University of Alberta

Problem Diagnosis in Enterprise Web Systems using Scenario Trace Analysis

by

Ananth Kumar Venkateswaran

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2008

Library and Archives Canada
Published Heritage Branch
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada
Direction du Patrimoine de l'édition
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-47431-0
Our file Notre référence
ISBN: 978-0-494-47431-0

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

To my beloved Mother and Father, who made me what I am.
You are my everything.

Abstract

Enterprise web systems have come of age over the last decade with most of the day to day activities automated to a possible extent. Current state web systems being distributed and running on multiple environments constantly encounter problems that are highly transient in nature and having very less information to help resolving the issue. Eventually, with a plethora of possible causes to consider, large amounts of time are spent by maintenance teams in checking out every single possibility.

In this thesis we have designed and implemented a prototypical diagnostic framework that aims to solve the two fold problem of effectively zeroing in on the section of the system producing the errors and minimizing the time taken to resolve the diagnosed fault.

The framework concentrates on traces corresponding to user requests and feeds related information into a decision table structure that identifies the problem causes. We also include evaluation results that illustrate the effectiveness of the framework.

Acknowledgements

I would like to sincerely thank my supervisor, Dr. Kenny Wong for his sincere support and guidance. He was highly inspiring and motivated me with constructive feedback.

I would like to thank the members of my examining committee for their insightful thoughts and feedback.

I would also like to thank Mr. Serge Mankovski, CA Labs for the effort he took in reaching out to the people in the industry with this work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In an increasingly internet dominated era, web-based systems that evolved over a period of time have become highly powerful and complex in nature. Such systems, built from enterprise components, range from conventional e-commerce systems deployed by Amazon, e-bay, and so on, to complex online banking systems comprising of highly secure application components and rich domain-specific business logic. Development platforms have so far been successful in keeping up to pace with the increasing business complexity of organizations, offering cutting-edge technology upon which such component-based systems are built. With this increasing dependance on web-based systems to run their businesses, it is of paramount importance to corporations to quickly eliminate problems arising in these systems and maintain high performance levels.

1.1 Motivation

Electronic commerce is fast developing into a major income provider to many organizations and is only expected to grow with high volumes of transactions being done online throughout 2010 [6]. Unfortunately though, the increasing complexity of web-based systems has made it even more difficult for people involved in the maintenance sector to keep them running and meet user expectations of a highly reliable and efficient online experience. The exploding growth of web-based systems has actually opened up a whole new set of problem scenarios typically not encountered in simple client-server environments or other non-distributed systems. These problems are spread across the entire environment on which the production system is deployed, and often remain unnoticed until the affected part of the system fails.

Finding root causes for infrastructure-related problems and performance issues is more difficult in distributed systems as compared to non-distributed ones. These systems give rise to a variety of dynamic errors such as deadlock contention among multi-threaded operations, unexpected usage of resources by parallel processes, unexpected change in system settings and configuration issues, memory leak problems and so on. A problem caused by a diminishing system resource in one host of a large, distributed system may go unnoticed until it becomes critical and violates the service level agreement. The high volume of traffic flowing through these systems may lead to a ripple effect of failures causing a degradation of overall system performance. Such problems, if noticed earlier, could prevent systems from degrading and help minimize the downtime of that system.

Another important fact that has to be considered is the IT budget allocated every fiscal year by corporations to develop new systems as per changing business needs, and to maintain their current production systems within the service level agreements. Unfortunately, most of the IT budget is allotted to the latter rather than the former. A recent study [7] of software maintenance costs in typical, large application systems states that the support cycle of such systems typically spans between 67% to 80% of the overall life-cycle cost. An important issue the report stresses is that only half of the overall maintenance budget is spent on enhancing the application's capabilities to cater to the increasing needs of its users. The remaining half consists of refinements to the application to adapt it to the changing environment on which it is deployed and code maintenance to keep the system running smoothly. It thus becomes important to reduce the maintenance costs of these systems and better utilize the budget for enhancements and system improvements.

It is hence necessary to simplify and automate as much as possible the task of proactive problem detection and accurate root cause analysis of problems in enterprise systems. Current diagnostic systems built for distributed enterprise-level infrastructure deal mostly with performance-related problems in the individual components present within the application. There are few systems that provide enough clues to identify the root causes of such problems or try to correlate symptom data across domains to effectively identify dependency issues. In other words, current diagnostic systems are highly capable of identifying failing or erratically behaving components but fail to recognize the true cause of those failures, which is more important from an analyst's point of view to help debug problems quickly.

One good place to look at potential causes for system problems is the scenario trace data, which contains comprehensive information about the system and individual components in the context of user activities happening through the system. The definition of scenario here is the set of actions or events that take place in a user session spanning one or more use cases defined to perform an activity. Usually problems such as excessive memory consumption, high CPU consumption, thread contention, etc., result in the user scenario terminating prematurely (as a result of a particular component failing to execute) or deviating erratically from the normal execution path to perform unwanted or incomplete actions. While diagnostic systems can identify the dying components in these scenarios, they rarely succeed in diagnosing the exact cause for the failure. This phenomenon is evident in errors that have common components involved in multiple scenarios and the effect of one failing scenario impacts other scenarios in ways potentially unnoticed until enough damage has been done.

1.2 Objectives

The focus of this thesis is to identify subtle errors happening in scenarios that terminate prematurely or deviate from the normal path of execution. Such errors, though difficult to track in a system, often are indicators of massive failures to occur, and it is important for diagnostic systems to understand these errors and lead the system maintenance team to look into the right areas of concern. On the other hand, each identified fault usually comes along with a large set of potential causes. Eliminating the irrelevant causes and quickly concentrating on the most important ones is critical for a successful diagnosis.

The thesis outlines in detail a diagnostic framework built on the principles of hypothesis testing and the process of elimination. The underlying idea of the framework is to identify abnormally behaving scenarios in distributed component-based systems caused by failures and point to the possible causes with enough reasoning to reduce the mean-time-to-resolution of the failures detected.

Our approach aims at not only identifying errors present in scenario traces of the system but also to correlate the scenario data over a significant period of time to help identify probable root causes of such errors. The main idea is to subject the trace information corresponding to a problem condition through a number of comprehensive checks, pertaining to possible causes for that particular symptom, and detail the reasons of failure and areas of concern. Eventually, the diagnostic framework outlined in the thesis helps to minimize the amount of detail the system analyst has to look at to debug a potential problem and directly points him to those areas that need immediate attention. As such, the maintenance team will be able to reduce downtime and avoid violating the agreed service levels. We have tried to support our claims of the framework by evaluating it on a typical web-based system subjected to a set of dynamically injected problem conditions. These randomly generated errors lead to the components in the system to behave erratically (either failing to execute or perform unwanted operations). We then use our framework to identify these erratically behaving components in the context of scenario traces and locate the possible causes. The results outline that the framework is highly promising to reveal failures in web-based systems and effectively points out the problem causes. We also compare our framework against other diagnostic techniques and outline the advantages our technique has over the others.

The remainder of the thesis is organized as follows. Chapter 2 describes the background and the state of the art in diagnosis as it relates to web-based systems, outlining the need for scenario trace analysis. Chapter 3 presents the research hypothesis and defines the overall framework for the cause determination of a typical set of problems manifested in web-based systems. Chapter 4 describes the results of an evaluation that determines the capability of the diagnostic framework to effectively identify failure causes by utilizing scenario trace data. A concluding discussion of the hypothesis, results, contributions, and future work is provided in Chapter 5.

1.3 Contributions

The contributions in this thesis focus on identifying performance-related problems in web-based enterprise applications and help to automate the process of locating the most probable cause(s). We outline the contributions below:

- We undertook a comprehensive study on symptoms (or effects) and related causes and successfully came up with a summary of effect-cause relationships with respect to performance-based problems. Each effect with related causes are then modeled in a decision table structure that forms an integral part of the analytic portion of the framework.

- We developed a unique integer-sequence representation of scenario traces and devised a custom data structure to effectively organize these traces and related information. This was done to improve the efficiency of retrieving data present in the log files.

- We devised a decision-table-based rule execution technique that performs a step-by-step check across the recorded log information from scenario traces. The effect-cause decision table is used to identify probable causes corresponding to the effects.

- We performed an evaluation to validate the effectiveness of our framework on a case study system, which had to be customized and configured to mimic a distributed setup.

- We programmatically generated live web traffic to flow through the case study system by means of batch scripts developed using an open-source load-generator mechanism.

- We developed a dashboard style visualization tool that hooks into the diagnostic framework to display the related problem diagnosis information in an easy-to-understand manner. The visualization option also contains functionality for the user to perform advanced analysis on the processed log files across different time periods thus aiding in identifying failure patterns over different time windows

- Finally, we also critically analyzed the effectiveness and qualitative advantages of our framework by comparing with an open source diagnostic tool.

Chapter 2

Background

"When there is a perceived problem with an application or supported service, a diagnostician must have the tools and information at his/her disposal to pinpoint the problem with reasonable certainty, in hopes of avoiding the problem in future" [8]. This chapter starts by introducing the concept of diagnosis in general terms and later delves deeply into the common diagnostic practices carried out in the context of web-based systems. The first half of the chapter discusses the functioning of state-of-the-art diagnostic mechanisms typically used in large-scale web systems. The second half of the chapter then briefly discusses the different types of problems occurring in web-based systems. At the end, the chapter considers a comprehensive set of diagnostic mechanisms based on trace analysis, which laid the foundation for our diagnostic framework. Finally, we end with a concise summary of the entire chapter.

2.1 Definition of Diagnostics

The word "diagnosis" originates in ancient Greek culture and is described by the words "dia" which means "by", and "gnosis" which means "knowledge". The person normally conducting the process of diagnosis is known as a "diagnostician".

Diagnosis usually signifies a broad category of analysis-based testing. Typically, it would be one of these tests or a combination that would help a diagnostician in deciding the cause of a disease or defect.

The same meaning resonates across the software engineering domain as well. The process of identifying a problem condition based on symptoms related to the respective system's behavior is termed as software system diagnosis. Initially, software system diagnosis was more related to troubleshooting defects in hardware. The most prominent troubleshooting procedures were modeled as flowcharts with the technician having to traverse a set of options that eventually led to the diagnosis of the problem on hand. The following sections elaborate on the evolution of diagnosis in software engineering with a special focus on current diagnostic mechanisms as applied to web-based systems.

2.2 Diagnostics in practice

Problem diagnosis is a well defined and commonly used term in the medical community. But of late, the application software industry has been giving an unprecedented response to the process of collecting system data, analyzing problem symptoms, and devising corrective measures to prevent the problem from recurring or, in other words, the process of diagnosing failures.

Although support for diagnosis were not meant to be a design consideration of application software, organizations have slowly started realizing the importance and need for a robust technology that can alert system personnel of impending dangers to the system and help take corrective action before a major catastrophe happens. Some industry surveys [9] too confirm the need for an efficient diagnostic mechanism to support their application software. This need is critical for several reasons.

- Among companies with $1B or more in revenues, nearly 85% experienced incidents of performance degradation.

- 40% of the unplanned downtime was due to application failures.
- The cost of down time of mission-critical applications averaged over $100,000 per hour.
- IT groups spent 24% of their time in resolving application slow-downs.
- 80% of unplanned downtime could have been mitigated by application development and operations working together.

Clearly, IT personnel spend much of their time and valuable resources in reacting to application software problems. But what remains to be seen is the efficiency of the installed diagnostic mechanism to effectively help them understand system behavior and provide timely resolution of encountered problems.

2.2.1 Enterprise Web Components - A Primer

Since the framework presented in the thesis focuses on component execution flow in scenario traces as they happen in a web-based system, it is important to understand the basics of the enterprise-level web components and their features. The following section outlines the individual components in an enterprise system and their role in executing the user scenarios.

The Java 2 Enterprise Edition (J2EE) [10] defines a standard for developing multi-tier enterprise applications. It provides an architectural framework in three tiers upon which developers can build their enterprise systems. Figure 2.1 shows the three tiers of the J2EE framework. Most scenario trace logging-mechanisms concentrate on recording data about a specific set of components (discussed in the following subsections). Most of the problems discussed in the thesis actually re-

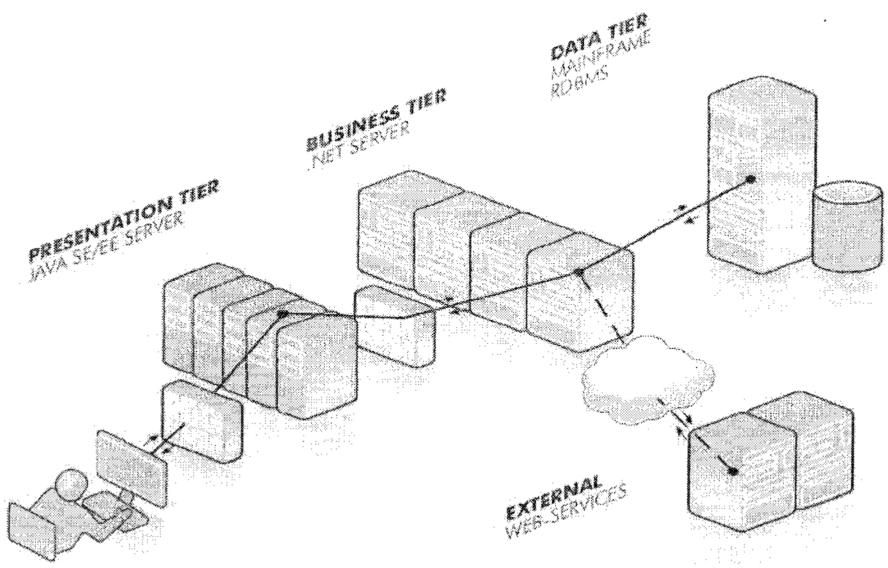

Figure 2.1: The execution path of a typical scenario in a multi-server web environment

volve around these components and it would suffice to record data around them as execution passes through the components in the scenarios.

Web Tier

The J2EE web tier provides a runtime environment (or container) for web components. J2EE web components are either servlets or pages created using the Java Servlet Pages (JSPs) technology [11]. Java Server Pages are a combination of HTML constructs and servlet tags that render dynamic content executed by the servlet container in the middleware and display it to the user based on the HTML constructs defined in the JSP page. Servlets on the other hand are the base entities of the enterprise framework that dynamically process requests and the corresponding responses. This combination of JSP and servlets are then seamlessly accessed by the user over normal HTTP just like any other webpage.

Business Tier

Enterprise Java Beans (EJBs) [12] are the business tier components and are used to handle business logic. The domain knowledge in areas such as banking, finance, retail and so on are embedded within these business tier components. EJBs are usually executed in a separate EJB container distinct from the servlet containers and often interact with the underlying databases in the Enterprise Information System (EIS) tier in order to process requests. EJBs can be easily accessed either through the web tier components or through stand-alone Java-based applications. Modern day frameworks have enhanced the functionality of EJBs by adding attributes such as messaging, security, transactionality and persistence.

The above-mentioned set of functionality is readily made available to applications when requested via an XML deployment descriptor. This XML deployment descriptor contains metadata that associates both structural and behavioral information to a particular component. There must be an association of every Business Tier component with a respective XML descriptor or, in other words, EJBs are monitored and controlled through the XML descriptors.

EIS Tier

EIS provides the information infrastructure critical to the business processes of an enterprise. Examples of EISs include relational databases, enterprise resource planning systems, mainframe scenario processing systems and legacy database systems. The J2EE Connector architecture [13] defines a standard architecture for connecting the J2EE platform to heterogeneous EIS systems. For example, a Java Database Connectivity (JDBC) connector is a J2EE Connector Architecture compliant connector that facilitates integration of databases with J2EE application servers. It is highly important that the developers adhere to the specifications of the JDBC

API [14]. Application developers should communicate with the vendors databases using the JDBC API. The main advantage of JDBC is that it allows for portability and avoids vendor lock-in. As all databases must adhere to the same specification, application developers can replace the one they are using with another one without having to rewrite their application.

With emerging technologies in enterprise-applications infrastructure, different frameworks such as Springs, Struts, Portals etc., have evolved over time to provide rich and meaningful context to the business domain that they are designed for. However, it is only the high-level design that has changed and not the underlying components that form the application itself. Hence, nagging component related problems still remain at large and need to be addressed.

2.2.2 Functioning fundamentals of diagnostic tools

Web-based systems always need to be monitored to record performance levels and also ensure these levels are maintained at a suitable level of functioning. To do so, different types of monitoring and diagnostic mechanisms are put in place to effectively observe the different parts of the system. Since web-based systems usually consist of distributed components, the monitoring mechanism needs to correlate performance data across the entire network and effectively process it to identify bottlenecks within the system. The most generic way of collecting information related to the health of the application is to instrument the right parts of the system housing the application and have agent programs collect the necessary data without impeding the normal performance of the system as a whole.

Most of the diagnostic tools these days rely on agent technology that relay collected observations to a centralized processing server, which then efficiently corre-

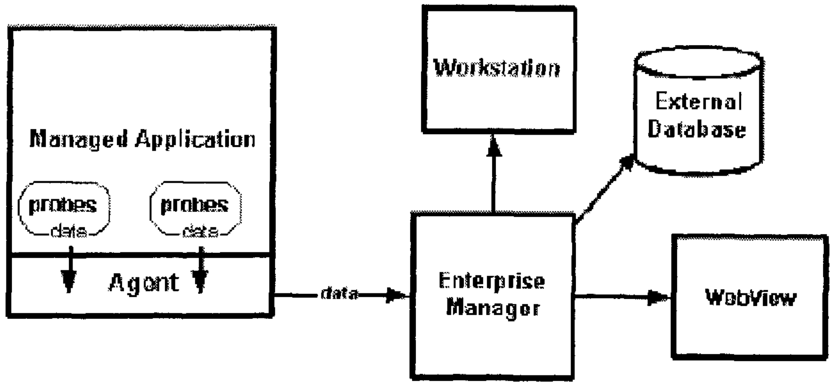

Figure 2.2: Architecture of a typical diagnostic tool [2]

lates the information and prepares it to be analyzed either online or off-line. Figure 2.2 is an architectural view of a typical diagnostic tool.

The format or type of data usually collected by the agent collectors depends on the type of instrumentation used by the diagnostic tool and often varies from one product to another. Figure 2.3 and Figure 2.4 show two different types of log formats used for recording system information by their respective tools. Though the objective is to record component related information it is quite evident from the formats that each diagnostic tool understands its own proprietary log format and processes it accordingly.

An interesting fact to be noted here is the research undertaken by global corporations to standardize the log formats instead of the proprietary formats collected by each of the vendor-specific diagnostic tools. IBM, for instance, has developed the Common Base Event (CBE) format [15] to help organizations that deploy enterprise systems to avoid vendor lock-in. Efforts are very much in the direction of forming an open standard format and a simple API to help generate CBE logs, which can then be fed into the diagnostic tools for processing. Figure 2.5 illustrates

```
VERSION;1.2
MACHINE;localclient.raleigh.ibm.com;9.27.133.227
Client_Request;2004-07-29T08:41:08.783000-04:00;
2104668451;Client;lookupCustomerRequest;wrapped;2way;
<methodEntry threadIdRef="2104668451"
    time="1091104868.783" methodIdRef="5"
    classIdRef="2" objIdRef="28"
    ticket="25" sequenceCounter="1"
    stackDepth="1"/>;2
<?xml version="1.0" encoding="UTF-8"?>
```

Figure 2.3: Log format-IBM Data Collector. This log represents the data corresponding to a single component recorded as part of a purchase order use case in a supply chain management application

a typical CBE log corresponding to a single component in a web-based system.

Most diagnostic tools work with statistics generated out of the log files that are processed. The data is aggregated in certain time intervals and corresponding minimum, maximum, and average values are calculated. Some important sets of data collected and processed are, for example, system data (e.g., CPU consumption, memory consumption, I/O rate), application server statistics (e.g., data source- and thread pool-utilization) and JVM-statistics (e.g., heap utilization, GC, thread count) which are the important statistics necessary to identify potential performance bottlenecks. To name a few, DynaTrace Diagnostics [16] and CA/Wily Introscope [2] are the most prominent ones among the state-of-the-art commercial diagnostic tools.

The diagnostic tools often come with rich visualization dashboards summarizing the collected metrics and displaying them to the user. This aids in effective understanding of system information and helps debug potential problems at a much quicker pace. Figure 2.6 shows the statistical snapshot obtained from Wily Introscope running on an example system. The signal lights in the far end corners of the page indicate the overall health of each section of the system. An application-level

```
<appender name="glassbox" class="org.apache.log4j.RollingFileAppender">
    <param name="File" value="glassbox.log"/>
    <param name="Append" value="true"/>
    <param name="MaxBackupIndex" value="5"/>
    <param name="MaxFileSize" value="10MB"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d{DATE} %-6r [%t] %-5p %c %x - %m %n"/>
    </layout>
  </appender>
```

Figure 2.4: Log format-log4j. This log information is from a servlet in an enterprise application.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<CommonBaseEvent creationTime="2008-04-28T21:03:11.78Z"
        globalInstanceId="CBED27BEBECAA891A3382AF7FD0156611DD"
        msg="Variable changed to: 1" version="1.0.1">
        <sourceComponentId component="Application"
                componentIdType="Application" location="142.104.68.52"
                locationType="Hostname"
                subComponent="int com.rigi.daytrading.actions.DoTestAction.count"
                componentType="Application"/>
        <situation categoryName="ReportSituation">
                <situationType xsi:type="ReportSituation" reportCategory="TRACE"/>
        </situation>
</CommonBaseEvent>
```

Figure 2.5: Log format-CBE. This event log corresponds to a servlet executed in a stock trading application.

trend graph shows the performance statistics of the application over time intervals. The bottom half of the page shows the resource utilization levels of the environment running the application. Figure 2.7, on the other hand, shows some of the detailed statistics collected by Wily Introscope. This page allows the user to drill down into specific components (i.e, Java Server Pages, Servlets, JDBC connections, EJBs, etc.,) and visualize their performance trends over a time period.

Although diagnostic tools originally were designed to perform statistical analysis of key performance indicators collected from the application and the environment on which the application is hosted, more functionality is being added to their repertoire. The ability to discern component call traces and collect pertinent data

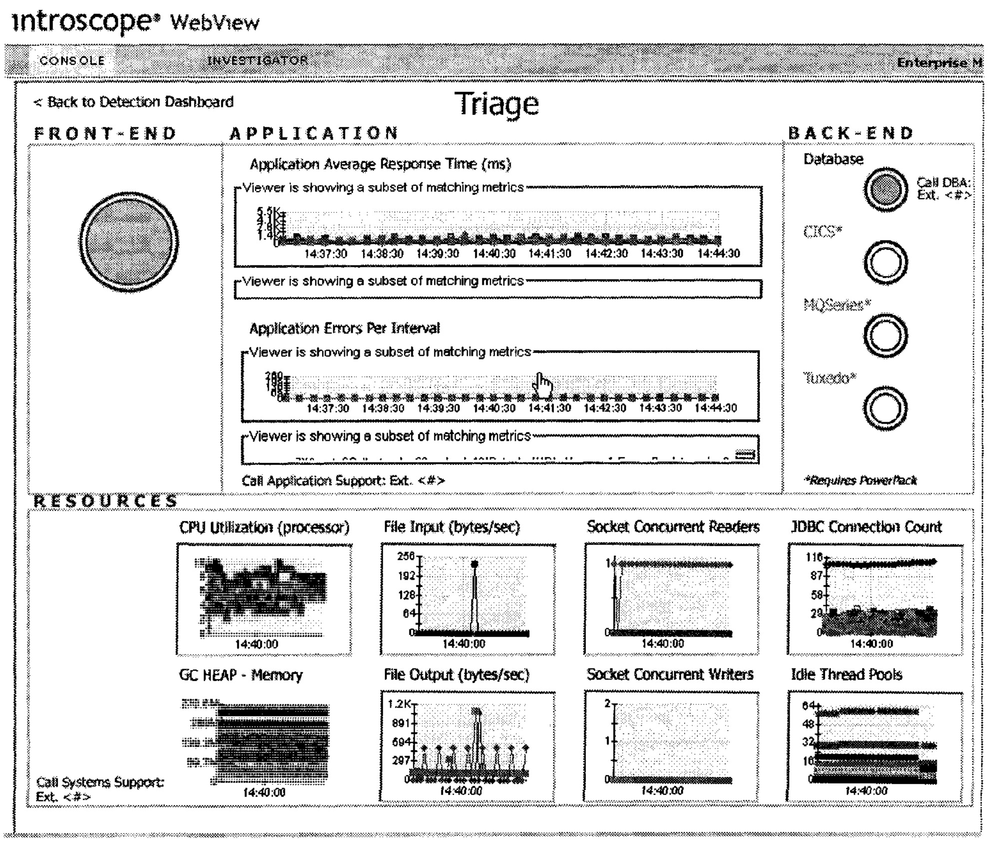

Figure 2.6: Introscope-Dashboards

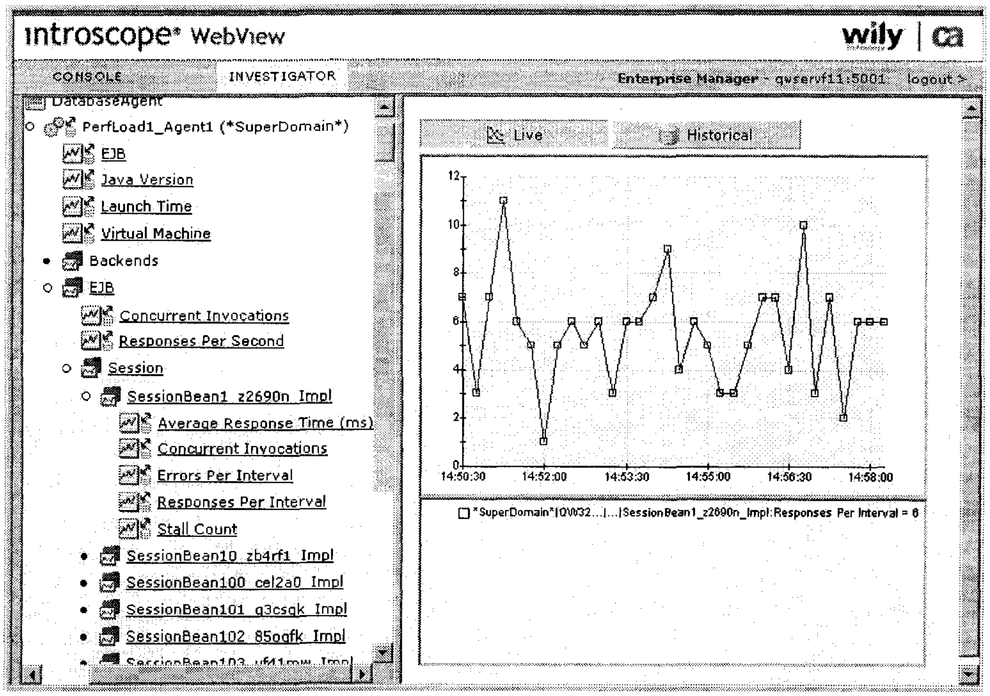

Figure 2.7: Introscope-Investigator view

in the context of the sequence of calls happening is the most notable enhancement in modern day diagnostic tools. BMC Appsight [17], Wily introscope [2], DyntaTrace Diagnostics [16] are some of the commercial products that come embedded with the ability to record trace related information in their latest releases.

2.3 Types of failures in web applications

Enterprise web systems are plagued by various performance-related problems. Table 2.1 (for middleware level failures) and Table 2.2 (network and database-level failures) enumerate a number of symptoms related to performance-based problems and their probable causes.

As is evident from Table 2.1 and Table 2.2, the symptom of a performance issue does not explicitly reveal the cause of the problem. Often what is thought of as a cause is really a symptom and one may need to investigate further to find the root

Symptoms	Sample Causes
High response time for specific transactions or most transactions	Excessive resource consumption by transaction Too much synchronization wait time Too much time to get inside the connection or server pool Improper settings such as pool size Excessive delay for external web-services Undersized system
Erratic transaction response time	Excessive garbage collection High resource utilization Erratic response of external web services
Application failures or time outs	Programming errors Improper error condition handling Data specific problems Memory exhaustion Memory leaks Socket exhaustion File handler exhaustion
High CPU utilization	Poor/Inefficient algorithms Poor design choices consuming significant time in underlying layers Poor implementation redundant work Undersized system Improper transaction routing
High memory utilization or too frequent garbage collection	Memory leaks Objects persist for unnecessarily long time Pool size too large Undersized system Lots of short lived objects

Table 2.1: Types of Faults in Web Applications-Application Tier [1]

Symptoms	Sample Causes
High network utilization between servers	Too many remoting calls Too much data transfer per call
High IO Rate	Too many SQL calls Improper database or query design Poor/Inefficient algorithms Insufficient cache Pool size too large for configuration leading to thrashing
Too high synchronization delays	Poor algorithm design - Not enough parallelism Excessive execution time for sub-transactions Locks being held for too long
Excessive resource consumption by transaction	Poor algorithms Poor design choices consuming significant time in underlying layers Poor implementation Too many remote calls Too much data transfer for remote calls Objects held for too long Poor SQL query and/or database design
Long pool queue or utilization	Too much resource consumption by transactions Large transaction execution time Database queries Incorrectly sized pool

Table 2.2: Types of Faults in Web Applications-Network and Data Tier [1]

cause. To find the root cause, it is important to identify the individual scenarios experiencing the performance problems and their execution path in the environment in which they are running. Without such information, performing problem diagnosis is the same as shooting in the dark, and it is easy to jump to the wrong conclusions.

Nevertheless, some of the causes outlined can be rejected outright on the grounds of good design, strict adherence to requirements specifications, and thorough QA work. For example, causes such as improper design choices, underperforming algorithms and implementation issues might be ruled out. This leaves only those causes that occur while the system is in operation. Still there is very little time at this point to effectively diagnose the causes with respect to the symptoms. Thus, we need to deploy effective diagnostic mechanisms that can capture errors and help prevent system components from failing.

2.4 Limitations in single-component analysis

The previous sections gave an overview of the functioning of monitoring and diagnosis mechanisms available for enterprise applications. A major need is to move ahead to comprehend the problems occurring in such applications from a scenario point of view. Although there have been powerful diagnostic tools built to diagnose and repair performance-based issues in large distributed systems, many have concentrated on the problems occurring in the individual component and fail to track the error to the context in which it occurs, i.e., the scenarios that contained the execution of the error-prone components. The commercial products that record the scenario traces and related information too are primitive at this stage, hindered by the overhead cost to the infrastructure while collecting the traces, and having the ability to only judge errors that exactly match to those that have previously been modeled into the diagnostic tools.

Often, the failing component has different sets of reasons for failure for different scenarios or user requests and unless the failure tracking models are comprehensive in matching every symptom to its related cause, many performance-related problems go unnoticed. Furthermore, it is interesting to note that the cause for such failures even change temporally based on the period for which the data was analyzed. It is thus necessary that a comprehensive analyzer does not just focus only on the component in which the error occurred but also help visualize the root cause across a chain of component executions.

On the other hand, while problem determination systems can effectively identify massive failures, subtle errors happening in the system may go unnoticed. A detailed analysis of information collected around the subtle errors generated by the components and the environment while executing individual scenarios will give the necessary diagnostic information for the system personnel to help avoid upcoming problems early. It is a challenge for the next generation diagnostic mechanisms to help identify subtle, intermittent and hard to reproduce failures early and effectively prevent their recurrence in the future.

As can be seen, capability to analyze the collected monitoring information over a period of time and effectively diagnose the cause of a problem has been a difficult research challenge in the recent years, especially with the non-deterministic nature of the errors happening in a distributed system and the disconnected nature of the monitoring information itself. Organizing this monitoring data in the form of traces provides a meaningful context to diagnostic analysis and is capable of identifying undiscovered problem areas.

2.5 Approaches to detect problems using trace analysis

The need for diagnosing problems in the context of traces led to a number of techniques being developed in the past. Each of the techniques concentrated on different sets of problems and collected traces relevant to the problem condition. This section covers various techniques used for trace collection and analysis, and the motivation that led to the design of our proposed framework. The work of Dickinson et al. [18], Yuan et al. [19], Pinpoint [20, 21], and Magpie [22] are most similar to our work. These approaches collect run-time information and use data mining techniques to simplify debugging of complex software. A major distinction to start with is to understand the instrumentation techniques used by these approaches. In particular, they use a static instrumentation to collect system data, thus impairing the flexibility of the system personnel to attach and detach instrumentation mechanisms at run-time. This flexibility of dynamically instrumenting applications is useful, considering the overhead an instrumentation mechanism can contribute to a production system (possibly up to 12% utilization of system resources, which otherwise could have been used for the production environment). Our goal is to utilize a dynamic instrumentation technique that alleviates any such problems that come with static instrumentation and give the user the ability to attach and detach the mechanism at run-time

One of the stages of the Software Development Life Cycle (SDLC) where the process of debugging errors using execution traces occurs is in the beta testing stage. In beta testing, the analysts examine execution reports coming from the users to determine the causes of failures. The work by Dickinson et al. is based on the paradigm of call profiles wherein programs are instrumented accordingly to collect relevant call profiles. These call profiles are subjected to a distance metric-based clustering to group like profiles together. Depending on the nature of one or more of the profiles in each of the clusters, executions are either termed to be failed or correct. As is the case with any other unsupervised method of learning, this methodology produces a large number of false positives by reporting unusual but normal behaviors as anomalies. Secondly, the onus is on the analyst to manually identify the cause of the problem every time. Our proposed framework moves away from the clustering technique to group like traces together and follows a much definite grouping technique based on an integer-sequence matching. Moreover, our approach classifies unusual but normal behaviors as probable failures and does not raise an alarm for every such anomaly thus capable of reducing the number of false positives.

A similar methodology but inclined more towards classification and learning techniques was adopted by Yuan et al. Their technique was based on analyzing system call traces emanating from an application that very specifically encountered reproducible failures and applying a supervised classification algorithm to label them accordingly. The correlation of the traces was then based on the labels that each of these traces had. This label also acted as the root cause of the problem. Interestingly though, Yuan et.al. concentrated only on system level call traces, which proved to be insufficient in identifying performance-based problems and their causes in applications. In general, if the cause of a problem is in a function that does not make system calls, finding its location from system call traces may not be possible. Apparently, the methodology of utilizing system call traces is to identify and debug operating system level errors and does not focus on high-level component-based failures.

Another approach that focussed on performance-related problems was devised by Cohen et al. [23] in which they correlate aggregate metrics such as CPU consumption, memory utilization and so on. The major drawback in this case too is that Cohen et.al's technique relies on the labeling of previously recorded traces into normal and anomalous. In addition, a significant limitation is the inability of the technique to identify unknown errors that have not previously been identified. Since Cohen et.al's technique too relies on classification algorithms and prior knowledge of identified errors, it makes it much more vulnerable in identifying new errors that happen. Our framework can capture any such new errors provided they can be identified by the effect-cause relationship defined by our framework.

Magpie, one of the first working prototypes of its kind, builds workload models in e-commerce request-based environments. It works by collecting event traces of every activity performed as part of a request, ranging from a context switch to an I/O operation and so on. Each of these events are then labeled using character bits. Labeled sequence-mappings are then clustered together based on the Levhenstein string-edit distance metric [24]. In the end, strings that do not belong to any sufficiently large cluster are considered anomalous requests. Finally, to identify the root cause (the event that is responsible for the anomaly), Magpie builds a probabilistic state machine that accepts the collection of request strings. Magpie processes each anomalous request string with the machine and identifies all transitions with sufficiently low probability. Events that correspond to such transitions are marked as the root causes of the anomaly. In using the Levhenstein string-edit distance, there is a possibility of grouping a normal and an abnormal scenario together as they may correspond to the same edit-distance. In such cases what may be perceived to be anomalous might in fact be normal behavior and thus ending up being a false positive. To remove this indeterminacy in grouping of like traces together, we employ a conservative trace pattern matching technique that does not out-rightly classify abnormal traces into errors, but mark them as probables.

Pinpoint works more towards a client-server environment and looks at each of the components touched as part of a client request. This approach follows the middleware instrumentation approach and follows each request to the end. It looks for faults with known symptoms (e.g., network timeouts) using an auxiliary fault detector. It is also able to detect statistically significant deviations from the norm using probabilistic context-free grammars (PCFG). Pinpoint uses two independent techniques, clustering and decision trees, to look for correlations between the presence of a component in a request and the failure of the request.

Their PCFG-based approach may not be effective on raw function-level traces due to the variability. To locate the root cause of a problem from classified traces, Pinpoint's decision trees or clustering of coverage data can be quite useful. When applied to function-level coverage data, both techniques detect only a narrow class of problems. If a problem is not manifested by a difference in function coverage across traces, it will not be detected. Pinpoint also fails to identify errors occurring in tightly coupled components and completely misses multi-threaded requests, which are quite typical these days on an e-commerce application. Furthermore, they fail to attribute component failures to bad inputs that may occur in user requests. While Pinpoint is limited towards source-code related errors and the performance degradation arising out of them, our framework would capture performance-related errors regardless of the source it comes from, either be the source-code or the environment.

Interestingly, both Magpie and Pinpoint make heavy use of data mining principles of clustering and classification and hence require an iterative learning process or sensitive parameter tuning to improve their diagnostic capability.

2.6 Summary

In this chapter we provided a concise yet descriptive explanation of diagnosis and looked at the capabilities of diagnosis techniques.

The introduction outlined the history of the term diagnosis and its relevance to current practices. A study was cited, reflecting the need for effective diagnostics technology in web applications. Since the proposed framework was designed for enterpriser applications, a brief overview of the different types of components in enterprize applications was provided.

This chapter also gave a general overview of diagnostic capabilities and tools and outlined the typical architecture that these tools have, the log formats used by them and the analytics employed by these tools to effectively weed out candidate causes of problems. The chapter also looked at the different types of performance and resource related problems that frequently occur in web applications and the possible causes related to each symptom. Since each symptom has more than one cause associated with it there is the need for a framework that helps the analyst to discard causes for a given set of symptoms and quickly focus on the direct ones to resolve problems effectively.

Due to the limitations in diagnosis of individual components, a need for analyzing scenario flows in a typical web application rather than only individual components in the web applications was discussed. Related work was summarized on trace-based diagnostic frameworks specifically built for analyzing performance and resource-based problems in web applications. We concluded by emphasizing the need for a framework that looks at subtle and hidden errors from scenarios of massively distributed enterprise web applications.

Chapter 3

Approach

This chapter starts by emphasizing the need for recording scenario traces and related component information to identify errors undetected by normal monitoring mechanisms. The main focus is to provide a detailed overview of the hypothesis based framework designed to identify performance and resource-related problems occurring in web applications. The framework is illustrated through a proof of concept scenario that focuses on a typical performance-related problem. Visualization features forming a part of the diagnosis process are also outlined. The chapter then concludes by summarizing the base model of the diagnostic framework and the fundamentals upon which further enhancements of fault models can be built.

3.1 Scenario trace analysis - Motivation

As discussed in Chapter 2, enterprise applications are moving away from large monolithic systems and simple client/server configurations to highly interconnected, multi-tier, distributed architectures designed to run on a heterogeneous collection of servers. In such environments, each of the different servers and components that comprise the system generally produce a large amount of monitoring data. Although this information can be vital in analyzing the health of the components individually, it remains a major challenge to stitch this information together to get a conclusive understanding of the overall system behavior.

Debugging performance and resource-related problems in this situation is problematic because the analysts would have to correlate the entire monitoring information over an enormous infrastructure built out of individual components. It makes the scenario even more difficult in the cases where subtle, intermittent errors happen, that may or may not trigger alerts in individual component-monitoring mechanisms and thus fail to provide relevant information to help track the cause of the problem. Analyzing such errors in the context of the scenarios that were affected could give a better hint to the cause. Usually, the failing component functions normally in the other scenarios but fails due to unexpected reasons in very few ones.

Also, replicating such errors is difficult because they are not deterministic in nature. Typically, the system personnel are left pondering over the reason for failing user requests, while waiting for these errors to reoccur, to pinpoint the cause of the problem.

It is known that the overhead incurred by the monitoring and diagnostic mechanisms affects the performance of the web systems being managed. Current tools only monitor critical information required for the healthy behavior of the production system and often do not capture additional information to help diagnose occasionally failing user requests, so as to keep the resource overhead within the stipulated levels. When comprehensive information about a trace is needed, monitoring mechanisms record information only for a very short period of time to reduce the incurred performance and resource overhead.

Recently, system personnel and infrastructure management teams have realized the importance of debugging issues at the scenario level versus raising alarms for individual component degradations alone. Moreover, it is much easier for the system analyst to debug the issues from a user's perspective by looking at the cohesive connection of component calls rather than trying to identify what caused an individual component to fail. Correlating monitoring data over a set of such scenario traces could help in better locating the causes of a problem.

Currently, there is a need for a diagnostic mechanism that can collect the required information, in the context of scenario traces, with low overhead while at the same time also collect enough information to anticipate upcoming degradations and to help safeguard the system from them.

3.2 Fault Model

For our framework, we now define a fault model that is representative of most of the performance-based problems and related ones occurring in a typical web application. The model focusses on a class of problems ranging from excessive resource consumption to components having programmatic errors. For each problem, we characterize the trace corresponding to the error and the information required to initially analyze the problem. The following subsections provide a very high-level overview of the different classes of problems effectively diagnosed by our framework.

Host setup and configuration problems

Application servers that are mis-configured or sustain performance issues over time usually have execution prematurely ending or terminating, or have parts of the program spending too much of time in execution when requests pass through them. The same parts of program may have executed normally on other hosts in the distributed environment, but either fail or consume more time (other resource issues can also occur) while running through specific host(s). What makes such problems susceptible to be missed by the current diagnostic mechanisms is that the hosts' configurations are constantly changing and what was running perfectly for a period in time no longer runs due to a very small update in the configuration settings. Furthermore, such configuration changes usually affect only a small set of components in this environment while most of the components usually have a problem free execution history on the same host environment.

Database problems

Whenever a database failure occurs such as a RAID disk failing, the I/O rate tends to exceed the normal processing rate as the bulk of the operations are carried out by the remaining disks. As a result, the components that contain database calls and query processing have slower execution times than the average. The problem in this case is not with the components or the application servers but the database response. This has to be correctly identified in cases where the response is too slow and the execution of the calling component times out.

Deadlock and starvation problems

If a small number of processes deadlock, they stop generating trace records, making their traces different from processes that still function normally. Similarly, if a process consumes large amounts of CPU time, the other processes waiting for resources start starving thus causing the user request to be terminated prematurely or ending up spending large amounts of time in different sections of the application code. Such processes can either happen at the application level or at the database level and it is necessary to pinpoint the cause to the proper section of the web application. Race conditions or starvation problems in application servers are typical examples of this category of problems.

Point-of-time errors

System settings and resource utilization levels (memory utilization, CPU utilization, I/O rate etc.,) may lead to transient problems, affecting the normal execution of components. Such problems may be hard to identify as they are in-deterministic in nature and usually do not have a pattern.

Although these errors may happen for a very short period of time and fade away depending on the capability of the host environment to adapt itself, they can become a potential threat to the resources, if left unnoticed. It is important to identify such issues early to reduce downtime later.

3.3 Scenario trace data collection

Scenario trace data collection or run time path tracing has long been a concept of interest for monitoring system performance but systems people have not been able to harness the full potential due to various hindrances. The first and foremost step involved in collecting scenario trace data information is to instrument the underlying application strategically so as to collect enough information without hampering the system performance. Much research has gone into the various points of instrumentation in enterprise-level systems and each of the techniques were found to have their own set of advantages and disadvantages. Figure 3.1 shows an overview of the different points of instrumentation possible in an application.

Based on the context of recording component calls and the order in which they are executed, source code instrumentation is an effective technique for collecting scenario trace data. This instrumentation technique acts on the underlying application at a much coarser level of granularity as opposed to the other types of instrumentation that deal with byte-code.

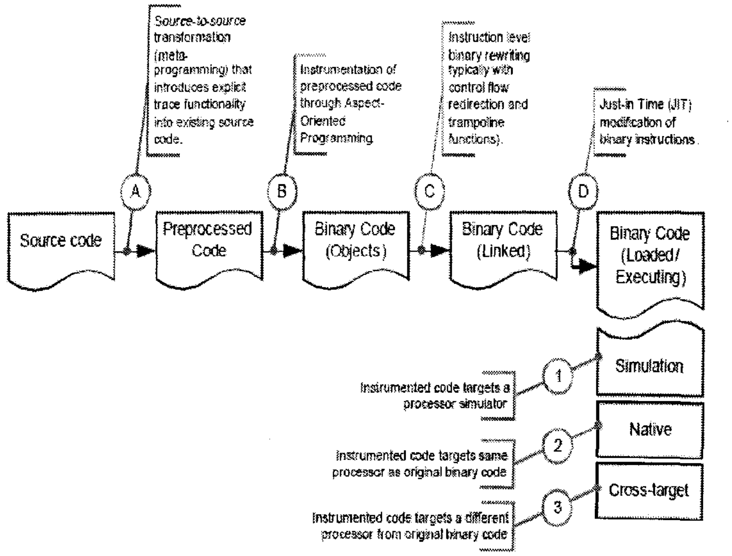

Figure 3.1: Instrumetation techniques hooking into different sections of the system. A comprehensive set of code modificaiton points that can be used to inject instrumentation code. [3]

Source code instrumentation of the application also cuts down on most of the overhead involved in byte code instrumentation by ignoring unwanted system level details such as memory registers and cache buffers etc. Byte code instrumentation is better suited for on-the-fly attaching and detaching of instrumentation probes, but in the context of recording information for a scenario that spans across multiple system hierarchies, it is easier to have the instrumentation probes to be static and placed well before the application is deployed. But again, the user is hampered in this type of static instrumentation technique to have a prior knowledge of the application source code, which is not possible in most of the cases. This limitation is in addition to the resource overhead generated by static instrumentation.

It is ideal if the instrumentation comes with the ability to instrument source code and also dynamically insert probes at the required points, thus achieving a minimal user intervention at the source code level and enough flexibility to turn-on and turn-off the instrumentation. COMPAS [25] is one such instrumentation framework that performs pseudo-source code transformation to install probes without hooking directly into the application middleware but using the components' deployment descriptor. These deployment descriptors contain the meta-data (information regarding the structure and behavior of the component) needed to derive the internal structure of the components. The probe in COMPAS acts as a proxy element and has a one-to-one relationship with each of the application's component and uses the meta-data information to weave a proxy layer on top of it, which then tracks the component call in the context of an execution trace. COMPAS thus achieves the objective of source code instrumentation without requiring a prior knowledge of the application source code. COMPAS is highly customizable and with its ability to instrument all the three tiers of the web application, recording scenario trace information is made easy.

3.3.1 Considerations for scenario trace data collection

Today's enterprise applications are generally required to handle high loads of traffic generated by concurrent users. Runtime path tracing in such systems involves tracing each of the user requests as they pass through the different tiers that make up the entire application. Sequences of such requests for a user session form a scenario. For example, in an online banking system, a user performs a set of activities that consist of logging in, viewing the account information, transferring money between accounts, executing an online payment and finally terminating the session by logging off. To effectively analyze information gathered around these set of activities that form a scenario, it is necessary to preserve a unique identifier across the chain of component calls in the scenario and maintain the order in which the components were called. Ideally, an instrumentation mechanism needs to meet the following requirements to be able to perform effective runtime path tracing:

- (Req1) The mechanism must be able to identify between different user requests entering the system at the thread level and generate a unique scenario-id for each such request.
- (Req2) The mechanism is required to tag the scenario-id with request-specific information (RSI) so that component calls and related information can be mapped to the corresponding scenario-id.

Another important aspect of this scenario-id generation is about defining the lifetime of the scenario-id. Since each scenario consists of the set of actions performed between the first request issued by the user (logging into the system) and last request (eventually logging off), a unique session-id is generated every time a user logs in and is discarded once the user logs-off. In certain circumstances, where the user terminates the session by either closing the application window or not performing any action for a particular period of time, the user has to authenticate himself again before resuming the next session, thus issuing a request for a new scenario-id to be created. Achieving the segregation of scenarios can be particularly challenging when a system is distributed across a network.

The RSI (Req2) refers to the component related details with respect to the particular request. The kind of component data that needs to be recorded is outlined in Section 3.3.2.

Thus far, there have been tools that have the above requirements met in a manner that involves instrumenting either the software application itself or the underlying middleware and collecting the necessary information. Because the J2EE specification does not explicitly address the requirements for runtime path tracing, middleware approaches for J2EE have involved using non-standard, non-portable and vendor-specific mechanisms. The biggest issue with using non-standard mechanisms is that developers get locked into using a particular middleware instrumentation and lose the flexibility of being able to quickly change this mechanism should the need arise. Another drawback happens to be the inability of these instrumentation mechanisms to work seamlessly across multiple execution environments such as J2EE and .NET.

This calls for non-intrusive, multi-platform instrumentation that works across a distributed architecture consisting of middleware and back-end systems, regardless of the vendors. So far, instrumentation mechanisms have been successful in evolving to support application servers from multiple vendors that run on either Java Virtual Machines or .NET VMs but fail to work for cross-integration between the two. Commercial diagnostic tools that come bundled with their own instrumentation mechanisms usually have different versions for application servers running on different platforms. Open source instrumenting mechanisms, on the other hand, such as COMPAS, do have conceptual techniques to integrate between different platforms, but still have not been successful in producing a working version.

3.3.2 Trace data collection format

The first step involved in any diagnostic framework is to collect enough data. The most important issue though in fulfilling this objective is to not overrun limited resources of the system being instrumented.

To record data with respect to scenario traces, the ideal part of the system that can be instrumented is the middleware. As shown in Figure 2.1, most of the application logic is concentrated in the Business Tier and all scenarios flow through the

Business Tier for processing. Any end-to-end scenario can be tracked from a single point, the Business Tier of the deployed application. So the instrumentation mechanism is always better placed in the middleware housing the business logic of the application. Apparently, due to the limitations of the instrumentation mechanisms, as outlined in Section 3.3.1, deployed web applications still need to be running on a single type of virtual machine across the entire stack of application servers, to be able to track the end-to-end information from scenario traces.

The runtime path tracing approach discussed in Section 3.3.1 requires user request tagging (Req1). One way of achieving this is by inserting a unique value into the local thread object when a new user request enters the system. Modern day application frameworks usually look after the process of generating unique user requests at the middleware level. Conversely, if user requests are restricted to entering the system in the Web Tier, then the HTTP server (a component of the web server) can be instrumented such that each new request for an HTTP connection can be tagged. In this case, a new request for an HTTP connection would represent a new user request.

On each invocation of a component corresponding to a user request, following information is logged:

- Unique scenario-id
- Thread id
- Sequence number for components called
- Call depth (for finer granularity recording)
- Component details (Name and context)

- Execution environment details

- Performance data

Component calls are then grouped together based on the scenario-id, which corresponds to a single scenario. A set of component calls ordered in the sequence in which they were executed and tagged to the same scenario-id forms the scenario trace, which is then ready to be processed. Table 3.1 shows a typical scenario trace comprising of a set of activities performed by a user over a single session of log-in. Table 3.1 also contains the scenario-id preserved across the entire scenario trace.

In addition to the above mentioned purpose, further analysis can be applied to scenario trace information for a number of additional purposes. For example, by analyzing the runtime paths, developers can easily construct UML diagrams that can help to deduce the overall system structure. Component relationships can be identified, which helps developers gain a better understanding of their system. For instance, understanding inter-component relationships enables developers to anticipate potential conflicts and debug problems as well as allowing developers to reason about their system design (which in turn can have a major impact on system performance).

3.4 Diagnostics Framework : Design Overview

The framework was designed to tackle the problem of eliminating the necessity of manually checking every possible cause associated with the detected problem and minimize the overall time taken to resolve the problem. Apart from locating the section of the system that is a potential cause, the framework supports its hypothesis with a set of related diagnostic information that would help the analyst to better understand the situation. To diagnose any detected problem, the framework

Scenario Id	Component Type and Name	User requests
UUID-1084477462-965986-1930759028	JSP—template-jsp—jspservice—/bank/mainservice;SPUTINOW	
UUID-1084477462-965986-1930759028	JSP—template-jsp—jspservice/bank/accountListenerService;SPUTINOW	
UUID-1084477462-965986-1930759028	EJB—Session—TxControllerBean—getTxsOfAccount;SPUTINOW	
UUID-1084477462-965986-1930759028	JTA—TransactionImpl—Commit;SPUTINOW	
UUID-1084477462-965986-1930759028	EJB—Session—AccountControllerBean—getDetails;SPUTINOW	Account History checking
UUID-1084477462-965986-1930759028	JTA—TransactionImpl—Commit;SPUTINOW	
UUID-1084477462-965986-1930759028	JSP—template-jsp—jspservice;SPUTINOW	
UUID-1084477462-965986-1930759028	EJB—Session—TxControllerBean—getTxsOfAccount—/bank/accountHistService;SPUTINOW	
UUID-1084477462-965986-1930759028	JTA—TransactionImpl—Commit;SPUTINOW	
UUID-1084477462-965986-1930759028	EJB—Session—AccountControllerBean—getDetails;SPUTINOW	
UUID-1084477462-965986-1930759028	JTA—TransactionImpl—Commit;SPUTINOW	
UUID-1084477462-965986-1930759028	JSP—template-jsp—jspservice;SPUTINOW	
UUID-1084477462-965986-1930759028	JSP—template-jsp—jspService—/bank/mainservice;SPUTINOW	Funds Withdrawl
UUID-1084477462-965986-1930759028	JSP—template-jsp—jspService—/bank/atmservice;SPUTINOW	
UUID-1084477462-965986-1930759028	EJB—Session—AccountControllerBEan—getDetails;SPUTINOW	
UUID-1084477462-965986-1930759028	JTA—TRansactionImpl—Commit;SPUTINOW	
UUID-1084477462-965986-1930759028	EJB—TransactionControllerBEan—withdraw;SPUTINOW	
UUID-1084477462-965986-1930759028	JTA—TRansactionImpl—Commit;SPUTINOW	
UUID-1084477462-965986-1930759028	EJB—Session—AccountControllerBEan—getDetails;SPUTINOW	
UUID-1084477462-965986-1930759028	JTA—TRansactionImpl—Commit;SPUTINOW	
UUID-1084477462-965986-1930759028	JSP—template-jsp—jspService—/bank/atmAckservice;SPUTINOW	
UUID-1084477462-965986-1930759028	JSP-template-jsp-jspservice;SPUTINOW	Transfer Funds
UUID-1084477462-965986-1930759028	EJB-Session—TxControllerBean-transferFunds;SPUTINOW	
UUID-1084477462-965986-1930759028	JTA-TransactionImpl-commit;SPUTINOW	
UUID-1084477462-965986-1930759028	JSP-template-jsp-jspservice;SPUTINOW	
UUID-1084477462-965986-1930759028	JSP-template-jsp-jspservice;SPUTINOW	

Table 3.1: Illustration of a typical scenario trace consisting of unique scenario-id and including multiple user requests

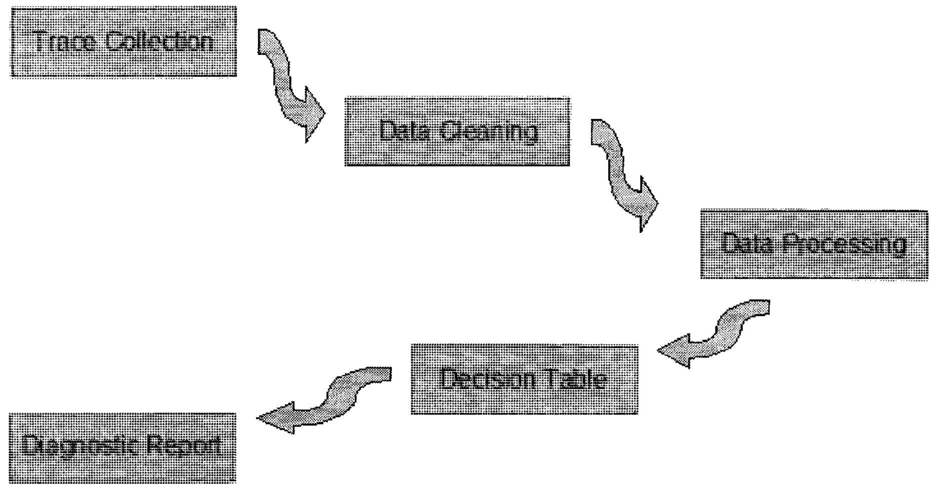

Figure 3.2: High level design overview of the diagnostic framework

follows a three-step process. First, we collect component-level traces using an instrumentation technology available for J2EE architecture based on the principles of low overhead instrumentation discussed in Section 3.1.1. The COMPAS technology with enough customizations writes out component calls of each scenario to log files in the format described in Section 3.3.2.

The second step is to extract relevant information from the collected log data and categorize them based on premature termination or difference in behavior as compared to perceived normal traces. In this step, we look at two kinds of cases, those that have terminated component execution and those cases that significantly deviate from a normal pattern and may pose potential threats to the system.

In the third step, the framework uses trace information collected over time to identify possible faults and tries to provide relevant data to the analyst to help debug the issue. Also in this step, we feed the information collected from the faulty scenarios and similar traces to a primary and secondary decision table and try to output highly probable causes corresponding to the problem encountered in addition to eliminating inapplicable ones. Figure 3.2 outlines the diagnostic framework, which expands the three steps described above into individual processing stages.

Furthermore, these techniques both look at performance issues and programmatic errors in components of the web system and efficiently correlate information from both aspects to help identify the causes. What eventually remains is the distilled set of causes related to the problem at hand with the other causes eliminated by the decision logic.

3.4.1 Data Preprocessing

The data collected from the scenario traces using the COMPAS framework is in a raw format. Appendix A illustrates the raw data collected for one executed component of a scenario trace. Parsing the raw data and extracting relevant information as needed by the diagnostic framework forms the data preprocessing step. Since the COMPAS framework can be customized to record a lot of information about a component, our framework focusses on only the subset of information as recorded in the sample log data given in Appendix A. This limits the overhead involved in gathering run-time information about the components of the web application. The following subsection outlines the format of the preprocessed data that contains the relevant information needed by the framework.

Extracting relevant information from raw traces

Each component call that is recorded as part of a scenario, represented by its unique scenario-id and thread-id, is logged to a central location in the middleware deploying the web application. The individual log files need to be normalized to a single time zone for effective processing. Synchronizing log data across multiple systems and various sources requires processing the time stamps and writing out the log information collected in a cohesive time order to a central location. This capability of COMPAS helps us to correctly navigate the centralized log data and also facilitates speedy processing.

The instrumentation mechanism logs trace data information to a predefined location which is then collected and preprocessed to obtain the necessary information. The preprocessed log files then contain data in the format as shown below. Since the unique session-id generated by COMPAS is a complex sequence of integers, we simplify the session-id in the preprocessing stage to a unique integer random number.

```
{scenario_id;
    thread_id;
        component_name;
            host_name;
                execution_time;
                    database_calls;
                        query_execution_time}
```

3.4.2 Trace mapping and segregation

Once the traces have been preprocessed and data organized in the required format, the next important step is to identify patterns within the traces and segregates normal looking traces and faulty ones accordingly. To achieve this objective, we outline the following assumptions.

Assumption 1: *A scenario that terminates prematurely i.e., whose execution does not have a corresponding component exit to the respective component entry, is deemed to be a definite faulty scenario.*

Assumption 2: *A scenario that does not have a prematurely terminating component in its trace but has a pattern that is similar to the definite faulty scenario trace and has a low occurrence rate, is deemed to be a probable faulty scenario. In order to judge similarity, a subsequence in the probable faulty string should exactly match the definite faulty scenario.*

But before applying the assumptions to separate traces into normal and faulty ones, we propose a mapping process that represents the component calls in a scenario trace as an integer sequence. This representation simplifies the task of processing a large log file containing thousands of scenario traces and their related information. While each of the techniques described in Section 2.5 employ classification, clustering, or string-edit distance to group like traces together, our framework makes use of a less process intensive integer mapping to group traces together.

To order the scenario traces, each component is mapped to a unique integer index. So, each time a component call is recorded, the preprocessor checks a hashtable for an existing index for the component or assigns a new one if no index exists. Table 3.2 and Table 3.3 represent a typical use case consequently mapped to a sequence of indexes.

The integer sequences are then subjected to a segregation process in which the definite faulty scenarios and probable faulty scenarios are separated from each other based on the formulated assumptions. This marks the completion of the first phase of the diagnostic framework.

Trace data in the preprocessed log files needs to be associated with the com-

Scenario Id	Component	Host	Execution Time	Component Index
105421439	validateServicePac-entry	mithrandir	0.027s	1
105421439	validateRequest-entry	mithrandir	0.023s	3
105421439	validateRequest-exit	mithrandir	0.025s	3
105421439	checkCompatibilityRequest-entry	mithrandir	0.005s	7
105421439	checkCompatibilityRequest-exit	mithrandir	0.0365s	7
105421439	getWarrantyInfoRequest-entry	gimligloin	0.0258s	13
105421439	getWarrantyInfoRequest-exit	lauds	0.0125s	13
105421439	checkHardwareWarrantyWithinPeriodRequest-entry	gimligloin	0.0003s	5
105421439	checkHardwareWarrantyWithinPeriodRequest-exit	gimligloin	0.0258s	5
105421439	validateServicePac-exit	gimligloin	0.0254s	1

Table 3.2: A successfully executed scenario trace depicting a set of normal user requests

Scenario Id	Component	Host	Execution Time	Character Mapping
105421439	validateServicePac-entry	mithrandir	0.027s	1
105421439	validateRequest-entry	mithrandir	0.023s	3
105421439	validateRequest-exit	mithrandir	0.025s	3
105421439	checkCompatibilityRequest-entry	mithrandir	0.005s	7
105421439	checkCompatibilityRequest-exit	mithrandir	0.0365s	7
105421439	getWarrantyInfoRequest-entry	gimligloin	0.0258s	13

Table 3.3: Scenario trace depicting premature termination

Trace	Associated Files	Format
1-1-2-2-3-3-4-4-5-5-4-4-1-1-	1-1-2-2-3-3-4-4-5-5-4-4-1-1-m1.txt	*m1_host;m1_exec_time;m1_query_time*
	1-1-2-2-3-3-4-4-5-5-4-4-1-1-m2.txt	*m2_host;m2_exec_time;m2_query_time*
	1-1-2-2-3-3-4-4-5-5-4-4-1-1-m3.txt	*m3_host;m3_exec_time;m3_query_time*
	1-1-2-2-3-3-4-4-5-5-4-4-1-1-m4.txt	*m4_host;m4_exec_time;m4_query_time*
	1-1-2-2-3-3-4-4-5-5-4-4-1-1-m5.txt	*m5_host;m5_exec_time;m5_query_time*

Figure 3.3: Example data structure for the framework

ponent integer sequences. For this, we organize the integer sequences with their associated data in a file based data structure for efficient and quick retrieval of information corresponding to individual components. Figure 3.3 outlines a typical data structure that contains trace data information for a single scenario trace. This tree-based structure helps in extracting the required pieces of information related to a single component in question whose data is normally spread over the voluminous log file. The rationale is to speed up the processing of the analytical portion of the diagnostic framework.

3.4.3 Identifying faulty components in definite faulty scenarios

Once certain traces have been identified as definitely faulty (scenario traces having premature termination), the next step is to identify the component which caused the scenario to terminate prematurely. One place to start looking at the cause for the failure is the last component in the chain of component calls that experienced premature termination and then work up the chain to help identify the cause. For example, a scenario trace represented as *1-3-3-7-7-13-* indicates that the component *(1)* and *(13)* did not have a matching exit. To start, the most eligible candidate to be scrutinized would be the last component in the call chain that failed out-rightly and caused the scenario trace to terminate, i.e., *(13)*. This would then be followed by the next candidate in the chain of component calls that did not have a matching exit i.e, *(1)* in this example. The framework would also list out normally executed components such as *(3)* and *(7)* as candidates in the faulty scenario trace if these components have been tagged as eligible candidates in other scenario traces (such as one with *(3)* prematurely terminating). The framework considers the candidates in the above order to identify the cause along the chain of component calls rather than traverse backwards from the initially terminated component and analyze each of the components executed along the call path. In some scenario traces the root cause may not be in any of the candidate components identified by the framework. In such cases, the framework can be adjusted to analyze remaining components in the trace once the initial candidates have been analyzed.

Hence in the example mentioned in Table 3.3, the first candidate component that would be analyzed is *(13)*. Every piece of detail corresponding to the component *(13)* is then fed to a set of decision tables for processing to identify the causes and determine the most probable ones. In this case, the files corresponding to the faulty scenario that contain data for component *(13)* as well as other files that had data corresponding to component *(13)* being successfully executed form the inputs to the decision making module. The same process is followed for *(1)* and any other components possibly having failures in other scenario traces.

3.5 Decision Making Process

The next stage involves the decision making process in which information about the faulty scenarios is then compared against properly terminating scenarios to determine the reason behind the failure of the component. The goal here is to point out the possible cause of the failure while eliminating most other causes through historical data analysis.

The decision making process tries to relate symptom information extracted from the scenario traces and runs them through a set of checks. This correlation is done across data such as the host machines on which the components ran, the average execution time, the existence of database calls and the query execution times. This is the minimal information required to diagnose the problems outlined in the fault model.

The problems encountered in a typical distributed web application are related to a huge number of possible causes and each has a rich set of interleaved symptoms attached to it. This makes it very difficult for the analyst to identify correctly where the problem is and what caused the problem. Still, in most commercial diagnostic tools, anaysis is real time and thus scenario trace data ia avoided due to relatively costly processing. Even diagnostic tools that do analyze scenario traces do not correlate symptom information over a period of time and identify those parts of the system which caused the scenario to fail.

To help the analyst to reveal the root cause of problems, our framework is designed to identify the causes using scenario traces over a substantial period of time and zero-in on the exact cause. The discovered causes are somewhat sensitive, however, to the time period considered. For now, we allow the operator to run correlations over different time periods of collected traces to help localize the cause.

Typical diagnostic checks have been converted into a decision table model consisting of conditions, actions and rules (shown in Figure 3.4). The decision table shows the possible symptoms that reflect each of the problems described in the fault model and outlines the effects. The predicates that define the conjunction and disjunction of conditions are defined in the Rules column. For example, while considering the fifth column in Figure 3.4, the conjunction of four conditions such as the candidate component (C(x)) in the faulty scenario having run on the same host, C(x) having run successfully in other scenario traces, C(x) having run on the same host as both the faulty and normal scenarios and C(x) taking more than average execution time, fires up an effect that says that the component might have failed in the faulty scenario traces due to resource utilization issues in the Application Tier. The other conditions do not hold true in this case and can hence be eliminated from second-level checks. These effects then form the basis for executing the set of checks in the secondary decision table. The modeling of the effect-cause relationship for each of the effects in the primary decision table is taken care of by a secondary decision table, detailed in Section 3.5.1.

For each faulty scenario that needs to be analyzed, a series of such checks are performed by the primary decision table and the corresponding effects are then presented to the secondary effect-cause decision tables that run a set of second-level checks that output the probable cause in the form of a diagnostic report for

	Application Tier Failures		**Rules**							
Conditions	C(x) has run on the same host		Y	Y	Y	Y	Y	Y	Y	Y
	C(x) has same input paramters for all failing requests		N	Y	N	N	N	N	N	N
	C(x) has run successfully in other scenarios		N	N	Y	Y	Y	Y	Y	Y
	C(x) has run on the same host as both the faulty scenario and the normal scenario		N	N	N	Y	Y	Y	Y	Y
	C(x) has taken more than the average execution time		N	N	N	N	Y	Y	Y	Y
	Data Tier Failures									
	C(x) contains database calls		N	N	N	N	N	Y	Y	Y
	Query was sucessfully executed		N	N	N	N	N	N	Y	Y
	Query execution times in C(x) is greater than average		N	N	N	N	N	N	N	Y
Effects	C(x) has a problem running on H(x)		X	X						
	Problem in H(x) while executing C(x)		X		X					
	Rresource utilization issues in Application Tier casued problem in C(x)					X	X			
	Database connection problems							X		
	Resource utilization issues in Data Tier casued problem in C(x)								X	X

Figure 3.4: Decision table designed as per the Fault Model. C(x): Candidate component in the faulty scenario. H(x):Host on which C(x) was executed for the faulty scenario

the system personnel. In the primary decision table, the checks are divided into Application Tier checks and Data Tier checks. The system personnel is given the flexibility to choose the type of checks to apply.

The rationale for choosing decision tables is that they are one of the most powerful and highly efficient rule based models and are often preferred for very complex or extensive sets of conditions.

The framework currently handles all problem situations discussed in Section 3.2, and is extensible to new types of problems. The rule checker in the framework is modeled to effectively iron-out these errors but is not rigid in its implementation to look out only for such types. As was discussed earlier the rule checker can be extended to include other significant set of errors depending on the system being monitored. An important analysis here would be to mention that an extended rule checker wouldn't hamper the performance of the framework significantly as the data structure that logically organizes the trace data is so designed to extract only the data needed for the current rule being processed and doesn't have to look into the log file every time a rule is checked, whereas the other techniques discussed in Section 2.5 involving classification and learning tend to become slower with the induction of additional fault finders. Apparently, this facilitates the process of converting the framework into a much more dynamic mechanism and help detect errors at run-time.

3.5.1 Effect-cause relationship

Narrowing down to a set of effects in the primary decision table modularizes and simplifies the process of identifying the causes. We represent the effect-cause relationship as a secondary decision table, since a single effect can have multiple causes associated to it. While the preliminary table identifies the most probable effects(s), the secondary decision table performs a set of checks to identify the most probable cause(s) pertaining to that effect.

Based on the conjunction of predicates in the effects category, corresponding checks in the causes category are executed. The combination of conjunctions and disjunctions (rules) in the effects category reflects the effects identified by the primary decision table.

While cause-based checks are mostly based on the details collected at a particular point in time when a scenario trace fails, checks involving the CPU consumption, memory consumption and I/O rate are based on a comparison against average values determined from the normally executed scenario traces. The average values are calculated for each of the components from the normal traces and anything in excess of a stipulated threshold is marked as a failure. As was mentioned previously, these thresholds are usually based on statistical measurements learned over time and differ from one diagnostic tool to another. The thresholds are set at a global 98 percentile in our framework, based on values collected over the time span of the log. For example, the thresholds for CPU consumption, memory consumption and I/O rate could be 15%, 7% and 12% over the respective averages.

An illustrative example depicting a typical resource related problem is outlined in Section 3.5.2. This section walks through the primary and the secondary decision tables to identify the effects corresponding to the problem condition and the outcome in the form of most probable causes.

			Rules					
Effects	C(x) has a problem running on H(x)		Y	Y	N	N	N	N
	Problem in H(x) while executing C(x)		N	Y	Y	N	N	N
	Resource utilization issues in Application Tier casued problem in C(x)		N	N	N	Y	N	N
	Database connection problems		N	N	N	N	Y	N
	Rresource utilization issues in Data Tier casued problem in C(x)		N	N	N	N	N	Y
Causes	C(x) fails for the same input parameter		X	X				
	Change in H(x) configuration settings		X	X				
	High CPU utilization			X	X	X		
	High memory consumption			X	X	X		
	High I/O rate - Application and Database			X	X	X		X
	Improper pool utilization- Application Server Thread Manager			X	X	X		
	Application thread contention			X	X	X		
	JDBC connectivity/Improper pool utilization - Database Thread Manager						X	X
	Database deadlocks							X

Figure 3.5: Effect-cause decision table

3.5.2 Illustration

This section illustrates the working of the framework through a sample scenario to better understand the diagnosis process. This explanation of the diagnostic framework and its decision making process is from the collected traces of a scenario with only one use case. The scenario considered is a simple purchase order use case where a request is placed to purchase a resource, followed by order confirmation and delivery stages. Many intermediate processes such as checking for resource availability, warranty information, etc., are captured as component calls.

Table 3.2 shows a typical scenario trace that corresponds to the use case outlined above. The log files that collected the scenario traces are processed to extract information such as the scenario-id, component name, host name, execution time, etc. The diagnostic system also counts the number of instances of each scenario as an additional statistic.

Table 3.3 shows the same scenario but with a premature termination at the servlet *getWarrantyInfoRequest* or component *(13)*. On feeding the scenario traces and related information as shown in Table 3.2 and Table 3.3 to the diagnostic framework the problem area was narrowed down to the host *lauds*. The following paragraph details the functioning of the primary and secondary decision tables to identify the exact cause.

Based on the checking process outlined in Section 3.5, the data collected for the faulty trace *1-3-3-7-7-13* was analyzed. A conventional diagnostic tool would pull up an alert for the failing component in question i.e, *(13)* and identify the problem to be that of a worker thread not being allotted. In addition it would also come up with possible causes of server settings, heavy resource consumption by the transactions, excess synchronization time for the current threads, slow database queries, incorrectly sized pool for the application server as well as the database. As can be seen, there are many different causes for a single type of problem condition or symptom. The proposed framework tries to eliminate most of the irrelevant ones and narrows it down to the most probable cause(s).

To start with, an analysis is done within the same set of collected traces to see if the method *(13)* had run on the same host i.e, *lauds*. In this case, *(13)* had run on the same host in all the collected traces. The next check sees if *(13)* had successfully run in a normal scenario trace where the faulty trace is a subsequence.

Since the component *(13)* had successfully run in a normal scenario, a third check sees if the hosts that ran *(13)* successfully in the normal trace match the host that ran *(13)* in the faulty scenario. This determines if *(13)* had problems running just on the host *lauds*.

The next check sees if *(13)* had taken more than the average execution time in the faulty scenario as compared to the normal scenario. In this case, *(13)* had taken more than average execution time. To check if the excessive execution time was due to Java or database calls, the next check verifies database calls. Since there were no database calls in the component, the decision table drills deeper into the possible causes for processing delays of a Java component, which is taken care of by the secondary decision table. These checks include heavy CPU consumption by other components, I/O delays, and finally thread allocation and contention issues. Based on the analyzed information from components running at the same time, the CPU consumption and I/O delays are ruled out, while failing thread allocation remains as a possible cause. Thus, diagnosis proposes the problem cause to be a thread allocation issue, and prompts the human operator to look into the thread manager of the application server for more details. Effectively, we rule out a number of options automatically in terms of effects (primary decision table) and causes (secondary decision table) and zero-in on the problem area successfully. Contrarily, other diagnostic tools analyzing just the details of the failing component usually output an exhaustive set of problem causes and leave it to the human operator to check each one out. By correlating trace information over time, we are successful in eliminating a majority of the proposed causes and leave out only the most probable ones. The same process applies to the entire set of problems outlined in the fault model.

On a general note, the framework is successful in answering a few interesting questions such as:

- Whether component x failed on a host while having successfully run in other hosts?
- Is the component failure due to bad design or environment settings?
- Are definite failure scenarios the only failure cases in the system?
- Do two similar faulty scenarios failing at the same component share the same symptom?

These are typical questions whose answers very much help in ruling out irrelevant causes and pinpointing the problem area quickly.

3.6 Visualization

One of the most important feature of a diagnostic tool is the user interface to help the user understand the state of the system being monitored. Our aim in developing a visualization feature for the implemented framework was to make it usable and

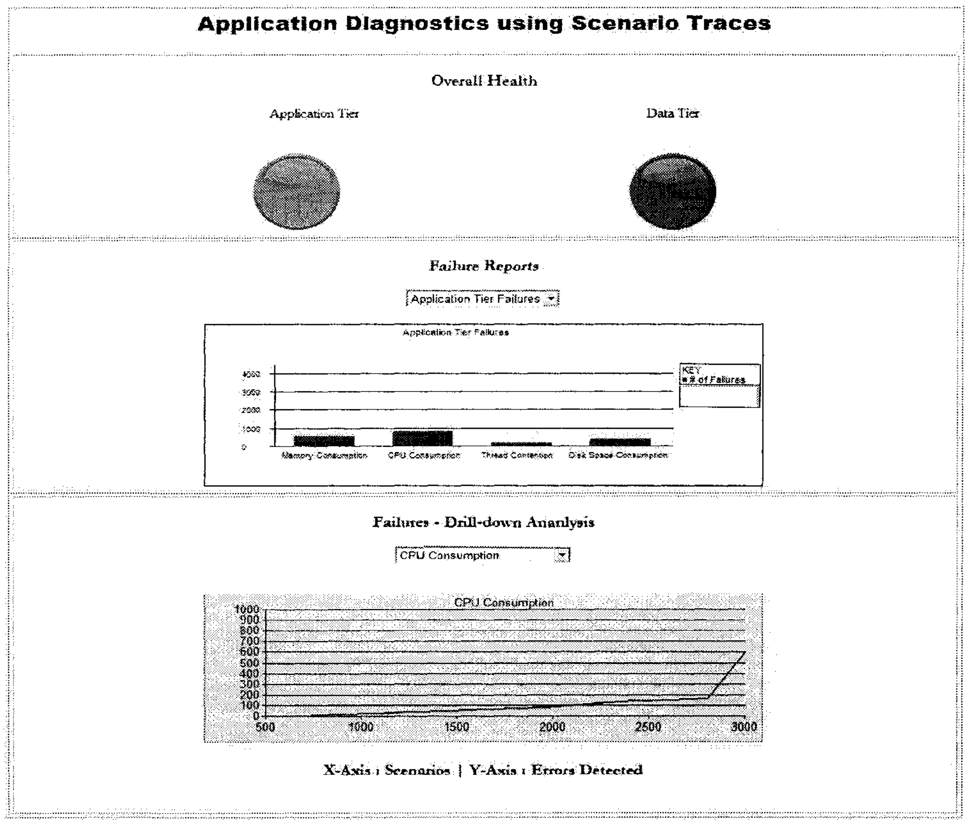

Figure 3.6: Summarized Analysis

allow enough flexibility to explore the results.

The first design decision while developing the user interface was to make it web-based. This minimizes the need to install it on every user machine. It maintains a central repository from which data can be accessed from anywhere on the network.

The dashboards and other analytical features are laid out on two separate web pages. The first page, as shown in Figure 3.6, contains a summary of the statistics gathered from the data-set selected by the user. This dashboard page consists of three important statistics arranged in a drill down fashion. The first pane on the page indicates the overall health of the system with a green or red symbol under the

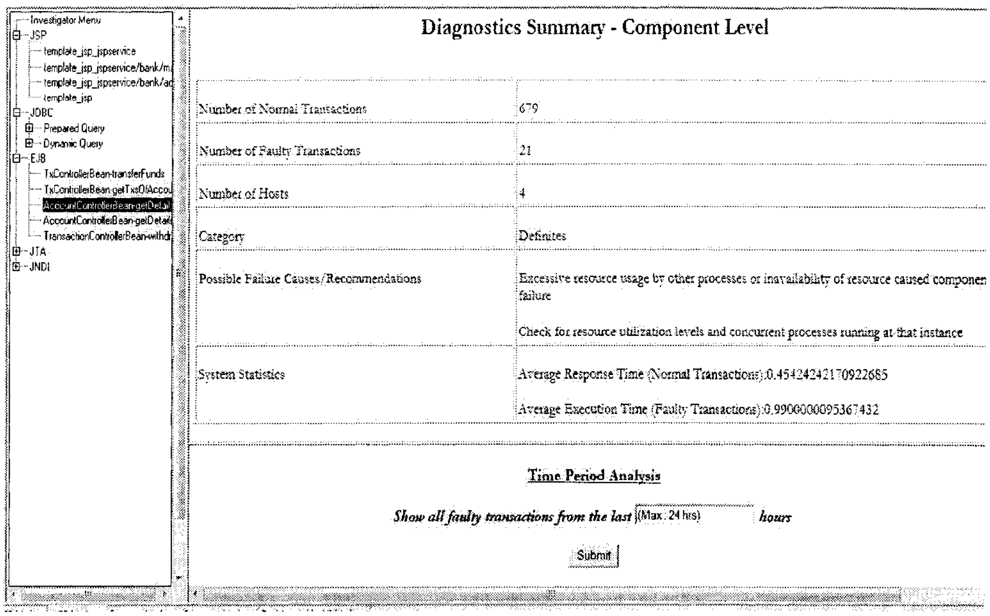

Figure 3.7: Investigator View - Detailed component statistics

labels-Application Tier and Data Tier. Green indicates that the number of Application Tier errors are well within the threshold levels while red indicates that this threshold is breached. We have assumed an error threshold of 15% of the number of scenarios passing though the system. For example, if the number of application tier errors detected are more than 15% of the number of scenarios that passed through the system, the health indicator under the Application Tier label would turn red.

The second pane summarizes the number of individual errors identified across each of the error types modeled in the framework. The third pane further drills down into the individual error categories and displays the number of errors detected over time. This dashboard helps the analyst in understanding the trends in errors within the system and identifying the times when the possibility of errors is higher.

The second page, also dubbed as the Investigator page, allows the analyst to explore the data across time and study the behavior of the traces. This page as illustrated in Figure 3.7 and Figure 3.8 has a tree view of the entire set of components

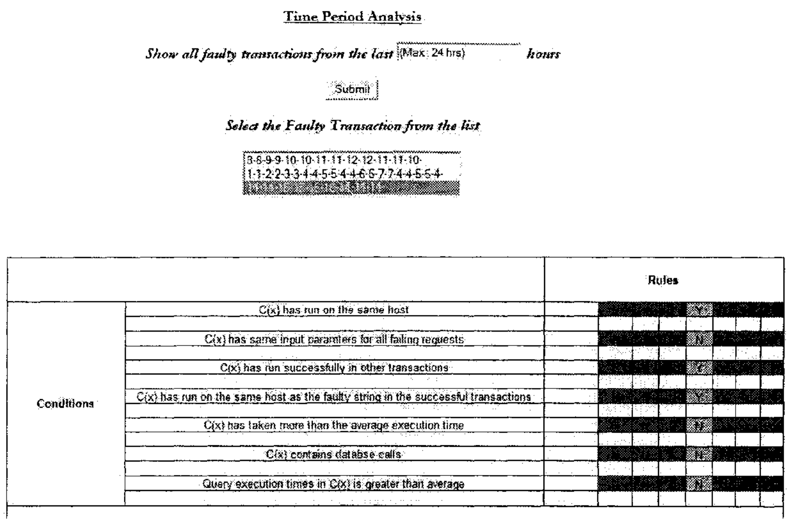

Figure 3.8: Investigator View - Decision Table logic

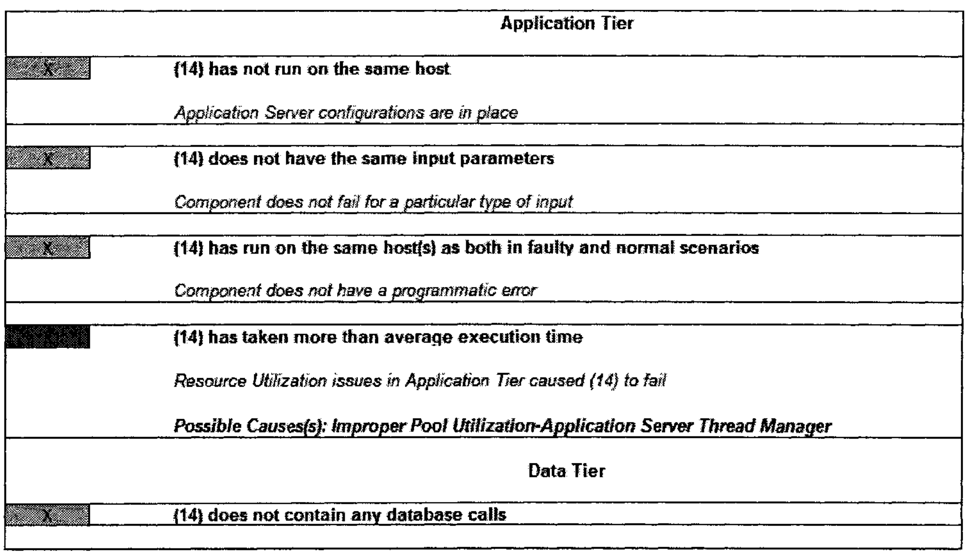

Figure 3.9: Ruling out problem causes for the selected faulty component (14)

instrumented by the framework, and each component can be individually selected to view the corresponding details. While summarized statistics of the component selected is displayed at the top of the page, the bottom half is used to choose the time for which the decision table logic is to be applied on the data-set.

By varying the time interval and looking at the results, we would be able to better judge the cause of the problem. For example, a failing database connection problem perceived in one time-interval could be a server configuration problem that had not been noticed until checking for earlier time periods. Allowing the user to process data over different time periods might help in catching these errors. Figure 3.9 shows the rule checking process on the selected scenario trace and narrows it down to the section of the system perceived to be having a problem with respect to the trace. This page displays the results of the rules applied to the trace data and any interesting statistics derived.

3.7 Summary

This chapter was structured to detail the diagnostic framework that forms the core of the thesis. The first half of the chapter concentrated on the issues for setting up the instrumentation to record relevant data, and the structure and format of the log data to be used by the framework. The second half details the diagnostic framework itself with the various stages of processing, such as the data cleaning, data processing, and feeding the processed data to the primary and secondary decision tables to perform relevant checks. An illustrative example that walks through the primary and secondary decision tables follows the discussion on the framework design. Finally, we end the chapter with the visualization options that lets analysts reveal the nature of the problems and the corrective actions suggested by the framework

Chapter 4

Evaluation

This chapter focuses on the evaluation of the proposed framework. A case study web application is chosen for the evaluation. The objective was to select an application that was built upon J2EE, consisting of all the web components discussed in Section 2.2.1. The ideal open-source application available for this purpose was the Duke's Bank application [4]. Apart from the flexibility the system provides in customizing its components, this application has been widely used as the benchmark application in research involving enterprise web systems, thus adding to the applicability of the tests conducted on this system. The evaluation process starts by laying out a comprehensive test bed consisting of automated load generators and fault injection mechanisms, the details of which are explained in Section 4.2. Section 4.3 briefly summarizes the overall results obtained in the evaluation process and the inferences gathered around the collected data. The chapter concludes by comparing the framework with an open-source diagnostic tool and analyzes the advantages and improvements the framework has over that tool.

4.1 Experimental system setup and configuration

The evaluation of the diagnostic framework was performed on an open-source, distributed, web-banking application called the Duke's Bank application. The system handles most of the use cases for a normal online banking system. The set of high-level use cases that we used for evaluation purposes are outlined in Appendix B.

4.1.1 Design Details : Duke's Bank Application

The Duke's Bank application consists of two different types of clients, one a standalone client that is used for managing accounts and customers, and another that is web-based and enables the user to perform online activities. The users access the information maintained in a database through enterprise beans. The objective of the Duke's bank application was mainly to illustrate the functioning of the various components present in the Web Tier, Business Tier, and the EIS Tier. Figure 4.1 demonstrates the high level architectural composition of the Duke's Bank application.

As in every J2EE application, the entire Business Tier logic is encapsulated within the enterprise beans. For a deeper understanding of the EJB functionality, Figure 4.2 describes the interaction between the session beans and the entity beans and how the web client interacts with the underlying database tables through the session beans.

Within each of the beans, the functional logic is coded into methods that correspond to different actions such as withdrawal, transfer, and viewing account balances.

Session Beans

The Duke's Bank Application has three session beans: *AccountControllerEJB*, *CustomerControllerEJB*, and *TxControllerEJB*. These beans form the core of the business logic implemented in the application. They also serve to abstract the underlying server-side logic and database accesses from the user.

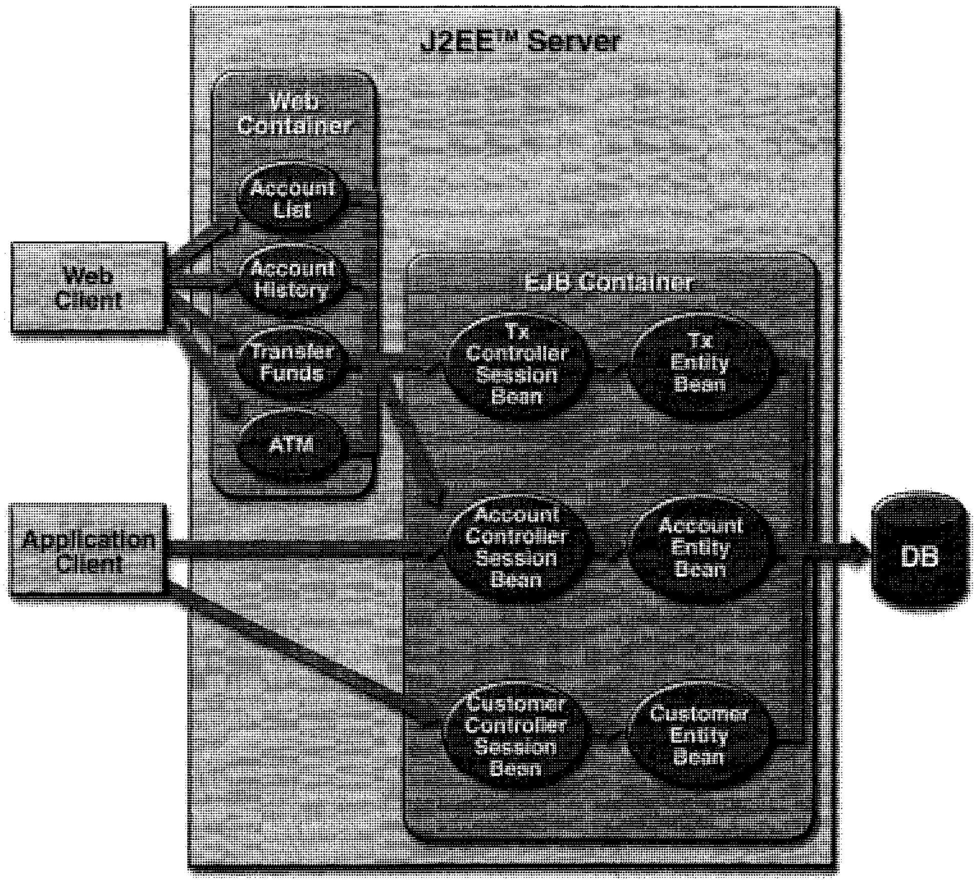

Figure 4.1: Indicates the various components present in the Duke's Bank Application and the arrows indicate the interaction and the flow of logic through the components when scenarios occur [4]

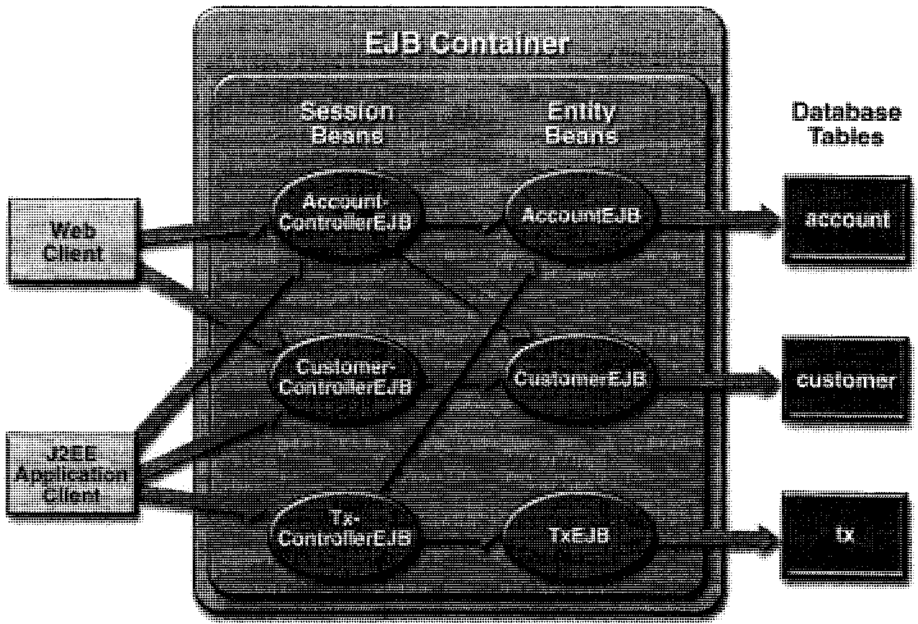

Figure 4.2: Session and Entity bean setup in the application layer [4]

Entity Beans

Each business entity represented in our simple bank, the Duke's Bank application has a matching entity bean:

- *AccountEJB*
- *CustomerEJB*
- *TxEJB*

These entity beans form the means to access the database tables outlined in Figure 4.3. Based on the table to be queried the entity bean has an instance variable that performs the necessary database operations. In other words, these entity beans contain the SQL statements necessary to work on the data present in the underlying tables. For example, the *create()* method of the *CustomerEJB* entity bean calls the SQL INSERT command.

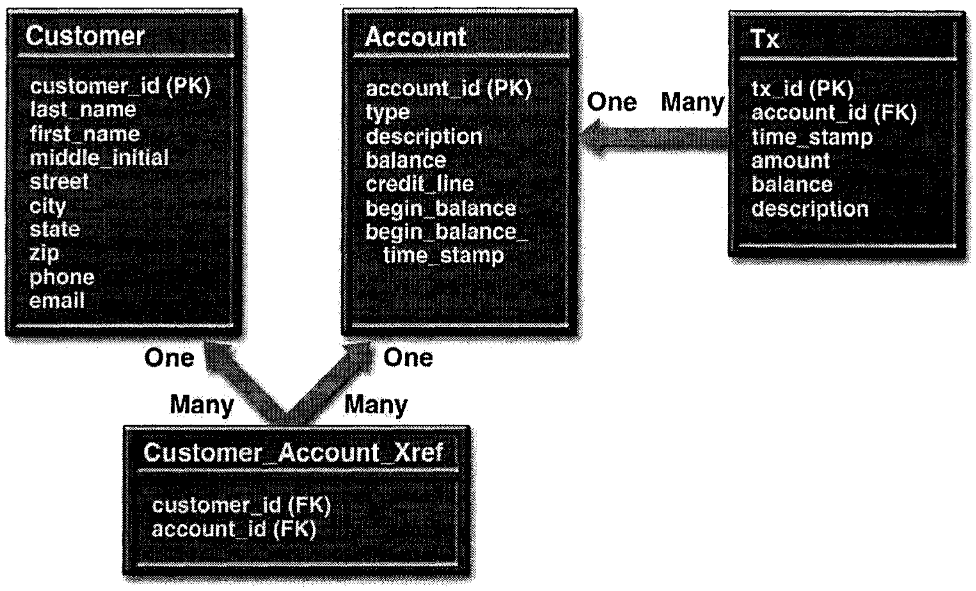

Figure 4.3: Database Entity Relationship Diagram [4]

Database Setup

Figure 4.3 shows relationships between the database tables. The *customer* and *account* tables have a many-to-many relationship: A customer may have several bank accounts, and each account may be owned by more than one customer. This many-to-many relationship is implemented by the cross-reference table named *customer-account-xref*. The *Account* and *Tx* tables have a one-to-many relationship.

Web Client

Most of the interactions in the Duke's Bank application happen through the web client. To look at a simple use case, when a user tries to access his account history through the web client, the application initially calls the *login.jsp* page for the user to first authenticate him on the system and then transfers to the *accountHist.jsp* page, once the EJB *AccountHistoryBean* retrieves the account history of the previous transaction made by the user over a period of time. Figure 4.4 shows a sample screen-shot of the *accountHist.jsp* after a user logs in and tries to view his account

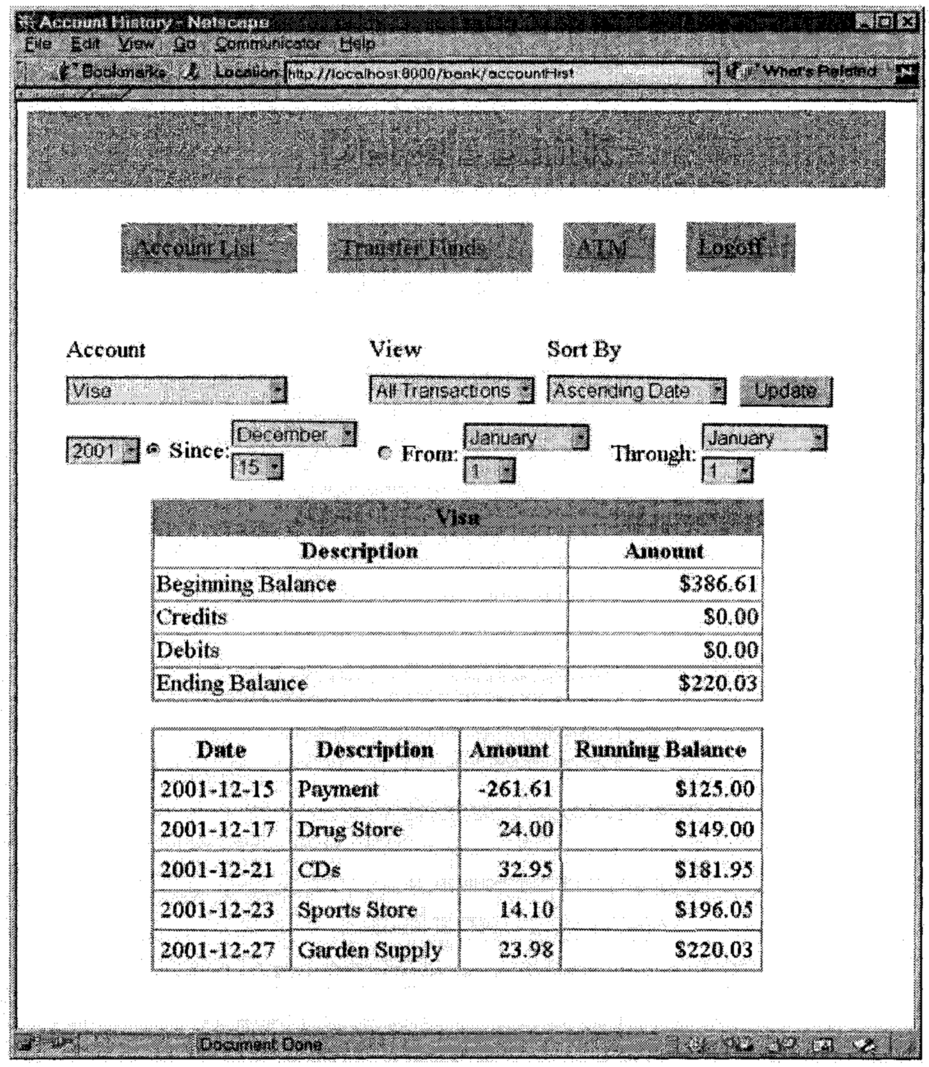

Figure 4.4: Login Page : login.jsp

history.

4.1.2 Environment settings

The setup of the case study mimics a distributed environment consisting of three systems running different configurations of Windows and Linux. Table 4.1 details the exact configurations of these boxes on which the application middleware and database servers were set.

The application code was customized to suit the environment and deployed on

System Name	OS Version	Browser Version	IP	RAM
SIMONETTE	X86-based PC running Microsoft Windows 2000 Professional 5.0.2195	IE 6	129.128.23.65	1023 MB
SPUTINOW	X86-based PC running Microsoft Windows XP Professional 5.1.2600	IE 7	129.128.23.127	1023 MB
STETTLER	i686 running Linux 2.6.9-55.0.6.ELsmp Scientific Linux SL release 4.5 (Beryllium)	Firefox 1.5	129.128.23.63	1023 MB

Table 4.1: Evaluation environment configuration details

JBoss 4.3.08 GA release instances running on the three systems concurrently. The changes required for the application code and configuration parameters detailed in the deployment descriptors were effected using NetBeans IDE 5.0.

4.2 Error Injection in Trace Data

Much emphasis was given to the types of errors that could manifest themselves in a web application, as detailed in Section 3.2. It is highly important that the evaluation too includes a comprehensive set of error conditions outlined in the fault model to effectively analyze the usefulness of the diagnostic framework.

In this regard we have considered a fault injection mechanism for our case study that consists of a set of faults to be injected at the application and data tier levels. The web traffic is taken care of by a load generator that generates the necessary load through the web application to exactly mimic normal traffic happening across a real time system. A typical log data file collected over a day's traffic through the system would essentially contain thousands of scenarios logged in it in addition to a sporadic injection of errors in many of these scenarios. The effect of these errors in the system is then studied with the help of the proposed diagnostic framework.

4.2.1 Types of Injected Faults

The generated faults not only affect the deployed application and the way the user experiences it, but also have a significant deteriorating effect on the infrastructure or the underlying system. The following is the detailed list of injected errors.

1. **Memory Consumption** - This type of error is injected by controlling the amount of memory allocated to the JVM of the JBoss application servers running on each of the host machines. Occasionally, one or all of the JVMs are made to consume maximum memory for a given scenario and the resulting errors are recorded in the log file.

2. **CPU Consumption** - This type of error concentrates on the processing cycles for each of the hosts. Occasionally, a heavy process is added and the effects on the application running at that time is recorded.

3. **Thread Contention** - Spurious threads are created while processing some of the scenarios thus trying to impact the functioning of the other threads processed in parallel.

4. **Disk space consumption** - Although this type of error can effectively be identified by current state monitoring mechanisms, the behavior of scenarios when such conditions happen is interesting to know for our framework. This type of error is injected by intentionally blocking out the disk space available.

5. **I/O Consumption** - This type of error is injected in the database server by intentionally making a database partition fail thus increasing the I/O rate in the remaining database disks.

6. **Database connection consumption** - This type of error is injected by reducing the number of connections returned back to the pool. Usually when the pool size is reduced the load generator floods the application server with a large set of requests, more than the database pool can handle.

7. **Database/Application Server deadlocks** - Deadlocks are situations in which one contending request holds onto a resource required by another equally contending request, while the later holds a resource required by the former.

Both the requests cannot progress further until one of them releases the resources they hold. This property is applicable to both application servers and databases. Table-lock kind of deadlocks, where locks are granted for a fixed period of time to allocate enough time for simultaneous requests to take action with not being able to write to the tables are in addition to thread-related deadlocks in databases, while in application servers it is more at the thread level.

While many of the above mentioned errors such as thread pooling, database table lock etc. are embedded in the banking application source code, other types of faults are injected based on the underlying principles of a fault injecting mechanism called JAFL [26].

4.2.2 Load Generator

Generating ample load as necessary forms an important part of the evaluation. While there are many prominent shareware and freeware load generators such as Apache JMeter [27], Doctor [28] , Grinder [29] and so on, we settled for a simple yet powerful record and playback load generator called ***Sahi***.

Sahi [5] is a rich JavaScript-based load generating mechanism that works using proxy technology. Sahi needs to be configured as a proxy server and the web browser needs to access Sahi as a proxy. This allows the Sahi tool to work on the components in the application without having to understand the source code of the underlying application. Sahi acts as a wrapper on the application source code.

The list of use cases outlined in Appendix B were recorded as Sahi scripts and used for generating the necessary load. Another useful feature with Sahi is the ability to run the scripts within batch files. Using this feature we randomly generated

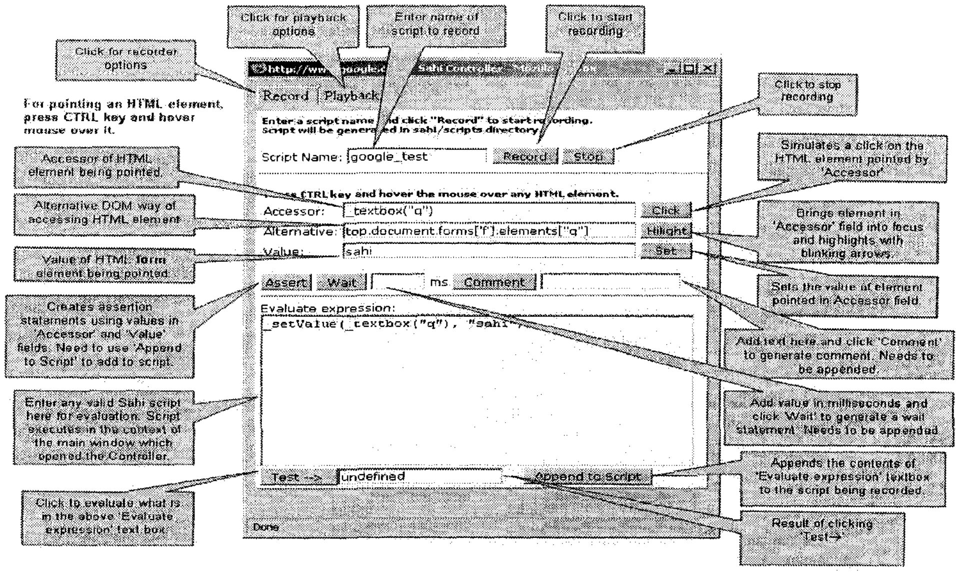

Figure 4.5: Snapshot of Sahi automation tool [5]

tens of thousands of use case based requests to be fed to the application. Figure 4.6 shows a sample of the script file corresponding to one use case. The script peforms the automatic playback of a user logging in, navigating to his account-details page, viewing the transactions history sorted based on the description and then logging off. Each of the batch script contains millions of lines of such code spanning multiple use cases.

Usually Sahi is run on all three host machines simultaneously to maximize the traffic flowing through the distributed setup. Sufficient delays are also embedded between each requests to mimic the normal flow of user requests and are altered to flood the system occasionally based on the type of faults injected. Figure 4.5 is the snapshot of the Sahi dialog box that executes the recorded scripts

```
_setValue(_textbox("j_username"), "200");
_setValue(_password("j_password"), "j2ee");
_click(_submit("Submit"));
_click(_link("Account List"));
_click(_link("Checking"));
_setSelected(_select("sortOption"), "Description");
_click(_submit("Update"));
_click(_link("Logoff"));
_click(_link("Logon"));
```

Figure 4.6: Use Case - View Account Details (Sahi script)

4.3 Problem Diagnostics Testing Results

With the infrastructure in place, the next set of steps are to run random samples of load on the application setup and inject faults on a periodic basis. We conducted multiple experiments for typical work-day traffic flowing through the system. Three separate load generator scripts were produced with varying load effects and run on each of the systems. These batch scripts were written in a way so as to mimic a distributed setup in the sense that the user requests coming out of each of these scripts were targeted not only for the application servers running on the parent machine but also on the other two machines on the network. The traffic was made to flow randomly across the three machines to maintain a sufficient balance in load with occasional bursts of excessive load when faults were injected.

On an average, the framework was able to recall 95% of the errors injected into the application. The detailed results are outlined in Table 4.2 and summarized in Figure 4.7. The framework is more successful in identifying application server problems whereas it has a less recall for the database errors. We also include the probables in the recall percentages as our framework was capable of marking those scenarios as probable faulty ones.

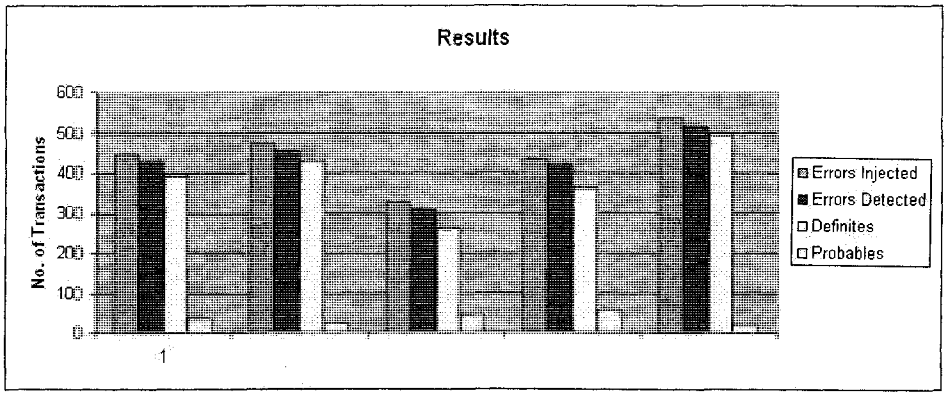

Figure 4.7: Recall rate of errors detected by the framework for 5 tries)

The remaining 5% of the errors are normal scenarios as perceived by the framework (i.e., the framework does not recognize the deviation in pattern for these scenarios). The precision percentage of the framework in correctly attributing the exact cause to the injected problem averages at 93%. On analyzing the reason for missing out on the exact cause of a detected problem, we noticed that on an average around 3% of the missed errors were due to wrongly identified cause (identified from the secondary decision table), but with the correct effect (identified from the primary decision table).

In calculating the thresholds for the CPU consumption, memory consumption and I/O rate we use the logic of setting up the values based on the global 98 percentile band as described in Section 3.5.1. For the evaluation system, these values were set at exactly the same ones as the examples mentioned in Section 3.5.1.

We also measured the time taken to perform the analysis over collected log data. We used a Windows machine running a 2.4 GHz Pentium 4 processor and 1GB RAM to record the performance of our modules. We took an average over 5 runs on the same log file containing 4036 scenarios. The overall time taken to

Scenarios	Errors Injected	Errors Detected	Definites	Probables	Recall Percentage (Definites+Probables)	Precision	Precision percentage
4481	448	430	394	36	95.98	402	93.48
5905	472	453	430	23	95.97	421	92.71
4036	323	307	261	46	95.04	286	93.15
4308	431	418	362	56	96.98	388	92.82
5945	535	514	495	19	96.07	481	93.57

Table 4.2: Comprehensive Results - Data Table

process the raw log file and perform the analysis was 1055.45 seconds. We also measured the average performance of the individual modules used for processing the raw data, and the analysis. We found that the data processing module took about 795.65 seconds on an average, while the analysis module just took 259.8 seconds on the average. It is clear that the data processing module consumes a major portion of the time taken. This time can be further reduced by bypassing the preprocessing stage, where the raw trace is parsed to extract the necessary information, if the instrumenting mechanism can directly record data in the preprocessed form.

4.3.1 Individually Injected Errors

We also conducted several tests to identify the effectiveness of the framework over each of the individual types of errors. Multiple runs were done on the system, upon the same infrastructure as used for the previous evaluation, but with only one kind of injected error in each run. Table 4.3 and Table.4.4 outline the precision and recall of the framework with respect to injected application tier and data tier errors.

The framework is effective in identifying the application tier errors with an average recall of 97% while the recall rate gets reduced to around 93% when data tier errors are injected individually in the system. The precision percentage on the other hand does not change from the average value of 93% for each of the tiers.

Error Types		Memory Consumption	CPU Consumption	Thread Contention	Disk space Consumption	Deadlock
Scenarios		14526	13716	14448	12897	15108
Errors Injected		1308	1233	1590	1548	1209
Errors Detected		1257	1194	1521	1500	1167
Recall		96.1	96.83	95.66	96.89	96.52
Precision		94.27	93.89	94.57	94.12	93.68

Table 4.3: Summary of Results-Individually injected Application Tier errors

Error Types		I/O Consumption	Database Table Lock	Database Pooling	Deadlock
Scenarios		13614	14637	13110	14142
Errors Injected		1089	1170	1179	1698
Errors Detected		1017	1082	1109	1570
Recall		93.38	92.47	94.06	92.46
Precision		92.77	93.31	92.9	92.45

Table 4.4: Summary of Results-Individually injected Data Tier errors

4.4 Comparisons with an open source diagnostic tool

This section compares our results with that of an open source diagnostic framework–Glassbox [30]. Glassbox is an intuitive diagnostic tool that is popular in industry. One user friendly feature of Glassbox is its capability to monitor the application without having to understand the source code. The agent mechanism works on Aspect Oriented Technology.

Glassbox uses the conventional architecture followed by other diagnostic tools of having agents on each of the machines spread across the distributed environment and transmitting monitoring information to a centralized knowledge-base and diagnosing mechanism, which produces the error conditions and additional statistics about the application being monitored. Another advantage of Glassbox is its capability to diagnose errors at run-time.

Figure 4.8 shows the typical diagnostic report of Glassbox. This single report

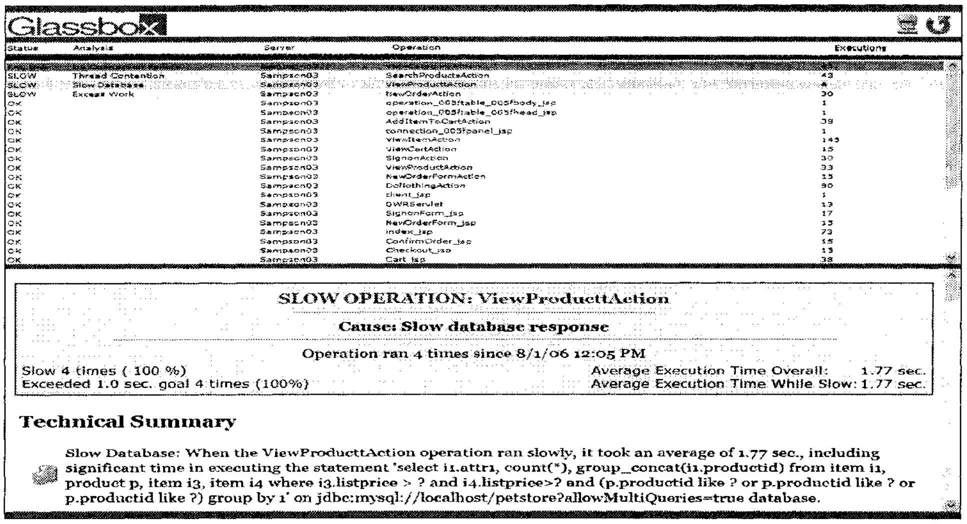

Figure 4.8: List of components by Glassbox

acts as a one stop shop to keep track of the health of the system being monitored.

We found Glassbox to be the most suitable candidate to compare our results with because it is similar in diagnosing failures as that of our framework. This fact helps to bring out the difference between two similar-looking approaches and their advantages and disadvantages in diagnosing a similar set of failures. The following list presents the types of faults diagnosed by Glassbox.

- Slow database query
- Database connection failure
- Broken database query
- Failing Java
- Slow Java
- Bottleneck or thread contention

- Slow or Failing web service
- Inefficient Java code
- Slow or Failing remote procedure call
- Faulty HTTP
- Java Mail Issues
- Broken FTP

In the evaluation study, we plugged in the Glassbox diagnostic mechanism into the same web application setup and recorded the component faults diagnosed by Glassbox. Table 4.5 outlines the number of faults detected by each mechanisms and the types of faults respectively.

The results show that Glassbox always picks up more faults than detected by our framework, with a huge difference in CPU consumption and I/O consumption categories of errors. As far as precision is concerned, Glassbox is marginally lower than our precision rates in errors related to CPU consumption and I/O consumption. Glassbox usually identifies components as faulty if running above a performance threshold of 5% above the historic runs so far. But usually, the user requests traversing through these supposed faulty components would have succeeded without any problems. The lower number detected by our framework is because it picks up only those components that led to a scenario failing prematurely (*definites*) or those deviating away from normal pattern (*probables*). This significantly reduces the number of errors to be looked into while Glassbox produces more false positives. Setting performance thresholds, as in Glassbox, can be a challenging task with the ever changing face of system behavior. Hence diagnostic systems like Wily Introscope [2] and DynaTrace [16] usually come with a statistical threshold

Application Tier	Memory Consumption	CPU Consumption	Thread Contention	Disk space consumption	Total
Errors Injected	36	27	45	31	139
Errors Detected	34	26	43	30	133
Glassbox Detected	N/A	86	45	N/A	
Glassbox Precision	N/A	92.44	97.77	N/A	
Data Tier	**I/O Consumption**	**Database table lock**	**Database Pooling**	**Database deadlock**	**Total**
Errors Injected	49	22	36	90	197
Errors Detected	47	22	34	86	189
Glassbox Detected	89	N/A	36	N/A	
Glassbox Precision	90.89	N/A	97.22	N/A	

Table 4.5: Comparative results between Glassbox and the proposed framework

learning mechanism that varies the threshold levels depending on the changing behavior of the system.

The most distinguishable fact between our framework and Glassbox is that our framework works on the perspective of components performing normally or failing in the course of individual scenarios whereas Glassbox looks at every component as an individual entity without considering any specific user request or scenario. When any error condition appears on one of the components, Glassbox fails to identify the exact cause unless the fault lies directly in the component itself. Usually however, Glassbox emits additional information related to the error that occurred in a single component such as the request URL corresponding to the component being executed. Figure 4.9 shows the typical additional information provided by Glassbox for a component failing due to a database connection issue.

In a few cases this URL might contain the session-id or related request-specific id but most of the times it becomes impossible to make out from the URL information the request that the component belongs to. This is a very important piece of information that our framework records as the same component is called multiple number of times on multiple servers and by multiple user requests and may prove to be a daunting task to identify the system that emitted the error and in what context.

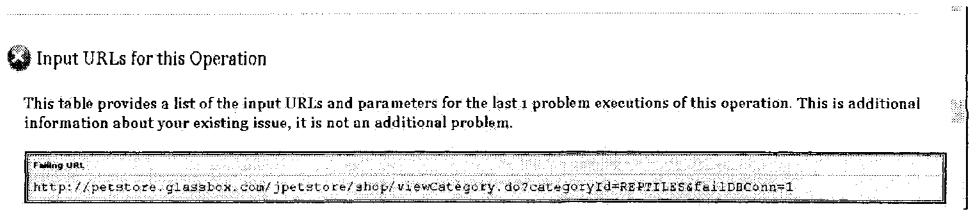

Figure 4.9: Component failure-Additional information and parameters involved

This would have been a simple puzzle to solve if the system happened to be a single server system and not running on distributed servers. In such a case, a session id would suffice in identifying the user request and narrowing down the faulty component and the cause. Identifying a faulty component is not a difficult task if a fault occurs every-time user requests access the component in question. Usually, such errors are transient and we need to collect as much information as possible about the context of a request. Meanwhile, reconciling the context back to the specific user request using the session-ids recorded by Glassbox is a tedious manual process.

On the other hand, Glassbox provides useful run time statistics about the failing components, such as average execution times and classification of processes into slow or normal based on previous execution times. Our framework borrows on this idea of collecting the average response times and other run-time statistics of the components, to help pinpoint a problem. Glassbox error messages clearly state the reason and can also effectively determine the causes in simple cases such as JDBC connection failure, corrupted helper/dispatcher methods, slow or failing Java, etc. A call tree displays the class level and method level information that contains the error, which is helpful.

Glassbox also has a comprehensive knowledge base of error conditions and possible causes of failures to consult before constructing the error report. This

Common Solutions

For Database Connection Failures

1. Correct your operation's connection URL, the Application Server is trying to connect to the wrong IP, Port or Database Name.
2. Increase the size of the DB Connection pool on your application server.
3. Bounce your DB associated with your operation, it is not up or frozen
4. Make your application server close the DB connections when a query completes (note: careful doing this if you have a connection pool).
5. Configure your DB to allow more connections.
6. Fix the network, the App Server cannot ping the database server machine.

Figure 4.10: Common solutions for database connectivity problems by Glassbox

knowledge base is constantly updated for new errors and derives knowledge from frequently identified errors over time. Consequently, Glassbox can output a set of possible causes that corresponds to a fault, but lacks the ability to prioritize the possibilities based on the run-time statistics it collects about to the failing component. Figure 4.10 is a typical set of problem causes outlined in the form of common solutions for database connection problems identified by Glassbox.

This feature may open a large set of possibilities to consider, many of which are not relevant to the situation. It would be time consuming for the analyst to check every possible cause. Experience may help reduce the number of possibilities to check, but relies on much human effort. If only a highly probable and relevant subset of possibilities could be presented to the analyst, the problem could be resolved more quickly. Our diagnostic framework works on these principles of automating the checks with respect to the problem causes and reducing the number of options presented to the analyst. We address a major drawback of Glassbox and move one step further in completely automating the diagnosis process of enterprise level web systems.

4.5 Summary

This chapter evaluated the capabilities of the proposed framework and discussed significant improvements over existing techniques. The main focus of the chapter was to asses the accuracy of the framework in detecting errors. The evaluation study setup with load generator and fault injection was outlined. The evaluation discussed the results, as well as the advantages and limitations of the framework. Finally, the chapter compared the results with that of an open source diagnostic tool called Glassbox.

Chapter 5

Conclusion and Future Work

Changing business trends have fueled the growth of web-based enterprise systems. With the increasing complexity of web frameworks, the complexity in managing such systems has increased. With systems composed of distributed entities, a small configuration change in one section of the system may ripple across multiple layers and manifest itself as a major fault in a different section altogether. Trying to identify the root cause of the detected problem is still a challenging and daunting task and corresponds to what can metaphorically be stated as "searching for a needle in a haystack".

Application and infrastructure management still remains a key issue in the enterprise systems arena with a major portion of the maintenance and support budget being spent on it. Although there has been a significant increase in the level of proactive problem determination and elimination, these mechanisms are still weaker than the judgement provided by the experienced systems analyst or operator. Much diagnostic knowledge still remains in the brain of the human operator, with error detecting tools only complementing the operator's actions.

Looking into the future, error detectors and diagnostic tools should become much more intelligent than they are and help to automatically discover the root cause of a problem quickly.

Our research had the goal of assisting and automating the problem diagnosis and root cause determination process. In this direction, we designed and developed a diagnostic framework that uses a powerful and intuitive rule processing technique known as decision tables.

The literature survey in this thesis compared a wide variety of problem detection tools and other supporting mechanisms, outlining the main issues and limitations. Many features of our framework were inspired from the functionality present in these tools. The literature survey also briefly explored the different kinds of errors that can possibly manifest themselves in web-based systems. Performance-related and infrastructure-based problems, which happen to be highly transient in nature, are particularly challenging. The next-generation diagnostic tools need to tackle these problems effectively, which formed the focus of our framework. We described the problems of interest in the fault model and directed the decision logic accordingly. We have also made the diagnostic framework as flexible as possible to be easily expanded to detect more kinds of errors. Another advantage is that the framework is domain independent.

To demonstrate the feasibility of our framework we deployed a web-based system and tried to mimic a realistic distributed infrastructure. For evaluation purposes, we tested our framework against a random set of failures injected in the deployed web-based system, and analyzed the effectiveness of the framework in detecting the errors and revealing the causes. We compared the results with a highly regarded open-source diagnostic mechanism called Glassbox. The results we obtained from our evaluation study were promising with regards to the accuracy of the framework in revealing the causes quickly.

Future Work

The diagnostic framework was built to handle a defined set of errors in the fault model, considered reflective of the common and most disruptive faults usually occurring in a web-based system. We still would like to extend the framework to capture and diagnose errors that are not confined to just the Application Tier and the Data Tier. Errors occurring in the Presentation layer and those from infrastructure components such as load balancers and caches are also important. Since the root cause may lie somewhere before in the flow of component calls, even before the request reaches the Application Tier, the capability to identify such errors is definitely a future research direction.

The evaluation of the framework does have some limitations. It was difficult to address non-injected errors that would creep into any system accepting high volumes of traffic. Although our framework identified a number of such errors, without enough supporting evidence to validate the claims, it was not possible to document the findings. An area of future work would be to extend the framework to support web applications in real-time with all the supporting mechanisms in place.

The visualization options provided by the framework are not as comprehensive as other commercial diagnostic applications. It will be useful to add more analytical features such as temporal trend analyzers for every component and other useful statistical dashboards.

Finally, most of the diagnostic tools in the market are not able to perform online processing of traces due to the large volume of data that comes bundled with each of these traces. Even a very few that collect trace related information are not capable of performing analysis in real-time. This impairs the ability to reason about error conditions as and when a problem is perceived. Since our framework uses only a subset of information usually collected by trace analyzers, our aim is to add online processing to further reduce the wait time in resolving an error.

Bibliography

[1] S. Pertet and P. Narasimhan. Causes of Failure in Web Applications, accessed december. `http://www.pdl.cs.cmu.edu/ftp/stray/CMU-PDL-05-109.pdf`, Accessed August 2008.

[2] Wily Introscope. `http://www.wilytech.com/solutions/products/Introscope.html`, Accessed August 2008.

[3] G. Waddington, N. Roy, and C. Schmidt. Dynamic Analysis and Profiling of Multi-threaded Systems. `http://www.cs.wustl.edu/~schmidt/PDF/DSISChapterWaddington.pdf`, Accessed August 2008.

[4] Duke's Bank Application. `http://docs.jboss.org/jbossas/getting\_started/v4/html/dukesbank.html`, Accessed August 2008.

[5] Sahi Load Generator. `http://sahi.co.in/w/`, Accessed June 2008.

[6] L. Earl. An Overview of Organisational and Technological Change in the Private Sector, 1998-2000. *Ottawa: Statistics Canada Catalogue*, 2002.

[7] J. Koskinen. Software Maintenance Costs. `http://users.jyu.fi/~koskinen/smcosts.htm`, Accessed August 2008.

[8] Wikipedia. Diagnostics in Medicine. `http://en.wikipedia.org/wiki/Diagnosis`, Accessed August 2008.

[9] High Availability Q&A: Failures, Standards and Metrics. `http://www.gartner.com/webletter/ibmglobal/edition2/article5/article5.html`, Accessed May 2008.

[10] I. Singh. *Designing Enterprise Applications with the J2EE Platform*. Addison-Wesley Professional, 2002.

[11] J. Hunter and W. Crawford. *Java Servlet Programming*. O'Reilly Media, Inc., 2001.

[12] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly Media, Inc.

[13] R. Sharma, B. Stearns, and T. Ng. *J2EE Connector Architecture and Enterprise Application Integration*. Addison-Wesley Professional, 2001.

[14] G. Hamilton, R. Cattell, and M. Fisher. *JDBC Database Access with Java: A Tutorial and Annotated Reference*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[15] D. Bridgewater. Standardize messages with the Common Base Event model. *IBM DeveloperWorks*, 2004.

[16] DynaTrace Software. `http://www.dynatrace.com/en/Product.aspx`, Accessed August 2008.

[17] BMC Appsight. `http://www.bmc.com/products/proddocview`, Accessed July 2008.

[18] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. *Proceedings of the 23rd International Conference on Software Engineering*, pages 339–348, 2001.

[19] C. Yuan, N. Lao, J.R. Wen, J. Li, Z. Zhang, Y.M. Wang, and W.Y. Ma. Automated known problem diagnosis with event traces. *Proceedings of the 2006 EuroSys conference*, pages 375–388, 2006.

[20] E. Fratkin E. Fox A. Brewer E. Chen, M. Y. Kiciman. Pinpoint: problem determination in large, dynamic Internet services. *Dependable Systems and Networks, 2002. Proceedings. International Conference on*, pages 595–604, 2002.

[21] M. Chen, A.X. Zheng, J. Lloyd, M.I. Jordan, and E. Brewer. Failure diagnosis using decision trees. *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 36–43, 2004.

[22] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: real-time modelling and performance-aware systems. *9th Workshop on Hot Topics in Operating Systems, Lihue, Hawaii, May*, 2003.

[23] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J.S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation-Volume 6*, pages 16–16, 2004.

[24] M. Gilleland. Levenshtein distance, in three flavors. `http://www.merriampark.com/ld.htm`, Accessed June 2008.

[25] A. Mos. *A Framework for Adaptive Monitoring and Performance Management of Component-Based Enterprise Applications*. PhD thesis, Dublin City University, Ireland, 2004.

[26] E. Courses and T. Surveys. Comparing Error Detection Techniques for Web Applications: An Experimental Study. *Seventh IEEE International Symposium on Network Computing and Applications, 2008. NCA'08.*, pages 144–151, 2008.

[27] K. Hansen. Load Testing your Applications with Apache JMeter, accessed december. `http://javaboutique.internet.com/tutorials/JMeter`, Accessed August 2008.

[28] S. Han, K.G. Shin, and H.A. Rosenberg. DOCTOR: An IntegrateQ SQftware Fault InjeCTiOn EnviFJonment for Distributed Real-time Systems, accessed may 2008.

[29] Grinder. `http://grinder.sourceforge.net/`, Accessed May 2008.

[30] Glassbox. `http://www.glassbox.com`, Accessed August 2008.

Appendix A

Raw Trace Log Format


```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<TransactionTracerSession User="admin"
    agentIdRef="UUID-1084477462-965986-1930759028-1763333143"
    threadIdRef="834896968"
        EndDate="2008-02-28T18:01:32.843-07:00"
            StartDate="2008-02-28T18:01:02.249-07:00"
                Version="0.1" Duration="30594">


<TransactionTrace
    StartDate="2008-02-28T18:01:02.249-07:00"
    Duration="30594" AgentName="WebLogic
        Agent" Host="sputinow" Process="JBoss"
        Domain="SuperDomain"
        EndDate="2008-02-28T18:01:32.843-07:00">


<CalledComponent
    ComponentName="jdbc%pointbase%//localhost%9082/demo"
        MetricPath="Backends|
        jdbc%pointbase%//localhost%9082/demo"
    ComponentType="Backends"
```

```
        RelativeTimestamp="0"
            Duration="30594">
<CalledComponents>


<CalledComponent ComponentName=
"INSERT INTO ACCOUNTHIST
    (ID, TYPE, HANDLE, RECORD) VALUES (?, ?, ...)"
    MetricPath="JDBC|SQL|Prepared|Update|
    INSERT INTO ACCOUNTHIST
    (ID, TYPE, HANDLE, RECORD) VALUES (?, ?, ...)"
    ComponentType="JDBC"
    RelativeTimestamp="0"
    Duration="30594">
<Parameters>
<Parameter Value="Sockets|Client|localhost|Port 9082"
    Name="Socket write"/>
<Parameter Value="INSERT INTO ACCOUNTHIST
    (ID, TYPE, HANDLE, RECORD) VALUES (?, ?, ...)"
        Name="Prepared SQL"/>
</Parameters>
</CalledComponent>
</CalledComponents>
<Parameters>
<Parameter Value="Pooled
    Threads" Name="Thread Group Name"/>
<Parameter Value="executeUpdate"
    Name="Method"/>
<Parameter Value="Normal"
    Name="Trace Type"/>
<Parameter Value="[ACTIVE]
```

```
ExecuteThread: '7'
for queue: 'JBoss.kernel.Default
    (self-tuning)'" Name="Thread Name"/>
        </Parameters>
    </CalledComponent>
</TransactionTrace>
```

Appendix B

Use Cases

The load generator scripts used the following list of use cases to generate web traffic on the Duke's Bank application setup (Tables follow on the next three pages).

Categories	Use Case description	Actions
Account List	View account balances	Log in using the login.jsp page with user id and password Click on the Account List link View balances Log off
	Detailed account balances (Account Type and order)	Log in using the login.jsp page with user id and password Click on the Account List link Choose an account type and click on it Choose the "View" type and "Sort By" type Click Update View detailed balances Log off
	Detailed account balances (Multiple Views)	Log in using the login.jsp page with user id and password Click on the Account List link Choose an account type and right click and open in a new tab Click on the Account List link on the existing page View account balances in both accounts Log off
	Detailed account balances (Change in Time Period)	Log in using the login.jsp page with user id and password Click on the Account List link Choose an account type and click on it Choose the "View" type and "Sort By" type Select the year , since and from dates View account balances for specified period Log off
	Detailed account balances (Forced logoff)	Log in using the login.jsp page with user id and password Click on the Account List link Choose an account type and click on it Choose the "View" type and "Sort By" type Select the year , since and from dates View account balances for specified period Close the window without logging off
	Detailed account balances (Time out)	Log in using the login.jsp page with user id and password Click on the Account List link Choose an account type and click on it Choose the "View" type and "Sort By" type Select the year , since and from dates View account balances for specified period User becomes inactive after reaching the account balances page

Table B.1: Use Cases-Account List

Categories	Use Case description	Actions
Transfer Funds	Simple Transfer	Log in using the login.jsp page with user id and password Click on the Transfer Funds link View the list of accounts and account balances Select two different accounts and transfer $100 from one to the other View the curent balances for each of the accounts on the confirmation page Log off
	Transfer amount in excess of balance	Log in using the login.jsp page with user id and password Click on the Transfer Funds link View the list of accounts and account balances Submit $200 in excess of balance to another account View the insufficent balance message display page Log off
	Transfer between same accounts	Log in using the login.jsp page with user id and password Click on the Transfer Funds link View the list of accounts and account balances Select the same accounts and transfer $100 from one to the other View the error message Log off
	Multiple Transfers (Normal)	Log in using the login.jsp page with user id and password Click on the Transfer Funds link View the list of accounts and account balances Open two windows for multiple transfers Transfer simultaneosly funds between different accounts View balance information for each transfer Log off
	Multiple Transfers (Overlapping accounts)	Log in using the login.jsp page with user id and password Click on the Transfer Funds link View the list of accounts and account balances Open two windows for multiple transfers Transfer funds with atleast two overlapping accounts View balance information for each transfer Log off

Table B.2: Use Cases-Transfer Funds

Categories	Use Case description	Actions
ATM	Amount Withdrawl/Deposit	* Log in using the login.jsp page with user id and password * Click on the ATM link * Select the account to withdraw from or deposit to * Enter the amount and aubmit * View the account balances * Log off
	Multiple withdrawls/deposits	* Log in using the login.jsp page with user id and password * Click on the ATM link * Select two different accounts to perform the withdrawl and deposit actions * View the balances * Log off
	Insufficent balances	* Log in using the login.jsp page with user id and password * Click on the ATM link * Select the account to withdraw from or deposit to * Enter the amount and aubmit * View the error message page * Log off

Table B.3: Use Cases-ATM