Submission to International Performance and Dependability Symposium

Pinpoint: Problem Determination in Large, Dynamic Internet Services

Mike Y. Chen
U.C. Berkeley
445 Soda Hall
Computer Science Division
Berkeley, CA 94720
Ph: +1(510) 643-9435
Fax: +1(510) 642-5775
mikechen@cs.berkeley.edu

Emre Kıcıman
Stanford University
452 Gates Building, 4-A
Stanford, CA 94305
Ph: +1(650)725-0256
emrek@cs.stanford.edu

Eugene Fratkin
Stanford University
508 Gates Building
Stanford, CA 94305
Ph: +1(650)497-5374
fratkin@cs.stanford.edu

Armando Fox
Stanford University
446 Gates Building
Stanford, CA 94305
fox@cs.stanford.edu

Eric Brewer
U.C. Berkeley
445 Soda Hall
Computer Science Division
Berkeley, CA 94720
brewer@cs.berkeley.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 component(s) 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 much more difficult.

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 [19] and Hewlett-Packard's E-Speak [13], 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. As clients connect to these services, their requests are dynamically routed through this system. Current Internet services, such as Hotmail [9] and Google [7], are already hosted on thousands of servers [1] and continue to grow. The large size of these systems results in more places for faults to occur. The increase in dynamic behavior means there are more paths a request may take through the system to be serviced, and thus results in more potential failures due to "interaction" faults among components. The difficulty of managing these large and dynamic systems is evident in that many outages still occur and 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 [22, 4]. Recent event correlation techniques based on dependency models [5, 23, 6, 12] use statically generated dependencies of components to determine which components are responsible for the symptoms of a given problem. Two limitations of using dependency models are that they are difficult to generate accurately and they are difficult to keep consistent with an evolving system. Another limitation of static dependency models is that they reflect the dependencies of logical components, and are not expressive enough to capture the effects of replicated components. For example, two identical requests may use different instances of the same (replicated) components. As a result, dependency models can be used to identify which component is at fault, but

[1] Hotmail 7000+ [10], Google 8000+ [21]

not which instance of the component.

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. Additionally, this analysis tends to be robust in the face of multiple independent faults and spurious failures.

This data clustering 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. Both of 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 allow 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 J2EE demonstration application, Pet Store [20]. We use a workload that mimics the request distribution of TPC-WIPSo (ordering) web commerce benchmark [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 in that we were able to automatically identify the root causes 76% of the time with an average rate of 66% 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

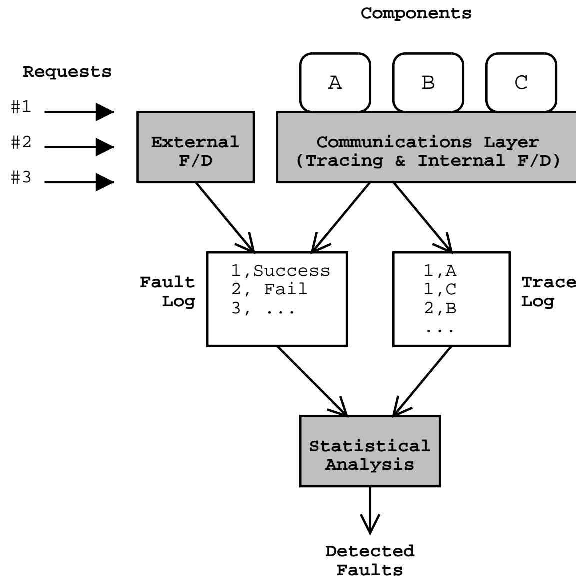

Figure 1: The Pinpoint Framework.

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.

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, previous work in this area and future work. We conclude in section 6.

2 Pinpoint Framework

To validate our approach to 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.

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 and 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 undue 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_id 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, 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 file server failing, and then being replaced by a hot swap. Though these failures do not become visible to user, system administrators should still be interested in tracking these errors if they happen too frequently. 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.

2.3 Data Analysis

Once the client request traces and the failure/success logs are available, they are given to Pinpoint's analysis subsystem. Here, the data analysis uses clustering and dependency analysis to discover sets of components that are highly correlated with failures of requests.

Diagnosing root-causes is a two step process: component dependency discovery, followed by clustering algorithms such as basket analysis. First, data is run through a dependency discovery algorithm to try to find out which components failures are most dependent on. Dependency discovery considers both occurrences of a component in failed requests as well as its presence in successful ones. This technique is necessary to discover independent faults that would be missed by running only clustering algorithms, such as basket analysis. The reason for that is that some of the basket analysis algorithms place items in only one cluster, hence a fault could be correlated with at most one of its independent causes. For instance, if A causes Fault, and B causes Fault, output of basket analysis will produce only (B causes Fault) or (A causes Fault) but not both.

When there are no faults caused by independent sets of components, running the two-step analysis is equivalent to running only the clustering step. In either case, when failures are caused by several interacting components, clustering typically generates a superset of the real faults, whereas dependency discovery is likely to produce a subset. There reason for that is that if components A and B only appear together and combination of A and C causes fault then cluster would consist of (A,C,B,Fault) or a superset. In the case of dependency discovery if A and B are causing fault and almost always appear together then dependency analysis will first include only one of the components. Having added this component, contribution of the other one would be not significant enough to be correlated with failure.

3 Pinpoint Implementation

We have implemented a prototype of Pinpoint on top of Java2 Enterprise Edition (J2EE) middleware, a network sniffer, and a commercial off-the-shelf data analysis program, PolyAnalyst [18]. Our prototype does not require any modifications to be made to J2EE applications. The only application knowledge our prototype requires are application-specific checks to enable external fault detection and, even in this case, no application component needs to be modified. Because of this, Pinpoint can be used as a problem deter-

mination 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 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 passing 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 app, 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 [15] 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 [11] 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".

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

Snifflet listens for client request ids in the HTTP response headers of the service. In the case of some failures, such as if a client cannot connect to a service, failures 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 take advantage of a commercially available data analysis tool, Polyanalyst[18], to analyze our data. As was described earlier, we use two data analysis techniques to correlate between sets of components and failures of requests. Here, we give a summary description of how we apply these techniques to problem determination in Pinpoint. We first use a dependency discovery algorithm, called ARNAVAC [17], to identify potential independent faults. We then apply a clustering algorithm from the data mining field called basket analysis, commonly used to analyze shopping behavior.

When analyzing consumer behavior, basket analysis determines which sets of products are likely to be bought together. Data is represented as a matrix with each row being a transaction and each column a product. For each transaction and product, a 1 or 0 is placed in the appropriate matrix cell if the item was purchased as part of that transaction.

In Pinpoint, our input to the basket analysis algorithm is a matrix where a row is a client requests, and columns are components. An additional column, ”failure”, represents whether the request failed or was successful. Table 1 displays hypothetical data in this format.

Basket analysis outputs a set of components that are likely to co-occur with failures. The analysis also outputs confidence, which is the probability of error happening given that all the components correlated to failures appeared. This is an important value since almost any component can be used in the failed request, hence almost any component can be correlated with some confidence. By adjusting this threshold, one can attempt to distinguish between the real causes and accidental correlations. The confidence we use is 65%, the default setting of PolyAnalyst.

4 Evaluation

To validate our approach, we ran an e-commerce service, the J2EE Pet Store demonstration application, and systematically injected faults into the system over a series of runs. 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 55 tests that included single-component faults and faults triggered by interactions between two, three and four components. For each test, we ran Pinpoint for five minutes, and, during this period, injected faults and made as many requests as possible to exercise our installed Pet Store application. We restarted the Pinpoint system 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 definition consists of an un-ordered trigger set of "faulty" components which, together, are responsible for the fault, and 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 these problems span the axes from predictable to unpredictable behaviors 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 match 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, 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, 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, which is closely related to recall, and precision. Recall and precision are the same metrics used in other fields, including Data Mining, Information Retrieval, and Intrusion Detection. Recall is defined to be 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, say A, gives a recall of 50%.

However, for fault management systems, we introduce a metric, accuracy, to represent the result of testing whether (recall == 100%). Thus, a result is accurate if and only if all real faults were identified. We believe accuracy is a better metric for fault management systems because identifying a subset of the real faults may misdirect the diagnosis and thus has little value. Accuracy is a boolean metric, so any given result is either accurate or not accurate.

The second metric, precision, is the ratio between correctly identified faults and predicted faults. For example, if 2 components, A and B, are faulty, predicting a set of A, B, C, D, E has a precision of 40%.

A system with low recall is not useful because it fails to identify the real faults. A system that has high recall with low precision is not useful either because it floods users with too many false positives. An ideal system should predict a minimal set of faults that contains the actual faults. However, in practice, there is 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 included, which decreases precision.

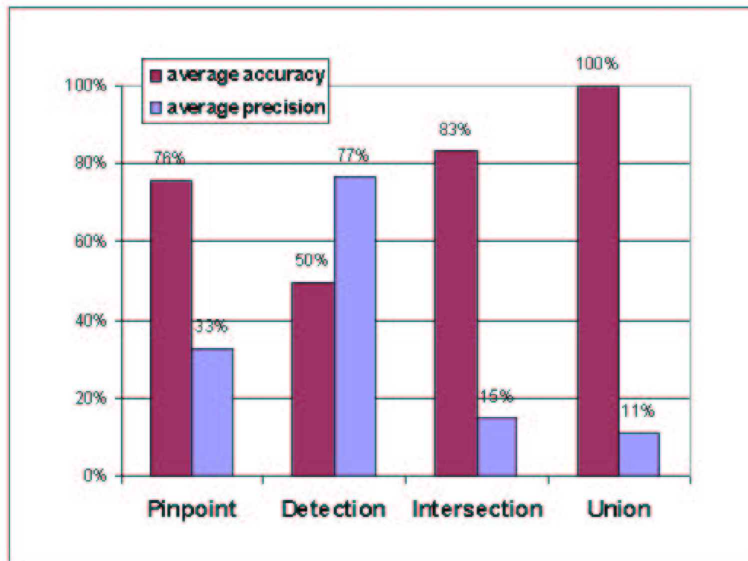

Figure 2: Average accuracy and precision of Pinpint, and three other failure analysis techniques

4.3 Evaluation Results

We compare 3 types of failure analysis techniques to Pinpoint's clustering analysis. The first is Detection, which returns the set of components recorded by our internal fault detection framework. This is similar to what an event correlation system would generate. Adding more filtering rules could potentially improve precision, but has no positive effect on accuracy. The second is Intersection, the set of components that are used in all failed requests. This approximates what a system using dependency model would generate. Third is Union, the set of components that are used in any of the failed requests. We have found that Pinpoint provides high accuracy while providing reasonable precision.

Figure 2 shows the average accuracy and average precision across all 55 runs for the 4 techniques. We have found that Union and Intersection generally predicts supersets of the actual faults. Therefore, they have high accuracy at the cost of low precision, meaning that they flood users with many false positives.

The set recorded by Detection differs from the actual faults in two ways. First, latent faults, faults that occur but are not detected until a later component is used in processing the same request, may cause it to report the wrong components and have low accuracy. Second, faults may cascade and cause other components to fail at the same time, resulting in a superset of the actual faults being recorded. This effect is often called an event storm, although we have not observed any significant occurrences in our system.

Pinpoint looks at the components used in request, and has high accuracy because it is better at isolating the set of components that cause latent faults. The tradeoff is that Pinpoint's prediction is limited by the coverage of the different combination of components that are used. For example, if A, B, and C are always used together, then Pinpoint will report all of them even when only one of them is faulty.

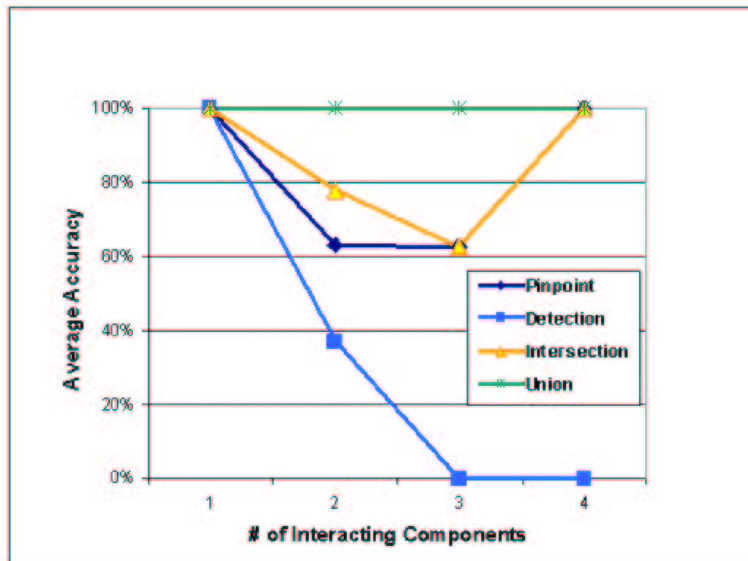

Figure 3: Accuracy vs. fault-length (number of interacting components causing fault)

To better understand how the 4 techniques perform under latent faults, we plot the accuracy of them as the "fault length" of the latent faults increases, where fault length is the number of components interacting to cause a fault. Figure 3 shows that all 4 techniques have 100% accuracy when faults are detected at the component they occur. As the fault length increases, the accuracy of Detection drops to 0% while both Pinpoint and Intersection remain above 62% for lengths of 2 and 3 and both jump to 100% for lengths of 4.

4.4 Performance Impact

We compared the throughput of the Pet Store application hosted on an unmodified J2EE server and on our version with logging turned on. We measured a cold server with a warm file cache for 3 5-min runs, and found that the online overhead of Pinpoint to be 8.4%. We did not measure the overhead of the offline analysis.

5 Discussion

5.1 Pinpoint Limitations

One limitation of Pinpoint is that it can not distinguish between sets of components that are tightly coupled and are always used together. In the Pet Store application, we have found sets of components that are always used with the components that we injected faults in, shown in Figure 4. As a result, Pinpoint reports the super set of the actual faults. In order to isolate faults to improve precision, one potential technique is to create synthetic requests that exercise the components in other combinations. This is similar to achieving

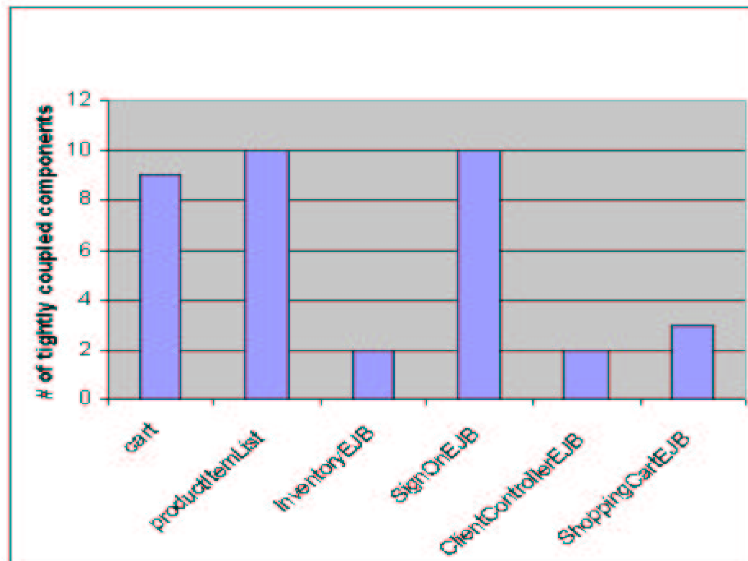

Figure 4: No. of tightly-coupled components associated with each of the components where faults were injected

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 won't be able to login if the component responsible for creating new accounts have stored an incorrect password. The state corruption induced by the create_account 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.

Since Pinpoint monitors at the middleware and has no application knowledge and the requests, persistent (deterministic) failures due to pathological input can not be distinguished from other failures. For example, users 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.

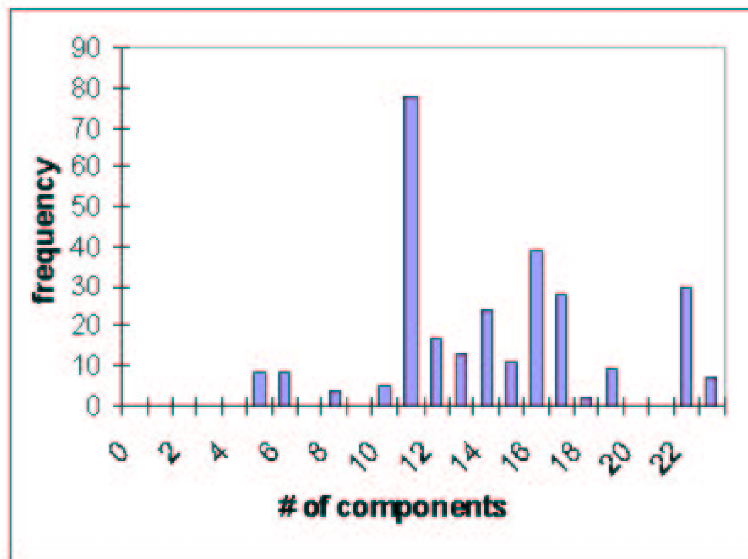

Figure 5: Histogram of No. of components used per dynamic page request

5.2 Application Observations

In the J2EE Pet Store application, the average number of application components used in dynamic page requests is 14.2 with a median of 14 and maximum of 23 (shown in Figure 5). The large number of components used in requests motivates 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 [22, 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 [8], IBM's Tivoli [14], and Altaworks' Panorama [3]. These system mainly use two approaches. The first approach is expert systems that use rules (or filters) input by humans or obtained through machine learning techniques. The second approach uses dependency models, such as Yemini et al. [23], Choi et al. [6], and Gruschke[12]. 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. [16] 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. We are exploring ways of using other factors to differentiate between failing and successful requests, such as timing and the ordering of components used within a request.

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.

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 nor 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. This 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 basket analysis and dependency discovery 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, and Dave Patterson for their helpful suggestions and stimulating discussions.

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] Aaron Brown and David 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] Google Corporation. Google. `http://www.google.com/`.

[8] Hewlett Packard Corporation. Hp openview. `http://www.hp.com/openview/index.html`.

[9] HotMail Corporation. Hotmail. `http://www.hotmail.com/`.

[10] Jim Gray. Dependability in the internet era. `http://research.microsoft.com/~gray/talks/InternetAvailability.ppt`.

[11] Apache Group. Log4j project, 2001. `http://jakarta.apache.org/log4j`.

[12] 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.

[13] HP. e-speak, 2001. `http://www.e-speak.hp.com/`.

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

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

[16] Michael J. Katchabaw, Stephen L. Howard, Hanan L. Lutfiyya, Andrew D. Marshall, and Michael A. Bauer. Making distributed applications manageable through instrumentation. *The Journal of Systems and Software*, 45(2):81–97, 1999.

[17] Mikhail V. Kiselev. Polyanalyst 2.0: Combination of statistical data preprocessing and symbolic kdd technique. In *ECML-95 Workshop on Statistics, Machine Learning, and Knowledge Discovery in Databases*, Heraklion, Greece, 1995.

[18] Megaputer. Polyanalyst 4.4, 2001. `http://www.megaputer.com/`.

[19] Microsoft. .net, 2001. `http://www.microsoft.com/net/`.

[20] Sun Microsystems. Java pet store 1.1.2 blueprint application, 2001. `http://developer.java.sun.com/developer/sampsource/petstore/petstore1_1%_2.html`.

[21] Dave Patterson. Networks 3: Clusters, examples. `http://www.cs.berkeley.edu/~pattrsn/252S01/Lec10-network3.ppt`.

[22] Isabelle Rouvellou and George W. Hart. Automatic alarm correlation for fault identification. In *INFOCOM*, pages 553–561, 1995.

[23] A. Yemini and S. Kliger. High speed and robust event correlation. *IEEE Communication Magazine*, 34(5):82–90, May 1996.