deployment descriptor, we selectively delete entries from the Java Naming Directory server (JNDI). The effect of this error is that future lookup requests for the deleted entries will fail, as if the lookup request, component's deployment information, or naming server was misconfigured.

Omission errors: To inject this error, we intercept a method call and simply omit it. If the function should have returned a value, we return 0 or a null value. While omission errors are not the most realistic of the failures we inject, they do have similarities to some logic bugs that would cause components to not call others; and to failures that cause message drops or rejections. Moreover, omission errors are unexpected errors, and how well Pinpoint detects these

problems may give us insights into how well other unexpected faults will be detected.

Overload: To simulate failures due to the system overloads of a flash crowd or peak loads, we adjusted our load generators to overload our testbed. In our system, the first signs of an overload came at the database tier, with database queries timing out.

Source code bug injection: Even simple programming bugs remain uncaught and cause problems in real software [114, 120], so introducing them can be a useful method of simulating faults due to software bugs [34]. We injected these bugs into the Petstore 1.3.1 application using an automated bug injector that we wrote for the Java language. We use the Polyglot extensible compiler framework [106] as a base for a Java-to-Java compiler that can inject several different simple bugs, summarized in Table 5.2. While these are all minor bugs, evidence suggests that no bug is so trivial that it does not occur in real software [65]. Some of the more common (but much more involved) source code bugs we did not inject include synchronization and deadlock issues and subtle API incompatibilities.

For our experiments, we first generate an exhaustive list of the spots where a bug can be injected within a component, after eliminating locations in unused code. Then, we iterate over these "bug spots" and inject one bug per run of the application. At runtime, we record when this modified code is exercised to track what requests may be tainted by the fault. Section 5.4.1 describes the effect of some of these bugs.

Empirically, we have found that several kinds of low-level hardware and OS faults, such as memory corruptions, CPU register bit-flips, and I/O errors do not often manifest as application-level failures that would otherwise go unnoticed [23, 93], and therefore we do not inject these classes of faults in our experiments. These faults usually either have no visible effect on system operation, or cause easily visible process crashes or machine hangs. While bit-flips and memory problems can cause application data corruption, as reported in [50], simple explicit checks, such as checksums, are

Bug Type	Description	Code changes	Num
Loop errors	Inverts loop conditions;	Good: loop{ while(b){**stmt;**} Bad: loop{ while(!b){**stmt;**}	15
Mis-assignment	Replaces the left-hand-side of an assignment with a different variable	Good: $\mathbf{i} = f(x)$; Bad: $\mathbf{j} = f(x)$;	1
Mis-initialization	Clear a variable initialization.	Good: int $i = \mathbf{20}$; Bad: int $i = \mathbf{0}$;	2
Mis-reference	Replaces variables in expressions with a different but correctly typed variable	Good: $Avail = InStock -$ **Ordered**; Bad: $Avail = InStock -$ **OnOrder**;	6
Off-by-one	*E.g.,* replaces < with <= or >= with >	Good: for($i = 0$; i<$count$; i++) Bad: for($i = 0$; i<=$count$; i++)	17

Table 5.2: **Overview of injected source code bug types and injections.** None of these bugs are detected by the Java compiler.

Figure 5.6: **A masked failure in Petstore 1.1.** During an Inventory database failure, Petstore 1.1 attempts to mask the problem, showing items as back-ordered, instead of displaying an error

more likely to reliably detect these corruptions. This is feasible because, when the data is first created, there exists a known good state of the memory, and a checksum can be calculated. Unfortunately, this known good state does not exist for application behavior in general.

We expect that exceptions and omissions are extremely likely to affect the structural behavior of an application, while source code bugs may or may not cause changes in the application structure. By injecting this range of faults, we test both whether our algorithms detect anomalies when the application's structural behavior changes, and whether more subtle faults that cause user-visible errors are also likely to change the application's structural behavior.

Together these injected faults cause a variety of errors in our testbed. As an example of a mild failure, faults injected into the InventoryEJB component of Petstore 1.1 are masked by the application, such that the only user-visible effect is that items are perpetually "out of stock" (see Figure 5.6).[3] At the other end of the spectrum, injecting an exception into the ShoppingController component in Petstore 1.3 prevents the user from seeing the website at all, and instead displays an internal server error for all requests (see Figure 5.7).

The final part of our fault injection methodology is the verification that the faults

[3]The InventoryEJB component manages the descriptions of items and their availability in the store.

Figure 5.7: **An unmasked failure in Petstore 1.3.** Petstore 1.3 does attempt to mask some serious failures, such as a problem in the ShoppingController. Any attempt to access the site results in this obvious error.

we inject actually cause user-visible errors. We do this in our experiments by modifying our load generator to verify all the HTML output of an application with MD5 hashes from fault-free runs. To make this verification feasible, we force our dynamic applications to produce mostly deterministic output by resetting all application state between experiments and by running a deterministic workload. In addition, we write special-case filters in the load generator to canonicalize any remaining non-deterministic output, such as randomized order numbers, before hashing. When the HTML returned by the service fails its MD5 check, this failure is recorded in the load generator's logs, and the load generator attempts to continue with the session.

Even though we are in control of the fault injection, it is not trivial to determine which client requests in an experiment are actually failing. Component interactions and corrupted state or resources can all lead to cascading failures that cause requests to fail even when we do not explicitly inject faults into them. As our ground-truth comparison, we mark a request as having failed if (1) we directly inject a fault into it; (2) the request causes an HTTP error; or (3) the returned HTML document fails our MD5 hash.

For the majority of our experiments, we collected application traces from Java exception injections, omission faults, and source code bug injection experiments. Due to the time it takes to run these experiments, we collect these traces once, then analyze them off-line. In addition, we injected JNDI corruption and overload failures, and analyzed the failures in real-time.

In practice, we expect Pinpoint to be deployed as a real-time monitor. We believe real-time monitoring using statistical techniques is feasible since the bottleneck of our

analysis system is not the statistical techniques. When running on-line experiments, such as those described later in Chapter 8, the bottleneck of our analysis is our unoptimized instrumentation of JBoss. If we take the instrumentation bottleneck out of the critical path of our analysis by analyzing recorded observations of behavior off-line, we are able to process the 5 minutes of data captured in our experiments in less than a minute with even our unoptimized Java monitor, indicating that real-time monitoring with statistical techniques is feasible.

No fault injection scheme can accurately mimic the variety and effects of failures that occur in the real world. However, given the number and breadth of failures we have injected into our applications and our use of an enterprise-ready middleware software as the base of our testbed, we have confidence that the application's behavior following a failure realistic, even though the fault itself is artificially caused.

5.3.4 Comparison monitors

To evaluate Pinpoint's ability to detect failures, we implemented several types of monitors for comparison. Here we describe our implementation and experience with various kinds of low-level and application-specific monitors. Since we do not have real users generating workload on our site, we do not include business-level metrics monitors to watch for changes in user behavior.

HTTP monitoring

Our own load generator doubles as a straightforward HTTP monitor, recording whenever any of its requests generates an HTTP error, such as an HTTP 500 response, an indication of an internal server error. At Internet services, HTTP return codes are readily available through internal web server logs. In addition, third-party monitor such as those provided by Keynote Systems [80], can monitor the status of HTTP requests they initiate.

To ease the comparison of our HTTP monitor's fault detection to Pinpoint's, we modified our Internet service to return each request's unique ID to the load generator as an extra HTTP header. This allows our load generator to record the unique IDs

of the requests it believes have failed, and compare that list directly to Pinpoint's results.

HTML monitoring

Our load generator also acts as an HTML monitor, searching for obvious signs of failures within the HTML document returned by a web application. Specifically, our simple HTML monitor scans for the appearances of the keywords "error" and "exception" in the HTML text. This is designed to detect explicit error messages displayed by an application, as well as Java exceptions being propagated to the HTML of a document. In our applications, this monitor does not generate any false positives. In general, however, such a simple monitor would detect false positives when the words "error" or "exception" occurred naturally on a page, such as when a bookstore was selling a book about failures, or a discussion forum for users asking for help solving their problems.

Exception monitoring

To compare Pinpoint to a simple Java exception monitor, we modified the Java runtime classes to detect when exceptions were created. Whether an exception is considered to be a failure depends on both the kind of exception and where in the program it manifests. *E.g.,* an end-of-file exception is normal at the expected end of a file, but a failure condition in the middle of the file.

Though we expected that some exceptions would be raised during normal operation of the system, we were surprised by the degree to which this is true. We found that even a fault-free run of Petstore 1.3.1 on JBoss generates over 27k Java exceptions during startup, and another 35k Java exceptions under client-load for 5 minutes. We analyzed these exceptions and found that no one type of exception (declared, runtime, core java.lang exceptions, application exceptions, etc.) accounted for these apparently "acceptable" exceptions (see Table 5.3). Furthermore, many of the exceptions that were thrown were the same kind of exceptions that are thrown during real faults, *e.g.,* we saw over 10k `ClassNotFoundExceptions`. This variety of exceptions comes from many benign causes: `FileNotFoundExceptions` occur while

Class	Count	Num. of types
java.∗(other than java.lang)	14154 (52%)	11
java.lang.∗	12083 (44%)	16
javax.∗	809 (3%)	8
org.apache.∗	116 (0%)	2
org.jboss.∗	60 (0%)	6
other	28 (0%)	2
Total	**27250 (100%)**	**45**

(a) Number of normal observed exceptions, categorized by Java class

Description	Count
Declared	20015 (73%)
Runtime	7203 (26%)
Throwables, errors, and unknown	32 (0%)

(b) Number of normal observed exceptions, categorized as declared or runtime exceptions

Table 5.3: **Summary of exceptions observed during a normal run of JBoss.** We found a large variety of exceptions occurring during normal operation, with no simple classification to distinguish between "acceptable" and exceptions that occur during true failures. For example, `java.lang.NullPointerExceptions`, a common exception during true errors, occurred 575 times during normal operation.

searching for optional configuration files; and `ClassNotFoundExceptions` occur when searching for a Java class through a chain of class loaders—if the class is not found in one class loader's context, the system searches for it with the next class loader. Because of the quantity and variety of benign exceptions, however, we concluded that building an application-generic exception monitor was not feasible for this class of system.

Log monitoring

Log monitoring is a common error detection mechanism in deployed systems. Though not as involved as application-specific test suites, log monitors are still system- and application-specific, usually requiring operators to write regular expression searches

to match potential errors in server log files. We wrote a simple log monitor, searching for "ERROR" messages, in our testbed's application log, and found that it detected "failures" in almost all our experiments, including false alarms in all of our fault-free (manually validated) control experiments. After some study, we concluded that distinguishing these false alarms from the true failures was non-trivial, and disregarded these log monitoring results from our comparison.

Other monitors

Since we are purposefully injecting non-fail-stop faults, we did not implement a low-level ping or heartbeat monitor. As none of our injected faults, including our overload experiments, caused our servers to crash or hang, we can assume that these ping and heartbeat monitors would not have noticed any of our injected faults.

Nor do we include application-specific test suites in our comparison, since deciding what application functionality to test would have been the determining factor in detecting many of these failures, as a test suite can be engineered to detect almost any expected failure. Additionally, Pinpoint's main improvement in comparison to application-level monitors is not strictly its ability to detect failures, but it's ability to detect failures using low-maintenance, application-generic techniques.

5.4 Fault detection rate

Our fault detection experiments measure the recall of our Pinpoint fault detector across the various kinds of faults we inject, described in Section 5.3.3. The results of our systematic fault detection experiments are summarized in Figure 5.8. Both path-shape analysis and component interaction analysis performed well, detecting 70-90% of most of the different kinds of faults we injected, as compared to a 25-50% fault detection rate for our comparison monitors. The one exception is source code bug injections, which were much harder to detect by all our Pinpoint monitors and comparison monitors. We discuss source code bugs in detail in Section 5.4.1.

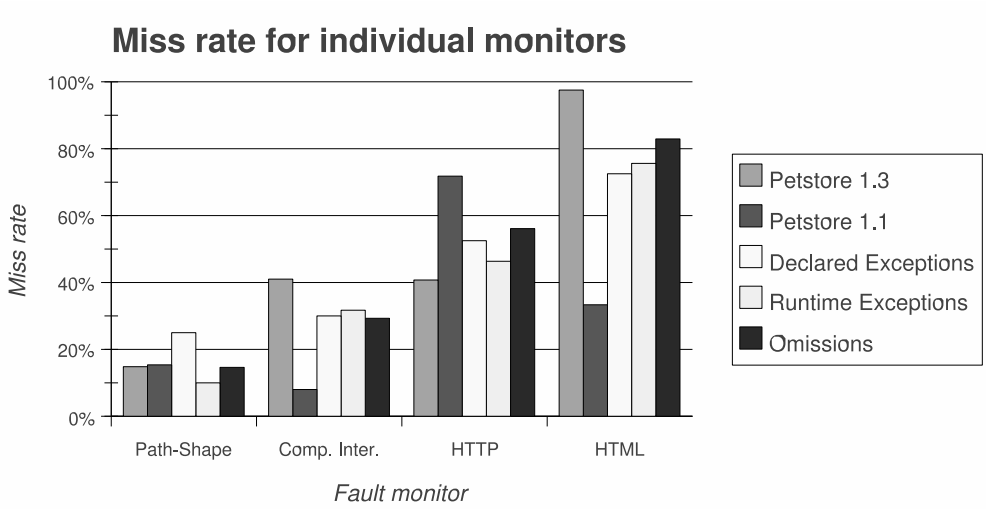
In addition to the fault detection experiments summarized in Figure 5.8, we ran several experiments injecting naming server corruptions while running the RUBiS

application, as well as overloading our testbed system with more clients than it could handle. In these tests, Pinpoint correctly detected each of these types of faults. In the case of the overloaded fault, our testbed database would usually saturate before the rest of the system, causing exceptions in the middleware (timeouts, etc.) when attempting to contact the database. Pinpoint correctly noticed these anomalies, usually discovering anomalous behavior in the database table "component" or the entity beans responsible for mediating application communication with the database.

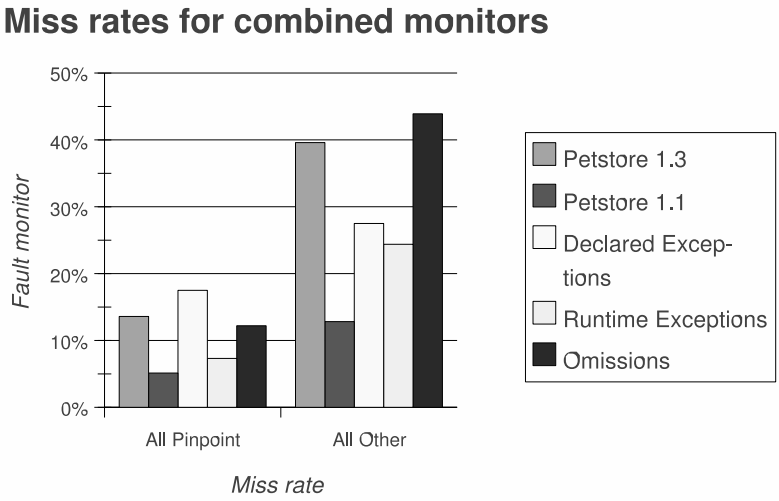
Studying these results, we find that the detection rates for Pinpoint's analyses present significant improvements over the other monitors, reducing the miss rate by 30-70%. The combined Pinpoint monitors are strictly better than the combined comparison monitors. That is, no fault detected by either of the comparison monitors is missed by the combined Pinpoint monitor.

To better understand what kinds of failures Pinpoint detected and did not detect, we looked at several factors that might be affecting Pinpoint's detection capability, including the type of fault injected, the severity of the failure, and various aspects of the components into which we injected the failure. We found one primary factor for component interactions was the number of requests affected by the fault: major failures, where more than 1% of requests were affected, were much more likely to be detected than minor failures. This explains why component interaction analysis did better detecting failures in Petstore 1.1 as compared to Petstore 1.3. The latter application splits its functionality across more components, many of which are not used frequently. While our analyses do not do as well detecting these failures, it is possible that at least some of these effects are an artifact of our experimental setup. With only $\approx$ 1000 requests per experiment, a minor failure affects fewer than 10 requests, and may not be noticeable by the dynamic thresholding algorithm of our path shape analysis, or statistically significant for our χ^2 test of goodness of fit in our component interaction analysis. The effect of this limitation in on-line system would be to increase fault detection times for failures in rarely exercised components, as the monitor would have to wait for more requests to arrive before declaring an anomaly with statistical confidence.

Other factors that reduced the effectiveness of component interaction analysis



(a) Individual monitors



(b) Combined monitors

Figure 5.8: **Comparing fault detection miss rates.** The miss rate of Pinpoint's path shape and component interaction monitors is significantly better than the comparison monitors. Combined, Pinpoint's monitors are strictly better than the combined comparison monitors.

included faults in components with little structural behavior, *e.g.,* components that never call other components. Similarly, path shape analysis did poorly when failures occurred toward the end of a path, and therefore did not affect the path shape.

5.4.1 Source code bugs

Here, we delve into detail on the effects of the source code bugs we injected into the Petstore 1.3.1 application, describing in more detail their impact on the system, and Pinpoint's performance in detecting them.

Most bugs caused relatively minor problems (such as an extra "next page" button, where no next page exists) rather than major problems that would keep a user from browsing the site and purchasing products. Overall, only a third of the bugs we injected kept any user session from completing. Of the bugs that did keep sessions from completing, almost all affected less than 50 sessions during an experiment. Only one bug injection in the shopping cart code was more serious, causing all sessions (over 400) to fail.

After running these experiments, we found that path-shape analysis and component interaction analysis did not do significantly better at detecting source code bugs than the HTTP monitors:

Monitor	Bugs detected (of 41)	% detected
HTTP	4	10%
HTML	1	2%
Path shape ($\alpha = 2$)	5	12%
Path shape ($\alpha = 4$)	3	7%
Path shape ($\alpha = 8$)	3	7%
Component interactions	5	12%

Upon inspection, the reason was that most of the software bugs that were injected, even though they caused user visible changes to the output of the application, did not cause major functional changes in the application, nor cause component-level changes internally. This indicates that analyzing component interactions (either directly or through path-shape analysis) is not likely to detect this class of simple source code

bugs, and it may be fruitful to analyze other system behaviors as well.

Pinpoint's analyses were able to detect bugs when they did change a component's interactions, however. For example, the bug with the most impact on user sessions kept the shopping cart from iterating correctly over its contents. While this behavior is internal to the shopping cart and not noticeable by itself, it had a secondary effect that JSP tags in the presentation layer no longer had items to display and so stopped iterating as well. As the iteration of JSP tags is visible by our instrumentation, Pinpoint did notice this change, and reported that the JSP tags related to displaying cart items were behaving anomalously.

5.5 Faulty request detection rate

Once we have detected a failure in the system, it can be important to estimate the impact of the failure; that is, what kinds of requests and how many of them, are failing. Determining which specific requests are failing gives us the ability to judge the significance of the failure and possibly help narrow down a cause. Our request-oriented path-shape analysis, as well as the HTTP and HTML monitors, can help do this. Other monitors, including component-interaction analysis, many of low-level heartbeat monitors and business-level metrics cannot directly identify individual failing requests.

We evaluate failing request identification separately from failure detection because, though dependent, they are separable issues. In particular, we can potentially detect a fault in a system without identifying all (or even most) of the failing requests.

We apply the same general metrics of recall and precision which we introduced earlier, except now we apply them to measure how well Pinpoint identifies faulty requests given that a fault exists. In this context, perfect recall means that we have identified all the true failing requests; and perfect precision means that we have not mistakenly identified any successful requests as faulty.

Unlike our fault detection evaluation Section 5.4, we can measure both the precision and recall of identifying failing requests, since we have a variety of both failing and successful requests in each of our experiments.

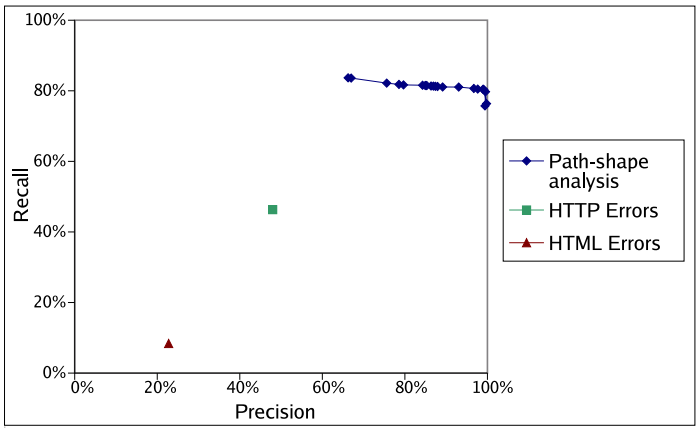
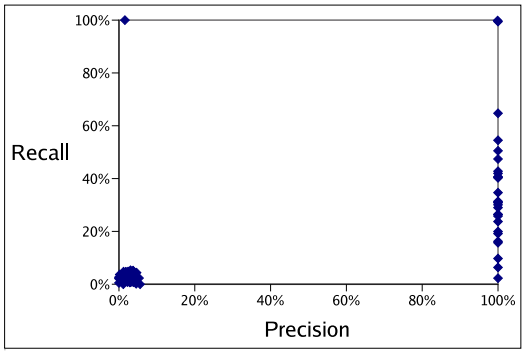


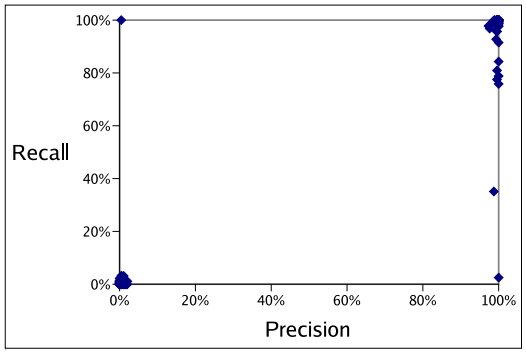
Figure 5.9: **The precision and recall of discovering faulty requests with path-shape analysis as we vary α sensitivity.** Here, we show how our sensitivity threshold affects the number of requests correctly and incorrectly labeled across all our Petstore 1.1 and Petstore 1.3 fault injection experiments. As expected, there is a trade-off between high precision and high recall, but we also see that sensitivity strongly affects precision, but has a less pronounced effect on recall. For comparision, we also mark the precision and recall of our HTTP and HTML error monitors.

In Figure 5.9, we investigate how adjusting the α parameter affects the recall and precision of our path-shape analysis. We see that our sensitivity threshold does allow us some control over the precision and recall over the precision and recall of our faulty request detector. While this figure shows the average recall and precision across all of our fault injection experiments, it is worth noting that there is a heavy bimodal distribution of detected and undetected failures in individual experiments. That is, precision and recall in an individual experiment is either significantly better or significantly worse than the average shown here, as illustrated in Figure 5.10.

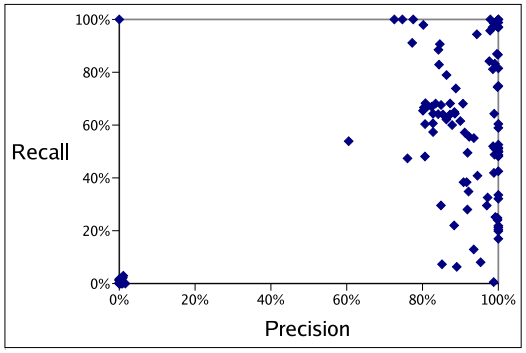
Overall, we found our path-shape analysis does a good job of detecting faulty requests without detecting false positives. It is worth noting that the faulty request identification precision and recall values in this section are during anomalous periods. Because of our dynamic thresholding, we can catch most faulty requests during these times, (even if their individual PCFG scores are otherwise acceptable), and avoid detecting false positives when the system is behaving normally.



(a) HTML monitor



(b) HTTP monitor



(c) Path-shape monitor, $\alpha = 4$

Figure 5.10: **Distribution of recall and precision in faulty request detection.** The distribution of recall and precision across experiments is strongly bimodal. In some fault injection experiments, recall and precision are both zero, with no fault being detected. In others, recall and precision are both high. We have added a small degree of random jitter to our data points to show the density of their distribution. Our path shape analysis monitor has a lower number of fault injection experiments where no fault is detected is lower, as compared to HTTP and HTML monitoring, though the latter have better recall and precision when they do detect a failure.

5.6 Fault detection time

Another measurement of the efficacy of a monitor is how quickly it detects failures. Machine crashes are usually detected by low-level monitors within seconds, while higher-level failures can take longer to detect. When running application-level monitors, how often a test suite is run usually determines how long it will take them to detect failures. Business-metric monitors take many minutes to notice a failure, as problems that affect business metrics can take some time to affect these indicators. The most comprehensive fault monitor, customer service complaints, can take hours to days to report failures, depending in the severity of the failure and how well a customer service department is integrated with the technical staff of the service. Pinpoint's goal is to detect the higher-level failures within the time scales of low-level monitors.

To test the time to detection, we monitor the RUBiS application in real-time, and arbitrarily picked one component, SBAuth, into which we inject failures.[4] We measure the time it takes Pinpoint to detect a failure by injecting an exception into SBAuth and recording how long it takes for Pinpoint to notice a statistical significant anomaly in RUBiS's behavior. We also spot checked our results against fault injections in several other of RUBiS's components to ensure that we were not seeing behavior unique to faults in SBAuth.

Overall, Pinpoint detected the failure within the range of 15 seconds to 2 minutes, depending on the system's and Pinpoint's configurations. To explore what affects Pinpoint's time-to-detection, we repeated our experiment many times, varying the client load, the periodicity with which Pinpoint checks for anomalies, and the amount of immediate "history" Pinpoint remembers as a component's current behavior.

At low client loads (*e.g.,* 10 clients, or $\approx$ 85 requests/min), our injected fault is triggered infrequently, causing Pinpoint to detect failures relatively slowly. Additionally, the nature of the RUBiS load generator causes a high variance, as randomized client behavior sometimes triggers faults rapidly and sometimes slowly. As shown in Figure 5.11, increasing the load to 50 clients ($\approx$ 430 requests/min) improves the time

[4]SBAuth provides user authentication services, verifying user names and passwords and returning user IDs to the rest of the system.

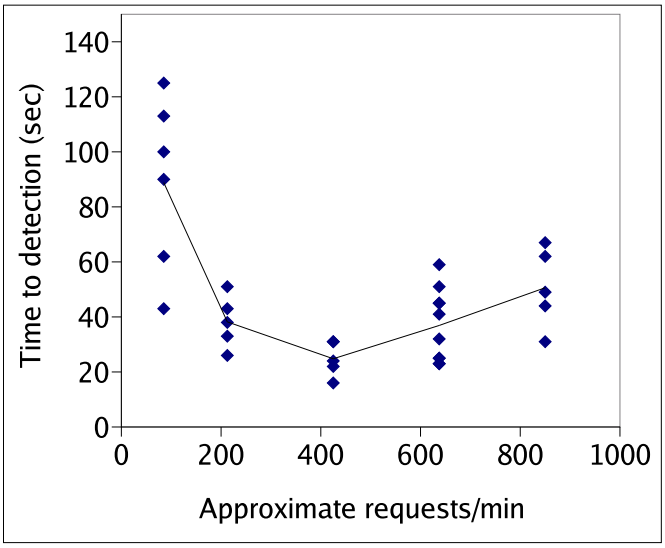


Figure 5.11: **The time to detect a failure as we vary the client load on the system..** The time to detection improves as more of the system is exercised with more clients, then degrades as the extra load induces queuing delays in reporting observations. In these experiments, the current component behavior is based on the prior 15 seconds of observations, and Pinpoint searches for anomalies every 5 seconds.

to detection, and the variance across our experiments decreases. As we continue to increase the client load, however, our time to detection begins to increase again. Due to a performance artifact in our instrumentation of JBoss, placing too high a load on the server causes observations to be delayed in the reporting queue, meaning that at any given time, Pinpoint is not analyzing the current state of the system. At 100 clients ($\approx$ 860 requests/min), detection of the failure is being delayed by 40 seconds. Without either reducing load or dropping observations from the reporting queue, the queue does not recover, and observation delays only increase over time. We believe that a more efficient observation system that avoided reporting delays would allow continued improvement with increased client load.

By decreasing the amount of history Pinpoint remembers to represent a component's current behavior, we increase Pinpoint's sensitivity to changes in application behavior. In short, changes in behavior can dominate the model of current behavior more quickly if there is less "old" behavior being remembered. However, this affect causes a much less pronounced affect on detection time than the client load.

We also experimented with changing how often Pinpoint searches for anomalies, but found that as long as the period was significantly less than our time-to-detection, it was not a major factor in the time to detect a fault.

5.7 False positive rate

False positives can be a serious issue in anomaly detectors. In our analysis of Pinpoint, we recognize two kinds of false positives: *semantic false positives*, where Pinpoint correctly detects a change in system behavior, but this change is acceptable and correct behavior, not a failure; *algorithmic false positives*, where our algorithms make statistical mistakes, as the available information about system behavior incorrectly classifies correct behavior as anomalous. [5]

Alongside each of our fault detection experiments in this chapter, we ran control experiments where we monitored a fault-free (and change-free) Internet system. In none of these control experiments did Pinpoint show any false positives, leading us to infer that Pinpoint is resilient to algorithmic false positives under stable workloads. We did, however, notice algorithmic false positives when identifying faulty requests during a failures, as discussed in Section 5.5, as well as algorithmic false positives in the fault detection of non-structural behaviors, as described in Section 5.9.

As the rate of semantic false positives is tied to the rate of administrative action, frequency of hardware and software updates, and other environment-specific factors at an individual Internet service, we do not believe that we can realistically test the *rate* of false positives in our testbed. We can, however, study Pinpoint's resilience to marking common day-to-day changes to a service as anomalies. To do just this, we ran two case studies that involved the kinds of common system changes that are likely to cause noticeable changes in system behavior.

[5] In addition to false positives, there is the separate issue of judging the importance or criticality of a failure. Sometimes called impact analysis, this is basically the problem of determining whether a true failure is worth fixing, and is not directly addressed by this dissertation. Some heuristics for determining the importance of a true failure might incorporate the magnitude of the behavioral change, how often the affected component is historically used, or the number of requests affected by the failure. However, it is not obvious how well any of these heuristics would work.

In our first case study, we significantly changed the load offered by our workload generator—we stopped sending any ordering or checkout related requests. In our second case study, we upgraded the Petstore v1.3.1 to a bug-fix release, Petstore v1.3.2. For both our path-shape and component interaction analyses, we used a historical analysis based on the behavior of Petstore 1.3.1 under our normal workload. Of course, we expect that major software upgrades will significantly change application functionality and require Pinpoint to retrain its models of system behavior. As discussed in Chapter 2, minor software upgrades, such as this bug-fix upgrade, occur much more frequently, on the order of days or weeks, whereas major software upgrades occur every few months.

In both of these studies, neither our path-shape analysis nor our component-interaction analysis triggered false positives. In the workload-change case study, none of the paths were anomalous—as to be expected, as they are all valid paths. And though the gross behavior of our components did change with the workload, the fact that we analyze component interactions in the context of different types of requests compensated for this, and we detected no significant changes in behavior.[6]

In the upgrade to Petstore 1.3.2, we also did not detect any new path shapes; our component behaviors did change noticeably, but still did not pass the threshold of statistical significance according to the χ^2 test.

Though not comprehensive, these two studies suggest that our fault detection techniques are robust against reporting spurious anomalies when application functionality has not significantly changed.

Despite Pinpoint's apparent robustness against reporting algorithmic and semantic false positives, we do not make the claim that false positives will not occur. Instead, in Section 9.3, we argue that the problems caused by false positives can be avoided if we can ensure that actions taken in response to false positives are inexpensive and safe.

[6]As noted in 4.1, resilience to workload variation comes at the cost of not detecting failures whose only visible symptom is a change in user behavior.

5.8 Limitations

Here, we describe two limitations of Pinpoint that we have observed. The first is a reconfirmation of one of Pinpoint's assumptions. The second is a limitation we discovered in analyzing components with multi-modal behaviors.

5.8.1 Request-reply assumption

We decided to test Pinpoint's assumption of monitoring a request-reply based system in which each request is a short-lived, largely independent unit of work by applying Pinpoint to a remote method invocation (RMI) based application. While RMI is a request-reply system, a single interaction between the client and server can encompass several RMI calls, and the unit of work is not always well defined.

In our experiment, we monitored ECPerf 1.1, an industry-standard benchmark for measuring the performance of J2EE implementations. ECPerf contains 19 EJBs and 4 servlets. Because running unmodified applications is one of Pinpoint's goals, we decided to use the simplest definition of a unit of work in the context of this system, a single RMI call from the client to the server. This avoids the extra maintenance and deployment effort required either to mark up a client-server interaction with explicit `begin-work` and `end-work` markers or to understand the client-server interaction in enough detail to interpret particular RMI calls as beginning or ending a unit of work.

Unfortunately, under this simple definition of a unit of work, most of the resultant paths we captured end up being single component calls, with no structure behind them. Thus, when we injected faults into ECPerf, there are no observable changes in the path, since there was very little behavior in the path in the first place. When a faulty component is called, we observe an incoming RMI and a return result, showing the same behavior at a coarse-grain as when there is no failure.

Path-analysis did detect some anomalies in the system when we injected faults, but these anomalies did not occur in the same RMI requests that we injected faults into. Presumably, some state in the client was corrupted, and affected later RMI calls that did have more structure behind them. The end result of detecting anomalies in secondary failing requests, but not in the first request that failed is that it becomes

more difficult to track down the root cause of a problem.

To generalize Pinpoint to an RMI-based system, we would have to expand our definition of a path to encompass multiple interactions, though this would likely entail application-specific tailoring.

5.8.2 Multi-modal behavior

We found another limitation of our component interaction analysis was the monitoring of components with multi-modal behaviors. In this context, we use the term multi-modal behavior to refer components which take on one of multiple behavior profiles, each of which is considered correct, but all of which are significantly different from one another.

While monitoring the clustered version of Petstore 1.1, one of the components we monitored was the middleware-level naming service. This service has one behavior mode in the front-end tier, where it mostly initiates name lookups, and another behavior mode in the middle tier, where it mostly replies to lookup requests.

We found that in monitoring this component, with multiple modes of behavior depending on the deployed location of the component instance, our method of modeling component interactions was highly prone to false positives. In essence, our component interaction model as described in Section 4.3.1 attempts to capture a non-existent average of the multiple modes, and subsequently detects all the components as deviating from this average.

One possible solution to easing this limitation is to use a model that captures multiple modes of behavior, though this has the danger of mis-classifying truly anomalous behavior as simply "another mode." Another option is to extend the component-identification scheme to differentiate between components placed, for example, in the web tier vs. backend, thus allowing Pinpoint's analysis to build separate models for each. The difficulty with this latter option is to ensure that the naming scheme captures the right details, without requiring extensive *a priori* knowledge of a component's behavior, and also to avoid splitting valid component groups apart to the extend that the models lose their ability to validate behaviors across many peers.

Statistics	Description	Type
NumElements	Number of objects stored	State
MemoryUsed	Total memory used	State
InboxSize	Size of Inbox in last second	State
NumDropped	Dropped request in last second	Activity
NumReads	Reads processed in last second	Activity
NumWrites	Writes processed in last second	Activity

Table 5.4: **Metrics that are monitored to detect failures in SSM.**

5.9 Fault detection with non-structural behaviors

In this section, we describe our experiences detecting failures by monitoring non-structural behaviors in a semi-persistent clustered hashtable—a system whose structure would likely not vary in the face of many failures—using the techniques described in Section 4.5. We present experiments run in the context the Session State Management (SSM) prototype [93]. Pinpoint's monitoring has also been deployed in the context of the DStore persistent clustered hash table. Details about Pinpoint and DStore are available in [70].

SSM is a clustered hash table designed and optimized for storing user session state within large scale Internet services. This session state requires persistence for only a bounded length of time, and includes data such as a user's shopping cart or their recent searches. Clients of SSM, application software nodes within the Internet service, store data on some number of SSM nodes in the cluster. How many copies of the data are saved depends on the data's durability requirements. Clients are responsible for remembering where in the cluster their data is saved. The specifics of SSM's protocols decouple the performance and reliability of individual requests from the performance and reliability of individual nodes in the SSM cluster. This allows SSM nodes to be rebooted and managed as if they were stateless machines.

To monitor SSM, we made minor modifications to its core software to record and report several metrics that we believed might show correlations with faulty behavior. These metrics, shown in Table 5.4, included both state statistics and activity statistics, as defined in Section 4.5.

These metrics are periodically (once every second) reported to a central Pinpoint

analysis engine, which determines whether any nodes of the system might be acting anomalously. If anomalous behavior is found, the offending node is rebooted, with the goal of at least temporarily curing transient failures.

While SSM is able to gracefully tolerate performance failures in individual nodes, we do wish to recover these nodes to prevent an eventual decay in overall system performance. In the following benchmark, we ran an experiment with 6 SSM nodes and 3 load generating machines, each sending approximately 450 requests per second. We injected a performance fault into one of the nodes of SSM by introducing a 1ms sleep before handling each request. This performance fault simulates the kind of failure that might occur because of software aging. Figure 5.12 shows the resulting performance of the overall cluster as we inject this performance fault every 60 seconds. Each time the fault is injected, Pinpoint detects it within 5-10 seconds and triggers a recovery of the node through reboot. During both the performance fault and subsequent recovery, all requests are serviced properly. This combination of automatic fault detection and repair is a compelling example of autonomous recovery.

During normal behavior, some of the metrics we monitored did appear individually anomalous. However, our strategy of combining statistics, as described in Section 4.5.3, successfully filtered these potential false positives out. We found that true failures caused multiple metrics to appear anomalous simultaneously, and therefore were not filtered out.

5.10 Experience in real environments

To validate the usefulness of using statistical monitoring for fault detection in real environments, we have applied this approach to analyzing system and application logs from two large Internet services.

5.10.1 Amazon.com

Amazon.com is a popular e-commerce site, and one of the largest such sites on the Internet today, selling millions of items, from books and DVDs to apparel and toys. Amazon.com provided us with samples of logs from a period of trouble-free operation,

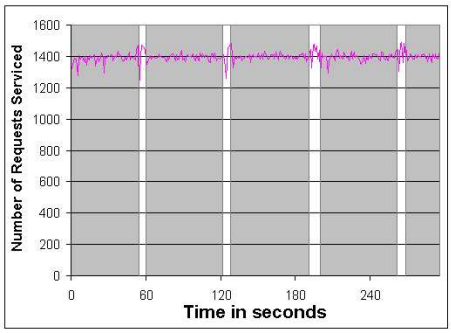


Figure 5.12: **Detecting performance failures in a single SSM node.** A 1ms sleep is injected before each request. Pinpoint detects the failure within 5-10 seconds and triggers a reboot of the faulty node. The white bands indicate periods during which a fault is being injected into an SSM node. The width of each white band is the time taken to detect a failure and trigger a reboot. As noted, the design of SSM decouples the overall system performance from failures in any single node, hence the stable overall performance during periods of fault injection.

as well as three sets of sample logs from three different periods of failure.

Amazon.com's architecture is similar to the tiered architecture we described in Chapter 2. The key points are that Amazon.com runs many distinct software services that interact to fulfill user requests. Each software service is deployed on a, possibly large, set of machines. We did not modify Amazon.com's infrastructure or software to collect our logs, and instead analyzed data that was already being collected.

Many software services at Amazon.com, including the front-end tier and many of the backend services, collect some form of *transaction record* logging information about the functions being performed at each service for each request. Figure 5.13 shows an illustration of the kind of information available in these logs.

While some of these transaction records are marked with a request ID, not all are, and in the log samples we are analyzing, we do not have enough information to tie transaction records across the system together into a complete record of the runtime path used to service a end user's request. Instead, we look to analyze the data in these logs using a form of component interaction analysis.

Since we do not know what the most useful unit of failure might be, *e.g.,* whether it is hosts, processes, or something else that might be the best indicator of a fault, we cast a broad net: we treat the value of each $\langle key = value \rangle$ pair as a potentially

 # information about the incoming request
`URL` = *http://www.amazon.com/exec/obidos/search-handle-form/...*
`ID` = *1234567890*
`name` = `www.amazon.com`
`port` = 80

 # information about processing node
`Hostname` = *internal host*
 # information about the processing of request
`Methods` = {*methodA*,*methodB*,*methodC*}
`Time` = *123 ms*
 # information on returned values
`Size` = *123 bytes*
`Status` = 200

Figure 5.13: **An illustrative example of the kinds of information available in Amazon.com's log format.** Transaction records are formatted as $\langle key = value \rangle$ pairs. While some keys are standard across Amazon.com's systems, other keys are service-specific. This example lists the information that might be captured as a request first enters the Internet service. At other tiers in the system, the logs might include information about contacted databases or the type of software service being called.

significant component identifier. Some of these values may represent actual instances of software or hardware components, while others may be more abstract entities, such as protocol port numbers or even query strings. Not all of these entities will prove useful, but at this point, we believe that the important ones will show their significance in later stages of our analysis, while the unimportant ones will not.

We assert that if two component identifiers co-occur in the same transaction record, then a potential interaction or relationship exists. By observing the co-occurrences of component identifiers in many transaction records over time, we see the true interactions reinforced, while occasional, circumstantial co-occurrences will appear as noise.

At this point, we have almost enough information extracted from these logs, both component identifiers and component interactions, to run our component interaction analysis as described in Chapter 4. However, our analysis, and especially our peer-based component interaction analysis, depends on knowing the component classes in the system. Additionally, our historical analysis also has difficulties as many of the component identifiers appearing in the logs are nominally transient (for example, many search queries enter the system, but each of their URLs is different as each is executing a different search). While we do not discuss it in detail here, suffice it to say that we are able to extract a sufficient approximation of these classes using a data clustering algorithm. The approach we take is similar to the approach we present in Chapter 6 for extracting complex data structure definitions from the simple data types of the Windows Registry.

Once we have this approximation of component classes and their membership information, we apply our component interaction analysis in a straightforward manner. We take 5-minute-long snapshots of the transaction records in our sample logs, and use each 5-minute snapshot to build a historical reference model of component behaviors. Rather than simply take an arbitrary snapshot (*e.g.,* the first snapshot) as a reference of believed good behavior, we assume simply that over time, it is likely that most snapshots are will show good behavior. So, we use every 5-minute snapshot as a reference model (and hence a point of comparison) for all of the following 5-minute

snapshots.[7]

For convenience, we generate a single *anomaly score* for a snapshot by summing the anomaly scores of all components across all comparisons to existing reference models. More formally, for a snapshot s_t, reference models $\{r_1, ..., r_{t-1}\}$, and components C existing in s_t, we calculate:

$$anomaly(s_t) = \frac{1}{t-1} \sum_{i=1}^{t-1} \sum_{c \in C} comparison(c, s_t, r_i) \tag{5.1}$$

where $comparison(c, s, r)$ calculates a comparison function (such as χ^2) between the behavior of component c in s and the behavior of the same component in r. While this summation does have some obvious biases, such as a bias against systems with many unique components, it is useful for quickly visualizing trends of change. One correction that we do make is to introduce a prior belief that there is no anomaly in the system:

$$anomaly(s_t) = \frac{1}{t-1+k} \sum_{i=1}^{t-1} \sum_{c \in C} comparison(c, s_t, r_i) \tag{5.2}$$

The constant k effectively reduces the influence of unrepresentative snapshots taken early in the analysis process by discounting the anomaly score early in the process. As t grows and we have built up more reference models, the influence of k weakens, and in the limit, Eq. 5.2 approaches Eq. 5.1.

We analyzed sets of sample logs from each of three different historical failures. In each of these cases, our analysis of the logs found detectable signs of anomaly at the same time or earlier than Amazon.com's existing detection mechanisms. In two of the failures, shown in Figures 5.14 and 5.15, the anomaly score jumped several orders of magnitude during the failure. In the third failure, shown in Figure 5.16, the anomaly score shows generally more noise and increases by only a single order of magnitude during the failure. We also analyzed 2 hours of logs captured during a trouble-free period showed a low level of background "anomaly-ness", consistent with the levels

[7] In a real deployment one would eventually have to throw away old reference models, but because our sample logs are only a couple hours long, we have the luxury of keeping and using all the models we build as references.

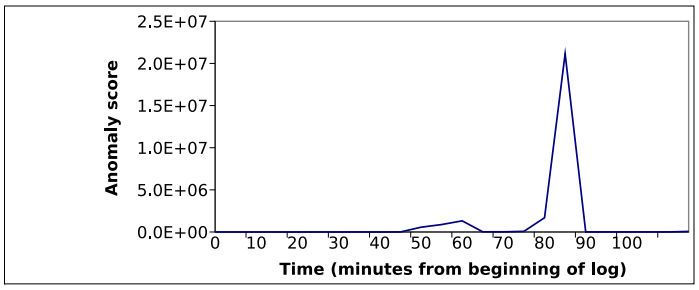


Figure 5.14: **Anomaly calculations during a failure.** The problem occurs at time 80, and shows precursors minutes earlier

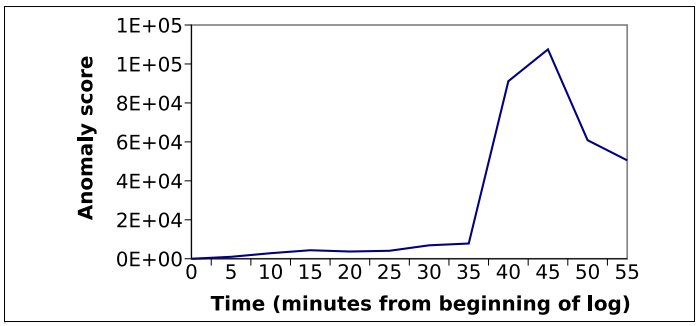


Figure 5.15: **Anomaly calculations during a second failure.** This failure occurs at time 40

of anomaly-ness during the non-faulty periods in Figures 5.14 and 5.15 but found no spurious spikes. While we would have to analyze many days of logs to estimate a false positive rate for our detection techniques, the lack of anomalies in these 2 hours of logs captured during a fault-free period, as well as the non-faulty periods of our other sample logs, is encouraging.

This analyses suggest that major failures in a real systems are detectable using statistical monitoring, as well as suggesting two additional benefits:

- First, we found that statistical monitoring was able to detect a variety of different kinds of failures without requiring special knowledge of the application or system, and without requiring special configuration to watch for particular types of failures. That is, the same mechanism was able to detect failures in different layers of the system, including the network, the database, and load balancer. In some cases, such as the database failure, statistical monitoring detects

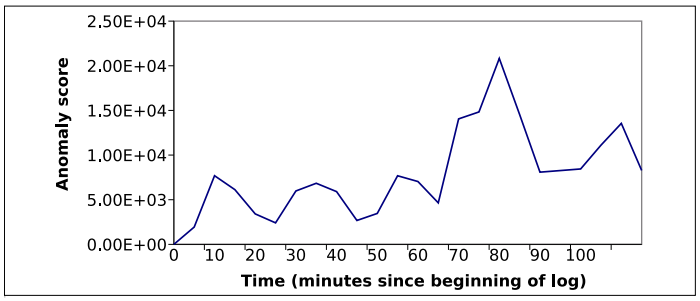


Figure 5.16: **Anomaly calculations during a third failure.** A failure occurs at time 80. Background noise is high, but without analysis of additional logs, it is not possible to tell whether this is related to the onset of a failure.

the failure even though we are not directly monitoring database behavior.

- Secondly, we found that in one of the three sets of logs we analyzed, our statistical monitor noticed anomalous behavior 5-10 minutes before the failure was noticed by Amazon.com's current process. Given that this failure was repaired within 20 minutes of being discovered, a 5-10 minute head start could have been significant.

5.10.2 Large Internet service #2

In addition to applying statistical monitoring to analyze a kind of structural behavior at Amazon.com, we have been able to apply our non-structural behavior analysis to metrics captured at a second large Internet site. The captured metrics include quantitative statistics such as response time and CPU usage. In this analysis, we analyzed only activity statistics, such as response time and CPU usage, using the median absolute deviation, as described in Section 4.5.

We analyzed metrics captured in real time from on the order of a hundred machines, and marked machines whose metrics were not matching their peers. While our results are preliminary, we were able to detect a number of real failures. We found that the nodes with the most significant deviations in their metrics were, upon further investigation, failing. The failures we noticed included several that were not detected by existing monitors at the site. As example of the variety of failures we were able to detect, two of the underlying faults we found were a node whose hard

drive was full and a rack whose network bandwidth was being hogged by a single errant machine.

During this evaluation, we made several interesting observations. First, we found that, as in our Amazon.com experiments, statistical monitoring was able to detect a variety of underlying fault types without monitoring for each type of fault explicitly. The implication is that we do not need to have significant *a priori* knowledge of a site's vulnerabilities to faults to be able to detect these failures. Secondly, we found that in this site, it was common to have components that demonstrated bi-modal behavior, confirming the limitations as described in Section 5.8 of building a model based on the average behavior of the whole population of a class of components. In this case, the bi-modal behavior was easy to recognize and filter. Solving this problem in the more general case, however, remains a topic of future work.

5.11 Summary

This chapter described our evaluation of statistical monitoring applied to fault detection:

- In the context of our testbed Internet service, our prototype statistical monitor, Pinpoint reduced the number of undetected faults by 30-70%, depending on the specific class of fault. Moreover, in an online deployment, our prototype detected failures within 15 seconds to 2 minutes, depending on client load.

- We find that injecting failures caused by simple source code bugs can have a visible affect on Internet service output without causing interfering with major functionality. In these cases, our statistical monitor does not detect significant anomalies in system behavior, nor does it detect significantly more source code bugs than alternative low-level failures.

- In our testbed, our prototype monitor showed resilience against false alarms during common day-to-day system changes, such as extreme workload variations and minor software upgrades.

- We demonstrated that statistical monitoring of non-structural behavior can also be a useful fault detector when a system does not have significant structural behavior to analyze. We showed this in the context of Session State Manager (SSM), a clustered hash table, where Pinpoint monitored for performance and other failures. In this deployment, Pinpoint detected failures within 5-10 seconds. Because SSM is designed to tolerate crash failures and reboots, we were able to use Pinpoint's alarms to trigger automated recovery of failed nodes without harming the performance or functionality of the overall system.

- We discussed the early results of applying statistical monitoring techniques to detect failures in real log data garnered from two large Internet services. While preliminary, the results are also promising, demonstrating that statistical monitoring can detect real failures in large systems without requiring a detailed understanding of the system itself, without anticipating the specific kinds of faults that might occur, and in at least some cases, without explicitly monitoring the portions of the system that were the cause of the failure.

Together, we believe our experience supports our hypothesis that statistical monitoring can be an important tool for detecting failures in large systems. Specifically, the properties of statistical monitoring techniques seem to be particularly well suited to poorly understood component-based systems, such as rapidly changing and complex Internet services.

Chapter 6

Inferring system structure

In addition to fault detection, there are several stages of the fault management process that can benefit from statistical monitoring. In this chapter, we study the problem of describing the structure of a system, and present a statistical monitoring approach to infer patterns that describe its structure.

Descriptions of a system's structure are an important part of deciding whether the system is in a good or bad state, and may also aid operators' understanding of the system. Unfortunately, the structures that would help us more easily detect problems are often not explicitly represented in the system's state. As a simple example, consider the information lost when a program's high-level source code is compiled to low-level hardware instructions. To enable debugging or reasoning about the program's execution, we must either re-insert the lost information through annotations or refer back to the original source code. The challenge is that not all systems we wish to "debug" or reason about have an equivalent to high-level source code which we can reference.

In the context of Internet services, we may be able to tell whether or not two back-end servers are behaving similarly. However, we may not always know whether they *should* be behaving similarly. In Chapters 4 and 5, we assumed that this information was already available, through well-known component types, directories, manifests, or similar sources. Unfortunately, these sources are not always available or sufficient. For example, we may be externally analyzing the machine-level behavior of the nodes

in a service, without knowledge of what software components are running on each machine or a manifest identifying the peer groups within the machines in the service. In this case, analyzing the behaviors of the machines in a service over time may allows us to approximate whether two machines should be behaving similarly based on whether they have behaved similarly in the past.

In this chapter, we focus on inferring hidden system structure in a different domain: configuration settings stored in the Windows Registry. The Windows Registry stores hundreds of thousands of configuration settings. Each individual setting has a basic type (String, integer, boolean, etc.) associated with it. Usually many configuration parameters, grouped together, control a single logical part of the system's behavior. For example, one group of parameters might control a device driver, while another group might control a printer configuration. These groups often have strict constraints on their format, but these constraints are not enforced, and sometimes are not even well-known. This makes detecting misconfigurations difficult.

While the Windows Registry is a very different environment from the Internet services environment discussed in the rest of this thesis, the size and complexity of the registry means that statistical monitoring is still relevant. Primarily, statistical monitoring gives us the ability to analyze a large amount of data to make useful assertions about an otherwise poorly understood system. Gathering a description of the complex data structures within the Windows Registry is not otherwise feasible, as the operating system's policies and innumerable third party software applications and corporate IT policies combine to define the configuration settings in the registry. Moreover, the data-driven approach lets us provide customized results for specific environments, and the automatic nature of the analysis enables us to easily adapt as the underlying system changes.

The next section provides more background on the configuration problem in general and the Windows Registry specifically. The rest of this chapter details how we apply statistical monitoring to automatically infer the hidden structure within the Windows Registry and evaluate its use in detecting misconfigurations.

6.1 Misconfigurations

Managing the configuration of computer systems today is a difficult task. Too easily, a computer user or administrator can make a simple mistake or lapse and misconfigure a system, causing instabilities, unexpected behavior, and generally unreliability. Bugs in the software that changes these configurations, such as installers, only worsen the situation. Unfortunately, while there are many constraints which can differentiate between valid and invalid settings, few of these constraints are explicitly written down, much less written down in a form that could be automatically applied to detect misconfigurations.

The difficulty of managing a complex configuration is a wide-spread problem, affecting a large variety of systems. In [107], Oppenheimer studies failures at three large Internet services and finds that configuration errors were the largest category of operator mistakes that caused end-user visible downtime. Studies of wide-area network systems indicate that misconfigurations in BGP are responsible for almost 3 of every 4 BGP routing announcements [94]; and that misconfigurations are a significant cause of extra load on DNS root servers [17].

In [49], Ganapathi *et al.* study problems related to the Windows Registry, and find that faulty configuration data in the registry can cause a variety of failures, from general system instability to hiding critical functionality from a user or causing normal functionality to have unanticipated side effects. These misconfigurations can occur for any number of reasons, including failed application installations or uninstallations that leave behind an inconsistent configuration; a malicious or buggy program that corrupts a user's configuration; untested interactions between different versions of a library or program; or a user who is simply unaware of the side-effects and semantics of a configuration parameter.

Ganapathi *et al.* show that over one third of the misconfigurations they studied could have been detected and diagnosed by proactive monitoring of the registry. The challenge of proactive monitoring, however, is knowing what misconfigurations might occur and how to look out for them. Automatically discovering likely signs of misconfigurations is the focus of this chapter.

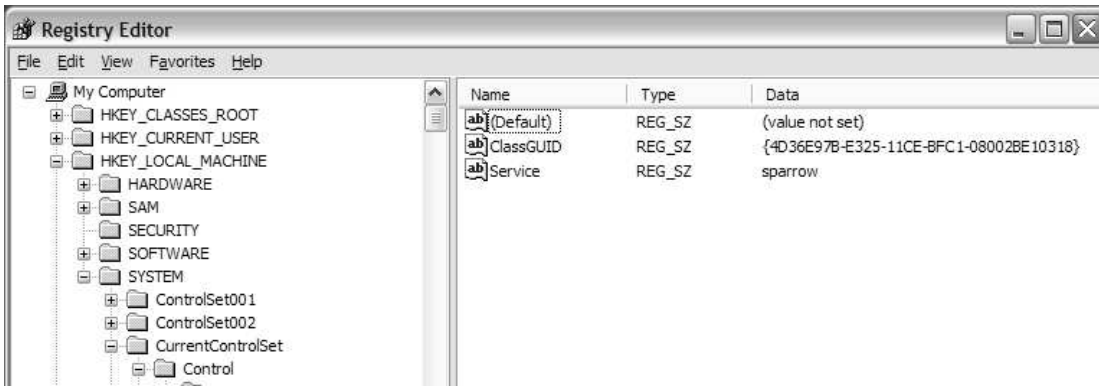


Figure 6.1: **Some of the hierarchical keys and values in a typical Windows Registry..** HARDWARE, SECURITY and SYSTEM are among the top-level keys, and each has several subkeys (though only SYSTEM is expanded to show its subkeys in this view). ClassGUID is the name of a value, and {4D36E97B...} is the actual content of that value.

6.2 Background: Windows Registry

The Windows Registry provides centralized storage for information and settings about the hardware, operating system, applications, users and user preferences on a Windows PC. The registry provides a hierarchical structure for settings, allowing keys to have subkeys and named values, similar to the directory and file structure of a file system, as shown in Figure 6.1. It is up to clients of the registry to decide how to organize their own settings, though some conventions are generally followed. For example, vendors usually place their user-specific application settings underneath the key \HKEY_CURRENT_USER\Software\[VendorName].

For the purposes of our analysis, we use a slightly simplified representation of the Windows Registry structure. Rather than having hierarchical sets of keys containing $\langle name, value \rangle$ pairs, we add the name of the values as a leaf key in our tree of registry keys and assign values directly to the leaves in our tree. This minor change simplifies our model, without causing loss of information contained in the registry.

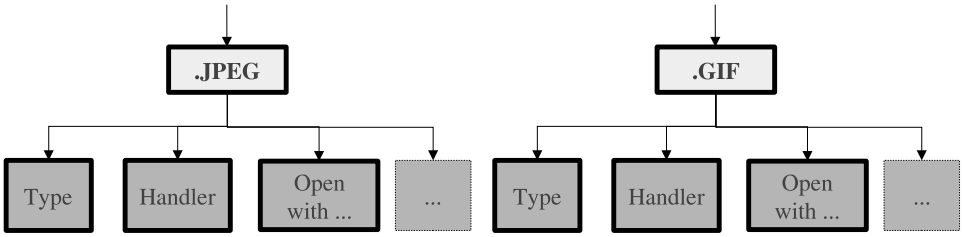


Figure 6.2: **An example configuration class.** This figure shows two examples of a file type registration. Both the .GIF and .JPG file type registrations have a similar structure, containing configuration settings that describe how to open and how to name GIF and JPEG image files, respectively. By discovering such similar structures, we can recognize configuration classes that can help us better organize and manage the Windows Registry.

6.3 Approach

To better detect misconfigurations in the Windows Registry, the first part of our approach is to analyze registry data from systems that we believe are correctly configured. These systems include research and support desktops under use, as well as freshly installed computers, all of which appear to be working correctly, and whose users are not complaining of particular problems with their configuration. From these snapshots, we will extract from them what we call *configuration classes*, the equivalent of complex data type definitions and describe the format of groups of settings within the registry. As shown in Figure 6.2, one configuration class we can infer from the Windows Registry is the group of keys describing file type registrations. Every file type registration uses several keys grouped together to describe, such as, what to call a type of file, and how to open the file. The same key structure is repeated for each registration of a file type.

In the absence of the ground truth definition of configuration classes, we attempt to infer their existence by analyzing the keys in the Registry using a data clustering

algorithm. In this analysis, we look for similar groups of keys occurring in multiple places in the registry as signs of an implicit configuration class. Thus, while in Chapter 4 we used statistical techniques to detect changes in observable structures, here we are using statistical techniques to infer the complex structure of the Registry from relatively unstructured observations of the Registry's content.

Once we have found these configuration classes, we search for rules that describe the constraints and invariants on them. We hypothesize specific constraints based on the data in our registry snapshots, and validate our hypotheses against all our snapshots before fully believing them to be valid constraints.

By taking a statistical monitoring approach to learning configuration classes and inferring correctness constraints, not only are we able to automate the process, but we are also able to tailor our learned constraints to specific environments. For example, in many environments, it is not necessary to configure a web proxy, and the proxy configuration can correctly remain empty. But, in most corporate IT environments, setting a proxy is a prerequisite for functioning web access. Having the ability to easily infer a set of constraints from examples means that we can learn these environment-specific constraints easily as well.

PeerPressure, described in [127], takes a similar statistical monitoring approach to isolate configuration errors in the Windows Registry. The primary difference between it and the work described in this chapter is that PeerPressure's goal is troubleshooting after the fact, whereas our goal is to infer enough information from known-good registries to be able to detect misconfigurations as they occur, regardless of whether they have yet caused a problem for the user.

6.4 Discovering configuration classes

Configuration classes are a natural first building block to discovering more about the structure of the information stored in the Windows Registry. We know that many types of information stored in the registry, including software registrations, per-user account information, and hardware settings among others, are all repeated for each instance of the entity they describe. Discovering this extra structure within

a registry allows us to create general configuration constraints that apply across all of the instances of a class. We look for repeated groups of configuration settings that share a common hierarchical structure: two items in a configuration belong to the same class if more than a threshold amount of their substructure is identical.

Note that configuration classes ignore the ancestors of a key in the hierarchy. That is, whether or not two keys are in the same location in the hierarchy does not affect our decision to put the keys together in a configuration class. Basing our decision only on the substructure of keys allows us to define a finer-granularity of class and also allows us to detect configuration classes that are spread out across the registry in separate user accounts, backups of parts of the registry, etc.

6.4.1 Class discovery algorithm

We use data clustering, described in 2.5.3, to identify configuration classes by grouping together configuration settings with similar structures. There are two main flavors of data clustering algorithms, each of which requires some *a priori* information about what the resulting clusters should look like. Partitioning methods, such as K-means clustering, requires one to know how many clusters should be created, while hierarchical clustering requires one to know how far apart distinct clusters should be from each other. In this context, we do not know how many configuration classes we might find, and therefore, find hierarchical clustering to be more appropriate. In recent years, a third flavor of data clustering algorithms, correlation clustering, has been developed which requires no desired number of clusters or distance threshold, but instead a definition of similarity and dissimilarity [8, 41].

Our prototype uses a hierarchical, bottom-up clustering method using arithmetic averages (UPGMA) and calculates the distances between clusters based on the simple convex average metric [61, 73]:

$$\text{distance}(a, b) = \frac{\sum {a_i}^2 - {b_i}^2}{N} \tag{6.1}$$

where a and b are the two clusters being compared. a_i and b_i represent the proportion of registry keys within each cluster that contain the i^{th} subkey. For example,

if half of the keys in the cluster a contained the subkey `TYPE`, then $a_{\texttt{TYPE}} = 0.5$. N is the number of unique subkeys occurring in the keys of a and b together.

We stop clustering when we meet a threshold distance. If the threshold is set too high, then we may fail to notice commonalities between slightly different sets of keys and either find too many distinct configuration classes or find that keys are not grouping together classes at all. In contrast, if the threshold is set too low, then dissimilar groups of keys will be lumped together, and we will find configuration classes whose members might not have much to do with each other at all. In the experiments shown in this chapter, we set this threshold to 0.01. Empirically, we found this threshold to work well in keeping a good balance in discovering configuration classes.

The complete algorithm for discovering configuration classes and extracting names for them is shown in Algorithm 1. The first step (before data clustering is applied), is to filter out keys with little substructure. This removes keys, such as leaf-nodes in the hierarchy of registry keys, with so little structure that they are unlikely to be part of a configuration class.

Once we have initialized each key into its own unary cluster, we begin running our data clustering algorithm. The $findClosestClusters()$ function searches for and returns the two clusters closest to each other. We merge that pair together, and add the combined cluster to our set of clusters. Once we have merged all clusters less than a threshold distance apart, we stop. At this point, the resulting clusters larger than $minclustersize$ are our discovered configuration classes.

While we are using this algorithm in the context of the Windows Registry, the outline of the algorithm and how we apply it to infer similarities and structures between groups of elements is quite general. This should come as no surprise, as data clustering as been used to organize and discover structure in data sets for over 80 years. The specialization that has to occur when applying data clustering to infer structure in a specific domain is determining what observations are most likely to give us clues about the implicit structure we wish to discover. Once we have identified what observations are likely to be most useful, we can define a distance metric on those observations and apply data clustering.

Algorithm 1 Algorithm for data clustering and naming.

load registry R
$clusters = \{k | k \in R, |k.subkeys| \geq minsubkeys\}$ {ignore keys that have too little substructure}
loop
  $pair = findClosestClusters(clusters)$
  if $pair.distance \leq thresholddistance$ **then**
    $clusters.remove(pair)$
    $newcluster = merge(pair)$
    $clusters.add(newcluster)$
  else
    exit loop
  end if
end loop
for all c s.t. $c \in clusters$, $|c| \geq minclustersize$ **do** {loop through resulting configuration classes}
  $name_c = \{s | \forall k \in c, s \in k.subkeys\}$
end for

Once we have our set of configuration classes, we need an additional step of analysis to create identifiers for each class. We do this by extracting the common substructure of the keys in each cluster as the name of the configuration class. To double-check that this class name is appropriate, we can verify that all or almost all of the keys in the registry snapshot that match this substructure are within the discovered cluster. While the specific structure we use is registry-specific, using the common structure of the items in the cluster to define its identity is a generic one.

A more sophisticated naming system might only use those features which differentiate this class from others, in effect, reducing the size of the name without reducing its efficacy. Though this would be more efficient, our experiments have not yet warranted the added complexity.

6.4.2 Naming class instances

Once we have discovered a configuration class, it is useful to also know how to name instances of the class. This becomes especially important later, as we look for settings in the registry that might be referencing these instances (see Section 6.5). While

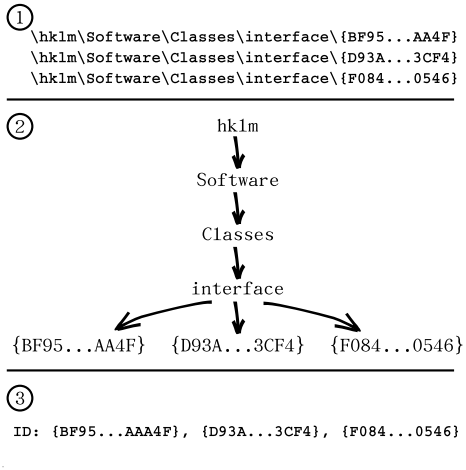


Figure 6.3: **Learning likely identifiers for class instances.** By looking at the hierarchical difference in the names of the keys shown here, we can infer that *hklm*, *Software*, *Classes*, and *interface* are not identifying names, and that the strings on the level of $BF95...AAA4F$ are.

determining the exact identifier is difficult, we can easily determine a superset of identifying strings by looking at the differences in the hierarchical key names of the instances. Figure 6.3 shows an example of this process. Here, we look at three keys that differ only in their last elements, presenting us with an obvious identifier for each instance.

The problem of determining the identifying keys becomes more difficult when the configuration class is spread out across the registry in different locations in the hierarchy. For example, completely identifying user-specific configuration settings often requires using both the user's ID string and a randomized ID together. One example is the configuration class formed by the keys:

```
\hku\S...797\....\shellnoroam\bags\55\shell
\hku\S...707\....\shell\bags\8\shell
\hku\S...451\....\shell\bags\3\shell
```

Using our heuristic of looking at the branch-points in the tree represented by these keys, we see that the total set of identifying strings is $\{S...797, S...451, shellnoroam, shell, 55, 8, 3\}$.

In this example, the first branchpoint represents the user ID, and the final branchpoint is an identifying number. Together with the middle branchpoint, this creates a likely superset of the identifies for a key. We call this a superset because it is possible that some of the branchpoints in the hierarchy may not actually be necessary to uniquely identify the keys. For instance, the difference between keys with the *shell* and *shellnoroam* in the above example may not be significant. Unfortunately, without more information than is available here, we cannot automatically determine whether some of these identifying strings are irrelevant, so we include them all.

6.5 Generating hypotheses

In Chapter 4, we described how to use the easily observable units of software components and their behaviors to build models of probably correct behavior. Once we have extracted the structure of configuration classes from the registry, we can use them in much the same way, as a building block to detect misconfigurations and errors in the registry. The primary difference in how we use configuration classes versus components is based on the discrete, unordered nature of the values that configuration settings take on, as opposed to the ordered counts of interactions that we observe in component-based software systems.

In this section, we describe how we look for both *internal* constraints that describe the valid structure of values within an instance of a configuration class, and the *external* relationships across configuration classes and arbitrary keys in the registry. We describe four kinds of constraints and how to generate them: size constraints, value constraints, reference constraints, and equality constraints.

Our general strategy for discovering these constraints is to first start with a template rule that describes the form of the constraint. Then we iterate over the sample registry snapshots and the discovered configuration classes and fill in the template to generate a hypothesis for a constraint. We validates this hypothesis against the data in each of the snapshots to ensure that it meets our confidence thresholds and either accept or reject the hypothesis as a valid constraint.

While the particular template rules we use are specific to our domain, rule finding

is a general technique that has been widely used in various domains, including to discover likely program invariants and errors in systems code [45, 46].

6.5.1 Size constraint

The size constraint specifies that, within a given configuration class, the value of a subkey always has a fixed size. This constraint is an example of an internal constraint, as it only refers to the structure within a single configuration class. Intuitively, the size constraint is likely to describe configuration settings that take a fixed form, even when the values of the form vary.

The template for a size constraint is "$\forall i \in C, |i.subkey| = x$", for a given size x and configuration class C. Using Algorithm 2 and a set of registry snapshots, we hypothesize possible size constraints.

Algorithm 2 Inferring size constraints.

for all c s.t. $c \in set\ of\ configuration\ classes$ **do**
  for all $subkey$ s.t. $subkey \in name_c$ **do**
    if $\exists x$ s.t. $\forall k \in c, |k.subkey \rightarrow value| = x$ **then** {Found hypothesis}
      propose $|c.subkey \rightarrow value| = x$
    end if
  end for
end for

6.5.2 Value constraint

The value constraint is an internal constraint that declares that, within a configuration class, the value of a subkey always takes on one of a small set of values. For example, a value constraint easily describes situations where a key represents an option setting, such as a choice between *TRUE* and *FALSE*.

The template for a value constraint is "$\forall i \in C, i.s \in \{x_1, x_2, ..., x_n\}$" for a small set of values X, a subkey s and configuration class C. To fill this template, we look at all instances of a given type across our sample registries, and, for each subkey within the structure, sees what possible values it has. If the number of values is much less than the number of samples, we form the appropriate hypothesis. In our

implementation, we use $i < lg(n)$ as the threshold for determining whether $i \ll n$. Algorithm 3 summarizes this procedure.

Algorithm 3 Discovering values constraints.

```
for all c s.t. c ∈ setofconfigurationclasses do
  for all subkey s.t. subkey ∈ name_c do
    if |unique(i.subkey → value|i ∈ c)| ≪ |c| then {Found hypothesis}
      propose c.subkey ∈ unique(i.subkey → value)|i ∈ c
    end if
  end for
end for
```

Currently, we only looks for constraints that limit registry values to one of a small number of enumerable values, and do not attempt to discover constraints that limit values to being within a continuous range of values, such as real values between $[0, 1.0)$.

6.5.3 Reference constraint

The reference constraint is an external constraint, and specifies that a particular key in the registry must always reference an instance of a particular configuration class. For example, a default printer setting should name the configuration settings for a printer registration.

The template for a reference constraint is "$k \in ID(i)|i \in C$", for some configuration class C, and where $ID(i)$ is the set of strings identifying the instance i of the configuration class C. Algorithm 4 shows how we infer a hypothesis from this template. We first create a hashtable of all the values in the registry. As we add the values to the hashtable, we preprocess the values to better match registry semantics, such as by lower-casing all strings. We also filter out any values that are too small, under the belief that values such as "1" are so common that they are more likely to generate false positives than true constraints. We then iterate over the configuration classes in the registry, and make a list of all the keys whose values match one of the instances of the configuration class.

While our internal constraints take advantage of the fact that most configuration classes are repeated many times—sometimes thousands of times—to provide a high

confidence in a hypothesis, our external constraints only have one sample per registry snapshot. Because of this lower sampling, it is that much more important to cross-validate reference constraints across many registry snapshots. Our prototype hypothesizes reference constraints for registry snapshot being analyzed, then creates the final constraint only if the reference constraint exists in all of the snapshots.

Algorithm 4 Discovering reference constraints.

Put all values in registry into a hashtable t s.t. $t[v] = \{$ all keys with value $v\}$
for all c s.t. $c \in setofconfigurationclasses$ **do**
  for all id s.t. $\exists i \in c, id \in ID(i)$ **do**
    if $|t[id]| > 0$ **then** {Found hypotheses}
      $\forall k \in t[id]$, **propose** $\exists i \in c$ s.t. $k \rightarrow value \in ID(i)$
    end if
  end for
end for

6.5.4 Equality constraint

The equality constraint specifies that a set of keys in the registry must always have the same value, though it does not constrain what that value may be. The equality constraint is an external constraint, and is the only one of our correctness constraints that does not refer explicitly to configuration classes.

To discover equality constraints, we simply extract all the keys from a registry snapshot into a hashtable, hashed by their values. As in the previous subsection, we lower-case strings in a preprocessing stage as we place keys in the hashtable, since many values, like host names and user names in Windows, are case-insensitive. We filter out small values to avoid false matches. We then iterate over the hashtable, looking at all keys with a common value, and hypothesize that any set of keys whose values are identical should always be equal to one another. Algorithm 5 summarizes this procedure.

To create a cross-validated equality constraint, we iterate over the hypotheses from one registry snapshot, and look for mostly-identical hypotheses among the hypotheses from the other registry snapshots. We then intersect the keys on the left-hand-side of these equality constraint hypotheses to create a final equality constraint. If we

cannot find similar hypotheses in each of the other snapshots, then we invalidate the hypothesis.

Algorithm 5 Discovering equality constraints.

Put all values in registry into a hashtable t s.t. $t[v] = \{$ all keys with value $v\}$
for all v s.t. $t[v] \neq \emptyset$ **do**
  propose $\exists x$ s.t. $\forall k \in t[v], k \rightarrow value = x$
end for

6.6 Evaluation

We have built Glean, a prototype implementation of the algorithms described in this chapter, and have used it to analyze eight registry snapshots from different Windows XP machines. All of these machines were desktop deployments within the same corporate environment. Each of the registries had about 200,000 keys. Each of the performance numbers presented in this section is the mean of 3 successive runs of Glean on a 2GHz Intel Pentium IV machine with 512MB of memory.

6.6.1 Configuration classes

The first step of our analysis of the Windows Registry, discovering the configuration classes, began by filtering out registry keys with little substructure. This step left between 25,000 and 31,000 keys in each of our registry snapshots for us to analyze. We merged these keys into (on average) 1600 clusters. Only 1500 keys of the 25,000 were unique and did not fall into any of our discovered clusters.

These 1600 clusters varied greatly in size, with the plurality of clusters having a small size (half contained only two keys), while the largest cluster having over 5500 keys grouped together. The mean size of the cluster was 15 keys; the median size was 6 keys. For the rest of the experiments described, we disregarded clusters with two or fewer items in them.

The largest configuration class is identified by the signature:

```
(DEFAULT)
```

```
TYPELIB
PROXYSTUBCLSID32
PROXYSTUBCLSID
```

and represents registrations for Component Object Model (COM) interfaces. A COM object, essentially a shared library, advertises the fact that it implements an interface by creating a key labeled with the interface's well-known identifier in a well-known location, at \HKLM\ Software\ CLASSES\ INTERFACE* . The PROXYSTUBCLSID32 or PROXYSTUBCLSID subkey then points to the class identifier of the COM object. For example, if a COM object with class id A implemented an interface identified as I, then it would create the key \HKLM\ Software\ CLASSES\ INTERFACE\I. It would also create the subkey \HKLM\ Software\ CLASSES\ INTERFACE\I\PROXYSTUBCLSID32, and assign a value to PROXYSTUBCLSID32 of A. Other subkeys can be added to convey optional information about the COM object.

The keys within the configuration class identified by the above signature is for interface libraries that have both 16-bit and 32-bit versions.

At the same location in the registry, we also discover a second configuration class, with 1700 instances. Even though it is stored at the same location, it differs significantly from the above, and represents 32-bit COM interfaces that do not provide a 16-bit interface library:

```
(Default)
PROXYSTUBCLSID32
NUMMETHODS
```

Also, at the same location in the registry, Glean was able to distinguish the configuration class identified by:

```
CLSID
(Default)
CURVER
```

This configuration class represents version-independent identifiers for interfaces, and is used to solve the problem of finding the latest version of a COM interface when

multiple versions are installed simultaneously. This configuration class maps from a version-independent identifier for an interface to the identifier for the newest version of the interface installed on the system. Thus, a program that does not require a specific version of an interface can use this configuration setting as a layer of indirection to look up the newest version of an interface.

Other interesting configuration classes Glean discovers includes the file type registrations for video files (AVI, WMV, ASX, MPEG, ...) and other formats. Glean also finds separate configuration classes for trust settings for various security certificates; security settings for different Internet zones; hot-fix patch descriptions; and many others. While we were not able to manually check and verify all of the configuration classes that Glean found in the registry, we did continue our spot checks across tens of configuration classes. We found that most of the configuration classes Glean found appeared to map to an extensible configuration point, where the same configuration structure was being repeated at the same location in the Windows Registry as multiple objects (*e.g.,* printers, drivers, mimetypes) were added to the operating system or an application). Other discovered configuration classes represented user-specific configuration settings spread across the Windows Registry in individual user's hierarchies. These classes included settings such as Internet Explorer's default font and color settings.

We did find a small minority of configuration classes that seemed to have been incorrectly inferred. This occurred when different keys in the registry shared similar structures, apparently though happenstance. For example, two COM object registrations, a "digital video encoder" configuration, and the configuration for the "default waveout device" all contained the subkeys `FRIENDLYNAME`, `CLSID`, and `FILTERDATA`, and were clustered together as a configuration class. This seems like a likely false positive of our configuration class clustering.

For the rest of our experiments in this chapter, we arbitrarily chose the configuration classes discovered in one of our registries as the "canonical set" of classes to use when analyzing all our registries. We chose a canonical set in order to ease cross-validation of our results across our various registry snapshots. Though we would have preferred to generate a canonical set by merging the configuration classes of many

registries, time constraints kept us from implementing this feature.

Including the I/O time to read the registry snapshot and write the configuration clusters to disk, generating these configuration clusters takes 4 minutes. The main resource constraint on our C# -based prototype is its unoptimized memory usage—it uses several hundred megabytes of memory to cluster the keys in a registry snapshot.

In addition to code optimization, there are more efficient algorithms for clustering large datasets that would improve upon our prototype's resource usage. For example, the Cure algorithm, described in [60], requires only space linear in n to perform a hierarchical clustering. Like other hierarchical clustering algorithms, Cure does require $O(n^2 log(n))$ time in the worst-case, but when data points have low-dimensionality, the time complexity is reduced to $O(n^2)$. Moreover, this algorithm allows for sampled clustering for dealing with extremely large data sets, allowing the time complexity to scale as $O(s^2)$ where s is the sample size, while still producing a quality clustering of the data.

6.6.2 Correctness constraints

After generating our configuration classes, we analyzed our registry snapshots to infer the effectiveness of size, value, reference, and equality constraints. Altogether, Glean discovered 2785 size, 2706 value, 672 reference, and 1859 equality constraints.

Both kinds of internal constraints were generated and validated across three registry snapshots. Due to functional limitations of our initial prototype, the external constraints were generated from the analysis of a single registry, meaning they are less likely to generalize well across registries. Our prototype takes 3min 40sec to generate and validate the internal constraints, and 56sec to generate the external constraints.

Of the size constraints, 238 were rules declaring that the value of a key must be empty (size=0). Some of the more notable size constraints found included that the *CLSID* subkeys (an abbreviation for the class id used to reference to COM objects) of most configuration classes had a size of 38, the correct length of a COM ID. Similar size constraints were found on keys that used different names, such as *EVENTCLASSAPPLICATIONID*, *APPID*, and *CLASSGUID*, to refer to class ids. Whereas a manually created constraint would likely only have looked for the well-known $CLSID$

and would have missed these others, Glean found all these automatically.

The value constraints show how Glean can infer clear and useful constraints on configuration settings. In one set of keys, located at \hklm\System\controlset*\services*, Glean discovers that the $TYPE$ subkey must have a value of 16 or 32, clearly refering to a distinction between 16-bit and 32-bit services. Glean also correctly generates value constraints on the perceived type and content type (or mime type) of the configuration classes for the various file type registrations described above. Glean correctly limits the video file types to being perceived as video files, and makes similar constraints on image files, audio files, compressed files, etc.

Among the reference constraints that Glean finds is one that declares that various "shell extensions" keys (that declare how files are opened in the Windows graphical interface) be limited to a class of COM registrations that provide details on context menu handlers, icon handlers, and other signatures of COM objects that are capable of handling file-related actions. Included among the equality constraints that Glean finds are all the various keys that store the host name of a machine. Glean also discovers many registry keys that store user names.

We found that almost all of the constraints that we inspected to be reasonable. But, there are corner cases that cause Glean to behave poorly. For example, if a set of default user preferences is replicated within a registry, once for each user of the machine, it can quickly pass the required threshold of support to generate a rule that incorrectly constraints these preferences.

In particular, Glean is also vulnerable to poor sampling among its registry snapshots. For example, if Glean is fed registry snapshots from machines that do not have a particular application installed, Glean will obviously not be able to generate any constraints on that application's configuration settings. Worse, if Glean is fed bad registry snapshots, it can generate constraints that are too loose, and fail to detect problems. For now, the solution is to carefully choose the snapshots that Glean bases its constraints on, though the long-term approach is to scale up Glean's analysis techniques to analyze many more registry snapshots at once and/or use representative sampling of registries, and assume only that *most* of the snapshots are correct.

6.6.3 Real-world misconfigurations: PSS logs

To evaluate Glean's ability to detect real configuration errors, we analyze a database of 43 serious registry problems gathered by the Strider project from Microsoft's Product Support Services (PSS) knowledge base and e-mail case logs on customer issues and solutions. These problems manifest in many different ways, from incorrect functionality to silent errors, and are caused by missing or corrupted registry data. These 43 problems do not include configuration errors that depend on context, such as leaving a corporate proxy setting enabled on one's laptop while traveling. For each of these configuration errors, our database includes the offending registry key; whether the key's existence, absence, or an invalid value causes the error; and natural language descriptions of the symptoms and solution to the problem. Unfortunately, this database does not include enough information for us to determine the configuration class of the key as found in our own registry snapshots. If the key does not appear in our snapshot, we pessimistically assume that Glean would not be able to determine its configuration class.

We evaluate each of these configuration errors against Glean's discovered consistency constraints. Overall, Glean successfully detects 33% (14/43) of these errors, with several being detected by multiple constraints. Our most successful constraint is the equality constraint, which detected 13 of these configuration problems. The size and enumeration constraints each detected 4 errors, and the reference constraint detected 1.

The configuration errors that Glean's constraints did not catch fell largely into two categories. The first category of errors Glean missed were errors that added or removed keys to the registry, but did not affect the value stored in a key. As Glean's constraints are mostly value-oriented, they did not notice these structural changes. This indicates that a fruitful direction of future work would be to generate constraints on the sub-structure of keys. In a preliminary analysis, we find that if Glean had included a simple constraint that the configuration class of a key be stable over time, Glean would have detected 44% of our 43 serious registry problems.

The second category of errors Glean missed were those that changed the value of keys with unknown configuration classes. Part of this problem lies in our pessimism in

assigning our configuration classes to the problem keys in our database. Glean would have detected several more errors if we had, for example, optimistically assigned configuration classes to new keys based on their location in the registry's hierarchy of keys. However, a larger problem exists when Glean's training set comes from machines without the same applications as the machines Glean is meant to guard. For example, one of the errors Glean failed to detect was in a configuration for Microsoft Money v. 11, not installed in the snapshots used to train Glean.

6.7 Summary

This chapter discussed the application of statistical monitoring to infer patterns and regularities in a system's structure, for the purpose of helping operators better understand a complex system, and to aid in detecting problems in the system.

We studied this problem in the specific context of the Windows Registry, a configuration database of hundreds of thousands of configuration settings. While each individual settings is strongly typed with a simple data type, there is no explicit description of the complex data structures that govern the organization of the configuration database. We applied data clustering to this hierarchical structure, and automatically inferred data type definitions for over a thousand configuration classes, from DLL registrations, file handlers, drivers, and printer settings. We used these learned configuration classes as a building block for expressing configuration constraints on the structure of the registry; these constraints, in turn, were able to detect a third of the most common Windows Registry errors.

Chapter 7

Correlating faults to causes

As described in Chapter 2, today's Internet services (e-commerce, search engines, enterprise applications and others) commonly suffer from brown-outs, application-level failures and other problems, resulting in the failure of user requests. It is critical to quickly determine the source of such problems to reduce the overall downtime of the system.

The focus of this chapter is applying statistical monitoring to fault localization: isolating the likely root causes of system failures. Current techniques often rely on operators to input extensive knowledge about the system and how faults might propagate through it, or to dynamically drill down and track errors through the layers of the system [13, 116]. But the more complicated the system is, the more difficult it is to understand and drill down through the system.

The need for fault localization arises when the only detectable symptom of a failure is an end-to-end failure. In such a failure, the operators of an Internet service do not see symptoms of a problem close to its cause, such as a machine crashing in the backend of the system. Instead, the signs of failure occur far from their root cause. In the extreme case, faults are not detected within the system's boundaries at all, and are only noticeable, for example, as timeouts at test clients or web server front ends. The challenge then is to trace back from these symptoms of a failure to the cause of a problem.

Automating a complete solution to diagnosing the root cause of a problem is

well beyond our grasp—even the phrase "root cause" has thorny philosophical consequences: Is the root cause of a performance failure a faulty piece of hardware? or a poor procurement process for better hardware? or is the true root cause a poor design and testing process at the hardware manufacturer? The influences on Internet service development and evolution occur at many scales and time periods. We therefore distinguish fault localization, the process of finding *where* a problem lies in a large system, from *fault diagnosis*, the process of finding what the root cause of a problem is and why that problem is occurring, and focus our efforts on fault localization.

7.1 Challenges

The fundamental challenge of fault localization is that problems often appear as *end-to-end failures* in the operation of the system as a whole, without causing obvious failures in the system's pieces. Simply noticing that something has gone wrong is not enough to tell us where to look to fix it. Other times, a problem's symptoms are simultaneous errors in many components, and isolating the source of the problem from its cascaded failures is the challenge. In both cases, we are trying to take a set of symptoms occurring across the system, and use them to point us back toward the cause of the problem elsewhere.

Fault localization is difficult because we rarely fully understand the dynamic behavior of a system well enough to trace back from the symptoms of a failure to their underlying cause. Most fault localization techniques, including event correlation systems, are based on static dependency models describing the relationships among the hardware and software components in the system. These dependency models are used to determine which components might be responsible for the symptoms of a given problem [16, 36, 58, 132]. The first major limitation of traditional dependency models is the difficulty of generating and maintaining an accurate model of a constantly evolving Internet service. Their second major limitation is that they typically only model a logical system, and do not distinguish among replicated components. However, since large Internet services have thousands of replicated components, there is a need to distinguish among them to find the instance of the component that is at

fault.

7.2 Approach: Correlation

Ideally, we would be able to directly observe the fault propagation paths that connected the symptoms of a failure to the location of its cause. Given the myriad ways that failures can propagate, through direct and indirect interactions, data and control channels, and both logical and physical resource dependencies, tracking the exact propagation path, regardless of its form is not practical. Instead, we recognize that the runtime path of a client request naturally encapsulates many of these fault propagation paths, such as the data and control channels, direct interactions, and resource dependencies[1]

For each request, we record the set of components used to service it, together with its believed success or failure, based on whether or not it exhibited any symptoms of a failure. Then, to help us trace back to the location of a failure, we analyze many requests together, and search for a correlation between the failing requests and the features of their runtime paths through the service. In other words, we look for components that are part of the failing requests, but not part of the successful ones. If we find a correlation between failures and the use of particular components, then we say that these components are a potential cause of the failure.

Of course, correlation does not imply causality, but it does help us narrow down our list of possible causes. Another trap we must be aware of using a correlation technique to identify a component which is correlated with the detection of the failure, and not the failure itself. For example, if our detection mechanism only detected faulty requests when they happened to use a particular component $A_{falsesuspect}$, then our correlation algorithm might falsely identify $A_{falsesuspect}$ as the cause of the failure. Of course, if this component is used in successful paths as well the faulty requests, then it will likely not have a high correlation to the failures.

A primary assumption of our approach is that requests are generally independent, in that a fault occurring in one request does not directly cause the failure of a separate

[1]As defined in Chapter 4, a runtime path is the ordered set of coarse-grained components, resources, and control-flow used to service a client's request.

request. For example, if some bad request corrupted the state of a component which caused all future requests using that component to fail, our approach would be able to localize the corrupted component, though not the original bad request. In contrast, if a bad request directly caused another request to fail, without first causing a failure in a more persistent intermediary, then our approach likely would not be helpful.

Also, in attempting to find a correlation between component usage and request failures, we are assuming that our traces of components are already capturing the root-location of a fault. That is, our approach simply narrows down an existing list of system components to the likely cause of a failure. In contrast, our correlation-based approach would not be as useful if the cause of a failure was in a part of the system not captured in our tracing infrastructure.

As in fault detection, tracing real requests through the system enables us to easily adapt our analysis to dynamic systems and rapidly evolving systems where using static dependency models is not practical. Moreover, using an automatic analysis allows us to consider a much larger volume of observations than a person manually monitoring the same system would be able to take into account.

We have evaluated two different correlation algorithms. We presented the first algorithm, data clustering in joint work with Mike Chen [33]. While the results of data clustering showed that statistical monitoring was better than alternatives such as stack analysis, our second algorithm, based on using decision trees for correlation, is more robust and better performing. We present the use of decision trees in this thesis.

We learn a decision tree to classify (predict) whether a path shape is a success or a failure based on its associated features. These features correspond to the path information that Pinpoint collects, such as the names of EJBs, IP addresses of server replicas in the cluster, etc. Of course, we already know the success of the requests we are analyzing—what interests us is the structure of the learned decision tree. Examining which components are used as tests within the decision tree function tells us which components are correlated with failed requests. In our experiments, we use the ID3 algorithm for learning a decision tree, although recent work suggests that C4.5 decision trees might perform better [30]. More background information about

decision trees is presented in Chapter 2.

The training set for our decision-tree learning is the set of paths classified as successful or failing by our fault detector. The input to our target function is a path, and the output of the function is whether or not the path is failing. Our goal is to build a decision tree that approximates our observed failures based on the components and resources used by the path.

Once we have built a decision tree, we convert it to an equivalent set of rules by generating a rule for each path from the root of the tree to a leaf. We rank each of these rules based on the number of paths that they correctly classify as anomalous. From these rules, we extract the hardware and software components that are correlated with failures.

Note that decision trees can represent both disjunctive and conjunctive hypotheses, meaning that they have the potential to learn hypotheses that describe multiple independent faults as well as localize failures based on multiple attributes of a path rather than just one, *i.e.,* caused by interactions of a set of components rather than by individual components. More interestingly, it allows us to at least partly avoid specifying *a priori* the exact fault boundaries in the system. With the caveat that this will require more observations to make a statistically significant correlation, we can allow the decision tree to choose to localize to a class of components, a particular version of a component, all components running on a specific machine, etc., rather than stating before-hand that we want to localize to a particular instance of a component.

7.3 Decision tree evaluation

In Chapter 5, we evaluated the ability of Pinpoint, our prototype fault monitor, to detect anomalies when a failure occurs. Here, we analyze how well the fault localization aspect of our prototype is able to determine the location of a fault within the system. This evaluation looks both at how well our decision tree algorithms localize faults given a perfect fault detector, and how well our algorithms localize faults given the results of Pinpoint's anomaly detection.

For this evaluation, we use the same Internet service testbed described in Section 5.3. We use the runtime path traces as recorded during our fault detection experiments, and use these traces as the input to our fault localization algorithm. Separately, we use our knowledge of fault injections and our various fault detectors, including the results of our path shape monitor, to mark individual requests in our traces as successful or faulty.

7.3.1 Metrics

We continue to use the metrics of recall and precision to measure fault localization ability. After localizing a fault, a fault monitor returns a set of components suspected of causing the failure, ranked by the degree to which we believe each component might be responsible for the failure. We simplify our evaluation by ignoring this ranking, and simply consider all statistically significant suspects as equal.

In this context, with only one fault injection per experiment, recall becomes a Boolean metric, indicating whether the faulty component is a member of the suspect set. Precision measures how many innocent suspects there are in the suspect set.

7.3.2 Results

The overall results of our localization tests comparing Pinpoint's detection and localization techniques to each other are shown in Figure 7.1. In this figure, we show how well our decision-tree based localization and our component interaction based localization fare in our experiments.

We show the results from applying our decision tree algorithm to three variants of faulty request data, each showing how well the decision tree fares as the requests classified as faulty become more and more "noisy," and compare them to the localization achieved through component interaction analysis. In the first variant, we mark only the requests into which we directly injected failures as having failed ("Injected failures"). Intuitively, this provides the cleanest data for localizing the primary cause of a failure, as it ignores cascading failures and has no statistical error in the fault detection mechanism. These results are the most competitive with the localization

rates of our component interaction analysis—the only false suspects that occur are due to the structure of the application itself. For example, the decision tree cannot distinguish between two components that are always used together. Also, there exist components which appear to correlate very well with some failures—without having any causal relation with the failures—hiding the true cause of a fault.

The second variation of our request data marks requests as faulty if we have either injected them with a fault or if they have been affected by a cascading fault ("Injected and cascaded failures"). These results are noisier, and introduce false suspects as our algorithms attempt localize both the primary and cascaded fault simultaneously.

The last variation of our request data marks requests as faulty or successful based on the results of our path shape monitor, described in Chapter 4 ("Path shape detection"). The results of analyzing this data show the highest miss rate, as we contend with noise both from the PCFG selection mechanism and the cascaded faults.

Not represented in Figure 7.1, but worth noting, is that in our clustered experiments with Petstore 1.1, is that the decision tree algorithm was able to narrow down the location of the faulty component even when it was unable to pinpoint the exact component instance that was failing. In these cases, it was able to point to either a class of components containing the faulty component or to the machine running the faulty component.

Finally, Figure 7.1 also shows the localization ability of our component interaction analysis-based fault detector. Since component interaction analysis directly identifies a suspected faulty component, there is no second analysis step to localize a failure. In our experiments, component interaction analysis did quite well at discovering the location of a fault. This implies that for many faults, symptoms of the failure are likely to occur close near their cause.

From this, we conclude that a decision tree's ability to localize failures depends heavily on the noise in the traces, confirming a well-known limitation of statistical models on noisy data. Here, the decision tree's localization capability drops when we add cascaded failures and false positives from runs of the path-analysis algorithm. This indicates that heuristics for separating primary from cascaded faulty requests, such as picking the first anomalous request from each user's session as the primary

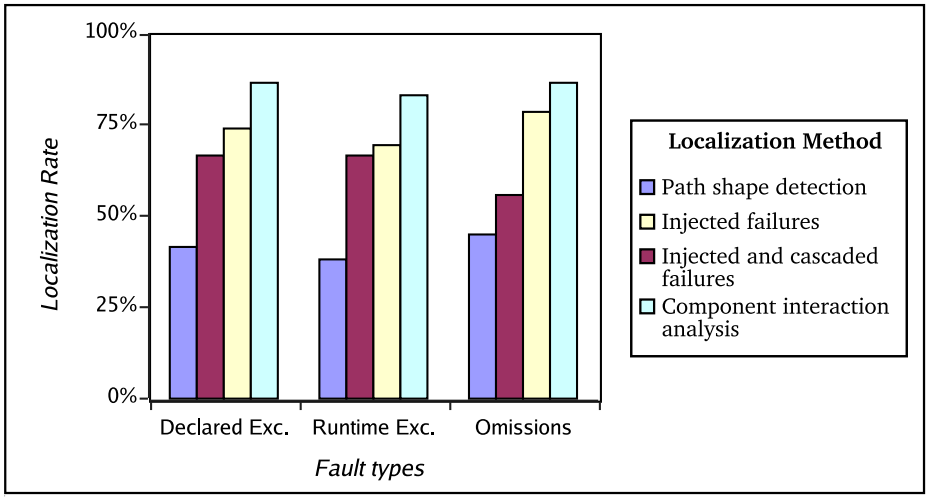


Figure 7.1: **Localization recall of our decision-tree fault localization per fault type.**

fault, are likely to improve the performance of decision-tree localization in real deployments. We discuss the problem of statistical modeling and data quality further in Section 9.1.

7.4 Limitations

One limitation of our Pinpoint localization prototype is that it cannot distinguish between sets of components that are tightly coupled and always used together. In the Petstore application, we found sets of components that are always used with the components into which we injected faults. As a result, Pinpoint reports a superset of the actual faulty components. To better isolate faulty components and improve precision, one potential technique is to modify the system to randomize some interactions by modifying load balancers or occasionally breaking affinity to ensure that components are used in many different combinations. In replicated systems, another option is to randomize the interactions between components by, for example, round-robining a front-end component's calls to a set of replicated backend components.

Another limitation of Pinpoint is that it does not work well when faults cascade, such as when a fault in one component corrupts the state of a neighboring component

and permanently affects its functionality. In this case, subsequent requests using the second component will fail, without showing any dependency on the root cause of the problem. For example, a user will not be able to login if the component responsible for creating new accounts has stored an incorrect password. In this case, the state "corruption" induced by the account creation request may be subsequently localized to the password-verification component. One potential solution is to extend the request tracing to account for shared state by logging what persistent state was read or written by each request. This would allow us to correlate failures across requests through their common state dependencies.

A third limitation derives from Pinpoint's application-generic monitoring. Since Pinpoint has no application-specific knowledge about requests, deterministic failures due to pathological inputs cannot be distinguished from other failures. For example, a user may have a bad cookie that consistently causes failures. One possible solution is to extend Pinpoint to record attributes of the requests themselves, and use them as another possible factor in distinguishing between failed and successful requests.

7.5 Summary

This chapter discussed the application of statistical monitoring to the problem of fault localization. We prototyped a statistical monitor in the context of Internet services specifically, and evaluated its results in the context of our Internet service testbed.

We found that by monitoring the same runtime paths we had been observing for the purposes of detecting failures, as in Chapter 4, we were often able to effectively trace backwards from the visible symptoms of a failure to its underlying cause within the system. As in our previous applications of statistical monitoring, we were able to attack the fault localization problem without having an *a priori* detailed understanding of how faults might manifest or propagate through they system, taking advantage only of the two simple assumptions. First, that the general location of a fault was visible to our tracing system and, secondly, that requests to the service generally failed independently, that is, a fault in one request does not directly cause a failure in a separate request.

Chapter 8

Integration with Fault Management Processes

This chapter describes how automated fault detection and localization may integrate into the overall fault management process at an Internet service. Strictly speaking, how automated fault management techniques may be integrated into the management process of any system is a policy matter, separate from the techniques presented in Chapters 4 through 7. However, we are also interested in the broader results of applying statistical monitoring to the overarching goal for which it was originally conceived: lowering the mean time to recover from failures in Internet services.

We begin by discussing general issues that may arise in a variety of situations. Then, in the second half of this chapter, we present and evaluate an autonomous recovery system that integrates automated fault detection with automated repair, and discuss some of the specific details we confronted during the integration.

8.1 Integration with fault management process

Our purpose in investigating statistical monitoring is to reduce the overall time to recover from failures and improve the reliability of Internet services. To fulfill this goal, we must at some point integrate automated statistical monitoring into the overall fault management process at an Internet services. However, we do not advocate the

dramatic step of replacing existing manual and semi-automated processes wholesale. Though a slightly non-technical issue, one of the primary concerns with integrating an automated analysis into an existing fault management process is convincing operators to trust the analysis. To help build this trust, we believe it is prudent to follow a cautious approach of augmenting existing fault management processes by using statistical monitoring in parallel with existing techniques.

In exploring the three problems of extracting system structure, detecting faults, and localizing faults, we have implicitly assumed a staged process of fault management, where the outputs of one stage of the process are the inputs to the next stage. For example, the results of our path-shape analysis for fault detection feed into our fault localization algorithms. While this is not the most precise characterization of the fault management process—consider our component interaction analysis, which contains elements of both fault detection and localization—it is a useful one, and we continue to use it in this section.

Integrating statistical monitoring into existing fault management processes involves many of the same issues that occur when making any change to a fault management process. Inherently, the designers of a system must make a policy decision that takes into account the deployment and maintenance costs of the change, the reliability of a new technology or functionality, and how the new technology will affect the time-to-recover from failures and the impact of failures. Designers must be careful to take into account the worst-case scenarios for improving or worsening the *overall* reliability of the system. They must also consider how the new technology will be perceived by the people who will be interacting with it, including whether operators are likely to understand and trust it.

While most of these considerations, as well as the final policy decision, are necessarily environment- and system-specific, statistical monitoring has the potential to be a powerful tool within a fault management process. Because a statistical monitor itself operates largely automatically and can update its models as the underlying system changes (*e.g.,* without requiring people to describe the change), the maintenance costs associated with the system are low, and the deployment costs are limited to the setup of the observational infrastructure. A statistical monitor's *automatic* analysis

of a system's behavior mean that the monitor is likely to reduce the time it takes to notice patterns and changes in the system's behavior, thus potentially reducing the time-to-recover as well. The use of interpretable statistical techniques can provide operators with an explanation of the monitor's results, contributing to the monitor's understandability and operator confidence in the monitor.

How a statistical monitor affects the overall reliability of a system is highly dependent on how the fault management process is designed to react to the results of a statistical monitor. There are three broad options for reacting to a statistical monitoring stage, whether it is statistical monitor for fault detection, localization, etc.

1. If we are automating multiple stages of fault management, we can choose to actively push the results from one automated analysis to the next stage of automated analysis. For example, in the case of fault detection, an alarm might immediately trigger a fault localization analysis and then an attempt at repairing the failure. A fully automatic process is of benefit when a safe, automated response is available and the improvement in performance and reliability due to the speed of the automated response is greater than the performance and reliability cost of false positives. Practically, this means that each automated stage of the fault management process should be cheap, safe, and likely to succeed in a reasonable proportion of cases.

2. We can choose to actively push the results of a stage to human operators, for them to handle the next stage of analysis or repair. This may be best when an automated response to an analysis result is not available or would be unsafe. However, since the cost of a false positive, requiring the attention of a human operator, is high, the threshold for sending an alarm or report to a human operator should also be relatively high.

3. We may make the results of a statistical monitor passively available, where human operators may look at them if another event triggers their interest in the current state of the monitored system. For example, if operators receive complaints from users, they may look at the current state of the monitor to

help direct their attention to the most likely problem spots. Making the results passively available rather than automatically responding or actively requesting the attention of a person may be the best option if there is both no cheap and safe automatic response, and the analysis generates too many false positives for a person to look at each one.

Note that these three scenarios are not mutually exclusive. We can treat each option as a distinct response to a statistical monitor's analysis, and set a separate threshold for triggering each response. For a given response, our threshold for triggering it will be based on the expected utility of triggering the response, taking into account the rate of false positives and the cost and safety of the response. For example, if we have a statistical monitor for fault detection, we can choose to always make the current results of the fault detector passively available to operators, effectively setting the threshold for triggering passive reporting to zero. At a higher threshold, if the fault detector notices a potential problem—even only a minor problem—then the detector may trigger a rapid response of fault localization and attempted repair. Finally, if this failure appears to be a major problem, the highest threshold of alarm, then a human operator may be notified. By using all three integration options simultaneously but applying separate thresholds of action for each option, we can effectively take selective advantage of automated analysis, enjoying the benefits while avoiding the disadvantages. Moreover, during initial deployment, one can integrate a statistical monitor with relatively high thresholds (or even infinitely high thresholds) for triggering automatic or manual responses, and slowly lower thresholds if and when the monitor proves itself to the human operators of the system.

8.2 Autonomous recovery

In this section, we explore the issues involved in building the fully autonomous fault management process. That is, we consider the scenario where a fault detector automatically triggers a cheap and safe response to a potential problem.[1]

[1]The autonomous recovery prototype discussed in this section is joint work with George Candea, and was developed with the help of Shinichi Kawamoto, Pedram Keyani, and Steve Zhang. It appeared in [25] and [24].

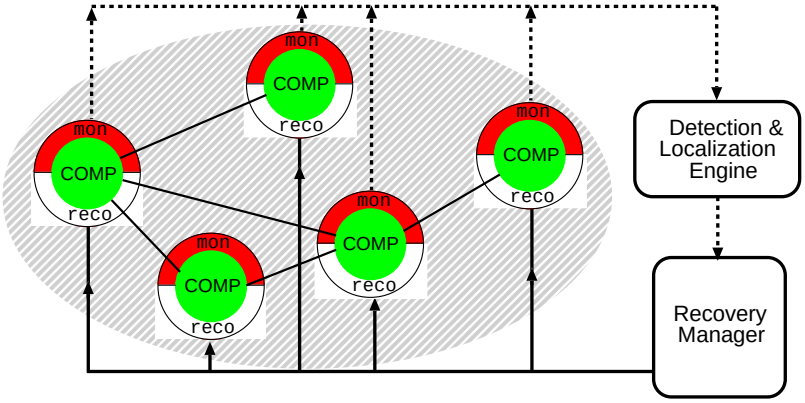


Figure 8.1: **A design pattern for autonomous recovery.** Each component is wrapped with a monitor and a recovery container. The monitor reports on a component's behavior to a centralized detection and localization engine. The recovery container is responsible for restarting and cleaning a component's state to recover from transient software problems.

Our design for autonomous recovery is illustrated abstractly in Figure 8.1. One of the lessons we learned while building this prototype was that we could not directly combine a fault detection and localization system with a recovery system, but needed an additional recovery manager to mediate between our detection and recovery subsystems. For example, without a recovery manager, our fault detector believes a component in the middle of recovery is still failing, because its behavior is different from normal. Thus, the fault detector requests the "faulty" component be recovered again. This process repeats, never allowing the component the time to stabilize its behavior and function properly. A separate recovery manager can have the end-to-end knowledge required to damp this behavior, as well as to provide a vehicle for implementing a variety of policies.

In addition to using statistical monitoring to detect failures in component interactions, as described in Chapter 4, we add a recovery container around each component. These recovery containers implement a microreboot strategy, as described in [23], which selectively reinitializes a component's in-memory state and control flow, thus repairing many software faults, including data structure corruptions and deadlocks.

8.2.1 Recovery manager and policy

The recovery manager keeps track of the previously-rebooted subset of components and, if the new subset is the same, it chooses to restart the entire application instead of again restarting that subset of components. If the problem persists even after rebooting the entire system, the policy module can notify a system administrator by pager or email, as necessary. Of course, as noted earlier, a reasonable policy would be to wait for a high confidence of failure before triggering a relatively high cost action such as notifying an human administrator.

Our prototype policy only employs two levels of rebooting (microreboot followed by a full reboot), but more levels could be used for other kinds of applications. The policy employed in our autonomous recovery manager is as follows:

1. Given a received failure report, ignore all components with a score of less than 1.0 (since the scores are normalized, 1.0 is the threshold for statistical significance).

2. The existence of several components with a score above 1.0 indicates something is wrong: either one component is faulty and the other ones appear anomalous because of their interaction with it, or indeed we are witnessing multiple simultaneous faults. We choose to act conservatively and microreboot all components that are above the 1.0 threshold.

3. For the next interval of time Δt, failure reports involving the just-rebooted components are ignored, since it takes a while for Pinpoint to realize the system has returned to normal. In our experiments in this chapter, Δt was set to 30 seconds.

4. If subsequent failure reports (after Δt) indicate that the just-recovered components are still faulty, we can either repeat the reboot-based recovery a number of times, or directly proceed to restarting the entire application. Our current policy implements the latter.

After a full application restart, if the problem persists and appears to be a significant problem, then an administrator should be notified.

A detailed analysis of the various policies and the trade-offs involved is beyond the scope of this chapter; exploring the space of possible policies is an important piece of future research. For example, in other types of systems, it may be better to just microreboot the top n most anomalous components (for some parameter n). Or, it may be preferable to microreboot these components serially, instead of all at once. We have purposefully built our recovery manager such that new policy modules can be plugged in, to encourage further research on the topic. An interesting problem we have not addressed is dealing with faults that keep reappearing, either because they are triggered by a recurring input, or are simply deterministic bugs.

One final issue is that of recovery manager availability: should the recovery manager go down, nobody will be watching over the system. We have built the recovery manager such that it can restart quickly. After a restart, the manager will have lost its recent history, and this may introduce a period of vulnerability to, for example, spuriously rebooting a just-rebooted component. However, this period of vulnerability will pass in time Δt. The recovery manager can be run in a simple infinite loop to restart upon crashing, or another part of the system, such as the application server, could be responsible for watching the recovery manager and restarting it if it appears to have failed.

8.2.2 Evaluation

We implemented our approach in the same testbed and fault injection environment that we described in Chapter 5, using the JBoss implementation of the J2EE middleware standard, and deploying the RUBiS auction application atop it. The version of RUBiS we used in this experiment was modified to use the Session State Management (SSM) data store for semi-persistent data in order to enable microrebooting [93]. We placed the web server and application middleware on an Athlon 2600XP machine with 1.5 GB of RAM; the database, Pinpoint and SSM were each hosted on Pentium 2.8 GHz nodes with 1 GB of RAM and 7200 rpm 120 GB hard drives. The client simulator ran on a 4-way P-III 550 MHz multiprocessor with 1 GB of RAM. In choosing the number of clients to simulate, we aimed to maximize system utilization while still getting good performance; for our system, this balance was reached at 350

concurrent clients for one application server node. All machines were interconnected with a 100 Mbps Ethernet switch and ran Linux kernel 2.4.22 with Java 1.4.1 and J2EE 1.3.1.

The metric we use to evaluate the impact of failure and recovery on end users is *action-weighted goodput* (G_{aw}). G_{aw} aims to count the requests which make (or do not make) a successful contribution to a user's high-level action. Such an action may be a search for an item on an e-commerce site, the completion of a purchase, or retrieval of an e-mail message. An action may consist of several HTTP requests, and G_{aw} considers these requests to be successful only if all requests are successful. In other words, if a user cannot complete an action, all the action's requests are considered to be lost work. Thus, G_{aw} accounts for the fact that the damage caused by a request failure depends on context of a user's task. Also, G_{aw} captures the fact that tasks with more actions often correspond to greater amounts of work from the user's point-of-view.

In Figure 8.2, we illustrate the functioning of the integrated system in reaction to a single-point fault injection; this is representative of the reaction to the other categories of faults we injected. Each sample point on the graph represent the number of successful and failed requests during the corresponding 1-second interval. Requests appear to be "failing" prior to the fault injection point because of G_{aw} 's accounting: if an HTTP request fails, then all past requests within the user's task are marked as failed as well, to reflect that the user's work has been lost.

In Figure 8.3, we zoom in on the interval between 332 and 360 seconds, to analyze the events that occur; the horizontal axis now represents seconds. We mark on the graph the points (along with the time, to millisecond granularity) at which the following events occur: we inject the fault (t_1), then the first end user request to fail as a consequence is at t_2, the Pinpoint analysis engine sends its first failure report to the recovery manager (t_3), the recovery manager decides to send a recovery command to the microreboot (t_4), the microreboot is initiated (t_5), and finally the microreboot completes (t_6) and no more requests fail.

The system recovers on its own within 19.4 seconds of the first end user failure. This time compares favorable to the recovery times witnessed in Internet services

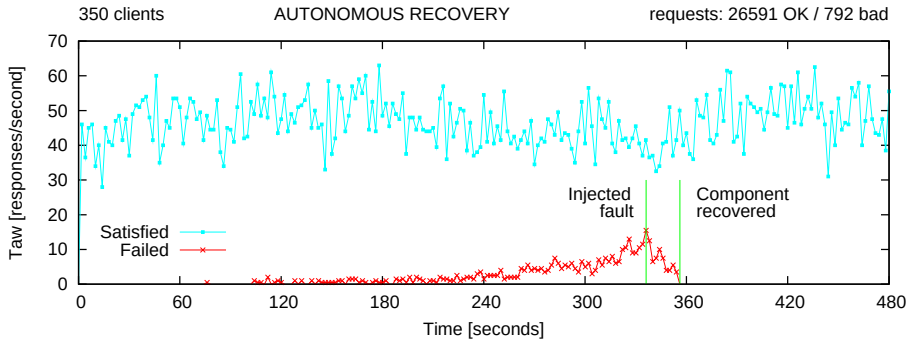


Figure 8.2: **Timeline of autonomous recovery.** We corrupted an internal data structure in SB_ViewItem, setting it to null, which results in a NullPointerException for SB_ViewItem callers. Labeled light-colored vertical lines indicate the point where the fault is injected and where the faulty component completes recovery, respectively.

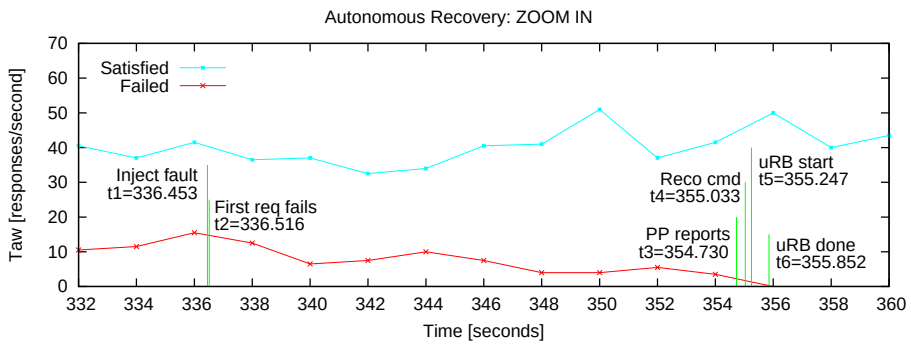


Figure 8.3: **Detail of autonomous recovery timeline.** Zooming in on the time interval [332, 360]. Recovery time is 19.4 seconds from the time the first end user notices a failure; of this interval, 18.5 seconds is spent by Pinpoint noticing that *SB_ViewItem* is faulty.

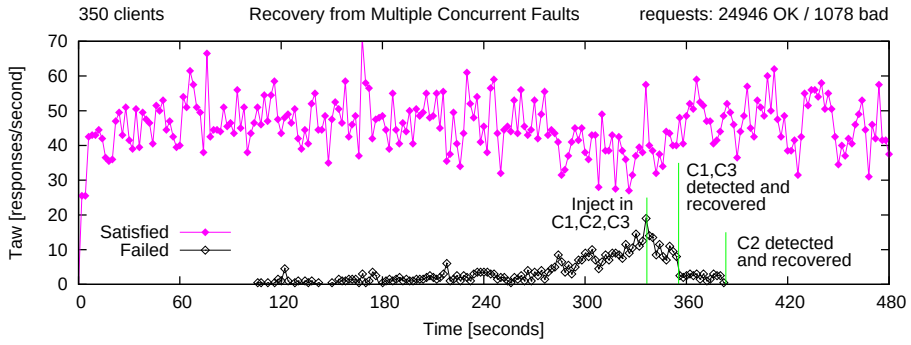


Figure 8.4: **Timeline of autonomous recovery from multiple failures.** We simultaneously injected a data corruption fault in SB_PutBid (C_1), a Java Exception fault in SB_ViewUserInfo (C_2), and a Java Error fault in SB_SearchItemsByRegion (C_3). The more frequently called C_1 and C_2 are automatically recovered within 20 seconds, while the less often used C_3 is recovered 46 seconds after injection.

involving human assistance; recovery there can range from minutes to hours.

Finally, in Figure 8.4, we show the result of a multi-point injection: three different faults in three different components, respectively. Our system notices and recovers two of the components within 19.9 seconds, and the third component 46.6 seconds after the injection. The reason for the delay is that, in our workload, the first two components are called more frequently than the third. Thus, the Pinpoint analysis engine receives more observations sooner, letting it achieve statistical confidence of a problem sooner.

These results show that combining a statistical monitoring approach to fault detection with microreboots, a general fault recovery technique, allows us to build an application server that autonomously recovers from a variety of component-level failures. Since the failures detected by our statistical monitor are not exactly the same set as those recoverable by microreboots, there will be a diminishment in the size of

the failures that can be autonomously recovered end-to-end, as compared to the class of failures detectable by our monitor or the class of failures recoverable through microreboot. However, we found our system to be effective in detecting and recovering from realistic transient faults, with no *a priori* application-specific knowledge.

While, as described in Chapter 5, we have not observed our statistical monitor to report false alarms, they certainly are a possibility. However, cheap recovery makes the cost of these false positives negligible, making autonomous recovery a good first line of defense. Even if the microrebooting ends up being spurious or ineffective, it does no harm to try, and has the benefit of simplifying fault management in many cases.

8.3 Summary

This chapter presented a preliminary investigation of integrating statistical monitoring into a broader fault management process. We first described various integration options for combining statistical monitoring into an existing process, and discussed the considerations for deciding whether to automatically respond to reports from a statistical monitor, actively send reports from a statistical monitor to a human operator, or simply passively provide the reports for human operators. We advocate using each of these options simultaneously, choosing different sensitivity thresholds before exercising each response, in order to balance the potential benefit of a particular response versus its cost and the likelihood of false alarms.

In the second half of this chapter, we described our experiments combining statistical monitoring for fault detection with a general mechanism for quickly and safely recovering from failures. We found that, with the addition of a simple recovery manager to mediate between our fault detector and recovery mechanism, we were able to autonomously recover from a wide variety of failures. Such autonomy, occurring in "machine time" rather than "human time," improves the reliability of a service by (at least temporarily) repairing problems within seconds. This autonomy is particularly useful for large scale systems where there are a small number of administrators managing thousands of machines. While our implementation exploited particular properties

of J2EE programming, such as the existence of well-defined components and explicit state management, we believe the techniques presented here can be applied more generally to non-J2EE systems as well.

Chapter 9

Discussion and future work

The application of statistical techniques to problems of fault management has significant benefits, such as avoiding a need to deeply understand the semantics of the monitored application. However, there are also challenges when using statistical techniques. In this chapter, we discuss these issues and how they have affected our work across our application of statistical monitoring to fault detection, inferring of system structure, and fault localization. We end the discussion in this chapter by presenting avenues for future work.

9.1 Data quality

Statistical monitors are severely limited by the quality of the data being analyzed. If the observed behaviors or structure are inappropriate proxies for our purposes, our sample of observations is not representative of the breadth of all observations, or our observations are too noisy, then the results of an analysis will suffer.

Inappropriate proxies: In our monitoring of activity and state metrics for the session state manager (SSM), discussed in Section 4.5, we found that a number of the initial metrics we planned to monitor were inappropriate proxies for the properties of the system we cared about—namely whether or not the system was working. For example, CPU usage was simply too variable in this system for us to meaningfully compare across the nodes in our cluster. Monitoring this metric would have, at best,

had no effect on our monitoring performance and, at worst, would have added false positives.

A similar issue occurred when we applied our path-shape and component interaction analyses to our observations of a remote-method-call-driven J2EE application, described in Section 5.8.1. In that case, our assumption that every request to the system was a largely independent unit of work was broken by a system whose workload was driven by many intertwined remote method invocations. Breaking this assumption meant that our observed component interactions and path shapes no longer represented the important application-level behavior of the service, and our ability to detect and localize failures in this system was poor.

Unrepresentative data: While monitoring for faults, an anomaly detection approach assumes that an anomalous behavior is an indication of a change in the system's functionality, such as due to a failure. However, if the observations being used to generate the monitor's models of acceptable behavior do not capture most or all of the acceptable behaviors of a system, then our models of acceptable behavior will necessarily be incomplete, and cause many acceptable behaviors to marked as anomalous instead.

In the context of inferring system structure in the Windows Registry, we found the same problem of unrepresentative data lowering the effectiveness of our statistical monitoring. In our experiments, we inferred extra structural information from observations of a number of sample Windows Registries. However, some of the problems that we found occurred in the real-world Product Support Services (PSS) logs included misconfigurations of applications not installed on our sample machines. This effectively limited our ability to reason about and detect problems in the configurations of these applications. If our training data had included a wider sample of registries and included registry information from these machines, we might have been able to detect more of the frequent real-world misconfigurations represented in the PSS logs.

Noisy data: When capturing data from a real system, we cannot expect our captured data to be a perfect record of the system's behavior. Whether because of skews in the clocks of distributed nodes, software bugs in the instrumentation,

external interference on our observed behaviors, or the dropping of observations due to excessive load, our observations of a system will contain noise. This noise can affect the ability of a monitor to make statistically significant assertions.

We saw the effects of noisy data most significantly in our fault localization experiments in Chapter 7. As we added various levels of noise to our labels of faulty and successful requests, we saw a decrease in the efficacy of our fault localization algorithms. We also saw the effects of noise in our fault detection monitoring in our monitoring of Amazon.com. Background noise in our observations, caused by heavy sampling and almost constant changes in the workload and behavior of the system caused our analysis to show a small but persistent level of anomalies in our all our experiments. The effect was to make it more difficult to tell whether a spike in anomalies was likely to represent a real failure until it reached a high threshold.

9.2 Algorithmic considerations

In parallel to questions about data quality, we must question how robust a particular analysis algorithm is to the potential problems of data quality. Choosing the wrong data to observe and monitor is something that any algorithm would be hard-pressed to compensate for. However, classification and anomaly detection algorithms should be robust to a certain level of noise, and learn and generalize from system observations while avoid overfitting to the data. For example, if a fault monitor believed that only the exact behaviors it had already seen were acceptable, then it is likely that the fault monitor would raise many false alarms. On the other hand, if an algorithm generalizes too much, it might accept too many behaviors as normal and not detect true failures. There are several techniques, including cross-validation methods and lowered sensitivity thresholds, that can help to avoid or mitigate issues of overfitting. For more details on overfitting in statistical techniques, see any standard machine learning textbook, such as [44, 62].

We must also consider the statistical assumptions that a particular algorithm makes on the underlying data being observed. In our work, we have explicitly chosen

to use distribution-free or parameter-free algorithms which make minimal assumptions on the statistical distribution of the underlying data. This is sensible in our environment because we have no reason to believe that the behaviors we observe follow any particular distribution. The effect of choosing distribution-free algorithms is to reduce the statistical confidence of our analysis as compared to analyzing the same data with an algorithm which made accurate assumptions about the data's distribution. Of course, if those assumptions were violated, then the confidence of the algorithm might be misplaced.

In addition to assumptions on the statistical properties of the data being analyzed, analysis algorithms and data models can make other, more subtle assumptions as well. As one example, our modeling of component interactions assumes that a class of components has only a single mode of behavior. As discussed in Section 5.8.2, we found that when monitoring some lower-level components this assumption was violated. As a result our component interaction models did not accurately represent the behavior of these components, and erroneously marked them as faulty when they were not.

We must also recognize the fundamental limits of statistical techniques to automate the process of reasoning about an application's behavior. For example, simply because an application's behavior is anomalous or has changed does not mean it has failed. Similarly, just because a component is correlated with a failure does not mean the component is the cause of a failure. While it may be convenient to act as if an analysis has detected or failure or localized the cause of a fault, we must tread carefully and ensure that the benefit of acting correctly when our convenience was warranted is not overwhelmed by the cost of acting wrongly at other times.

9.3 False positives and other mistakes

Intuition might suggest that minimizing the false positive rate of a detection mechanism while maintaining a high or perfect true positive rate would result in a reliable and useful fault detector. [5] refutes this intuition, declaring it to be the *base rate fallacy*. The refutation is that, when looking for rare events, such as failures, any

non-zero false positive rate will overwhelm a detector even if it detected all true failures perfectly. In the context of computer security, [5] argues that this makes intrusion detection systems based on anomaly detection unusable in practice.

In the context of our broader project, Recovery Oriented Computing, we argue that false positives are only a problem when dealing with them is expensive or unsafe. We advocate making the cost of online repair for failures sufficiently low, such that a reasonable degree of "superfluous recovery" in response to false positives will not incur significant overhead.

We advocate a cheap, safe first response to a failure, such as reboots of nodes in the system or microreboots of individual software components [23]. Rebooting and microrebooting are both successful at recovering from a wide range of failures by returning the faulty component to a known good state. They recover from corruptions of soft-state, memory, resource leaks, deadlocks, and other software transient problems, though they do not help in cases of persistent state corruption or permanent hardware failures. Automatically tying some form of rebooting as a response to the detection of a fault avoids the cost of involving a human operator in the case of a false alarm, but also sharply reduces the time to recover in the case of a true failure. We have successfully demonstrated the autonomic recovery both in the context of J2EE middleware [24] in which microreboots make it almost free to recover from transient failures; and in two storage subsystems for Internet services [70, 93] where, in response to Pinpoint's non-structural behavior monitoring, any replica can be rapidly rebooted without impacting performance or correctness. Cheap recovery has another benefit for fault detection as well: when false positives are inexpensive, it is reasonable to lower detection thresholds to catch more faults (and more false positives), and potentially catch problems earlier.

When operators notice that a Pinpoint monitor is reporting semantic false positives, *e.g.,* because of a major software upgrade to the system, retraining Pinpoint's models is equally cheap and safe. Retraining Pinpoint takes on the order of minutes in our experiments and is tied primarily to the time it takes to observe most of the system's functionality being exercised. The safety cost is a small window of vulnerability during the time of retraining. However, depending on the severity of the

change to the underlying system, we may minimize this vulnerability by continuing to use older models, perhaps with higher thresholds to avoid false positives while still detecting major failures.

9.4 Security

Last, but certainly not least, is the hazard of malicious adversaries attacking statistical monitoring. To date, little work has been done studying the effect of a malicious adversary on the results of statistical techniques. However, it appears likely that if adversaries obtained knowledge of the model used by a failure detector, they might be able to induce specific failures that did not appear anomalous to the failure detector. Alternatively, if adversaries had the ability to pollute or influence the training data used by a statistical technique, they could potentially make failures appear correct, or correct behavior appear faulty.

9.5 Future Work

There is a wide range of opportunities for future research based on statistical monitoring and the management problems of Internet services.

Future work directly leading from this dissertation includes further exploration of statistical techniques for modeling and detecting anomalies in the structural behaviors of component interactions and path shapes. Exploring how more advanced techniques might improve robustness to noise and otherwise more closely suit the data gathered during statistical monitoring could significantly improve accuracy and precision.

There are more structural behaviors that can be monitored, which may allow us to detect different classes of failures. For example, monitoring data access patterns, such as whether or not individual records in a database have been read or modified, could provide useful insight to detecting data corruptions in persistent state.

Another area of future research lies in demonstrating the scalability of statistical monitoring. There is significant room for demonstrating scalability through exploration of both the trade-offs in sampled observation of structural behaviors and the

efficacy of the resultant monitor; as well demonstrating scalability by parallelizing our analysis pipeline. There is also opportunity here for machine learning and statistical learning advances to improve scalability of statistical monitoring. Recent improvements in on-line learning of decision trees, such as [43], are promising in that they provide good bounds on correctness, while requiring only O(1) work per new observation. Enabling similar on-line learning, especially parallelizable, scalable techniques suitable for giant-scale services, will only improve the speed at which we can detect and react to problems at Internet services.

In the broader context, we believe that this dissertation leads to two general thrusts of future research, described in the rest of this section: first, applying statistical monitoring to a wider variety of management tasks, such as capacity planning, provisioning, and change impact analysis, in addition to fault management; and secondly, broadening the application of the statistical monitoring techniques discussed in this dissertation across a wider variety of systems.

9.5.1 Applying to wider variety of tasks

Machine learning techniques have the potential to bridge the gap between high-level requirements and low-level behaviors and control in several different aspects of systems management, including behavior analysis and understanding, end-to-end fault management, auto-configuration, and others. As an example in the context of the end-to-end fault-management process, reinforcement learning has the potential to learn the appropriate reactions to a detected failure given the symptoms of the failure and the systems history of recovering from similar failures. Another promising application of machine learning techniques to system management lies in the area of policy specification and auto-configuration. For any non-trivial system, it is difficult for an operator to understand the interaction between an application, its workload, and its environment. Careful application of machine learning techniques has the potential to simplify the process of configuring and reconfiguring a system, both by supporting the manual inspection and analysis of a system and, in other circumstances, by directly transforming high-level decisions into corresponding low-level configuration and policy parameters.

9.5.2 Generalizing to other systems

Generalizing statistical management techniques for application to a wider variety of systems will first require common models of management issues, and recognition of which aspects of these management issues are general and which are system-specific. Secondly, statistical management techniques may be aided by system modifications to improve observability and controllability, such as an improved tolerance for false positives without sacrificing correctness.

In addition to Internet services, large-scale distributed systems and networks are obvious targets for statistical management, but there is also the potential to apply similar techniques to smaller-scale, yet still complex systems, including desktop environments. Under certain assumptions, many machine learning techniques may be trained by aggregating observations from a large number of similar small-scale systems, with the results being distributed afterward to individual systems.

The key indicator of the applicability of machine learning techniques in these scenarios is that each requires a human operator to interpret low-level details about a complex and poorly understood system and its environment to intuit their relationships to higher-level goals. Using statistical techniques to simplify and automate this task might enable operators to concentrate on their high-level goal of building a robust, well-performing and secure system.

9.5.3 Generalizing fault localization across systems

Many large-scale systems, as diverse as Internet service clusters, inter-domain routing in the Internet, and software systems, suffer from the common problem of fault localization: when the system fails to function properly, it is often difficult to determine which part of the system is the source of the problem. We illustrate this problem using three diverse examples. Our first example, as described in Section 7, is the root cause localization problem in Internet services. Second, detecting the source of a large-scale outage in Internet routing can be a a nightmare—network operators often call other operators and exchange large volumes of emails on the operator mailing

lists [105]. Finally, it is well-known that diagnosing bugs in large-scale software systems using even the best debugging techniques is a laborious task involving several human hours or more [92].

The communities that build these large-scale systems have historically taken different approaches to solving this problem—alternatively referred to as fault diagnosis, alarm correlation, root cause analysis, and bug isolation in the context of a wide variety of systems [16, 22, 29, 30, 35, 59, 112, 118]. Despite this, we take the position that many of the challenges are common across a surprisingly diverse set of these systems.

In [83], we capture the commonality of the root cause localization problem across these different systems by defining an abstract system model and formalizing the localization problem for this model. Our model explicitly represents the nature of end-to-end failures, captures the common theoretical issues and challenges, and separates system-specific challenges into the process of mapping a system representation into the abstraction. While the generality of the model will definitely not capture several intrinsic details of a system, it does provide the ability to re-use techniques from other systems and tune them for system-specific needs.

The primary motivation of this abstract modeling is to set up a clear bridge that enables researchers in different communities to share knowledge in a common language. In particular, we hope to enable and attract theory and machine learning researchers to attack this general problem. To highlight this promise, we show in [83] how one can leverage existing techniques to solve specific aspects of the general problem and briefly illustrate how these solutions have been applied in the three application domains mentioned above. We hope that, in the future, this model will enable easier sharing of solutions across domains, as well as encourage non-systems researchers to attack the computational problem directly.

Chapter 10

Conclusions

In this dissertation, we have focused on the challenging problem of detecting application-level failures in Internet services, without requiring *a priori* knowledge of application functionality or semantics.

We demonstrated that using structural behaviors as a proxy for application functionality allows us to detect a wide variety of failures. In our testbed environment, our approach reduces the number of missed failures by 30-70%, depending on the type of injected failure. We achieve this while remaining resilient to false alarms from algorithmic noise and minor day-to-day changes to to the system and its environment. Applying our approach to captured logs and metrics from two large Internet services, we have shown that statistical techniques do detect real failures without requiring specific knowledge either about the system or about anticipated failures. Furthermore, we combined our automated fault detection with a general recovery mechanism to create an autonomous recovery system that quickly and automatically detects and recovers from a large class of failures within an Internet service.

In addition, we demonstrated how the same statistical monitoring approach can be applied to two other problems in fault management: extracting hidden structure from observations of a system; and localizing failures back to their possible causes. As with our approach to fault detection, we validated these additional applications of statistical monitoring in the context of realistic systems, inferring complex data types in the Windows Registry and localizing failures in our testbed Internet service.

The success of this dissertation in applying statistical monitoring to three different problem areas in fault management is a significant early step in creating systems management techniques that can deal with the scale, complexity and rate of change of future computer systems.

Bibliography

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, New York, 2003.

[2] Altaworks. Panorama. `http://www.altaworks.com/solutionsPanorama.htm`.

[3] Jo ao B. D. Cabrera, Lundy Lewis, and Raman K. Mehra. Detection and classification of intrusions and faults using sequences of system calls. *SIGMOD Rec.*, 30(4):25–34, 2001.

[4] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental Concepts of Dependability. In *Third Information Survivability Workshop*, October 2000.

[5] Stefan Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of the 6th ACM conference on Computer and communications security*, pages 1–7. ACM Press, 1999.

[6] S. Bagchi, B. Srinivasan, K. Whisnant, Z. Kalbarczyk, and R.K. Iyer. Hierarchical Error Detection in a Software Implemented Fault Tolerance (SIFT) Environment. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):203–224, 2000.

[7] Lisanne Bainbridge. The ironies of automation. In Keith Duncan, Jens Rasmussen, and Jacques Leplat, editors, *New Technology and Human Error*. John Wiley and Sons, Inc., 1987.

[8] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 238–250, November 2002.

[9] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[10] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: real-time modelling and performance-aware systems. In *9th Workshop on Hot Topics in Operatings Systems (HotOS IX)*, May 2003.

[11] Michael Barnes. J2EE application servers: Market overview. The Meta Group, March 2004.

[12] Peter Bodík, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jon Hui, Armando Fox, Michael I. Jordan, and David Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC '05)*, June 2005.

[13] A. Bouloutas, S. Calo, and A. Finkel. Alarm Correlation and Fault Identification in Communication Networks. *IEEE Transactions on Communications*, 42(2/3/4):523–533, Feb/Mar/Apr 1994.

[14] Eric Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, 5(4):46–55, July 2001.

[15] Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley Professional, 1995.

[16] Aaron Brown, Gautam Kar, and A. Keller. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. In *Seventh IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[17] Nevil Brownlee, Kimberly Claffy, and Evi Nemeth. DNS Measurements at a Root Server. In *Sixth Global Internet Symposium*, November 2001.

[18] Business Internet Group San Francisco,. A Report on Web Application Integrity, May 2002.

[19] Business Internet Group San Francisco,. Government Web Application Integrity, May 2003.

[20] Business Internet Group San Francisco. The Black Friday Report on Web Application Integrity, January 2003.

[21] Bz Research LLC. Third Annual Java Use and Awareness Study, November 2004.

[22] Matthew Caesar, L. Subramanian, and Randy H. Katz. Towards localizing root causes of BGP dynamics. Technical report, U.C. Berkeley UCB/CSD-04-1302, 2003.

[23] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. A Microrebootable System: Design, Implementation, and Evaluation. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, December 2004.

[24] George Candea, Emre Kıcıman, Shinichi Kawamoto, and Armando Fox. Autonomous Recovery in Componentized Internet Applications. *To appear in Cluster Computing Journal*, 9(1), February 2006.

[25] George Candea, Emre Kıcıman, Steve Zhang, Pedram Keyani, and Armando Fox. JAGR: An Autonomous Self-Recovering Application Server. In *Proceedings of the 5th International Workshop on Active Middleware Services*, 2003.

[26] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Appliations*, 2002.

[27] Ben Chamy. BlackBerry endures another outage. *CNET News.com*. June 22, 2005. `http://news.com.com/BlackBerry+endures+another+outage/2100-1039_3-57580%43.html`.

[28] Ben Chamy and Robert Lemos. VoIP provider Vonage suffers outage. *CNET News.com*, August 2004. `http://news.com.com/VoIP+provider+Vonage+suffers+outage/2100-7352_3-529%3439.html`.

[29] Di-Fa Chang, Ramesh Govindan, and John Heidemann. The Temporal and Toplogical Characteristics of BGP Path Changes . In *Proceedings of the 11th IEEE International Conference on Network Protocols*, November 2003.

[30] Mike Chen, Alice Zheng, Jim Lloyd, Michael Jordan, and Eric Brewer. A Statistical Learning Approach to Failure Diagnosis. In *Proceedings of the First International Conference on Autonomic Computing*, May 2004.

[31] Mike Y. Chen, Anthony Accardi, Emre Kıcıman, David Patterson, Armando Fox, and Eric Brewer. Path-Based Failure and Evolution Management. In *The 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[32] Mike Y. Chen, Emre Kıcıman, Anthony Accardi, Armando Fox, and Eric Brewer. Using Runtime Paths for Macro Analysis. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.

[33] Mike Y. Chen, Emre Kıcıman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *The International Conference on Dependable Systems and Networks (IPDS Track)*, June 2002.

[34] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Curushankar Rajamani, and David Lowell. The Rio File Cache: Surviving

Operating System Crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[35] Ram Chillarege. Self-testing software probe system for failure detection and diagnosis. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 10. IBM Press, 1994.

[36] J. Choi, M. Choi, and S. Lee. An Alarm Correlation and Fault Identification Scheme Based on OSI Managed Object Classes. In *Proceedings of IEEE Conference on Communications*, June 1999.

[37] CNET. Google experiences brief outage. *CNET News.com*. May 8, 2005. `http://news.com.com/Google+experiences+brief+outage/2100-1038_3-5699450%.html`.

[38] Ira Cohen, Jeffrey S. Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[39] Ira Cohen, Moises Goldszmidt, Steve Zhang, Terence Kelly, Armando Fox, and Julie Symons. Capturing, Indexing, Clustering, and Retrieving System History. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[40] W. J. Conover. *Practical Nonparametric Statistics*. John Wiley and Sons, Inc., New York, NY, 3 edition, 1999.

[41] Erik D. Demaine, Dotan Emanuel, Amos Fiat, and Nicole Immorlica. Correlation Clustering with Partial Information. In *Proceedings of the 6th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and 7th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM-APPROX 2003)*, pages 1–13, August 2003.

[42] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph Hellerstein, and Wei Hong. Model Driven Data Acquisition in Sensor Networks. In *Proceedings of Thirtieth International Conference on Very Large Databases*, August 2004.

[43] Pedro Domingos and Geof Hulten. Mining High-Speed Data Streams. In *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2000.

[44] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. John Wiley and Sons, Inc., New York, NY, 2 edition, 2001.

[45] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72, 2001.

[46] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[47] Tom Fawcett and Foster Provost. Adaptive fraud detection. *Data Min. Knowl. Discov.*, 1(3):291–316, 1997.

[48] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 78–91, October 1997.

[49] Archana Ganapathi, Yi-Min Wang, Ni Lao, and Ji-Rong Wen. Why PCs Are Fragile and What We Can Do About It: A Study of Windows Registry Problems. In *Proceedings of the International Conference on Dependable Systems and Networks '04)*, June 2004.

[50] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[51] Alorie Gilbert. Hotmail outage peeves some e-mail users. *CNET News.com*. February 23, 2005. `http://news.com.com/Hotmail+outage+peeves+some+e-mail+users/2100-1038_3%-5587906.html`.

[52] Alorie Gilbert. Power outage knocks CheckFree offline. *CNET News.com*. June 15, 2005. `http://news.com.com/Power+outage+knocks+CheckFree+offline/2100-1038_3-5%748539.html`.

[53] Jim Gray. Why Do Computers Stop and What Can Be Done About It? In *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems*, January 1986.

[54] Jim Gray. A Census of Tandem System Availability Between 1985 and 1990. Technical Report 1, Tandem Computers, January 1990.

[55] Steven D. Gribble. Robustness in Complex Systems. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

[56] Steven D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, October 2000.

[57] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, Nikita Borisov, Steve Czerwinski, R. Gummadi, Jason Hill, Anthony Joseph andy Randy H. Katz, Z. Morley Mao, Steve Ross, and Ben Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Journal of Computer Networks*, 35(4), March 2001.

[58] B. Gruschke. A New Approach for Event Correlation based on Dependency Graphs. In *Proceedings of the 5th Workshop of the HP-OpenView University Association (HPOVUA)*, April 1998.

[59] B. Gruschke. Integrated Event Management: Event Correlation using Dependency Graphs. In *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 98)*, October 1998.

[60] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: an efficient clustering algorithm for large databases. In *ACM SIGMOD International Conference on Management of Data*, pages 73–84, June 1998.

[61] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer-Verlag, 2001.

[62] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer-Verlag, New York, NY, 2001.

[63] Hewlett Packard Corporation. HP Openview. `http://www.hp.com/openview/index.html`.

[64] Matt Hines. Outage nearly over, PayPal says. *CNET News.com*. October 13, 2004. `http://news.com.com/Outage+nearly+over%2C+PayPal+says/2100-1032_3-54068%04.html`.

[65] David Hovemeyer and William Pugh. Finding Bugs is Easy. In *Companion Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Onward! Track*, October 2004.

[66] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. *Computer*, 30(4):75–82, 1997.

[67] Jim Hu. Amazon.com hit with outages. *CNET News.com*. December 6, 2004. `http://news.com.com/Amazon.com+hit+with+outages/2100-1030_3-5479954.htm%l`.

[68] Jim Hu. MSN Messenger outage blamed on 'data center' issue. *CNET News.com*. February 8, 2005. `http://news.com.com/MSN+Messenger+outage+blamed+on+data+center+issue/21%00-1032_3-5568012.html`.

[69] Jim Hu. Microsoft Money blackout approaching day four. *CNET News.com*, July 2004. `http://news.com.com/Microsoft+Money+blackout+approaching+day+four/2100-%1012_3-5289896.html`.

[70] Andrew C. Huang and Armando Fox. Cheap Recovery: A Key to Self-Managing State. *ACM Transactions on Storage*, 1(1), February 2005.

[71] IBM. Tivoli Business Systems Manager, 2001. `http://www.tivoli.com`.

[72] Dean Jacobs. Distributed Computing with BEA WebLogic Server. In *Proceedings of the Conference on Innovative Data Systems Research*, January 2003.

[73] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[74] The JBoss Group. Jboss. `http://www.jboss.org/`.

[75] Guofei Jiang, Haifeng Chen, Cristian Ungureanu, and Kenji Yoshihira. Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC '05)*, June 2005.

[76] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash crowds and denial of service attacks: characterization and implications for CDNs and web sites. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 293–304, New York, NY, USA, 2002. ACM Press.

[77] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure Data Analysis of a LAN of Windows NT Based Computers. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 178–187, October 1999.

[78] E. Keogh, S. Lonardi, and W Chiu. Finding Surprising Patterns in a Time Series Database In Linear Time and Space. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 550–556, July 2002.

[79] Keynote. Keynote Consumer 40 Internet Performance Index. `http://www.keynote.com/solutions/performance_indices/consumer_index/con%sumer_40.html`.

[80] Inc. Keynote Systems. `http://www.keynote.com/`.

[81] Inc. Keynote Systems. Virus Slows Internet; Google Searches Crawl. *Press Release*, July 2004. `http://www.keynote.com/news_events/public_services_portal/google072604.%html`.

[82] Emre Kıcıman and Armando Fox. Detecting Application-Level Failures in Component-based Internet Services. *IEEE Transactions on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, September 2005.

[83] Emre Kıcıman and Lakshminarayanan Subramanian. Root Cause Localization in Large Scale Systems. In *Proceedings of the First Workshop on Hot Topics in System Dependability*, June 2005.

[84] Emre Kıcıman and Yi-Min Wang. Discovering Correctness Constraints for Self-Management of System Configuration. In *Proceedings of the International Conference on Autonomic Computing*, May 2004.

[85] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C. Snoeren. IP Fault Localization via Risk Modeling. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, May 2005.

[86] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 83–93, New York, NY, USA, 2004. ACM Press.

[87] Ted Kremenek and Dawson Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Proceedings of the 10th International Static Analysis Symposium*, June 2003.

[88] D. Richard Kuhn. Sources of failure in the public switched telephone network. *IEEE Computer*, 30(40), April 1997.

[89] Craig Labovitz, Abha Ahuja, and Farnam Jahanian. Experimental Study of Internet Stability and Backbone Failures. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 278, Washington, DC, USA, 1999. IEEE Computer Society.

[90] Robert Lemos. MSN fighting Messenger difficulties. *CNET News.com*. October 11, 2004. `http://news.com.com/MSN+fighting+Messenger+difficulties%2C+virus/2100-7%349_3-5406282.html`.

[91] Robert Lemos. Republican Web sites downed by mystery outage. *CNET News.com*. October 20, 2004. `http://news.com.com/Republican+Web+sites+downed+by+mystery+outage/2100-%1028_3-5419882.html`.

[92] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug Isolation via Remote Program Sampling. In *ACM SIGPLAN 2003 Conference on Programming Languages Design and Implementation*, June 2003.

[93] Benjamin C. Ling, Emre Kıcıman, and Armando Fox. Session State: Beyond Soft State. In *The 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[94] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding BGP Misconfiguration. In *Conference of the Special Interest Group on Data Communication (SIGCOMM)*, August 2002.

[95] Christopher D. Manning and Hinrich Shutze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, MA, 1999.

[96] Evan Marcus and Hal Stern. *Blueprints for High Availability*. John Wiley and Sons, Inc., New York, NY, 2000.

[97] Michael Mesnier, Eno Thereska, Gregory R. Ganger, Daniel Ellard, and Margo I. Seltzer. File classification in self-* storage systems. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 44–51, May 2004.

[98] Sun Microsystems. Java2 Enterprise Edition (J2EE). `http://www.javasoft.com/j2ee/`.

[99] Elinor Mills. Google ups ante in mapping rivalry. *CNET News.com*, July 2005. `http://news.com.com/Google+ups+ante+in+mapping+rivalry/2100-1032_3-5803%462.html`.

[100] Tom M. Mitchell. *Machine Learning*. WCB/McGraw Hill, 1997.

[101] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, Sebastopol, CA, 3 edition, 2001.

[102] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, October 1996.

[103] John D. Musa. *Software Reliability Engineering*. Osborne/McGraw-Hill, 1998.

[104] Kiran Nagaraja, Xiaoyan Li, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services. In *4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, March 2003.

[105] NANOG: The North American Network Operators Group. `http://www.nanog.org/`.

[106] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, Warsaw, Poland, April 2003.

[107] David Oppenheimer, Archana Ganapathi, and David Patterson. Why do Internet services fail, and what can be done about it? In *4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, March 2003.

[108] Sujay Parekh, Neha Gandhi, Joe Hellerstein, Dawn Tilbury, T. S. Jayram, and Joe Bigus. Using Control Theory to Achieve Service Level Objectives in Performance Management. (1–2):127–141, July 2002.

[109] David A. Patterson. A Simple Way to Estimate the Cost of Downtime. In *Proceedings of the 16th Systems Administration Conference*, November 2002.

[110] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[111] James Reason. *Human Error*. Cambridge University Press, 1990.

[112] Joshua A. Redstone, Michael M. Swift, and Brian N. Bershad. Using Computers to Diagnose Computer Problems. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems*, 2003.

[113] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. SABER: smart analysis based error reduction. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 243–251, New York, NY, USA, 2004. ACM Press.

[114] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. SABER: Smart Analysis Based Error Reduction. In *Proceedings of the International Symposium on Software Testing and Analysis*, Boston, MA, July 2004.

[115] Reuters. Walmart.com site restored after outage. *CNET News.com*. July 14, 2005. `http://news.com.com/Walmart.com+site+restored+after+outage/2110-1038_3-%5788436.html`.

[116] I. Rouvellou and G. W. Hart. Automatic Alarm Correlation for Fault Identification. In *Proceedings of IEEE INFOCOM*, April 1995.

[117] Norman F. Schneidewind. Successful Application of Software Reliability Engineering for the NASA Space Shuttle. In *International Symposium on Software Reliability Engineering*, pages 71–82, November 1997.

[118] M. Steinder and A. Sethi. Increasing robustness of fault localization through analysis of lost, spurious and positive symptoms. In *Proceedings of IEEE INFOCOM*, June 2002.

[119] David Sullivan. *Using probabilistic reasoning to automate software tuning*. PhD thesis, Harvard University, Cambridge, MA, September 2003.

[120] Mark Sullivan and Ram Chillarege. Software Defects and their Impact on System Availability – A Study of Field Failures in Operating Systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, June 1991.

[121] Mark Sullivan and Ram Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 475–484, July 1992.

[122] David M. J. Tax, Alexander Ypma, and Robert P. W. Duin. Pump failure detection using support vector data descriptions. In *IDA '99: Proceedings of the Third International Symposium on Advances in Intelligent Data Analysis*, pages 415–426, London, UK, 1999. Springer-Verlag.

[123] TeaLeaf Technology. TeaLeaf Technology Assists Priceline.com 'Super Computer', January 2003. `http://www.tealeaf.com/news/press_releases/2003/0113.asp`.

[124] Anshuman Thakur and Ravishankar Iyer. Analyze-NOW - An Environment for Collection and Analysis of Failures in a Network of Workstations. *IEEE Transactions on Reliability*, 45(4), December 1996.

[125] Transaction Processing Performance Council. TPC-W Benchmark Specification. http://www.tpc.org/wspec.html.

[126] Kalyan Vaidyanathan and Kenny Gross. Proactive detection of software anomalies through mset. In *Proceedings of the Workshop on Predictive Software Models*, September 2004.

[127] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of*

the 6th Symposium on Operating Systems Design and Implementation (OSDI), December 2004.

[128] Yi-Min Wang, Chad Verbowski, and Daniel R. Simon. Persistent-state Checkpoint Comparison for Troubleshooting Configuration Failures. In *Proceedings of the IEEE Conference on Dependable Systems and Networks*, 2003.

[129] K. Whisnant, S. Bagchi, B. Srinivasan, Z. Kalbarczyk, and R.K. Iyer. Incorporating Reconfigurability, Error Detection and Recovery into Chameleon ARMOR Architecture. Technical Report CRHC-98-13, University of Illinois at Urbana-Champaign, 1998.

[130] Weng-Keen Wong, Andrew Moore, Gregory Cooper, and Michael Wagner. Bayesian network anomaly pattern detection for disease outbreaks. In *The Twentieth International Conference on Machine Learning*, August 2003.

[131] Wei Xu, Peter Bodík, and David Patterson. A Flexible Architecture for Statistical Learning and Data Mining from System Log Streams. In *Proceedings of the Workshop on Temporal Data Mining: Algorithms, Theory and Applications at The Fourth IEEE International Conference on Data Mining (ICDM'04)*, November 2004.

[132] A. Yemeni and S. Kliger. High Speed and Robust Event Correlation. *IEEE Communications Magazine*, 34(5), May 1996.

[133] Steve Zhang, Ira Cohen, Moises Goldszmidt, Julie Symons, and Armando Fox. Ensembles of Models for Automated Diagnosis of System Performance Problems. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2005.