Detecting and Localizing Anomalous Behavior to Discover Failures in Component-Based Internet Services

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

16th December 2003

Abstract

Pinpoint is an application-generic framework for detecting and localizing likely application-level failures in component-based Internet services. Pinpoint assumes that most of the system is working correctly most of the time, builds a model of this behavior, and searches for deviations from this model. Pinpoint does not rely on a priori application-specific knowledge to discover failures. To find application-level failures, Pinpoint monitors two low-level behaviors that reflect high-level functionality: path shapes and component interactions. When Pinpoint detects anomalies, it uses its observations of system behavior to correlate the anomalies to their likely causes—a set of components likely to be faulty. In our experiments, Pinpoint correctly detected and localized over 85% of the faults we injected into a J2EE-based Internet service, and was resilient to false-positives in the face of normal changes, such as changes in the workload mix. In comparison, existing application-generic failure detection techniques, such as heartbeats, pings, log file monitoring and HTTP monitoring [24], are easy to deploy because they are application-generic, but cannot detect failures in application-specific functionality, such as silent misbehavior or the erroneous high-level behaviors cited above. In contrast, application-specific, high-level detection techniques, such as end-to-end tests of service functionality, must be custom-built for each application and updated as the application evolves (which can be weekly for some large Internet services). Although such techniques can detect application-level misbehaviors, they often cannot localize the failure to a particular subsystem or application component. An ideal monitor would combine the ease of deployability and maintenance found in application-generic low-level monitors with the more sophisticated detection capabilities of application-specific high-level monitors.

In this paper we present Pinpoint, an application-generic framework for monitoring component-based Internet services, and discovering and localizing failures without requiring a priori knowledge about the application. The insight of Pinpoint is that aggregating a lot of low-level information can reveal high-level behaviors, if (a) the right information is monitored, (b) we can exploit as assumptions certain characteristics about the massively-parallel behavior of large Internet services, and (c) we can assume that the service is working mostly correctly most of the time, allowing us to flag anomalous behavior as 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

1 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)[14] found that of the 40 top-performing web sites (as identified by KeyNote Systems[17]), 72% had suffered user-visible failures in common functionality. Of these, 64% were technical glitches, such as items not being added to a shopping cart or an error message being displayed. These application-level failures, or brown-outs, include failures where only part of the functionality of a site goes down, and failures where functionality is only intermittently unavailable to end-users. Our conversations with Internet service operators confirm that detecting and localizing these failures is a significant problem: one large site estimates that about 93% of the time they spend recovering from application-level failures is spent detecting (75%) and diagnosing them (18%) [9]. Other sites we spoke with agreed that brown-outs can sometimes take days to detect, though they are usually repaired quickly once found.

Fast detection is therefore a key problem. The main challenges facing today’s failure-monitoring techniques can be summarized as follows. Low-level failure detection techniques, such as heartbeats, pings, log file monitoring and HTTP monitoring [24], are easy to deploy because they are application-generic, but cannot detect failures in application-specific functionality, such as silent misbehavior or the erroneous high-level behaviors cited above. In contrast, application-specific, high-level detection techniques, such as end-to-end tests of service functionality, must be custom-built for each application and updated as the application evolves (which can be weekly for some large Internet services). Although such techniques can detect application-level misbehaviors, they often cannot localize the failure to a particular subsystem or application component. An ideal monitor would combine the ease of deployability and maintenance found in application-generic low-level monitors with the more sophisticated detection capabilities of application-specific high-level monitors.

In this paper we present Pinpoint, an application-generic framework for monitoring component-based Internet services, and discovering and localizing failures without requiring a priori knowledge about the application. The insight of Pinpoint is that aggregating a lot of low-level information can reveal high-level behaviors, if (a) the right information is monitored, (b) we can exploit as assumptions certain characteristics about the massively-parallel behavior of large Internet services, and (c) we can assume that the service is working mostly correctly most of the time, allowing us to flag anomalous behavior as 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. Behaviors that fall outside this norm are marked as anomalous and considered to be possible indications of a failure. Finally, Pinpoint correlates anomalies to their probable causes in the system, the likely faulty components. In previous work we showed how aggregating many request paths can be used to correlate observed failures to faulty components [10] and to help localize failures resulting from rapid service evolution [9]. We extend that work in the present paper as follows:

1. We show how existing machine learning techniques for anomaly detection, specifically probabilistic context-free grammars and decision tree learning, can detect and localize likely failures in componentized applications without a priori knowledge of the correct behavior or configuration of the application.

2. We evaluate these techniques by integrated them into an instrumented componentized-application framework (J2EE) and measuring their sensitivity (number of injected faults correctly detected), selectivity (precision of fault localization), and false positive rates.

Pinpoint does not attempt to detect problems before they occur. Rather, we focus on detecting a failure as quickly as possible once it occurs, both to keep it from affecting more users and to prevent cascading faults. Similarly, Pinpoint does not provide root-cause diagnosis, but rather localization of the fault within the system; combined with a simple generic recovery mechanism such as microreboots [6], this is often sufficient for fast recovery.

Section 2 describes Pinpoint’s three-phase approach to detecting and localizing anomalies, and the two types of low-level behaviors that are monitored—path shapes and component interactions. Sections 3.1 and 3.3 explain in detail the algorithms and data structures used to detect anomalies in each of these behaviors. In section 4, we describe an implementation of Pinpoint and an experimental testbed, and present experimental evaluation in section 5, including both an evaluation of Pinpoint’s effectiveness at discovering failures and its resilience to false-positives in the face of normal changes in behavior. We then conclude by discussing related work and future directions.

2 Three-Phase Approach

2.1 Target System

Pinpoint was designed for systems with the following properties:

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 can be thought of as a single large black-box component), or physical node boundaries (e.g. a workstation running a single application, such as the Web server front-end to an Internet service).

2. Queue-like: the system can be characterized as one or more queues whose processing portion (service time) can be broken down as a path, an ordered set of the names of components that participate in the servicing of that queue item.

3. High volume of largely independent requests: request traffic can be characterized as interleaved largely-independent requests (e.g. from different users), and there is sufficiently high request volume that most of the system’s common 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 three-tier structure (Web servers, application logic, persistent store) [4] of many such services. (2) arises because an interactive Internet service can naturally be thought of as a request-processing engine, whose “queues” consist of incoming HTTP requests from end users. (3) arises because of the combination of large numbers of (presumably independent) end users and high-concurrency design within the servers themselves [33].
2.2 Observation, Detection, Localization

The Pinpoint methodology for detecting and localizing anomalies is a three-stage process, shown in Figure 1, of observing the system, detecting anomalies in its behavior, and correlating these anomalies to a probable cause.

1. Observation: We capture the path of each request served by the system: an ordered set of coarse-grained components, resources, and control-flow used to service the request. We extract two specific low-level behaviors from a path: component interactions and path shapes. We demonstrate that analyzing these particular low-level behaviors often reveals high-level behaviors.

2. Analysis: We build a dynamic 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. Behaviors that are anomalous with respect to the dynamic model are flagged as possible failures.

3. Localization: We attempt to identify what features of anomalous paths (components in the path) are correlated with (predictive of) the observed anomaly; these are marked as suspected-faulty components.

Note that Pinpoint does not attempt recovery itself; how and whether to attempt recovery when an anomaly is observed are decisions left to a separate recovery subsystem to which Pinpoint feeds suspected-faulty information.

In the observation phase, Pinpoint captures the runtime path of a request: the control flow, components, and resources associated with servicing the request. Rather than modifying each application or the operating system to collect this information, we concentrate on applications built using standard middleware frameworks such as J2EE. In this case, the instrumentation can be put in the middleware, so that any application using that middleware is automatically instrumented, as long as the application is structured as a set of components with narrow interface boundaries.

In the analysis phase, Pinpoint first builds a dynamic model of “normal” behavior under the assumption that “the system is mostly working correctly most of the time,” and then looks for deviations from this model. Various algorithms can be used for building the model and detecting anomalies. Dynamic models are more likely to be representative of the system’s normal behavior than models generated from human-written specifications and do not have to be manually rebuilt as the system evolves. Building dynamic models quickly is practical because of the high traffic and large number of independent requests: a large fraction of the service’s code base and functionality is exercised in a relatively short amount of time. A busy e-commerce site serving hundreds of requests per second might exercise all its functionality well within a minute. This will not work for many 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 requests most of which behave normally.

There are two axes of anomaly detection: historical analysis looks for anomalies relative to past behavior of the system, whereas peer analysis look for anomalies in the behavior of a particular component relative to the behaviors of its replicated peers. Historical analysis can detect acute failures, but not those that have always existed; peer analysis, which only works for components that are replicated, is resilient to external variations that affect all peers equally (such as work-load changes), but a correlated failure that affects all peers equally will be missed. This is summarized in Table 2.2.

Finally, in the localization phase, Pinpoint uses decision tree analysis to discover the components that are most highly correlated with a particular anomaly detected by historical or peer analysis. Localizing the failure should enable targeted online recovery.

3 Algorithms and Data Structures

From each path collected, we extract two behaviors: path shapes and component interactions. Our dynamic models for path shapes are based on probabilistic context-free grammars; those for component interactions are based on computing a set of expected relative probabilities that one particular component will make a call to another based on training data, and then comparing the similarity of the observed and expected distributions using a chi-square test. We discuss each in detail.

3.1 PCFG’s Detect Path Shape Anomalies

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. 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). Figure 2 shows an example of what the paths shapes look like for a simple system.
Figure 2: The upper half of this figure shows a set of three paths flowing through a system made of three different kinds of components. The lower half of this figure shows the shape of each path. Paths 1 and 2 have the same shape, while path 3 is slightly different.

To detect anomalous path shapes, we use a slightly modified probabilistic context-free grammar (PCFG) [23], a structure used in natural language to calculate the probabilities of different sentences being generated by a particular language and the probabilities of different parses of a sentence. A PCFG consists of:

- A set of symbols (both terminals and non-terminals), \{N^i\}, i = 1, \ldots, n
- A designated start symbol
- A set of grammar rules, \(N^i \rightarrow \zeta^j\), where \(\zeta^j\) is a sequence of zero or more symbols.
- A set of probabilities corresponding to the production rules such that \(\forall i \sum_j P(N^i \rightarrow \zeta^j) = 1\).

A PCFG is used to calculate the probability of a particular parse tree by taking the product of the probabilities of all the production rules used in the parse tree. The probability of a sentence occurring in a language can then be calculated by summing the probabilities of all the legal parsings of that sentence. Figure 3 shows an example PCFG corresponding to the observations of figure 2.

We use PCFGs to model the probabilities of different path shapes in our system, based on the set of observed path shapes. To determine whether any given path should be characterized as anomalous, we use our learned PCFG to calculate a score for the path, based on how probable it is. If this score is below a set threshold, we consider the path to be anomalous.

In our case, we are more interested in scoring the deviation of a path from the norm, which is slightly different than simply taking its probability. For example, consider a simple grammar that represents a language made of the two equally likely sentences, \(ab\) and \(ba\). The probability of either of these normal sentences \(0.5\). However, in a slightly larger grammar made of 100 equally likely sentences, the probability of each sentence will be only \(0.01\), even though none of the sentences can be considered deviant.

Our method for scoring paths is based on the deviation of the probability of a production rule, \(N^i \rightarrow \zeta^j\) from the mean probability of all the rules rooted at \(N^i\). We lower-bound the deviation for a rule to \(0\). The score for the path as a whole is the average deviation of its constituent rules. As shown in Figure 4, this scoring function does a good job of separating normal paths from faulty paths.

To detect anomalies, we use a dynamic threshold that triggers whenever the number of paths scoring above the \(N\)th percentile of our normal score distribution is greater than \(c(1 - N)\). Any path above this \(N\)th percentile will be marked as anomalous. For example, any path with a higher score than we’ve seen before (i.e., above the 100th percentile) will be marked anomalous. Similarly, if \(c = 4\), and more than 8\% of our paths are scoring above the historical 98th percentile, than we mark those paths as anomalous. In our experiments, we use a dynamic threshold based on \(c = 10\).

The time complexity of building a PCFG is \(O(pL)\), where \(p\) is the number of paths in the training set, and \(L\) is the length of a path. The space complexity of building and storing a PCFG is linear in the number of components in the system, though it also depends on the branching factor of the grammar tree and the number of calls made by each component.

To build a model for historical analysis, we build a PCFG based on the last \(N\) hours or days of the system’s behavior, or based on a PCFG captured during a known good period of system behavior. In this analysis, we want to make sure that we observe the system for long enough that we capture most of the different behaviors in the system. This period of time will vary between application domains and between applications.

For peer analysis, we wish to detect paths that are anomalous in comparison to the other paths we are seeing at the

\[
\begin{align*}
S &\rightarrow A & p &= 1.0 \\
A &\rightarrow B & p &= 0.66 \\
A &\rightarrow BB & p &= 0.33 \\
B &\rightarrow CC & p &= 0.4 \\
B &\rightarrow C & p &= 0.2 \\
B &\rightarrow CB & p &= 0.2 \\
B &\rightarrow \$ & p &= 0.2 \\
C &\rightarrow \$ & p &= 1.0
\end{align*}
\]

Figure 3: Example PCFG corresponding to figure 2. \(S\) is the start symbol, \(\$\) is the end symbol, and \(A, B, C\) are the non-terminal and terminal symbols of the grammar.
moment. To do this, we can build a PCFG for the last N minutes of observed path shapes. Since PCFGs can be built incrementally, one compelling option is to use generational PCFGs. As we are detecting anomalies with one PCFG, we are training a new PCFG, rotating it in after N minutes of training.

### 3.2 Decision Trees Localize Path-Shape Anomalies

Once we have detected anomalous paths, we want to localize the anomaly to the set of components most highly correlated with it. Our assumption is that at least some of these correlated components are the cause of the anomaly.

To discover which components seem to be causing observed anomalous paths, we use decision-tree learning. As shown in Figure 5, a decision tree represents a discrete-valued function, where each branch of the tree is a test of some attribute of the input, and where the leaves of the tree hold the result of the function. In our case, the attributes correspond to the path information that Pinpoint collects, such as the names of EJB’s, IP addresses of server replicas in the cluster, etc.

Decision tree learning is the process of deciding what questions to ask at each node of the tree in order to build the most accurate classification function. The general approach to building a tree is to calculate the entropy of the data at each node of the tree, and choose a question that will split the data in a way that minimizes the entropy of the child nodes. The more we lower the entropy, the more accurately and confidently we can classify the data, though we do have to worry about overfitting the tree to errors in the training data. In Pinpoint, we use the ID3 algorithm for learning decision trees [27].

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. For example, one rule that would be generated from the tree in Figure 5 is IF ((isRound == YES) AND (isShaded == YES)) THEN classify as A. We rank each of these rules based on the number of paths that they correctly classify as anomalous. From these rules, we trivially extract the features that are correlated with failures.

When localizing a failure, 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 the components and resources used by the path.

Note that decision trees can represent both disjunctive hypotheses, meaning that we can learn hypotheses that describe multiple independent faults, as well as conjunctive hypotheses, meaning that we can 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 individual components. More interestingly, it allows us to avoid specifying a priori the exact fault boundaries in the system. For example, rather than assuming that we want to localize to a particular instance of a component, 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.

Once a decision tree to classify the data on hand has been learned, we look at which attributes were discovered to be the best classifiers. These components and resources are the ones most likely to be related to the actual fault. The better they act as a classifier, the more correlated they are to the anomalous paths.
3.3 Dynamic Call Structure Characterizes Component Interactions

The second low-level behavior that we analyze is component interactions. We represent the behavior of a component $A$ as the links by which runtime paths enter $A$ from other components, or leave component $A$ to other components. We weight each link by the proportional number of times that link is used. The total weight of all these links sums to 1. A component’s interactions are the relative frequencies of these links. If these relative frequencies do not look the way they used to (historical analysis) or do not look like the relative frequencies of similar components (peer analysis), an anomaly as flagged. If more than one component is found to be anomalous, we can attempt to correlate these anomalies to specific features of the components.

While there is overlap in the failures that can be detected with component interaction and path-shape analysis, there are important differences as well, since path-shape analysis looks at each request path individually whereas component interaction analysis provides a view at the system’s behavior across many client requests. A failure in a password checker that denied access to all users would not be detected by the path-shape analysis, since it is normal for at least some requests to be denied by the password checker, but component interaction analysis would reveal that the overall interactions between the password checker and the next component in the system changed considerably. Conversely, an occasional error that affects only a few requests might not significantly change the proportions of component interactions, but path-shape analysis would detect changes in the individual paths.

We generate our model of normal behavior by averaging the links and weights across a number of samples. For historical analysis, the model of good behavior for each component instance is generated from samples of its behavior over a representative period of time. For example, an Internet site with pronounced weekly cycles of behavior might generate a historical model based on samples of component behaviors of the component taken during the same time period a week ago. For peer analysis, the model of good behavior for a component is generated from the current behavior of all its replicated peers. For example, the good behavior model for a front-end web server on an Internet site would be created by averaging the current behavior of all the front-end web servers.

We then measure the deviation between a single component’s behavior and our model of normal behavior using the $\chi^2$ test of goodness-of-fit:

$$ Q = \sum_{i=1}^{k} \frac{(N_i - w_i)^2}{w_i} $$

where $N_i$ is the weight of link $i$ in our component instance’s behavior; and $w_i$ is the expected weight of the link
Anomaly Detection

We can achieve this localization using the same decision-tree algorithm described in Section 3.2. When applying the decision-tree to the localization of failures in components, our inputs are the components and our classification attributes are any distinguishing features we have observed about the components, such as what type of component it is, what operating system it is deployed on top of, etc.

Table 2 summarizes Pinpoint’s analysis techniques.

Table 2: Summary of data structures and algorithms used for anomaly detection and correlation for each of the low-level behaviors (path shapes and component interactions) captured.

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Anomaly Det.</th>
<th>Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path shapes</td>
<td>Probabilistic Context-Free Grammar</td>
<td>Decision tree on components in paths</td>
</tr>
<tr>
<td>Component interaction</td>
<td>( \chi^2 ) Test of goodness-of-fit</td>
<td>If needed, decision tree on attributes of components</td>
</tr>
</tbody>
</table>

4 Experimental Setup

We connected Pinpoint to a widely-used middleware system, J2EE, and injected various faults into applications running on top of this middleware to evaluate Pinpoint’s feasibility as a real technique. In this section, we describe how the three-phase approach—instrument/collect, analyze, correlate—maps onto our prototype, and discuss the workload and faultload used for evaluation, before presenting our results in Section 5.

4.1 Instrumentation, Analysis and Correlation

J2EE (www.javasoft.com/j2ee) is a widely-adopted middleware standard for constructing enterprise applications from reusable Java modules, called Enterprise Java Beans (EJBs). A J2EE application server provides a standard runtime environment, containers for instantiating EJB’s, and a mechanism for connecting the EJB-based application to a Web server with which end users interact (see Figure 6 for a schematic view). We modified JBoss, a widely-used open-source implementation of a J2EE application server, to support Pinpoint’s instrumentation. For each client request entering the system (via HTTP servlets or Java server pages, depending on the specific application deployed), Pinpoint records the request’s path through the system, noting each component touched and the order in which they are touched.

Key observation points instrumented in JBoss include: a request entering the system via HTTP; method-call entry and exit for each EJB used by the request; method-call entry and exit for Java RMI (remote invocation), allowing us to instrument distributed J2EE applications; SQL queries to a persistent-state database made via Java Database Connector (JDBC) drivers; and a request leaving the system via a dynamically-generated Web page called a Java Server Page (JSP). Note that these observation points are generic to any J2EE application.

The observations are sent over the local network to a centralized node for logging and analysis; this task is done in a dedicated thread per Java virtual machine, to keep it out of the critical path of servicing the request. Our instrumentation adds 1150 new lines of code spread over 17 Java source files. With our unoptimized implementation, collecting the observations increases request latency by \( 2 - 40 \) ms depending on the length of the path, and decreases system throughput by \( 17\% \), from 35 to 29 requests/sec on our baseline testbed; the deployment of commercial instrumentation packages such as IntegriTea (www.tealeaf.com) on large sites such as Price-line.com suggests that such fine-grained instrumentation is even more practical if some engineering effort is focused on the implementation.

We also built a plugin-based analysis engine, and created plugins corresponding to the anomaly-detection algorithms described in sections 3.1 and 3.3. Analysis can be either on-line, receiving observations directly from the system being monitored, or off-line (trace-based), playing back previously recorded observations. Because our own experiments...
involved 100s of runs of our applications, requiring more than 16 hours of cluster time, we used an off-line analysis for the experiments in this paper, collecting the application traces first, and analyzing them off-line separately.

4.2 Application and workload

We would like to test Pinpoint in a large, live Internet service to detect actual failures. Though we are in the process of just such a deployment, it is still only in the beginning stages. In the meanwhile, we test Pinpoint on three small-scale J2EE applications that we have deployed in our own testbed:

- Petstore version 1.1 is Sun’s sample J2EE application that simulates an e-commerce web site (storefront, shopping cart, purchase, order 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. The main disadvantage of this application is its small size and simplicity; its main advantage is that we have modified it to run across a 4-node cluster.

- Petstore version 1.3.1 is a significantly re-architected version of Sun’s original sample application. Version 1.3.1 is actually a suite of applications, including order processing, administrative, and supply-chain—47 components in all, 310 files, and 10K lines of code. Because of the new architecture, we were unable to cluster Petstore 1.3, however, its complexity still makes it very useful as test application.

- RUBiS 1.4.1 is an auction-website, developed at Rice University for experimenting with different software architectures. RUBiS contains over 500 Java files, and over 25k lines of code. More important for our purposes, RUBiS has 6 EJBs and several servlets. The main advantage of RUBiS is that it comes with a thorough load generator that uses a probabilistic transition table to exercise the system instead of pre-recorded traces.

With all our applications, we run the database and the Pinpoint observation engine continue to run on separate machines from the application.

For the Petstore applications, the workload presented by our load generator simulates traces of several parallel, distinct user sessions with each session running in its own client thread. Borrowed from the TPC-W load generator specification[29], each client waits for a random “think time” (negative exponential with a mean of 7 seconds) between each request. A session consists of a user entering a site, performing various operations, such as browsing or purchasing, and then leaving the site. We choose session traces so that the overall load on 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 artificially block all the traces early in their life cycle.

Our second application, RUBiS [8], is an open-source web-based auction application developed at Rice University and modeled on eBay. RUBiS contains 582 Java files and about 26K lines of code; in its default configuration, it has about 33,000 items for sale, distributed among 40 categories and 62 regions. There is an average of 10 bids per item, or 330,000 entries in the bids table. The users table has 1 million entries.

4.3 Fault Load

To test whether Pinpoint can detect and localize failures, we inject a series of faults into our applications. In turn, we injected each Enterprise-Java Bean component in the system with an expected exception, an unexpected exception, and with an omitted call fault. We chose to inject these particular failures because they test the range of an application’s ability to cope with failures; an application should be able to gracefully handle an expected exception, whereas a null call should never occur.

To selectively inject faults, we interpose on the J2EE application server to intercept method calls into EJB’s. We inject three kinds of failures. Declared exceptions are Java exceptions that appear in the signatures of component methods, and which applications are expected to handle gracefully and/or mask from the end user. Undeclared exceptions, or runtime exceptions, include null pointer exceptions, di-
vide by zero exceptions, assertion failures, and other erroneous runtime conditions, which may or may not be handled gracefully by the application. Omission faults are caused by intercepting all method calls to a target component and replacing them with null calls and/or returning a null value from the call. This effectively blocks all access to the target component’s functionality. The effects of omission fault are less predictable than those of declared or runtime exceptions.

Together, these injected faults cause both user-visible failures and in many cases additional secondary failures. For example, sometimes our injected failure does not cause an HTTP error to be returned to our load generator, but instead corrupts some session information. Our load generator then continues with its trace, eventually triggering a secondary fault an HTTP error at some later time. As an example of a mild failure, faults injected into a the InventoryEJB component of Petstore 1.1 are masked by the application. The only affect seen by the user is that items appear to be “out of stock.” At the other end of the spectrum, injecting a fault into the ShoppingController component in Petstore 1.3 prevents the user from seeing the website at all, instead displaying an internal server error for any request. None of the faults we injected caused our applications or application servers to crash or hang.

5 Experimental Results

We next present a set of experiments to show how Pinpoint fares in detecting and localizing failures in a realistic, though small, Internet service. In addition, we test how Pinpoint’s false-positive rate may fare in the face of “normal” anomalies, such as workload changes and software upgrades. In the experiments presented here, we focus on using historical path-shape and component interaction analyses to detect failures.

5.1 Individual examples of failures

First, let us look at a few real examples of how our techniques catch failures. Figure 7 shows one normal and one anomalous path for the commitorder request type. This failure was not directly injected by us, but rather is a secondary fault that was caused by a corruption of session state caused by our fault injection.

These two paths are obviously different, but it is not immediately obvious that the one on the right, asking for the user to sign in, is wrong. By inspecting the user’s session trace, however, it becomes clear that it is in fact a failure. Prior to coming to this page, the user has already created a new account, logged in, validated his billing address and is on the last stage of purchasing when he is asked again to sign in. Being already logged in and interacting with the system as if he is logged in, the user should not have to reauthenticate. Because of a fault we injected during the account creation process however, the system loses track of this user session information.

Our PCFG, by tracking the kinds of paths that are used per-request type flags this as a very anomalous path. In contrast, watching for HTTP error codes does not catch this page, nor is it likely that scanning the returned HTML page for errors would detect a problem.

Once we have detected anomalous paths, we use the decision-tree learning algorithm to localize the fault to the parts of the system that are likely to be causing it. Figure 8 shows the decision-tree resulting from a path-shape analysis of a failure in the TheInventory component.. Here, we injected a fault into the TheInventory component running on just one machine in the Petstore 1.1 cluster. The decision-tree learning algorithm chose the component name attribute as the most important classifier: if the path does not use TheInventory, it will succeed. If TheInventory is used, the second attribute is tested, whether or not the path uses a particular machine. If it does then the path is likely to be one of our detected anomalous paths. By reading this generated decision-tree, we easily discover where a fault is.

In one of our experiments, we injected a fault directly into the CatalogEJB component of Petstore 1.3. To detect the failure, we ran our component interaction analysis and, as shown in Figure 9, that the CatalogEJB component was very anomalous. In this case, we see that CounterEJB is not only the most anomalous, but that it is significantly more anomalous than other components in the system. The other components have some anomalies as well: when CounterEJB fails, their behavior is affected as well, though not as seriously as CounterEJB’s own.

Using our component-interaction analysis to diagnose failures, we see a different view of the system. Rather than inspecting individual paths and slicing across components,
Figure 8: A decision-tree learned for classifying faulty paths in one of our experiments.

- 1 9.41 CatalogEJB
- 2 1.09 ShoppingCartEJB
- 3 0.34 ShoppingControllerEJB
- 4 0.12 JspServlet
- 5 0.02 MainServlet

Figure 9: The components in Petstore v1.3.1, ranked according to their degree of anomaly. When comparing the $\chi^2$ goodness-of-fit scores of components, we normalize them by their thresholds such that a score greater than 1 is anomalous. For brevity, we only show the top five ranking components. The analysis clearly (and correctly) discovers the component CatalogEJB as being anomalous.

we look at individual components and slice across all the requests flowing through it. Though this usually makes localizing the problem to a specific component simpler, it is conversely more difficult to tell which requests and which users might be affected by a given component’s failure.

5.2 Aggregate results

Both our analyses performed well at detecting these failures, detecting over 90% of the faults we injected across our experiments. We tested our component interaction and path-shape analysis, against two other low-level techniques: HTTP error code monitoring and log monitoring. HTTP error code monitoring looks at the HTTP return code of every request, and marks it as having failed if it notices an error. To detect failures, log monitoring simply watches the server logs for keywords. In these experiments, we monitored for the keywords “exception”. We also ran a log-monitor that checked for exceptions in the server logs. We did not include it in our graphs because of its high false-positive rate—it erroneously detected failures in every one of our control experiments.

The comparison of each technique’s miss rates are shown in Figure 10. Component interaction analysis and path-shape analysis complemented each other across our experiments. Component interaction detected all the failures in Petstore 1.1., while path-shape analysis detected all the failures in Petstore 1.3 and in RUBiS.

We should note that though HTTP error detection did detect failures in most of our failures, it tended to discover secondary fault that generated HTTP errors and missed most of the primary faults. With just the information from the HTTP errors, one would have a more difficult time tracking down the initial cause of the problem. In contrast, path-shape analysis detects both the primary and the secondary faults.

Figure 11 shows the detection rate of our monitors for each of the types of faults. Pinpoint’s detection rate does not significantly vary based upon the type of fault we injected. This is in contrast to HTTP error monitoring, which significantly varied in its coverage.

In Figure 12, we investigate how adjusting percentile-based thresholding parameter affects the accuracy and false-positive rate of our path-shape analysis. We see that we have a relatively low false-positive rate of 1-2%, even as we accurately detect over 80% of the failed requests in the system. It is important to note that we do not have to detect all failed requests to detect all the faults in our experiments. Even detecting just a few of the anomalous requests caused by a fault is often enough. We have marked on the figure the points on the curve where we detect over 90% and 100% of the faults in our experiments.

The overall results of our localization tests comparing Pinpoint’s detection and localization techniques to each other are shown in Figure 13. 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 fairs as the requests classified as faulty become more and more “noisy”.

Figure 10: The miss rate of our analysis techniques with HTTP error monitoring for each of our applications. A bug with our testbed prevented us from capturing the correct detection rate of HTTP error monitoring for RUBiS.
This graph compares the relative detection rate of our tested monitoring techniques for each type of fault. While both path-shape and component interaction analyses do well across all our injected fault types, http error code monitoring vary depending on the type of fault.

Detection rate and false positive rate are affected as we change the thresholds used to detect failures.

This graph compares the relative performance of our localization techniques for each type of fault.

First, we apply our decision tree to only the faults that we injected failures into. These results are competitive with our component-interaction analysis—the only misdiagnoses or false positives 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.

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 what we measure as misdiagnoses when we actually diagnose 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.

5.3 False positives

Anomalous behavior is not always an indicator of a failure. We use the term false positive to refer to the erroneous flagging of a condition as a failure when it actually is not. Two possible causes of false positives are system upgrades (both hardware and software) and significant variations in workload.

To test Pinpoint’s resilience against erroneously marking these “normal changes” 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.

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 our threshold according to the $c^2$ test. Though not comprehensive, these two experiments indicate that our fault detection techniques are robust against reporting spurious anomalies when application functionality has not changed.

In addition to these two sources of false-positives we tested against, another interesting cause of false positives occurs when an application switches to a different, but correct, operating mode because of external conditions. For example, under heavy load, CNN.com dramatically simplifies its “headline” pages rather than deny service [19]; such a large-scale change would likely appear anomalous when it is triggered. If mode switching occurs often, it should be classified as normal behavior, but activation of rarely-exercised modes will confuse our historical analysis (though not our peer analysis.)

There are various ways to deal with these remaining false positives. When the change to the system can be predicted, such as a controlled software upgrade, or a predictable change in workload, one approach is to give system operators the ability to notify Pinpoint of modifications that might trigger false positives, such as network or software upgrades. Pinpoint could be instructed to ignore false positives for a bounded time period (or decrease its sensitivity thresholds) to allow the system to “re-settle”, although as a result a true failure during this interval may be mistakenly ignored.

A more interesting way to handle false positives is not to attempt to filter them at all. If the cost of online repair can be made sufficiently low, attempts at “superfluous recovery” may not incur enough overhead to cause damage, as long as they do not occur too often. This has been successfully demonstrated in the context of a storage subsystem for Internet services [21] in which any replica can be rapidly rebooted at any time without impacting performance or correctness. We are in the process of exploring this approach by integrating Pinpoint with a microreboot recovery mechanism for J2EE [7].

6 Related Work

Today’s technology for fast fault detection leaves much room for improvement: Oppenheimer finds that earlier detection of problems might have avoided or mitigated 65% of user-visible failures in one large-scale Internet service [25], but the required higher-level monitors were prohibitively expensive to build and maintain. Most sites monitor “core behavior” metrics such as click-through rates, but these are useless in localizing faults.

Whole-program paths [18] and Magpie [2] both capture path-like dynamic control flow of a program to localize performance bottlenecks. Magpie uses stochastic context free grammars to model system behavior at a lower level than we do, focusing on resource consumption and performance modeling. Several commercial products provide request tracing facilities for J2EE systems; PerformaSure (www.sitraka.com/software/performasure), AppAssure (www.alignmentsoftware.com) are applied in pre-deployment testing and IntegriTea (www.tealeaf.com) can be applied to a live system, validating our position that it is practical to record paths at a fine level of detail. As far as we know, none of these tools performs application-level failure detection or failure diagnosis.

Localizing failures has also been challenging. Event-correlation systems for network management [28, 3] and commercial problem-determination systems such as OpenView (www.hp.com/openview) and Tivoli (www.tivoli.com) typically rely on either expert systems with human-generated rules or on the use of dependency models to assist in fault localization [34, 11, 15]. Aguilera et al. [1] and Brown et al. [5] have used dynamic observation to automatically build such dependency models. These approaches 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 annotation to identify the components.

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 [12, 30], Windows Registry debugging [31], finding bugs in system code [13], and detecting possible violation of runtime program invariants regarding variable assignment [16] or or assertions [20]. Although Ward et al. previously proposed anomaly detection as a way to identify possible failures for Internet sites [32], they start with a statistical model based on 24 hours of observing the system, whereas Pinpoint builds and adjusts its model dynamically.
7 Future Directions

Pinpoint detects injected and secondary faults in a realistic but small-scale test application. The value of path-based analysis for failure management has been demonstrated in one production Internet service already [9], and we are currently in discussion with a large Internet retailer to apply Pinpoint, initially to their logged traces of instrumentation and possibly later to a live system.

Pinpoint has particular synergy with applications that allow fast partial recovery from transient failures. In such cases, false positives can nearly be ignored, since the cost of recovery is so low that attempting it even if there was no real failure does not materially impact performance. Pinpoint has successfully been used to provide a degree of self-management to a storage subsystem designed for Internet applications [21].

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. For example, tagging each path with the ID’s of specific database table rows that it touches would allow correlation of requests that appear to be independent but are actually coupled through shared persistent state, allowing us to detect cases in which one request corrupts a database table row but the effect is noticed much later by a different request. We expect to report on results using this technique in the near future.

We believe Pinpoint’s techniques may be applicable to other types of systems besides interactive Internet services. Applications running on overlay networks, or peer-to-peer applications, could use Pinpoint if the application has a concept of request paths and if instrumentation can be put in place to capture them; centralizing the information for analysis would be more challenging in a widely-distributed application. 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 [22], making sensor nets a potential appealing target as well. [26] are using request tracing for localizing configuration-related failures in large Internet systems; they have not yet attempted Pinpoint-like statistical analysis, but most of the infrastructure for doing it is now in place.

8 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. In particular, we showed that analyzing 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 genericity and ease of deployability of low-level monitoring with the sophisticated failure-detection abilities usually exhibited only by application-specific high-level monitoring.

Pinpoint assumes that the system is working mostly correctly most of the time, and that under normal operating conditions, a large fraction of the code paths representing the system’s code paths representing the system’s functionality are exercised over a relatively short period of time, allowing the rapid creation of dynamic models of baseline behavior. This is a reasonable assumption for Internet services, which though complex typically provide a fairly limited number of discrete interactions to the user. We also exploited the fact that many interactive Internet services are being built on standard middleware platforms: by modifying the middleware, any application using the middleware platform benefits, and collecting the required data from a live system minimally affects its throughput, validating the feasibility of the approach.

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


