Automated Extraction of Behaviour Model of Applications

by

Tandra Chakraborty

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Highly replicated cloud applications are deployed only when they are deemed to be functional. That is, they generally perform their task and their failure rate is relatively low. However, even though failure is rare, it does occur and is very difficult to diagnose. We devise a tool for failure diagnosis which learns the normal behaviour of an application in terms of the statistical properties of variables used throughout its execution, and then monitors it for deviation from these statistical properties. Our study reveals that many variables have unique statistical characteristics that amount to an invariant of the program. Therefore, any significant deviation from these characteristics reflects an abnormal behaviour of the application which may be caused by a program error.

It is difficult to get the invariant from the application's static code analysis alone. For example, the name of a person usually does not include a semicolon; however, an intruder may try to do a SQL injection (which will include a semicolon) through the 'name' field while entering his information and be successful if there is no checking for this case. This scenario can only be captured at runtime and may not be tested by the application developer. The character range of the 'name' variable is one of its statistical properties; by learning this range from the execution of the application it is possible to detect the above described abnormal input. Hence, monitoring the statistics of values taken by the different variables of an application is an effective way to detect anomalies that can help to diagnose the failure of the application.

We build a tool that collects frequent snapshots of the application's heap and build a statistical model solely from the extensional knowledge of the application. The extensional knowledge is only obtainable from runtime data of the application without having any description or explanation of the application's execution flow. The model characterizes the application's normal behaviour. Collecting snapshots in form of memory dumps and

determine the application's behaviour model from them without code instrumentation make our tool applicable in cases where instrumentation is computationally expensive.

Our approach allows a behaviour model to be automatically and efficiently built using the monitoring data alone. We evaluate the utility of our approach by applying it on an e-commerce application and online bidding system, and then derive different statistical properties of variables from their runtime-exhibited values. Our experimental result demonstrates 96% accuracy in the generated statistical model with a maximum 1% performance overhead. This accuracy is measured at the basis of generating less false positive alerts when the application is running without any anomaly. The high accuracy and low performance overhead indicates that our tool can successfully determine the application's normal behaviour without affecting the performance of the application and can be used to monitor it in production time. Moreover, our tool also correctly detected two anomalous condition while monitoring the application with a small amount of injected fault. In addition to anomaly detection, our tool logs all the variables of the application that violates the learned model. The log file can help to diagnose any failure caused by the variables and gives our tool a source-code granularity in fault localization.

Acknowledgements

I would like to take this opportunity to thank my Supervisor, Professor Dr. Paul A.S. Ward for his precious guidance and continuous support throughout this work.

I am grateful to Professor Dr. Bernard Wong and Professor Dr. Derek Rayside for being my thesis readers, whose constructive comments and feedback helped improve the work.

I thank the members of the Shoshin Lab for sharing their thoughts and providing insightful discussions. I also thank Lori Paniak and Ronaldo Garcia, Computing Technology Specialist at University of Waterloo and my family for their support and assistance. My appreciation also goes to the Department of ECE for their technical and administrative assistance.

To my friends and those who had been part of my life, thanks for your inspirations, enthusiasm, friendship and love.

My deepest gratitude goes to my mom, Shipra Chakraborty and my father Nanda Dulal Chakraborty, for their selfless love and inspiration along the whole journey of my life.

Dedication

This thesis is dedicated to my Parents.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The web serves as an essential part of the day-to-day operation of numerous organizations. The majority of businesses, ranging from start-ups to large corporations, depend heavily on the web to attract clients, connect with suppliers, and generate revenue. This dependency requires the constant availability of web services, as the cost of failed online transactions can be substantial, i.e. a single minute of downtime could cost a merchandiser thousands of dollars in lost sales. For example, in 2011, Amazon.com experienced a series of outages during the American Thanksgiving holiday weekend, which cost the vendor an estimated $25,000 per minute of downtime [89]. According to a survey conducted from October to November 2014 [39],

- the annual cost of unplanned application downtime averages $1.25 billion to $2.5 billion for Fortune 1000 companies
- the average cost per hour for an infrastructure failure is $1 million
- the average hourly cost of critical application failure is $500,000 to $1 million.

The real costs of downtime are, however, inestimable, and include lost or disappointed clients, damage to a company's reputation, negative impact on the stock price, and reduced

employee efficiency.

Intensive studies have investigated the root causes of application downtime [17, 36, 63, 89] and various pre-emptive techniques have been adapted to mitigate its impact. For example, rollback recovery techniques reinstate the failing application to a state prior to the failure [16, 55, 58], software rejuvenation cleans up the internal states of the system proactively in order to prevent failure [46, 50], and replication techniques create and coordinate replicas of the applications to ensure their constant availability [2, 15, 61]. Of these techniques, replication is the one most widely used. Replicating a web application across multiple servers can make their services constantly available. Large companies like Facebook, Google, and Amazon build data centers in order to support massive Internet activity by their users. Each data center houses tens to hundreds of servers, and different web applications are replicated and deployed in those servers. Although applications are deployed only when they are fully functional and their failure rate is relatively low, failure still occurs.

In a data center where thousands of instances of an application are running on thousands of servers and one of those instances fails to operate correctly, it is difficult to diagnose the cause of the inconsistency when all the others are running flawlessly. An application fails to serve its purpose correctly when it deviates from its expected normal behaviour. Its behaviour can be interrupted by several factors, such as, software bugs, configuration errors,resource limitations, hardware failures, incorrect access controls, or misconfigured platform parameters [35]. Diagnosing these problems requires the analysis of a problem's characteristics and associated error messages, followed by the investigation of different aspects of the application that could be causing the problem. Although application developers can leverage signal handlers, exceptions, and other platform supports to handle system errors, it is nearly impossible to predict all such failures and create suitable error intimations [49]. As a result, diagnosing these problems requires a great deal of domain expertise, so it is difficult to automate this process.

The automation of problem-diagnosing is generally obtained through behaviour-matching. The behaviour of a program is defined as an activity that has observable effects on its execution [9]. Incorrectly designed or implemented behaviour causes errors in software applications. Previous approaches towards automated fault diagnosis take the description of a program's expected behaviour as input [9, 88, 95], which requires the application developer's manual effort to define expectations. This limitation gives rise to the need for the automated construction of a program's behaviour model. Behaviour models targeting performance diagnosis are built from performance costs, which incorporate resource consumption [8], the execution time of system calls [4], and network throughput [29]. The models are dedicated to performance-problem-related diagnosis and are not applicable to the non-performance-related issues. Request flows are observed in instances of component-based systems [13, 17] to determine more general types of faults. However, their implementation still needs prior knowledge of the system's implementation.

Deriving the normal sequence of an application's state is another effective way to automate the process of generating program behaviour model. The state of an application is the contents of the memory locations, where the application stores data in variables at any given point in the application's execution [64]. Automated determination of an application's state from the application's exposed data at its runtime has been explored by numerous projects [35, 36, 63, 92, 104, 105, 106]. Ding *et al.* described a mechanism of building an application's signature by combining application's states, collected from tracing several system calls. This technique is able to identify the cause of errors in an application by comparing the application's normal signature to the signature at the time fault took place. Although this approach is helpful in determining causes of certain failures in an application, it cannot report abnormal behaviour unless a request for failure diagnosis is issued. Other approaches used state information to detect only configuration error which requires a knowledge base about the symptoms of previously reported problems [63, 104, 105, 106]. Cheng proposed a runtime state model [19] where the states of an

application are determined at runtime using breakpoints in different execution paths of the application. From these breakpoints changes in the values of variables are recorded, and a state model is built based on the changes. Although this model can correctly identify some abnormal behaviours and their causes, setting breakpoints at execution paths for identifying changes in each variable affects the performance of the application. Hence, this method is infeasible for monitoring an application at runtime.

The limitations in the existing methods discussed above prompt the need for a tool that can determine an application's normal behaviour without having prior knowledge of its expected behaviour and without affecting the performance of the application. The tool also needs to report any abnormal behaviour automatically in order to diagnose failures caused by factors that could either have been previously-discovered or are not yet discovered.

We propose a mechanism to learn the behaviour of an application in terms of the statistical modeling of its variables, which can determine the application's normal behaviour without the requirement of having any prior knowledge of the expected behaviour. Every variable can take different values throughout the execution of the application; exhibiting a unique statistical characteristics and deviation from this characteristic can impact the behaviour of the application. For instance, if the *no. of items* variable in a shopping cart takes a value between 0 and 99 throughout the execution of an e-commerce application and then suddenly takes a value of -10, the application will behave abnormally, as no e-commerce application expects a negative number of items. Consequently, the statistics of values taken by different variables used across an application at runtime are the representation of the application's behaviour model. We can take another example here. To determine the behaviour model of a four-way traffic signal system, the *time of the day* can play an important role. Suppose traffic from north to south is busier during the day while the opposite direction is busier in the evening. If one day, the traffic appears busier from north to south in the evening, this indicates abnormal behaviour of that traffic system.

A possible reason for this occurrence might be an accident on that direction of the road. If the day-to-day statistics of the traffic condition with respect to the *time of the day* is recorded, a deviation from the normal behaviour can be discovered automatically. So, in this case, the statistical property of the traffic condition at a particular *time of the day* is a building block of its behaviour model.

Similarly, the statistics of values of variables taken by the application has a significant impact on determining the normal behaviour of that application. In his PhD thesis, Vijayaraghavan has provided a taxonomy of e-commerce risks and failures [103] in which he has pointed out failures in an application due to lack of testing of different ranges of values of variables used across the application. For example, if an online shopping store does not have the ability to check for a negative number of items at checkout, then it can lead to an abnormal status and create inconvenience for sellers. It is possible that the software testing team has not checked this issue, thinking that the application developer must have ensured it, so concentrates instead on more critical issues. Assumptions such as these can have critical impacts. In 2014, two weeks before Christmas, some of Amazons third-party retailers in the UK saw their wares, from PS4 games to beds and mattresses, reduced to just 1 penny each due to an hour-long pricing software glitch [22]. Eagle-eyed shoppers had a field day, but scores of small businesses were left having to absorb heavy losses. This happened due to a third-party application from the Derry-based firm RepricerExpress that helps retailers increase their sales on Amazon by automatically repricing listings faster than their competitors. But rather than earning RepricerExpress subscribers a profit, the software glitch repriced a wide range of items into the virtually free range. This unfortunate incident could have been averted if the abnormal pattern in the pricing of items had been monitored, Under monitoring, it is possible to alert sellers quickly and reduce their losses and inconvenience. In November 2013, a pricing error was exhibited by Reebok's online store. Trainers worth €100 were marked as 'free', with customers being charged for delivery only. Reebok did not honour the orders, but instead refunded the delivery charge and

gave customers 20 percent off their next order. These examples of abnormal behaviour of applications give us incentive to determine the normal characteristics of variables in terms of the statistics that build the behaviour model of the application and determine abnormal behaviour automatically.

We implement a tool to extract the behaviour model of an application from its exposed data at runtime. Our tool collects snapshots of the application at random intervals. A snapshot is a collective memory of the application at a certain period of time, and the memory comprises different variables defined by the application's developers[73]. Our tool calculates different statistical properties of the variables present in each snapshot. A *baseline* model is generated when a significant amount of snapshots is collected and their statistics are calculated. This *baseline* model is then validated with other snapshots collected after the learning phase. The validation results in a validated model that contains the variables, which can contribute in modeling the behaviour of the application. Next, the tool starts a continuous monitoring process which keeps comparing the most recently collected snapshots with the *baseline* model on the basis of the validated model. Should any variable show a variance in its statistics in a significant amount of snapshots, an alert is generated for the user and the variables are recorded with their value in a log file. If the application fails to serve its purpose, the log file can be examined and the cause of failure can be localized in source-code, using the variables recoreded in the log file.

Our method of behaviour model formation does not rely on source-code examinations and does not need any prior knowledge of the structure of the application. We propose our solution to be implemented in a replicated environment, such as data centers where hundreds of servers can run instances of the same application. If even one of these instances exhibits anomalous behaviour, we aim to detect when and why that anomaly occured. The solution is evaluated by applying it to an e-commerce application (online grocery store) [74] and online bidding application (RUBiS[24]) in replicated computing nodes. Both of

these applications are Entity Java Bean applications implemented in Java Enterprise Edition(Java EE) platform. All the data is collected from the memory dump (heap dump of Java Virtual Machine) of these applications during runtime, our tool can successfully extract the behaviour model for both applications at a maximum cost of 1% of the application's performance overhead.

Furthermore, our tool can calculate a statistical model of variables used across an application, so that any deviation from this model is an indication of abnormal behaviour. It is however, worth noting here that an anomaly or the abnormal behaviour of an application may or may not result in failure. To elaborate, suppose a person is sweating profusely. It is possible that he or she is suffering from high blood pressure and needs to see a doctor. However, it is also possible that this person is simply exercising. While sweating is not something that occurs under normal conditions, it does occur in both cases. Similarly, a deviation in an application's behaviour may not lead to failure but it can narrow down the search space required for problem diagnosis. Our study aims to find the behaviour model of an application from the statistics of its variables which can then help to localize the reason for the failure at a source-code granularity.

1.1 Contribution

Our research makes the following contributions:

1. We devise a tool that can detect an application's normal behaviour without any code instrumentation, annotation, or prior specification of its expected behaviour.

2. The tool calculates a statistical model of all variables used across the target application that results an invariant of the application.

3. Our solution builds the behaviour model solely based on the extensional knowledge of the application which is not achievable from the static code analysis alone.

4. Our tool is evaluated on two applications and the experimental result demonstrates 96% accuracy in modelling the application's behaviour with a maximum 1% performance overhead.

5. Our analysis on data collected at runtime is aimed to diagnose general faults of the application rather than being limited to a particular type of fault.

6. Our tool can help to localize the root cause of failure and map it to any problem at the source-code granularity.

Monitoring the behaviour of a software application has been addressed by numerous researche. Most of the existing approaches used the pre-specified expected-behaviour to verify whether the application is running correctly. The verification process also needs to instrument the source-code of the application in order to get data from its all possible execution path. We create a tool that can collect data from the application without accessing the source-code or instrumenting it. Moreover, our tool does not require any knowledge of the expected behaviour of the application. Our tool generates the behaviour model of any application by only analyzing its runtime exposed data.

Our tool learns the normal behaviour of the application in terms of the statistical properties of all variables accessed throughout its execution, and then monitors it for any deviation from these statistical properties. Our study discloses that many variables have unique statistical characteristics that amount to an invariant of the program. As a result, any significant deviation from the invariant reflects an abnormal behaviour of the application that may be caused by a program error. The statistical property varies by the data-type of the variable. For example, for a variable with numeric data-type, the property

will be minimum, maximum, average, and distribution of its values taken at runtime. On the other hand, a string variable will have minimum, maximum, average, and distribution of string length along with the character range as its property. The statistical model of all variables results to the behaviour model of the application.

Our solution does not assume any intensional knowledge of the application. To elaborate, our tool does not require to know possible inputs to different variables used in the application. It builds the behaviour model solely based on the extensional knowledge of the application that is not achievable from its source-code analysis alone. A static code analysis tests the application with a specific range of input which are decided by the application-test-team, whereas our tool is not restricted to pre-defined sample inputs. It analyzes data that is entered by the clients when the application is online. All data are collected from the captured snapshots(heap dump) of the application. We implement a heap reader to read the snapshots of Java applications. We devise a statistical analysis engine which learns the behaviour model of the application in learning phase, validates the model in the validation phase, and then continuously monitors the application for any violation of the learned behaviour model.

We evaluate our tool by applying it on an e-commerce application and an online bidding system. It derives different statistical properties of variables from the application's runtime-exhibited values. Our experimental result demonstrates 96% accuracy in the generated statistical model. The high level of accuracy indicates the success of our tool in deriving the application's normal behaviour. Moreover, our tool also correctly identified two anomalous condition while monitoring the application with a small amount of injected fault. In case of a small replicated system with 10 computing nodes, our monitoring tool may cause maximum 1% performance overhead of the application where this percentage will be relatively low in a large system.

Our work targets to help diagnosing general failures of an application. As we discussed above, we do not assume any knowledge of the behaviour of the application, similarly we do not assume any prior record of failure of the application. Any violation of the statistical model of variables reflects an anomaly of the application which may or may not lead to a failure. Moreover, if the application is crashed, we can still take snapshot of the application server and compare it to the snapshot taken before the crash happened. This comparison may help to diagnose the root cause of the failure. Our tool is designed to detect anomalous conditions which may be caused by a known or completely new program error. Therefore, our tool is not limited to be used only for previously known type of failure.

Our tool logs the variables which violate the learned model in the monitoring phase. The log file contains the variable name and the specific test result which it fails while monitoring the application. When an anomalous condition is caused by a program error, the detail information from the log file can help to localize the cause of the error. The variables are created and accessed inside the application's source-code. As a result, the log file generated by our tool can map the program error caused by the variables, to any problem at source-code granularity.

1.2 Thesis Organization

The remainder of this thesis is structured as follows.

- ***Chapter 2*** contains the introduction of basic terminology and the background for the work along with a review of existing approaches toward behaviour modeling and anomaly detection of software applications . A summary of the limitations of existing approaches concludes the chapter.
- ***Chapter 3*** is a brief overview of the architecture of the solution and explanation of the algorithm implemented for the formation of statistical model.

- ***Chapter 4*** describes the implementation and execution flow of out tool. The execution flow includes Snapshot collection, Filtering Techniques, The learning process, the validation process and the monitoring process.

- ***Chapter 5*** describes the experimental setup and evaluation process of the tool using an e-commerce Application named AffableBean [74] and an online Bidding system called RUBiS [24].

- ***Chapter 6*** summarizes the contribution and limitation of this project and discusses possible future work.

Chapter 2

Background and Related Work

This chapter defines the terminology frequently used in this thesis and also explains existing approaches regarding the behaviour modeling of software systems. The types of data available in a software system along with their collection mechanisms are described with examples. As well, existing approaches related to this work and their limitations are summarized at the end of the chapter.

2.1 Basic Terminology

The terminology used throughout this thesis follows that of Avizienis *et al.* [5] and the thesis of Munwar [73]. For completeness, relevant definitions are reproduced and some of them are redefined in accordance with this project. These definitions are described below.

- A *system* is an entity that interacts with other entities (that is, other systems such as software, humans, the physical environment, etc.). These other entities define the environment of the given system. A system is composed of a set of components put together in order to interact, where each component is another system. This

recursive definition stops when further decomposition is either not possible or not of interest.

- An *application* is a type of software that allows users to perform specific tasks.
- The *behaviour* of an application is defined as an activity that has observable effects in its execution [9].
- The *snapshot* of an application is a collective memory of the application at a certain period of time. The memory is comprised of different variables defined by the application's developers.
- The *service* delivered by an application is its behaviour as perceived by its user(s).
- The *function* of an application is what it is intended to do and is described by the functional specification in terms of functionality and performance.
- A *service failure* is an event that occurs either when the delivered service does not comply with the functional specification, or when the specification do not adequately describe the application's function.
- An *error* is the status of the system that may lead to its subsequent service failure.
- A *fault* is the cause of an error.
- A *model* is a description of some characteristics of the application that can be used to study or predict those characteristics.
- In the context of this thesis, a *behaviour model* of an application is the statistical characteristics of the variables used across the application.
- An *anomaly* is a departure or deviation from the normal or expected characteristics as determined by a model. It is important to note that anomalies do not always reflect

errors or failures in a system. Additionally they may also occur due to normal, albeit uncommon, events such as, a sudden change in user behaviour.

- The *health* of an application is the degree to which its observed behaviour and performance conform to the expected behaviour and performance.
- *Monitoring* is the act of observing a system for the purpose of ensuring that certain properties are maintained. In the context of this work, the purpose of monitoring is to observe the statistical properties of variables used across an application.
- *Diagnosis* is the process of identifying causal factors underlying an observed anomaly. Here the terms 'diagnosis', 'problem determination', 'fault localization', and 'root cause analysis' are used interchangeably.
- The *target* application is the application to be monitored. The terms 'monitored entity' and 'target application' are used interchangeably.
- A *monitoring system* is the entity that monitors the target system. It is often part of a larger managing system, whose role extends to other system management functions.

2.2 Data Collection

The monitoring of a large-scale software system has to deal with the collection of different types of data from the system. These data can be either pulled from the running system or pushed as a notification to the monitoring entity, and then processed for further analysis. It is important to know the types of data to be collected before building the monitoring system. For completeness, a basic introduction to the different types of data and how they are collected from a target system is described in this section.

2.2.1 Type of data

The data collected by the monitoring system can be defined as a variable which measures an attribute or a parameter of the monitored entity. An attribute either represents an instantaneous property of the monitored entity (e.g., free memory size) or an aggregation of the underlying measure over a specified time interval (for example, CPU utilization) [73]. There are two broad types of variables: qualitative and quantitative. Each of these is broken down into two sub-types:

- Qualitative data can be ordinal or nominal
- Quantitative or numeric data can be discrete (often, integer) or continuous

Qualitative data always have a limited number of alternative values, as a result, such variables are also described as discrete. All qualitative data are discrete, while some quantitative data are discrete and some are continuous. For statistical analysis, a qualitative data can be converted into a discrete numeric data by simply counting the different values that appear.

Qualitative data

Qualitative data arise when the observations fall into separate distinct categories [99]. Examples are:

- Colour of eyes: blue, green, brown etc.
- Exam result: pass or fail
- Socio-economic status: low, middle, or high.

Such data are inherently discrete, in that there are a finite number of possible categories into which each observation may fall. This type of data is classified as:

- Nominal or Categorical data where there is no natural order between the categories (e.g., eye colour)
- Ordinal data where an ordering exists between the categories(e.g., exam results, socio-economic status, etc.).

Quantitative Data

Quantitative or numerical data arise when the observations are counts or measurements [99]. The data are said to be discrete if the measurements are integers (e.g., number of people in a household, temperature) and continuous if the measurements can take on any value, usually within some range (e.g., weight, height).

A monitoring system may collect either numeric or categorical data or both of them. For instance, a system's log files contain categorical data. These log files are written when some events are triggered in the system, for example in an e-commerce application when a customer buy a product, it will be an event and get logged as a transaction in the application. However, performance metrics such as CPU utilization and memory usage are numeric data. The type of data to be collected, depends on the level of monitoring required by the system.

2.2.2 Data collection mechanism

Variables are used to record different types of data from the system. These variables may be read and updated either by the monitored entity or the monitoring entity. The monitoring logic or instrumentation that updates these variables is often part of the system structure [73]. In cases where such instrumentation does not exist, it is possible to instrument components of a software system statically or dynamically.

Monitoring interfaces define the mechanism to collect data. For instance, JMX [85] specifies transport protocols, encodings, and mechanisms to collect data from the monitored entity. In general, two mechanisms exist to collect data. A monitoring entity can use polling (pull mechanism) to read the variables when needed, such that, a monitoring entity can send a request(pull mechanism) to the monitored entity to read the size of free memory available at certain periods of time. Alternatively, the monitored entity can send notifications (push mechanism) containing the data to the monitoring entity. So for example, log files can be written or updated by the monitored system and pushed to the monitoring entity.

Various types of data collection mechanisms have been implemented over the decades. Now-a-days most software systems built for general use are component-based distributed information systems and an application is a component of those systems. As our work aims to develop system monitoring at the application level, it is useful to describe how an application relates to other components in a system. A basic introduction to the interaction between the application and other system components is given in the next section. The thesis of Munwar [73] has a very good introduction to these topics; therefore the relevant materials are reproduced here.

2.3 Component-Based Distributed Software Systems

Distributed Software systems for network-based services are typically built using a component-based platform. Examples of standard component-based distributed systems include Java Platform Enterprise Edition (Java EE) [76], .Net [29] etc. Components of the same system are distributed across different computing nodes in these platforms. These initiate the use of middleware that takes care of the remote communication, data exchange, object naming, registration, discovery, object life-cycle management, security, etc. These component-based

software systems are typically organized in multiple tiers. For example, to support an online store a basic system includes a database server for persisting data, an application server providing the execution environment for the application, and a web server comprising an HTTP server and other software to render results of service invocations. Each tier may run in different machines having different Operating System.

2.3.1 The Java Platform, Enterprise Edition

Java EE is one of the most popular platforms to implement distributed, component-based software systems. Java EE specifies application program interfaces (APIs) and interac-

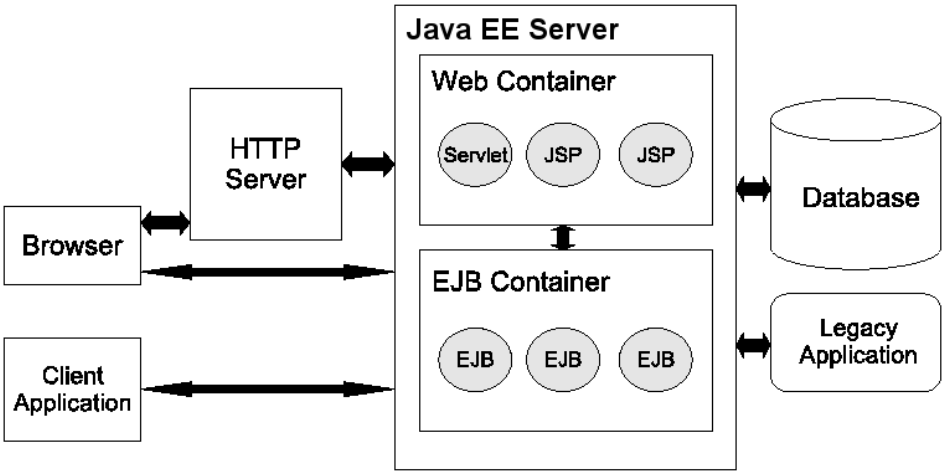

Figure 2.1: Overview of a Java EE-based architecture [73]

tions for basic services needed for distributed and enterprise computing. It also defines interfaces, roles, and deployment details of components in the framework. A simple Java EE-based system is illustrated in Figure 2.1. A Java EE server is a runtime environment for executing Java EE applications. It consists of component containers, which take care of the components lifecycle, thread management, concurrency control, resource pooling, replication, access control, etc. It also implements various common services and libraries.

A Java EE server allows the execution of multiple applications or many instances of the same application concurrently. Many such servers exist on the market, for example,IBM WebSphere, BEA WebLogic, Oracle Application Server, JBoss, Apache Tomcat, Glassfish and Jonas.

A *Java EE application* is a combination of many specialized components. A typical Java EE application can be accessed via its web interface by making HTTP requests, by using native Java calls, or by employing other means such as webservice calls. On the server side, HTTP requests for dynamic content are handled by web components such as Java Servlets or Java Server Pages (JSP), which are managed by a web container. The application logic concerned with the processing of business data is implemented in Enterprise Java Beans (EJBs). These EJBs can be accessed using a remote method invocation (RMI) protocol. The Java EE specification classifies EJBs into three different types. A session bean is a component that acts temporarily on behalf of a client. This component can be stateful (for example, keeping track of a customers shopping cart) or it can be stateless (for example, only computing a formula given some input). An entity bean is an EJB that provides a mapping to persistent data, typically a row in a database table. A message-driven bean allows an application to provide asynchronous functionality. For example, such a component can accept a customer order, adding it to a queue of pending orders; when resources become available, the orders are removed from the queue for processing. Web components and enterprise beans execute in containers, which provide the linkage between components and services and functionality implemented by the underlying runtime. Java EE applications typically require connection to back-end data sources, which may include database servers or legacy systems.

Serving user requests in a typical Java EE-based system entails processing by many components of different types. A typical flow of execution may include the following: a client requests a service through a web page; the request is assigned to a thread at the

server, which executes a Servlet. The Servlet code retrieves a reference to a Session EJB component and executes one of its methods; the Session EJB causes one or more Entity EJBs to either be instantiated or fetched; the data mapped to the Entity EJBs is retrieved by using a connection to the back-end database; once the data is fetched at the session EJB, it is processed, and then returned to a JSP component; in the JSP, the results are put in HTML format and sent to the client. While serving the request, the components involved may utilize common services such as transactions or logging.

2.3.2 Monitoring Infrastructure

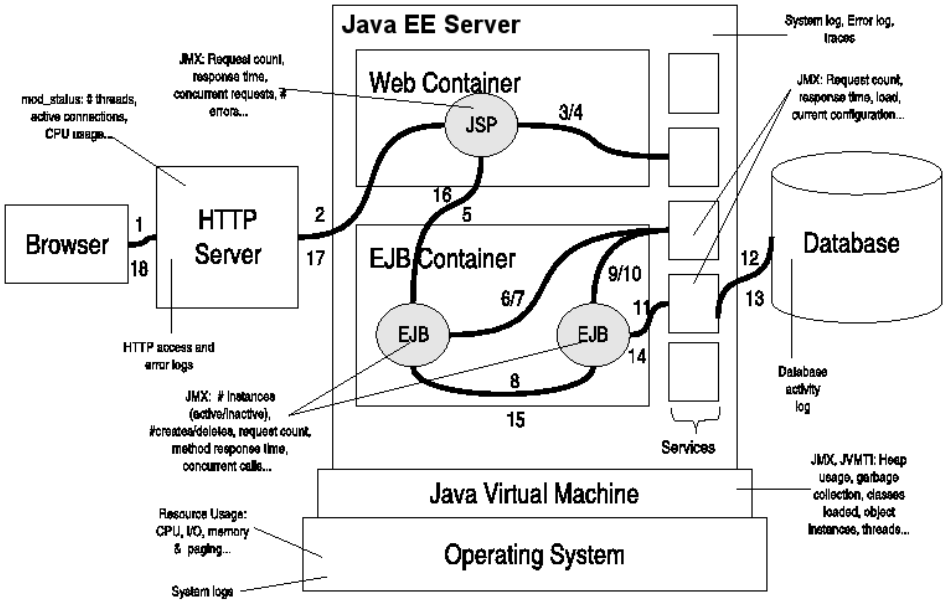

Figure 2.2: Monitoring infrastructure of a Java EE-based system [73]

Software systems expose much data to facilitate their monitoring and management. Each component can be monitored via a multitude of performance metrics and events, each detailing some aspect of its state, behaviour, or performance. Much of the available

data can be accessed through predefined mechanisms such as logging, tracing, or polling of management interfaces. Additional data can be collected on-demand at runtime by instrumenting parts of the system. Monitoring a software system, therefore, entails dealing with potentially large volumes of data. A glimpse of the amount of the data available can be illustrated by considering the monitoring infrastructure of a basic Java EE-based system. Figure 2.2 presents an overview of some important sources of information available from various parts of such a system. Below, the main subsystems are described, the type of data they provide, and how such data can be collected.

Generally a software system requires an operating system to function. When distributed, multiple operating systems support the software system. Most commodity operating systems provide mechanisms and tools to monitor resource usage, user activity, process behaviour, etc. In Unix, for example, metrics are exposed through a virtual file system mounted at /proc. Utilities such ps, vmstat, iostat, and netstat make access to the data even more convenient. Similarly, the Windows Management Instrumentation (WMI) [30] allows for the monitoring of many aspects of a system when using Windows. Besides these conventional monitoring facilities, much more data can be collected via dynamic instrumentation [14, 71, 100] and dynamic insertion of interceptors between components via hotswapping [97]. Software systems commonly rely on runtime environments executing above the operating system layer. These runtimes not only make it possible to develop portable software but also implement features to improve robustness and performance. Examples of these features include sandboxing, automatic memory management and exception handling, runtime code optimization and replacement, etc. Such runtimes include the Java Virtual Machine (JVM) [80] and Microsofts Common Language Runtime (CLR) [27]. A Java EE-based system requires a JVM to execute. The JVM provides different interfaces for monitoring. The JVM Tool Interface (JVMTI) [81] enables debugging as well as profiling of Java applications. A JVM can also be monitored via a standardized management interface, namely the Java Management Extensions (JMX) [85] interface. JMX allows data

related to various aspects of the JVM, including the number and state of threads, memory usage, classes instantiated, and garbage collection to be accessed easily. In addition, it is possible to instrument Java bytecode dynamically at runtime [79]. Monitoring probes that were not considered at design and implementation time can now be retrofitted when the need arises. The availability of runtime bytecode instrumentation in the JVM allows Java applications to take advantage of approaches like dynamic aspect-oriented programming (see, for instance, [40]), whereby monitoring aspects can be added dynamically. This represents another potential source of monitoring data. Most Java EE-based systems require a database management system (DBMS) to manage persistent data. These DBMS expose a rich set of monitoring data to facilitate their tuning and maintenance (see, for example, [26]). Examples of the available data include details on query execution, table activity, application connections, I/O, threads, memory, storage, and locking.

Java EE applications are typically accessed via their web front-end. As such, HTTP servers are the first subsystems to handle user requests. They usually serve static content (for instance, images) directly, but redirect requests for dynamic content to an application server. They may also provide authentication and encryption services. HTTP servers also make state, performance, and error-related data available through log files or monitoring interfaces. An HTTP server usually logs requests received, return codes, execution time, etc. The application server lies at the centre of a Java EE-based system, as it provides the middleware and the runtime environment to execute the application logic. Significant events (for instance, exceptions) which occur during a servers execution are typically logged or sent in the form of notifications to registered listeners.

Most Java EE servers are JMX-enabled [85], which allows a management entity to monitor and manage them. Many subsystems of a Java EE-based system may be shipped with embedded instrumentation that makes more detailed information available on request basis (for example, using the ARM API [56]). Another way to monitor an application built

on Java EE is collecting the memory dump of a running java application. Parsing this memory dump can provide relevant monitoring data of the application. Memory dumps can be read using VisualVM [87], Jconsole [82], Memory Analyzer [38], Jhat [83]. Although memory dump has been used only for detecting memory leak so far[10, 13, 34, 57, 72], it can also be utilized to get overall information of application's behaviour.

A Java EE server is itself organized into multiple subsystems, which include component containers (for instance, web and EJB) and modules for transactions management, database connection management, thread pool and object pool management, etc. Each such subsystem exposes data related to the state, behaviour, and performance of the subsystem. A Java EE application and its components can also make fine-grained monitoring data available. Because of standardization, much monitoring data related to applications is generic (that is, applies to all applications that conform to the Java EE specification). Still, application-specific monitoring can be made available by instrumenting the application. Data on web components, such as Servlets, may comprise the number of requests being served over time or at any time instant, number of errors encountered, response time, etc.

As illustrated above, even a basic Java EE-based system can produce a large amount of monitoring data. A few hundred metrics may be available from the application server and the DBMS for an application such as an online store. Production level Java EE-based systems are generally larger and more complex, comprising clustered web and application servers, replicated databases, load balancers, etc. Effectively monitoring such systems is very challenging. The difficulty lies in using the data generated by these systems to good effect; that is, for quickly detecting errors and failures and for localizing their causes. Furthermore, collecting all this data would not only adversely affect performance, but would create significant overhead for handling the collected data. An important aspect of the challenge is to contain this overhead, while not sacrificing effectiveness of problem

determination.

2.4 Literature Review

Consistent performance and dependability are the two major criteria for a software system [73]. The performance of a system is a measure of how efficiently it can deliver correct service and is evaluated through attributes such as resource utilization, throughput and response time. Dependability is defined as the ability to provide services that can defensibly be trusted within a certain time-period and is evaluated through the following attributes:

- *Availability* - readiness for correct service
- *Reliability* - continuity of correct service
- *Safety* - absence of catastrophic consequences on the user(s) and the environment
- *Integrity* - absence of improper system alteration
- *Maintainability* - ability for a process to undergo modifications and repairs

Confidentiality, which means, the absence of the unauthorized disclosure of information is also used as an attribute of dependability around issues of security. Security is a composite of confidentiality, integrity, and availability, and its guarantee is crucial for most computer systems. This is a challenging and a massive research domain in itself, therefore outside the scope of this thesis. However, it should be pointed out that monitoring a system's behaviour may allow detection of certain forms of security attacks such as denial of service and hacking by unauthorized activity.

Reliability and availability are quantifiable by direct measurements and are significant for most long-running software systems. On the other hand, safety is most relevant in the context of mission-critical systems. The behaviour and performance of safety-critical systems are generally predefined by formal specifications, so meeting these specifications is a must. However, Amazon Web Services(AWS) have used formal specification to verify their real-world systems systems since 2011 [75]. They use PlusCal or TLA+ for writing their specification of the system and then verify it using model checker. They claimed that they found subtle bugs in system which were not possible by other means. So formal specification turns to be an important design requirement for them. However, in our work, we are focusing on application programs rather than system programs. Amazon is using formal specification for distributed systems, they did not mention it for web applications. Web applications might not have clear specifications and their update is more frequent compare to distributes systems. As a result, writing formal specification for large systems is worthwhile but putting manual effort for specifying a web application's behaviour may not be a good investment of time.

A software system should maintain a certain level of performance and dependability. A monitoring system is built to monitor whether or not the target system is maintaining these basic criteria. To perform correctly, software needs to be available and reliable throughout the runtime. In large part, it can achieve reliability and availability if it can always exhibit normal behaviour. As mentioned earlier, deviation from normal behaviour can be a potential cause for system failure and it can also help to diagnose the root cause of failure. The question that now arises is how to perceive the normal behaviour of a software. An effective approach to determining a software's normal behaviour is to compare its runtime behaviour against its design-time specifications. However, specifications are not always well-defined. An alternative approach is to detect the normal behaviour of a software from its exposed data at runtime. Previous work based on failure diagnosis resulting from the abnormal behaviour of a system has been done by extracting state information, event

log analysis, and metric correlation. Some of these approaches require statistical analysis, while others use data mining. Most require prior knowledge of the internal structure of a system but techniques do exist to view the system as a black-box.

2.4.1 State-Based Approach

State has been defined and explained by researchers based on the context of their work. The construction of a finite-state machine was suggested by a number of researchers [8, 18, 21, 25, 68] for model checking, which is a technique for verifying the correctness of a system. Numerous state-based techniques have been applied to construct the behaviour model of a system; these can be used to check for predefined specifications or for fault diagnosis.

Corbett *et al.* [25] developed a model extractor called Bandera, which can generate a program model from java source code. This program model is intended to be an input to existing verification tools for checking the correctness of a program. The Bandera can also map the output of the verification tool back to the source-code if it finds any violation of the specifications. It uses slicing technique to select relevant parameters that should be present in the generated model. This is a static approach that requires user specifications along with internal knowledge of the source-code. However, there is also dynamic approaches in model extraction [68]. While the static approach is complete in that it describes all possible executions of a system, it may still output some invalid behaviours as it cannot see the runtime values of variables. In contrast, the dynamic approach describes behaviour from runtime traces. This approach is not complete as it is nearly impossible to explore all possible execution paths. In the dynamic method, the runtime information is collected by code annotation or instrumentation which may affect the performance of the application. A good comparative analysis on model extraction can be found in [68].

Ernst *et al.* proposed a tool named Daikon [41], a dynamic system to detect likely invariants of a program . The dynamic invariant detector runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. It keeps track of every write of every variable (both locals and fields) which slows the program down. Daikon instruments the program in order to capture values of different variables while executing the program. The generated invariants are very insightful for forming the behavior model of the program but the required instrumentation can be computationally expensive.

Whitaker *et al.* [106] used a Virtual Machine Monitor (VMM) to determine the deviation point from the normal behaviour of a system. They recorded several disk updates of the target application along with a time stamp in a storage called the Time Travel Disk (TTD). The TTD was implemented in a parent Virtual Machine along with an analysis engine. In case of failure they ran the target application as a child virtual machine and the analysis engine issued software probes to find the deviation point from the normal state to faulty state using a binary search algorithm. By utilizing the disk update and time stamp information, their tool was able to find the point at which the failure occurred. It is dedicated to find configuration errors in an application. Because this approach needs to use software probes, which are issued based on failure type, it might not work in case of occurrence of an unknown failure.

Strider [104] is a block-box approach developed by Redmond *et al.*, that can detect the abnormal behaviour of a system due to configuration error. When a failure occurs in Strider, the configuration data are presented as a state-vector which is very large in size. The vector's size is reduced by state-diffing and state-tracing operations. Specifically, they compare the bad state (state after failure) with the good state (state without failure) and find the possible configuration parameters which caused the problem. These parameters are checked against a computer genomic database, which has been formed through the knowl-

edge of troubleshooting previous configuration errors by users. This approach is highly useful for determining configuration errors but it cannot determine any other parameters that may be responsible for abnormal behaviour of a system. Also, because it depends on the troubleshooting database, it is possible that it might not detect a configuration error that was not previously discovered.

Ding *et al.* [35] proposed a tool that can capture the runtime behaviour of an application using system call traces of the application along with signals and environment variables. They record a trace of the application whenever a system call is invoked and suspends the application. These traces are collected from multiple runs of an application and in long runs, they limit the size of the file containing the trace. All of these traces are integrated in a signature bank, so if a failure occurs and a user asks for diagnosis, this signature is compared with the traces collected at the failure time by a classifier and the root cause of the failure is reported from there. However, if a user does not request a diagnosis, the tool cannot detect any deviation from normal behaviour and will continue integrating the traces to the Signature. Moreover, when the signature file becomes too large for the application is limited in size, it is possible to lose important data which might then lead to failure.

2.4.2 Event Log Analysis

Event log processing is done in large organizations (such as those with data centers), where log data generated on a daily basis are many gigabytes in size [47]. Using data mining to discover event patterns from event logs has been the topic of numerous research studies [47, 93, 94, 102]. A log file clustering algorithm was proposed by Risto [102], where a vocabulary of words appears in log files. Using this vocabulary, anomalous log lines can then be detected. However, for large distributed systems, the space complexity would be massive. Furthermore, although log clustering can be automated, human interaction is needed to understand the end result of the log profiler in order to determine any anomalies

in the system.

Abnormal behaviour detection based on a default system log is discussed in a self-similarity- based approach [59], where a self-similarity is derived from a cosine distance between event distributions to find anomalies in a cloud application. The main idea is to observe a systems trend under both normal and erroneous conditions. However, this model cannot be guaranteed to work properly if the access to system log files is restricted. Moreover, errors that were not previously logged may not be recognized using this approach.

A hybrid framework of different statistical models to find anomalies in cloud applications is discussed in [53]. Three techniques, including the Holt-Winters algorithm, the Adaptive Statistical Filtering and the Ensemble Algorithm, are focused on here. Applying a hybrid statistical model requires going through several parameter selection processes, which makes it computationally very expensive and can cause lower performance problems. Moreover, as soon as the system becomes more complex, parameter selection can be extremely time-consuming. Therefore, the applicability of this type of framework in a real system might not be affordable by small cloud business providers.

A framework of problem determination in Internet Services based on the statistical analysis of event logs is discussed by XuYuXu in [107]. This framework detects the anomalous component by message-flow through different system components. In this approach, a message-id is used in all log lines for a message going through different components. Similarly, Chen *et al.* proposed Pinpoint [17], which can trace all client requests through the different components of a system. In case of failure of any request, it performs data clustering and statistical analysis to correlate the failure to the components most likely to have caused them. In these approaches, the system needs to be instrumented to capture the flow of information through all components and detect the faulty one in case of failure.

Correlations between various data collected at multiple points of a distributed system are analyzed to find faults in a complex system [107]. In this approach, log data is collected from multiple points, such as the application server, database server, web server, etc., in order to find the correlation following the Gaussian Mixture model. A fault is detected as an outlier from the probability density boundary. The problem with this approach, however, is that it needs to collect data from different points, which can be quite expensive when the system is largely distributed. Additionally, the collection of data will consume the network, and the diversity in log files from different type of servers is also an issue. In the end, the correlation might not always be found, in which case the fault will not be detected properly.

Research on detecting abnormal behaviour using Complex-Event Processing (CEP) is being done on a large scale. CEP is defined as "a set of methods used for understanding and controlling event-driven information systems" [66]. It correlates events by the way they process massive data. A monitoring system can be designed on a single CEP engine [66, 101], on a distributed architecture [6, 7, 37, 65] of CEP engines, and also on,CEP engines that split CEP queries into sub-queries and execute them on many nodes [1, 11, 33, 91, 98]. A single CEP engine may become a bottleneck if the amount of monitored data exceeds the processing capacities of that engine. In distributed CEP, architectures suffer from the potentially large number of messages exchanged between the engines. Furthermore, CEP engines that are distributed based on rule-splitting can have operator placement problems, which are difficult to solve.

A dynamic architecture for cloud performance monitoring and analysis via CPE(DCEP4CMA) is defined in [69], where a hybrid of centralized and distributed CEP is used. This monitoring system can switch between different CPE architectures based on the underlying cloud environment. It is, in effect, monitoring the environment built on multiple layers, which means it needs to collect data from all layers of the system. However, this can degrade the

performance of a large distributed system.

All of these approaches require some event stream coming from the cloud application as well as some predefined rules to decide whether that event is directing to an abnormal status of the application. Additionally, the rules need to be defined by the domain experts, which might not be possible for a newly deployed system. CEP-based monitoring requires a different computation unit to handle complex events along with domain expertise to decide rules for the analysis. Complex-Event Processing is potentially a good candidate to monitor an application, but it is computationally expensive and can cause unwanted performance degradation.

2.4.3 Performance Metric-Based Approach

Abnormal behaviour detection based on metrics correlation has been done by numerous researchers. A model defined by Sharma *et al.* utilized invariant network formation among monitoring metrics at the normal state of a system to identify its faulty state when those invariant edges got broken [96]. However, building the invariant network includes having prior knowledge of the systems behaviour.

Pollack *et al.* [92] implemented a tool called Genesis, which uses the correlated performance of a system with different loads, along with work-flow and dependency between components. They used a clustering technique to record the relation between load and performance in order to evolve normality model. The clusters are empty at the initial time of execution, but grow in size with the captured load and performance information gathered from the monitoring database. When a new cluster is formed or an existing clusters update significantly differs from the others, they report abnormal behaviour. After detecting abnormal behaviour, Genesis generates a snapshot and checks the snapshot against Service Level Objectives and Policy-based triggers. If it is diagnosed as a fault, the snapshot

is saved in the state-history, which can later be used to diagnose other faults. Although the researchers mention the clustering of load and performance, they do not discuss how work-flow and dependency affect the cluster. This approach detects the normal behaviour of a system based on pre-defined parameters, so it might not detect all possible deviation from the normal behaviour of an application.

In the fingerpointing approach [90], the authors propose a mechanism to detect local and global anomalies. Local anomalies are detected via specific threshold values that depend only on that computing node, after which system-wide global anomalies detection is performed. Differences in anomalies for each node are observed, but there is no comparison between the metrics in different nodes. Global fingerpointing depends on the accuracy of local anomaly detection. Here, they instrument each server node to collect different levels of application data, focusing on time series data along with OS and protocol level metrics. Hence, using this approach requires access to all levels of data, which is not always available, and on doing statistical analysis on predefined performance metrics.

Forming clusters from inter-metric correlation is proposed in [54]. Clusters are formed from similar metrics and the entropy of the cluster is calculated. The WilcoxonRankSum test is then applied to identify shifts in the in-cluster entropy from a normal to a faulty state. This approach used SigScore and BayesianScore algorithms, which require information on component dependencies. The complete component dependency information might not always be available, in which case this approach may not work.

2.5 Prior Work Limitations

An overview of prior efforts in the area of behaviour detection of software systems is discussed in this chapter. In addition, various limitations in previous work are addressed.

A summary of these limitations is given below.

1. Both static and dynamic model extraction approaches are dedicated to verifying the correct behaviour of a software system and need to access the source code of the software in order to do so. It is difficult to predict or generate all possible values that a variable can take in the application's runtime, so static approach works well for validating the application's design against its expected behaviour but it may not do the same for anomaly detection for unusual values taken by the variable.

2. Existing dynamic approaches learns the runtime behaviour of an application by tracing its execution path by instrumentation or annotation that can affect the performance of the application in production. So devising a tool for learning the runtime behaviour of the application without instrumentation is worthwhile.

3. Formal specifications(as practised by Amazon) is useful for generating abstract model that can be verified by model checker and ensure the design is right but it cannot ensure executable code correctly implements the verified design.

4. Some approaches detect the abnormal behaviour of a system based on some threshold values of predefined parameters, so they limit their applicability to find anomalies when they occur only for these parameters.

5. An approach that uses a virtual machine monitor and disk storage to find points of failure applies software probes that need to know the type of failure. If an unknown failure occurs, it cannot detect the deviation point properly.

6. An approach using system call traces to capture the runtime behaviour of an application will not work for detecting deviations in normal behaviour in applications unless a problem diagnosis request is issued.

7. Some approaches need to instrument different components to trace the flow of information in order to identify the faulty component, but this requires having knowledge

of the internal-structure of the system.

8. Most of the metric-based approaches rely on predefined performance metrics to detect anomalies, making their applicability limited to detecting failures caused only by those parameters.

Chapter 3

Solution Overview

This chapter describes the architecture of our solution. The algorithms implemented to determine the behaviour model of the target application are explained here. The behaviour model is used to detect anomalous behaviour of the target application in a continuous monitoring process.

3.1 Solution Architecture

Our solution is designed to model the behaviour of a target application. It comprises five components. These components analyze the snapshots collected from the target application and learn their statistical model in the learning phase and then validate the learnt model in the validation phase. Afterwards, the validated model is utilized to test the continuously collected snapshots and the test result is monitored for any anomalous behaviour by the application. An overview of the solution architecture is presented here.

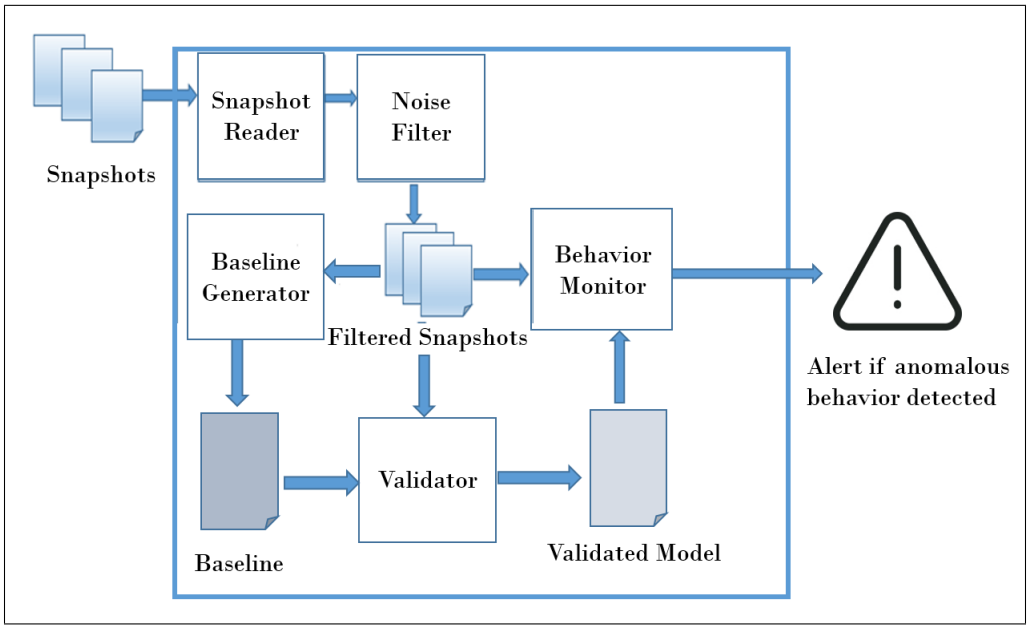

Figure 3.1: Architecture of The Behaviour Monitor

As we can see in Figure 3.1, our solution these following components:

- The ***snapshot reader*** takes the snapshots of the target application as input and convert them to an appropriate input format for the analyzer. In our work the collected snapshots are in binary format, so we implement a mechanism to read the binary file and convert it to a text file, which is the input format for our analyzer. The snapshot reader can be designed according to the specific format of snapshot. The implementation details of every component is discussed in the next chapter.

- The ***noise filter*** takes the output files generated by the snapshot reader as input and filter out the noise from them. To illustrate, a snapshot of an application may contain data which are not application-specific. For example, snapshot of an application running in a web server contains information that are responsible for running the server itself, which is common for all applications running on that server. The solution

is designed to monitor the system in application-level, so data that is not directly related to the application is considered to be noise. Filtering out this kind of noise is essential for learning the behaviour model of the target application.

- The ***baseline generator*** executes the learning phase of our tool. It analyzes the filtered snapshots and calculate the statistical measures (range of values, average, character-range used to form a string, etc.) of each variable present in the snapshot. Based upon learning a sufficient number of snapshots, it generates the *baseline* model. The *baseline* model contains the statistical characteristics of each variables present in the analyzed snapshots. Next, it is forwarded to the validation phase, which validates the learned *baseline* by comparing it to collected snapshots afterwards.

- The ***validator*** validates the learned statistical model present in *baseline*. It starts taking the filtered snapshots which are collected from the application after the *baseline* has been formed and then generate windows containing statistical measures of a user-defined number of snapshots. The size of the window is the number of snapshots present in that window. Each window is identified by its starting snapshot sequence number and ending snapshot sequence number. For example, a window of size 10 means it has the statistics of 10 consecutive snapshots and it can be identified by snapshots from 41 to 50. These windows are used to validate the *baseline* through different validation tests. The tests are run in the validation phase of our tool. A regression analysis and different range-testings are run to compare each window to the *baseline*. The variables that pass all the tests remain in the validated model and rest of them are not considered in the monitoring phase. The validator outputs the validated model of the target application. The validaion phase is necessary to reduce the number of false positive anomaly detection.

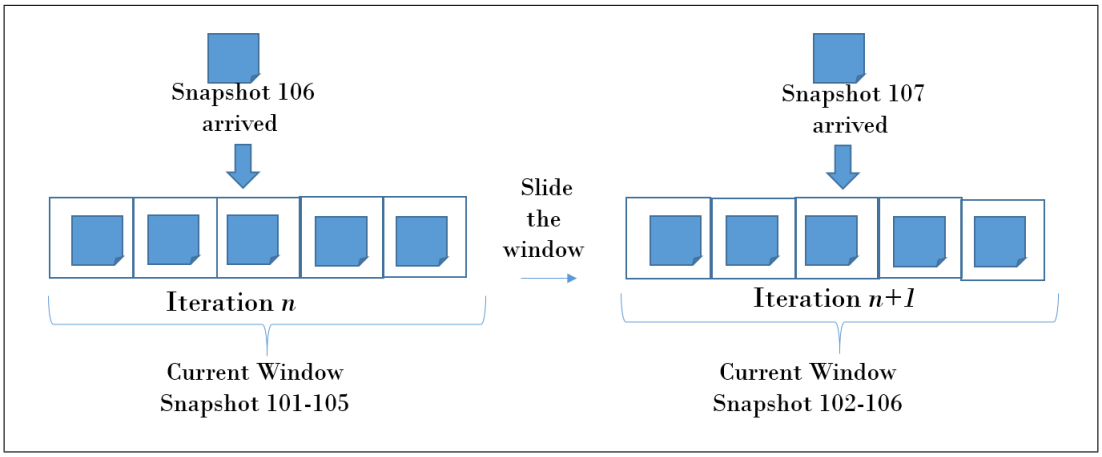

Figure 3.2: Sliding Window Generation

- The ***behaviour monitor*** monitors the application until any external interruption turns it off. It starts taking the filtered snapshots which are collected after the validation phase. It implements a sliding window of snapshots where each window is formed with the statistics of a user-defined number of snapshots. The next collected snapshot after the last window, and the window itself are compared by regression analysis and range testing in the same way as the validation phase. Each time a snapshot is collected, a new window is generated by sliding the previous window by one snapshot. Each window has a fixed size and identified by the starting and ending snapshot sequence number. Figure 3.2 illustrates this step for a window with size 5. When a new snapshot(snapshot 106) is collected then the current window (snapshot 101-105) and the new snapshot(snapshot 106) are compared in the current iteration. In the next iteration, the window slides by one snapshot (snapshot 102-106) and gets compared with the newly collected snapshot (snapshot 107). The behaviour monitor run tests to compare the new snapshot with the current window based on the parameters present in the validated model. If the new snapshot fits with the current window then the monitor does not take any action, however if the new snapshot does not fit then the monitor observes a sufficient number(x) of iterations and write the

result in a log file. Upon finding x consecutive failure in the fit test, the behaviour monitor detects it as an anomaly and generate alert for user. The checking for x consecutive failures is done in order to reduce the number of false positive alerts.

We have introduced the building blocks of the architecture of our proposed solution in the current section. The algorithms for the learning, validation, and monitoring phases will be explained in the next sections. For a quick overview of the algorithms, we summarize the execution flow of our tool as follows: first the snapshots are collected and filtered. Afterwards, the filtered snapshots go through the learning, validation, and monitoring phases based on their collection sequence. The outcome of the learning phase is the *baseline*, that contains the statistical properties of the first N snapshots. Next, the validator runs the validation test to compare the *baseline* and the validation windows and generates a validated model. Next, the validated model is forwarded to the behaviour monitor. Finally, the behaviour monitor runs monitoring tests to compare the sliding windows with the recently collected snapshot based on the parameters present in the validated model.

3.2 Learning Phase

The *baseline* generator learns the first N snapshots and generate *baseline*. It learns the different statistical property of each variable such as the range of value, the average value taken by the variables present in snapshots in learning phase. The learning process depends on the data type of the variable. The baseline generator keep aggregating the learned snapshot with existing *baseline* until N number of snapshots are learnt.

```
   Input: Snapshots S_1, S_2, .., S_N
   Output: Baseline
 1 begin
 2 |   vasrSet := { };// initialize an empty dictionary for storing
   |      statistics of values in each snapshot with the corresponding
   |      variable as key
 3 |   baseline := { };// initialize an empty dictionary for storing the
   |      aggregate statistics of values in N snapshots with the
   |      corresponding variable as key
 4 |   f_1 := ReadSnapshot(S_1);// Read the first snapshot for baseline
   |      initialization
 5 |   fS_1 := NoiseFilter(f_1);// Apply Noise filtering on the first snapshot
 6 |   for all var in fS_1 do
 7 |   |   baseline[var]=calculateStatistics(var,valueList(var));// calculate
   |   |      statistics of values of each variable present in the first
   |   |      snapshot and store it to the initial baseline
 8 |   end
 9 |   for i:=2 to N do
10 |   |   f_i := ReadSnapshot(S_i);// Read the collected snapshots
11 |   |   fS_i := NoiseFilter(f_i);// Apply Noise filtering on each snapshot
12 |   |   for all var in fS_i do
13 |   |   |   varSet[var]:=calculateStatistics(var,valueList(var));// calculate
   |   |   |      statistics of values of each variable present in the
   |   |   |      current snapshot and store it in the dictionary
14 |   |   end
15 |   |   baseline:=calculateAggregateStatistics(varSet,baseline);
16 |   |   i:=i+1;
17 |   end
18 |   output baseline;
19 end
```

Algorithm 1: generateBaseline

```
   Input: var,A value list of the variable, V
   Output: Statistics of V, V'
 1 begin
 2   if var is a number then
 3     minV:=calculate minimum(V);
 4     maxV:=calculate maximum(V);
 5     avgV:=calcualte average(V);
 6     V'.append(minV,maxV,avgV);
 7     V'.append(V);
       // we need to keep all the values in order to do the regression
          analysis
 8   else if var is a String then
 9     minV:=calculate minimum(stringlenght(V));
10     maxV:=calculate maximum(stringlength(V));
11     avgV:=calcualte average(stringlength(V));
12     if all V is alpha then
13       type:=alpha;
14     else if all V are alphanumeric then
15       type:=alphanumeric;
16     else
17       type:=notalphanumeric;
18     V'.append(minV,maxV,avgV,type);
19     V'.append((V));
       // we need keep all the values in order to do the regression
          analysis
20   else
21     V'.append(V);
22   output V';
23 end
```

Algorithm 2: calculateStatistics

```
   Input: varSet,baseline
   Output: Aggregated Statistics of varSet and baseline, baselineOut
 1 begin
 2     for all var in varSet and baseline do
 3         baselineOut[var]:=[ ];// initialize with empty list
 4         V:=value(varSet[var]);
 5         V_base:=value(baseline[var]);
 6         if var is a number then
 7             if V[0] < V_base[0] then
 8                 baselineOut[var].append(V[0]);
 9             else
10                 baselineOut[var].append(V_base[0]);
11             if V[1] > V_base[1] then
12                 baselineOut[var].append(V[1]);
13             else
14                 baselineOut[var].append(V_base[1]);
15             baselineOu[var].append(average(V[2],V_base[2]));
16             baselineOut[var].append(V[3..length(V)]);
17             baselineOut[var].append(V_base[3..length(V_base)]);
18         else if var is a String then
19             if V[0] < V_base[0] then
20                 baselineOut[var].append(V[0]);
21             else
22                 baselineOut[var].append(V_base[0]);
23             if V[1] > V_base[1] then
24                 baselineOut[var].append(V[1]);
25             else
26                 baselineOut[var].append(V_base[1]);
27             baselineOu[var].append(average(V[2],V_base[2]));
28             if V[3]==V_base[3]==alpha then
29                 baselineOu[var].append(alpha);
30             else if V[3]==V_base[3]==alphanumeric then
31                 baselineOu[var].append(alphanumeric);
32             else
33                 baselineOu[var].append(notalphanumeric);
34             baselineOut[var].append(V[4..length(V)]);
35             baselineOut[var].append(V_base[4..length(V_base)]);
36         else
37             baselineOut.append(V);
38             baselineOut.append(V_base);
39     end
40     output baselineOut;
41 end
```

Algorithm 3: calculateAggregateStatistics

3.2.1 Learning Single Snapshot

A variable can have multiple instances in a single snapshot(in our project, snapshot is the memory dump of Java Virtual Machine), so we need to calculate the statistics of the values taken by each variable. This calculation is done for each snapshot Algorithm 2. The statistics of every variable inside a snapshot is calculated based on their data type. For example, a variables having integer value or floating point value will be learnt by calculating its average, minimum and maximum range. On the other hand a variable with string data type will be learnt by the maximum, minimum and average of its length. Every string variable has a unique character range. It can be alpha-numeric or alpha or other than alpha-numeric. Therefore, the baseline generator learns the character range as a property for the variable. In case of boolean variable the value is stored as it has either $True$ or $False$ value. We also keep the values of each variable in order to test their distribution in the validation phase.

3.2.2 Learning Aggregated Statistics

The *baseline* of N snapshots is calculated by the aggregated statistics of N snapshots using Algorithm 3. Each time a snapshot is collected and the statistics of the snapshot is calculated, it gets aggregated in the *baseline* by invoking the *calculateAggregateStatistics()* method. After learning N snapshots, the *baseline* will have the aggregated statistics of N snapshots and return it as *baseline* from the Algorithm 1. It is then validated in the validation phase and then also used in the monitoring phase for comparison with incoming snapshots.

3.3 Validation Phase

The validator validates the *baseline* with the snapshots collected after N snapshots. These snapshots are filtered and pass through the validator and get grouped in separate windows. The windows are of same size and identified by the start and end snapshot sequence number. A regression analysis is run to find if these windows are a good fit with the learnt *baseline*. There might be a certain number of variables which are hard to model(random variables or monotonously increasing variables), so they will fail the validation and if they are forwarded to the monitoring phase, they will generate false alerts which are not expected. To elaborate, in the algorithm we are holding a hypothesis for a match between the distribution of values learnt in *baseline* and the validation window. If the hypothesis is rejected for any variable in the validator, we are not considering them in the monitoring phase.

3.3.1 Validation Window Generation

Algorithm 4 is explaining how the validator generates the validation windows and invoke the *validation()* method(Algorithm 6) on each window. Algorithm 5 is implemented for generating window of size w from snapshot S_i to S_j. The windows are generated in a similar way the *baseline* is generated. When the window generator is invoked it initializes the window with the statistics(Algorithm 2) of the first snapshot, in case of snapshots S_i to S_j, it is the snapshot S_i. Then it keep aggregating(Algorithm 3) the snapshots with the window till it reach the wth snapshot. If there are M snapshots to be tested in the validation phase then M/w number of windows are generated. Each window is sent to the *validation()* function along with the the *baseline* in order to validate the learning phase. Algorithm 6 explains the steps to validate the window with the *baseline*. As we can see, three types of testing are done in the validation phase. We are going to explain these test briefly in the following subsections.

Input: Snapshots $S_{N+1}, S_{N+2}, .., S_M$,Window Size w, *baseline*
Output: Validated Model

```
1  begin
2      vm := [ ];// initialize an empty list to store the validated
           variables
3      result := {};// initialize an empty dictionary to store the result of
           variables in each validation window with corresponding variable
           as key
4      vm_temp := {};// initialize an empty dictionary to store the list of
           results of variables from all validation window with
           corresponding variable as key
5      ivm:=[ ]; // initialize an empty list to store the variables which do
           not pass validation
6      for i:=N+1 to M do
7          window:=windowGenerator(S_i, S_{i+1}, .., S_{i+w-1});
8          result:=validation(window,baseline) for all var in result do
9              if var not exists in vm_temp then
10                 vm_temp[var]:=[ ];
                   // initialize an empty list as value of the var key
11             vm_temp[var].append(result[var])
12         end
13         i := i + w;
14     end
15     for all var in vm_temp do
16         if (count(False) in vm_temp[var])>= v * M/w then
17             ivm.append(var);
               // this will only insert those variables which failed the
                  validation more than half of the time windows are sent
18         else
19             vm.append(var);// this will only insert those variables which
                  did not fail the validation more than half of the time
                  windows are sent
20     end
21     output vm;
22 end
```

Algorithm 4: Validator

```
   Input: Snapshots S_i, S_{i+1}, .., S_j
   Output: Window
 1 begin
 2     varSet := { };// initialize an empty dictionary for storing
           statistics of values in each snapshot with the corresponding
           variable as key
 3     window := { };// initialize an empty dictionary for storing the
           aggregate statistics of values in snapshots i to j with the
           corresponding variable as key
 4     f_i := ReadSnapshot(S_i);// Read the first snapshot for baseline
           initialization
 5     fS_i := NoiseFilter(f_i);// Apply Noise filtering on the first snapshot
 6     for all var in fS_i do
 7         window[var]:=calculateStatistics(var,valueList(var));// calculate
               statistics of values of each variable present in the first
               snapshot and store it to the initial baseline
 8     end
 9     for k:=i+1 to j do
10         f_k := ReadSnapshot(S_k);// Read the collected snapshots
11         fS_k := NoiseFilter(f_k);// Apply Noise filtering on each snapshot
12         for all var in fS_k do
13             varSet[var]:=calculateStatistics(var,valueList(var));// calculate
                   statistics of values of each variable present in the
                   current snapshot and store it in the dictionary
14         end
15         window:=calculateAggregateStatistics(varSet,window);
16         k:=k+1;
17     end
18     output window;
19 end
```

Algorithm 5: windowGenerator

```
   Input: window,baseline
   Output: Validation test result for each variable,result
 1 begin
 2 |   for all var in window and baseline do
 3 |   |   result[var]:=True;// initialize with True means we start with an
 |   |       assumption that our hypothesis is true
 4 |   |   V_win:=value(varSet[var]);
 5 |   |   V_base:=value(baseline[var]);
 6 |   |   if var is a number then
 7 |   |   |   V_1:=value(V_win[3..length(V_win)]);
 8 |   |   |   V_2:=value(V_base[3..length(V_base)]);
 9 |   |   |   if hypothesisTest(V_1,V_2)==rejected then
10 |   |   |   |   result[var]:=False;
11 |   |   |   if V_win[0] < V_base[0] or V_win[1] > V_base[1] then
12 |   |   |   |   result[var]:=False;
13 |   |   else if var is a String then
14 |   |   |   if V_win[3]!=V_base[3] then
15 |   |   |   |   result[var]:=False;
16 |   |   |   else
17 |   |   |   |   V_1:=value(V_win[4..length(V_win)]);
18 |   |   |   |   V_2:=value(V_base[4..length(V_base)]);
19 |   |   |   |   if hypothesisTest(V_1,V_2)==rejected then
20 |   |   |   |   |   result[var]:=False;
21 |   |   |   |   if V_win[2] < V_base[0] or V_win[2] > V_base[1] then
22 |   |   |   |   |   result[var]:=False;
23 |   |   else
24 |   |   |   if V_win is not a subset of V_base or V_base is not a subset of V_win then
25 |   |   |   |   result[var]:=False;
26 |   end
27 |   output result;
28 end
```

Algorithm 6: validation

3.3.2 Hypothesis Test

The validation phase validates the baseline by doing regression analysis which is a null-hypothesis test for testing the similarity in distribution of the variables present in both of the *baseline* and the validation *window*. This test can be the Kolmogorov-Smirnov test, the Student-T test or the Student-T Confidence interval test or the Chi-square Test. This test is done to validate the fact that the variables in the *baseline* and in the windows are following the same distribution. It ensures that our tool has learnt the statistical property of the variable properly. To keep it a general algorithmic solution here, we are not mentioning any particular test name. We will discuss the test in the next chapter. In case of variables having integer or floating point data type the testing is done directly on the value list and for the variables having string data type the test is done on their string length.

3.3.3 Range of Value Test

The range of a variable is tested to find if the learning phase has modeled the range of values taken by a variable correctly. This statistical property is important to know as we are not having any intentional knowledge about the application; we are not doing code analysis. What we are learning about the application is the extensional knowledge which is learning the application's behaviour from run time which requires learning all possible statistical properties of a variable accessed in the application. For example, it is possible that an e-commerce application does not have a checking for negative number of items in a shopping cart. In that case if someone try to checkout with negative number of items and the application crashed for that, the range checking is going to be helpful and the operator can quickly identify the reason behind this crash. So beside hypothesis testing of variables it is equally important to learn their range and validate it to make sure we have learnt it properly.

In case of string variables, the minimum and maximum string length is tested. We can see the necessity of this testing from an example of *name* variable . Suppose the application developer assume the length of *name* should be 35 character long, where the database developer thought it should be 30 character long but the tester tests it only for maximum 20 character. Now this is possible because in companies engineers work in different group and they might not have the communication for deciding this fact. So when the application is in production it might get a 60 character long name, which is unusual and never tested by the tester. It is possible that someone has tried to write a long sting of random characters instead of *name* and the application server has not check it and pass it to database server and the database server is not accepting it and application get stuck there. Now if they use our tool for monitoring, they can easily find that all the value of *name* were following a range of string length and suddenly one *name* appeared to be longer than the maximum length found so far from the run time behaviour of the application. Therefore, range testing for string variables is an interesting statistical property.

In case of variables having categorical value, such as boolean value, we test whether the variable is taking either True or False in the learning phase and suddenly takes an opposite value during the validation phase. This testing is done because, the learning phase learns the application for a significant amount of time and if a variable takes the same boolean value in all snapshots in the learning phase and change its value in the validation phase, it means this variable's statistics is not contributing to model for the application. Hence it is advantageous to keep out this variable from monitoring phase to reduce unwanted false positive anomaly alert.

3.3.4 Character Range Test

The character-range testing is done for string variables to verify the type of character range learnt, is the only possible character-range for that variable. If we take the same example

as above, if someone tries to enter a SQL query in place of a name, for example, someone typed "Robert; Drop Table Customer;" and it went through the application and deleted the Customer table from database. It will take a long time for the operator to know where things went wrong when no customer can see their data, but if there was a way to check that the variable *name* is not getting the same character range(alpha or alpha-numeric or not alpha-numeric) as it gets in normal execution of the application it becomes very fast to localize what went wrong. So to detect this kind of unusual event, modeling the character-range property of a string variable is essential.

The different types of validation testing is done with a purpose to model the application's run time behaviour correctly. A validated model, vm is generated by this testings after M number of snapshots in M/w windows are compared with the *baseline*. This model includes only those variables which do not reject the hypothesis. These variables can be modeled by our approach correctly and can be used to monitor the application for anomalous behaviour. In order to do the validation accurately, we filter out only those variables which fail the hypothesis test in v number of the validation windows.The value of v ranges from 1 to M/w. These variables are stored in a separate list, ivm for future analysis. The validator authenticates the accuracy of the modeling and reduce the possibility of false positive alerts in the anomaly detection process. The validated model is used by the behaviour monitor in the monitoring phase.

3.4 Monitoring Phase

The monitoring phase is a continuous process which will keep monitoring the application unless a system operator turn off the behaviour monitor. We have discussed how the validated model of a target application is generated in section 3.3. This validated model is used here to compare only those variables which are learnt correctly. The monitor generates

sliding windows of snapshots where the window is slided by one snapshot each time. Once the required number of snapshots get collected, the continuous process of monitoring is started.

3.4.1 Sliding Window Generation

The behaviour monitor generates sliding windows by invoking the method *windowGenerator()*(Algorithm 5) where the assignment of *start* and *end* pointer of the window makes it slided. Snapshots S_{start} to S_{end} are sent as parameter to the *windowGenerator()* method, where *start* is initialized with the snapshot's sequence number L and the *end* is set to $L + sw - 1$. Here, sw is the size of the sliding window and $L > M$ where M is the last snapshot's sequence number in the validation phase. The monitor needs to wait for n snapshots to be collected where $n > end$, which means there are enough number of snapshots present to form the first window. The continuous loop of monitoring starts and keep monitoring the collected snapshots.

Algorithm 7 is explaining the procedure to call sliding window generator and call the *monitor()* function (Algorithm 8) with the current snapshot and the current window. The number of snapshots collected always need to be greater than the *end* pointer of the last window. And this is also ensuring that the comparison function is not invoked until enough number of snapshots have been collected. As it is a continuous monitoring process, we do not have the end limit for the number of collected snapshots here. The sliding window is generated by sliding the previous window by one snapshot as we have seen in Figure 3.2. It is achieved here by incrementing *start* and *end* pointer of previous window at the end of while loop. To elaborate, each time a window returns the current status of monitoring and the *start* and *end* is incremented by 1 in order to slide it by 1 snapshot. A system operator can decide from where they want to start monitoring or can choose the option to start monitoring automatically right after the validation phase, according to the decision,

Input: Snapshots S_L, S_{L+1},..,Window Size sw,Validated Model,vm
Output: Alert if anomalous behaviour detected

```
1  begin
2      anomaly := [ ];// initialize an empty list to store the consecutive
           three results to detect anomaly
3      result := {};// initialize an empty dictionary to store the result of
           variables in each validation window with corresponding variable
           as key
4      currentSnapshot := { };// initialize an empty dictionary for storing
           statistics of values in each snapshot with the corresponding
           variable as key
5      start:=L;
6      end:=L + sw − 1;
7      while new snapshot S_n collected and n > end do
8          f_n := ReadSnapshot(S_n);// Read the collected snapshots
9          fS_n := NoiseFilter(f_n);// Apply Noise filtering on each snapshot
10         for all var in fS_n do
11             if var is in vm then
12                 currentSnapshot[var]:=calculateStatistics(var,valueList(var));
13         end
14         currentWindow:=windowGenerator(S_start, S_start+1, .., S_end);
15         result:=monitor(currentWindow,currentSnapshot);
16         flag:=False;
17         for all var in result do
18             if result[var]==False then
19                 flag:=True
20         end
21         anomaly.append(flag);
22         if length(anomaly)==x then
23             if anomaly[0]==anomaly[1]==...=anomaly[x]==True then
24                 Generate Alert;
25                 anomaly:=[ ];
26             else
27                 anomaly:=[ ];
28         start:=start+1;
29         end:=end+1;
30     end
31 end
```

Algorithm 7: monitoringPhase

```
   Input: window,snapshot
   Output: Monitoring test result for each variable,result
 1 begin
 2     generate an empty logFile to log the status of monitoring for all var in window
       and snapshot and baseline do
 3         result[var]:=True;
 4         V_win:=value(varSet[var]);
 5         V_snap:=value(snapshot[var]);
 6         V_base:=value(baseline[var]);
 7         if var is a number then
 8             V_1:=value(V_win[3..length(V_win)]);
 9             V_2:=value(V_snap[3..length(V_snap)]);
10             V_3:=value(V_base[3..length(V_base)]);
11             if hypothesisTest(V_1,V_2)==rejected then
12                 if hypothesisTest(V_2,V_3)==rejected then
13                     result[var]:=False;
14                     Write the variable name and value in logFile and the type of
                       testing;
15             if V_win[0] < V_snap[0] or V_snap[1] > V_win[1] then
16                 if V_base[0] < V_snap[0] or V_snap[1] > V_base[1] then
17                     result[var]:=False;
18                     Write the variable name and value in logFile and the type of
                       testing;
19         else if var is a String then
20             if V_win[3]!=V_snap[3] then
21                 if V_base[3]!=V_snap[3] then
22                     result[var]:=False;
23             else
24                 V_1:=value(V_win[4..length(V_win)]);
25                 V_2:=value(V_snap[4..length(V_snap)]);
26                 V_3:=value(V_base[4..length(V_base)]);
27                 if hypothesisTest(V_1,V_2)==rejected then
28                     if hypothesisTest(V_2,V_3)==rejected then
29                         result[var]:=False;
30                         Write the variable name and value in logFile and the type of
                           testing;
31                 if V_win[0] < V_snap[0] or V_snap[1] > V_win[1] then
32                     if V_base[0] < V_snap[0] or V_snap[1] > V_base[1] then
33                         result[var]:=False;
34                         Write the variable name and value in logFile and the type of
                           testing;
35         else
36             if V_win is not a subset of V_snap or V_snap is not a subset of V_win then
37                 if V_snap is not a subset of V_base or V_base is not a subset of V_snap then
38                     result[var]:=False;
39                     Write the variable name and value in logFile and the type of
                       testing;
40     end
41     output result;
42 end
```

Algorithm 8: monitor

she can assign the value to *start*.

3.4.2 Anomaly Detection

The behaviour monitor invokes the function *monitor()*(Algorithm 8) for comparing the current snapshot with the current window for each variable if it exists in the validated model *vm*. The different types of testing we mentioned in the last section are also performed in the monitoring phase. While monitoring, if a variables fails the hypothesis test for the current window and the current snapshot, we are also checking whether it is also failing the testing between *baseline* and the current snapshot. It is possible that the application exhibits behaviour similar to the its starting point, in that case the current window might not have that similarity but *baseline* will definitely have it. Therefore, if a variable fails both the current window and *baseline* then it ensures that variable is exhibiting a statistics which was never seen before in the run time of the application. Similarly, the range checking, the character range checking are done both for the window and the *baseline* with the current snapshot.

The output of the *monitor()* method is tested for getting any variable failing the hypothesis test or the range test. An anomaly list is created at the beginning of the monitoring phase(Algorithm 7. A flag is used here in order to keep track of any single variable that has returned a $False$ from this method. If every variable passes the test then the flag remains $False$, and entered in the anomaly list. The status of monitoring is always logged in a file, $logFile$ to identify the cause immediately in case of an anomaly detected. If the flag gets $True$ for any variable, it means that some variable from the current snapshot has failed the test. If the flag turns $True$ for x consecutive windows, then the monitor generates an alert for anomalous behaviour. To ensure, x consecutive failed results, we keep track of the length of the anomaly list. Each time the length reach x, we check its value, and if we find it is not True for all x entry, then we simply empty the list and keep monitoring.

It becomes straight forward to identify the cause of anomaly by looking at the *logFile* and the snapshot collected at the time when the comparison started failing the test, as the same snapshot remains in x consecutive testing, we can safely make this argument that anomalous behaviour of the application started at that time when the snapshot was taken. The *logFile* contains the failing variables and their values. These variables exist in the application code, as a result it gives our solution a code level granularity in the process of anomaly detection and fault localization.

In this chapter, we have discussed algorithm for all the functions that are implemented in our project. We did not explain the function *ReadSnapshot* and $NoiseFilter$ which are used by the snapshot reader and the noise filter component respectively. These two functions are implemented based on the programming language used by the monitor and the application developer. We have given a general solution overview in this chapter, so any function that is implementation specific we did not elaborate here. We are going to discuss the implementation process briefly in the next chapter.

Chapter 4

Implementation

This chapter describes the detail implementation of our tool. Our target is to build an automated monitoring system which does not require any code instrumentation. It does not require access to source code or any system log file. As it is mentioned in the solution overview(Chapter 3) the input of the algorithm is the snapshots of the target application. In our tool, snapshots are the memory dump of the running application. Java Virtual Machine(JVM) comes up with a great utility called *jmap(Java Memory Map)* which can be used to collect memory dump of the running application. A brief overview of Jave EE is given in section 2.3.1, where features of JVM are also highlighted. In this chapter,we explain briefly, the snapshot collection and reading process along with the noise filtering techniques and the regression analysis used for monitoring the application. All the algorithms described in the Chapter 3 are implemented in Java and Python programming language. The original source code is available online in Grepcode [78]. The snapshot reader and noise filter is implemented in Java(version 1.8.0_60 with 64−Bit Server VM). The baseline generator, the validator and the behaviour monitor are implemented in Python(version 2.7.6 with GCC 4.8.2).

4.1 Snapshot Collection

The target application is built on Java EE platform which needs JVM to run the application in our project. JVM provides various utilities to debug the application. One of the way to debug an application is to analyze its heap dump. Therefore, it is advantageous to use the existing utility of JVM and collect the snapshot of the application in the form of heap dump.

4.1.1 Heap Dump

A heap dump is a snapshot of memory at a given point in time. It contains information on the Java objects and classes in memory at the time the snapshot was taken. It comes in various format like HPROF or PHD (Portable Heap Dump) [62]. The content of Java heap dump can be written to a binary file. Heap dumps can contain two types of objects:

- Alive objects only (those are objects, which can be reached via a Garbage Collector root reference)
- All objects (includes the no longer referenced, but not yet garbage collected objects along with alive objects)

Live objects can be determined from various VM internal mechanisms. Creating a full heap dump with dead objects takes much longer compare to live objects and also consumes more disk space. Therefore, collecting only live objects of a heap dump is more efficient than collecting all objects.

4.1.2 Collecting Heap Dump

Heap dumps can be created in 2 ways. If the Java option
"-XX:+HeapDumpOnOutOfMemoryError" is used while executing the Java program,then

the JVM will create a heap dump whenever it encounters an out of memory error before it quits. Therefore, in this approach heap dump cannot be taken unless the application runs out of memory and exit. Using the JVM utility JMAP(Java Memory Map), heap dumps can be created from the command line:

$$jmap - dump : live, format = b, file =< filename >< PID > \tag{4.1}$$

This will create a binary file containing all information of live objects of the Java application at the timestamp when the command is run. The *jmap* command can run in any machine that has JDK(Java Development Kit) installed in it. The *jmap* [84] command can extract the heap dump along with heap histogram from a running JVM using the process id of the JVM. The process ID can be extracted by the *jps* command.The heap dump can be stored in a binary file . The *jmap* command prints out the HPROF file in binary format. The HPROF is a heap profiler [79]. For faster analysis we create heap dumps of live objects. Garbage collected objects are out of this context.

4.2 Snapshot Reader

The snapshots are collected in the form of heap dumps of the application server. When the *jps* command is run, it prints the process id of the application server. Using the process id of the application server *jmap* command is run as shown in command 4.1. We use Jhat to read the heap dump in this project. It is an utility that comes with JDK which visualizes the heap dump in the localhost port using http server. Jhat utility requires a human operator to look at the result in web-browser to understand it. We implement a customized heap dump reader by rewriting the source code of Jhat and make its output in an input format for the tool which does not require a human operator to understand the heap dump.

The snapshot reader is the parser which parses the binary heap dump file. Once the file is read, the snapshot reader extracts all the class names, their instances, static and non-static data members and reference to those class names. The snapshot reader reads the heap dump and output the following type of Java classes:

- An abstract Java thing visitor class. Java thing can be Java Heap Object, Java Object reference or Java Value
- An Array Type codes as defined in Java Virtual Machine specification
- A class to represent null values, unresolved references.
- Classes to represent Boolean, Byte, Char, Double, Float, Integer, Long, Short, Static fields in an instance.
- Class file to extract class names from HPROF file.
- A Class file containing reference chain from and to some target object upon query. The Object Query feature is not used here. The assumption is, there is no information available about the application other than the process id. So it is not feasible to write query for individual objects to get detail information. This feature can be investigated in future.
- A class representing the Stack Frame
- A class representing the stack trace, that is an ordered collection of stack frames.

We are interested in the statistics of variables accessed in the execution of the application. So we need a mechanism to read the value of the variables efficiently and correctly. A heap dump contains more than thousand of classes, their references and instances. Now to access the different variables inside each classes, which are the field of the classes we need to read each of the instances created by the class. When a heap dump is created

```
    break;
}
case HPROF_LOAD_CLASS: {
    int serialNo = in.readInt();        // Not used
    long classID = readID();
    int stackTraceSerialNo = in.readInt();
    long classNameID = readID();
    Long classIdI = new Long(classID);
    String nm = getNameFromID(classNameID).replace('/', '.');
    classNameFromObjectID.put(classIdI, nm);
    if (classNameFromSerialNo != null) {
        classNameFromSerialNo.put(new Integer(serialNo), nm);
    }
    break;
}

case HPROF_HEAP_DUMP: {
    if (dumpsToSkip <= 0) {
        try {
            readHeapDump(length, currPos);
        } catch (EOFException exp) {
            handleEOF(exp, snapshot);
        }
        if (debugLevel > 0) {
            System.out.println("    Finished processing instances in heap dump.");
        }
        return snapshot;
    } else {
        dumpsToSkip--;
        skipBytes(length);
    }
    break;
}
```

Figure 4.1: A screen-shot of source code of Hprofreader

all instances of all classes get dumped in a single binary file. We use different methods to read the class names, their field's name, type of the fields and their data type and most importantly their value in each instance of the class. The source code is not included in the thesis but we want to discuss some of the frequently used class files and methods in order to give a comprehensive idea about how snapshots are read in this project. The original source code is available online at [78]. We re-implement some of these classes and add our own package to utilize those classes.

The *HprofReader* class(com.sun.tools.hat.internal.parser.HprofReader) is used to read the snapshot. It identifies a Java class by the magic number which is a pre-specified value to distinguish the Java class file. The value is always 0xCAFEBABE. In short, when the first 4 bytes of a file is 0xCAFEBABE, it can be regarded as the Java class

Java Bytecode	Type	Description
B	byte	signed byte
C	char	Unicode character
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L	reference	an instance of class¡classname¿
S	short	signed short
Z	boolean	true or false

Table 4.1: Type Expression In Java Heap [31]

file [31]. The binary file(heap dump) is read using *readUnsignedByte()* function by a *java.io.DataInputStream* object referring to the file. A screen-shot of small part of the code of *Hprofreader* is shown here in Figure 4.1. The *readHeapDump()* method is called based on the byte read from the file and its function is to read subsequent bytes to find all the classes, their instances, their ids and name form those ids. The entire heap dump is parsed and get stored in the *Hashtable*. A reference to *Hashtable* is return to the instance of *Snapshot* class(com.sun.tools.hat.internal.model.Snapshot) which is used for any queries on the snapshot.

All class names from the heap dump can be extracted using the *getClass()* method which is implemented inside the *Snapshot* class. In order to get the instances of the classes we use *getInstances()* method from the same class and to get the field inside the class we call the *getFieldsForInstance()* function. Java heap dump exhibits the data type with special symbol. We read those symbols by the *getSignature()* method and analyze the variables according to their data type. Table 4.1 is a list of data types and their corresponding

symbols inside the heap dump.

The class files we mentioned above are present in the original source code, but all of them are designed to show the information in a browser which listen to query request for specific classes. As we are not assuming any prior code level knowledge about the target application except the language it is written in, we cannot run query for a specific class. Therefore, we re-implement these class files and methods to return all class names, their instances, their values and data types. In order to read and print the heap dump in text format we have implemented a *viewer*package which is not present in the original source code. We are not providing the source code for this package but we want to describe the purpose of each classes implemented in this package for a better understanding of our work. Inside this package we have implemented these following class files:

- *QueryLoader* class is used to pass the instance of *Snapshot* class to *HeapReader* class. Its *setModel* method instantiates *HeapReader* class to read the snapshot and do all possible queries on the snapshot.

- *QueryFilter* class is used for doing queries on *classes* which pass the filter. In the process of noise filtering, we get *class names* which are application specific, and an "exclude class list" which are not application specific. In order to model the application accurately, we forward the data of only *application specific classes* to further components.

- *HeapReader* class's *read()* method instantiates *AllClassesQuery* class and *InstancesCountQuery* class to get information about the heap dump's *classes* and their instances.

- *AllClassesQuery* class is used for finding all the *class names* except those are present in the "exclude class list". We use the *getClass()* method here.

- *InstancesCountQuery* class is used for resulting the total number of instances of each *classes* which are returned in an array from the *AllClassesQuery* class's *allClass()* method.

- *AllServerClassesQuery* class is used for getting the *package names* which are common for all applications running in the same server, not for any specific application.

- *PlatformClasses* is used for getting the *class names* those are platform specific, not application specific. These *classes* are dumped in all Java programs using the same version of JVM. We need to record them to filter out noise.

- *InstancesQuery* class is used for ensuring a particular *class* is queried only when it has at least one instance. Its *instances()* method is invoked by the *read()* method of *QueryFilter* with the *class id* that has been extracted from the heap dump. From the *class id* it extracts the name of the *class* and output it. It calls the *objectquery()* method of *ClassQuery* class with the input *class id*

- *ClassQuery* class is used to output static and non-static fields's name of the *class* whose id it gets as input. It also prints the data type and the value of each field. As mentioned above here we call the *getFieldsForInstance()* and *getSignature()* methods to get the field and data type information respectively. The value of the filed as printed using *printThing* method which is implemented inside this class file.

There are more than thousands of *classes* exist in a single heap dump and each of these *classes* can have zero to hundred plus instances. Not all of them are related to the application directly. Some of them are platform classes which remains in every heap dump, some are server classes. In order to model the target application correctly and efficiently, it is essential to get details about only application specific classes, which contains information strictly related to the application. So we implement the noise filter to filter out the classes which are not contributing in modeling the application's behaviour.

4.3 Noise Filter

The noise filter is an essential component of our tool. As we discussed in the previous section, we do not assume any source code level knowledge about the target application, we need a mechanism which can learn the class names which are strictly related to the application's runtime behaviour. The noise filter is implemented to perform this job. Thee filtering techniques have been applied in our tool:

4.3.1 Filter1

In the heap dump there exist several instances of platform classes. These classes are dumped for every Java application. Therefore, the first filter is applied to exclude these platform classes as they will not carry relevant information about the behaviour of a particular application. Example of these platform specific classes are class [B ,class [C ,class [D, etc. These are the one dimension array of references that get dumped for all Java programs. They contains reference to classes of different data types. After closely inspecting these platform classes from different heap dumps we figure out that they do not contain specific information for any application. So we use the existing *exclude platform* parameter of the *HprofReader* class's constructor method to filter out these classes from our snapshot.

4.3.2 Filter2

The second filtering technique is applied to eliminate packages which are common for multiple applications running on the server. Those are packages containing the application server-specific classes. In order to resolve these packages, we collect the heap dump of the application server when no applications were deployed. We use the snapshot reader to read the heap dump and output the *package names* using the *AllServerClassesQuery*

class. The output is written to the "exclude package list" file and send as parameter to the *QueryFilter*'s *read()* method. The *QueryFilter* then exclude the classes of these packages while printing the instances of all classes present in the snapshot. Examples of these packages are *org.apache.derby.iapi.services.io*, *org.apache.felix.gogo.shell*, etc. These packages are not interesting because they will have common properties for all deployed application on the server. So classes inside those packages are not contributing in learning the behaviour of the application.

4.3.3 Filter3

The third filtering technique is applied to filter out classes which do not instantiate any object. These classes exist in the heap dump with zero instances. We use *InstancesQuery* to find the total number of instances of a class. If any class returns with zero number of instances then our tool does not take its data for further analysis. It is obvious that a class with zero instance will not give us any value for analysis. So we filter them out from the snapshots.

The noise filter clarify most of the classes which are not specific to a particular application. After filtering out irrelevant classes, the tool output data of interesting classes to filtered snapshots. These filtered snapshots have the class name, their total number of instances, value and data type of all fields in each instance. As we do not have the source code level knowledge about the application, it is still possible that some classes may remain in the filtered snapshots which are not relevant to the target application, however, the noise filter will not filter out something which is linked to the target application's behaviour. It is better to have more information than losing some important information. With that concept the filtered snapshots are passed to the learning, validation and monitoring phase.

4.4 Baseline Generator

The baseline generator initiates the learning phase of our tool. We have explained the learning phase in the Chapter 3 using algorithms which we implement to build this component. Algorithm 1 is implemented to generate the *baseline*. As mentioned at the beginning of this chapter, we implement the algorithm in Python. We import the *defaultdict* data structure from the *collections* module. This is a container data type. It is a dictionary where we can add a growing list as value for a key and do not need to initialize the dictionary with that key. The data structure is well suited for storing multiple layer of data. The filtered snapshots that the baseline generator gets from noise filter has all the class names, their filed names, number of instances and value of each field by each instance. We need to keep the data of the fields for each instance under each class in a structure that will be easily accessible for efficient computation,as a result we find *defaultdict* is a best choice.

The baseline generator reads the filtered snapshots and add the class name as key and their (filed, value) pair for each instance as the value of the key. As we have seen in Algorithm 1, the dictionary, *baseline* is initialized with the statistics built from the first collected snapshot. After that, each time a snapshot is read and filtered, the *calculateStatistics()* method is called for calculating statistics of all the classes present in that snapshot. Algorithm 2 presents the pseudo code of the whole process of reading each filed of a class and calculating their statistics based on their data type. At the end of calculation for each snapshot, its statistics is aggregated with existing baseline using the *calculateAggregateStatistics()* method(Algorithm 3). Finally, when N snapshots are read and integrated in the *baseline*, we get a initial model of the application. The *baseline* is then forward as input to the validator because we want to make sure that our tool has learnt the model correctly and detect anomalous behaviour in the monitoring phase efficiently.

4.5 Validator

The validator validates the output of the learning phase. It generates fixed size windows containing statistics of filtered snapshots and then test them with the *baseline* using regression analysis and range testing. The algorithms for validation phase are explained in the section 3.3. As we do not assume any prior knowledge about the application's behaviour, it is not possible to know how the value of each variables are distributed. Therefore, we did not choose the Student T test as it requires the data to be normally distributed. The Chi-square test needs the data to categorical, but from our observation on the baseline, we found most of our data is continuous and if they are categorical like boolean data, they keep the same value most of the time. We have implemented the two-sample KolmogorovSmirnov test(KS test) for the regression analysis as it does not require the data to be normally distributed.

The two-sample Kolmogorov-Smirnov test is a nonparametric test of the equality of continuous, one-dimensional probability distributions that can be used to compare two samples of data. The test quantifies a distance between the empirical distribution functions of two samples. The null distribution of this statistic is calculated under the null hypothesis that that the samples are drawn from the same distribution. The distributions considered under the null hypothesis are continuous distributions but are otherwise unrestricted. According to wikipedia, the two-sample KS test is one of the most useful and general nonparametric methods for comparing two samples, as it is sensitive to differences in both location and shape of the empirical cumulative distribution functions of the two samples.

We implement the validator in Python which has a great library resource for statistics. The *scipy* is a module of Python which is build for complex statistical computation. We import the *ks_2samp*() method from the *scipy.stats* module. The *ks_2samp*() method takes two list of values as parameter and returns a difference value(D-value) and a probability

value(P-value). If the P-value is close to 1, it means two list of values are coming from the same distribution. It is standard practice to use the P-value 0.05 or 0.01 to accept or reject the null hypothesis. In our tool, we reject the null-hypothesis of two samples coming from the same distribution if the P-value is smaller than 0.05. When a variable is present in the *baseline* and in the current validation window, we compare its value using the kS test. If the test returns less than 0.05 as P-value, then the null-hypothesis is rejected for the variable which means it fails the test. And if a variables fails the test for more than v of the total number of windows(line no. 15 of Algorithm 4), then we do not enter it in the validated model, we store it in separate list for future analysis.

The validator test each window for range test along with KS test. As we have discussed in the last chapter in section 3.3, all these tests are done to test the accuracy of the model we have learnt in the *baseline*. Therefore, the final validated model will only contain those variables which statistics are validated, that means they can be modeled properly using statistical analysis. This validated model pass to the behaviour monitor as input and reduce the possibility of having false alert of anomalous behaviour.

4.6 Behaviour Monitor

The behaviour monitor initiates the monitoring phase. This is a continuous process which can only be stop by a system operator. It monitors the incoming snapshots in sliding windows. We have explained the monitoring phase in the last chapter where we discuss how the sliding windows are generated and tested with each incoming snapshot and the *baseline*. The window-size can be set by the system operator and based on this size the monitor waits to start testing until the tool has collected and filtered n snapshots where n is greater than the *end* pointer of the window. We have discussed in the section 3.4, how the *start* and *end* pointer of the sliding windows are initialized and updated in each window.

The testing is done by invoking the *monitor* method(Algorithm 8), where the current window is tested with the current snapshot for each variable if it exists in the validate model. In case of failure, the variable is tested again with the *baseline*. If a variable fails both of the testing, then it returns *False*. The validated model is used here as a filter. Variables those are not present in the validated model are not compared between the window and the snapshot. It is useful as we do not want the tool to notify user frequently with false alerts. In order to detect an application's anomalous behaviour, the monitor keep track of the result from validation. If x consecutive windows fails the comparison test, then it generates alert for the user. The behaviour monitor does not stop on alert, it keeps monitoring for the next snapshots and possible anomalous behaviour.

The tool also keep a log of the variables which fails the test in monitoring phase. The log file makes it facile to localize the variable for which consecutive tests fail and that gives the application developer to investigate the source code and find the problem with that variable. Our tool is not only designed to detect anomaly but also to map the anomaly back to the source code. We have evaluated the tool using two applications. In the next chapter we are going to discuss the experimental setup and the evaluation result of our tool.

Chapter 5

Evaluation

We evaluate our tool with two target applications: an e-commerce application and to an online bidding application and describe the experimental setup and result in this chapter.

5.1 Configuration of computing Node

- We have deployed our application on 7 computing node that are named from 'Styx 01' to 'Styx 07'.
- Each of the computing node has $2TByte$ storage, 24 core CPU(Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz) and 64Gb RAM.
- Each of them have operating system Ubuntu 14.04.4 LTS (GNU/Linux $3.13.0-76-$ *generic* x86_64).

5.2 System Setup for E-commerce System

We implement an e-commerce system using the source-code from the Netbeans e-commerce demo program AffableBean [74]. It emulates an online grocery store where clients can buy products from different categories, add or update their shopping cart and get confirmation on checking out.

5.2.1 Database Server

We setup MySQL version−5.6.19 [86] as the database server in the computing node 'Styx 01'. All other nodes are accessing the database server of this machine. The database is named 'affablebean'. A JDBC connector is used here to connect the application server to the database server.

5.2.2 Application Server

Each of the computing nodes is running an Application Server. We setup Glassfish-4.2 [77] as the application server. This open source edition of Glassfish provides a server for the development and deployment of Java Platform, Enterprise Edition (Java EE platform) applications and web technologies based on Java technology. It has the following features:

- A lightweight and extensible core based on OSGi Alliance standards.
- A web container.
- An easy-to-use Administration Console for configuration and management.
- Update Tool connectivity for updates and add-on components.
- Support for high availability clustering and load balancing.

5.2.3 Client

We use Jmeter−2.8 [43] to simulate the client. It is an open source software that is designed to test functional behaviour and measure performance of web application. We found five types of shopper and estimate their shopping frequency by analyzing the current online shopping trend. These categories are as following:

Just Browser: This type of shoppers usually does not buy anything, scan through different categories of products and their price. Approximately 20% of shoppers are of this type.

Shopaholic: This type of shoppers buys products frequently. They browse and buy almost everything. 20% of shoppers are from this category.

Confused Shopper: 10% of shoppers are from this category. They are confused about what to buy. They browse, add product to cart and change their cart frequently. They keep updating their cart all the time.

General Shopper: 30% of the shoppers are general type. They browse and shop in a moderate way.

Confident Shopper: This type of shoppers have a specific list to buy items, they browse less and buy exactly what they need. 20% of shoppers are these type.

The computing node 'Styx 07' is set up for running the Client. Each server gets request from five categories of client. So there are 5 thread group for each server. Just Browser thread group having 20 user threads. Shopaholic thread group has 20, confused thread group has 10, general thread group has 30, and confident thread group has 20 user threads. Each of the thread group running 500-1000 times, producing 85,000 users for each server. In total 595,000 customers shopping on 7 servers.

5.3 System set up for RUBiS

We implement an online bidding system called RUBiS [24]. RUBiS is an auction site prototype, modeled after eBay, that is used to evaluate application's design patterns and server's performance. This benchmark implements the core functionality of an auction site: selling, browsing, and bidding. There are three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. The buyer and seller sessions require registration. In addition to the functionality provided during visitor sessions, during a buyer session users can bid on items and consult a summary of their current bids, rating, and comments left by other users. An auction starts immediately and lasts typically for no more than a week. The seller can specify a reserve (minimum) price for an item. RUBiS is a free, open source initiative. We implement the EJB version of RUBiS.

5.3.1 Database Server

We use the same database server as the AffableBean application. The database is named 'rubis'. The database is populated with 236 categories from 62 regions and 1000000 users. Each user put bid on items from different categories. Here also we use JDBC connector to connect the application server to the database server.

5.3.2 Application Server

Each of the computing nodes is running one Application Server. We use Glassfish-4.2 as the application server. Glassfish is well structured for implementing applications in Entity Java Bean(EJB). Also it has an interactive admin port that makes it easier to configure the server to use any third party API required to deploy the application.

5.3.3 Client

RUBiS source-code comes up with a Client emulator. The database is populated by users using the file named "rubis.properties". The same file is also used to assign the number of clients to be generated. Emulator is setup to send different types of request such as log in, store bid, view item etc. Also the workload for the emulator can be assigned from different type of workload files saved in the folder named *workload*, that is included in the source-code of RUBiS. We run the emulator with different workloads while taking the snapshots. The workloads are different in terms of the probability of different page request. Some of these generate clients who do more browsing, some of these generate clients who bid a lot of items and some of these sell and buy in more frequency. We setup the emulator in the computing node'Styx 03' that can send client request to all application servers using the emulator.

5.4 Experimental Result and Discussion

We collected the snapshots of both applications from each of the seven computing nodes using the process explained in the section 4.1. When the application server is started, we get the process id of the server using the *jps* command. Next, we use the *jmap* command 4.1 to get the snapshots(heap dumps). We collected 400 snapshots with a 3-minute interval between each snapshot. After collection, each snapshot goes through the snapshot reader and noise filter. Afterwards, the filtered snapshots go through the learning, validation, or monitoring phase based on their sequence of collection. The outcome of the experiment on the target applications are discussed in the following subsections.

5.4.1 Filtering Techniques

The filtering techniques filter out the classes that are not directly instantiated by the target application. We apply three filters, the first one clear out the platform classes, the second one filter out common packages for all applications running on the same server, and the third one clear out the classes with zero instances. We take two snapshots from each of the applications and tabulate the number of classes before and after applying filters.

The AffableBean Application

Before Noise Filter: 11540 Classes in the 45th Snapshot			
After Noise Filter	Filter 1	Filter 2	Filter 3
No of classes:	8211	1618	617
Before Noise Filter: 11545 Classes in the 100th Snapshot			
After Noise Filter	Filter 1	Filter 2	Filter 3
No of classes:	8212	1618	499

Table 5.1: Filtering techniques applied to snapshots of the AffabeBean application

We take the 45th snapshot of the AffableBean application as an example here. We can see in Table 5.4.1, without any filtering the snapshot had 11540 classes to analyze. The number went down to 8211 by applying the first filter. The second filter cleared out another 6593 classes. Finally, the third filter was applied which gave us 617 classes to analyze. Moreover, our experiment finds that not all the classes are present in all the snapshots and even though they are present, they may not instantiate any object. For example, the number of classes went down from 11545 to 499 for the 100th snapshot which is less than the number of filtered classes in the 45th snapshot.

The RUBiS Application

Before Noise Filter: 11205 Classes in the 40th Snapshot			
After Noise Filter	Filter 1	Filter 2	Filter 3
No of classes:	8143	1672	524
Before Noise Filter: 11236 Classes in the 80th Snapshot			
After Noise Filter	Filter 1	Filter 2	Filter 3
No of classes:	8145	1672	524

Table 5.2: Filtering techniques applied to snapshots of the RUBiS application

We got similar result from the experiment on the RUBiS application. The noise filter reduced the total number of classes to analyze from 11205 to 524 in the 40th snapshot (Table 5.2). The first filter reduced the number of classes by 3062, next, the second one filtered out another 6471 classes. Finally, the third filter cleared out another 1,148 classes. So by applying all three filters, we get 524 application-specific classes to analyze. Similarly, number of classes reduced to 524 from 11205 in the 80th snapshot.

We present the outcome of the learning, validation, and monitoring phases for both applications in the following subsections. The algorithms for these phases are discussed in the chapter 3 and their implementation is described in the chapter 4. For a quick revision, we can summarize the steps as follows: first, the learning phase is executed by the baseline generator and outputs the *baseline* model of the target application. Next, the *baseline* is validated with the validation windows. Finally, the validated model is utilized by the behaviour monitor in the process of continuous monitoring.

5.4.2 Learning Phase

We divided the 400 snapshots to be analyzed in three phases. We generate the baseline with 40 snapshots because in the process of collecting 40 snapshots the tool went through 2 hours($3minutes * 40 = 120minutes$) of execution time. The emulator sent different requests and the tool built its baseline model from the data exhibited in this time period. The outcome of the learning phase is following below.

The AffableBean Application

Total Number of Classes : 617								
Type	Integer	Signed Short	Long Integer	Double	Float	Boolean	Reference	Total
Boring	628	3	49	3	2	1089	5503	7277
Not Boring	24	2	9	1	0	19	65	120
Total	652	5	58	4	2	1108	5568	7397

Table 5.3: Category of variables in the learning phase of the AffableBean application

Boring Variables		
Reference	Null Reference	Total
3423	2080	5503

Table 5.4: Number of variables referring to null in the learning phase of the AffableBean application

The RUBiS Application

Total Number of Classes : 640								
Type	Integer	Signed Short	Long Integer	Double	Float	Boolean	Reference	Total
Boring	637	4	59	3	5	1101	5667	7476
Not Boring	13	0	7	1	0	11	37	69
Total	650	4	66	3	6	1112	5704	7545

Table 5.5: Category of variables in the learning phase in the RUBiS application

Boring Variables		
Reference	Always Null Reference	Total
3432	2235	5667

Table 5.6: Number of variables referring to null in the learning phase in the RUBiS application

We discovered two categories of variable from our experiment :

The Boring Variables: These variables do not change their value from the beginning of execution of the tool to the baseline model generation. We present their result in Table 5.3. We group the variables based on their data type. The variables in the group of reference type can refer to a string or any other class. If a boring variable is referring to a string then it takes the same value for all instances, otherwise it is referring to the same class. There are some variables that refer to null throughout the learning phase, so they are also enlisted in boring category(Table 5.4 and Table 5.6).

The Non Boring Variables: These variables exhibit different range of values and consequently their statistics are interesting. We present them under the non boring category in Table 5.3 and Table 5.5. For variables that are type of numbers(integer, float, double, short, and long integer) we calculate their average, minimum, maximum, and their probability distribution. For reference variable, our tool learns them as string variables because the heap dump contains either a string value or a reference to some class name. The tool calculates the minimum, maximum, average, and probability distribution of string length of this type of variables. Moreover, it learns the character-range of the string variable such as alpha, alpha numeric, and not alpha numeric. If they are alpha or alpha numeric then they are normal string variables, otherwise, in most cases they contain a class name.

The baseline contains the different statistics of variables. For example, the a shopping cart class in the AffableBean application has the following representation in baseline:

Class cart.ShoppingCart					
Name	Type	Minimum	Maximum	Average	Distribution
total	Double	0.0	311.0	156.990291262	Normal_greater
numberOfItems	Integer	0	104	52	Normal

Table 5.7: Example of statistics of variables in the baseline of the AffableBean application

Similarly, the 'User' class in the RUBiS application has the following representation in baseline:

Class entity.Users					
Name	Type	Minimum	Maximum	Average	Distribution
password	string, alpha-numeric	9	14	13.8556701031	Normal_greater
nickname	string, alpha	5	10	9.85910652921	Normal_greater
email	string,not alpha-numeric	22	32	31.7079037801	Normal_greater

Table 5.8: Example of statistics of variables in baseline in the RUBiS application

The variables shown in Table 5.7 are of number type. Our tool calculates their minimum, maximum, average and probability distribution. The distribution is measured using the P-value of 1 sample KS test. If the value is always greater than zero and follow a normal distribution, it is classified as normal_greater distribution and if the the value is always smaller than zero and normally distributed, then it is classified as normal_less. Moreover, a variable can follow a normal or exponential distribution. Otherwise, we present its distribution as undefined.

Similar statistics are calculated for the variables present in Table 5.8. In this example, the variables are string type, so the baseline contains the statistics over their string length as well as the character-range of each variable.

5.4.3 Validation Phase

The validator runs different types of validation test(section 3.3). 80 snapshots(41 to 120) are tested in 8 validation windows, each of size 10. Each window contains the statistics of variables in the same way as the baseline(Table 5.7). We take only those variables that pass all the tests in every window.

Result for The AffableBean Application

Total Number of Classes : 617								
Type	Integer	Signed Short	Long Integer	Double	Float	Boolean	Reference	Total
Boring	627	3	49	3	2	1088	5497	7269
Not Boring	25	2	9	1	0	20	71	128
Total	652	5	58	4	2	1108	5568	7397

Table 5.9: Category of Variables in the validation phase of the AffableBean application

As we can see in Table 5.9, the total number of variables remains the same as the learning phase. Eight of the boring variables change their value in this phase. So they are added to the non boring variable list. One previously null referring variable exhibits a different reference during the validation phase.

Boring Variables		
Reference	Always Null Reference	Total
3418	2079	5497

Table 5.10: Number of variables referring to null in the validation phase for the AffableBean application

Number of Variables Failing Test : 8				
Integer	Long Integer	Boolean	Reference	Total
3	3	1	1	8

Table 5.11: Number of variables failing the validation test for the AffableBean application

Eight variables fail the validation test including one boring variable of boolean data type. Upon closely inspecting these variables we find that they are monotonously increasing variable such as 'readtime' or 'readquerytime'. These variables contains time stamp as their values and updated with system clock and we do not model them. However, it will be interesting to build a separate model for these type of data in order to detect anomaly due to clock synchronization error. Other example of continuously increasing variables are 'confirmation_ number' or 'lastupdatedqueryId'. The 'confirmation_number' keeps increasing with every check out, similarly query-id gets updated with the latest query initiated by customers. We find that most of the variables that fail the validation test, need individual modeling technique. Moreover, they will fail the test in the monitoring phase as well and generate false positive alerts. Therefore, we do not included them in the validated model.

The RUBiS Application

Total Number of Classes : 640								
Type	Integer	Signed Short	Long Integer	Double	Float	Boolean	Reference	Total
Boring	637	4	59	3	5	1100	5665	7473
Not Boring	13	0	7	0	1	12	39	72
Total	650	4	66	3	6	1112	5704	7545

Table 5.12: Variables Found in Validation Phase in the RUBiS application

As we can see in Table 5.12, the total number of variables remain the same as learning phase. Three variables exposed different result than learning phase. Three of the boring variables change their value in this phase. Two of them are reference type and one is boolean type. So they added to the non boring variable list. One boolean type variable fails the validation test, that was boring in the learning phase.

Boring Variables		
Reference	Always Null Reference	Total
3430	2235	5665

Table 5.13: Number of not null and always null reference in validation phase in the RUBiS application

Number of Variables Failing Test : 4		
Integer	Long Integer	Total
1	3	4

Table 5.14: Number of variables failing the validation test for the RUBiS application

Four variables fail the validation test. When we closely inspect the variables failing the test we find they are either monotonously increasing variable like 'readtime' or 'lastUpdatedQueryId'. These variables contains time stamp as their values. As these variables are failing the test and their reason of failing is reasonable, we do not include them in the validated model and proceed to monitoring phase with variables that pass the validation phase for all windows.

5.4.4 Monitoring Phase

The behaviour monitor takes the validated model as an input and starts monitoring the incoming snapshots. It starts generating sliding windows and test every snapshot with the most recent window. We assign the window size to 10 in our experiment. The outcome of the monitoring phase for snapshot no. 121 to 400 is as follows:

The AffableBean Application

Total Number of Classes : 618								
Type	Integer	Signed Short	Long Integer	Double	Float	Boolean	Reference	Total
Boring	627	3	48	3	2	1088	5492	7263
Not Boring	27	2	10	1	0	21	82	143
Total	654	5	58	4	2	1109	5574	7406

Table 5.15: Category of Variables in the monitoring phase of the AffableBean application

Boring Variables		
Reference	Always Null Reference	Total
3417	2075	5492

Table 5.16: Number of variables referring to null in the monitoring phase of the AffableBean application

Six boring variables change their value in the monitoring phase, three of them fail the comparison test as they exhibit a different value than the baseline and the validated model. By closely inspecting them, we find that they are reference type variables and referring to a different class name in the monitoring phase. As we implement the EJB version of the application, there are some built in API that are used by the application, which we consider as black box and therefore, we did not explore the reason of failure in details.

A variable named 'total' from the 'ShoppingCart' failed the monitoring test a few time. This variable contains the total amount of prices of items in a cart and from the statistical analysis, we find its distribution varies from snapshot to snapshot. For instance, its value

sometimes takes an exponential distribution and sometimes a normal distribution, because of this variation, it is possible in some snapshots that it fails the KS test.

The RUBiS Application

Total Number of Classes : 640								
Type	Integer	Signed Short	Long Integer	Double	Float	Boolean	Reference	Total
Boring	637	4	59	3	5	1100	5665	7473
Not Boring	13	0	7	0	1	12	39	72
Total	650	4	66	3	6	1112	5704	7545

Table 5.17: Variables found in the monitoring phase of the RUBiS application

Boring Variables		
Reference	Always Null Reference	Total
3430	2235	5665

Table 5.18: Number of not null and always null reference in the monitoring phase of the RUBiS application

In case of RUBiS application only one variable failed the validation phase and that was the variable named 'logSessionString'. This variable contains the log of a session, and it exhibited a minimum string length that was not present in the baseline model or any validation window. By close inspection, we found that it was not an anomaly, it was a new log message recorded for a transaction with a new item. Although it was not anomaly, but it ensured that in case of any variable fail the range-test for anomalous situation, our tool can successfully detect it.

Anomaly Detection

In order to test the correctness of the statistical model generated by our tool, first we run the monitoring phase without any fault injection. The application should behave normally through out the monitoring phase in that case. With this assumption, any variables that violate their statistical modelling will generate a false positive result for anomaly detection. We summarize the result as following:

The AffableBean Application

Total Number of Snapshots	280
Total number of snapshots with false positive anomaly	33
Frequency of False Positive Alert	11.7857%
Maximum number of variables showing false positive anomaly in any snapshot	3
Maximum rate of false positive in any snapshot	2.09%
Rate of false positive anomaly over all snapshots	3.49%

Table 5.19: False positive anomaly detection in the monitoring phase of the AffabeBean application

We calculated the rate of false positive anomaly of the snapshots by the following calculation:

Total Number of Non boring Variables	143
Total number of variables violating their model	5
Percentage of false positive anomaly	5/143*100% = 3.49%
Maximum number of variables violating the model in any snapshot	3
Maximum rate of false positive anomaly in any snapshot	3/143*100%=2.09%

We then calculated the accuracy of our model by the formula:
$Accuracy = \frac{tp+tn}{tp+tn+fp+fn}$

False positive(fp)	True Negative(tn)	False negative(fn)	True positive(tp)
5	138	0	0
Accuracy =96.5%			

The RUBiS Application

We did similar calculation for the RUBiS application. The result is as follows:

Total Number of Snapshots	280
Total number of snapshots with false positive anomaly	4
Frequency of False Positive Alert	1.42857%
Maximum number of variables showing false positive anomaly in any snapshot	1
Maximum rate of false positive in any snapshot	1.38%
Rate of false positive anomaly over all snapshots	1.38%

Table 5.20: False positive anomaly detection in the monitoring phase of the RUBiS application

Total Number of Non boring Variables	72
Total number of variables violating their model	1
Percentage of false positive anomaly	(1/72)*100% = 1.38%
Maximum number of variables violating the model in any snapshot	1
Maximum rate of false positive anomaly in any snapshot	(1/72)*100%=1.38%

False positive(fp)	True Negative(tn)	False negative(fn)	True positive(tp)
1	71	0	0
Accuracy =98.61%			

As we have seen in the above calculated results, the statistical model generated by our tool is mostly accurate with a very less percentage of false positive result. In order to prove the accuracy of our tool in detecting true anomaly, we tested our tool on the AffableBean application by injecting small amount of fault. In our experiment, we were able to capture two anomalous situation.

- We re-implemented the AffableBean application and allow it to take a negative number as an input for the variable 'numberofItems' inside the 'ShoppingCart' class. In the monitoring phase, we sent a client request with negative number of items and it got captured in snapshot no. 300. While analyzing, this snapshot our tool successfully detected this anomaly by the range test where it finds a new minimum value for the variable 'numberofItems'.

- We sent a check out request with the string 'Drop Table Customer;' as an input to the 'name' field while taking snapshot 180 to 185 and this string got captured in the snapshot no. 181. From the baseline and validation phases our tool has learned the character range of the variable 'name' is alpha and putting a semicolon violates this range. As a result our tool successfully detected it as an anomaly. Moreover, it writes the variable that fails the monitoring test with its test result in the log file that can be used in future to localize any application-level failure.

Limitation

We collected snapshot every 3 minutes, so we were able to capture anomaly only when data with different statistical value got captured in the snapshot. If any object with unusual

value is created and garbage collected in this interval, we were not able to capture its variance. We aim to overcome this limitation in our future work.

5.5 Performance Overhead of Application for Monitoring

The collection of snapshot halts the application for a few seconds. From our experiment we found that for an application with less feature a heap dump collection can halt the system for maximum 5.74 seconds. On average it can take 3.0972 seconds. We recorded the heap dump-collection time for the two target applications and tabulated their statistics in the following Table.

Application Name	Minimum Time(seconds)	Maximum Time(seconds)	Average Time(seconds)
RUBiS	1.65	11.11	3.0972
AffableBean	1.04	5.74	1.5133

Table 5.21: Time spent for snapshot collection

Our tool is designed for monitoring application in replicated nodes. So when one node gets halted for snapshot collection, any other node can serve the client. So the effect of monitoring will not be visible. Moreover, the maximum time for collecting a heap dump was 11.11 seconds(Table 5.21). We collected snapshot in every 180 seconds. So it is a (11.11*100/180) = 6.17% hit on one computing node. In a replicated system, it will be 6.17% hit on one of the k computing nodes. For example, if we take a small system with 10 replicated nodes, it becomes 0.617% hit on the total system. So it will not cause a visible performance degradation for the total system. Therefore, our tool can be used to monitor an application when its is online and extract its normal behaviour from runtime-exhibited data.

Chapter 6

Conclusion and Future work

In this chapter we summarize our work. We conclude our thesis by providing direction to the future research regarding our work.

6.1 Conclusion

We proposed a solution to develop a behaviour model of an application which does not require any source-code examination or prior specifications about the expected behaviour from the application developers. To predict the abnormal behaviour of an application, we first need to perceive its normal behaviour. Numerous variables play vital roles in changing the behaviour of an application. Finding those variables often requires domain knowledge, code instrumentation, and annotation. Our solution does not require knowledge of the internal structure of an application or access to source code, nor does it need any code instrumentation. We conclude our work as follows:

- We describe the importance of the reliability and availability of software applications and explain how the downtime of web applications may affect an organizations reputation and cause significant monetary loss.

- Even large companies that maintain thousands of servers in data centers suffer from application downtime. In these centers, applications are replicated and deployed in several computing nodes. Although the applications are only deployed when fully functional, failure may still occur and, upon failure, it is nearly impossible for a human operator to find why one instance of an application failed when the others are functioning properly. Instead, automation is required to detect the problem and avoid future failures.

- Existing approaches regarding abnormal behaviour detection require either domain knowledge about the problem or a database of previously found failures. Moreover, these methods can only detect specific types of error, and some require access to the source code of the application. Approaches tracing the execution path of the application need to set frequent breakpoints, which then affect the performance of the application throughout the monitoring process.

- Our analysis on data collected at runtime is aimed at finding general faults rather than being restricted to a specific type of faults. Our solution does not require annotation or access to an applications source code, nor does it need to trace each event in the execution path.

- Our solution addresses the application-level monitoring of a system, whereas most existing approaches target platform-level monitoring. It is important to monitor a system at the application level, as it is the most exposed component and is intended for interaction with users.

- We implement a tool that can detect an applications normal behaviour without any code instrumentation. The behaviour is modeled as the statistical characteristics of variables used across the application. Variations in these characteristics have a significant impact on its execution. The statistical properties of these variables are determined directly from the captured snapshots of the application at runtime, so

there is no dependency on prior specifications by the application developer.

- We have described our proposed solution and implementation process in detail. The solution is implemented in Java and Python. In this thesis, we develop the tool for applications built in the Java Enterprise Edition platform, but we believe the solution can also be implemented for applications written in other development platforms.

- Our solution has three phases: learning, validation, and monitoring. In the learning phase, the collected snapshots are learned and a baseline model is generated, which is then validated with the snapshots collected in the validation phase. The purpose of validation is to reduce false positive anomaly alerts. The validator generates a validated model, which is then used in the monitoring phase throughout the lifetime of the application, unless the monitor is explicitly turned off.

- We evaluated our tool on an e-commerce application and an online bidding system and explained our results in the Evaluation chapter of this thesis. We discovered that there are certain variables which are interesting and that their statistics can contribute to building the applications behaviour model. At the same time, there are also some boring variables which do not change their values throughout the execution time. We record them because their characteristic of being boring is a part of the normal behaviour of the target application; if for some reason they change their value, our tool can immediately report it as anomalous behaviour and the cause can be investigated at once. Hence, we report both types of variables and discuss how their statistics can be used to determine the normal behaviour of the target application.

- We calculated the accuracy of the model generated by our tool and we find it is 96% accurate in case of the AffableBean application and 98% for the RUBiS application. Also the false positive rate is very low for both of the application. These results demonstrate the utillity of our technique to generate behaviour model of a software

application.

- Our tool keeps logs of variables and their values that exhibit variances in statistics. Should an anomaly occur, the log file can be used to identify the problem causing the variables. Because these variables exist in the application code, it gives our solution a code-level granularity in the process of anomaly detection and fault localization.

6.2 Future Work

Our hope is to extend our work in the future toward the better performance of our tool.

1. The snapshot collection process needs improvement, as the heap dump collection halts the system for few seconds in order to obtain a perfect memory dump at that point. If the process of the heap dump collection can be optimized, we can guarantee better monitoring performance. In a replicated environment, if one server is busy, another can serve the client, so a few seconds halt-time will not be noticeable. Even so, we still believe that we can be more efficient if we can collect a customized dump without halting the system.

2. We collect snapshots from the application every three minutes in our evaluation process. It is possible that a variable may expose abnormal statistics in the time between two snapshot collections, in which case our tool will not be able to capture that portion. However, if we start collecting snapshots more frequently, it might negatively affect the performance of the application. In future research efforts, we want to find a way to resolve this problem without affecting the performance of the running application.

3. We captured periodic heap dumps and they contain only fields of classes, not the value of local variables. A well-known tool named Daikon [41] captures all possible

writes for both local variables and fields. But in order to do it, the tool slows down the performance of the application. As our tool does not record all writes, it may produce less strong program invariants. However, the performance of the application is less affected by using our tool than Daikon. Generating a behavior model from strong and accurate program invariants without affecting its performance is a goal that we want to achieve.

4. We want to differentiate our tool's generated model with the same for other existing tools. In this project, we evaluate the applicability of our tool only, so we were not able to find its appropriateness in terms of finding anomalies compare to other tools. Moreover, we want to detect various types of anomaly and turn our tool to a robust anomaly detector.

5. The false positive alerts generated by our tool is noteworthy and its frequency for the AffableBean application is quite high(11%). We want to add some filtering techniques in this alert generation process. These filtering techniques can be formed using previously reported anomaly or with input from the system operator. Based on the output of the filter, false positives will only be generated for valid anomalies. Automating the whole process of anomaly detection and at the same time, having lower false positive alerts are very desirable properties for any monitoring system. We want to achieve this property in our future research.

6. Our tool can perform as an behaviour model generator for existing tools that need the expected behaviour to be specified by the application developers. For example, the output of our tool can be used as an input for Pip [95]. Also we did not consider the relationship among different variables. Existing approaches such as Genesis [92], information-theoretic monitoring [54], invariant relationships [96] are addressing the inter-variable relations. We can incorporate our tool with these mechanisms and get high level of accuracy in anomaly detection.

7. We implemented our solution only for a Java EE application, but we would like to implement it for applications built in other development platforms to see how efficiently the solution works in those cases.

References

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[2] Peter A Alsberg and John D Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd international conference on Software engineering*, pages 562–570. IEEE Computer Society Press, 1976.

[3] Inc. Amazon Web Services. Amazon Web Service (AWS). http://aws.amazon.com.

[4] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 307–320, 2012.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004.

[6] Bartosz Balis, Bartosz Kowalewski, and Marian Bubak. Leveraging complex event processing for grid monitoring. In *Parallel Processing and Applied Mathematics*, pages 224–233. Springer, 2009.

[7] Luciano Baresi and Sam Guinea. Event-based multi-level service monitoring. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 83–90. IEEE, 2013.

[8] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *HotOS*, pages 85–90, 2003.

[9] Peter C Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems (TOCS)*, 13(1):1–31, 1995.

[10] Michael D Bond and Kathryn S McKinley. Tolerating memory leaks. In *ACM Sigplan Notices*, volume 43, pages 109–126. ACM, 2008.

[11] Boris Jan Bonfils and Philippe Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *Information Processing in Sensor Networks*, pages 47–62. Springer, 2003.

[12] Douglas J Brown, Bill Suckow, and Tianqiu Wang. A survey of intrusion detection systems. *Department of Computer Science, University of California, San Diego*, 2002.

[13] George Candea, Emre Kiciman, Shinichi Kawamoto, and Armando Fox. Autonomous recovery in componentized internet applications. *Cluster Computing*, 9(2):175–190, 2006.

[14] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[15] Miguel Castro, Barbara Liskov, et al. A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

[16] K Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

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

[18] Yen-Yang Michael Chen, Anthony Accardi, Emre Kiciman, David A Patterson, Armando Fox, and Eric A Brewer. *Path-based failure and evolution management*. PhD thesis, University of California, Berkeley, 2004.

[19] Xi Cheng. Anomaly detection and fault localization using runtime state models. Master's thesis, University of Waterloo, 2016.

[20] CISCO. Webex. https://www.webex.ca/en/.

[21] Edmund Clarke, Orna Grumberg, and D Long. Verification tools for finite-state concurrent systems. In *A decade of concurrency reflections and perspectives*, pages 124–175. Springer, 1993.

[22] COMPUTERWORLDUK. The curse of online retail: 6 great price glitches. http://www.computerworlduk.com/galleries/it-business/curse-of-online-shopping-6-great-price-glitches-3594598/#2.

[23] COMPUTERWORLDUK. Top software failures 2015/2016. http://www.computerworlduk.com/galleries/infrastructure/top-10-software-failures-of-2014-3599618/#3.

[24] OW2 Consortium. Rubis: Rice university bidding system. http://rubis.ow2.org/.

[25] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Păsăreanu, Ro Bby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2000.

[26] IBM Corporation. Db2 v8.2 - system monitor guide. ftp://ftp.software.ibm.com/ps/products/db2/info/vr8/pdf/letter/db2d3e80.pdf.

[27] Microsoft Corporation. Clr - the common language runtime. http://msdn.microsoft.com/netframework/programming/clr/default.aspx.

[28] Microsoft Corporation. Dcom architecture. http://msdn.microsoft.com/library/en-us/dndcom/html/msdndcomarch.asp.

[29] Microsoft Corporation. The .net framework. http://msdn.microsoft.com/netframework/.

[30] Microsoft Corporation. Wmi - windows management instrumentation. https://msdn.microsoft.com/en-us/library/windows/desktop/aa394582%28v=vs.85%29.aspx.

[31] CUBRID. Understanding jvm internals. http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/.

[32] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[33] Gianpaolo Cugola and Alessandro Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, 2013.

[34] Kiran Deshmukh, Aarti Bhagure2 Prof Zafar Ul Hasan, R Anil, and Yogesh R Nagargoje. Detecting resource leaks in java program.

[35] Xiaoning Ding, Hai Huang, Yaoping Ruan, Anees Shaikh, and Xiaodong Zhang. Automatic software fault diagnosis by exploiting application signatures. In *LISA*, volume 8, pages 23–39, 2008.

[36] S. Duan and S. Babu. Guided problem diagnosis through active learning. In *Autonomic Computing, 2008. ICAC '08. International Conference on*, pages 45–54, June 2008.

[37] Grzegorz Dyk. *Grid Monitoring Based on Complex Event Processing Technologies*. PhD thesis, Masters thesis, University of Science and Technology in Krakow, 2010.

[38] Eclipse. Memory analyzer (mat). http://www.eclipse.org/mat/.

[39] Stephen Elliot. Devops and the cost of downtime: Fortune 1000 best practice metrics quantified. *International Data Corporation (IDC)*, 2014.

[40] JBoss Enterprise. A framework for organizing cross cutting concerns. ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/enUS/db2f0e81.pdf.

[41] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[42] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[43] Apache Software Foundation. Apache jmeter. http://jmeter.apache.org/.

[44] Apache Software Foundation. Apache module mod status. https://httpd.apache.org/docs/2.4/mod/mod_status.html.

[45] The Apache Software Foundation. Tomcat server. http://tomcat.apache.org/.

[46] Sachin Garg, Aad Van Moorsel, Kalyanaraman Vaidyanathan, and Kishor S Trivedi. A methodology for detection and estimation of software aging. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 283–292. IEEE, 1998.

[47] Eduard Glatz, Stelios Mavromatidis, Bernhard Ager, and Xenofontas Dimitropoulos. Visualizing big network traffic data using frequent pattern mining and hypergraphs. *Computing*, 96(1):27–38, 2014.

[48] Zhen Guo, Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 259–268. IEEE, 2006.

[49] Jungwoo Ha, Christopher J Rossbach, Jason V Davis, Indrajit Roy, Hany E Ramadan, Donald E Porter, David L Chen, and Emmett Witchel. Improved error reporting for software that uses black-box components. In *ACM SIGPLAN Notices*, volume 42, pages 101–111. ACM, 2007.

[50] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 381–390. IEEE, 1995.

[51] Apprenda Inc. Apprenda. https://apprenda.com/.

[52] Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems. *Software: Practice and Experience*, 44(7):805–822, 2014.

[53] Ali Imran Jehangiri. *Distributed Anomaly Detection and Prevention for Virtual Platforms*. PhD thesis, Göttingen, Georg-August Universität, Diss., 2015, 2015.

[54] Miao Jiang, Mohammad A Munawar, Thomas Reidemeister, and Paul AS Ward. Efficient fault detection and diagnosis in complex software systems with information-theoretic monitoring. *Dependable and Secure Computing, IEEE Transactions on*, 8(4):510–522, 2011.

[55] David B Johnson and Willy Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 171–181. ACM, 1988.

[56] Mark W Johnson. Monitoring and diagnosing applications with arm 4.0. In *Int. CMG Conference*, pages 473–484. Citeseer, 2004.

[57] Maria Jump and Kathryn S McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *Acm Sigplan Notices*, volume 42, pages 31–38. ACM, 2007.

[58] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *Software Engineering, IEEE Transactions on*, (1):23–31, 1987.

[59] Hyukmin Kwon, Taesu Kim, Song Jin Yu, and Huy Kang Kim. Self-similarity based lightweight intrusion detection method for cloud computing. In *Intelligent Information and Database Systems*, pages 353–362. Springer, 2011.

[60] Leslie Lamport. Computation and state machines. *Unpublished note*, 2008.

[61] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[62] Fabian Lange. Create and understand java heapdumps(act 4). https://blog.codecentric.de/en/2011/08/create-and-understand-java-heapdumps-act-4/.

[63] Ni Lao, Ji-Rong Wen, Wei-Ying Ma, and Yi-Min Wang. Combining high level symptom descriptions and low level state information for configuration fault diagnosis. In *LISA*, pages 151–158, 2004.

[64] P.A. Laplante. *Dictionary of Computer Science, Engineering and Technology*. Taylor & Francis, 2000.

[65] Philipp Leitner, Christian Inzinger, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Application-level performance monitoring of cloud services based on the complex event processing paradigm. In *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*, pages 1–8. IEEE, 2012.

[66] Long Li, Buyang Cao, and Yuanyuan Liu. A study on cep-based system status monitoring in cloud computing systems. In *Information Management, Innovation Management and Industrial Engineering (ICIII), 2013 6th International Conference on*, volume 1, pages 300–303. IEEE, 2013.

[67] Michael R Lyu et al. *Handbook of software reliability engineering*, volume 222. IEEE computer society press CA, 1996.

[68] Lucio Mauro Duarte. Behaviour model extraction from software. In *Theoretical Computer Science (WEIT), 2013 2nd Workshop-School on*, pages 1–8. IEEE, 2013.

[69] Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. D-cep4cma: a dynamic architecture for cloud performance monitoring and analysis via complex event processing. *International Journal of Big Data Intelligence 5*, 1(1-2):89–102, 2014.

[70] Peter Mell and Timothy Grance. The nist definition of cloud computing (draft). *NIST special publication*, 800(145):7, 2011.

[71] Alexander V Mirgorodskiy and Barton P Miller. Autonomous analysis of interactive systems with self-propelled instrumentation. In *Electronic Imaging 2005*, pages 188–202. International Society for Optics and Photonics, 2005.

[72] Nick Mitchell and Gary Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In *ECOOP 2003–Object-Oriented Programming*, pages 351–377. Springer, 2003.

[73] Mohammad Ahmad Munawar. *Adaptive Monitoring of Complex Software Systems using Management Metrics*. PhD thesis, University of Waterloo, 2009.

[74] Netbeans. Affablebean application. https://netbeans.org/kb/docs/javaee/ecommerce/design.html.

[75] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[76] Orace. Java ee documentation. http://www.oracle.com/technetwork/java/javaee/overview/index.html.

[77] Oracle. Glassfish server. https://glassfish.java.net/documentation.html.

[78] Oracle. Grepcode source file. http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/com/sun/tools/hat/Main.java/.

[79] Oracle. Java heap profiler documentation. http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html.

[80] Oracle. Java virtual machine specification. https://docs.oracle.com/javase/specs/jvms/se7/html/.

[81] Oracle. Java virtual machine tool interface specification. http://docs.oracle.com/javase/1.5.0/docs/guide/jvmti/jvmti.html.

[82] Oracle. Jconsole. http://docs.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html.

[83] Oracle. Jhat documentation. http://docs.oracle.com/javase/6/docs/technotes/tools/share/jhat.html.

[84] Oracle. Jmap documentation. http://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html.

[85] Oracle. Jmx. http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html.

[86] Oracle. Mysql. https://dev.mysql.com/doc/.

[87] Oracle-java.net. Visulavm. https://visualvm.java.net/.

[88] Sharon E Perl and William E Weihl. Performance assertion checking. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 134–145. ACM, 1994.

[89] S Pertet and P Narasimhan. Causes of failure in web applications. parallel data laboratory, 2005.

[90] Soila Pertet, Rajeev Gandhi, and Priya Narasimhan. Fingerpointing correlated failures in replicated systems. In *Proceedings of the second workshop on tackling computer systems problems with machine learning*, pages 21–30, 2007.

[91] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 49–49. IEEE, 2006.

[92] Kristal T Pollack and Sandeep M Uttamchandani. Genesis: A scalable self-evolving performance management framework for storage systems. In *Distributed Computing*

Systems, 2006. ICDCS 2006. 26th IEEE International Conference on, pages 33–33. IEEE, 2006.

[93] Thomas Reidemeister, Miao Jiang, and Paul AS Ward. Mining unstructured log files for recurrent fault diagnosis. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 377–384. IEEE, 2011.

[94] Thomas Reidemeister, Mohammad A Munawar, and Paul AS Ward. Identifying symptoms of recurrent faults in log files of distributed information systems. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 187–194. IEEE, 2010.

[95] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, volume 6, pages 115–128, 2006.

[96] Abhishek B Sharma, Haifeng Chen, Min Ding, Kenji Yoshihira, and Guofei Jiang. Fault detection and localization in distributed systems using invariant relationships. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–8. IEEE, 2013.

[97] Craig AN Soules, Jonathan Appavoo, Kevin Hui, Robert W Wisniewski, Dilma Da Silva, Gregory R Ganger, Orran Krieger, Michael Stumm, Marc A Auslander, Michal Ostrowski, et al. System support for online reconfiguration. In *USENIX Annual Technical Conference, General Track*, pages 141–154, 2003.

[98] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator placement for in-network stream query processing. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 250–258. ACM, 2005.

[99] SurfStat. Summarising and presenting data. https://surfstat.anu.edu.au/surfstat-home/1-1-1.html.

[100] Ariel Tamches and Barton P Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI*, volume 99, pages 117–130, 1999.

[101] Pedro Henriques Dos Santos Teixeira, Ricardo Gomes Clemente, Ronald Andreu Kaiser, and Denis Almeida Vieira Jr. Holmes: an event-driven solution to monitor data centers through continuous queries and machine learning. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pages 216–221. ACM, 2010.

[102] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 2003 IEEE Workshop on IP Operations and Management (IPOM)*, pages 119–126, 2003.

[103] Giridharan Vilangadu Vijayaraghavan. *A taxonomy of e-commerce risks and failures*. PhD thesis, Citeseer, 2003.

[104] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. *Science of Computer Programming*, 53(2):143–164, 2004.

[105] Andrew Whitaker, Richard S Cox, and Steven D Gribble. Configuration debugging as search: Finding the needle in the haystack. In *OSDI*, volume 4, pages 6–6, 2004.

[106] Andrew Whitaker, Richard S Cox, and Steven D Gribble. Using time travel to diagnose computer problems. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 16. ACM, 2004.

[107] Henry XuYuXu. Problem determination in message-flow internet services based on statistical analysis of event logs. Master's thesis, University of Waterloo, 2009.

[108] Tommaso Zoppi. Multi-layer anomaly detection in complex dynamic critical systems. 2015.