Pinpoint: Problem Determination in Large, Dynamic Internet Services

Mike Y. Chen, Emre Kıcıman*, Eugene Fratkin*, Armando Fox*, Eric Brewer
Computer Science Division, University of California, Berkeley
**Computer Science Department, Stanford University*
{mikechen, brewer}@cs.berkeley.edu, {emrek, fratkin, fox}@stanford.edu

Abstract

Traditional problem determination techniques rely on static dependency models that are difficult to generate accurately in today's large, distributed, and dynamic application environments such as e-commerce systems. In this paper, we present a dynamic analysis methodology that automates problem determination in these environments by 1) coarse-grained tagging of numerous real client requests as they travel through the system and 2) using data mining techniques to correlate the believed failures and successes of these requests to determine which components are most likely to be at fault. To validate our methodology, we have implemented Pinpoint, a framework for root cause analysis on the J2EE platform that requires no knowledge of the application components. Pinpoint consists of three parts: a communications layer that traces client requests, a failure detector that uses traffic-sniffing and middleware instrumentation, and a data analysis engine. We evaluate Pinpoint by injecting faults into various application components and show that Pinpoint identifies the faulty components with high accuracy and produces few false-positives.

Keywords: Problem Determination, Problem Diagnosis, Root cause Analysis, Data Clustering, Data Mining Algorithms

1. Introduction

Today's Internet services are expected to be running 24x7x365. Given the scale and rate of change of these services, this is no easy task. Understanding how any given client request is being fulfilled within a service is difficult enough; understanding why a particular client request is *not* working—determining the root cause of a failure—is an even greater challenge.

Internet services are very large and dynamic systems. The number of software and hardware components in these systems increases as new functionalities are added and as components are replicated for performance and fault tolerance, often increasing the complexity of the system. Additionally, as services become more dynamic, e.g., to provide personalized interfaces and functionality, the way that client requests are serviced becomes more and more varied. With the introduction of Internet-wide service frameworks encouraging programmatic interactions between distributed systems, such as Microsoft's .NET [20] and Hewlett-Packard's E-Speak [15], the size and dynamics of a typical Internet service will only continue to increase.

Today, a typical Internet service has many components divided among multiple tiers: front-end load balancers, web servers, application components, and backend databases, as well as numerous (replicated) subcomponents within each [22]. As clients connect to these services, their requests are dynamically routed through this system. Current Internet services, such as Hotmail [8], a web-based email service, and Google [7], a search engine, are already hosted on thousands of servers and continue to grow.[1] The large size of these systems, together with the increase in their dynamic behavior, means an increase in their complexity and more potential for failures to occur due to unanticipated "interaction" faults among components. That these failures actually occur is evidenced by the fact that few services deliver availability over 99.9% in a real-world operating environment.

1.1. Background

The focus of this paper is problem determination: detecting system problems and isolating their root causes. Current root cause analysis techniques use approaches that do not sufficiently capture the dynamic complexity of large systems, and they require people to input extensive knowledge about the systems [24, 4]. Most root cause analysis techniques, including event correlation systems, are based on static dependency models describing the relationships among the hardware and software components in the system. These dependency models are used to determine which components might be responsible for the symptoms of a given problem [5, 25, 6, 13]. The first major limitation

[1] Hotmail 7000+ [11], Google 8000+ [14]

of traditional dependency models is the difficulty of generating and maintaining an accurate model of a constantly evolving Internet service. Their second major limitation is that they typically only model a logical system, and do not distinguish among replicated components. However, since large Internet services have thousands of replicated components, there is a need to distinguish among them to find the instance of the component that is at fault.

1.2. A Data Clustering Approach

We propose a new approach to problem determination that better handles large and dynamic systems by:

1. Dynamically tracing real client requests through a system. For each request, we record its believed success or failure, and the set of components used to service it.
2. Performing data clustering and statistical techniques to correlate the failures of requests to the components most likely to have caused them.

Tracing real requests through the system enables us to support problem determination in dynamic systems where using dependency models is not possible. This tracing also allows us to distinguish between multiple instances of what would be a single logical component in a dependency model.

By performing data clustering to analyze the successes and failures of requests, we attempt to find the combinations of components that are most highly correlated with the failures of requests, under the belief that these components are causing the failures. By analyzing the components that are used in the failed requests, but are not used in successful requests, we provide high accuracy with relatively low number of false positives. This analysis detects individual faulty components, as well as faults occurring due to interactions among multiple components. This approach is well suited for large and dynamic Internet services because:

- Live tracing of client requests allows us to analyze both the logical and physical behavior of a system. Because tracing does not require human intervention to adapt to system changes, Pinpoint scales to constantly evolving Internet services.
- Data clustering techniques allow us to quickly summarize the large amount of collected traces, and correlate them with believed failures. Because the Pinpoint analysis is automated, it does not require extra effort on the part of service developers and operators to run on large services.

The Pinpoint approach does make two key assumptions about the system being measured. First, the system's normal workload must exercise the available components in different combinations. For example, if two components were always to be used together, a fault in one would not be distinguishable from a fault in the other. Secondly, our data clustering approach assumes that requests fail independently—they will not fail because of the activities of other requests. These assumptions are generally valid in today's large and dynamic Internet services. Service requests tend to be independent of one another, due to the nature of HTTP, and the highly replicated nature of Internet service clusters allows components to be recombined in many ways to avoid single-points of failure.

We have implemented our approach in a prototype system called Pinpoint, and used Pinpoint to identify root causes in a prototype e-commerce environment based on the Java 2 Platform Enterprise Edition (J2EE) demonstration application, PetStore [21]. We use a workload that mimics the request distribution of the TPC web e-commerce ordering benchmark (TPC-WIPSo) [2]. We instrumented J2EE server platform to trace client requests at every component, and had a fault-injection layer that we used to inject 4 types of faults. The results demonstrate the power of our approach. We were able to automatically identify the root causes of single-component failures 80-90% of the time with an average rate of 40-50% false positives without any knowledge of the components and the requests. The rate of false positives is better than other common approaches that achieve similar accuracies.

The contributions of this paper are: 1) a dynamic analysis approach to problem determination that works well and 2) a framework that enables separation of fault detection and problem determination from application-level components. The rest of this paper describes our approach to automating problem determination and the experimental validation of this work. Sections 2 and 3 present a detailed design and implementation of a framework, Pinpoint, that uses our approach. Section 4 describes our experimental validation. Section 5 discusses limitations of Pinpoint and previous work and future work in this area. We conclude in Section 6.

2. The Pinpoint Framework

To validate our data clustering approach to problem determination, we designed and implemented Pinpoint, a framework for problem determination in Internet service environments. Our framework, shown in Figure 1, provides three major pieces of functionality to aid developers and administrators in determining the root cause of failures:

Client Request Traces: By instrumenting the middleware and communications layer between components, Pinpoint dynamically tracks which components are used to satisfy each individual client request.

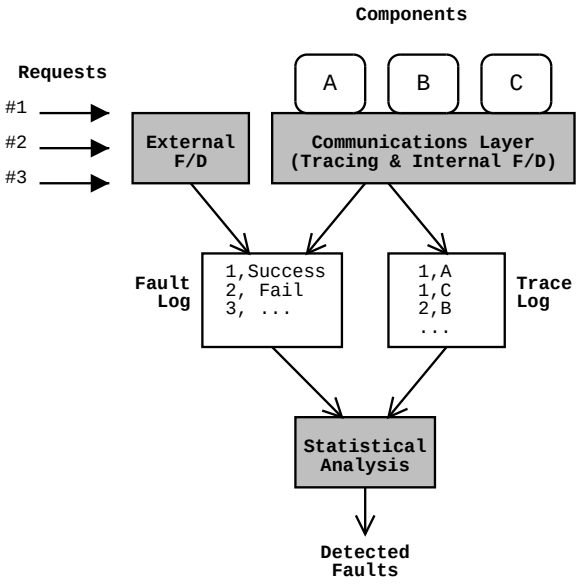

Figure 1. The Pinpoint Framework.

Failure Detection: Pinpoint provides both internal and external monitoring of a system to detect whether client requests are succeeding or failing. Internal fault-detection is used to detect assertion failures and exceptions, including errors that are later masked by the system. External fault-detection is used to detect end-to-end failures not otherwise detectable.

Data Clustering Analysis: Pinpoint combines the data from tracking client requests with success and failure data for each request and feeds it into a data analysis engine to discover faulty components and component interactions.

2.1. Client Request Tracing

As a client request travels through the system, we are interested in recording all the components it uses, at various granularities. At a coarse granularity, we are interested in the machines and, depending on the size of the service, the clusters being used. At a finer granularity, we are interested in logging individual software components, component versions, and, if practical, even individual data files (such as database tables, and versions of configuration files). Our goal is to capture as much information about possible differentiating factors between successful and failed requests as is practical.

When a client request first arrives at the service, the request tracing subsystem is responsible for assigning the request a unique ID and tracking it as it travels through the system. To avoid forcing extra complexity and excessive load on the components being traced, the tracing subsystem generates simple log outputs in the form of <request_ID, component_ID> pairs. This information is separately collated into complete lists of all components each request touched.

By modifying the middleware beneath the application components we are interested in, we can record the ID of every request that arrives at a specific component without having any knowledge of the applications and without modifying the components. When an application component makes a nested call to another component, the middleware records that another component is about to be used, and forwards the request ID to the next component along with the call data. The changes required to implement this subsystem can often be restricted to the middleware software alone, thus avoiding modifying application-level components. Whether this is possible depends on the specific middleware framework used and details of the inter-component communication protocol.

2.2. Failure Detection

While the tracing subsystem is recording components being used by client requests, an orthogonal subsystem is attempting to detect whether these client requests are successfully completing. Though it is not possible to detect all failures that occur, some failures are more easily noticeable from either inside or outside of the service. Therefore, our framework allows for both internal and external failure detection to be used.

Internal failure detection is used to detect errors that might be masked before becoming visible to users. For example, a frontend failure that gets replaced by a hot swap may have no externally visible effects. Though these failures do not become visible to the users, system administrators should still be interested in tracking these errors to repair the systems. Internal failure detectors also have the option of modifying the middleware to track assertions and exceptions being generated by application components.

External failure detection is used to detect faults that will be visible to the user. This includes complete service failures, such as network outages or machine crashes. External detection can also be used to identify application-specific errors that generated user-visible messages.

Whenever a failure or success is detected, the detection subsystem logs this along with the ID of the client request. To be consistent with the logs of the client tracing subsystem, the two subsystems must either pass client request ids between each other, or use deterministic algorithms for generating request IDs based on the client request itself.

Client Request ID	Failure	Component A	Component B	Component C
1	0	1	0	0
2	1	1	1	0
3	1	0	1	0
4	0	0	0	1

Table 1. A sample input matrix for data analysis

2.3. Data Analysis

Once the request traces and failure/success logs are available, they are given to Pinpoint's analysis subsystem. A sample input is shown in Table 1. The data analysis uses a data clustering algorithm to discover sets of components that are highly correlated with failures of requests.

Clustering algorithms structure data by grouping similar data points together. In our analysis, we calculate similarity based on how often components are and are not used together in requests. The details of the clustering algorithms we used in our implementation are presented in Section 3.3.

Before running this clustering analysis, we first must prepare the data. During the logging stage, our requests are indexed by request ID, with each request ID matched to the components used in that request. We transpose this data for the clustering process and instead index by component, matching each to the requests it was used in. We also add a failure data point and mark it with all the requests that we believe have failed.

The clustering algorithm then groups these components and the failure data point together. The interesting result, for our purposes, is the set of components clustered with our failure data point. These are the components whose occurrences are most correlated with failures, and hence where the root cause is likely to lie.

3. Pinpoint Implementation

We have implemented a prototype of Pinpoint on top of the J2EE middleware platform, a network sniffer, and an analyzer based on standard data clustering techniques. Our prototype does not require any modifications to be made to J2EE applications. Only our external fault detection module requires application-specific checks—and these do not require modification of application components. For this reason, Pinpoint can be used as a problem determination aid for almost any J2EE application.

3.1. J2EE Platform

Using Sun's J2EE 1.2 single-node reference implementation as a base, we have made modifications to support client request tracing and simple fault detection. We have also added a fault injection layer, used for evaluating our system. We discuss fault injection as part of our experimental setup in Section 4.1.1.

J2EE supports three kinds of components: Enterprise JavaBeans, often used to implement business and application logic; Java Scripting Pages (JSP) used to dynamically build HTML page; and JSP tags, components that provide extensions to JSP. We have instrumented each of these component layers.

We assign every client HTTP requests a unique ID as it enters our system. We store this unique ID in a thread-specific local variable and also return it in an HTTP header for use by our external fault detector. With the assumption that components do not spawn any new threads and the fact that the reference implementation of J2EE we are using does not support clustering, storing the request ID in thread-specific local state was sufficient for our purposes. If a component had spawned threads, we would likely have had to modify the thread creation classes or the application component to ensure the request ID was correctly preserved. Similarly, if our J2EE implementation used clustering, we would have to modify the remote method invocation protocol and/or generated wrapper-code to automatically pass the request ID between machines.

Our modified J2EE platform's internal fault detection mechanism simply logs exceptions that pass across component boundaries. Though this is a simple error detection mechanism, it does catch many real faults that are masked and difficult to detect externally. For example, when running an e-commerce demonstration application, a faulty inventory database will generate an exception, which will be masked with the message "Item out of stock" before being shown to the user. Our internal fault detection system is able to detect this fault and report it before it is masked.

3.2. Layer 7 Packet Sniffer

To implement our external failure detector, we have built a Java-based Layer 7 network sniffer engine, called Snifflet. It is built on a network packet capture library, Jpcap [1], which provides wrappers around libpcap [17] to capture TCP packets from the network interface. We have implemented TCP and HTTP protocol checkers to monitor TCP

and HTTP failures. Snifflet uses a flexible logging package, log4j [12] from the Apache group, to log detected failures.

Snifflet detects TCP errors such as resets and timeouts, including server freezes, and detects HTTP errors such as 404 (Not found) and 500 (Internal server error). It also provides an API that enables programmers to analyze HTTP requests and responses, including content, for customized failure detection. We have implemented custom content detectors for the J2EE server that looked for simple failed responses, such as "Included servlet error".

Snifflet listens for client request IDs in the HTTP response headers of the service. Some failures, such as when a client cannot connect to a service, occur before Snifflet can find an ID for a client request. In these cases, Snifflet generates its own unique request ID for logging purposes.

3.3. Data Clustering Analysis

In our implementation of Pinpoint, we use a hierarchical clustering method, an unweighted pair-group method using arithmetic averages (UPGMA), and calculate distances between components using the Jaccard similarity coefficient. For our purposes, UPGMA's main advantage is that it calculates distances between clusters by averaging the distance among all pairs of points within the clusters. This provides a much less extreme calculation of this distance than other methods, which use a nearest-neighbor or farthest-neighbor calculation. The Jaccard similarity coefficient calculates distance between two points based on the ratio of the number of requests they appear in together out of all the requests the two points appear in total. More details on these algorithms can be found in standard data clustering textbooks, such as [23, 18].

4. Evaluation

To validate our approach, we ran an e-commerce service, the J2EE PetStore demonstration application, and systematically injected faults into the system over a series of runs. We used Pinpoint to monitor the system and diagnose the faults that we injected, and compare its results to other problem determination techniques. In this section, we detail our experimental setup, describe the metrics we used to evaluate Pinpoint's efficacy, and present the results of our trials.

4.1. Experimental Setup

We ran 133 tests that included single-component faults and faults triggered by interactions between two, three and four components. For each test, we ran the PetStore application, monitored by Pinpoint, for five minutes. During this period, we ran a client emulator that generated a workload on the application, while injecting deterministic faults into the system. We restarted the application server between each test to avoid contaminating a run with residual faulty behavior from previous runs. The setup was a closed system with a single transaction active at any time. Different transactions used different sets of components.

Our physical machine setup has a server running on one machines and clients on another. The J2EE server runs on a quad-PIII 500MHz with 1GB of RAM running Linux 2.2.12 and Blackdown JDK 1.3. For convenience, Snifflet also runs on the same machine. The clients run on a PIII 600MHz with 256MB of RAM running Linux 2.2.17 and Blackdown JDK 1.3.

4.1.1 Fault Injection

In our experiments, we model faults that are triggered by the use of individual components, or interactions among multiple components. A fault is defined by 1) the un-ordered trigger set of "faulty" components which are together responsible for the fault, and 2) the type of fault to be injected. In these experiments, we inject four different types of faults:

- Declared exceptions, such as Java RemoteExceptions or IOExceptions.
- Undeclared exceptions, such as runtime exceptions.
- Infinite loops, where the called component never returns control to the callee.
- Null calls, where the called component is never actually executed.

We chose to inject these particular faults because they cause failures that span the axes from predictable to unpredictable behaviors, simulating the range (if not the composition) of problems that can occur in a real system. In real systems, declared exceptions are often handled and masked directly by the application code. Undeclared exceptions are less often handled by the application code, and more often are caught by the underlying middleware as a "last resort." Infinite loops simply stop the client request from completing, while null calls prematurely prevent (perhaps vital) functionality from working.

It is important to note that our fault injection system is kept separate from our fault detection system. Though our internal detection system does detect thrown exceptions relatively trivially, infinite loops are only detectable through TCP timeouts seen by our external fault detector, the Snifflet. The null call fault is usually not directly detectable at all. To detect null call faults, our fault detection mechanisms must rely on catching secondary effects of a null call, such as subsequent exceptions or faults.

To inject faults into our system, we modified the J2EE middleware to check a fault specification table upon every

component invocation. If the set of components used in the request matches a fault's trigger set, we cause the specified fault to occur at the last component in the set that is used. For example, for a trigger set of size 3, a fault is injected at the third component in the trigger set used in a request.

4.1.2 Client Browser Emulator

To generate load on our system, we built a client browser emulator that captures traces of a person browsing a web site, and then replays this log multiple times during test runs. The client dynamically replaces cookies, hostname information, and username/password information in the logs being replayed to match the current context. For example, unique user ID's need to be generated when creating new accounts, and cookies provided by servers need to be maintained within sessions.

The requests include: searching, browsing for item details, creating new accounts, updating user profiles, placing orders, and checkout.

4.2. Metrics

To evaluate the effectiveness of Pinpoint, we use two metrics: *accuracy* and *precision*. A result is accurate when all components causing a fault are correctly identified. For example, if two components, A and B, are interacting to cause a failure, identifying both would be accurate. Identifying only one or neither, would not be accurate. When we measure the accuracy of a problem determination technique, we are measuring how often its results are accurate.

Precision, the second metric we use in our evaluation, is the ratio between correctly identified faults and predicted faults. For example, predicting the set $\{A, B, C, D, E\}$ when only A and B are faulty gives a precision of 40%.

Other fields—including Data Mining, Information Retrieval, and Intrusion Detection—use precision and *recall*, instead of accuracy. Recall is defined as the ratio between correctly identified faults and actual faults. For example, if 2 components, A and B, are faulty, identifying both components would give a perfect recall of 100%. Identifying only one of the two components gives a recall of 50%. However, for fault management systems, we believe accuracy is a better metric because identifying a subset of the real causes may misdirect the diagnosis and thus has little value.

A system with low accuracy is not useful because it fails to identify the real faults. A system that has high accuracy with low precision is not useful either because it floods users with too many false positives. An ideal system would predict a minimal and correct set of of faults. In practice, however, there is a tension between having high accuracy and high precision. Maximizing precision often means that potential faults are being thrown out, which decreases accuracy. Maximizing accuracy often means that non-faulty components are also included, which decreases precision.

To visualize the trade-offs an analysis technique makes between accuracy and precision, we plot its accuracy versus false positives (1 - precision) as we vary the technique's sensitivity. This plot is called a Receiver Operating Characteristic (ROC) curve. Generally, at a very high sensitivity, an analysis technique will be very accurate, but also return very many false positives. As we decrease the sensitivity, we reduce the accuracy, but also reduce the number of false positives returned. The ROC curve is especially useful because it allows us to examine analysis' behavior without arbitrarily choosing a sensitivity value. In our experiments we evaluate analysis techniques by comparing their ROC curves.

4.3. Evaluation Results

We compare Pinpoint's clustering analysis to two traditional failure analysis techniques. The first is *detection*, which returns the set of components recorded by our internal fault detection framework. This is similar to the result a monitoring system would generate, returning the component where a failure is manifesting. At its lowest sensitivity, this technique returns the single component where the fault was detected. At higher sensitivities, detection inspects the call stack, and returns the components in the call chain.

The second analysis technique we compare ourselves to is *dependency checking*, which returns the components that the failed requests use. In this technique, a component is nominated as a potential fault if it occurs in more than some percentage of the failed requests. This percentage is the inverse of the sensitivity setting. For example, by setting the sensitivity to 0%, this technique returns only components that occurred in 100% of the failed requests. Setting the sensitivity close to 100% returns all the components used in any of the failed requests. It is worth noting that our implementation of dependency checking takes advantage of Pinpoint's dynamic request tracing for dependency discovery. Hence, the quality of its results are an over-estimation of how well dependency checking would perform using a static model. The key difference between this technique and Pinpoint's cluster analysis is that dependency checking does not take the traces of successful requests into account.

We show in Figure 2 the summary results for all our experiments, comparing Pinpoint with the other techniques. Note that Pinpoint consistently has both a higher accuracy and a higher precision than detection and dependency checking. This improvement is most striking in the results for single-component failures, shown in Figure 3. Here, we see that Pinpoint achieves an accuracy of 80-90% with a relatively high precision of 50-60%. In comparison, dependency checking never has a precision higher than 20% for

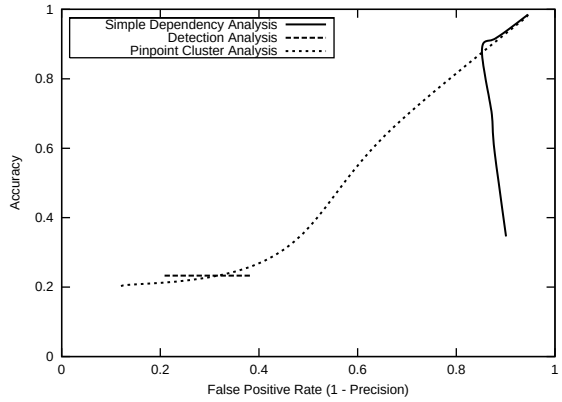

Figure 2. Summary accuracy vs. false positive rate over all tests

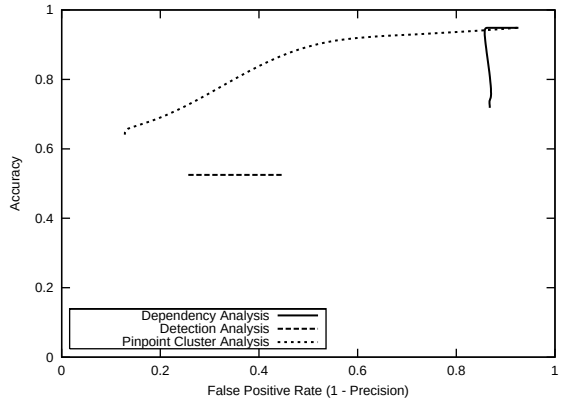

Figure 3. Accuracy vs. false positive rate for single-component faults.

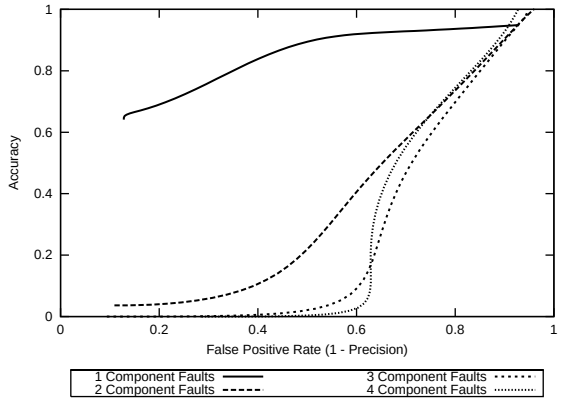

Figure 4. Pinpoint's accuracy vs. false positive rate for interacting component faults.

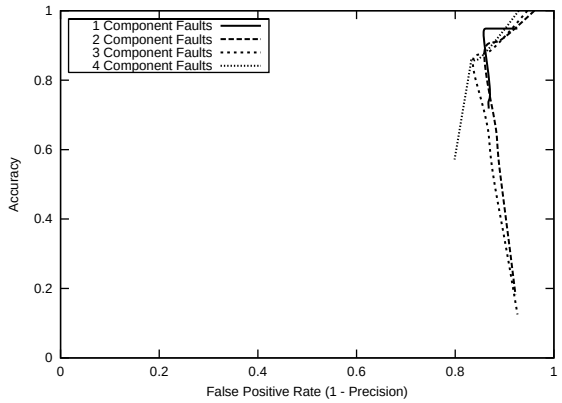

Figure 5. Dependency's accuracy vs. false positive rate for interacting component faults.

similar accuracies.

To better understand how the 3 techniques perform under latent faults—faults that occur but are not manifested as failures until a later component is used—we show their ROC curves as the "fault length" of the latent faults increases, where fault length is the number of components interacting to cause a failure. Figure 4 shows that Pinpoint has a very high accuracy and precision for single-component faults. As we generate latent faults, however, the Pinpoint's ROC curve worsens, though it still remains a significant improvement as compared to Detection and Dependency analysis.

In Figure 5 we see that the results of Dependency analysis do not appear to be affected by the fault length. Although it consistently has a high accuracy (up to 100%), Dependency always has a very low precision of about 15%. Figure 6 shows that Detection analysis is heavily affected by the fault length. Detection always has a high precision of about 30%, but its accuracy varies from 50% at single-component faults, down to 0% accuracy for three or more component-faults.

4.4. Performance Impact

We compared the throughput of the PetStore application hosted on an unmodified J2EE server with on our version with logging turned on. We measured a cold server with a warm file cache for three 5-minute runs, and found that the online overhead of Pinpoint to be 8.4%. We did not measure the overhead of the offline analysis. The uncompressed trace files generated by Pinpoint average about 2.5k per request. Compressed, however, they average only 100 bytes per request.

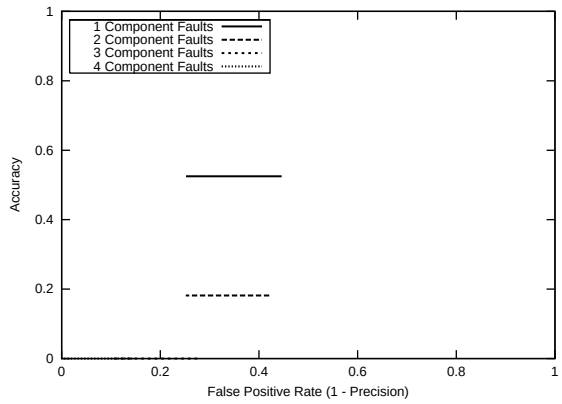

Figure 6. Detection's accuracy vs. false positive rate for interacting component faults.

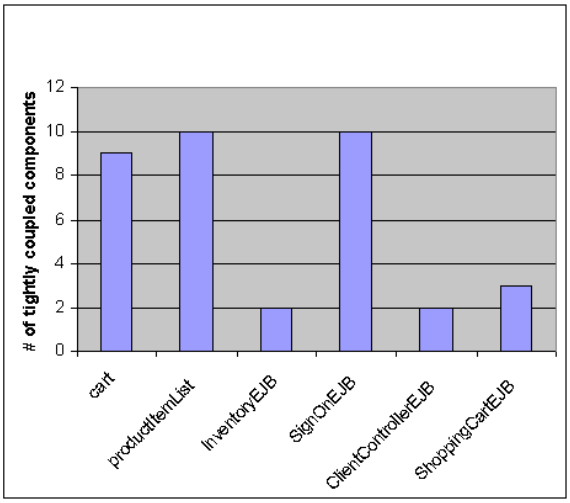

Figure 7. No. of tightly-coupled components associated with each of the components where faults were injected

5. Discussion

5.1. Pinpoint Limitations

One limitation of Pinpoint is that it cannot distinguish between sets of components that are tightly coupled and are always used together. In the PetStore application, we have found sets of components that are always used with the components that we injected faults in, shown in Figure 7. As a result, Pinpoint reports the super set of the actual faults. To better isolate faulty components and improve precision, one potential technique is to create synthetic requests that exercise the components in other combinations. This is similar to achieving good code coverage when generating test cases for debugging.

Another limitation of Pinpoint, as well as existing approaches, is that it does not work with faults that corrupt state and affect subsequent requests. The non-independence of requests makes it difficult to detect the real faults because the subsequent requests may fail while using a different set of components. For example, a user will not be able to login if the component responsible for creating new accounts has stored an incorrect password. The state corruption induced by the account creation request is subsequently discovered by the login request. One potential solution is to extend the current tracing of functional components to trace shared state. For example, Pinpoint could trace the database tables used by components to find out which sets of components share state. Implementing this extension is part of our current plans for extending Pinpoint.

Since Pinpoint monitors at the middleware and has no application knowledge about the requests, deterministic failures due to pathological inputs can not be distinguished from other failures. For example, a user may have bad cookies that consistently cause failures. One possible solution is to extend Pinpoint to record the requests themselves and use them as another possible factor in differentiating failed requests from successful ones.

Pinpoint also does not capture "fail-stutter" faults where components mask faults internally and exhibit only a decrease in performance. Fail-stutter examples include transparent hot swaps and disks getting slower as they fail. Timing information would need to be used to detect fail-stutter faults and perform problem determination.

5.2. Application Observations

In the J2EE PetStore application the average number of application components used in requests of static pages is 3. Using our workload, the average for requests of dynamic pages is 14.2 with a median of 14 and maximum of 23 (shown in Figure 8). The large number of components used in requests motivate the monitoring of components at the middleware layer and the importance of using automated problem determination techniques.

5.3. Related Work

There has been extensive literature on event correlation systems [24, 4], mostly in the context of network management. There are also many commercial service management systems that aid problem determination, such as HP's OpenView [9], IBM's Tivoli [16], and Altaworks' Panorama [3]. These systems mainly use two approaches. The first approach uses expert systems with rules (or filters) input by humans or obtained through machine learning

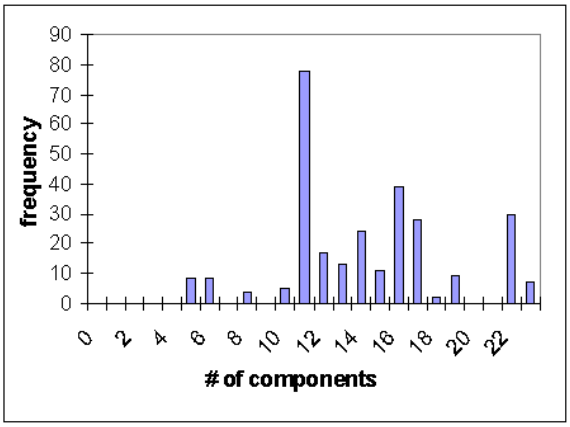

Figure 8. Histogram of No. of components used per dynamically generated page request

techniques. The second approach uses dependency models [25, 6, 13]. However, these systems do not consider how the required dependency models are obtained.

More recent research has focused on automatically generating dependency models. Brown et al. [5] use active perturbation of the system to identify dependencies and use statistical modeling of the system to compute dependency strengths. The dependency strengths can be used to order the potential root causes, but they do not uniquely identify the root cause of the problem, whereas our approach uniquely identifies the root case, and is limited only by the coverage of the workload. The intrusive nature of their active approach also limits its applicability in production systems. In addition, their approach requires components and inputs to be identified before the dependencies can be generated, which is not required in our approach.

Katchabaw et al. [19] introduce a set of libraries that programmers can use to instrument components to report their health to a central management system. The approach requires management code to be written for each component, and requires the code to be correct and to function when the component itself is failing. We take a black-box approach where we instrument application servers to trace requests without knowing the implementation details of the components. Our black-box approach enables independent auditing of the components without the overhead of writing additional code for each component.

5.4. Future Work

We plan on investigating additional factors and tradeoffs that affect accuracy and precision of problem determination. In particular, we are exploring ways of loosening our assumption of request independence by tracking state sharing across requests, as well as using timing and performance logging to diagnose performance degradations in Internet services. We are also investigating using other statistical techniques in our analysis. For example, our initial experiences using dependency analysis to discover multiple independent faults are promising.

There are also scaling issues that we need to address before we deploy Pinpoint in a real, large-scale Internet service. The current tracing mechanism needs to be extended to trace across machine boundaries. In addition, techniques such as request sampling can be used to reduce logging overhead. We also plan to automate our statistical analysis process and integrate it with an alert system to provide on-line analysis of live systems. In addition, we plan to integrate Pinpoint with other recovery-oriented computing techniques [10] to further reduce mean time to recovery (MTTR).

6. Conclusions

This paper presents a new problem determination framework for large, dynamic systems that provides high accuracy in identifying faults and produces relatively few false positives. This framework, Pinpoint, requires no application-level knowledge of the systems being monitored or any knowledge of the requests. This makes Pinpoint suitable for use in large and dynamic systems where this application-level knowledge is difficult to accurately assemble and keep current. As such, it is an important improvement over existing fault management approaches that require extensive knowledge about the systems being monitored.

Pinpoint traces requests as they travel through a system, detects component failures internally and end-to-end failures externally, and performs data clustering analysis over a large number of requests to determine the combinations of components that are likely to be the cause of failures. The runtime tracing and analysis is necessary for systems that are large and dynamic, such as today's Internet systems.

7. Acknowledgements

We are very grateful to Aaron Brown, George Candea, Kim Keeton, Dave Patterson, and the anonymous reviewers for their very helpful suggestions.

References

[1] Network Packet Capture Facility for Java. `http://jpcap.sourceforge.net/`.

[2] TPC-W Benchmark Specification, http://www.tpc.org/wspec.html.

[3] Altaworks. Panorama. http://www.altaworks.com/product/panorama.htm.

[4] A. Bouloutas, S. Calo, and A. Finkel. Alarm Correlation and Fault Identification in Communication Networks. *IEEE Transactions on Communication*, 42(2/3/4), 1994.

[5] A. Brown and D. Patterson. 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, May 2001.

[6] J. Choi, M. Choi, and S. Lee. An Alarm Correlation and Fault Identification Scheme Based on OSI Managed Object Classes. In *IEEE International Conference on Communications*, Vancouver, BC, Canada, 1999.

[7] G. Corporation. Google. http://www.google.com/.

[8] H. Corporation. HotMail. http://www.hotmail.com/.

[9] H. P. Corporation. HP Openview. http://www.hp.com/openview/index.html.

[10] David Patterson et al. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report CSD-02-1175, UC Berkeley Computer Science, 2002.

[11] J. Gray. Dependability in the Internet Era. http://research.microsoft.com/~gray/talks/InternetAvailability.ppt.

[12] A. Group. Log4j Project, 2001. http://jakarta.apache.org/log4j.

[13] B. Gruschke. A New Approach for Event Correlation based on Dependency Graphs. In *5th Workshop of the OpenView University Association: OVUA'98*, Rennes, France, April 1998.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2002. Chapter 8.12.

[15] HP. e-Speak, 2001. http://www.e-speak.hp.com/.

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

[17] V. Jacobson, C. Leres, and S. McCanne. tcpdump, 1989. ftp://ftp.ee.lbl.gov/.

[18] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[19] M. J. Katchabaw, S. L. Howard, H. L. Lutfiyya, A. D. Marshall, and M. A. Bauer. Making distributed applications manageable through instrumentation. *The Journal of Systems and Software*, 45(2):81–97, 1999.

[20] Microsoft. .NET, 2001. http://www.microsoft.com/net/.

[21] S. Microsystems. Java Pet Store 1.1.2 Blueprint Application, 2001. http://developer.java.sun.com/developer/sampsource/petstore/petstore1_1%_2.html.

[22] D. Oppenheimer and D. A. Patterson. Architecture operation and dependability of large-scale Internet services. In *Submission to IEEE Internet Computing*, 2002.

[23] H. C. Romesburg. *Cluster Analysis for Researchers*. Lifetime Learning Publications, 1984.

[24] I. Rouvellou and G. W. Hart. Automatic Alarm Correlation for Fault Identification. In *INFOCOM*, pages 553–561, 1995.

[25] A. Yemini and S. Kliger. High Speed and Robust Event Correlation. *IEEE Communication Magazine*, 34(5):82–90, May 1996.