Detecting Application-Level Failures in Component-based Internet Services

Emre Kıcıman and Armando Fox
{emrek,fox}@cs.stanford.edu

***Abstract*— Most Internet services (e-commerce, search engines, etc.) suffer faults. Quickly detecting these faults can be the largest bottleneck in improving availability of the system. We present Pinpoint, a methodology for automating fault detection in Internet services by (1) observing low-level, internal *structural* behaviors of the service; (2) modeling the majority behavior of the system as correct; and (3) detecting anomalies in these behaviors as possible symptoms of failures. Without requiring any *a priori* application-specific information, Pinpoint correctly detected 89–96% of major failures in our experiments, as compared with 20–70% detected by current application-generic techniques.**

***Index Terms*— Internet services, application-level failures, anomaly detection**

I. Introduction

A significant part of recovery time (and therefore availability) is the time required to detect and localize service failures. A 2003 study by Business Internet Group of San Francisco (BIG-SF) [1] found that of the 40 top-performing web sites (as identified by KeyNote Systems [2]), 72% had suffered user-visible failures in common functionality, such as items not being added to a shopping cart or an error message being displayed. These failures do not usually disable the whole site, but instead cause brown-outs, where part of a site's functionality is disabled or only some users are unable to access the site.

Many of these failures are *application-level failures* that change the user-visible functionality of a service, but do not cause obvious lower-level failures detectable by service operators. Our conversations with Internet service operators confirm 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 [3]. Other sites we spoke 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 reported user-visible failures [4]. Fast detection of these failures is therefore a key problem in improving Internet service availability.

We present Pinpoint, a prototype monitor for quickly detecting failures in component-based Internet services, without requiring *a priori* information about the correct behavior of an application. Pinpoint's insight is that failures that affect a system's user-visible behavior are likely to also affect a system's internally visible behaviors. Monitoring those behaviors that are closely tied to service functionality, Pinpoint develops a model of the normal patterns of behavior inside a system. When these patterns change, Pinpoint has high confidence that the service's functionality has also changed, indicating a possible failure.

Specifically, Pinpoint monitors inter-component interactions and the shapes of *paths* (traces of client requests that traverse several components) to quickly build a dynamic and self-adapting model of the "normal" behavior of the system. Once it notices an anomaly, Pinpoint correlates it to its probable cause in the system, a set of likely faulty components.

We recognize that the most rigorous test of Pinpoint's usefulness is through validation in real Internet service environments, and are currently deploying Pinpoint in such services for exactly this reason. In this paper, however, we take advantage of a controlled testbed environment to test Pinpoint with a different kind of rigour: by methodically injecting a wide-variety of failures, including source-code bugs, Java exceptions and others, into all the componentes of several applications running on our testbed cluster, we evaluate how well Pinpoint discovers failures and characterize Pinpoint's strengths and weaknesses.

A. Application-Level Failures

We define an *application-level failure* as a 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 layer is the operating system, the highest layer is the application, with various other layers in between (*e.g.,* libraries, middleware software, standard protocols, etc.). In this model, an application-level failure manifests solely in the application layer, though the cause of the failure may be in another layer. In particular, an application-level failure is *not* a fail-stop failure, as this would generally cause several layers of the software stack to stop.

Figure 1 shows an example of application-level failure that the authors have encountered at one web service. Although this page should be displaying a complete flight itinerary, it shows no flight details at all. Instead, it shows only an incorrect confirmation date. While the authors are not privy to the detailed cause of this failure, it appears that no symptoms are manifest below the application-layer: the site responds to pings, HTTP requests, and returns valid HTML.

B. Current Monitoring Techniques

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

Fig. 1. Instead of a complete itinerary, this screenshot of an airline site shows no flight details at all. A naive fault monitor, however, would find no problems: the site responds to pings, HTTP requests, and returns valid HTML.

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 faults 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.

From our discussions with Internet service operators, we find that existing detection methods fall into three categories. First, *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.

Secondly, *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 will both miss real failures and cause false-alarms. For these reasons, neither the sites that we have spoken with, nor those studied in [4] make extensive use of these monitors.

Thirdly, *user-activity monitors*, watch simple statistics about the gross behavior of users and compare them to historical trends. Such a monitor might track the searches per second or orders per minute at a site. These monitors are generally easy to deploy and maintain, and at a site with many users, can detect a broad range of failures that affect the given statistic, though they do not often give much more than an indication that *something* might have gone wrong. Since these monitors are watching user behavior, they can generate false alarms due to external events, such as holidays or disasters. Finally, customer service complaints are the catch-all fault detectors.

C. Contributions

We introduced path-analysis for localizing failures in Internet services in [5] and in [3], [6] broadened our path-analysis techniques to show how this coarse-grained visibility into a system can aid fault management, fault impact analysis, and evolution management. The contributions of this paper are as follows:

1) We identify two kinds of system behaviors—path-shapes and component-interactions—that are easy to observe with application-generic instrumentation and serve as reliable indicators of changes in high-level application behavior.
2) We show how existing statistical analysis and machine learning techniques for anomaly detection and classification can be used to monitor these behaviors to discover failures without *a priori* knowledge of the correct behavior or configuration of the application.
3) We evaluate these techniques by integrating Pinpoint with a popular middleware framework (J2EE) for building Internet services. We inject a variety of failures into several J2EE applications, and measure how well and how quickly Pinpoint detects and localizes the faults. We also test Pinpoint's susceptibility to false positives by performing a set of common changes, such as a software upgrade.

Pinpoint does not attempt to detect problems before they happen. Rather, we focus on detecting a failure as quickly as possible after it occurs, to keep it from affecting more users and to prevent cascading faults. Also, Pinpoint attempts to notice where a failure is occuring in the system, and does not attempt to explain why it might be failing. Combined with a simple generic recovery mechanism, such as microreboots [7], simply knowing the location of a fault is often sufficient for fast recovery.

Section II describes Pinpoint's approach to detecting and localizing anomalies, and the low-level behaviors that are monitored. Section III explains in detail the algorithms and data structures used to detect anomalies in each of these behaviors. In Section IV, we describe an implementation of Pinpoint and our testbed environment. In Section V, we evaluate Pinpoint's effectiveness at discovering failures, the time to detect failures, and its resilience to false-alarms in the face of normal changes in behavior. We conclude by discussing related work and future directions.

II. Pinpoint Approach

With Pinpoint, we attempt to combine the easy deployability of low-level monitors with the higher-level monitors' ability to detect application-level faults. This section describes our assumptions and our approach.

A. Target System

Pinpoint makes the following 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 broken down as a *path*, a tree of the names of components that participate in the servicing of that request.
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.

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 [8] 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.

B. Observation, Detection, Localization

The Pinpoint approach to detecting and localizing anomalies is a three-stage process of observing the system, learning the patterns in its behavior, and looking for anomalies in those behaviors.

1) **Observation:** We capture the *runtime path* of each request served by the system: an ordered set of coarse-grained components, resources, and control-flow used to service the 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 normal 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.
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 component and manage their invocations, instrumenting the middleware gives us the visibility we require. And by instrumenting a standard middleware, such as J2EE or .NET, we have the ability to observe any application built atop it.

Before analyzing these observations, we "bin" our runtime paths by their request type. By analyzing each type of request separately. we aim to improve 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.

In the learning phase, Pinpoint builds reference models of normal behavior under the assumption that most of the time, most of the system is working correctly. Note that this assumption does not require the system to be completely fault-free. Quickly and dynamically building accurate and complete models depends on the assumption that there is a high traffic and a large number of independent requests to the service: a large fraction of the service's code base and functionality is exercised in a relatively short amount of time. 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.

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. These two models complement each other: a historical analysis can detect acute failures, but 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 anomaly detection function itself is model-specific. We describe both the models and the anomaly detection function in detail in the next section.

Once a failure has been detected, a separate policy agent is responsible for deciding how to respond to discovered failures. Discussion and analysis of how to react to possible failures is outside the scope of this paper, though we discuss one policy of rebooting failed components in [9].

III. Algorithms and Data Structures

From our observations, we extract two behaviors: path shapes and component interactions. The techniques we use to model these behaviors and detect anomalies are detailed in the rest of this section.

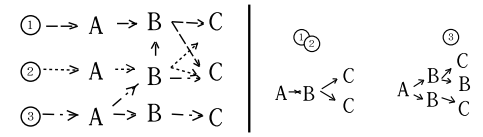
A. Modeling Component Interactions

The first low-level behavior that we analyze is component interactions. Specifically, we model the interactions between an instance of a component and each class of components in the system. The intuition behind looking for anomalies here is that a change in a component's interactions with the rest of the system is likely to indicate that the functional behavior of that component is also changing.

If A and B are both classes of components, and a_i is an instance of A, we measure the interactions between a_i and any instance in B. We do not analyze interactions between two individual instances because in many systems this level

```
1   9.41    Catal ogEJB
2   1.09    Shopp ingCa rtEJB
3   0.34    Shopp ingCo ntroll erEJB
4   0.12    JspSe rvlet
5   0.02    MainS ervle t
```

Fig. 2. This list shows the top five χ^2 goodness-of-fit scores of components after injecting an anomaly into CatalogEJB. The scores are normalized 1 is the threshold for statistical significance α. CatalogEJB, the most anomalous, is significantly more anomalous than other components.



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

Fig. 3. 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.

of interaction is not identical across instances of a component. 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.

We represent these interactions of an instance a_i 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 a through each interaction.

We generate our historical models of the component interactions by averaging the weights of links through them over time. Our peer model is generated by averaging the current behaviors of replicated peers in the system.

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:

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

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. Q is our confidence that the normal 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 normal 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 more prone to false-positives. In our experiments, we use a level of significance $\alpha = 0.005$. Figure 2 shows an example output from one of our fault detection runs.

B. PCFGs Model Path Shapes

The shape of a path is the ordered set of logical software components (as opposed to instances of components on specific machines) used to service a client request [6]. 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 boundaries are hidden).

Path shapes give a different view of the system than component interactions. Component interactions look at one component's behavior across many client requests, whereas a path shape gives the orthogonal view, inspecting a single request's interactions with many components. As an example of a failure detectable by path shape analysis, but not by component interactions, consider an occasional error that affects only a few requests out of many. Since few requests are affected, it might not significantly change the weights of the links in a component interaction model, but path shape analysis would detect anomalies in the individual paths. As a converse example, it is normal for a password-verification component to occasionally reject login attempts and path-shape analysis of a request that ended in a login failure would not be considered anomalous. However, component interaction analysis would be able to detect an anomaly if the component was rejecting more than the usual proportion of login attempts.

We model a set of path shapes as a probabilistic context-free grammar (PCFG) [10], 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 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 in Pinpoint, 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 3 shows an example of a trivial set of path shapes and the corresponding PCFG.

To learn a PCFG from a set of sample path shapes, we iterate over every branching point in the component call trees represented by the set of sample paths. For every branch in a path shape where a component N^i makes a set of calls to components ζ^j, we increment two counters. The first counter, c_{N^i} is associated with the component N^i and tracks the number of times this component has been seen across our sample set of path shapes. The second counter, $c_{N^i \rightarrow \zeta^j}$ is associated with the grammar rule $N^i \rightarrow \zeta^j$, and tracks the number of times N^i has called ζ^j. Once we have processed all the path shapes in our sample, we calculate P(R_{ij}) for each R_{ij} as $c_{N^i \rightarrow \zeta^j} / c_{N^i}$.

To build a historical reference model, we build a PCFG

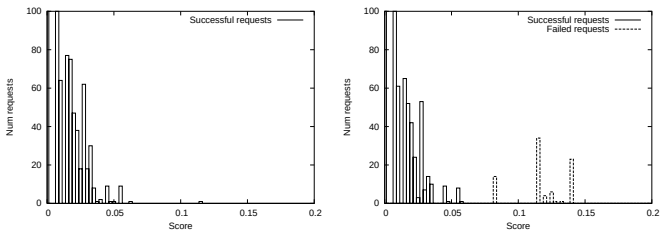


Fig. 4. The left graph is a histogram of request scores during normal behavior. The right shows the scores when we cause part of the system to misbehave. Some requests fail as a result, and are clearly separated from the successful requests by our scoring algorithm. This data is taken from our experiments with the Petstore 1.3, described in Section IV.

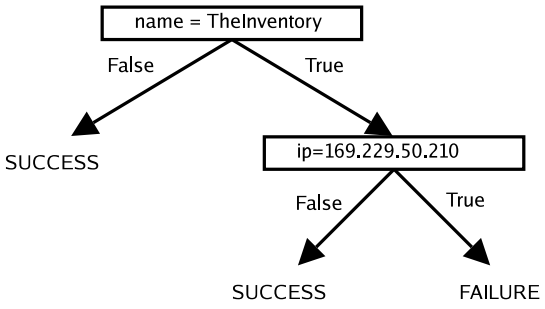


Fig. 5. A decision-tree learned for classifying faulty paths in one of our experiments. Here we injected a fault into TheInventory component on one machine in a cluster. The decision tree learning algorithm chose the "name=TheInventory" as the most important classifier, and the machine's ip address as the second.

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.

To determine whether a subsequently-observed path shape is anomalous, once we have built a reference model, we start at the root of the tree of component calls in a path shape and compare each transition to its corresponding rule R_{ij} in our PCFG. The total "anomaly" score for path shape is:

$$\sum_{\forall R_{ij} \in t} \min(0, P(R_{ij}) - 1/n_i) \quad (2)$$

where t is the path shape being tested, and n is the total number of rules in our PCFG beginning at 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 shows that this scoring function does a good job of separating normal paths from faulty paths.

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 anomalous. 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 anomalous. 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 anomalous, since $0.01 > 5 * (1 - 0.999)$. 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.

C. Decision Trees Localize Path Shape Anomalies

While detecting possibly faulty requests can be important in its own right, it is often more useful to discover what components in the system may be failing. Once we have detected anomalous path shapes using a PCFG analysis, we can attempt a second stage of analysis to localize the problem to specific components by searching for a correlation between anomalies and the features of our paths. If we find a correlation 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.

To find this correlation, we learn a decision tree to classify (predict) whether a path shape is a success or failure based on its associated features. These features correspond to the path information that Pinpoint collects, such as the names of EJB's, IP addresses of server replcias 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; looking at which components are used as tests within the decision tree function tells us which components are correlated with request failures. In our experiments, we use the ID3 algorithm [11] for learning a decision tree, though recent work suggests that C4.5 decision trees might perform better [12].

The training set for our decision-tree learning is the set of paths classified as normal or anomalous by our PCFG detector. The input to our target function is a path, and the output of the function is whether or not the path is anomalous. Our goal is to build a decision tree that approximates our observed anomalies 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.

Since anomalies detected by our component-interaction analysis already point to specific faulty components, we do not usually need to localize them further. However, if we notice anomalies in many components at once, it may be worthwhile to "localize" the problem to some common attribute or feature of the failing components.

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 sets of components rather than by indvidual components. More interestingly, it allows us to 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.

IV. Experimental Setup and Methodology

The goal of our experiments is to characterize Pinpoint's capabilities (detection rate, time to detection, and resilience to false alarms) in the context of a realistic Internet service environment, and to compare these capabilities against current techniques for failure detection.

The core of Pinpoint is a plugin-based analysis engine. Though we do not describe this prototype in detail, let it suffice that we have created plugins corresponding to the anomaly-detection and localization algorithms described in Section III.

In this section, we describe the setup of our small 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.

A. Instrumentation

We instrumented the JBoss open-source implementation of the J2EE middleware standard which provides a standard runtime enironment for three-tier enterprise applications [13], [14]. Figure 6 illustrates this architecture. 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[1]. 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.

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.

[1]Java Server Pages (JSP) provide a server-side scripting capability for creating dynamic web content.

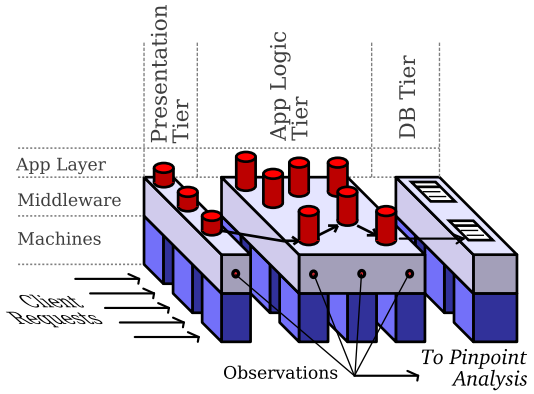


Fig. 6. J2EE provides a three-tiered architecture for building applications.

Our instrumentation is spread across several sites within the J2EE 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. On a single-node system, our unoptimized implementation can maintain full instrumentation without dropping observations at a steady state of **75** clients, and can tolerate bursts of up to 300 clients for minutes at a time (with subsequent queuing delays). 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 [3], [15].

B. Applications and workloads

We deployed three different applications in our testbed platform:

Petstore 1.1 is Sun's sample J2EE application that simulates an e-commerce web site (storefront, shopping cart, purchase, tracking, etc.). It consists of 12 application components (EJBs and servlets), 233 Java files, and about 11K lines of code, and stores its data in a Cloudscape database. Its primary advantage is that we have been able to modify the applciation 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, and is made of 47 components, 310 files, and 10K lines of code. Because of its new architecture, we were unable to cluster Petstore 1.3, but its increased functionality makes it a useful additional application.

RUBiS is an auction web site, developed at Rice University for experimenting with different design patterns for J2EE [16]. RUBiS contains over 500 Java files and over 25k lines of code. More relevant for our purposes, RUBiS has 21 application components and several servlets.

While RUBiS comes with a custom load generator, we built our own HTTP load generator for use with 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 so that the overall load the service fully exercises the components and functionality of the site. 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. The traces are designed to take different routes through the web service, such that a failure in a single part of the web service will not artifically block all the traces early in their life cycle.

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.

When we deploy these applications, we run our observation collector, the application, the database, and the load generator on separate machines. Our clustered version of Petstore 1.1 runs with one front-end node running the code to handle presentation, and three application-logic nodes.

C. Fault and Error Load

The goal of Pinpoint is to detect application-level failures, as described in Section I-A. 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 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 studies of failures in deployed systems as well as the faults injected by other researchers in their experiments [4], [18]–[21]. While some experiments focus on a range of byzantine faults, we found that most of the faults injected concentrated on problems that caused program crashes and other obvious failures, as opposed to triggering only application-level failures.

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 (we discuss this further in the Section IV-D.

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).

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 errors 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, overloads manfiested as an overloaded database, where the application tier would receive timeouts or errors when querying the database.

Source code bug injection: Even simple programming bugs remain uncaught and cause problems in real software [22], [23], so introducing them can be a useful method of simulating faults due to software bugs [20].

There are several types of failures that we explicitly decided not to inject. First, we do not inject low-level hardware or OS faults, such as CPU register bit-flips, memory corruptions, and IO errors because, empirically, these faults do not commonly manifest as application-level failures that would otherwise go unnoticed [7], but either cause easy-to-detect process crashes or are coerced to manifest as exceptions by the Java virtual machine in our testbed.

We expect that exceptions and omissions are extremely likely to effect 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. 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." At the other end of the spectrum, injecting an exception into the ShoppingClientController component in Petstore 1.3 prevents the user from seeing the website at all, and instead displays an internal server error for

[2]By generally accepted definition [17], *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 throwing a Java exception). In the rest of this paper, we use the "fault injection" to include both fault and error injection.

all requests.

In our experiments, we concentrate on faults that cause user-visible errors. We verify this in our experiments by modifying our load generator to verify all the HTML output 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 running a deterministic workload. Any remaining non-deterministic output, such as order numbers, are canonicalized before hashing.

Even though we are in control of the fault injection, it is not trivial to determine which client requests in an experiment failed. Component interactions and corrupted state or resources can all lead to cascading failures. As our ground-truth comparison, we mark a request as having failed if (1) we directly injected a fault into it, (2) it causes an HTTP error, or (3) its HTML output fails our MD5 hash.

We classify the faults we inject into two categories, based on their impact: A *major fault* affects more than 1% of the requests in an experiment, while a *minor fault* effects less than 1% of the requests.

No fault injection scheme can accurately mimic the variety and affects of failures that occur in the real world. However, given the number and breadth of faults we have injected into each of our three applications we are confident that our experiments capture a wide range of realistic fault behaviors.

D. Comparison Monitors

To better evaluate Pinpoint's ability to detect failures, we implemented several types of monitors for comparison.

Low-level Monitors: Our own load generator doubles as an HTTP monitor and an HTML monitor. It scans the HTML content being returned to users for obvious signs of errors and exceptions. In our case, whether the keywords "error", and "exception" appear in the HTML text.

Since we are purposefully injecting only application-failures, we did not implement a low-level ping or heartbeat monitors. As none of our injected faults cause our servers to crash or hang, we can assume that these ping and heartbeat monitors would not have noticed. Even under the overload conditions that caused our service to fail, none of our software processes or hardware nodes completely crashed.

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 all of them: 70% were declared exceptions, and 30% were runtime exceptions. 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 500 NullPointer exceptions. Because of these issues, we concluded that building an application-generic exception monitor was not feasible for this class of system.

Application-specific Monitors: 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-speciic, 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) for our testbed and found that it detected "failures" in almost all our experiments, including false-alarms in all our fault-free 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.

We do not 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 test for almost any expected fault. Additionally, Pinpoint's main improvement in comparison to these monitors is not its ability to detect failures, but the fact that Pinpoint is a low-maintenance, application-generic solution to high-level fault-detection. Since we do not have real users generating workload on our site, we do not include monitors which watch for changes in user behavior.

V. Experiments

In this section, we present experiments and analyses to answer the questions of whether Pinpoint detects and localizes failures, how quickly it detects failures, and whether Pinpoint may be prone to false-alarms during normal day-to-day operations.

To test its fault detection and localization rates, we connect Pinpoint to a widely-used middleware system, inject various faults and errors into applications running on top of this middleware, and evaluate how well Pinpoint detects and localizes the resultant failures. Because we inject faults into a live application running on enterprise-ready middleware, we have confidence that the application's behavior following a failure is realistic, even though the fault itself is artificially caused. By detecting the change in application behavior during a failure condition, we can realize that a fault has occurred.

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 collected these traces once, and analyzed them off-line. In addition, we injected JNDI corruption and overload failures, and analyzed these failures in real-time. In practice, we expect Pinpoint to be deployed as a real-time monitor.

A. Failure Detection

To measure detection capabilities, we use the metrics of *recall* and *precision*, two metrics borrowed from information retrieval research. 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 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.

Since a fault always exists in our fault injection experiments, we are not truly testing the precision of our detectors here. So, in this section we only measure the recall of our detectors. Later, in Section V-F we present two experiments designed to test Pinpoint's precision in detecting failures.

The results of our systematic fault detection experiments are summarized in Table I. Both path-shape analysis and component-interaction analysis performed well when detecting 89-100% of major faults. The exception was source code bug injections, which we discuss in detail in Section V-D.

In addition, 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 our tests, Pinpoint correctly detected each of these types of faults. In the case of the overloaded fault, our database would usually saturate before the rest of the system, causing exceptions in the middleware (time outs, 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 are quite stable across different applications and different fault types. Source code bug injection is the only bug that is not detected as well as the others. Even most minor faults are detected by our combined algorirthms. Together, Pinpoint's analyses detect 107 of the 122 exceptions and omissions we injected. In contrast, the efficacy of checking the HTTP error codes and HTML output for errors varies significantly by application. What works well for Petstore 1.1 does not appear to work well for Petstore 1.3. Even used together, the HTTP and HTML monitors detect only 88 of the 122 injected exceptions and omissions. All of these faults are also detected by our path-shape and component interaction analyses.

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 we injected, the severity of the failure, and various aspects of the components into which we injected the failure. As illustrated in Figure 7, we found that the primary factor in whether we detected a failure 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 faults.

The number of peers of a faulty component (how many other components it interacts with) also seems to be an important factor: the more peers a component has, the more likely we are to detect a failure in it. Together, this indicates that Pinpoint is more likely to detect a serious failure than a minor failure, and is more likely to detect a failure in a complicated component interacting with many others than a simple component with relatively little visible behavior.

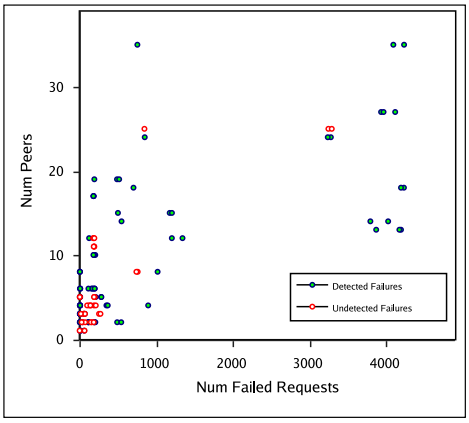


Fig. 7. This figure shows the correlation between the impact of a failure (*Num Failed Requests*), the number of peers our faulty component normally interacts with (*Num Peers*), and whether our component interaction analysis detects these faults. The lower the impact of a failure and the fewer peers the faulty component has, the less likely component interaction analysis is to detect the fault. Though not displayed here, our path shape analysis shows a similar correlation.

While our analyses do not do as well detecting minor failures, it is possible that this is 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 our dynamic thresholding algorithm, or statistically significant for our χ^2 testing.

B. Identifying Failing Requests

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. 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 the low-level heartbeat monitors and user-level monitors cannot directly identify individual failing requests.

We evaluate failing request identification separately from failure detection because, though dependent, they are separatable issues. In particular, we can potentially detect a fault in a system without identifying all (or even most) of the failing requests, though determining which specific requests are failing gives us the ability to judge the significance of the failure and possibly help narrow down a cause.

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.

Monitor	Declared Exc.	Runtime Exc.	Omissions	Src-PS 1.3	PS 1.1	PS 1.3	RUBiS
Path Shape, $\alpha = 2$	78% / 93%	90% / 96%	90% / 100%	12% / 50%	61% / 92%	90% / 91%	-
Path Shape, $\alpha = 4$	75% / 89%	90% / 96%	85% / 96%	7% / 37%	59% / 89%	84% / 88%	68%/-
Comp. Interaction	56% / 89%	68% / 96%	70% / 96%	12% / 37%	44% / 89%	84% / 88%	83/-%
HTTP Errors	48% / 64%	53% / 70%	44% / 65%	10% / 37%	43% / 83%	28% / 32%	-
HTML Monitoring	28% / 40%	24% / 35%	17% / 20%	2% / 13%	2% / 6%	66% / 72%	-

TABLE I

FOR EACH MONITOR, WE SHOW HOW WELL IT DETECTED EACH TYPE OF FAULT ACROSS ALL OUR APPLICATIONS, AND HOW WELL IT DETECTED ALL THE FAULTS IN EACH APPLICATION. IN EACH CELL, THE FIRST NUMBER INDICATES HOW WELL WE DETECTED ALL FAULTS IN THE CATEGORY, THE SECOND RATE IS HOW WELL WE DETECTED MAJOR FAULTS IN THE CATEGORY. SINCE WE ONLY INJECTED SOURCE CODE BUGS INTO PETSTORE 1.3, WE REPORT THOSE FAULTS SEPARATELY, AND DO NOT INCLUDE THEM IN THE OVERALL PETSTORE 1.3 REPORT.

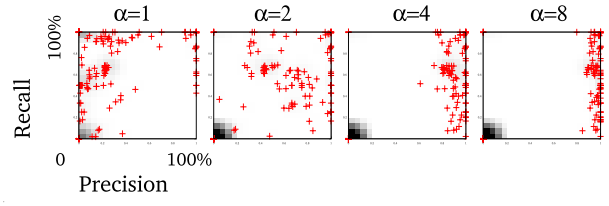


Fig. 8. The precision and recall of discovering faulty requests with path-shape analysis as we vary α. We display these results with a scatterplot of the precision and recall in each of our experiments instead of an averaged ROC-plot in order to emphasize the bimodal distribution of detected and undetected failures. As we raise $alpha$, the average precision and recall of experiments in which we detected failures improves, while the number of experiments in which we do not detect failure at all increases.

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

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 PCFG scores are individually acceptable), and avoid detecting false positives when the system is behaving normally.

In Figure 8, we investigate how adjusting the α parameter affects the recall and precision of our path-shape analysis. As we raise α, we lower recall and improve precision, from a median of $p = 14\%$,$r = 68\%|\alpha = 1$ to $p = 93\%$,$r = 34\%|\alpha = 8$. Note that we do not have to recall all the faulty requests to detect a fault in the system. Not including our source code bug expts, our path-shape analysis detects 83% of the faults when $\alpha = 8$.

C. Localizing Failures

Above, we have evaluated Pinpoint's ability to detect anomalies when a failure occurs. Here, we analyze how well Pinpoint can determine the location of a fault within the system once an anomaly has been noticed.

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 considering all statistically significant suspects as equal.

In this context, with only one fault injection per experiment, recall becomes a boolean metric, indicating whether or not the faulty component is a member of the suspect set. Precision measures how many false suspects there are.

The overall results of our localization tests comparing Pinpoint's detection and localization techniques to each other are shown in Figure 9. In this figure, we show how our well our decision-tree based localization and our component interaction based localization fare in our experiments.

We show the results for three variants of our decision tree, each showing how well the decision tree fares as the requests classified as faulty become more and more "noisy". First, we apply our decision tree to only the faults that we injected failures into. These results are competetive with our component-interaction analysis—the only false suspects that occur are due 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, hiding the true cause of a fault.

Second are the results for our decision tree applied to the requests known to have been injected with a fault or affected by a cascading fault. These results are noisier, and introduce false suspects when we actually localize the cascaded fault. Finally, the results for our end-to-end PCFG and decision tree show the highest miss rate, as we contend with noise both from the PCFG selection mechanism and the cascaded faults. Not represented on this graph, but worth noting is that in our clustered experiments with Petstore 1.1, when the decision tree was not able to pinpoint the exact instance of a component that was causing the problem, it was still able to narrow down the problem, either to the correct class of components, or to the machine the faulty component was running on.

From this we conclude that a decision tree's ability to localize failures depends heavily on the noise in the traces. 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 a user's session as the primary fault, are likely to improve the performance of decision-tree

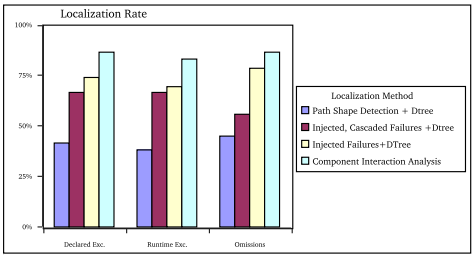


Fig. 9. Localization recall of our techniques per fault type.

localization in real deployments.

D. Source Code Bugs and Structural Behavior

Here, we delve into detail on the source code bugs that we injected into the Petstore 1.3.1 application, describing in detail what kinds of bugs we injected, how we injected them and the impact of the bugs on the system, and finally analyze Pinpoint's performance.

To see whether Pinpoint would be likely to detect faults introduced by minor bugs in source code, we wrote a source-code bug injector for the Java language. We use the Polyglot extensible compiler framework [24] as a base for a Java-to-Java compiler that can inject several different simple bugs, summarized in Table II. While these are all minor bugs, evidence suggests that no bug is so trivial that it does not occur in real software [25]. Some of the common (but much more involved) source code bugs we did not inject includes synchronization and deadlock issues, subtle API incompatabilities, etc. Realistic bug injection of these more subtle issues is an open area for future research.

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.

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, overall, than the HTTP monitors; and moreover, detected a low percentage of the bug injections in absolute terms. 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 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 component's interactions, however. For example, the bug with the largest impact on user sessions, kept the shopping cart from correctly iterating over its contents. While this behavior is internal to the shopping cart and not noticeable by itself, it also causes some JSP tags in the "view shopping cart" page to stop iterating—since the cart never sends its contents to be displayed. Pinpoint detected this failure, and noticed that the JSP tags related to displaying cart items was faulty.

E. Time to Detect Failures

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 and monitors can take longer. The periodicity of running application-test suites usually determines how long it will take them to detect failures. User-level monitors take up to many minutes to notice a failure, as problems that affect user behavior can take some time to affect the measured indicators. The most comprehensive fault monitor, customer service complaints, can take hours to days to report failures, based on the severity of the failure. 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. SBAuth provides user authentication services, verifying user names and passwords and returning user IDs to the rest of the system. We measure the time it takes Pinpoint to detect the error by injecting an exception into SBAuth and recording how long it took for Pinpoint to notice a statistically 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.

In general, 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 effects 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), our injected fault is triggered infrequently, causing Pinpoint to detect failures relatively slowly. Additionally, the nature of the RUBiS load generator causes high variance, as randomized client behavior sometimes trigger faults rapidly and sometimes slowly. As shown in Figure 10, increasing the number of clients improves the time 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. With too many clients, the observations from the system are delayed in

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

TABLE II

OVERVIEW OF SOURCE CODE BUG TYPES AND HOW MANY WE INJECTED INTO OUR APPLICATIONS. NONE ARE DETECTED BY THE JAVA COMPILER.

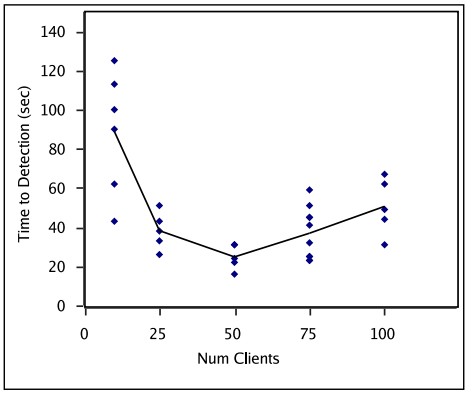


Fig. 10. The time to detect a failure as we vary the client load on the system. The time to detection improves as more of the system is exercised with more clients, then degrades as the extra load induces queueing delays in reporting observations. In these experiments, the current component behavior is based on the prior 15 seconds of observations, and Pinpoint searches for anomalies every 5 seconds.

the report queue, meaning that at any given time Pinpoint is not analyzing the current state of the system. We believe that a more efficient observation system 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 causes a much less pronounced effect 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.

F. Resilience to False Alarms

To determine Pinpoint's resilience against erroneously marking common day-to-day changes to a service as anomalies, we ran two experiments. In one, we significantly changed the load offered by our workload generator—we stopped sending any ordering or checkout related requests.

In our second experiment, 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. Talking with Internet service operators, we confirmed that software upgrades occur often in large sites. While a major software upgrade will change functionality significantly and require Pinpoint to retrain its models of system behavior, these major upgrades occur on the order of months, while minor software upgrades, such as bug fixes, occur on the order of days or weeks.

In both of these experiments, neither our path-shape analysis nor our component-interaction analysis triggered false-positives. In the workload-change experiment, 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.

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 experiments suggest that our fault detection techniques are robust against reporting spurious anomalies when application functionality has not changed.

VI. DISCUSSION

A. The Base-Rate Fallacy

The base-rate fallacy declares that, when looking for rare events, any non-zero false positive rate will overwhelm a detector with even perfect recall. [26] argues that this makes most existing intrusion detection systems unusable.

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. We advocate making the cost of online repair for failures sufficiently low, such that a reasonable degree of "superfluous recovery" will not incur significant overhead.

We have successfully demonstrated this autonomic recovery both in the context of J2EE middleware [9] in which microreboots make it almost free to recovery from transient failures; and in two storage subsystems for Internet services [27], [28] where, in response to another set of Pinpoint's anomaly

detection algorithms, 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, its reasonable to lower detection thresholds to catch more faults (and more false positives), and potentially catch problems earlier.

B. Limitations of Pinpoint

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.

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, a single RMI call from the client to the server. Unfortunately, most of resultant paths we captured were single component calls, with no structure behind them. Thus, when we injected faults into ECPerf, there was no observable change in the path, since there was very little behavior in the path in the first place. While path-analysis did detect some anomalies in the system when a fault occurred, they were not in the "requests" we injected with faults (presumably some state in the application was corrupted and affected later RMI calls). To generalize Pinpoint to this kind of system, we should expand our definition of a path to encompass multiple interactions, though this will likely entail application-specific tailoring.

Multi-Modal Behavior: We found another limitation of our component interaction analysis was the monitoring of components with multi-modal behaviors.

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 web tier (where it mostly initiates name lookups) and another in the application tier, where it mostly replies to lookups. When a component exhibits multiple modes of behavior depending on its physical location, our component interaction model 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 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, possibly more attractive, option is to extend the component-identification scheme to differentiate between components placed, for example, in the web tier vs the backend, thus allowing Pinpoint's analysis to build separate models for each. The difficulty is to ensure the naming scheme captures the right details, without splitting valid component groups apart to the extent that the models lose their ability to validate behaviors across many peers.

C. Other Statistical Learning Techniques

Though we do not claim that the algorithms we use are the best analysis methods for detecting anomalies in an application's structural behavior, we have tried several alternatives as we built our system. Here is a short description of our experiences with these alternate techniques.

Before settling on our PCFG scoring function presented in Section III-B, based on measuring the deviation from expected probabilities, 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.

Recently, together with Peter Bodik, we have used one-class support vector machines (SVM) to detect anomalies in the path shapes and component interaction 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 expection, the standard SVM tree-comparison techniques do not. We are currently investigating whether any feature-weighting additions to SVMs might improve the SVM analysis of path-shapes.

Additionally, we have experimented with using data clustering to localize failures [5], though we now believe decision trees are more appropriate because of their potential to gracefully handle scenarios of multiple failures and failures due to interacting components.

VII. Related Work

Request Tracing: Request paths have been used for black-box monitoring for performance troubleshooting modelling. Aguilera *et al* [29] have used packet-sniffing and statistical analysis to derive the call-graphs between black-boxes. In contrast to the direct tracing done by Pinpoint, this only produces a view of the majority behavior of the system, and thus hides any anomalies existant within the system. Magpie captures the path-like dynamic control flow of a program to localize performance bottlenecks [30]. Unlike Pinpoint, Magpie instruments the system at a very low-level, recording events such as thread context switches and I/O interrupts to build a model of a system's resource consumption and performance characteristics. In contrast to both of these efforts, Pinpoint's focus is on automating the initial detection of

failures, specifically, failures in application-level functionality which may not manifest as a performance problem.

In addition to these research efforts, several commercial products provide request tracing facilities for J2EE systems; PerformaSure and AppAssure are applied in pre-deployment testing and IntegriTea can be applied to a live system, validating our position that it is practical to record paths at a fine level of detail [31]–[33]. However, as far as we know, none of these tools performs application-level failure detection or fault localization.

Anomaly Detection: Anomaly detection has gained currency as a tool for detecting "bad" behaviors in systems where many assumed-good behaviors can be observed, including intrusion detection [34], [35], Windows Registry debugging [36], [37], finding bugs in system code [38], and detecting possible violation of runtime program invariants regarding variable assignment [39] or assertions [40]. Although Ward *et al* previously proposed anomaly detection as a way to identify possible failures for Internet sites [41], they start with a statistical model of a site's performance based on 24 hours of observation, whereas Pinpoint builds models of the site's structural behaviors.

Localizing Faults: Researchers have attacked the problem of localizing faults in many different contexts and with many different approaches. Event-correlation systems for network management [42], [43] and commercial problem-determination systems such as OpenView [44] and Tivoli [45] typically rely on either expert systems with human-generated rules or on the use of dependency models to assist in fault localization [46]–[48]. Brown *et al* [49] have also used dynamic observation to automatically build such dependency models. This approach can produce a rank-ordered list of potential causes, but they are intrusive and require a human to first identify the components among which dependencies are to be discovered. In contrast, Pinpoint can identify the root cause (modulo the coverage of the workload) non-intrusively and without requiring human identification of vulnerable components.

The primary differentiator between the work reported here and Chen *et al*'s fault localization in [12] is that these systems assume the existence of pre-labeled data (*e.g.*, failed or successful requests) and attempt to localize the fault to part of the system. In contrast, the work presented here assumes no prior knowledge of faults, and starts with fault detection.

VIII. Future Directions

Pinpoint detects injected and secondary faults in realistic but small-scale test applications. The value of path-based analysis for failure management has been demonstrated in one production Internet service already [3], and we are currently working with two other large Internet service to apply Pinpoint monitoring to their systems. In addition, versions of Pinpoint have been integrated as fault monitors for two systems at Stanford [27], [28]. Pinpoint has also been distributed to outside researchers as a basis for new experimentation, and is available upon request.

In addition to path shapes and component interactions, we are investigating additional lower-level behaviors that could be analyzed to reveal information about different high-level behaviors, such as database access patterns and structures of persistent data. Monitoring patterns in these behaviors may allow us to detect very different kinds of important failures.

As a monitored system changes, through software upgrades and other major modifications, it is necessary to retrain the reference models of normal behavior. Two simple strategies include 1) explicitly triggering retraining along with major software updates; and 2) constantly learning new models, and keeping many models on hand simultaneously as possibly correct representations of normal behavior. The challenge in choosing a strategy for retraining models is to both avoid false positives as the reference models drift from the system's actual (and correct) behavior, and to avoid false negatives by learning significant faulty behavior as normal. Exploring this challenge in the context of an evolving Internet service is an important avenue of future work.

We believe Pinpoint's techniques may be applicable to other types of systems where building models of application behaviors and policies to detect anomalies and inconsistencies. Applications running on overlay networks, or peer-to-peer applications, are potential targets, though in a widely-distributed application, centralizing the information for analysis would be more challenging. Sensor networks are a useful subclass of peer-to-peer systems whose applications are often data-analysis-centered by nature and in which data-aggregation machinery is already being put in place [50], making sensor nets a potential appealing target as well.

IX. Conclusions

Pinpoint's key insight is that aggregating low-level behavior over a large collection of requests, using it to establish a baseline for "normal" operation, and then detecting anomalies with respect to that baseline is an effective way to detect a variety of faults. Indeed, the key strength of machine learning techniques is to identify patterns and deviations from those patterns, from large collections of undifferentiated data. A large number of independent observations is necessary for this approach to work (and in particular to allow the baseline model to be built), but we argue that in today's componentized applications this is a safe assumption. As long as the system is working mostly correctly most of the time, the baseline model can be extracted from this very behavior. Furthermore, even complex services provide a fairly limited number of discrete interactions to the user, so the number of "legitimate" code paths tends to be much smaller than the number of possible code paths, which gives further intuition behind the success of the approach.

We showed that the specific techniques of PCFG's and decision tree learning to analyze component interaction and path shapes can yield information about a variety of realistic transient faults, with no *a priori* application-specific knowledge. This approach combines the generality and deployability of low-level monitoring with the sophisticated failure-detection abilities usually exhibited only by application-specific high-level monitoring.

Finally, by exploiting the fact that many interactive Internet services are being built on standard middleware platforms, we

can apply these techniques without modifying the applications themselves, with minimal impact on the applications' normal throughput.

We believe Pinpoint represents a useful addition to the roster of dependability-related uses of statistical anomaly detection, and hope to more deeply explore its potential with our ongoing work.

REFERENCES

[1] Business Internet Group San Francisco, "The black friday report on web application integrity," 2003.

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

[3] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," in *The 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, March 2004.

[4] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why do internet services fail, and what can be done about it?" in *4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

[5] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *The International Conference on Dependable Systems and Networks (IPDS Track)*, Washington, D.C., 2000.

[6] M. Y. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer, "Using runtime paths for macro analysis," in *9th Workshop on Hot Topics in Operating Systems*, Kauai, HI, 2002.

[7] *A Microrebootable System: Design, Implementation, and Evaluation*, 2004.

[8] D. Jacobs, "Distributed computing with BEA WebLogic server," in *Proceedings of the Conference on Innovative Data Systems Research*, Asilomar, CA, Jan. 2003.

[9] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox, "Autonomous recovery in componentized internet applications," *To appear in Cluster Computing Journal*, 2004.

[10] C. D. Manning and H. Shutze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA: The MIT Press, 1999.

[11] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[12] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer, "A statistical learning approach to failure diagnosis," in *Proceedings of the International Conference on Autonomic Computing*, May 2004.

[13] T. JBoss Group, "Jboss," http://www.jboss.org/.

[14] S. Microsystems, "Java2 enterprise edition (j2ee)," http://www.javasoft.com/j2ee.

[15] T. Technology, "Tealeaf technology assists priceline.com 'super computer'," January 2003, http://www.tealeaf.com/newsevents/pr/011303.asp.

[16] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and scalability of EJB applications," in *Proc. 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, WA, 2002.

[17] A. Avizienis, J.-C. Laprie, and B. Randell, "Fundamental concepts of dependability," in *Third Information Survivability Workshop*, Boston, MA, 2000.

[18] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[19] R. Chillarege and N. S. Bowen, "Understanding large system failures - a fault injection experiment"," in *In Proceedings of the International Symposium on Fault Tolerant Computing*, June 1989.

[20] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, C. Rajamani, and D. Lowell, "The Rio File Cache: Surviving Operating System Crashes," in *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, 1996.

[21] K. Nagaraja, X. Li, R. Bianchini, R. P. Martin, and T. 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)*. USENIX, March 2003.

[22] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability – a study of field failures in operating systems," in *Proc. 21st International Symposium on Fault-Tolerant Computing*, Montréal, Canada, 1991.

[23] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved, "Saber: Smart analysis based error reduction," in *Proceedings of the International Symposium on Software Testing and Analysis*, Boston, MA, July 2004.

[24] N. Nystrom, M. R. Clarkson, and A. C. Myers, "Polyglot: An Extensible Compiler Framework for Java," in *Proc. of the 12th International Conference on Compiler Construction*, Warsaw, Poland, Apr. 2003.

[25] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in submission. http://findbugs.sf.net/.

[26] S. 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*. ACM Press, 1999, pp. 1–7.

[27] B. Ling, E. Kiciman, and A. Fox, "Session state: Beyond soft state," San Francisco, CA, March 2004.

[28] A. C. Huang and A. Fox, "Cheap recovery: A key to self-managing state," submitted to ACM Transactions on Storage Systems.

[29] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proc. 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, New York, 2003.

[30] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: real-time modelling and performance-aware systems," in *9th Workshop on Hot Topics in Operatings Systems (HotOS IX)*, May 2003.

[31] A. Software, "Appassure," 2002, http://www.alignmentsoftware.com/.

[32] T. Technology, "Integritea," 2002, http://www.tealeaf.com/.

[33] Sitraka, "Performasure," 2002, http://www.sitraka.com/software/performasure/.

[34] D. E. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, vol. 13, no. 2, pp. 222–232, February 1987.

[35] G. Vigna and R. A. Kemmerer, "Netstat: A network-based intrusion detection approach," in *Proc. of the 14th Annual Computer Security Conference*, 1998.

[36] Y.-M. Wang, C. Verbowski, and D. R. Simon, "Persistent-state checkpoint comparison for troubleshooting configuration failures," in *Proc. of the IEEE Conference on Dependable Systems and Networks*, 2003.

[37] E. Kiciman and Y.-M. Wang, "Discovering correctness constraints for self-management of system configuration," in *Proceedings of the International Conference on Autonomic Computing*, May 2004.

[38] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Symposium on Operating Systems Principles*, 2001.

[39] S. Hangal and M. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the International Conference on Software Engineering*, May 2002.

[40] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Sampling user executions for bug isolation," in *The Workshop on Remote Analysis and Measurement of Software Systems*, Portland, OR, May 2003.

[41] A. Ward, P. Glynn, and K. Richardson, "Internet service performance failure detection," in *Web Server Performance Workshop held in conjunction with SIGMETRICS'98*, 1998.

[42] I. Rouvellou and G. W. Hart, "Automatic alarm correlation for fault identification," in *INFOCOM*, 1995.

[43] A. Bouloutas, S. Calo, and A. Finkel, "Alarm correlation and fault identification in communication networks," *IEEE Transactions on Communications*, 1994.

[44] Hewlett Packard Corporation, "HP Openview," http://www.hp.com/openview/index.html.

[45] IBM, "Tivoli business systems manager," 2001, http://www.tivoli.com.

[46] A. Yemeni and S. Kliger, "High speed and robust event correlation," *IEEE Communications Magazine*, vol. 34, no. 5, May 1996.

[47] J. Choi, M. Choi, and S. Lee, "An alarm correlation and fault identification scheme based on osi managed object classes," in *Proc. of IEEE Conference on Communications*, 1999.

[48] B. Gruschke, "A new approach for event correlation based on dependency graphs," in *Proc. of the 5th Workshop of the OpenView University Association: OVUA '98*, 1998.

[49] A. Brown, G. 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*, Seattle, WA, 2001.

[50] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "The design of an acquisitional query processor for sensor networks," in *SIGMOD*, San Diego, CA, June 2003.