FAILURE CHARACTERIZATION AND ERROR DETECTION IN DISTRIBUTED WEB APPLICATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Fahad A. Arshad

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2014

Purdue University

West Lafayette, Indiana

To my parents, Asia and Arshad, who worked hard to provide me with the finest education and taught me the importance of learning from failures.
To my wife Sarah Arshad, whose encouragement and support made it possible to finish this work.

ACKNOWLEDGMENTS

Many people have contributed to this dissertation and I am thankful for their help.

I would like to thank my advisor, Professor. Saurabh Bagchi, for his feedback, encouragement and patience during my graduate work. He taught me the importance of critical analysis and rigourous evaluation of your research. His feedback, both positive and negative, helped me to imporve the quality of my research and groomed me as a researcher. I am thankful that he provided me with enough space to work on problems that I wanted to work on and warned me in time of dangers when I went offcourse. I am privileged to have him as an important person in my academic carreer.

I would also like to thank my committee members, Professor. Arif Ghafoor, Professor. Samuel Midkiff and Professor. Charles Killian, for their constructive feedback and encouragement.

Finally, I would like to thank all my colloborators, both at Purdue and outside of Purdue, who contributed towards this dissertation. A special thanks to DCSL (Dependable Computing Systems Laboratory) members who were always willing to help and collaborate.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

ABBREVIATIONS xi

ABSTRACT xii

1 INTRODUCTION 1

1.1 Motivation 1

1.2 Solution Approaches and Contributions 2

1.3 Published Work 5

1.4 Outline 7

2 BACKGROUND AND RELATED WORK 8

2.1 Fault Model Definitions 8

2.2 Summary of Related Work 8

2.2.1 Characterizing and Detecting Developer Bugs 8

2.2.2 Characterizing and Detecting Configuration Bugs 10

2.2.3 Detecting Duplicated Requests to Avoid Performance Bugs . 11

2.2.4 Automated Debugging of Performance Problems 12

3 FAILURE CHARACTERIZATION AND DETECTION OF PROGRAMMING MISTAKES 14

3.1 Introduction 14

3.2 Three-tier Architecture 17

3.3 Middleware 18

3.4 Client Requests 21

3.5 Database 22

3.6 Fault Injection Methodology 23

Page

3.6.1 Types of faults injected 23

3.6.2 Parameters and mechanics of fault injection 26

3.7 Failure Classification 27

3.7.1 Non-silent 27

3.7.2 Non-silent-interactive 28

3.7.3 Silent 28

3.8 Consistency Checks 29

3.8.1 Application Generic 29

3.8.2 Application Specific 29

3.8.3 Learning Parameters for Application-Specific Checks 31

3.9 Experiments and Results 32

3.9.1 Testbed and Metrics 32

3.9.2 Failure Distribution in the EJB Container 34

3.9.3 Application Generic Check 36

3.9.4 Application Specific Checks 37

3.10 Lessons Learned 47

3.11 Discussion 48

4 FAILURE CHARACTERIZATION AND DETECTION OF CONFIGURATION MISTAKES 50

4.1 Introduction 50

4.2 Threats to Validity 53

4.3 Java EE Application Server Design 53

4.4 Classification of Configuration Problems 55

4.4.1 Type of errors in Java EE application servers 55

4.4.2 When does the configuration error occur? 57

4.4.3 In what manner do the configuration errors manifest? 57

4.4.4 Who is responsible? 58

4.5 Classification Methodolgy and Results 59

Page

4.5.1 Methodology 59

4.5.2 Study-1 Results 60

4.5.3 Study-2 Results 62

4.6 Fault Injector 65

4.7 Fault Injection Results 68

4.7.1 GlassFish Injection Results 70

4.7.2 JBoss Injection Results 72

4.8 Observations and Recommendations 73

4.8.1 Observations on configuration problem patterns 73

4.8.2 Recommendations for better configuration management . . . 74

4.9 Discussion 75

5 DETECTION OF DUPLICATE REQUESTS CAUSING PERFORMANCE PROBLEMS 77

5.1 Introduction 77

5.2 Example Bug Case 81

5.3 Design 84

5.3.1 Synchronous Tracing 85

5.3.2 Modeling Application Behavior 86

5.3.3 Duplicate Detection Algorithm 88

5.3.4 Usage Modes 90

5.4 Experimental Setup 91

5.4.1 Configurations: Hardware, Software, Tracing 91

5.4.2 Evaluation Metrics 93

5.4.3 NEEShub Infrastructure 93

5.5 Evaluation 94

5.5.1 Experimental Workload 94

5.5.2 Accuracy and Precision Results 95

5.5.3 Sensitivity and Overhead 97

Page

5.6 Discussion 101

6 FAILURE DIAGNOSIS WITH METRIC-BASED PERFORMANCE PROBLEMS 103

6.1 Introduction 103

6.2 Overview 105

6.2.1 Measurement Gathering 105

6.2.2 Profiling 107

6.2.3 Workflow of our Approach 108

6.3 Design 109

6.3.1 Selection of Pertinent Metrics for Modeling 109

6.3.2 Modeling Sequential Data 112

6.3.3 Detection of Suspicious Metrics 115

6.3.4 Detection of Anomalous Code Regions 117

6.4 Evaluation 119

6.4.1 Case 1: Hadoop DFS 119

6.4.2 Case 2: HBase 122

6.4.3 Case 3: StationsStat 124

6.4.4 Case 4: Mambo Health Monitor 126

6.4.5 Summary of Cases 5–7 129

6.4.6 Performance 130

6.5 Discussion 132

7 CONCLUSION 135

8 FUTURE WORK 137

LIST OF REFERENCES 139

VITA 148

PUBLICATIONS 149

LIST OF TABLES

Table Page

3.1 Web Request to EJB Request Mapping 19

3.2 Accuracy of Null-Call Check 36

3.3 Accuracy of Call-Length Check 38

3.4 Accuracy of Head-Tail Check 42

3.5 Accuracy and Precision of Data-dependent Call-length Check with Skip-EJB Injected 45

3.6 Aggregated Accuracy of Data-dependent Call-length Check with Skip-EJB Injected 46

4.1 Statistics of bug reports that form our studies 59

4.2 Study-1: Distribution of Configuration and Non-Configuration Problems 61

4.3 Study-2: Distribution of Configuration and Non-Configuration Problems 63

4.4 An example of mutated configuration values in `domain.xml` in GlassFish 67

4.5 Number of non-silent manifestations for boot-time injections in GlassFish and JBoss 69

4.6 Detailed Exception Distriubtion in JBoss when "/" was Injected 69

5.1 Summary of Performance Results 96

5.2 Tracing Overhead 100

6.1 Almost equal variance explained by first two PCs. 110

6.2 Average use of file descriptors per class in HDFS for the specific bug discussed in Section 6.4.1. 121

6.3 Performance measurements. 131

6.4 Some of the used metrics. 134

LIST OF FIGURES

Figure Page

3.1 Middleware functionality separation 18

3.2 DayTrader System Architecture 18

3.3 DayTrader: Web and EJB container in the middleware layer 20

3.4 Web and EJB Request Distribution for 750 users 22

3.5 Example code for emulated error types in business logic 24

3.6 Failure classification 28

3.7 Web Request Classification 29

3.8 Distributions for learning parameters for application-specific checks . . 32

3.9 Distribution by Failure Class 33

3.10 Failure Distribution Breakdown for the two kinds of Non-Silent Failures 33

3.11 Null-call Check: Performance of application-generic check that matches `null` on return 37

3.12 Call-Length Check: Performance of application-specific check that matches the normal length of a web request to detect 39

3.13 Head-tail Match: Performance of application-specific check that matches the first and the last EJB request names to detect 41

3.14 Data-dependent call-length check: Performance of application-specific check that matches the call length with the expected runtime call length in the data-dependent portfolio web request 43

3.15 Overhead of application-specific checks in WEB, EJB and DB in terms of time spent per web request 44

4.1 Java EE Server Components Interacting with Configuration Data . . . 54

4.2 Study-1 GlassFish: Frequency Distribution of Configuration Problems in each Dimension (Total = 101 bugs) 62

4.3 Study-1 JBoss: Frequency Distribution of Configuration Problems in each Dimension (Total = 23 bugs) 62

Figure Page

4.4 Study-2 GlassFish: Frequency Distribution of Configuration Problems in each Dimension (Total = 132 bugs) 63

4.5 Study-2 JBoss: Frequency Distribution of Configuration Problems in each Dimension (Total = 25 bugs) 64

4.6 WorkFlow for Time of Injection 65

4.7 Number of issues filed to GlassFish and JBoss (vers. 3-6) bug repositories per month 70

4.8 Frequency Distribution of Exceptions for non-silent manifestations for GlassFish in `domain.xml` with "/" injection 70

5.1 Duplicate bug-manifestation (with missing images) before and after the fix 82

5.2 Overview of the duplicate-detection workflow. 84

5.3 Algorithm to detect duplicate messages from function call-depth signal. 89

5.4 Probes that fire in the life-cycle of an HTTP request 92

5.5 HOME→LOGIN→LOGGINGIN: Function call-depth signal for three web clicks from browser 96

5.6 HOME→LOGIN→LOGGINGIN: Autocorrelation for three web clicks from browser 97

5.7 Sensitivity of GRIFFIN for each of one, two and three-clicks 98

5.8 Detection Latency 99

6.1 Performance bug due to branch misprediction. 107

6.2 Overview of the problem determination workflow. 108

6.3 PCA-based heuristic for filtering out metrics. 111

6.4 Creation of the normal-behavior hyper-sphere. 114

6.5 Algorithm to select the suspicious metrics from traces of a failed run. . 115

6.6 Algorithm to select the suspicious code regions from traces of failed run. 118

6.7 Results from ORION for the HDFS bug. 120

6.8 Results from ORION for the HBase bug. 122

6.9 Results from ORION for the StationsStat case. 125

6.10 Mambo Health Monitoring system. 126

6.11 Results from ORION for the MHM problem. 128

ABBREVIATIONS

CC	Correlation Coefficient
SUT	System Under Test
JVM	Java Virtual Machine
EJB	Enterprise JavaBeans
DB	Database
PCA	Principal Component Analysis
HTTP	Hypertext Transfer Protocol
API	Application Programming Interface
TCP/IP	Transmission Control Protocol/Internet Protocol
JSP	JavaServer Pages
MVC	Model View Controller
DBMS	Database Management System
NFS	Network File System
OSGi	Open Service Gateway initiative
JSR	Java Specification Request
JMS	Java Message Service
GUI	Graphical User Interface
JDBC	Java Database Connectivity
CMP	Container-Managed Persistence
CDI	Contexts and Dependency Injection
JSF	JavaServer Faces
Java EE	Java Enterprise Edition
MM	Manifested-on-Metrics
NEES	Network for Earthquake Engineering Simulation

ABSTRACT

Arshad, Fahad A. Ph.D., Purdue University, August 2014. Failure Characterization and Error Detection in Distributed Web Applications . Major Professor: Saurabh Bagchi.

We have seen an evolution of increasing scale and complexity of enterprise-class distributed applications, such as, web services for providing anything from critical infrastructure services to electronic commerce. With this evolution, it has become increasingly difficult to understand how these applications perform, when do they fail, and what can be done to make them more resilient to failures, both due to hardware and due to software? Application developers tend to focus on bringing their applications to market quickly without testing the complex failure scenarios that can disrupt or degrade a given web service. Operators configure these web services without the complete knowledge of how the configurations interact with the various layers. Matters are not helped by ad hoc and often poor quality failure logs generated by even mature and widely used software systems. Worse still, both end users and servers sometime suffer from "silent problems" where something goes wrong without any immediate obvious end-user manifestation. To address these reliability issues, characterizing and detecting software problems with some post-detection diagnostic-context is crucial.

This dissertation first presents a fault-injection and bug repository-based evaluation to characterize silent and non-silent software failures and configuration problems in three-tier web applications and Java EE application servers. Second, for detection of software failures, we develop simple low-cost application-generic and application-specific consistency checks, while for duplicate web requests (a class of performance problems), we develop a generic autocorrelation-based algorithm at the server end.

Third, to provide diagnostic-context as a post-detection step for performance problems, we develop an algorithm based on pair-wise correlation of system metrics to diagnose the root-cause of the detected problem.

1. INTRODUCTION

1.1 Motivation

Distributed systems are ubiquitous and affect our everyday life in many different ways. We interact with reservation systems, financial systems, supply chain systems and many others for modern business. We shop at online stores, buy stocks and do online banking transactions. All these activities involve movement of financial assets and are managed by enterprise class distributed web applications. These business critical applications require high reliability to keep their customers' trust. To provide reliability in complex distributed applications, it is important to characterize failures, i.e., understand the scenarios under which an application fails to function correctly. Further, with the increase in scale of the applications, it becomes increasingly difficult to detect both functional and performance errors.

To understand the ways an application can fail, failure characterization is a critical step. An application can fail due to a variety of different software problems—for example due to a programming bug by a developer, due to a configuration mistake by an operator or due to an environmental condition such as server overload. For failure characterization, the application can be studied in two ways, fault-injection-based or bug-repository-based. After failure characterization, an operator has an additional insight on various failure patterns and their frequency of occurrence. With this knowledge, error detection techniques can be implemented to help alert the operators and developers when a problem occurs.

Error detection is an important step in the dependability life-cycle of an application as it helps to find the time point at which an application fails. Error detection methods provide reliability to the application and are the basis for any post-detection diagnosis. All detection techniques depend on capturing the failure manifestation of an inherent problem in the system. Failure manifestation can be non-silent or silent—for example an `Exception` captured in the logs or a performance problem at the server respectively. Error detection techniques need to be accurate with low-latency of detection. The methods employed to implement the detection techniques should have low programming overhead.

Any post-detection automated diagnosis helps to provide more context to the developers and operators in finding the root-cause of a problem. Finding the root-cause has been primarily a manual `printf`-based step for decades. Programmers typically use hit and trial methods to diagnose failures. They go through the time consuming process of modifying code, compiling, running several times. With the advent of more flexible dynamic tracing frameworks [1, 2], there is an opportunity to develop automated diagnostic tools that help to determine the root-cause of a problem at a finer granularity.

1.2 Solution Approaches and Contributions

In this dissertation, we present failure characterization of programming and configuration problems by fault-injection and bug-repository-based study in typical web applications and their corresponding underlying infrastructure. Additionally, we propose detection techniques for both functional problems and a class of performance problems called "unneeded duplicate requests". Finally, we present our ideas on diagnosing performance problems at a finer granularity. Below we present these contributions as four pieces of works.

1. **Failure Characterization and Detection of Programming Mistakes by Developer:**
   An approach to failure characterization in an online stock trading benchmark application, called DayTrader [3] is shown. The application has the typical structure of a web service in that it is deployed on a web container and an EJB container, with the persistent state being stored in a back-end database.

Failures are classified from an end-user perspective, into three categories: *Non-silent, Non-silent-interactive*, and *Silent*. Non-silent failures are characterized by explicit error messages from the infrastructure that are visible to the end user, e.g., a servlet exception that results in an HTTP 500 status in the browser. Non-silent-interactive failures are the ones that a user notices interactively, e.g., a user searches for five valid stock quotes but is returned only three. Silent failures disrupt web service without any manifestation to the user.

Each of these classes of failures is studied by developing a fault-injection framework and by observing how DayTrader reacts to each injected fault. The fault injection scheme mimics typical software bugs encountered in middle-tier of an enterprise software system. Specifically, we target bugs in business logic and control logic of an application. Business logic has functionality specific to the business (such as, buying and trading stocks in our application), while control logic handles input HTTP requests and triggers the business functionality. Our fault trigger for errors in business logic is method invocation. For errors in control logic, at compile-time, we inject *Skip-EJB*, where a call to a business method and corresponding usage of returned data is commented out in the source code. Skip-EJB represents missing programming constructs and is a dominant type of software bug studied in previous work [4]. By observing the behavior of DayTrader to each injected error, we subsequently recommend one application-generic and three application-specific detection checks. Application-generic check is based on catching programming constructs like `NULL` while application-specific checks are based on learning from normal back-end request-flow lengths and message-types.

2. **Failure Characterization and Detection of Configuration Mistakes by Operator:**

   A bug-repository-based characterization-study of configuration problems in Java EE (Enterprise Edition) based application servers, GlassFish [5] and JBoss [6] is presented. We classify configuration-bugs according to four dimensions: *Type, Time, Manifestation* and *Responsibility.* The first dimension (*Type*) considers a configuration-bug due to a problem relating to a particular parameter option, incompatibility with an external library or a missing component. The second dimension (*Time*) refers to the timepoint at which the configuration-bug is triggered, i.e., pre-boot-time, boot-time, or run-time. The third dimension (*Manifestation*) refers to whether a bug is silent or non-silent (with a log message). The fourth dimension refers to who (developer or user) is responsible for a given type of configuration bug. Based on the characterization, we built a tool called `ConfGauge`, that injects configuration errors (misconfigurations). A fault-injection tool like `ConfGauge` will help to measure the robustness, the ability to handle unintended input, and can play an important role in the decision making process of choosing a Java EE vendor. `ConfGauge` injects human-induced configuration errors and measures the reaction of the system under test (SUT).

3. **Detection of Duplicate Requests Causing Performance Problems:** A detection algorithm that uses autocorrelation function to detect duplicate web requests sent by client browsers is presented. There are two root causes for the problem of duplicate web requests, missing component names [7–9] and duplicate Javascript inclusion. The first cause is due to the incorrect way in which browsers handle missing component names, or empty tags, such as, `<img src="">`, `<script src="">`, and `<link href="">`. The second cause is due to the same Javascript being included in the page twice, or more number of times [10].

   We present a general-purpose solution to the duplicate detection problem, in a system called GRIFFIN. By "general-purpose", we mean that the solution applies unmodified to all kinds of resources and browsers. The solution has at its heart the observation that the duplicate web requests cause a repeated signal, for some definition of "signal". The signal should be defined such that it can be easily traced in a production web server, without impacting computation or storage resources and without needing specialized code insertion. We find that the *function call depth* is the signal that satisfies these conditions, while preserving enough fidelity that the repeated sequence can be easily and automatically discerned. To automatically discern the repeated pattern, we use the simple-to-calculate autocorrelation function for the signal and at a lag, equal to the size of the web request (in terms of number of HTTP commands), GRIFFIN sees a spike in autocorrelation which it uses to flag the detection.

4. **Diagnosis of Metric-based Performance Problems:** We present ORION, a framework for localizing the origin of problems, as a result of abnormal changes in metrics, in distributed applications. ORION works by profiling a variety of metrics as the application is executing, either at declared instrumentation points (such as, method entry or exit) or asynchronously with a fixed periodicity. Through machine learning techniques, it verifies if the runtime profile is *similar enough* to profiles created offline of non-faulty application's executions. If it is not, ORION goes back through traces to indicate which metrics caused the divergence and from that, to the region of suspect code. We provide a rank-ordered list to the developer for inspection. ORION uses pair-wise correlations between the metrics to diagnose errors.

1.3 Published Work

The main part of this dissertation has appeared in the following publications.

- **Dangers and Joys of Stock Trading on the Web: Failure Characterization of a Three-Tier Web Service**: Fahad A. Arshad, Saurabh Bagchi, In the 30th IEEE Symposium on Reliable Distributed Systems (SRDS), 10 pages, Madrid, Spain, Oct 5-7, 2011.

- **Characterizing Configuration Problems in Java EE Application Servers: An Empirical Study with GlassFish and JBoss**: Fahad A. Arshad, Rebecca Krause, Saurabh Bagchi, In the 24th IEEE International Symposium on Software Reliability Engineering, Pasadena, CA (ISSRE), 10 pages, Pasadena, CA, USA, November 4-7, 2013.

- **Is Your Web Server Suffering from Undue Stress due to Duplicate Requests?**: Fahad A. Arshad, Amiya K. Maji, Sidharth Mudgal, Saurabh Bagchi, In the 11th ACM International Conference on Autonomic Computing (ICAC), 6 pages, Philadelphia, PA, June 18-20, 2014.

- **Automatic Problem Localization in Distributed Applications via Multi-dimensional Metric Profiling**: Ignacio Laguna, Subrata Mitra, Fahad A. Arshad, Nawanol Theera-Ampornpunt, Zongyang Zhu, Saurabh Bagchi, Samuel P. Midkiff, Mike Kistler and Ahmed Gheith, In the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS), 10 pages, Braga, Portugal, October 1-3, 2013.

While working towards this dissertation, I have also contributed to several published works which relate to the field of error detection and diagnosis, but are not the primary focus for this dissertation.

- **An Empirical Study of the Robustness of Inter-component Communication in Android**: Amiya K. Maji, Fahad A. Arshad, Saurabh Bagchi and Jan S. Rellermeyer, In the 42th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 12 pages, Boston, MA, June 25-28, 2012.

- **How To Keep Your Head Above Water While Detecting Errors**: Ignacio Laguna, Fahad A. Arshad, David M. Grothe, and Saurabh Bagchi, ACM/IFIP/USENIX 10th International Middleware Conference, November 30-December 4, 2009, Urbana-Champaign, Illinois.

- **Stateful Detection in High Throughput Distributed Systems**: Gunjan Khanna, Ignacio Laguna, Fahad A. Arshad, and Saurabh Bagchi, 26th IEEE International Symposium on Reliable Distributed Systems (SRDS-2007), pp. 275-287, Beijing, CHINA, October 10-12, 2007.

- **Distributed Diagnosis of Failures in a Three Tier E-Commerce System**: Gunjan Khanna, Ignacio Laguna, Fahad A. Arshad, and Saurabh Bagchi, 26th IEEE International Symposium on Reliable Distributed Systems (SRDS-2007), pp. 185-198, Beijing, CHINA, October 10-12, 2007.

1.4 Outline

The rest of this dissertation is organized as follows: Chapter 2 discusses background and related work in the field of software failure characterization, error detection and diagnosis. Chapter 3 presents failure characterization and detection of programming mistakes by developers in three-tier web applications. Chapter 4 presents failure characterization and detection of configuration mistakes by operators in Java EE application servers. Chapter 5 presents detection of duplicate requests that cause performance problems. We present our ideas on metric-based error diagnosis for performance problems in chapter 6. Conclusion and Future Work is presented in Chapter 7 and Chapter 8 respectively.

2. BACKGROUND AND RELATED WORK

2.1 Fault Model Definitions

We give an overview of the fault model that we follow in this dissertation.

- **Fault:** *Fault* is defined as an inherent defect, which is latent, and was introduced into the system at some point in time. An example of a fault is a programming mistake (e.g., a variable which is incorrectly `null` in a given application) or a wrong configuration (e.g., an incorrect configuration value such as a wrong filename).
- **Error:** When a fault is activated, it gives rise to an *error*. For example when a variable which is incorrectly `null`, is accessed, or when a missing file is accessed.
- **Failure:** A *failure* occurs when the error handling is not enough and the problem is manifested to the application user. An example is a `NullPointerException` or a `FileNotFoundException` manifested in the browser on sending a web request.

2.2 Summary of Related Work

2.2.1 Characterizing and Detecting Developer Bugs

Studying failure characterization involves two general methodologies, i.e., analyzing bug databases and employing fault injection. In the former case, multiple bug reports are analyzed and correlated to classify failures according to some criteria, e.g., [11] studies failures in JVM, classifies them based on failure manifestation, failure source and failure frequency. [12] studies failures in application servers and shows that 75% of all failures are related to service semantics and require application-specific efforts. Bug repository based failure analysis has some limitations, i.e., quality of bug reports and subjective administration, etc., therefore many researchers have resorted to fault-injection, e.g., [13], [4] emulates classes of software faults. For evaluation based on fault injection, different injection campaigns depending on *where* and *how* to inject errors exist. Faults can be injected in source code [14], or they can be emulated at runtime using an emulation technique [13]. We emulate faults in the application, and classify them based on manifestation at client's browser. Failures can also be analyzed at different layers in the software system, i.e., operating system [15], [16], network controller [17], virtual machine [11], application server [12] and application [18]. The type of injected fault for a given system also varies depending on the system under test, e.g., [16] shows that, one of the major causes *null pointer* is responsible for system crashes in operating system kernel. We study null-calls in a similar fashion at the application layer.

Detecting injected faults as studied by [19] can be done at a system level or at an application level. The study shows that low-level system monitoring tools are not enough to detect application level errors. Pinpoint [20], [18] treats failures as anomalies and applies machine learning techniques to detect application level failures in an application-generic way. For application-specific evaluation, assessing application problems is important, e.g., [21] applies a set of tests, by mutating call parameters, and analyzes robustness of web services code and application server infrastructure. Similarly, [22] uses a set of robustness tests to detect programming and design errors, along with classifying silent errors. We follow a similar approach, but instead of robustness failures, we focus on more general emulated failure classes.

Characterizing by failure type, especially variants of silent failures is studied by [17]. The study detects silent failures that impact database clients in a telephone network. It uses software fault injection at database tier and does data audit to reduce silent failures. An experimental study [23] quantifies distribution of bugs among different layers, i.e., business, presentation and data logic layer of Web and Desktop applications. Our study focuses on business and control logic in middle-tier and is for a three-tier application. Another study on silent errors in OS [15] estimates the percentage of silent failures by comparing the execution of a given API across different operating systems. Application-specific detection, diagnosis and recovery are all important. A study in [24] shows that application generic recovery techniques are not enough to recover from application failures. Our work tests this hypothesis from a detection perspective and is helpful to give useful hints for diagnosis. In addition, if detection results by our checks are taken into account by an application logger, then this can improve logging quality as shown in [14].

2.2.2 Characterizing and Detecting Configuration Bugs

From the perspective of configuration management, software systems reliability has been studied from different angles, characterization, testing, detection, diagnosis and recovery: (1). Failure characterization [25], where the goal is to study the cause and effect of failures to identify misconfiguration-patterns. (2). Resilience testing [26], where the goal is to study the reaction of a software system when injected with configuration errors to identify weaknesses in failure handling mechanisms. (3). Configuration error-detection [27], where the goal is to study failure manifestations of misconfigurations to identify patterns that detect a given configuration-error. (4). Failure diagnosis [28], where the goal is to find a root-cause of a given configuration problem. (5). Failure Recovery [29], where the goal is to find the candidate fixes for a configuration problem.

Failure characterization studies using bug-reports, both on configuration [25] and non-configuration [30] bugs, reveal common patterns that lead to failures. Yin *et al.* [25] studied user forums for five non-Java EE systems from the perspective of configuration errors. Some dimensions studied in their work, e.g. *type* and *manifestation* of configuration problem are similar to our work, while others e.g. *responsible-party* and *problem-time* are different. Another major difference is that their work focuses on non-Java EE systems, whereas we claim that Java EE based systems have distinctive differences (e.g. standardized APIs, modular architecture) that warrant separate attention. Also, as cloud-based Java EE system like OpenShift [31] and GlassFish-4 become mainstream, Java EE based systems would require more focused attention towards configuration-related issues.

Keller *et al.* [26] tested the configuration system of five non-Java EE systems by injecting misconfigurations using a tool called `ConfErr`. `ConfErr` is different from our work, as we inject misconfigurations that have been reported in bug repositories, while `ConfErr` injects misconfigurations that are derived from error models in human psychology only. We consider `ConfErr` complimentary to our work, `ConfGauge`, as both efforts give an indication of how robust a particular software is.

Besides failure characterization and configuration testing, configuration-error detection [27], *Barricade* [32] and diagnosis works like [33], *PeerPressure* [34], *Chronus* [35], *ConfAid* [36] are other areas that have been studied in configuration management research.

2.2.3 Detecting Duplicated Requests to Avoid Performance Bugs

Most of the existing approaches to handle duplicate requests are *not* at the application-level. TCP [37] is the classic example that uses sequence numbers along with a windowing-based mechanism to do duplicate detection of IP packets. Stateless protocols like HTTP have to deal with the request-response nature and maintain state at the application-level. Application-level works include similarity detection [38] deployed at web-proxy caches to eliminate redundant network traffic, duplicate-content detection [39] with clustering and similarity metrics [40]. These are directed at generic payloads and are therefore less accurate than our work. Another related area is schemes for avoiding the occurrence of duplicate requests in the first place, which are complementary to our work. These schemes are implemented either on the client-end [41] or on the server-end [42].

Finding relevant system events to detect and diagnose failures is often equated to the problem of finding a needle in a haystack. Over the last decade, several researchers have proposed solutions to this challenging problem [43–46]. The high-level objective here is to mine vast amounts of system data to find relevant signatures for failures. Once "syndromes" [43] or signatures are created [44], these can be used to detect problems in the future efficiently. Present day data centers often suffer from tens of minutes to hours of downtime due to inherent difficulties in diagnosing and fixing such failures. [45] and [46] partially automate this process, thereby, reducing manual effort and downtime. Our work falls within this broad umbrella. We automate the process of detecting duplicated web requests by looking at a compressed signal from system events, specifically function calls and returns.

2.2.4 Automated Debugging of Performance Problems

Various tools exist to help programmers and network administrators localize the problems in distributed applications. Most of these tools can be grouped into these main categories: traditional debuggers and execution replays, model checkers, and statistical methods. Each type of tools is appropriate for different scenarios, as we will discuss below.

Traditional debuggers such as gdb, and tools that enable execution replays [47,48] let the programmer find the exact line of code of the bug. However, for this to be feasible, the programmer needs to have a good guess of where the problem lies. Therefore, these tools are complimentary to our tool since our tool reports suspicious regions of code, and when it is not obvious where the bug is, the programmer can use these tools to pinpoint the bug. Attariyan *et al.* [49] propose a technique that determines the root cause of performance problems using taint analysis. It is suitable for cases where the source of the problem is user input or the configuration file.

Model checkers are useful for checking small applications against specifications [50, 51]—due to its exhaustive nature, it is not feasible for most real-world applications.

In addition, it is more appropriate for checking against specifications. Liu *et al.* [52] propose a technique that does live model checking and provides execution replay. The programmer writes a predicate that is invariant throughout the execution, and this predicate is checked as the application runs. When the predicate is violated, the system states leading to the violating state are given as output. While this approach works well for specifications, as the instruction that changes the system state from conforming to violation is usually the root cause of the problem, for other types of problems such as performance problems, the errors may have accumulated from different regions of the code before the specification is violated. Our tool does not assume that the instruction that makes the system state in violation is the root cause of the problem, and is thus more applicable to a wider range of problems.

There is a volume of work on statistical methods to detect and localize problems. Some of the work analyzes application logs [53–56]. However, there is often a one-to-many mapping between the log record corresponding to the problem and the actual code regions that could be the source of the problem. [57–59] analyze request flows to diagnose problems in request-processing applications. Other work analyzes metric values, typically using machine learning algorithms [60–63]. In [60] and [61], the signature of the current problem is compared to a database of known problems. If there is a match, the diagnosis and fixes used previously can be reused again. This approach is suitable for problems that are not easy to fix even if the root cause is known (e.g., overloaded servers), problems due to the environment, or hardware problems. In other situations, once a problem is diagnosed and fixed, it will not occur again, limiting the usability of the tool. The overall approach of [62] and [63] is similar to our approach in that machine learning models are trained based on training data. If the system is in an abnormal state, the metrics that are most abnormal are given to localize the problem. In addition, when this is not enough to pinpoint the location of the fault, our work goes a step further and provides a ranking of most suspicious code regions to reduce the programmer's effort needed to fix the problem.

3. FAILURE CHARACTERIZATION AND DETECTION OF PROGRAMMING MISTAKES

3.1 Introduction

Business critical enterprise applications require high reliability to keep their customer's trust. A customer trusts the service provider if all her web requests are satisfied, and possibly for a small fraction of requests, a noticeable failure is flagged to the customer and the customer's application state is not corrupted. An instance, where a failure of a web request is not made obvious to the consumer, but affects the end functionality, possibly at the time of a later transaction, is highly undesirable. To avoid this, the service provider needs an understanding of how software errors affect her web service, such as which failures occur in a silent manner and which in a non-silent manner; for the latter class, how are the failures logged (where—at the web server, the application server, or the database server, and within each, at which component; and the level of detail in the log). With an understanding of the failures, the service provider can start to design techniques to better log failures and to reduce the incidence of silent failures through appropriate, preferably non-intrusive, detection checks. In this chapter, an approach to failure characterization in an online stock trading benchmark application, called DayTrader [3] is shown. The application has the typical structure of a web service in that it is deployed on a web container and an EJB container (specifically, the Glassfish [5] application server), with the persistent state being stored in a back-end database. DayTrader is a benchmark derived from IBM's performance benchmark *Trade6* and is representative of an enterprise-class three-tier web service. It has been used in several recent works [64], [65], [66].

Failures are classified from an end-user perspective, into three categories: *Non-silent, Non-silent-interactive*, and *Silent*. Non-silent failures are characterized by explicit error messages from the infrastructure that are visible to the end user, e.g., a servlet exception that results in an HTTP 500 status in the browser. Non-silent-interactive failures are the ones that a user notices interactively, e.g., a user searches for five valid stock quotes but is returned only three. Thus, these failures need user circumspection for detection. Silent failures disrupt web service without any manifestation to the user. An administrator is unable to detect such failures from system or application logs, until informed by an external agent. The silent failures can have serious consequences such as emptying out the portfolio of a user, or giving an incorrect view of the portfolio to the user. This study goes further and characterizes silent failures according to the type of fault. Non-silent and non-silent-interactive failures are generally logged, while silent failures are normally unplugged. Each of these classes of failures is studied by developing a fault-injection framework and by observing how DayTrader reacts to each injected fault.

The fault injection scheme presented mimics typical software bugs encountered in middle-tier of an enterprise software system. Specifically, we target bugs in business logic and control logic of an application. Business logic has functionality specific to the business (such as, buying and trading stocks in our application), while control logic handles input HTTP requests and triggers the business functionality. Our fault trigger for errors in business logic is method invocation and we inject three variants of *null-call* [18]: *Null-Return*, where a `null` is returned on a call to a business method , *Null-Object-Return* where a `null` is cast into the return type and returned, *No-Op*, where an empty object of required return type is instantiated and returned. We also inject *unchecked exceptions*, a major fault type, especially in multi-threaded architecture [67]. We find that such exceptions are quite common due to the desire (sometimes a mistaken desire) to keep the code compact enough by omitting exception handling for lots of corner cases. For errors in control logic, at compile-time, we inject *Skip-EJB*, where a call to a business method and corresponding usage of returned data is commented out in the source code. Skip-EJB represents missing programming constructs and is a dominant type of software bug studied in previous work [4].

We inject errors that mimic common programming faults. Based on these injected errors, in DayTrader, we find that 49% of errors from null-call injections, 34% of errors from unchecked exceptions and 3% of skipped statements cause silent failures. A relevant study [12] shows that 29% of failures in application servers are silent.

By observing the behavior of DayTrader to each injected error, we recommend an intuitive application-generic check to handle the *null-call* injections where a null object is returned, and find that on average this low-cost generic check is able to detect 100% of such errors for *Null-Return*, and 29% for *Null-Object-Return*. This kind of check generalizes effortlessly across applications, and this result points web service developers or distributed middleware developers toward building primitives that will make such checks easy to insert in the application. However, one kind of null-call injection, *No-Op*, where an object of the correct return type is returned, but without executing any functionality of the called function, is impossible to detect to any meaningful degree using application-generic checks.

Application-specific checks require a study of application characteristics, but here we focus on checks that can be easily deduced by standard machine learning algorithms. We use learning to induce two application-specific checks—checking the length of any web request (in terms of the number of software components) and checking the first and the last method invoked in any web request. We find, that of the different kinds of web requests, these checks perform well for those that are not data dependent, i.e., the control path is the same regardless of the parameters in the request. However, two of the ten web requests in DayTrader are data dependent and are challenging to detect and require the need for checks that have a more detailed understanding of the application [68]. A data dependent web request is either dependent on data specified by a user (in a search box) or data fetched from the database for a given user. To detect data-dependent web requests with higher accuracy, we introduce a third type of application-specific check, one that incorporates information on runtime length of a web request.

To summarize, we believe this chapter makes the following contributions:

- It provides the distribution and classification of silent, non-silent, and non-silent-interactive failures in business and control logic of a representative three-tier web service.
- It presents an application-generic consistency check that can detect a subclass of errors that cause silent failures.
- It presents low-cost application-specific consistency checks that have a high coverage for unchecked exceptions.
- It presents a low-cost application-specific consistency check that can detect data dependent web requests.

3.2 Three-tier Architecture

We build a three-tier system with clients, a middleware and a back-end database. The client layer is composed of a browser emulator, Grinder [69]. It sends web requests to the middleware that hosts an application server, Glassfish. Middleware executes application logic and builds database requests, that are sent to back-end database. Within application logic, the functionality is subdivided into presentation, control and business logic as depicted by Figure 3.1. Our system architecture, shown in Figure 3.2, implements DayTrader, a java based web service application. DayTrader follows MVC (*model-view-controller*) architecture, where *model* refers to business logic implemented by java beans in EJB container, *view* refers to presentation logic implemented by JSPs in web container and *controller* refers to control logic implemented by a controller servlet in web container. In web container, our target component for fault injection is the controller servlet.

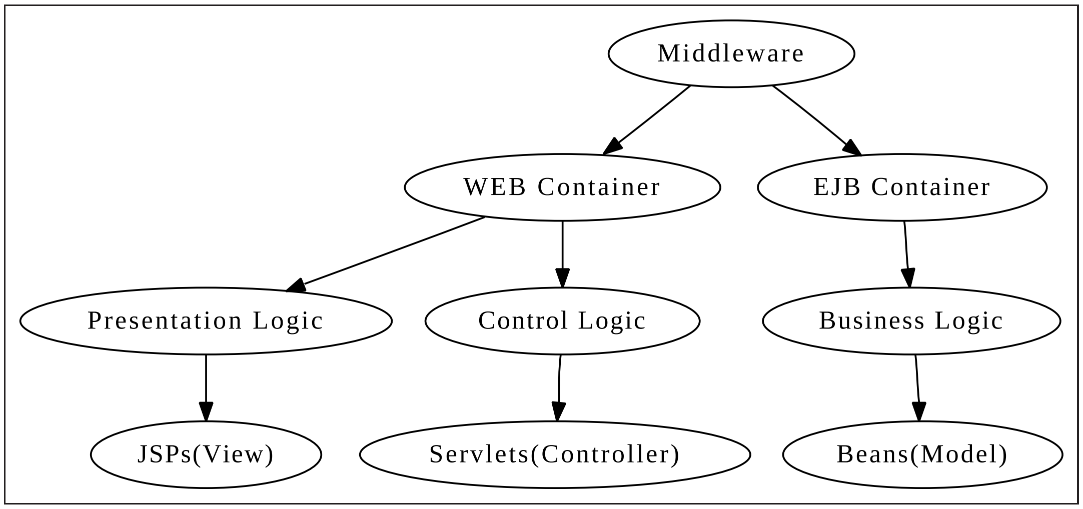


Fig. 3.1.: Middleware functionality separation

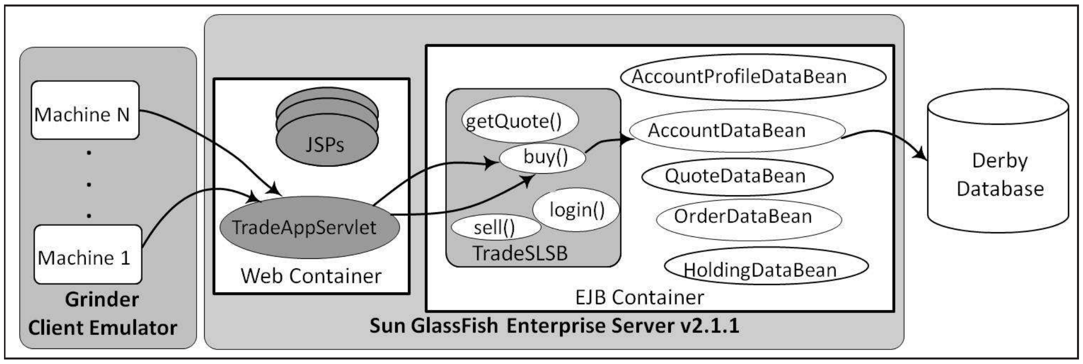


Fig. 3.2.: DayTrader System Architecture

3.3 Middleware

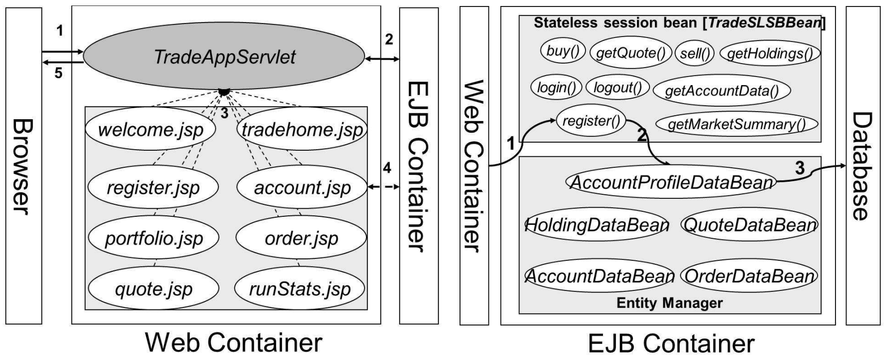
Middleware is composed of an application server, *Glassfish*, that executes application logic functions and builds corresponding database queries at runtime. The application, *DayTrader*, is deployed on top of the application server. On deployment, two major parts of application server: web container and EJB container work to receive and process HTTP web requests from clients.

Web container extends the web server functionalities and gives an environment to run servlets and JavaServer Pages (JSPs). DayTrader has a main controller servlet, *TradeAppServlet* (Figure3.3(a)) that is responsible for handling incoming HTTP requests and is the entry point for all client requests. This controller servlet implements the control logic while the supporting JSPs: *welcome.jsp, register.jsp, portfolio.jsp, quote.jsp, tradehome.jsp, account.jsp, order.jsp, runStats.jsp* implement the presentation logic. Web container executes a sequence of steps for every web request initiated by client browser: (1) *receive* an HTTP request; (2) *send* a request to the EJB in the EJB container (optionally, it may need to instantiate the EJB first); (3) *set-up* the response page template based on the request by instantiating the appropriate JSP; (4) *process* the response from the EJB and then use it to populate the response page template created by the JSP in step (3); (5) send response back to client browser.

[1] zero or more invocations

Table 3.1.: Web Request to EJB Request Mapping

Web Request	**EJB Requests**
Login	getClosedOrders->login->getAccountData->getHoldings->getMarketSummary
Home	getClosedOrders->getAccountData->getHoldings->getMarketSummary
Account	getClosedOrders->getAccountData->getAccountProfileData
Portfolio	getClosedOrders->getHoldings->getQuote∗[1]
Quotes	getClosedOrders->getQuote∗
Buy	getClosedOrders->buy->updateQuotePriceVolume
Sell	getClosedOrders->sell->updateQuotePriceVolume
Logoff	logout
Update	getClosedOrders->updateAccountProfile->getAccountData->getAccountProfileData
Register	register->login->getAccountData->getHoldings->getMarketSummary



(a) Web Functionality (b) EJB Functionality

Fig. 3.3.: DayTrader: Web and EJB container in the middleware layer

The EJB container is responsible for executing application's business logic. For DayTrader, it contains a *stateless session bean* and several *entity* beans as shown in Figure 3.3(b). The stateless session bean, *TradeSLSBBean*, maintains state only for a given business method call, e.g. `TradeSLSBBean.buy()`, and does not maintain state across business methods. Entity beans represent relational data corresponding to database tables with one-to-one mapping. Each entity bean corresponds to a table in the database whereas each instance of the entity bean corresponds to a row in that table. On receiving an EJB request from web container, EJB container executes the following sequence of steps, also shown in Figure 3.3(b): (1) *invoke* the requested EJB business method in *TradeSLSBBean* (2) *invoke* one or more entity beans (3) *send* SQL query to the database table corresponding to the entity bean to insert, update or load from back-end database.

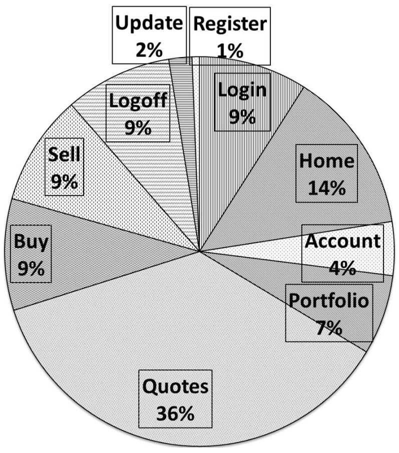
3.4 Client Requests

DayTrader customers send web requests to the middleware layer. Each web request results in several EJB method invocations in a stateless session bean, which we call *EJB requests*. An example invocation is *"Login"* web request, where a user enters her username and password, and sends the web request to the web container. The web container calls the *TradeSLSBBean.login()* in EJB container. This in turn calls several different entity bean methods according to application logic. Entity bean methods invoke *load, update* and *insert* operations on corresponding tables in the database.

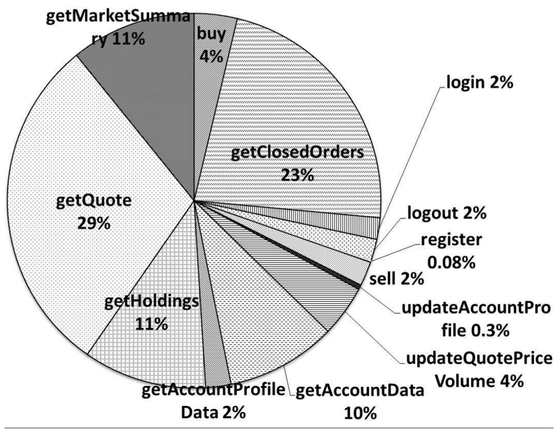
The client layer is represented by Grinder [69], a load testing distributed framework that emulates simultaneous users. We use a client workload provided by the DaCapo benchmark suite [70] for stressing DayTrader. The workload comprises of a set of stocks, users, and user sessions. Each user session starts with *"Login"*, followed by other possible web requests as defined by the application design. Each request ends with *"Logout"*. For a new user, the session starts from a *"Register"* request. A typical user session may be: *Login, Home, Portfolio, Sell, Account, Update, Quote, Buy, Logout*. We show the client web request distribution from our workload in Figure 3.4(a). Each web request results in multiple EJB requests (Table 3.1). The distribution of EJB requests is shown in Figure 3.4(b). We observe that *"Quotes"*, a read web request, is exercised with the highest percentage. This is expected of a real online user, who looks up stock quotes most of the time. Write web request like *"Buy"* and *"Sell"* have equal percentage.

3.5 Database

Database layer is composed of an independent database management system (DBMS) that stores tables specified by application designer. The DBMS, *derby* [71], is responsible for processing database requests, i.e., load, update and insert SQL queries.



(a) Web



(b) EJB

Fig. 3.4.: Web and EJB Request Distribution for 750 users

3.6 Fault Injection Methodology

In order to characterize faults, we built a fault injector that ultimately leads to silent and non-silent failures in web service based applications. Our faultload is representative of typical faults in three-tier applications. Our injected faults are similar to faults injected in [18], and we claim that they are general enough to cover a wide range of low-level programming errors. The key questions while designing our fault injector were: *What kind of faults to inject*? *Where to inject these faults*? *When to inject the faults*?

3.6.1 Types of faults injected

We inject Skip-EJB at compile-time through source code modification, while null variants and undeclared exceptions are injected at runtime. After injecting *undeclared* runtime exceptions, we study the effect of injection. We find that the practice of not handling such exceptions is quite common due to the desire (sometimes a mistaken desire) to keep the code compact enough by omitting exception handling for lots of corner cases. We decided not to inject *declared exceptions* as these exceptions are normally handled and masked by the application code itself. We also inject *Null-call*, where the called component is never executed. Specifically, we experiment with three variants of null-call, *Null-Return, Null-Object-Return, No-Op* as shown in Figure 3.5 where `foo()` calls `bar()`.

Null-Return

By injecting a Null-Return error in the EJB container, we mean that when a call to a method is made, a `null` is returned without executing the logic of the method.

```
foo(){
...
RObject x = bar();
...
}
RObject bar(){
...
return RObject;
}
```

Original Code

```
foo(){
...
//RObject x = bar();
Class RObjectClass = RObject.getClass();
RObject x = RObjectClass.cast(null);
...
}
```

Null-Object-Return Code

```
foo(){
...
//RObject x = bar();
RObject x = null;
...
}
```

Null-Return Code

```
foo(){
...
//RObject x = bar();
RObject y = new RObject();
RObject x = y;
...
}
```

No-Op Code

```
foo(){                  RObject bar(){
...                     throw new java.lang.RuntimeException();
RObject x = bar();      ...
...                     return RObject;
}                               }
```

Unchecked Exception Code

Fig. 3.5.: Example code for emulated error types in business logic

Null-Object-Return

By injecting a Null-Object-Return error in the EJB container, we mean that when a call to a method is made, then on return a `null` is cast into the return type of the method.

No-Op

By injecting a No-Op error in the EJB container, we mean that when a call to a method is made, an empty object of the correct return type is created by using the `new` operator. This empty object is returned to the caller. No-Op error can occur, e.g, when a programmer knows the return type, but incorrectly implements the functionality of the called method.

Unchecked Exceptions

On a call to a method, this throws an unchecked exception in the callee. The list of unchecked exceptions, that we inject, along with the typical reasons for their occurrence is: *ArithmeticException* due to a divide by zero, *ArrayIndexOutOfBoundsException* due to out of bound array access and *ClassCastException* due to bad casting, e.g., converting non-numeric to integer.

```
void doSell(ServletContext ctx, HttpServletRequest req,
            HttpServletResponse resp, String userID, Integer holdingID)
            throws ServletException, IOException {
String results = "";
try {
//  OrderDataBean orderData = tAction.
//       sell(userID, holdingID, TradeConfig.orderProcessingMode);
//  req.setAttribute("orderData", orderData);
    req.setAttribute("results", results);
} catch (java.lang.IllegalArgumentException e) {
        Log.error(e,"TradeServletAction.doSell(...)",
            "illegal argument, information should be in exception string",
            "user error");
} catch (Exception e) {
        throw new ServletException("TradeServletAction.doSell(...)"
            + " exception selling holding " + holdingID + " for user ="
            + userID, e);
}
  requestDispatch(ctx, req, resp, userID, TradeConfig
            .getPage(TradeConfig.ORDER_PAGE));
}
```

Listing 3.1: Example code for Skip-EJB injection

Skip-EJB

By injecting a Skip-EJB, we mean skipping a call to the EJB container from the web container, specifically, from the controller servlet. For Skip-EJB, where returned data is assigned to a variable and later used in `javax.servlet.HttpServlet-Request.setAttribute(String name, Object o)` method to pass the data to JSP, we skip both the EJB call and `javax.servlet.HttpServletRequest.-setAttribute` call in a given injection. An example of this is shown in Listing 3.1, where the injection of Skip-EJB is shown by the commented code.

3.6.2 Parameters and mechanics of fault injection

In deciding fault locations, we had two major goals: Injection should be non-intrusive, which means parsing the source code to find good locations should be avoided. Secondly, injections should mimic real programming mistakes. To achieve this, we emulate the manifestation of fault conditions in the EJB container (business logic). We emulate injection in *TradeSLSBBean's* methods in the EJB container using interceptors. In addition, our chosen error types, for injection, are representative for capturing the low-level programming mistakes as shown by previous studies.

Specifically, [12] shows that 60% of all failures are manifested as thrown exceptions. The number of previous works about *Null Pointer Exceptions* [16] is a proof of how common this type of failure is, and the motivation to inject null-call variants in our work. Programming mistakes in control logic are injected in TradeAppServlet in the web container. Next we choose fault *triggers*, i.e., when to inject a fault. For business logic, we inject whenever a call is made to invoke one of the methods of *TradeSLS-BBean* in the EJB container. This fault injection implementation is generic and can inject these faults in any typical three-tier web application. However, for control logic injection, we require source code modification.

We use *interceptors* for the EJB container and *filters* for the web container to implement the generic runtime injection and detection framework. An interceptor intercepts a call whenever a call is being made or returned from a java session bean. Similarly, a filter intercepts calls to the web container. Both interceptors and filters can run custom java code and can be configured to throw exceptions at runtime according to some failure injection rate.

3.7 Failure Classification

We classify failures with respect to their manifestation at user end. We relate this manifestation to whether the failure is originated from the application, or from the underlying container. We categorize failure manifestation in three groups: *Non-silent, Non-silent-interactive* and *Silent* as shown in Figure 3.6.

3.7.1 Non-silent

On receiving an HTTP web request from user's browser, if the response shows a blank page or an exception, we classify it as a non-silent failure. Non-silent failure can be either logged by the application or by the container as shown in Figure 3.6. A non-silent failure logged by an application comes from a `println` statement in application's logger. It shows that the developer of the application anticipated

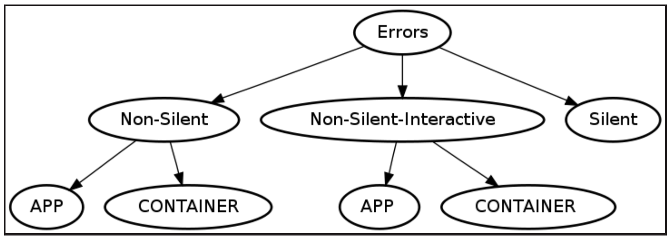


Fig. 3.6.: Failure classification

the failure. A non-silent failure that is logged by other modules in the container e.g. `javax.enterprise.system.container.web` is an example of a failure for which the application developer did not anticipate.

3.7.2 Non-silent-interactive

A non-silent-interactive failure occurs when a user interactively realizes in the same transaction that a failure has occurred. An example of a non-silent-interactive failure is a Null-Return injected in the `login()` method, which reports "Could not find account for + xxxxxx" in the browser for an existing user. Another example is when a user searches for five quotes but only three quotes show up in the result. Non-silent-interactive failures can be either logged by the application, or by the container.

3.7.3 Silent

Silent failures are those whose manifestation is never captured in logs until after multiple web requests, when a user recognizes that her transactions did not go through. An example would be when a user buys a stock, and when after a few days wants to sell it, she cannot find that stock in her portfolio. When the user contacts the administrator, the administrator is unable to determine from logs that the user purchased such a stock. This is the kind of failure mode that keeps owners of web services awake at nights and provides employment to many legions of lawyers.

3.8 Consistency Checks

3.8.1 Application Generic

Null-call Check

A rule is placed as an EJB interceptor which checks for `null` on return from every EJB request. Whenever a session bean's method is called, this rule flags if the return value is `null`. This rule has false positives as some of EJB requests like `logout()` do return `null` in normal case. We quantify the accuracy and precision of this rule.

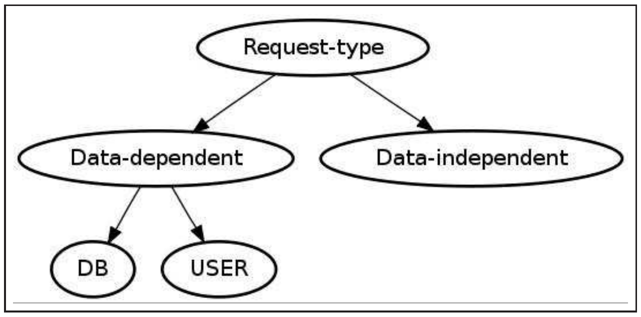


Fig. 3.7.: Web Request Classification

3.8.2 Application Specific

Call-length Check

For a given web request type, this check counts the number of corresponding EJB requests invoked, e.g., "Account" web request invokes getClosedOrders->getAccount Data->getAccountProfileData, which makes the call length of this web request three. Call length for each web request is learned from training as explained in the next sub-section. We observe that call length of web requests fall into two categories: *data-dependent* and *data-independent* as shown in Figure 3.7. Data-independent web requests have fixed call length in normal case, e.g., "Account" invokes three EJB requests. Data-dependent web requests have variable call lengths based on the entries in the database (DB) or entries entered by the user in a search box. Call lengths learned through training are used for this check; for data-dependent requests, the minimum observed call length is used. This check is triggered in a filter on return from a servlet, that handles the web request in the web container. If the observed length differs from the length inferred through training, an error is flagged.

For implementing this check, we use the `java.lang.ThreadLocal` API to monitor the web request as it passes through the three-tier system. ThreadLocal is used to create and update a local vector variable, called *call-vector*. As the call passes through to the EJB container, an interceptor adds every EJB request for the given web request to the call-vector. On return to the filter in the web container, call-vector has the given web request along with all the EJB requests invoked for that web request.

Head-tail Check

This check requires the normal tail EJB request names, that are learned in a training run as described in next sub-section. For a given web request, this check monitors head and tail EJB request, e.g., in `foo()->bar()->baz()` chain, if `foo()` and `baz()` are invoked in a web request, then the web request is assumed to successfully complete and no alarm is raised. The trigger for this check is the same as for call-length check, i.e., return from the servlet in a filter.

For implementing this check, we use the same call-vector, as in previous check. Errors like unchecked exceptions that can result in shortening of web requests will be caught by this check. The head-tail check is able to detect errors in some data dependent web requests which call-length check cannot detect, e.g., in “Portfolio” as long as the length is ≥ 2, call-length check is satisfied, but the head-tail check can catch the error if either the head or the tail does not match.

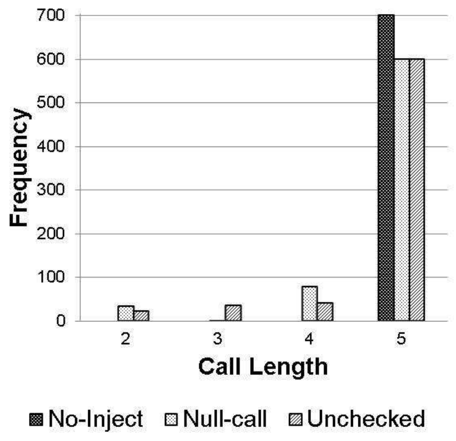
Data-dependent call-length Check

This check requires the knowledge of expected length of call-vector at runtime, in addition to the learned call-vector length. In a web request from a client, within the data-dependent call-vector, e.g, "Portfolio" (`getClosedOrders -> getHoldings -> getQuote*`) there is a constant part (`getClosedOrders -> getHoldings`) and a variable part (`getQuote*`). The constant part has information on the variable part—in this example, `getHoldings` returns a `Collection<HoldingDataBean>` containing information on how many subsequent EJB requests of type `getQuote` will be made. This check extracts the size of this `Collection<HoldingDataBean>` at runtime and matches the observed call length to expected call length (constant part + variable part) at the end of a web-request. For this check, the constant part would be learned through training as in call-length check, while variable part is extracted at runtime.

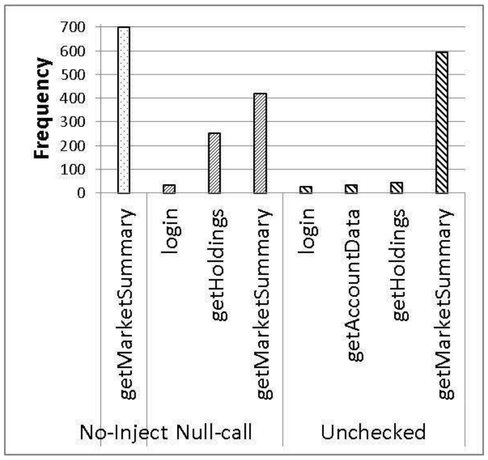
3.8.3 Learning Parameters for Application-Specific Checks

To learn the rule parameters, we run the workload in a training mode (where the buggy cases are expected to be a minority), and find frequency distribution of call length for each web request, as shown in Figure 3.8(a) for "Login". We find normal call length for a given web request, even in the presence of errors, by choosing the call length with the highest frequency, e.g., for "Login" this is 5. For data-dependent web requests, "Portfolio" and "Quotes", the call-length check is of the form "$\geq$*value*". We set the *value* equal to the minimum call length observed in training. This keeps the processing for the check simple and not too tied to the application, because we do not have to extract at runtime the part of data which determines the call length for data-dependent web requests.

For the head-tail check, we follow the same training approach as for data independent web request in the call-length rule, and find the head and the tail EJB methods that give the highest frequency. Figure 3.8(b) shows the distribution of tail EJB methods, and shows that `getMarketSummary()` is the normal tail EJB request for “Login” web request.



(a) Frequency Distribution of call length for Login web request



(b) Frequency Distribution of Tail EJB requests for Login web request

Fig. 3.8.: Distributions for learning parameters for application-specific checks

3.9 Experiments and Results

Our experiments quantify the distribution of occurrence of failure classes, for all types of injected errors. Further, we quantify the accuracy, precision and cost for our consistency checks. In our experiments, we uniformly inject in 5% of all EJB requests for business logic. For a given injection location, the injections for control logic are done for the complete duration of an experiment.

3.9.1 Testbed and Metrics

For our experiments, we use 750 concurrent users running the workload. The emulation of 750 users is achieved by Grinder framework that uses 15 client machines each emulating 50 users. Users send their web requests concurrently to a machine running the Glassfish application server, which sends the corresponding database calls to a derby database running on a third machine. The client machines are Quad-Core AMD Opteron 2.2 GHz with 8182 MB of memory. The application server and the database machines are an Intel Xeon 3.4 GHz with 1024 MB of memory.

To evaluate each of our consistency checks, we define *Accuracy* as the ratio of the number of correctly detected errors to injected errors. *Precision* is defined as the ratio of the number of correctly detected errors to the number of total detected errors. For application generic check, we give accuracy and precision at the granularity of an EJB request. For our application specific checks, we give accuracy and precision at the granularity of a web request, as they are evaluated at the end of each web request.

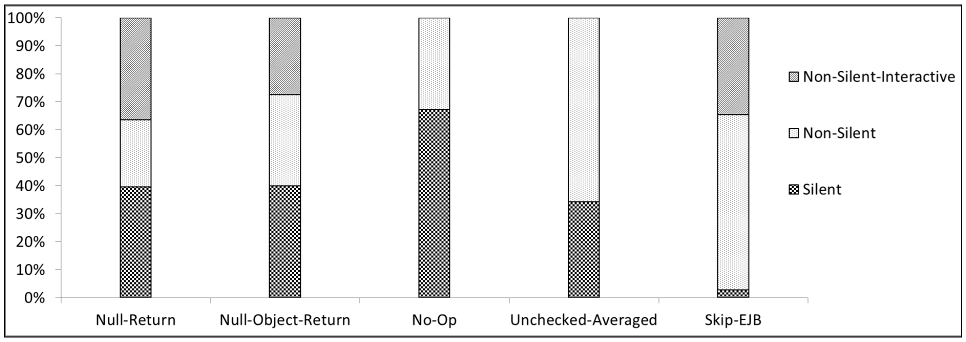


Fig. 3.9.: Distribution by Failure Class

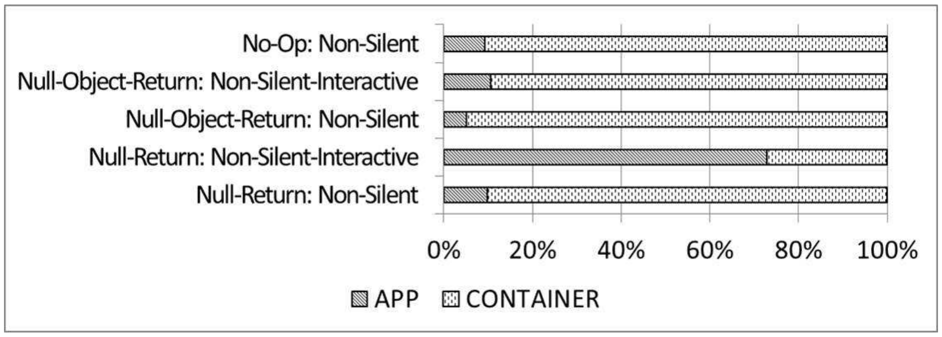


Fig. 3.10.: Failure Distribution Breakdown for the two kinds of Non-Silent Failures

3.9.2 Failure Distribution in the EJB Container

In this section, we determine the distribution of failure classes across each injection type in the EJB container. For the three kinds of unchecked exceptions, we aggregate, and give average accuracy, precision, and overhead in our results. Figure 3.9 shows that 40% of injected Null-Return errors are silent, which neither the end user nor the administrator can detect and they are not logged at the server either by the application or by the container. Non-silent failures constitute 24%, while non-silent-interactive failures are 36%. For Null-Object-Return type, silent failures remain almost the same as in Null-Return, while non-silent failures increase to 33% and non-silent-interactive decrease to 27%. This increase is because, when the callee returns, in Null-Object-Return, it returns an expected return type object, but with `null` in it. On return, caller continues with its processing, until it tries to use the returned object, and this is when a problem is detected. For No-Op errors, we observe 67% silent failures and 33% non-silent failures with no occurrence of non-silent-interactive failures. This happens, because for this type of injected error, we instantiate with `new` operator an object of the return type, and then return it to caller. For example, in `getQuote()` EJB request, the caller expects a "QuoteDataBean" object on return. When we inject Null-Return, the caller handles it in a general catch block (also logs it), and the quote result is not shown in the browser. Thus, the user interactively detects the failure. When we inject No-Op, the catch block at caller is never activated as the caller gets a "QuoteDataBean" object on return. Caller continues its buggy execution without logging anything and therefore a silent failure happens.

The objective of the next analysis is to see that for the failures (in business logic) that are logged (i.e., are non-silent), where are they logged—by the application or by the EJB container. We show error class breakdown, as logged by the application and the container in Figure 3.10 for both non-silent and non-silent-interactive failures. For Null-Return, out of all non-silent failures, 10% are logged by the application, while 90% by the container's logger. This suggests the need of null checking mechanisms at more places in application code. On the other hand, the majority of non-silent-interactive Null-Returns are detected (73%) by the application. This is understandable as an application developer would focus more on handling scenarios where a user should not see a failure interactively. The distribution for both non-silent and non-silent-interactive failures for Null-Object-Return errors is similar with the container logging majority of errors. This is conceivable for non-silent failures, but a low percentage of non-silent-interactive (11%) failures logged by application entails for extensive sanity checks on returned objects. For No-Op errors, 91% of the non-silent errors are detected by the container while the rest are detected by the application. This is because the developer did not anticipate the occurrence of No-Op errors and did not handle them in application code.

For unchecked exceptions, we see from Figure 3.9 that they are either silent or non-silent. Also, those that are silent (33%), i.e., normal operation in user's browser, are logged at the server. Majority of unchecked exceptions (67%) are non-silent, and all of them are logged by the container.

For Skip-EJB in Figure 3.9, it is observed that majority of Skip-EJB errors result in non-silent (63%) and non-silent-interactive failures (35%). A reason for this is that, whenever the presentation logic, i.e., JSPs try to access the data (using `javax.servlet.HttpServletRequest.getAttribute` method) that was suppose to be returned by the skipped EJB call, we encounter an exception that is not masked or handled by JSPs and it results in a user visible failure. However 3% of failures are still silent, indicating that presentation logic can be improved to make these silent failures non-silent. An example of the silent failure is whenever `updateAccountProfile` EJB request within "Update" web request is skipped. The returned data from `updateAccountProfile` is not subsequently used or validated (whether the update really happened in the back-end database) resulting in a silent failure.

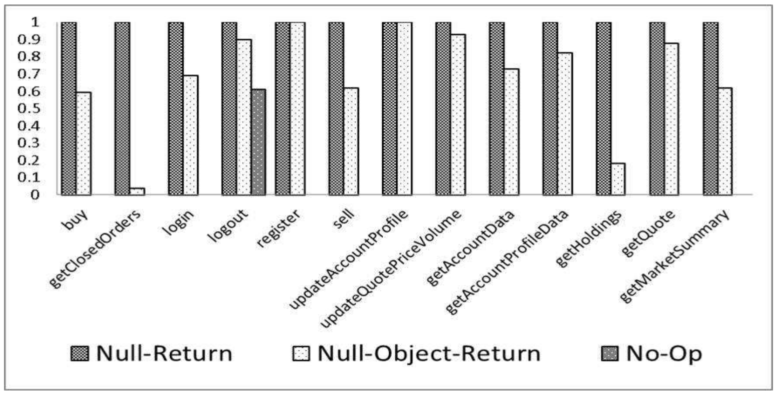
Table 3.2.: Accuracy of Null-Call Check

Type of Injection	Failure Class	Accuracy
Null-Return	Silent	100%
	Non-Silent	100%
	Non-Silent-Interactive	100%
Null-Object-Return	Silent	30%
	Non-Silent	22%
	Non-Silent-Interactive	35%
No-Op	Silent	1%
	Non-Silent	0%
	Non-Silent-Interactive	0%

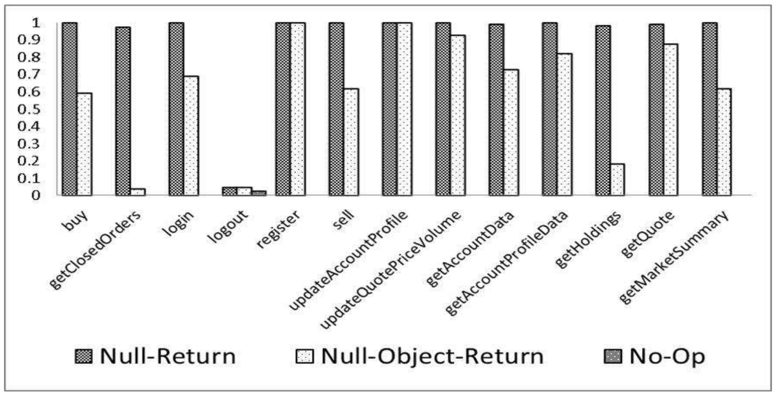
3.9.3 Application Generic Check

We implement our null-call application generic check, as defined in section 3.7.3. The accuracy and precision of this check is shown in Figure 3.11(a) and Figure 3.11(b) respectively, for each of the thirteen types of EJB requests. As we check for `null` on return from a method call, this check shows an accuracy of 100% for Null-Return.

For Null-Object-Return, accuracy is less than Null-Return. This type of error escapes the null-call check at caller because an object of the same type as expected is returned. For No-Op, no method except `logout()` is detected by null-call check. The reason for this is that, `logout()` expects a null on its return and null-call check catches it. This also raises false alarms as seen by low precision (4%) for `logout()`. In general, as we move from Null-Return to Null-Object-Return to No-Op, accuracy reduces. Table 3.2 shows accuracy of null-call check, for each failure class in each of the injection types. Unchecked exceptions are not detected at all by the null-call check.



(a) Accuracy



(b) Precision

Fig. 3.11.: Null-call Check: Performance of application-generic check that matches `null` on return

3.9.4 Application Specific Checks

We show accuracy and precision of each check in Figure 3.12 and Figure 3.13.

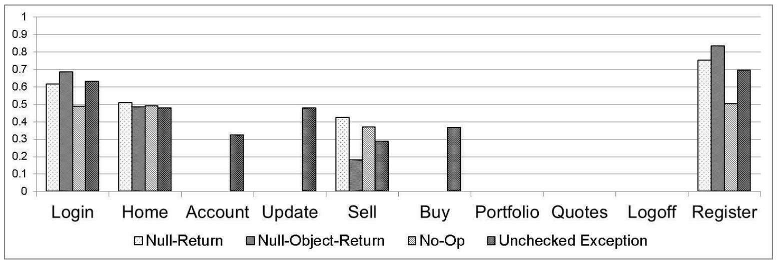
Call-length check

We observe in Figure 3.12(a), that all three variants of null-call in 6 of the 10 web requests ("Account", "Update", "Buy", "Portfolio", "Quotes", and "Logoff") are not detected by this check. From DayTrader's application code for the above

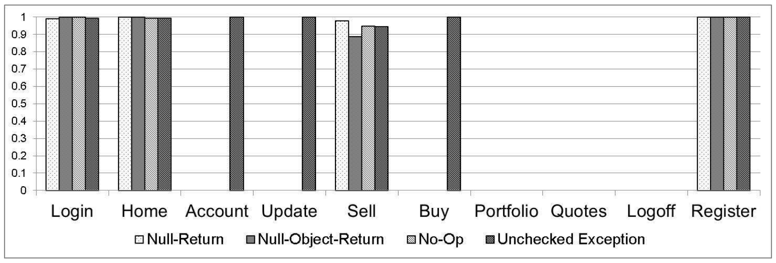
Table 3.3.: Accuracy of Call-Length Check

Type of Injection	Failure Class	Accuracy
Null-Return	Silent	0%
	Non-Silent	48%
	Non-Silent-Interactive	30%
Null-Object-Return	Silent	1%
	Non-Silent	57%
	Non-Silent-Interactive	11%
No-Op	Silent	0.2%
	Non-Silent	60%
Unchecked	Silent	0%
	Non-Silent	41%

6 web requests, we find that there is no code that checks for returned `null` and handles it. Handling `null` means that there is specialized code that executes when a `null` is returned, typically to terminate the web request. Therefore, for these web requests, subsequent EJB requests are made and the total call length is not affected. For web requests that handle a returned `null`, EJB request sequence is cut short, and an error is detected by this rule—in "Login", "Home", "Sell" and "Register". For unchecked exceptions, this check does not detect in "Portfolio", "Quotes" and "Logoff", while it detects with an average accuracy of 46% across the rest of the web request types. The rule for "Portfolio" and "Quotes" is of "$\geq$ *value*" form with learned *value* equal to 2 and 1 respectively, and also confirmed by Table 3.1. The reason for failing to detect is that the caller for the first EJB request, `getClosedOrders()` (always called since it is the first EJB request), contains a generic catch block, that handles the exception, and proceeds with the next EJB request. This means that "Quotes" would never be detected by this check as it checks for $\geq$ 1 EJB request



(a) Accuracy



(b) Precision

Fig. 3.12.: Call-Length Check: Performance of application-specific check that matches the normal length of a web request to detect

only. Also, since `getClosedOrders()` always proceeds with the next EJB request, `getHoldings()` in “Portfolio” is always invoked. This means that this check will not detect “Portfolio”, as the rule is that the EJB invocation chain length is ≥ 2. For web requests that are detected by this check, precision in general is good as shown in Figure 3.12(b).

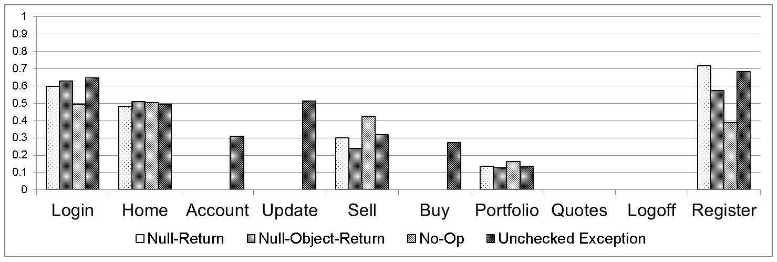
Accuracy of call-length check within each failure class for each injected error is shown in Table 3.3. It detects 57% of non-silent failures for Null-Object-Return type, which is an improvement when compared with null-call check. It also detects 60% of non-silent failures for No-Op, which the null-call check did not detect. In addition, it is able to detect 41% of unchecked non-silent failures because the length of web requests is changed.

Head-tail match

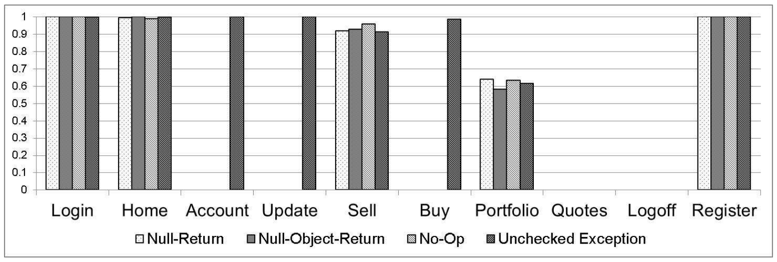
This check requires knowledge of first and last EJB request names for a given web request. The results of this check in Figure 3.13 show that, it is able to detect errors in "Portfolio", which the previous rule did not detect. To see why this happens, note that the "Portfolio" web request comprises of the EJB requests `getClosedOrders()`, `getHoldings()`, followed by zero or more invocations of `getQuote()`. The number of invocations will depend on how many stocks the user has in her portfolio. When the target of the error injection happens to be either `getClosedOrders()` or `getHoldings()`, this check detects the error because `getQuote()` never gets invoked. However, due to the nature of the workload (most users have multiple stocks in their portfolio), in a large majority of the cases (more than 80%) `getQuote()` is invoked at least once. In these cases, the head-tail match check fails. The averaged precision for "Portfolio" is 62%. For "Quotes" web request, this check does not detect for the same reason as in call-length check. The accuracy and precision results for the rest of the web requests are similar to call-length check results. Accuracy of head-tail check within each failure class for each injected error is shown in Table 3.4. The results are similar to call-length check.

Data-dependent call-length check

This check is targeted at improving the detection accuracy in data-dependent web requests, so we show the performance of data-dependent call-length check for DB-dependent web request ("Portfolio") in Figure 3.14. This check detects the unchecked exceptions with an average accuracy of 62%, an improvement over the 14% detection accuracy of the head-tail check. The improvement comes at the expense of monitoring the size of `Collection<HoldingDataBean>` which is returned by `getHoldings` EJB request. This rule is unable to detect whenever the target of injection is either `getClosedOrders` or `getHoldings` EJB request within "Portfolio" web request. This happens because total length of the web request is equal to



(a) Accuracy



(b) Precision

Fig. 3.13.: Head-tail Match: Performance of application-specific check that matches the first and the last EJB request names to detect

Table 3.4.: Accuracy of Head-Tail Check

Type of Injection	Failure Class	Accuracy
Null-Return	Silent	0%
	Non-Silent	45%
	Non-Silent-Interactive	36%
Null-Object-Return	Silent	1%
	Non-Silent	60%
	Non-Silent-Interactive	11%
No-Op	Silent	0%
	Non-Silent	68%
Unchecked	Silent	0%
	Non-Silent	42%

the expected length, i.e., 2 (`getClosedOrders` always proceeds with the next EJB request (`getHoldings`)). Also, when the target of injection is the last EJB request, i.e., `getQuote`, this rules misses to raise an alarm. This check is not able to detect the Null-Return or Null-Object-Return type of errors. The reason for that is again that the call length does not change. For No-Op, this rule shows low accuracy as it detects only when No-Op happens to be injected in `getHoldings` because the call length changes as `getHoldings` determines the number of subsequent `getQuote` EJB web requests. Call length check complements the head-tail check, which is able to detect whenever the target of injection is either `getClosedOrders` or `getH-oldings`. The overhead is the same as that in the other two application-specific checks (Figure 3.15). This is because the `java.lang.ThreadLocal` API is already monitoring the web request and the only additional work needed is to lookup the size of `Collection<HoldingDataBean>` whenever `getHoldings` returns it. Data-dependent call-length check is not able to detect the USER-dependent web

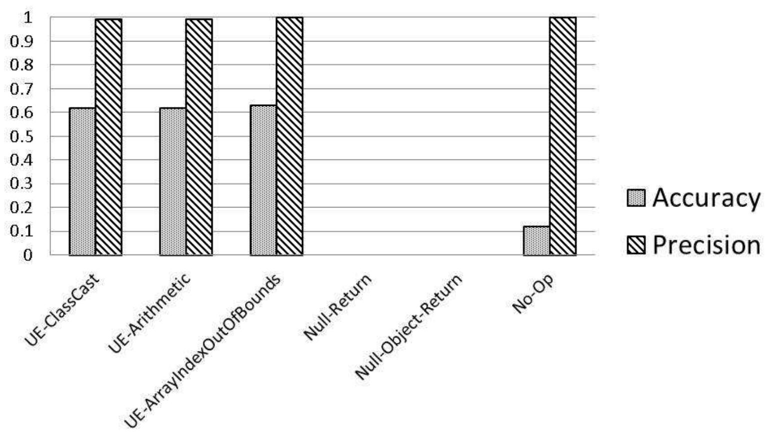


Fig. 3.14.: Data-dependent call-length check: Performance of application-specific check that matches the call length with the expected runtime call length in the data-dependent portfolio web request

requests like “Quotes” `(getClosedOrders -> getQuote*)`, as the number of `getQuote` EJB requests is determined by the user (client browser) as oppose to the backend database. For the “Quotes” web request, there is no dependence on the number of `getQuote` EJB requests invoked as in case of “Portfolio” where `getH-oldings` determines the number of `getQuote` EJB requests. To detect Skip-EJB errors, we conjecture that data-dependent call-length check would have a good detection performance as the runtime and expected call-vector length would differ in almost all the cases. We show detection accuracy and precision for Skip-EJB in Table 3.5. Data-dependent call-length check when Skip-EJB is injected performs well (100% accurate and precise for 10 out of 12 injections (see Table 3.5)). The intuition behind this is that the number of EJB requests within a web request is reduced and the runtime call length is lower than the expected call length. The `getHoldings` EJB call within “Portfolio” web request is not detected by data-dependent call-length check. This happens since `getHoldings` is not called, i.e., the rule cannot determine the expected length since it looks up the size of the collection returned by `getH-oldings`. The other EJB call, that data-dependent call-length check detects with

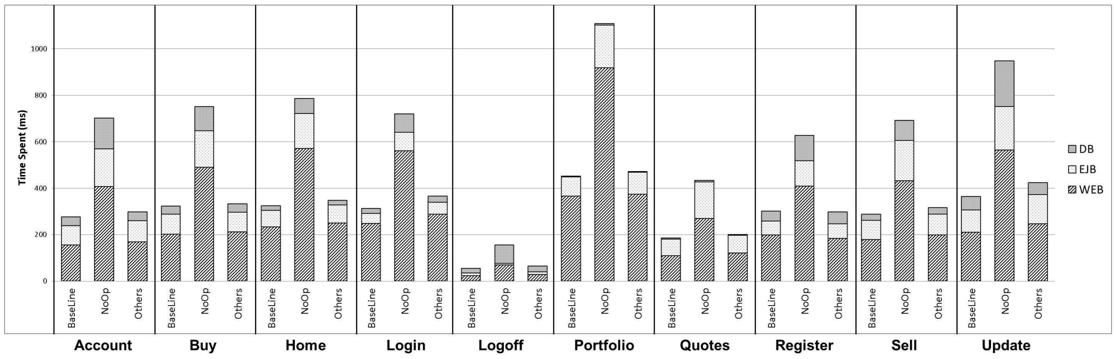


Fig. 3.15.: Overhead of application-specific checks in WEB, EJB and DB in terms of time spent per web request

a lower accuracy (97%) is `getQuote` within "Portfolio" web request. This happens whenever the preceding EJB call `getHoldings` returns the size of the collection as zero, meaning that the user does not have any stocks so, no subsequent EJB calls are made which is equivalent to skipping the EJB call (which is what was injected). Also it should be noted that, the web request "Quotes" is not injected in this type of injection. The reason behind this is that, our Skip-EJB injection is targeted towards the control logic servlet (TradeAppServlet) whereas the functionality of "Quotes" is implemented in a JSP and not the central controller servlet (TradeAppServlet). Table 3.6 summarizes the accuracy with respect to failure class and shows that all the silent failures are detected for Skip-EJB class of errors. It is observed that 93% of all Skip-EJB errors are detected.

Overhead: To understand the time overhead of each rule, we measure the time spent per web request, in the web container, the EJB container and the back-end database. The time spent in database is an overestimate as it includes network latency between the EJB container and database. We observe this latency to be small relative to time spent in the database. We compare these times to a baseline case, that has no detection mechanism. We show the overhead of call-length check in Figure 3.15. We aggregate the results for Null-Return, Null-Object-Return and unchecked types into an "Others" category. We do this as results for injections in "Others" category are

Table 3.5.: Accuracy and Precision of Data-dependent Call-length Check with Skip-EJB Injected

Web Request	**EJB Call Skipped**	**Failure Class**	**Accuracy**	**Precision**	**Logged**
Account	getAccountData	Non-Silent	100%	100%	Yes
Account	getAccountProfileData	Non-Silent	100%	100%	Yes
Update	updateAccountProfile	Silent	100%	100%	No
Buy	buy	Non-Silent-Interactive	100%	100%	No
Home	getAccountData	Non-Silent	100%	100%	Yes
Home	getHoldings	Non-Silent	100%	100%	Yes
Login	login	Non-Silent-Interactive	100%	100%	Yes
Logout	logout	Silent	100%	100%	No
Portfolio	getHoldings	Non-Silent	0%	0%	Yes
Portfolio	getQuote	Non-Silent-Interactive	97%	100%	Yes
Register	register	Non-Silent-Interactive	100%	100%	Yes
Sell	sell	Non-Silent-Interactive	100%	100%	No

Table 3.6.: Aggregated Accuracy of Data-dependent Call-length Check with Skip-EJB Injected

Failure Class	**Accuracy**
Silent	100%
Non-Silent	80%
Non-Silent-Interactive	99%

similar. We observe, that overhead is minimal for time spent in each of WEB, EJB and DB for the "Others" category. No-Op, however has considerable overhead. The time spent for No-Op in WEB is 2.5x, in EJB 1.9x and in DB 3x. The reason for this extra overhead is from the fact, that in No-Op, we instantiate a new object of the return type, which the caller accepts and processes for a longer time. The overhead results for head-tail check are similar, since it uses the same underlying monitoring.

3.10 Lessons Learned

In this section, we point out the lessons that we deduce from our injections and analysis of the resulting data. These lessons may be instructive to an application developer of web services.

(1) A caller method should flag a returned *null* as an incorrect operation and cause a failure notification to the end user. An application developer should log these cases, thus causing a non-silent error. While this does happen with some methods, further validation of checking that the returned object type is correct and its values are reasonable is not done at all in this application. Such validation should also be a part of careful application development. The first kind of validation can be created through static analysis of source code, while the latter will need application-specific checks inserted by the developer.

(2) In case of a correct return type, i.e., *Null-Object-Return*, a caller should have some sanity check, e.g. check size of returned object is greater than a threshold etc., before proceeding with the rest of its operations. This would detect the error immediately on return rather than delaying it till the null-return-object is used, thus reducing error propagation.

(3) The No-Op errors is very hard to detect in an application-generic manner because the right kind of data structure is returned, only no logic is executed in the invoked function.

(4) We find that detecting failures caused by unchecked exceptions is difficult. Therefore, for application developers aiming for high reliability of their developed web services, we would suggest putting explicit `catch` blocks specifically for each of the common unchecked exceptions. Our list of three unchecked exceptions is a good candidate set for the ones for which exception handling should be provided because we find that they occur most frequently.
(5) Silent failures constitute a significant percentage of the injected errors. Mechanisms both internal and external to the application to log possible manifestations to logs and convert some silent errors to non-silent will be very valuable.
(6) Data dependent web requests are hard to detect. We speculate they will require more sophisticated data-specific checks or stateful monitoring in the web container. They will require extracting and interpreting data in the request.
(7) For controller servlet, all the exceptions should be logged and thrown in the servlet as opposed to forwarding it to a JSP. This would help in pinpointing the failure location.
(8) Unlogged failures should be logged on detection to improve the quality of logs and the process of diagnosis.

3.11 Discussion

This chapter presents an experimental study on characterizing silent and non-silent failures in a three-tier web service application. It uses a fault-injection based scheme and classifies failures from a user's perspective. Three variants of null-call and three types of unchecked exceptions are used in the fault injection scheme to emulate faults at runtime in business logic. Additionally, skipped statements in control logic are injected at compile-time. The reaction of injecting each of these faults in DayTrader is classified into three failure classes, *Non-Silent*, where the server notifies the client of the failure, *Non-Silent-Interactive*, where a client detects by observation of an obvious failure manifestation in the same transaction, and *Silent*, where neither the client nor the administrator is able to detect. This study shows that 49% of errors averaged across three null-call variants result in silent failures, while 34% of unchecked exceptions cause silent failures. The majority of non-silent (90%) failures in Null-Return are logged by the container, while the majority of non-silent-interactive (73%) failures are logged by application. Majority of failure resulting from Skipped-EJB are either non-silent or non-silent-interactive 97%, but not all of these are logged.

A second issue investigated is the applicability of consistency checks to detect the injected faults. A simple null-call check that looks for `null` on return gives an accuracy of 100% for Null-Return in each of the three failure classes. Null-call check is an application-generic rule that can be implemented with almost zero cost. Our study also includes application-specific checks, i.e., *Call-length check*, where length of a web request is monitored , and *Head-tail check*, where the first and last EJB request name in a given web request are matched. *Data-dependent call-length check* is similar to call-length check with an additional ability to detect data-dependent web request ("Portfolio"). Data-dependent call-length check is most intrusive of all our application-specific checks as it requires knowledge of specific data structures to monitor the runtime length of a web request. Quantitatively, these three checks are able to detect between 40%-70% of the various error classes, except for the No-Op errors which go mostly undetected.

4. FAILURE CHARACTERIZATION AND DETECTION OF CONFIGURATION MISTAKES

4.1 Introduction

Downtime of software systems due to configuration problems is a critical issue for large-scale enterprise systems [72]. A configuration error in a software system can cause significant financial losses [73], [74]. A human mistake in entering a single character can result in a major outage; for example, a human error of misunderstanding the meaning of character "/" [75] caused all the search results returned by Google search engine to be falsely flagged as harmful for a period of up to 40 minutes in 2009. To understand the preponderance of configuration errors, consider that IT experts currently estimate that more than 80 percent of network outages occur due to configuration errors [76]. Arguably network configuration management is a challenging task, but it is safe to extrapolate and conclude that a significant fraction of other failures are also caused by configuration errors.

To minimize these losses, software systems should be built in a manner that is resilient to configuration errors. Further, to improve resilience, the testing of the developed software is a critical phase for all software products. Unfortunately, due to time-to-market constraints, artificial testing environments and insufficient configuration testing, bugs related to configuration problems often are pushed to users of the software.

To make matters worse, it is not clear who is responsible for a given configuration problem. Developers blame users of incorrect configuration at the user-end, while users blame the software developers. This blame game of each other leads to longer bug-resolution time. Though, user-training helps to minimize the occurrence of configuration problems, developers are expected to play a more pro-active role from the perspective of developing robust and resilient software.

In order to improve software resilience towards configuration bugs, we present a characterization-study of configuration problems in Java EE (Enterprise Edition) based application servers, GlassFish [5] and JBoss[1] [6]. Java EE application servers are complex with large number of configuration parameters. In addition, as they become an integral part of cloud-based platforms [31], both their configuration complexity and their importance are only bound to increase.

We classify configuration-bugs according to four dimensions: *Type, Time, Manifestation* and *Responsibility.* The first dimension (*Type*) considers a configuration-bug due to a problem relating to a particular parameter option, incompatibility with an external library or a missing component. The second dimension (*Time*) refers to the timepoint at which the configuration-bug is triggered, i.e., pre-boot-time, boot-time, or run-time. The third dimension (*Manifestation*) refers to whether a bug is silent or non-silent (with a log message). The fourth dimension refers to who (developer or user) is responsible for a given type of configuration bug.

Based on the characterization, we built a tool called `ConfGauge`, that injects configuration errors (misconfigurations) derived from bug databases. A fault-injection tool like `ConfGauge` will help to measure the robustness, the ability to handle unintended input, and can play an important role in the decision making process of choosing a Java EE vendor. `ConfGauge` follows a similar design-flow as `ConfErr` [26], a tool that injects human-induced configuration errors and measures the reaction of the system under test (SUT). `ConfErr` injects configuration errors derived from human psychology rather than being grounded in real bugs that have been observed in real software. Further, it only covers boot-time bugs, one point in one of four dimensions that we consider.

[1]We use the more common name, JBoss application server, as opposed to its recent new name WildFly

As opposed to an exhaustive parameter sweep, that can take a large amount of time, `ConfGauge` selectively injects a relatively small set of problems that are encountered in bug-repositories. This avoids the search-space explosion problem and reduces the running time of `ConfGauge`, thus making it an effective testing tool. `ConfGauge` injects character-strings in configuration data values that are stored as xml-attribute values. It then programmatically starts the application server, deploys an application, stops the server and analyzes the reaction to injection from the collected server logs. Ideally, a given injection should be manifested at boot-time (as oppose to waiting until a web user reports it at runtime) thus providing a direct feedback to the operator about a problem. Our experiments show that some injections result in silent manifestations after boot-time, thus requiring more intrusive configuration validation tests for Java EE servers.

`ConfGauge` specifically injects addition and removal of the forward-slash ("/") character and replacement by empty-string (""). The injections are done across both the application and the application server configuration values in GlassFish and JBoss. The results show that as high as 35% of injections result in silent failures in GlassFish while all injections result in non-silent manifestation in JBoss.

We claim the following contributions in this chapter:

- Characterization of configuration-problems for Java EE application servers, where we categorize real configuration problems across four dimensions, *Type*, *Trigger-Time*, *Manifestation* and *Responsible-party*.

- Fault-Injector for configuration-problems in Java EE application servers, where a configuration-resilience profile in terms of percentage of non-silent manifestations is provided for the two most widely used Java EE application servers.

- Recommendations for better configuration management where we provide suggestions for developers to avoid such mistakes.

4.2 Threats to Validity

All characteristic studies have limitations; for example, the number and type of bug-reports studied can be non-representative from the point-of-view of the studied problem. Studying bug-reports and classifying them is a manual process and can be subjective.

Each bug-report can state the problem with varying levels of detail for reproducing a given bug. The quality of a bug-report often depends on the expertise of the user submitting the bug-report. We do not have any way of characterizing the expertise of the user who submits a bug report. A bug report that does not have enough information to be classified as configuration-error is classified as non-configuration error. Also, an expert user might not report a given configuration problem as opposed to a novice user. Therefore, we argue that the statistics reported in this work are a lower-bound and in reality, configuration bugs likely occur with higher frequencies.

4.3 Java EE Application Server Design

Here we describe the design principles and implementation aspects of Java EE application servers that are germane to understanding the incidence of configuration errors in them. A Java EE application server provides tools and APIs to develop multi-tier enterprise applications. It is designed to implement several Java EE standards represented by Java Specification Requests (JSRs) [77]. Common modules in an application server include, web-container, ejb-container, persistence-management, command-line, GUI-based administration, messaging (JMS) etc. The functionality implemented by each module in a particular Java EE application server can vary. For example, vendor A can implement ejb-container functionality in one module, while vendor B might decide to implement it by a co-ordination of several modules.

Figure 4.1 presents basic modules and interfaces for a Java EE application server from the perspective of configuration. The arrow-heads in Figure 4.1 point to server components that require configuration data from outside. At *boot-time*, JVM parameter configurations are configured for a Java EE server. At *run-time*, an administrator can create server resources such as JDBC resources (connection pools), JMS (Java Messaging Service) resources, etc. These resources provide applications with a way to connect to other components, JDBC for database or JMS for asynchronous services (e.g., mail services). The creation and destruction of these resources, a configuration task requiring configuration parameters, is executed either by using the commmand-line interface or administration-console in the web browser. Both command-line and admin-console utilities, which come bundled with all major Java EE servers, are used to execute several other configuration tasks as well, such as deployment of a web (`*.war`) or an enterprise (`*.ear`) application.

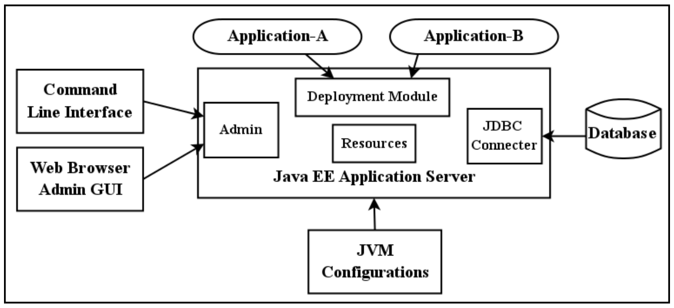


Fig. 4.1.: Java EE Server Components Interacting with Configuration Data

The components of GlassFish encountered in the studied bug-reports are: *admin, admin_gui, bean-validator, build_system, cdi, command_line_interface, configuration, deployment, distributed management, docs, ejb_container, elasticity, embedded, entity-persistence, grizzly-kernel, group_management_service, iaas, installation, jca, jdbc, jms, jsf, jts, l10n, load_balancer, logging, orb, OSGi, other, packaging, performance, rest-interface, rmi_iiop_failover, security, standalone_client, update_center, web_container, web_services* and *web_services_mgmt*. JBoss implements its server components at a coarser granularity when compared to GlassFish. The components from studied bug-reports include: *Build System, CMP service, EJB2, JMS (JBossMQ), Other, Test Suite, Web Services, Web (Tomcat) service, Weld/CDI, OSGi, Server,* *Clustering, Domain Management, Console, JSF, Security, Application Client, Naming* and *EJB*.

4.4 Classification of Configuration Problems

To characterize configuration errors, we study configuration issues in *four orthogonal dimensions: Type, Trigger-time, Manifestation, and Responsible-party*.

4.4.1 Type of errors in Java EE application servers

We are first going to define configuration and non-configuration bugs and then eliminate non-configuration bugs from further discussion.

Non-configuration

This is an error that results due to an interaction of internal components, without directly operating on data input by a human, either user or administrator. Examples of this include an incorrect returned object by a method, an incorrect method being called, a missing method call, an incorrect `cast` operation, etc. This type of error can manifest as failures called exceptions in Java programs. Typical examples of exceptions thrown at run-time due to non-configuration problems are `ClassCastException`, `ArithmeticException` etc. Other exceptions like `NullPointerException` can be thrown irrespective of whether the problem origin is non-configuration or configuration.

Configuration

This is a problem that is a result of operating on data that a human user configures in a file, in an administration console, in an environment variable or via command-line interface. The configuration errors can further be of the following types:

Parameter This a problem that occurs due to a wrong parameter type, value or format. Examples include an empty tag value in an xml tag (JBAS-929), an incorrect value passed by the user (GLASSFISH-17581), a missing forward slash "/" in resource path (JBAS-1115), etc.

Compatibility This is a problem that occurs due to an incompatibility with the environment, e.g., the JVM or a library. Examples of this type of configuration-problem are: JBAS-5275, where while reading xml-attribute values, the application server reads the values in different orders when run in different JVMs. The application server handles the values for attributes when read in forward direction, but fails to handle when read in backward direction for some JVMs. GLASSFISH-17764, where starting the cluster fails when started on the JRocket [78] JVM only. GLASSFISH-14922 where the enabling of the secure mode of the server (`enable-secure-admin`) fails on an old JVM.

Misplaced-Component A misplaced-component (henceforth called component only) is a type of configuration problem which occurs when a component or a file is missing or in the wrong location and subsequently causes an application to fail. Typically, a `ClassNotFoundException` or `NullPointerException` is thrown as a manifestation. Examples of this type of configuration-problem are: GLASSFISH-17462, where a missing file (`ssl.json`) manifests as a `NullPointerException` in server log, while the user is shown a HTTP 404 error screen in administration console. GLASSFISH-17649, where while configuring the server, `java.lang.NoClassDef-FoundError` is manifested due to a missing classpath dependency. AS7-1719, where due to a missing (`org.jgroups`) dependency, `ClassNotFoundException` is thrown when the server creates a lot of http sessions within a short time interval.

4.4.2 When does the configuration error occur?

Pre-Boot-time

If a Java EE application server or an application fails to compile or allocate pre-boot resources due to any type of a configuration problem, it is considered to occur at *pre-boot-time*.

Boot-time

If a configuration error is observed at application server startup or application deployment, then it is classified as a boot-time failure due to a configuration problem.

Run-time

An error that affects a running application on an application server while users are accessing the application is a *run-time* problem. Configuration problems that occur after an application is successfully deployed on the server fall in this category. Arguably this is the most serious of the three kinds because a user, and a potentially non-expert one, is affected by this.

4.4.3 In what manner do the configuration errors manifest?

Silent

A silent error can occur as a performance problem, data-corruption or a problem that is not directly detectable by the operator (e.g., network administrator) and is not manifested as a relevant exception in the server logs. An example of silent error is GLASSFISH-18875, where the deployment (a configuration action) of an `ear` archive is very slow and upon fixing, the deployment-time reduces from 50 minutes to 2 minutes only.

Non-Silent

A non-silent error occurs with a clear manifestation, typically as exceptions in the server logs. The silent errors are the more serious kind, due to the difficulty of detecting them.

4.4.4 Who is responsible?

We consider a configuration problem from two different perspectives of who inserted it — developer and user. A configuration bug from the user point-of view may not be a bug from developers point-of-view, for example, if the user forgot to set the value for a parameter "`host`", it is a configuration problem of the user. On the other hand, if the user set the value for "`host`" correctly, but the value was not correctly interpreted (resolving hostname to IP incorrectly etc.), then it is a bug by the developer and requires a code change. So, somewhat simplistically, we come up with the following definition of the classes.

Developer

A configuration bug that is fixed by a code change made by the developer.

User

A bug that is fixed by making the appropriate change in the configuration file, menu. etc.

4.5 Classification Methodolgy and Results

4.5.1 Methodology

We conduct our characterization study as two separate sub-studies, which we will call Study-1 and Study-2 (Table 4.1). In Study-1, we constrain the bug space from all bugs (GlassFish=15805, JBoss=4160) to bugs that have the state ‘Fixed’ for the attribute `RESOLUTION` and the state ‘Resolved’ or ‘Closed’ for the attribute `STATUS` in the time intervals [23-May-2005,16-Mar-2012] (GlassFish) and [11-Apr-2001,11-Mar-2012] (JBoss). This represents 64% of all GlassFish and 56% of all JBoss bugs. From this set, we uniformly sampled 1% of bugs giving us a total of 124 bug reports (101 (GlassFish) and 23 (JBoss)) for manual analysis.

Table 4.1.: Statistics of bug reports that form our studies

		# Issues Studied	**Time-interval**	**Versions**
GlassFish	Study-1	101	05/23/05–03/16/12	beginning till 4.0
	Study-2	132	08/03/11–07/12/12	3.1.2 (23 builds)
JBoss	Study-1	23	04/11/01–03/11/12	3, 4, 5, 6
	Study-2	25	11/01/10–09/21/12	7 (5 builds)

In Study-2, our goal is to focus in more on configuration bugs pushing automated queries to bug databases as much as we can. We constrain our search query time intervals to target one specific version of GlassFish (ver.3.1.2 - all 23 builds within it) and one specific version of JBoss (ver.7 - all 5 builds within it). We create a keyword-based search to find configuration bugs (Listing 6.4). This search query looks for keywords indicating configuration problems present in both the description and the comments section of the bug report. We believe that this will provide bug reports that indicate a mutual agreement (whether an issue is a configuration or non-configuration) between users and developers; developers typically put comments about the bug type in the comments section. This search query results in a total of 157 bug reports (132 (GlassFish) and 25 (JBoss)) that are now manually inspected in Study-2.

```
project = GLASSFISH AND (summary ~ "config* || setting* || setup
   || set-up || set up" OR description ~ "config* || setting* ||
   setup || set-up || set up") AND issuetype = Bug AND (
   resolution= Fixed OR resolution = Complete) AND CREATED >= "
   2011/08/03" and CREATED < "2012/07/17" AND (comment ~ "config*
    || setting* || setup || set-up || set up") ORDER BY created
   ASC, key DESC

project = AS7 AND (summary ~ "config* || setting* || setup || set
   -up || set up" OR description ~ "config* || setting* || setup
   || set-up || set up") AND issuetype = Bug  AND (resolution=
   Done) AND (status= Resolved or status=Closed)  AND created >=
   "2010/11/01" AND created < "2012/09/21" AND (comment ~ "config
   * || setting* || setup || set-up || set up") ORDER BY created
   ASC, key DESC
```

Listing 4.1: GlassFish and JBoss keyword-based search queries for automatically zooming into configuration bugs (with imperfect success)

4.5.2 Study-1 Results

In this subsection, we present the results of our analysis for Study-1, a total of 124 bug reports, from GlassFish and JBoss application server bug repositories [79], [80]. Table 4.2 shows the distribution of configuration and non-configuration problems. It is determined through manual analysis that more than one-third of problems in each server are configuration problems. This provides quantitative evidence for our claim that configuration errors are a significant source of concern.

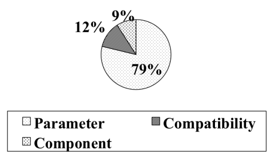
Table 4.2.: Study-1: Distribution of Configuration and Non-Configuration Problems

	GlassFish	JBoss
Configuration	33%	43%
Non-Configuration	67%	57%

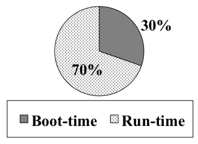
We observe that for GlassFish, nine components have more than 40% of their issues classified as configuration-problems. Among these nine components, the top two components having the most number of configuration problems are `admin` (6/14 (43%)), `admin_gui` (10/18 (56%)). This is an expected result as `admin` and `admin_gui` components in GlassFish are responsible for processing the configuration data passed in by the user. This pattern is not very visible for JBoss, mainly because of the few number (twenty-three) of bugs analyzed across many (nine) components.

Distribution by Classification Category

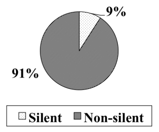
We present the percentage frequency distribution of configuration problems for GlassFish and JBoss in Figure 4.2 and Figure 4.3 respectively. For GlassFish, we observe a majority of the problems (79%) relate to configuration parameters, while 70% of the time, the configuration problem occurs at run-time. Additionally, 91% of all configuration problems are non-silent and 91% of the problems are attributed to the developer. For JBoss, only 40% of configuration problems are classified as parameter-based, while 50% are due to misplaced-components. In addition, there are no issues that occur silently, a fact which is also verified by our injection tool (Section 4.6).



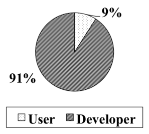
(a) Type



(b) Trigger-time

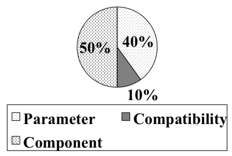


(c) Manifestation

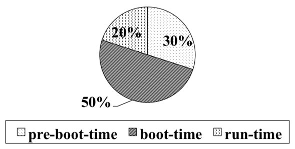


(d) Who is Responsible

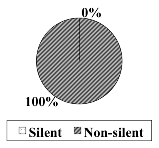
Fig. 4.2.: Study-1 GlassFish: Frequency Distribution of Configuration Problems in each Dimension (Total = 101 bugs)



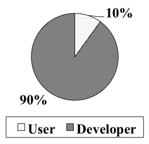
(a) Type



(b) Trigger-time



(c) Manifestation



(d) Who is Responsible

Fig. 4.3.: Study-1 JBoss: Frequency Distribution of Configuration Problems in each Dimension (Total = 23 bugs)

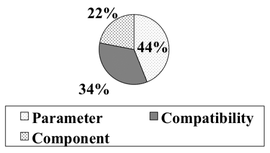
4.5.3 Study-2 Results

In this subsection, we present the results of our analysis for Study-2, a total of 157 bug reports from GlassFish and JBoss. We observe that our search (Listing 6.4) did help to find more configuration problems because we observe an increase in configuration problems from study-1 to study-2 (Table 4.3). Specifically, 33% to 62% for GlassFish and from 43% to 56% for JBoss.

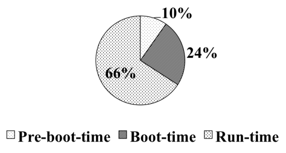
Table 4.3.: Study-2: Distribution of Configuration and Non-Configuration Problems

	GlassFish	JBoss
Configuration	62%	56%
Non-Configuration	38%	44%

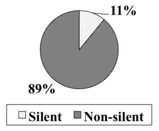
Distribution by Classification Category



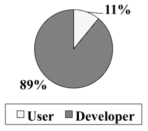
(a) Type



(b) Trigger-time



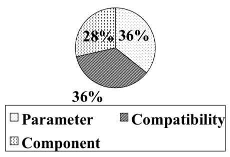
(c) Manifestation



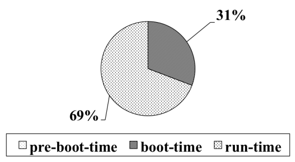
(d) Who is Responsible

Fig. 4.4.: Study-2 GlassFish: Frequency Distribution of Configuration Problems in each Dimension (Total = 132 bugs)

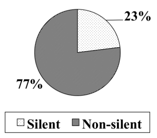
The percentage frequency distribution within each configuration problem dimension is presented in Figure 4.4 and Figure 4.5 for GlassFish and JBoss respectively. For Type (Figure 4.4.(a)), the parameter-based configuration problems for GlassFish decreases from 79% in study-1 to 44% in study-2, while compatibility issues increase from 12% to 34%. One reason for this is that study-1 analyzes issues across various



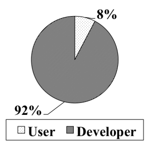
(a) Type



(b) Trigger-time



(c) Manifestation



(d) Who is Responsible

Fig. 4.5.: Study-2 JBoss: Frequency Distribution of Configuration Problems in each Dimension (Total = 25 bugs)

versions of GlassFish. Parameter names, values and format are more likely to change across versions (as opposed to within versions), thus giving a greater probability to see parameter-related issues for study-1 as opposed to study-2 (which focuses on one specific version of GlassFish). The distribution by trigger-time (Figure 4.4.(b)) is similar to study-1, where run-time configuration problems have the majority (66%) followed by boot-time (24%) problems. Also the patterns for problem-manifestation and responsible-party in GlassFish are similar to study-1.

For JBoss in study-2, we observe that all three subtypes, parameter, compatibility and misplaced-components have almost one-third share (Figure 4.5.(a)) of the configuration problems. This pattern is different from study-1's results, where parameter-based problems were dominant (50%) followed by misplaced-components (40%). The primary reason for this is that study-1 studied JBoss application server versions 3,4,5 and 6 which were very similar in their design, whereas study-2 studied version 7, which provided the same functionalities as previous versions, but had major structural differences and even required a migration guide [81] for migration to version 7 from previous versions. One well-known side-effect of this refactoring is that problems related to compatibility will happen more. This is confirmed by our analysis (an increase from 10% to 36%). For problem trigger-time in study-2, more run-time (69%) problems occur as opposed to more boot-time (50% problems in study-1). This could also be attributed to the major refactoring of JBoss version 7. The patterns for the last two studied catogories, *manifestation* and *responsible-party*, are similar for both study-2 and study-1 in JBoss.

4.6 Fault Injector

Based on insights from our characterization, we developed a configuration bug injection tool called `ConfGauge`. We believe that running `ConfGauge` with any given Java EE application server would give the user an opportunity to assess the configuration-resilience of the server. On the other hand, this can help organizations to evaluate a given application server before deploying it in a large-scale in the production environment.

`ConfGauge` follows a workflow as shown in Figure 4.6. Given a Java EE application server installation location and an application that can run on it, `ConfGauge` emulates normal server-management workflow, i.e., starts the server, deploys an application, adds and deletes server resources, and then stops the server. While executing this workflow, `ConfGauge` injects configuration problems. Currently `ConfGauge` injects parameter-based configuration problems at boot-time.

A fault injector should know the answers to the following questions:

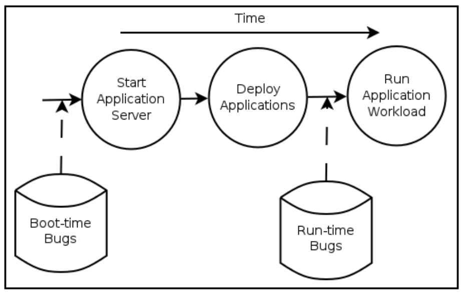


Fig. 4.6.: WorkFlow for Time of Injection

What to inject? `ConfGauge` injects character-string-based misconfigurations that were observed while conducting our characterization study. For this work, we choose to inject the character "/" and empty-string (""). Forward-slash character has been the cause of several parameter-related configuration problems in Java EE application servers [82] (JBoss), [83] (Geronimo), [84] [85] (GlassFish), [86] (TomEE) as well as in general Java programs [87] (RestEasy), [88] (OpenSSO), [89] (SailFin), [90] (JavaServerFaces) etc. Configuration problems related to empty-string for server attribute-values have also been reported in both GlassFish [91] and JBoss [92].

Where to inject? `ConfGauge` injects its misconfigurations in the `xml` configuration files that are maintained by the Java EE application servers for configuration management. Specifically, it injects in `xml-attribute` values within `domain.xml` (GlassFish) or `standalone-full.xml` (JBoss), `web.xml`, and `persistence.xml` files. `domain.xml` in GlassFish and its equivalent `standalone-full.xml` in JBoss maintain the major server configurations that are read during the boot-up process. The last two are configuration parameters of the application (SpecJEnterprise2010 in our case). `web.xml` is used to configure the application server resources used by a given deployed application, e.g., how URLs map to servlets, authentication information etc. `persistence.xml` is used to configure settings for the connection to back-end database, e.g, the driver-name, persistence-provider-name, etc.

When to inject? `ConfGauge` injects its misconfigurations at boot-time. Boot-time can be either application-server start or an application deploy operation.

How to inject? `ConfGauge` injects its boot-time misconfigurations by parsing the `xml` configuration files, mutating only one attribute-value at a time and writing the mutated file before the application-server or the application is started. For each injected string, the application server is programmatically (using CARGO API [93]) started, an application is deployed, the application server is stopped and the server log file is saved for analysis. This process is repeated for all injections. We define three injection-operators, *Add*, *Remove* and *Replace*. *Add* means that we append the injected character to a similar, already existing character in the attribute-value-string. *Remove* means that we remove the injected character from the correct attribute-value-string. *Replace* means that we replace the correct attribute-value with a different string, i.e., either "/" or the empty string. These operators were defined based on evidence from bug repositories [85], [82], [91]. An example of a sample injection for each of the *Add*, *Remove* and *Replace* operating on `domain.xml` is shown in Table 4.4.

Table 4.4.: An example of mutated configuration values in `domain.xml` in GlassFish

Mutation Operator	**Original Value**	**Mutated-Value**
Add	<jdbc-resource jndi-name="jdbc/__default" pool-name="DerbyPool"/>	<jdbc-resource jndi-name="jdbc//__default" pool-name="DerbyPool"/>
Remove	<jdbc-resource jndi-name="jdbc/__default" pool-name="DerbyPool"/>	<jdbc-resource jndi-name="jdbc__default" pool-name="DerbyPool"/>
Replace	<property name="URL" value="jdbc:mysql://hostname:3306 /specdb"/>	<property name="URL" value=""/>

Target Application: SPECjEnterprise2010 We need an application running on the Java EE application servers to stress it with a realistic workload. We deploy SPECjEnterprise2010 [94] on GlassFish and JBoss.

SPECjEnterprise2010 is an industry-standard Java Enterprise Edition (EE) benchmark that emulates a complete enterprise system. It describes an end-to-end business process and emulates an automobile dealership, manufacturing, supply chain management and order/inventory system. The set of functionalities that this benchmark includes are: *Dynamic Web page generation, Web Service based interactions, Trans-*

actional and Distributed components, Messaging and asynchronous task management and Multiple company service providers with multi-site servers. The system architecture corresponds to a 3-tier architecture, i.e., client, middleware and database.

The clients of SPECjEnterprise2010 are automobile dealers, who use a web based user interface to access the application. The interface allows customers (car dealers) to login to their accounts, keep track of dealership inventory, sell automobiles, manage a shopping cart and purchase automobiles.

4.7 Fault Injection Results

For one injection, `ConfGauge` mutates one string in one of the xml files, starts the application server, deploys the SPECjEnterprise2010 application, stops the application server and collects the server logs for analyzing the effect of injection. The number of non-silent manifestations for boot-time injections are presented in Table 4.5.

For a total of 132 injections of "/" character in GlassFish, both the server configurations in `domain.xml` and the application configurations in `web.xml` have silent failures for the majority of injections. Specifically, only 24.2% of remove-operator-based injections and 22% of add-operator-based injections result in non-silent manifestations with exceptions thrown in server log file. On the other hand, all the JBoss injections result in non-silent manifestations. For empty-string ("") injections, 75.9% in GlassFish and 100% in JBoss are non-silent. These results mean that in GlassFish, the throwing of exceptions is delayed until the end-users (users of SPECjEnterprise2010) try to access a particular resource. As an operator of the application server and applications, this would result in a higher frequency of issues filed by the users, a fact that is verified by observing (Figure 4.7) the total number of issues filed for each of the projects. We see qualitatively that the number of configuration-related issues follow a similar pattern.

Table 4.5.: Number of non-silent manifestations for boot-time injections in GlassFish and JBoss

GlassFish						**JBoss**			
		Application		**Server**		**Application**		**Server**	
Operator	**Injected character**	**web.xml**	**persistence.xml**	**domain.xml**	**Total**	**web.xml**	**persistence.xml**	**standalone-full.xml**	**Total**
remove	/	2/35 (5.7 %)	4/4 (100 %)	26/93 (28.0 %)	32/132 (24.2 %)	35/35 (100 %)	4/4 (100 %)	15/15 (100.0 %)	54/54 (100 %)
add	/	0/35 (0 %)	4/4 (100 %)	25/93 (26.9 %)	29/132 (22.0 %)	35/35 (100 %)	4/4 (100 %)	15/15 (100.0 %)	54/54 (100 %)
replace	“”	-	-	460/606 (75.9 %)	460/606 (75.9 %)	-	-	397/397 (100 %)	397/397 (100 %)

Table 4.6.: Detailed Exception Distriubtion in JBoss when “/” was Injected

File	**Remove**			**Add**		
	Exception	**#**	**Percentage**	**Exception**	**#**	**Percentage**
standart-full.xml	org.jboss.msc.service.StartException	11	73	java.lang.NullPointerException	1	6.6
	org.jboss.as.controller.persistence.ConfigurationPersistenceException	4	27	org.jboss.msc.service.StartException	13	87
				java.lang.IllegalArgumentException	1	6.6
web.xml	java.lang.IllegalArgumentException	16	46	org.jboss.msc.service.StartException	35	100
	org.jboss.msc.service.StartException	19	54			
persistence.xml	org.jboss.msc.service.StartException	4	100	org.jboss.msc.service.StartException	4	100

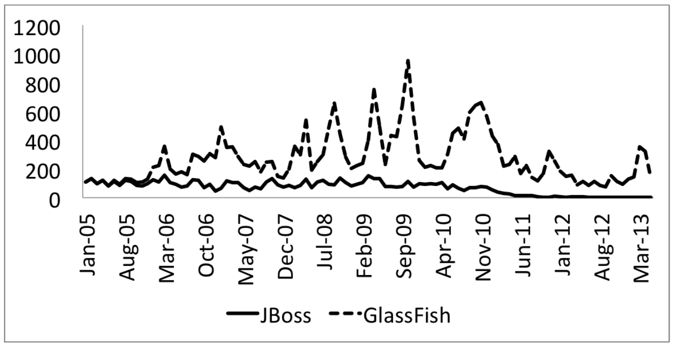


Fig. 4.7.: Number of issues filed to GlassFish and JBoss (vers. 3-6) bug repositories per month

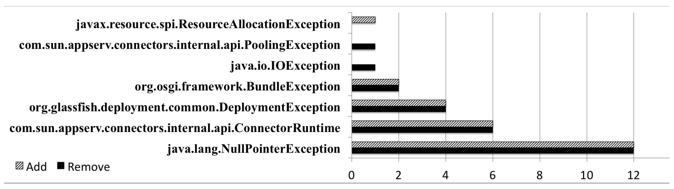


Fig. 4.8.: Frequency Distribution of Exceptions for non-silent manifestations for GlassFish in `domain.xml` with "/" injection

4.7.1 GlassFish Injection Results

Forward-slash Injection

For `domain.xml` in GlassFish, the non-silent exceptions distribution when forward-slash is injected is presented in Figure 4.8. It shows that `NullPointerException` occurs most frequently. For these `NullPointerExceptions`, the root-cause is due to the injection at two separate locations related to JMS configurations: (1). Injection in the attribute `pool-name` within xml-element `connector-resource`, where the pool-name is changed from "jms/value" to "jms//value", (2). Injection in the attribute `name` within the xml-element `connector-connection-pool`, where the name is changed from "jms/value" to "jms//value". Both of these attribute values should match for the correct operation, thus avoiding `NullPointerExcep-`

`tion`. Given that the standard way to delimit is using a single "/", we recommend a deployment verifier that would verify and favorably replace the extra forward-slash introduced, due the configuration error thus automatically fixing the issue. Note that GlassFish does have a verifier tool called `verify-domain-xml`, which is claimed to be much more than an XML syntax verifier. Its documentation [95] states: "Rules and interdependencies between various elements in the deployment descriptors are verified". We tested it with the injected error in `domain.xml`, and it did not catch the injected forward-slash configuration problem.

For most of the injections (94.3% with remove and 100% with add operator) in `web.xml`, the server log did not show any exceptions at boot-time. This fact where exceptions are delayed until the end-user accesses the application through the web browser is a point of concern.

```
<servlet>
        <servlet-name>purchase</servlet-name>
        <jsp-file>//purchase.jsp</jsp-file>
</servlet>
<servlet>
        <servlet-name>shoppingcart</servlet-name>
        <jsp-file>_shoppingcart.jsp</jsp-file>
</servlet>
```

Listing 4.2: Two sample injections (underlined) using *add* and *remove* operators in `web.xml` that were not detected at boot-time in GlassFish

Empty-String Injection

Out of 606 empty-string injections in `domain.xml` within GlassFish, 69.5% (421) of them resulted in the failure of server boot-up process while 6.4% (39) resulted in exceptions in server log. This means that there were still a significant number (146 (24.1%)) of empty-string-based problems that were silent at boot-time.

4.7.2 JBoss Injection Results

Forward-slash Injection

For JBoss, we observe in Table 4.6 that for all injected file types and both mutation operators, `org.jboss.msc.service.StartException` is thrown 80% (86/108) of the time. The `StartException` is a JBoss-defined `Exception` that is thrown when a service fails to start. This high-level exception in JBoss provides enough details in server log to diagnose the root-cause (injection location). As an example, consider the effect of applying the *remove* operator in the JMS queue entry name to change it from `"java:jboss/jms/LoaderQueue"` to `"java:jbossjms/LoaderQueue"` in `standalone-full.xml`. The corresponding server log entry is shown in Listing 4.3, where the corresponding problem (missing forward-slash) is observable (manifestation underlined in Listing 4.3). Another example with injection in `web.xml` is the second injection in Listing 4.2, which GlassFish did not detect whereas JBoss gave a clear manifestation of missing "/" as shown underlined in Listing 4.4. These are desirable manifestations and show that JBoss is robust enough to diagnose the injected configuration problems, both at the application and the server level. After diagnosis, a subsequent automated replacement or deletion (in the case of an extra character) of the forward-slash is still a desirable but missing feature.

```
15:58:44,916 ERROR [org.jboss.msc.service.fail] (ServerService Thread Pool -- 58)
    MSC00001: Failed to start service jboss.messaging.default.jms.queue.LoaderQueue:
    org.jboss.msc.service.StartException in service jboss.messaging.default.jms.queue
    .LoaderQueue: JBAS011639: Failed to create queue
Caused by: java.lang.RuntimeException: JBAS011846:
    Illegal context in name: java:jbossjms/LoaderQueue
```

Listing 4.3: Log entry after applying remove-operator on "java:jboss/jms/LoaderQueue" in JBoss (`standalone-full.xml`) to remove "/"

```
15:07:43,301 ERROR [org.apache.catalina.core] (ServerService Thread Pool -- 64)
    JBWEB001097: Error starting context /specj-specj: java.lang.
    IllegalArgumentException: JBWEB000273: JSP file shoppingcart.jsp must start with a "/"
```

Listing 4.4: Log entry after applying remove-operator on “/shoppingcart” in SPECjEnterprise2010 (`web.xml`) to remove first forward-slash

Empty-String Injection

All empty-string injections in `standalone-full.xml` within JBoss result in the failure of server boot-up process. The corresponding log entry, `''Server boot has failed in an unrecoverable manner"`, shows that all misconfigurations need to be fixed before a successful boot-up operation. This is a desirable outcome.

4.8 Observations and Recommendations

In this section, we highlight some of the high-level observations from our study and recommendations for improving the management of Java EE software systems.

4.8.1 Observations on configuration problem patterns

1. Our observations reveal that more than one-third of problems reported by users are configuration-related. These problems occur more in components that are at the interface between the user and the program.

2. Inter-version and intra-version configuration problems tend to have different characteristics. Inter-version problems are majorly parameter-related while intra-version problems have an almost equal share for each of *parameter*, *compatibility* and *missing-component* related issues.

3. Code refactoring or re-implementation results in an increased number of *compatibility* issues.

4. Parameter-based fault-injection, as done in our tool `ConfGauge`, based on bug reports, can provide a measure of robustness of the software.

5. Silent configuration-related failures (35% in GlassFish) found by `ConfGauge` provide evidence of inadequate configuration validation. Silent failures should be made non-silent via more intrusive validation checks.

4.8.2 Recommendations for better configuration management

1. **Automated Fixing of Parameter Values:** Today, incorrect parameter values specified by users of application servers are fixed by users themselves. They primarily go through the manual process of *searching* for similar problems, *communicating* with developers, *trial-and-error* of different possible values and even *applying* patches. We suggest to developers of application servers to take a more pro-active role and dynamically fix configuration mistakes that users make. For example, given that a file name cannot contain the character "/" and many configurations have a well-defined format, e.g., token separation of single "/" character, any extra "/" characters should automatically be removed at boot-time with a notice to the user. This process requires configuration validation, a step that is not mandatory according to Java specifications and hence not pursued by many vendors. To strongly enforce it, there is a need to push for this in the future versions of Java EE standards.

2. **Efficient Bug Repository Maintenance:** Operating a bug repository where it is easier for users to search and fix problems helps to reduce bug-resolution time and frequency of issues. From our experience while studying bug-reports in GlassFish and JBoss, we observed some desirable features in the bug-tracking systems which we recommend: (1). JBoss employs a duplicate bug-detection tool called `Suggestimate` [96]. `Suggestimate` clusters similar issues together. Also, as users are typing in their bug-reports, similar issues based on

the keywords used are presented. This helps to reduce average number of issues filed by users. (2). For a majority of fixed bugs, JBoss cross-references the bug-fixes giving an option for the developers and users to view the difference to previous version, thus providing an opportunity to better understand the fix.

4.9 Discussion

This chapter presented an empirical study for characterization of configuration problems in Java EE servers, followed by a fault-injection based evaluation. We presented the failure characterization by studying bug reports from the two most popular Java application servers — GlassFish and JBoss. The studied dimensions for each configuration-bug were *problem-type (parameter, compatibility, missing-component), problem-time (pre-boot-time, boot-time, run-time), problem-manifestation (silent, non-silent)*, and *problem-culprit (developer, user).* The key findings in our characterization are: (1). More than one-third of problems reported by users are configuration-related implying that configuration-perspective in software development requires considerable attention. (2). Inter-version problems are majorly parameter related, thus requiring more consistency in configuration-design from version-to-version. (3). Intra-version configuration-problems have an almost equal share for each of *parameter, compatibility, missing-component* subtypes. (4). Code refactoring or re-implementation results in an increased number of *compatibility* issues.

The second part of this chapter presented `ConfGauge`, a tool that injects misconfigurations based on evidence from bug-repositories. Through the injection process, `ConfGauge` automates the configuration-management life-cycle, i.e., start-server, make configuration changes, deploy applications, undeploy applications and stop-server. It subsequently analyzes the reaction of the server to each of the injections by determining the type of manifestation (silent or non-silent) from server logs. JBoss performs better than GlassFish with all of the injections giving a non-silent failure as opposed to only 65% non-silent failures in GlassFish. Based on the results and our analysis, we suggest automated fixing of parameter values.

5. DETECTION OF DUPLICATE REQUESTS CAUSING PERFORMANCE PROBLEMS

5.1 Introduction

We live in a world where web page views are worth bragging rights and cold hard cash. Big web sites tout the number of page views. Also, provisioning of servers for hosting web content is done by monitoring the number of page requests. If the number trends high, a decision is made to provision more servers. This is typically done through human deliberation, but in some leading edge deployments, through automatic means as well [31, 97]. Web page views are monetized through various means, such as, increasing the amount charged to an advertiser for ad placement on the page, increasing the number of advertisements shown for the page, and so on. Web page views are calculated, simply put, by tracking the number of requests sent by client web browsers for that page. Could this simple but fundamental view of the web world be afflicted by a little known problem?

The affliction of duplicated web requests Yes indeed! *The affliction is duplicate web requests.* In this, the client web browser sends two requests for the same web page, the second being a redundant duplicate request. This affliction does not affect poorly run web sites alone. It afflicts two of the top 10 most visited sites — CNN and YouTube [10]. On the academic side, we found that it affects HUBzero, a widely used open source software platform (originating from Purdue) for building powerful Web sites that support scientific discovery, learning, and collaboration [98]. We discovered this latter case while dealing with problem tickets submitted to a collaboration portal instantiated from HUBzero, called NEEShub, and this discovery triggered the line of work that we report here. The duplicated request issue causes two obvious problems. *First*, there is a spike in the traffic directed to the web server, caused by fruitless requests. For a web site that receives lots of views, this doubling can have a crippling effect due to increasing the network as well as the computational load. The increase in computational load becomes significant due to the fact that many content-rich pages today are dynamically generated by running complex, demanding scripts at the server end. *Second*, there is the potential problem of user state corruption. If the web site is tracking state, either by cookies or in another way, there is the possibility of corrupting this data.

Why do duplicate web requests happen? There are two root causes for the problem of duplicate web requests, which have been separately pointed out in many developer forums and blog posts [7–9]. The first cause is the incorrect way in which browsers handle missing component names, or empty tags, such as, `<img src="">`, `<script src="">`, and `<link href="">`. Equivalently, this could be caused by JavaScript which dynamically sets the `src` property on either a newly created image or an existing one:

```
var img = new Image();
img.src = ""; // More realistically, the RHS will be some code
    that will resolve to the empty string
```

Listing 5.1: Sample code that results in duplicate request

The most readable and comprehensive treatment of this first cause can be found in [7]. We will refer to this first root cause as *missing resource cause*. The second cause is the same Javascript being included in the page twice, or more number of times [10]. This is the root cause behind the duplicate web requests in CNN and YouTube. Two main factors increase the odds of a script being duplicated in a single web page: team size and number of scripts. It takes a significant amount of resources to develop a web site, especially if it is a top destination. In addition to the core team building the site, other teams contribute to the HTML in the page for things such as advertising, branding, and data feeds. With so many people from different teams adding HTML to the page, it is easy to imagine how the same script could be added twice, e.g., CNN and YouTube's main pages have 11 and 7 scripts respectively. A plausible scenario is two developers are contributing JavaScript code that requires manipulating cookies, so each of them includes the companys *cookies.js* script. Both developers are unaware that the other has already added the script to the page. This increases the time for the page to load along with the duplicate web requests problem. We will refer to this second root cause as *duplicate script cause*.

How to fix the problem? The "missing resource cause" happens because the HTML specification, version 4 [99][1] is silent on this seemingly esoteric aspect. Even though the specification indicates that the `src` attribute should contain a Uniform Resource Identifier (URI), it fails to define the behavior when src does not contain a URI. Consequently, different browsers behave in different ways. For example, Internet Explorer (IE) sends the duplicate request to the directory of the page rather than the page itself, while Firefox and Chrome send the duplicate request to the page itself. Further, the behavior of different browsers for handling different missing resources is different, e.g. , IE does not initiate a duplicate request with missing `script` while Firefox and Chrome do. The overall approach to handling this could be to write server-side code that will catch a similar request arising close in time to the original request and correlated with finding a missing URI in a tag. However, due to the differences in browser behaviors and for different tags, this would lead to ungainly code, with case statements for a large number of different cases. An indirect evidence comes from the fact that though this problem has been known for a while (since at least 2009), this solution is seldom deployed. The "duplicate script cause" of course has no easy solution available currently. The solution is mainly process-based — enabling better communication and coordination between developers writing or using scripts to create web pages. Thus, hopefully, the situation where two different developers include the same script on the same page or, more subtly, incorporate different but overlapping scripts on the same page, can be avoided.

[1]HTML4 is the latest version of the specification, except for a W3C "Candidate Recommendation" for HTML5 dated 04 February, 2014.

Our solution approach In this , we present a general-purpose solution to the above problem, in a system called GRIFFIN[2]. By "general-purpose", we mean that the solution applies unmodified to all kinds of resources and browsers. The solution has at its heart the observation that the duplicate web requests cause a repeated signal, for some definition of "signal". The signal should be defined such that it can be easily traced in a production web server, without impacting computation or storage resources and without needing specialized code insertion. We find that the *function call depth* is the signal that satisfies these conditions, while preserving enough fidelity that the repeated sequence can be easily and automatically discerned. To automatically discern the repeated pattern, we use the simple-to-calculate autocorrelation function for the signal and at a lag, equal to the size of the web request (in terms of number of HTTP commands), GRIFFIN sees a spike in autocorrelation which it uses to flag the detection.

When tested over a wide range of buggy and non-buggy behavior, we find that GRIFFIN performs well with respect to both the detection and the false positive. For example, we evaluate GRIFFIN on the production NEEShub web portal at Purdue, which is the portal created out of an ongoing NSF center called NEES, for providing computational tools and data upload/download facilities to earthquake scientists and engineers all over the US [100]. NEEShub has been operational since 2009 and has had 105,000 users over the last 12 month period. So a problem in it cannot be easily dismissed as a corner case in an obscure site. We find that GRIFFIN has no false positive and an 78% detection accuracy. To make GRIFFIN feasible in real production settings, we adopt a mix of synchronous and asynchronous approaches, both without modifying the application's source code, or even needing access to the source code. Synchronously we capture the call stack depth, using a built-in functionality, in the tracing tool called SYSTEMTAP. The SYSTEMTAP tool is highly efficient and has already been used in prior efforts for analyzing the properties and behavior of

[2]GRIFFIN is a mythical creature with the front legs, wings, and head of a giant eagle, and the body, hind legs, and tail of a lion. It is often used to guard treasures.

software systems [101]. Then, asynchronously, GRIFFIN calculates the autocorrelation function for various lags, filters the values, and flags a detection when the value exceeds a threshold. In addition to detection, GRIFFIN also provides some diagnostic insight, *i.e.*, gives an idea of the module where the root cause lies. It does this by inserting probes through SYSTEMTAP, determining the lag at which the autocorrlation function has a peak, and correlating the two to determine the suspect module.

Our contributions in this work can be summarized as follows.

1. We provide automatic detection of duplicate web requests in a manner that is generic to any web server and works across different web clients and different root causes of the problem.

2. We develop code to extract the signal from amidst a plethora of tracing data. We zoom in on the right signal to use through insights born out of troubleshooting web servers. Our method has in fact been merged within the SYSTEMTAP code repository.

3. We evaluate our scheme with a popular production web portal for science. We report on the performance overhead as well as error coverage from our evaluation.

5.2 Example Bug Case

The manifestation of the duplicate web request can be silent or non-silent. Silent implies there is no visual indication of the problem at the client web browser, while in the non-silent case, there is such an indication. With the NEEShub home page, we observed a non-silent manifestation whereby multiple images are not shown as depicted in Figure 5.1. An example of the silent case is that the browser after downloading the multiple Javascripts, generates duplicate web request from the multiple Javascripts.

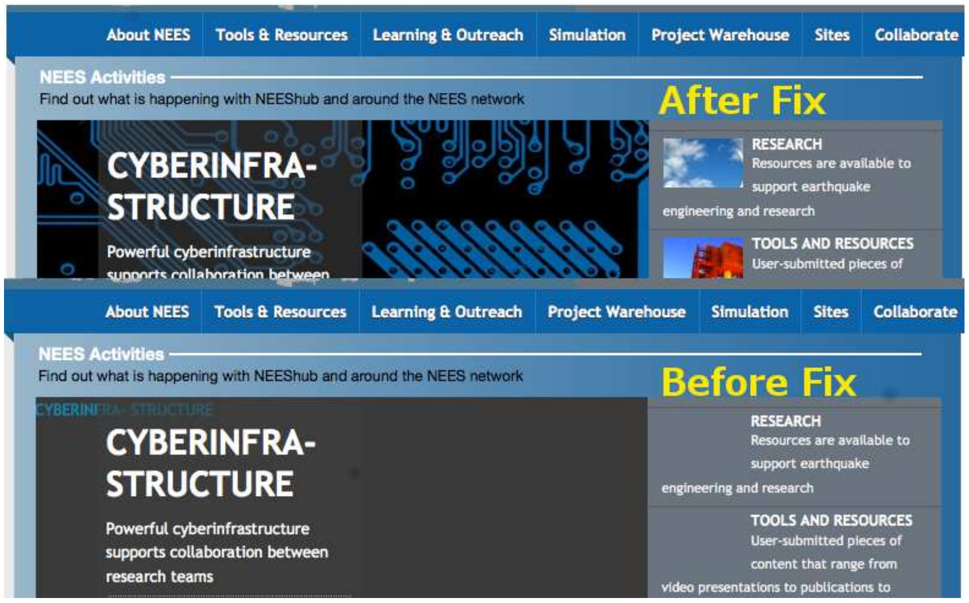

About NEES
Tools & Resources
Learning & Outreach
Simulation
Project Warehouse
Sites
Collaborate
NEES Activities
Find out what is happening with NEEShub and around the NEES network
After Fix
CYBERINFRA-STRUCTURE
Powerful cyberinfrastructure supports collaboration between
RESEARCH
Resources are available to support earthquake engineering and research
TOOLS AND RESOURCES
User-submitted pieces of
Before Fix
CYBERINFRA- STRUCTURE
Powerful cyberinfrastructure supports collaboration between research teams
content that range from video presentations to publications to

Fig. 5.1.: Duplicate bug-manifestation (with missing images) before and after the fix

Here we present a bug-case that was observed for the beta release of the main web portal of our NSF center called NEEScomm, meant for providing a cyberinfrastructure for earthquake engineers and scientists throughout the US `www.nees.org`. Griffin was able to detect it before the code update made it to the production site, and thus avoided the duplicate request problem. On accessing the homepage, the images that appear as part of background were missing (Figure 5.1). Figure 5.2 presents the code modifications that fixed the problem (no duplicate requests seen from client). In Figure 5.2, `$slide->mainImage` variable does not resolves to the image `XYZ.jpg` location. Instead, it resolves to the NUL character. Manual inspection revealed that the images were missing. To verify, we hard-coded a valid image location and it fixed the duplicate problem. Listing 5.3 shows the runtime state of the rendered HTML in Firefox browser. On lines 3 and 10, the empty `url()` is observed, while on line 4, the `src` field in `<img>` tag having a value of “/” pinpoints the root cause for the duplicate request to the base URL.

```
--- a/modules/mod_fpss/tmpl/Movies/default.php
+++ b/modules/mod_fpss/tmpl/Movies/default.php
- <span style=‘‘background:url(<?php echo $slide->mainImage; ?>) no-repeat;”>
+ <span style=‘‘background:url(media/system/images/XYZ.jpg) no-repeat;”>
- <img src=‘‘<?php echo $slide->mainImage; ?>” alt=‘‘<?php echo $slide->altTitle; ?>” />
+ <img src=‘‘media/system/images/XYZ.jpg” alt=‘‘<?php echo $slide->altTitle; ?>” />
- <span class=‘‘navigation-thumbnail” style=‘‘background:url(<?php echo $slide->thumbnailImage;
    ?>) no-repeat;”>&nbsp;</span>
+ <span class=‘‘navigation-thumbnail”style=‘‘background:url(media/system/images/XYZ.jpg) no-
    repeat;”>&nbsp;</span>
```

Listing 5.2: Code modification to fix unnecessary duplicate requests

```
 1 <div class=‘‘slide” style=‘‘position: absolute; opacity: 0; z-index: 89;”>
 2   <a class=‘‘slide-link” href=‘‘/fpss/track/35/L3Jlc291,,”>
 3     <span style=‘‘background:url() no-repeat;”>
 4       <img alt=‘‘NEEShub Release 5.0” src=‘‘/”>
 5     </span>
 6   </a>
 7 .
 8 .
 9 .
10 <span class=‘‘navigation-thumbnail” style=‘‘background:url() no-repeat;”> </span>
```

Listing 5.3: Runtime state of generated HTML as observed by Firebug

To understand how current browser versions (Chrome 32, Firefox 26) behave under unexpected input, we did a synthetic injection in `HTML` tags: `<span style:backg-`

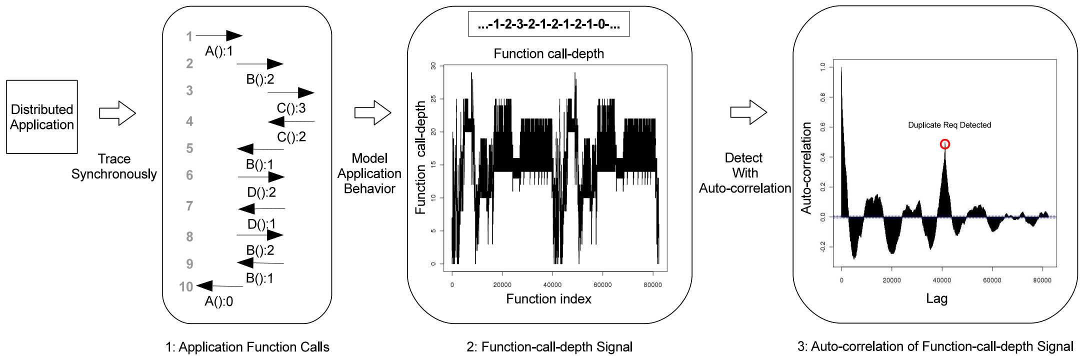


Fig. 5.2.: Overview of the duplicate-detection workflow.

`round=X>`,`<img src=X>, <script src=X>, <iframe src=X>, <link href=X>`. Here `X`, the injected character, had ASCII codes in the range 32-126 excluding alphanumeric characters. We found that, in addition to duplicate requests due to empty strings which have been reported before [7], the characters '?' and '#' also resulted in duplicate requests. `<span style:background=SPACE,EMPTY>` resulted in a duplicate request for both browsers. For Firefox, `<img src=SPACE>`, `<script src=SPACE,EMPTY>`, and `<link href=SPACE>` created duplicate requests. These injections provide evidence that browsers do behave differently and erroneously under unexpected special characters for URIs.

5.3 Design

Here we detail the design of Griffin to detect duplicate web requests. At a high level, it comprises three steps: model application behavior at the web server (in terms of the function calls and returns), create a signal of the function call depths, and compute the auto-correlation of the signal to trigger detection. Figure 5.2 shows these steps in Griffin.

5.3.1 Synchronous Tracing

Tracing can be divided into static or dynamic tracing. Static tracing involves modifying the application's source code to insert tracing statements followed by compilation and execution. Dynamic tracing involves instrumentation of live, in-production applications without needing to stop and restart them. Dynamic tracing can be sub-divided into asynchronous or synchronous tracing. Asynchronous tracing takes samples at regular intervals, from a running application, easing the possible impact on performance but also opening up the possibility of missing important events between samples. Synchronous tracing captures pre-defined events (function calls in our case) within the source code. In this work, we use synchronous tracing as it meets our following tracing framework goals.

1. The tracing should be done dynamically, i.e., the tracer should be able to connect and disconnect to an already running application without the need of stopping, recompiling and restarting.

2. The application should not be polluted with instrumentation within its source code.

3. The instrumentation should provide enough function call context for triggering post-detection diagnosis, e.g, the name of the called function, filename, classname of the object calling the function etc.

4. The instrumentation overhead should be small enough to debug problems in the production environment in an online setting.

We leverage SystemTap [2], a tracing/probing framework that can provide synchronous tracing data on Linux hosts. SystemTap is built on the same architecture as the well-known DTrace [1] tool for Solaris systems and can provide event-tracing across the whole system stack: kernel, applications, system-services, interpreters (PHP, Python, Perl, Java), databases, etc. This ability to look through the whole system with low probe-point programming overhead makes SystemTap a better fit than other tools like PIN [102] and Valgrind [103].

To enable tracing, SystemTap allows to write probe-point scripts. Probe-point scripts tell SystemTap two things. *(1). What event do you want to trace? (2). What do you want to print at the traced event-location?*. An example output of tracing function calls in PHP for the program in Listing 5.7 with SystemTap tracer-script in Listing 5.8 is shown in Listing 5.9. `abc.php` invokes the function call-chain (a()→b()→c()) from main-method. `php.stp`, that traces `abc.php`, has two probe-points, for function-entry and function-return. At both entry and return, `php.stp` logs, in order of their appearance in the `printf` call, timestamp, thread-id, function call depth, funcation name, file name, line number, and class name, if available. Other than `thread_indent_depth(long)` function, all the other functions are natively available in SystemTap .

```
function a() {
 echo ‘‘Func a”;
 b(); } a();
```

```
function b() {
 echo ‘‘Func b”;
 c(); }
```

```
function c() {
 echo ‘‘Func c”;
      }
```

Listing 5.7: `abc.php`, where a() calls b() and b() calls c()

5.3.2 Modeling Application Behavior

For modeling purposes, we define a numeric metric called *function call-depth* that represents the runtime function call-depth. At every function-call, the call-depth is incremented and at every return, it is decremented. *Our foundational intuition for modeling application behavior is that the flow of an application can be roughly represented by how function call depth changes.* The function call depth sequence for a given high-level web operation can be considered as a fingerprint of the high-level

```

probe process(''/usr/lib/apache2/modules/libphp5.so'').provider(''php'').mark(''function__entry'')
{
        printf(''PHP: %d %d => %d %s file:%s line:%d classname:%s\n'', gettimeofday_us(), tid(),
             thread_indent_depth(1),user_string_quoted($arg1), user_string_quoted($arg2), $arg3
            , user_string_quoted($arg4));
}

probe process(''/usr/lib/apache2/modules/libphp5.so'').provider(''php'').mark(''function__return''
    )
{
        printf(''PHP: %d %d <= %d %s file:%s line:%d classname:%s\n'', gettimeofday_us(), tid(),
             thread_indent_depth(-1),user_string_quoted($arg1), user_string_quoted($arg2),
            $arg3, user_string_quoted($arg4));

}
```

Listing 5.8: `php.stp` SYSTEMTAP script with function-entry/return probes

```

PHP: 1392668507729050 22061 => 1 ''a'' file:''/www/a_b_c.php'' line:18 classname:''''
PHP: 1392668507729120 22061 => 2 ''b'' file:''/www/a_b_c.php'' line:5 classname:''''
PHP: 1392668507729134 22061 => 3 ''c'' file:''/www/a_b_c.php'' line:11 classname:''''
PHP: 1392668507729146 22061 <= 2 ''c'' file:''/www/a_b_c.php'' line:11 classname:''''
PHP: 1392668507729158 22061 <= 1 ''b'' file:''/www/a_b_c.php'' line:5 classname:''''
PHP: 1392668507729167 22061 <= 0 ''a'' file:''/www/a_b_c.php'' line:18 classname:''''
```

Listing 5.9: Output of `php.stp` SYSTEMTAP script

operation. For further exploration of this intuition, let us first define some terms: *web-request, web-click, http-transaction*. Starting from the lowest level, a web request is the HTTP request sent by the web browser, such as, GET and POST. A web click is a human user clicking in the browser to send web requests. A single web click can generate multiple web requests. A set of web clicks done in a particular sequence, as permitted by the workflow in the website, is called an http transaction. An http transaction can consist of one or more web clicks; in typical usage this will be more than one web click. An example of an http-transaction of size two is going to the homepage followed by going to the login page (HomePage→Login).

Now coming back to our intuition for detecting duplicate web requests, consider that a duplicate web request will create a duplicated signal of the function call depths. It is easy to concoct a synthetic example where this intuition is violated. For example, consider two legitimate consecutive web clicks and the corresponding web requests: (a (b (c c') b') a') (d (e (f f') e') d') giving a call-depth sequences of (1 2 3 3 2 1) (1

2 3 3 2 1). This would give the appearance to Griffin of duplicated web requests. However, we find that for real web pages, the length of web clicks in terms of the number of function calls and returns tends to be much larger. This kind of accidental matching of the function call depth signal happens only very rarely for these real situations.

To get the call-depth at runtime, we add a function called `thread_indent_depth(long)` to SystemTap 's native scripts. This function returns a number corresponding to the depth of nesting. We call this function `thread_indent_depth(1)` in the probe-point SystemTap script (Listing 5.8). Here, the argument one means that at every function-call, increment the depth by one. We submitted this function to the SystemTap repository and it has been merged (commit-id:cecdb2dddae55b51081-4dc8a6d7510e5fa5e0d9f) into SystemTap 's master-branch and is available out-of-the-box after SystemTap is installed [104].

5.3.3 Duplicate Detection Algorithm

With the function call-depth sequence captured, the next goal is to detect whether the sequence has a repetitive pattern and to do this efficiently with respect to time. To do this, we use a common signal analysis technique to detect repeating patterns, *auto-correlation* [105] of the function call-depth signal. Auto-correlation of a signal x is defined by R_{xx} (Equation 5.1) as a function of lag-value t, where t varies from zero (perfect signal match with R_{xx}=1) to n, the sequence length in terms of the number of function calls and exits. Ideally for Griffin to detect duplicate web requests resulting from a single user web click, it would be possible to segment the web requests for each web click. But that is not always possible in practice, as we discuss in Section 5.4.1. Auto-correlation can be viewed as a sequences of *shift, multiply, sum* operations for all lag values on function call-depth signal. Intuitively, we are using auto-correlation to estimate the similarity between the signal and its time shifted versions for various

```
/* Compute auto-correlation for all lags */
$X$ ← load signal values
$\bar{X}$ ← get mean value
$C_0$ ← 0
$R_{xx}[t]$ ← get an empty array of size $n$
$Threshold$ ← 0.4
for each $t$ in range $n$:
  $sum$ ← 0
  for $j$ ← 0; $j < n$; $j$++:
    $sum$ ← $sum$ + $(X[j] - \bar{X}) * (X[j+t] - \bar{X})$
  $sum$ ← $sum/n$
  if $t$ = 0
    $C_0$ ← $sum$
  $R_{xx}[t]$ ← $sum/C_0$

/* Get the index where auto-correlation first becomes negative */
$index$ ← 0
for each $t$ in range $n$:
  if $R_{xx}[t]$ < 0
    $index$ ← $t$
    break

/* Check if auto-correlation is great than threshold beyond $index$ */
for $i$ ← $index$; to end of sequence
  if $R_{xx}[t] \geq Threshold$
    Print Duplicate Request Detected!
    break
```

Fig. 5.3.: Algorithm to detect duplicate messages from function call-depth signal.

values of the time shift. If the function-depth signal is exactly repeated twice, we expect to see a peak of 0.5 around the lag value of *n/2*.

$$R_{xx}[t] = \frac{C_t}{C_0} \text{ where t=0,\ldots,n}$$
$$C_t = \frac{1}{n} \sum_{s=max(1,-t)}^{min(n-t,n)} [X_{s+t} - \bar{X}][X_s - \bar{X}] \tag{5.1}$$

We present the auto-correlation-based duplicate-detection algorithm in Figure 5.3. After auto-correlation computation for all lag-values, we find the index at which the auto-correlation first becomes negative, call this t_0. For values of auto-correlation beyond t_0, we find if there is any value greater than a threshold value τ. If yes, we flag a duplicate-detection. For the duplication of a set of web requests once, we expect ideally an auto correlation peak of 0.5. But to tolerate the normal variation in function call-depth signal, we set the threshold τ to be a little lower than 0.5. We report on our sensitivity empirical study in Section 5.5.3. The reason for starting the search beyond t_0 is that then we eliminate the high values of autocorrelation that we will see due to the original signal being correlated with itself with small time lags.

5.3.4 Usage Modes

We envision Griffin to work in two scenarios, pre-production *testing* and *in-production*. In testing, developer's have control of the environment and trace segmentation is not an issue. Here, a possible concern by developers could be Griffin's detection latency, which is in order of seconds. Also, there are several works that show speed-up of autocorrelation-like functions using parallelization in software [106] and hardware like FPGAs [107], GPUs [108]. For in-production mode, operators' main concern could be the overhead of configuring and tuning Griffin and the application tracing overhead, which is incurred in the critical path of all web requests and responses. Griffin's configuration is minimal with only one threshold parameter for which we provide a recommendation (threshold=0.4) with our sensitivity analysis. In fact, this simplicity was appealing enough for us that we adopted this scheme in favor of more complex, and potentially better-performing, schemes that have a plethora of parameters. The tracing overhead with SystemTap is low enough as it is. To further minimize the tracing overhead, an operator can run Griffin in time intervals of low load on the web server .

```
global remote_ip

//Probe 1: Apache probe to get ip-address of incoming web-request
probe process(''/usr/*bin/apache2").function(''ap_process_request")
{
        remote_ip[tid()]=user_string(@cast($r->connection, ''conn_rec")->remote_ip)
}

//Probe 2: probe that receives the http-request from Apache in the PHP runtime
probe process(''/usr/lib/apache2/modules/libphp5.so").function(''php_apache_request_ctor")
{
        printf(''APACHE: %d %d %s\n", gettimeofday_us(), tid(), user_string_quoted($r->
            unparsed_uri));
}

//Probe 3: php function-entry
probe process(''/usr/lib/apache2/modules/libphp5.so").provider(''php").mark(''function__entry")
{
        printf(''PHP: %d %d %s => %d %s file:%s line:%d classname:%s\n", gettimeofday_us(), tid
            (), remote_ip[tid()], thread_indent_depth(1),user_string_quoted($arg1),
            user_string_quoted($arg2), $arg3, user_string_quoted($arg4));
}

//Probe 4:
probe process(''/usr/lib/apache2/modules/libphp5.so").provider(''php").mark(''function__return
    ")
{
        printf(''PHP: %d %d %s <= %d %s file:%s line:%d classname:%s\n", gettimeofday_us(), tid
            (), remote_ip[tid()], thread_indent_depth(-1),user_string_quoted($arg1),
            user_string_quoted($arg2), $arg3, user_string_quoted($arg4));
}
```

Listing 5.11: SystemTap probes for tracing

5.4 Experimental Setup

5.4.1 Configurations: Hardware, Software, Tracing

NEEShub infrastructure is running Apache/2.2.16 (Debian) web server in Prefork MPM (Multi-Processing Module) [109] mode, *i.e.*, with multiple processes and one thread per process, on a VM with Intel(R) Xeon(R) CPU E5-2643 0 @ 3.30GHz with 6GB RAM. The PHP-runtime (`libphp5.so`) version is 5.3.3 and is compiled with `--enable-dtrace` option in order for SystemTap (ver 2.4) to be able to intercept PHP-function calls and returns with its probes.

Listing 5.11 presents the SystemTap probes used for synchronous tracing. The objective that we discuss here is how to segment multiple users so that the analysis can be done on web clicks from a single user. This segmentation is conceptually done using a combination of fields such as, source IP address, port, etc. For ease of exposition, in the discussion in this section we will use IP address as the proxy for this combination. In Listing 5.11, we show the SystemTap probes that Griffin inserts. On receiving a web request, Apache probe (Probe 1) fires first. When probe 1 fires, we record the IP address of the incoming web-request in a global variable with the key equal to the Apache thread id. Since the design is such that the same thread id is going to complete the web click request, so, whenever subsequent probes fire they have the same thread id. The probe at the interface between the Apache web server and the PHP module is probe 2. Probe 3 gets triggered at all function calls in the PHP module and probe 4 at all function returns in the PHP module. In probes 2, 3, and 4, we read the global variable to get the IP address corresponding to thread id of the current thread. Thus, even though the IP address is visible only to probe 1, inward-situated probes 2-4 can also access the client-distinctive information. In terms of frequency, most probe 1's have a corresponding probe 2; each probe 2 gives rise to many probes 3 and 4. This is due to the design, common to most content-rich web sites, that dynamic content is generated through the PHP module and the PHP module invocations traverse a long chain of libraries. For example, for the NEEShub, on an average a single web click results in a total of 67,071 function calls and returns in the PHP module.

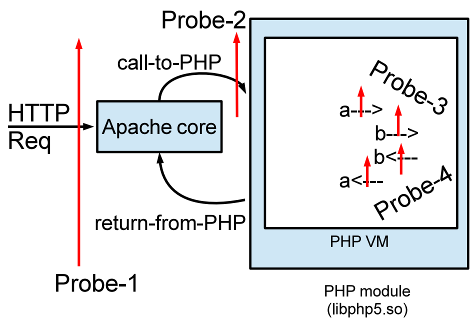


Fig. 5.4.: Probes that fire in the life-cycle of an HTTP request

5.4.2 Evaluation Metrics

We evaluate Griffin's detection performance with traditional definitions of accuracy and precision (Equation 5.2). We establish the ground truth through manual verification, at client-end, by checking duplicate requests for each web-click using browser debugging tools, Firebug and Chrome-dev-tools. We measure the overhead of Griffin in two areas, tracing overhead and detection overhead. Tracing-overhead is the fraction of total time, taken by SystemTap 's probes while processing a given web-click. Detection overhead or detection latency is measured in the standard way as the time elapsed for all the detection steps—getting the signal as input, computing auto-correlation, determining the trigger point, and sweeping through a series of time lag values to detect if there is a peak corresponding to the duplicate web request.

$$
\begin{aligned}
&Accuracy = \frac{TruePositives + TrueNegatives}{TruePositives + TrueNegatives + FalsePositives + FalseNegatives}\\
&Precision = \frac{TruePositives}{TruePositives + FalsePositives}
\end{aligned}
\tag{5.2}
$$

5.4.3 NEEShub Infrastructure

We apply Griffin to a large-scale real setup called NEEShub [100], a system built with HUBzero [98] open source software platform. HUBzero is a proven dynamic website building technology to support scientific research and educational activities. HUBzero has over 29 documented hubs [110] operating across various scientific disciplines. In addition to building dynamic websites, HUBzero provides a modular architecture that helps to quickly build tools (for data-analysis, simulation etc) for a given scientific discipline. We can view NEEShub as an instance of HUBzero where the website and tools are developed for Earthquake Engineering discipline. NEEShub architecture is composed of a front-end webserver and a back-end database. NEEShub's website usage shows an increasing yearly trend of number of unique users who logged into the website for past four years. Additionally, after its inception in 2009, the annual webserver hits for last 3 years has been over 32 million. Both, webserver hits and increasing website users mean that any duplicated web requests can cause a performance bottleneck or may lead to unneeded hardware upgrade.

5.5 Evaluation

5.5.1 Experimental Workload

Griffin's testing was conducted on a replica of the production site (www.nees.org), technically referred to as a "staging machine" where developers merge their code after doing the unit testing on their own development box. We made no modifications or synthetic error injections. Therefore, we expected to find few, if any, problems with the website.

We tested Griffin's duplicate-detection performance by sending a total of 60 HTTP transactions of varying sizes. The size of a transaction is measured by the number of web clicks incorporated within the transaction. Thus, the transaction HomePage$\rightarrow$Login has a size of two. Also, for the analysis (autocorrelation computation), the signal is considered the entire transaction. We used 20 transactions for each of the sizes 1,2,3. These 60 HTTP transactions were executed following different possible user workflows as enabled by the web portal. We tried to cover all the workflows that a typical user would follow while visiting the website. This is enabled by our having worked as part of the NEES team for 3+ years.

Ideally, the analysis in Griffin will consider the traces corresponding to a single web click from a single user. The combination of IP address, source port, etc. is meant to segment different users. Within a single user, we expect that different web clicks are handled by threads of different IDs. We empirically validated that this is *always* the case for all our transactions. This is explained by the design of the Apache web server, which has a maximum number of concurrent requests that can be served, given by the parameter `MaxClients` with a default value of 256. For Apache Prefork MPM, this translates to the total number of processes. When a request processing is completed, the process becomes idle and after an expiry time, is killed off. If the next request arrives within the expiry time, then with a probability $\frac{1}{MaxClients}$, it will be handled in a process with the same thread ID. If the next request arrives after the expiry time, then there is only an infinitesimal chance that it will be processed with the same thread ID. To account for these probabilities, plus other Apache modes (multi thread, etc.), we also evaluate GRIFFIN's performance with HTTP transactions of size greater than one. For HTTP transaction size greater than one, we are mimicking the situation where the duplicate request happened due to one web-click (e.g. , HomePage) but GRIFFIN analyzed two (e.g. , HomePage→Login) or three web clicks (e.g. , HomePage→Login→LoggingIn) together.

5.5.2 Accuracy and Precision Results

Out of the 7 duplicate request problems (among the 60 HTTP transactions), GRIFFIN was able to correctly find 4 duplicated requests i.e., *HomePage, Topics-page, SimulationWiki-page and Wiki-page.* SimulationWiki page was due to a Javascript-based duplication, while the other three were due to missing-resources. GRIFFIN missed 3 cases of duplicated requests, *warehouse, simulation and education* pages.

GRIFFIN's accuracy and precision with different HTTP transaction sizes is presented in Table 5.1. GRIFFIN provides an average accuracy of 80% across HTTP transactions of size one and two with no false positives. With three web clicks, GRIFFIN's performance degrades– here 0% precision is misleading in the sense that out of the 20 HTTP transactions of size three, only one (HOMEPAGE→LOGIN→LOGGINGIN) had a duplicate request which GRIFFIN did not detect. GRIFFIN falsely flagged 4 out of 20 transactions giving a false positive rate of 20% for HTTP transactions of size three. The reason why GRIFFIN did not detect HOMEPAGE web-click within HOMEPAGE→LOGIN→LOGGINGIN transaction is due to the significant difference of LOGGGINGIN function call-depth signal from the signals of HOMEPAGE and LOGIN web clicks (see the increase in function call-depth signal between index

100K to 150K in Figure 5.6). Here, HOME and LOGIN web clicks have an average function call-depth of 15.61 and 15.47 respectively while LOGGINGIN has an average of 32.42 making it significantly different. With HTTP transaction of size 3, Griffin is performing its analysis after combining these three signals into one. Thus, the divergence in the single combined signal means that the autocorrelation values, even with one duplication, tend to be low, and stay below the threshold. In practice, the HTTP transactions of size 3 will be very rare because of the discrimination that Griffin will be able to do using the thread ID.

Table 5.1.: Summary of Performance Results

	Accuracy	**Precision**
one-click	$90\% = \frac{18}{20}$	$100\% = \frac{3}{3}$
two-clicks	$70\% = \frac{14}{20}$	$100\% = \frac{4}{4}$
three-clicks	$75\% = \frac{15}{20}$	$0\% = \frac{0}{4}$

With the ideal (and practically common) case of analysis over HTTP transaction of size 1, Griffin shows 90% accuracy and 100% precision. As an example, the function call-depth and autocorrelation for HOME web-transaction is presented in

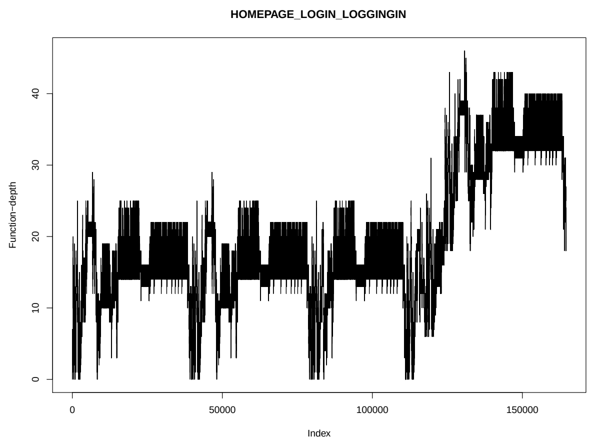


Fig. 5.5.: HOME→LOGIN→LOGGINGIN: Function call-depth signal for three web clicks from browser

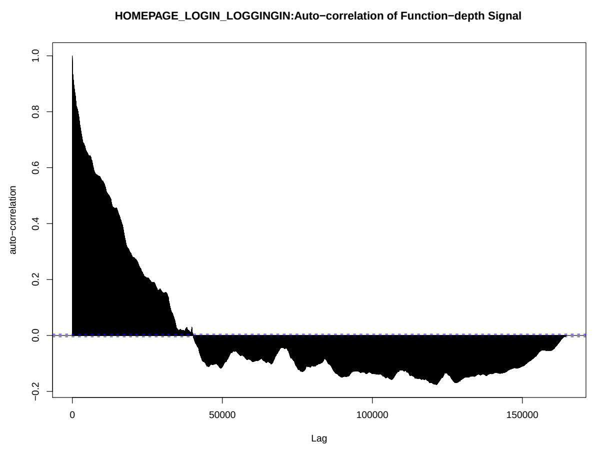
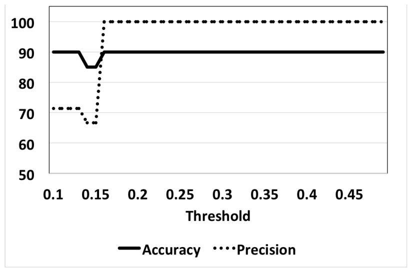


Fig. 5.6.: HOME→LOGIN→LOGGINGIN: Autocorrelation for three web clicks from browser

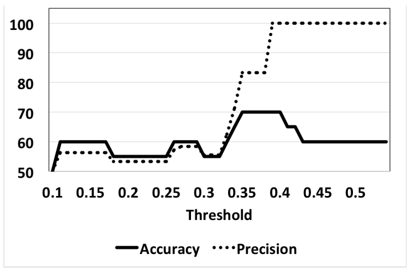
Figure 5.2. We see that the autocorrelation has a clear peak value of 0.4998 near a lag-value of 40,000 which is detected by Griffin (with a threshold set at 0.4). Manual checking, both at user-end and at server-end revealed that HOME web-request ("/") is being sent twice by the user's browser. Further inspection on the server revealed that a field called `hits` in the back-end database is incremented on every HOME web-transaction. We reported this hitherto unknown problem to the web developer at NEES, and it was subsequently fixed and not pushed into the production environment. Testing Griffin with HTTP transactions of size 2, we observe a drop in accuracy (to 70%). This happens due to the significant variability in the basic signal due to the very different nature of the function call invocations in the two web clicks. Expectedly, autocorrelating a divergent signal gives low autocorrlation values, which sometime fall below the Griffin threshold, which has a default value of 0.4.

5.5.3 Sensitivity and Overhead

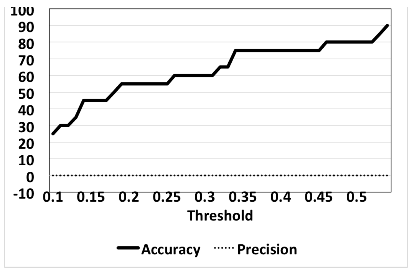
Griffin's sensitivity to different parameters, sequence length, threshold and number of traced contiguous web clicks is critical from a usability perspective. With an increasing number of contiguous web clicks, Griffin's accuracy and precision drop.



(a) One-click



(b) Two-click



(c) Three-click

Fig. 5.7.: Sensitivity of Griffin for each of one, two and three-clicks

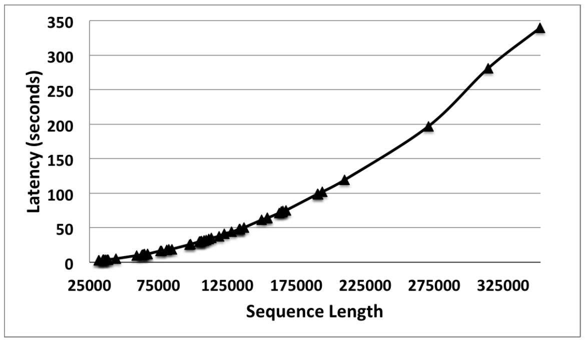


Fig. 5.8.: Detection Latency

The pattern of accuracy decreasing with increasing number web clicks holds true with increasing sizes of the traces. We present GRIFFIN's sensitivity with different thresholds in Figure 5.7. Looking at Figure 5.7(a) and Figure 5.7(b), we set GRIFFIN threshold to 0.4 as the default value for GRIFFIN to provide us zero false positives, *i.e.*, 100% precision. The user can decrease the threshold for fine tuning her system, but we suggest to not go below 0.35 (based on Figure 5.7(b)) as that can result in possible false positives.

The detection latency as a function of the sequence length (*i.e.*, the number of trace events due to SYSTEMTAP probes) is shown in Figure 5.8. It shows the expected behavior of greater latency with increasing sequence length. This is due to a larger number of autocorrelation computations for a longer trace length. However, the upper range of the sequence length is typically about 100K and with that we have a detection latency of about half a minute, which should be fast enough to be useful for the subsequent manual process of fixing the problem. The average tracing overhead across the 60 tested HTTP transactions is 28.6% with a standard devitation of 10.0%. The overhead for HTTP transactions for each size is presented in Table 5.2. The tracing overhead is independent of the length of the sequence and the differences seen are due to statistical variations. This overhead can be reduced simply by removing all the fields in the traces, say as in Listing 5.8, except for the thread ID (needed for segmentation), function call depth (needed for detection), and filename (needed for diagnostic context, as explained in next Section.

Table 5.2.: Tracing Overhead

	Tracing Overhead (Avg)	Tracing Overhead (Std. Dev)	Sequence Length (Avg)	Sequence Length (Std. Dev)
one-click	24.0%	6.6%	67,071	54,165
two-clicks	32.8%	11.6%	131,511	76,630
three-clicks	29.1%	9.1%	141,427	33,727

5.6 Discussion

GRIFFIN's Diagnostic-context: When GRIFFIN detects duplicate web-requests, a diagnostic-context about the detection would help the developers as a starting point for debugging. At detection-time, in addition to the autocorrelation value, we also have the lag when this autocorrelation value exceeded the threshold, call this t_{max}. We use t_{max} alongwith the information provided by probe-2 (Figure 5.4), a probe that records the HTTP-request going from `apache-core` to PHP-runtime, to provide the *diagnostic-context*. With the t_{max}, we get the nearest next fired probe-2 and extract a high-level component (module name) from the file name. For the duplicate bug of Figure 5.2, this simple scheme is able to flag `mod_fpss` module in Joomla, the Content Management System, on which HUBzero is built.

ID-based Trace Segmentation: Given two parallel web-clicks from two different users, the segmentation-problem (as discussed in Section 5.5) of getting a per-user click-trace can be solved using a variety of IDs available at the server-end. Specifically, for Apache server, the data structures, `request_rec` (created whenever the server accepts an HTTP request from client) and `conn_rec` (an internal representation of TCP connections in Apache) can help in filtering. An interesting scenario occurs when the same IP address represents different users, e.g. , multiple users behind a NAT (Network Address Translation) server. In this case simply segmenting users by IP address will not work and additional information, e.g. , port number is needed. This can be achieved with writing an Apache probe with SYSTEMTAP intercepting the function `ap_process_connection` (which is invoked in the call-chain for every incoming web-request) and reading the port field using `cast($c, "conn_rec")->remote_addr->port`.

Simple Strawman Schemes: Given that we have the function call depth and the sequence length, a simple scheme for detection of duplicate requests can be a threshold for each of the two parameters. Though simple in theory, we will need to maintain and learn per-web-request type thresholds, e.g. , different thresholds for each of Homepage, Login etc. Additionally, as the application is developed new web-requests would have to be benchmarked for learning the threshold. We think that for a busy operator, this poses too much of configuration overhead. An attraction of Griffin is that it requires little configuration for detecting a wide variety of duplicate request problems. Another possible technique is to use logs at the server and use some heuristic to flag detection when similar requests are received within a small time window. Such a scheme though would be fragile in practice and be unable to handle natural variations in user request patterns.

In this chapter, we have presented a systematic method and an automated tool called Griffin for detecting an important problem that afflicts many web servers, namely, duplicate client browser requests. This causes an artificially high load on servers and corrupts server and client state. Culling together many blog posts and developer forum reports, we identify the two fundamental root causes of the problem and come up with a solution that handles both, without needing special case logic for the two root causes or for different browsers. We use Griffin for detecting the problem in a production web portal for an NSF center at Purdue and identify that the problem is more widespread than previously identified. Our evaluation on the production site revealed no false positive. The dynamic system tracing using SystemTap is lightweight and the detection latency small enough (less than half a minute) as to be useful in practice. Our contributions were considered significant enough that the problem was fixed in the web portal and our addition to the dynamic tracing facility was accepted in its official release.

6. FAILURE DIAGNOSIS WITH METRIC-BASED PERFORMANCE PROBLEMS

6.1 Introduction

Debugging today's large-scale distributed systems is complex. Systems are composed of multiple software components often running on distributed nodes. The interactions between these components are complex enough that they cannot all be enumerated a priori. The unpredictability of the execution environment and its effects on the application execution increases difficulty in the debugging process. Failures can come from different layers of the system—network, hardware, operating system, middleware, and application layers. Thus in general, it is necessary to monitor the behavior of all the layers to understand the origin of failures.

Why another debugging tool? There exists a significant number of debugging tools today [49, 111, 112]. They work well for many kinds of failures though they require varying amounts of developer intervention. Despite the existence of this rich set of tools, the debugging process is time consuming and it often requires full domain knowledge of the program to find where problems originate in the source code. Profilers [113–115] can help in isolating performance bottlenecks, however, identifying the root cause of the bottleneck still remains essentially a manual process. Research on distributed systems has developed functional techniques that can help in debugging, such as program tracing and replay debugging [47, 48, 116], model checking [50–52], and log analysis [53–56]. But there remains work to be done to build on these techniques to create a usable debugging tool.

In this work, we focus on debugging distributed applications by identifying the region of code where a fault first becomes active. The developer can then focus on this region to fix the problem rather than spending time in examining the entire source code. We focus on *manifested-on-metrics* (MM) bugs, i.e., those bugs that manifest themselves as an abnormal temporal pattern in one or more metrics at the hardware, OS, middleware, or application layers. MM bugs can manifest as performance or correctness problems. Examples are resource leaks prior to an application crashing, or incorrect use of synchronization locks prior to the application hanging. We do not handle bugs that lead to incorrect output, data corruption or failures that do not affect a system-measurable metric.

Design approach. We present Orion, a framework for localizing the origin of MM faults in distributed applications. Orion works by profiling a variety of metrics as the application is executing, either at declared instrumentation points (such as, method entry or exit) or asynchronously with a fixed periodicity. Through machine learning techniques, it verifies if the runtime profile is *similar enough* to profiles created offline of non-faulty application's executions. If it is not, Orion goes back through traces to indicate which metrics caused the divergence and from that, to the region of suspect code. The mechanism is probabilistic thus a rank-ordered list is provided to the developer for inspection. This design approach is shared with a few prior software systems [60, 117]. However, unlike these prior systems, which only gather traces from one or two dimensions of the application, e.g., CPU and memory, Orion performs application profiling along a large number of metrics. These metrics do not have to be hand-picked by the developer. Orion first automatically selects important metrics from the entire set for detailed analysis and then uses correlations between the metrics to diagnose subtle errors.

Summary of findings. We deploy Orion and evaluate it on diverse distributed applications: HBase [118], Hadoop [119], an on-campus Java Enterprise Edition (JEE) application, called StationsStat, and an IBM regression testing application for its full system simulator called Mambo. We focus on failures that were difficult to debug manually. We demonstrate Orion with a total of 7 bug cases from these applications and find that the root cause is related to the top 3 abnormal metrics or code regions in 6 of them.

The main contributions of this work are:

(1) We introduce the concept of multi-dimensional metric profiling to provide problem determination for a wide variety of root causes. We show that, despite profiling a large number of metrics, it is possible to find the "needle in the haystack", i.e., the problematic metrics in a (possibly small) time window amidst a large number of normal metrics.

(2) We focus on a subtle class of problems that are not diagnosable by comparing instantaneous values of metrics against thresholds. Our approach develops a correlation-analysis algorithm that identifies when joint behavior of metrics is suspect.

(3) We successfully demonstrate ORION in four different distributed applications and under 7 failure conditions. In all of the cases, ORION pinpoints the origin of problems to abnormal metrics and in most of the cases it finds code regions that require human attention to fix the failures.

The rest of the chapter is structured as follows. In Section 6.2, we provide an overview of the execution of ORION. In Section 6.3, we present the detailed design. In Section 6.4, we describe the seven case studies. In Section 6.5, we discuss practical implications of this work.

6.2 Overview

6.2.1 Measurement Gathering

ORION collects measurements of multiple metrics at different levels in the system, i.e., hardware, OS, middleware and application by means of third-party monitoring tools. Using collected measurements, ORION builds models of normal behavior, which permits localizing the failures' origin.

The appendix shows the list of metrics that we measure. Although this is not an exhaustive list of all the metrics in the system, we tried to monitor as many metrics as possible from multiple layers. ORION does not impose a limit in the number of metrics used. Instead, it allows developers to use as many metrics as they think are useful for debugging. ORION addresses the *curse of dimensionality* problem by filtering out noisy metrics and automatically zooming into the metrics that are relevant for failure debugging.

Some examples of the metrics that we gather are:
End-user Application: per-servlet statistics such as processing time, request and exception counts; *Middleware:* cache hits and accesses, number of busy and created threads, and request processing time from the middleware layer (such as Apache Tomcat); *OS:* cpu- and memory-usage, context switches, file descriptors, disk reads/writes, packets received and transmitted, stack size; *Hardware:* L1/L2/L3 data and instruction cache hits and misses, TLB misses, branches taken/not taken/miss-predicted, load/store instructions, hardware interrupts.

Importance of collecting metrics from different layers. Bugs can manifest in different layers of the system, therefore it is necessary to monitor metrics from all layers. For example, a file-descriptor leak often manifests as an abnormal pattern in OS-related metrics such as memory and file-descriptor usage—Section 6.4.1 illustrates this case in a file-descriptor leak bug in Hadoop. Other bugs can be diagnosed faster if hardware-related metrics are analyzed. To illustrate this, consider the sample code in Figure 6.1. Here, a performance problem would arise if a developer does not realize the importance of the sorting operation at line 8 and comments it out as a result. Then, due to random nature of the data, branch misprediction will drastically increase for the *if* statement at line 11 causing a performance degradation. In our experiments with multiple runs of this bug, performance can be reduced by a factor of 2.4x. To our astonishment, we found that many developers are not aware of such bugs—in fact this was one of the *most* voted topics in *stackoverflow* [120] and also one of the most viewed questions with more than 214,000 views. To further verify this fact, we conducted a survey with 30 respondents with 3 to 7 years of experience in software industry. Only 23% of them realized removing the sort statement might degrade performance.

In cases like in Figure 6.1, hardware metrics related to branch-misprediction would be more useful than other metrics in finding the problem's root cause. Since it is difficult for developers to select a priori the important metrics in debugging, our tool starts with the entire set of metrics and automatically identifies those metrics with the zoom-in process that we describe in detail in Section 6.2.3. For example, when we used our tool in the above scenario, it identified `branch-mispredicted` as the top anomalous metric. When we used only OS-level metrics, it was not able to accurately identify the root cause.

6.2.2 Profiling

Orion can perform multi-dimensional profiling in two ways: *synchronous* and *asynchronous*. In asynchronous mode, metric collection happens asynchronously to the application. The measurement gathering is done by a process separate from the application process(es). The asynchronous mode does not interfere with the monitored application and therefore is a lightweight method. Orion collects OS metrics values from the Linux `/proc` file system, hardware metrics through PAPI [121] and

```
/*From a random data-set, add numbers greater than 127*/
int arraySize = 50000;
int data[] = new int[arraySize];
Random rnd = new Random(0);
for (int c = 0; c < arraySize; ++c)
    data[c] = rnd.nextInt() % 256;
/* No sorting causes branch misprediction */
//Arrays.sort(data);
long sum = 0;
for(int i = 0; i < 100000; ++i) {
    for (int c = 0; c < arraySize; ++c)
        if (data[c] >= 128)
            sum += data[c];
}
```

Fig. 6.1.: Performance bug due to branch misprediction.

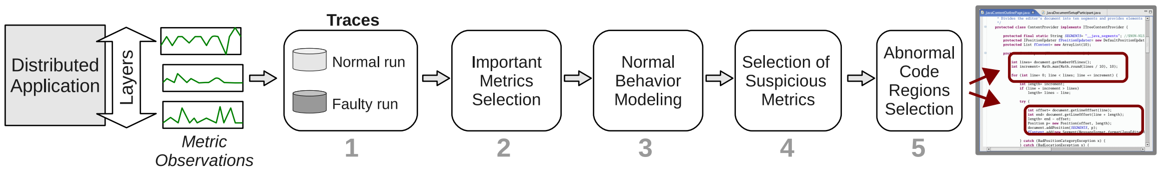


Fig. 6.2.: Overview of the problem determination workflow.

middleware- and application-metrics values from server containers by querying Java JMX connectors via a separate Linux process. This method requires offline processing to “line up” the metric collection points with the execution points of the application. This is done by using a common time base since all the involved processes execute on the same machine.

Synchronous profiling annotates code regions with a set of measurements. Whenever a code region begins and ends, this method collects measurements and labels them with the corresponding code-region name. For Java applications (such as in the Hadoop and HBase case studies), we use `Javaassist` to instrument binary code and to collect measurements at the beginning and at the end of classes/methods.

Notice that Orion can rely on other metric collection mechanism as long as they can provide at least the asynchronous mode of operation—the main novelty of Orion is its problem determination algorithms based on multi-metric data sets.

6.2.3 Workflow of our Approach

Figure 6.2 shows the steps in Orion to diagnose failures. **(1) Trace collection:** Orion uses two set of traces to localize the origin of problems: a *normal* and an *abnormal* trace file. Normal trace files are obtained by collecting metrics of the application when failures are not manifested. This can be runs of an earlier bug-free application version or, in the case of intermittent failures, sections of a failed run where the fault did not manifest itself. The abnormal trace file is obtained when the failure is manifested. Labeling a run as one or the other is a manual process.

(2) Selection of pertinent metrics for modeling: Orion filters out unimportant (or noisy) metrics from the analysis. This step can also be viewed as dimensionality reduction. We used an algorithm based on Principal Component Analysis (PCA) to reduce the dimensionality of the problem. The intuition behind this step is to eliminate metrics that do not provide much information for the rest of the analysis; an example is constant metrics.

(3) Normal-behavior modeling: Orion creates a baseline model using the normal-behavior traces. Given traces of n metrics, the algorithm splits traces into equally sized time windows and calculates pairwise correlations between all the n metrics for each time window. These correlation serve as a summary of the expected behavior of the application in different time windows.

(4) Suspicious metric selection: Abnormal traces are used to select the metrics that are correlated with a failure. From all the n metrics, the top-3 most abnormal metrics are presented to the user. The user can then focus on finding the problem's origin based on the abnormal metrics.

(5) Abnormal code-region selection: Often, finding suspicious metric information in step 3 is not sufficient to infer the origin of a failure. For example, a metric like CPU utilization may be affected by any region of code. Orion selects the code regions that have a high degree of association with the suspect metric(s). Orion highlights suspicious code regions so that users can focus on finding bugs that could have caused the problems within them.

6.3 Design

6.3.1 Selection of Pertinent Metrics for Modeling

Some dimensions (or metrics) are more important than others when debugging failures. We perform dimensionality reduction to: (i) eliminate redundant metrics from the analysis, and (ii) reduce computation overhead. We use Principal Component Analysis (PCA) as a baseline technique for dimensionality reduction. PCA uses

Table 6.1.: Almost equal variance explained by first two PCs.

Data				**Transformation Matrix**			
A	**B**	**C**	**D**	**PC-1**	**PC-2**	**PC-3**	**PC-4**
3	9	3	9	0.70	-0.11	0.68	-0.16
6	7	2	1	-0.21	0.31	0.46	0.80
1	6	2	1	0.66	0.07	-0.57	0.48
8	5	8	4	0.11	0.94	-0.03	-0.32
% of Variance Explained				**46.71**	**45.28**	**8.0**	**0**

an orthogonal transformation to convert a set of observations (of possibly correlated variables (into a set of values of linearly uncorrelated variables, i.e., principal components (PC). Applying PCA directly does not eliminate metrics from the analysis (which is our goal). Instead, PCA generates a new set of metrics which are linear combinations of the input metrics. We developed a heuristic following the idea in [122] to rank input metrics based on their importance. The idea is to rank metrics that are weighted heavily, especially in the first few PCs of the PCA-generated transformation (which contain most of the variance of the data), higher than metrics with smaller weights. Different from [122], we consider the contribution of each PC to explain the total variance in the data and design a new algorithm to rank metrics based on that.

To illustrate, consider Table 6.1 with 4 metrics **A**, **B**, **C**, **D** and 4 observation rows. In this experiment, the first principal component (PC-1) and the second principal component (PC-2) cover 46.7% and 45.3% of the variance of the data, respectively. In the transformation matrix generated by PCA, metric **A** has a weight of 0.7 in PC-1 and metric **D** has weight of 0.94 in PC-2. The heuristic used in [122] will rank **A** higher than **D**. But if we also consider the variance covered by each PC, metric **D** should get a higher rank than metric **A**.

Figure 6.3 shows our ranking algorithm. Here, in the transformation-matrix, we first adjust the weight of a metric by the corresponding contribution of that PC towards explaining the total variance of the data. Then, we give equal importance to

```
metrics: array of metrics names
W: transformation matrix generated by PCA.

percentagePCA ← getVarianceExplainedByPCA(W, normalizedData)
for each column in W:
  /* Multiply columns by variance coverage */
  column ← column * percentagePCA[column]
for each row in W:
  /* Calculate maximum value in row. Store in map */
  rowMaxValueMap[row-num] ← MAX{abs(row)}
rank ← SIZE(metrics)
/*Sort based on max-weight in each row. Key is row-num*/
for each key in valueSort(rowMaxValueMap)
  rankMap[metrics[key]] ← rank
  rank = rank - 1 /* Weight implies importance */
PRINT rankMap
```

Fig. 6.3.: PCA-based heuristic for filtering out metrics.

each column in the modified transformation matrix. We calculate maximum weight of a metric across all columns and then sort metrics based on these maximum weights to get a total ordering of metrics based on its importance. This allow us to filter out unimportant metrics from the rest of the analysis in Orion. The complexity of this algorithm is $O(n^2)$ where n is the number of metrics. We used the difference in ranks between normal and abnormal runs to chose important metrics. Performance bugs tend to change the correlation between metrics and in turn the weights in the PCA transformation matrix which changes relative ranking between metrics. More shift in metrics ranking indicates it was affected by the bug and hence should be chosen for detailed analysis. This first pass is very light weight and coarse as it lacks the notion of time. But it is very useful to reduce the overhead of detailed analysis in the following phase without losing vital information.

6.3.2 Modeling Sequential Data

Many bugs and performance anomalies develop a characteristic temporal pattern that can only be captured by analyzing measurements in a sequential manner (rather than by observing instantaneous snapshots of values). After reducing the number of metrics with the PCA algorithm, we build a baseline model that captures temporal patterns between metrics using correlation coefficients.

Observation window. Traces are split into non-overlapping windows of the same size. A window can be viewed as a matrix $S \times N$ in which S is the number of records (or samples) and N is the number of metrics. The set of records comprises one *observation window*. Since we do not know *a priori* the optimal size of observation windows (S), i.e., the window size that is sufficient to capture the temporal patterns that a failure shows, our algorithm sweeps through multiple sizes for the windows within a range (between sizes of 100 and 200 samples in our evaluation). The algorithm then finds the k-most abnormal windows (irrespective of its size) and, within those abnormal windows, the correlations and metrics that cause the unusual patterns. For our evaluation, we use a k value of 3. Section 6.3.3 describes our algorithms for the selection of suspicious metrics and code regions.

Correlation coefficients. For each observation window, ORION builds a vector of (pair-wise) correlation coefficients between all the metrics

$$CCV = [cc(1,2), cc(1,3), \ldots, cc(N-1,N))], \tag{6.1}$$

where $cc(i,j)$ is the correlation coefficient of metrics i and j, $i \neq j$. We denote this vector as a *correlation coefficient vector* or CCV. Correlation coefficients are calculated using the *Pearson correlation-coefficient* formula:

$$cc(X,Y) = \frac{1}{N-1}\sum_{k=1}^{N}\left(\frac{X_k - \bar{X}}{s_X}\right)\left(\frac{Y_k - \bar{Y}}{s_Y}\right) \tag{6.2}$$

where N is the number of elements of observations in the window, $\bar{X}$ and $\bar{Y}$ are the mean of variables X and Y respectively, and s_X and s_Y are the standard deviations of X and Y.

Normal-behavior Model. Using the normal-behavior traces, our framework creates a baseline model which is used in step 3 (from the main workflow) to select suspicious metrics. The model is a set of normal-behavior CCVs obtained by splitting traces into observations windows and computing a CCV for each window. We term this model as a *hyper-sphere*. Figure 6.4 shows the process of creating this hyper-sphere. The number of points in it corresponds to the number of windows that we obtained from the normal-behavior traces, and the dimensions (or features) are correlation-coefficients of metric pairs. Notice that, if we have N metrics in the analysis, the dimension of the hyper-sphere is $D = \frac{N(N-1)}{2}$.

The idea of using a hyper-sphere where dimensions are correlation coefficients is that we can use nearest-neighbor to pinpoint abnormal observation windows from the faulty traces. Since an observation window is translated to a single data point (i.e., a CCV), we can treat the problem of finding abnormal windows as an outlier detection problem via nearest-neighbor, i.e., an abnormal window would correspond to the CCV point that is the farthest away from the hyper-sphere.

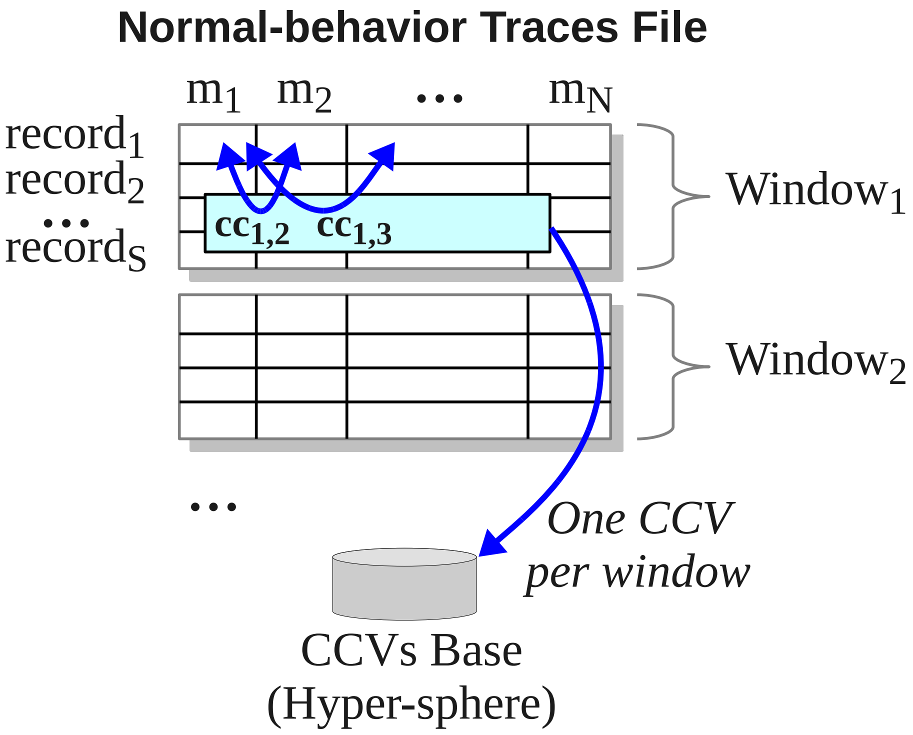


Fig. 6.4.: Creation of the normal-behavior hyper-sphere.

```
/* Get statistics of failed-run windows */
for each size $s$ in range $r$:
  $setOfWindows$ ← create windows set of size $s$
  for each window $w$ in $setOfWindows$:
    $d$ ← find NN distance of $w$ in the hyperSphere
    $ccs$ ← get top abnormal corr. coefficients of $w$
    Append {$w$, $d$, $ccs$} to $tuplesList$

/* Select the most abnormal windows*/
Sort $tuplesList$ based on distance $d$ (high to low)
$abnormalWindows$ ← get top elements in $tuplesList$

/* Build histogram of most abnormal metrics */
for each window $w$ in $abnormalWindows$:
  for each correlation $cc$ corresponding to $w$:
    From $cc$ add metrics X and Y to $histogram$

Print the most frequent metrics in $histogram$
```

Fig. 6.5.: Algorithm to select the suspicious metrics from traces of a failed run.

6.3.3 Detection of Suspicious Metrics

Motivation. The main motivation of our technique is that the manifestation of faults will change the correlations between some (affected) metric(s) and the rest of the metrics, while maintaining the legitimate correlations in the other metrics. To illustrate this idea, consider a bug where unused database connections are kept open—metrics such as file descriptors and open sockets will be affected by the bug and will exhibit a different temporal pattern than during workloads where the bug is not activated. However, correlations among the other metrics will not be affected.

Our goal is that, when faults are manifested, Orion finds the metric(s) that is (are) mostly associated with failures. This is performed by ranking metrics according to their contributions to correlation breakups and by selecting the top-k metrics in this ranking. The application developer can subsequently focus on reviewing the code which affects these suspicious metrics to locate the root cause of the problem.

Algorithm overview. The goal of the algorithm is to select the metrics that are most likely associated with the problem's origin. The algorithm's input is a normal-behavior hyper-sphere and traces of a failed run. The algorithm's output is a list of metrics that are ranked by abnormality degree. The algorithm is presented in Figure 6.5.

The algorithm has the following main steps:

Statistics creation per window: We create observation windows of multiple sizes from the failed-run traces file. For each window, we calculate two statistics: (1) the *nearest-neighbor* (NN) distance of the window from the hyper-sphere representing normality. This distance is calculated by first computing a CCV from the window and then by finding the euclidean distance between the CCV and the closest point in the hyper-sphere using the formula $d = \sqrt{\sum_{i=1}^{D}(cc_i - bb_i)^2}$, where cc_i and bb_i are correlation coefficients; (2) the dimensions that have the highest weight in making the CCV far away from the hyper-sphere. A dimension here corresponds to a correlation coefficient.

Abnormal window selection: Windows are sorted by their NN distance from high to low and only the top-k windows in the list are taken for further analysis ($k = 3$ for our evaluation). These windows correspond to time periods when abnormal behavior is manifested.

Select most abnormal metrics: Once the top-k abnormal windows are ranked, within each window, the correlation coefficients (CCs) are ranked by how much they contribute to the NN distance of that window. Now the top-k CCs are taken from each abnormal window, giving a total of $k \times k$ CCs. Recall that each CC involves two metrics. With these short-listed CCs, the metrics that are present in them are counted up and the top-k most frequently occurring metrics are flagged as the most abnormal metrics. Notice that we use the same parameter for filtering the top choices (windows, CCs, metrics). In theory, they are different parameters, but in practice the same value ($k = 3$) works well and reduces the search space of parameters, a desirable outcome for any deployable tool.

6.3.4 Detection of Anomalous Code Regions

After the suspicious metrics are detected, ORION highlights code regions that make metrics abnormal so that developers can focus on them to fix the problem. ORION first finds suspicious periods of time in which a metric shows an unusual temporal pattern (i.e., an abnormal window). Then, within that period, ORION looks for outlier observations, i.e., an abnormal code region.

Algorithm requirements. The algorithm for detecting abnormal code regions is similar to the abnormal metric-selection algorithm. A major difference is that only one metric is used in the analysis, i.e., the abnormal metric. This metric is given by the previous step, i.e., the metric-selection step. The user can opt to execute this algorithm using the top-two (and top-three and so on) abnormal metric(s) if the top-one abnormal metric does not help in finding the problem's origin. The algorithm's inputs are traces files (from the normal and the failed run) such as in Figure 6.4 but with only one column—this column corresponds to measurements of the abnormal metric. The output of the algorithm is the top few anomalous code regions. We assume that each record in this file is annotated with a code region.

Algorithm overview. The algorithm is shown in Figure 6.6. First, we construct a set of windows from traces of the normal run and another set from traces of the failed run. Second, we find NN distances of the windows of the failed-run from the normal-behavior windows. Then, to select the most abnormal windows, we rank the failed-run windows based on the NN distances (from high to low) and select only the top-k windows. Finally, we build a histogram of the occurrences of code regions in these abnormal windows— as we observe from our case studies, the faulty code regions in performance bugs execute frequently in the most unusual periods of time. The top-3 most frequent code regions are shown to the user.

How to compare one-dimensional windows? In the previous algorithm, we find the difference between two windows by calculating the Euclidean distance of their corresponding $CCVs$. We summarize the window's information by calculating

```
/* Get statistics of failed-run windows */
for each size $s$ in range $r$:
  $normalWins$ ← get windows from normal traces
  $failedRunWins$ ← get windows from abnormal traces
  for each window $w$ in $failedRunWins$:
    $d$ ← NN distance of $w$ from $normalWins$
    Append {$w$, $d$} to $tuplesList$

/* Select the most abnormal windows*/
Sort $tuplesList$ based on distance $d$ (high to low)
$abnormalWindows$ ← get top elements in $tuplesList$

/* Build histogram of abnormal code-regions */
for each window $w$ in $abnormalWindows$:
  Add code regions in $w$ to $histogram$

Print the most frequent code regions in $histogram$
```

Fig. 6.6.: Algorithm to select the suspicious code regions from traces of failed run.

aggregates of its values: *average*, *standard deviation*, *minimum*, *maximum* and *sum*. These aggregates become the features of a window. ORION then finds the dissimilarity between two windows by computing the Euclidean distance using these aggregates as features.

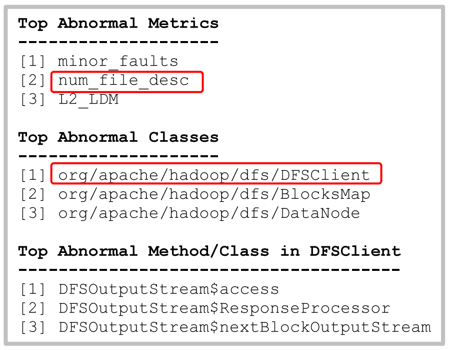
6.4 Evaluation

In this section, we describe how we debugged seven performance bugs in different distributed applications: Hadoop, HBase, a distributed system heavily used in IBM as regression framework, and a distributed application used by students in a large university. We present in detail the first 4 cases and provide a summary of the results of cases 5–7. In all of the cases, ORION reduces substantially the time spent in localizing the problem origin by showing the metric most perturbed by the fault, and if needed further, the abnormal code region. The process is fully automated so users do not need to have full understanding of the application and its components dependencies.

6.4.1 Case 1: Hadoop DFS

Hadoop is an open-source framework that supports data-intensive distributed applications [119]. It enables applications to work with thousands of computational nodes and a large amount of data. We use ORION to diagnose a file descriptor-leak bug that occurred in the Hadoop Distributed File System (HDFS) in version 0.17. The bug report is HADOOP-3067.

We collect all the OS and hardware level metrics given in Table 6.4 via synchronous profiling. All the Java classes and public methods within each class are instrumented. Since we are debugging the Hadoop DFS, we only consider the `java/org/apache/hadoop/dfs` package. A total of 45 Java classes and 358 methods in these classes are instrumented.



```
Top Abnormal Metrics
--------------------
[1] minor_faults
[2] num_file_desc
[3] L2_LDM

Top Abnormal Classes
--------------------
[1] org/apache/hadoop/dfs/DFSClient
[2] org/apache/hadoop/dfs/BlocksMap
[3] org/apache/hadoop/dfs/DataNode

Top Abnormal Method/Class in DFSClient
--------------------------------------
[1] DFSOutputStream$access
[2] DFSOutputStream$ResponseProcessor
[3] DFSOutputStream$nextBlockOutputStream
```

Fig. 6.7.: Results from ORION for the HDFS bug.

This bug is manifested as a failure in one of the HDFS tests (the `TestCrcCorruption` test.) The bug origin is that subclasses `DFSInputStream` and `DFSOutputStream` of the main class `DFSClient` did not handle open sockets correctly by not closing them when they are not used anymore. The patch that developers suggested to fix the bug included changes to the following code: class `DFSClient`, subclasses `DFSInputStream` and `BlockReader` (which is used internally by `DFSOutputStream`). We used the buggy version and revision 0.17 to obtain traces of a failed run, and code from a previous revision where the `TestCrcCorruption` test passed to get traces of a normal run.

Figure 6.7 shows ORION's results. The top-three abnormal metrics presented by ORION are: (1) `minor_faults` (No. minor page faults), (2) `num_file_desc` (No. open file descriptors), (3) `L2_LDM` (level-2 load miss). The 2nd metric is associated with the problem's origin since an increase in the number of open sockets caused by the bug affects directly the number of open file descriptors. Metrics (1) and (3) are both memory related and they are pinpointed as suspect because the bug also causes abnormal memory consumption patterns.

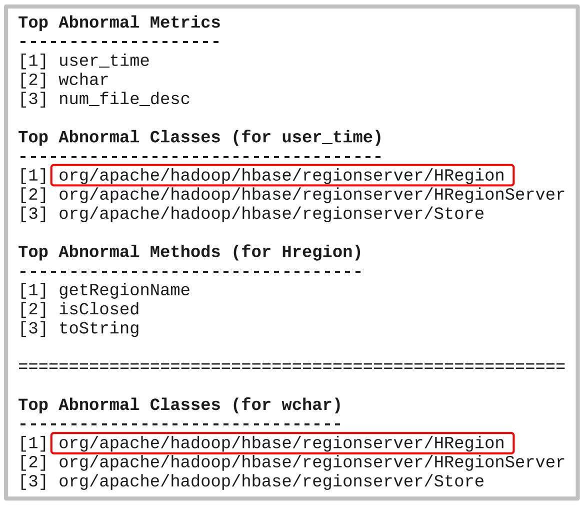
ORION also presents abnormal code regions, first, based on Java classes and second, based on subclasses (of the abnormal classes) and methods within them. ORION correctly pinpoints `DFClient` as the most abnormal class. Within `DFClient`, ORION highlights `DFSOutputStream` as the main abnormal subclass. This is only

Table 6.2.: Average use of file descriptors per class in HDFS for the specific bug discussed in Section 6.4.1.

Rank	**Class**	**Average # File Descriptors**
1	NamespaceInfo	6.0
2	INodeDirectory	1.31
3	INode	1.29
4	UnderReplicatedBlocks	1.25
5	DatanodeInfo	1.24
6	DataNode	1.21
7	DatanodeBlockInfo	1.2
8	DFSClient	1.16
9	DataBlockScanner	1.14
10	NameNode	1.13

partially correct— part of the bug fix is in `BlockReader` which is used internally by `DFSOutputStream` and `DFSInputStream`; however, `DFSOutputStream` does not require changes to fix the bug.

To see if a simpler, and currently practiced, approach can lead the developer to the origin of the bug faster, we set up the following hypothetical steps for hunting this bug. Suppose that a simple profiling tool indicates a high number of file descriptors in use. The developer then proceeds to examine which classes use file descriptors most. The answer to this question is shown in Table 6.2. The average number of file descriptors used per method is calculated by taking an average across all invocations of the methods of that class. From this, the developer would be likely to inspect the classes appearing near the top. It is only when one gets to the 8th ranked class that one gets to the class where the bug lies, `DFSClient`. Thus, this will lead to significant time manually inspecting classes 1–7 and ruling them out as the source of the bug.



```
Top Abnormal Metrics
--------------------
[1] user_time
[2] wchar
[3] num_file_desc

Top Abnormal Classes (for user_time)
------------------------------------
[1] org/apache/hadoop/hbase/regionserver/HRegion
[2] org/apache/hadoop/hbase/regionserver/HRegionServer
[3] org/apache/hadoop/hbase/regionserver/Store

Top Abnormal Methods (for Hregion)
----------------------------------
[1] getRegionName
[2] isClosed
[3] toString

======================================================

Top Abnormal Classes (for wchar)
--------------------------------
[1] org/apache/hadoop/hbase/regionserver/HRegion
[2] org/apache/hadoop/hbase/regionserver/HRegionServer
[3] org/apache/hadoop/hbase/regionserver/Store
```

Fig. 6.8.: Results from ORION for the HBase bug.

6.4.2 Case 2: HBase

HBase is an open-source, distributed, column-oriented database [118]. It operates on top of distributed file systems like the HDFS and is capable of processing very large scale of data with MapReduce. We use ORION to collect and analyze the metrics of a deadlock bug in HBase 0.20.3 (bug report HBASE-2097). We collect all the OS-level metrics shown in Table 6.4. There are 27 java classes that are instrumented from the `hadoop/hbase/regionserver/` package. They include classes to handle region columns, store data files, logs and many other abstractions. These classes include 184 methods which are all instrumented.

The bug, which manifests as an application's hang, is the result of two locks being acquired in an incorrect order. The bug lies in two methods, `HRegion.put` and `HRegion.close`. It is activated by running the HBase `PerformanceEvaluation` testing tool which is used to evaluate HBase's performance and its scalability. The bug manifestation is intermittent—it manifests on average 75% of the time—making it particularly difficult to localize.

We ran a previous bug-free version to generate normal-behavior traces and applied ORION against the traces of a failed run when the deadlock manifests. Figure 6.8

shows the results. The top abnormal metric, i.e., `user_time`, is the amount of user-level CPU time. This metric *per se* does not provide much insight into the failure origin since it is difficult to correlate that to a code region. We observe that, using `user_time` as our abnormal metric, the most abnormal code region is `HRegion` class, which is where the bug lies. Further, considering the 2nd most abnormal metric `wchar` (the number of bytes which the program has caused, or will cause to be written to disk), the flagged code region is also the `HRegion` class. This confirms that `HRegion` is where the developer needs to focus her attention.

The bug patch shows that the bug resides in the `HRegion` class. This class stores data for a certain table region and all columns for each row—a given table consists of one or more HRegions. The patch flips the order of acquiring the two locks (a write lock and then a read lock) and consequently the order of releasing them. It puts the change in both the `HRegion.put` and the `HRegion.close` methods. We speculate that spinning on locks in the deadlock situation causes the `user_time` metric to go awry.

The abnormal methods within `HRegion` that are flagged by ORION are `getRegionName`, `isClosed` and `toString` (Figure 6.8). They do not correspond to the methods where the bug lies (`put` and `close`). Through a detailed investigation, we identify the cause of this. The three flagged methods are invoked much more often within `HRegion` than are the erroneous methods. However, the three flagged methods and the erroneous methods occur close together in time. The algorithm in ORION, after it has zoomed into a time window where the fault manifested itself, considers frequencies of methods within that suspect time window to decide which methods to flag. This causes it to flag the most frequently invoked `getRegionName`, `isClosed` and `toString` methods.

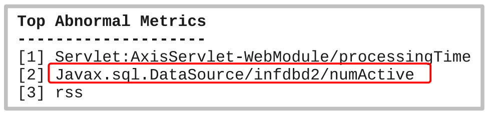
6.4.3 Case 3: StationsStat

StationsStat is a Java multi-tier application that is used to check the availability of workstations on Purdue's computing labs. Students on the campus use StationsStat daily to check the number of available Windows or Mac workstations for each lab on campus. StationsStat is managed by Purdue's IT department (ITaP) and runs in Apache Tomcat 5.5 on RedHat Enterprise Linux 5 with an in-memory SQL DB.

Due to an unknown bug, periodic failures were observed in which the application became unresponsive. System administrators received failure reports from their monitoring system, Nagios, or from user phone calls. Since the problem root cause was unknown, the application was restarted and the problem appeared to go away temporarily. StationsStat's administrators tracked 495 metrics from the OS, middleware, and application layers at 1 minute intervals (using asynchronous profiling) for more than two months. We collect the application, OS and middleware metrics in Table 6.4.

StationsStat was a challenging scenario, not only because the problem's root-cause was unknown, but also because there was no error-free data available to create the normal-behavior hyper-sphere. Fortunately, ORION can still work in this scenario by using *almost error-free* data. The administrators noticed that, after restarting StationsStat, the next failure was often seen only after a week or more—symptoms seemed to suggest that the problem, possibly a resource exhaustion bug, grew progressively from a service restart to a failure. ORION therefore used a data segment collected right after a restart to build the hyper-sphere representing normality. ORION also filtered out constant metrics in this phase which resulted in 70 non-constant metrics for the rest of the analysis.

In contrast with the previous cases, we conducted a blind experiment in which ORION gives us the suspicious metrics without us knowing the actual root-cause of the problem. Figure 6.9 shows the abnormal metrics that ORION finds. We then compared GRIFFIN's answer to the application developers best guess of the root-cause.



```
Top Abnormal Metrics
--------------------
[1] Servlet:AxisServlet-WebModule/processingTime
[2] Javax.sql.DataSource/infdbd2/numActive
[3] rss
```

Fig. 6.9.: Results from Orion for the StationsStat case.

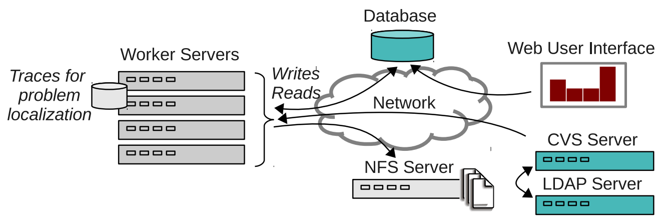


Fig. 6.10.: Mambo Health Monitoring system.

The results show that the suspicious metrics given by Orion matched well with what the developer thought to be the origin of the problem. The 2nd abnormal metric is the number of active SQL connections. The application had only one localized region where it made calls to the SQL driver that Tomcat used to handle database connections. The developer concluded that the SQL driver code was buggy since it was obvious that the few lines of SQL driver invocation code in his application was not. Upgrading the SQL driver fixed the problem and the application continues to run today providing an important function to students all over campus. Interestingly, the top metric flagged by Orion—the processing time of a servlet— had nothing to do with the bug. We found that this is due to large differences in workload between our normal and abnormal data sets (normal data were collected right after the server restart, while abnormal data were collected after the server has been up for a while). This negative result highlights the importance of getting the normal and the abnormal data sets under similar workload conditions.

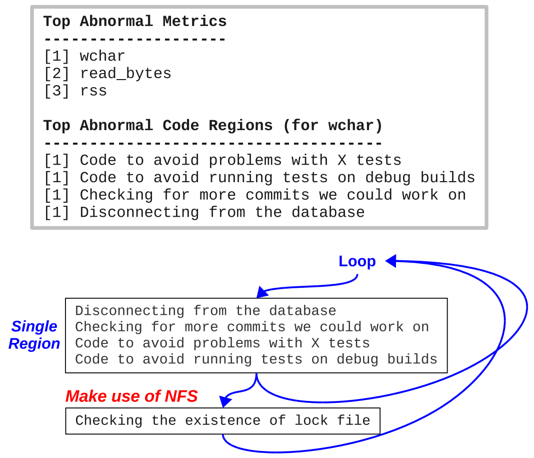
In this case study, there was no need for the additional step of Orion where it maps the abnormal metric to a code region. This is because only one small localized region of the application code had anything to do with SQL, which was implicated by the metric.

6.4.4 Case 4: Mambo Health Monitor

The Mambo Health Monitor (MHM) is a regression test system for the IBM Full System Simulator, commonly known as Mambo. Mambo is a computer architecture simulator for systems based on IBM's Power(TM) architecture. Mambo has been used in the development and testing of a wide range of systems, including IBM's Power line of server systems (Power5, Power6, Power7), the Cell(TM) processor used in the Sony Playstation3(TM), and IBM's BlueGene systems. The MHM executes tests on the simulator to detect regressions in behavior that may be introduced by new development. The tests are drawn from a test suite that covers the key functionality in all the major target systems. Test results are stored in an SQL database and are accessed through a web-based interface. Figure 6.10 shows the system's elements.

There is a servers farm which serve as MHM clients. Each client accesses a database to determine which tests have to be run. The client then checks out the code from a CVS repository (after authentication) and proceeds to run the test. The test execution sometimes requires specialized resources from the node on which it is executing, such as, a virtual network port. Upon completion, the client writes results in the database (success or failure) along with some informational items, such as, performance results.

Failures. A test-case can fail due to a problem in the environment or a problem with the architecture being simulated. Examples of environment-related problems causing a test to fail are many: a NFS connection that fails intermittently, a cron job fails to get authenticated with the LDAP server, Linux failing to map the simulator's network port to the machine's network port, /tmp filled up. A problem like this can make a developer falsely believe that her architecture code is buggy when in reality the problem lies in the environment. A key source of difficulty is that these problems are often transient and the software elements do not have error messages that correspond to the actual problem. We choose the problem of losing NFS mount as it has been a frequent problem for users over its seven year lifespan. We emulate NFS problems by dropping outgoing NFS packets with a probability of 0.1. The NFS packet dropping functionality is implemented by adding an iptables rule at the start of the faulty run.



```
Top Abnormal Metrics
--------------------
[1] wchar
[2] read_bytes
[3] rss

Top Abnormal Code Regions (for wchar)
-------------------------------------
[1] Code to avoid problems with X tests
[1] Code to avoid running tests on debug builds
[1] Checking for more commits we could work on
[1] Disconnecting from the database
```

Loop
Single Region
Disconnecting from the database
Checking for more commits we could work on
Code to avoid problems with X tests
Code to avoid running tests on debug builds
Make use of NFS
Checking the existence of lock file

Fig. 6.11.: Results from ORION for the MHM problem.

Code annotations. We run ORION in an asynchronous mode. The profiling process collects metrics at a 1 sec granularity. MHM was instrumented at 48 instrumentation points in 1400 lines of Tcl source code, which resulted in 2227 records. The Tcl script invokes Perl and bash scripts. Since these had been rigorously tested, we were told that they should be kept out of scope of our problem localization effort. The instrumentation code records a timestamp and an identifier for the code region. Unlike in the other applications, this code did not have finely granular methods (and of course no classes). Therefore, the points to insert instrumentation was a subjective decision and this was done based on the amount of comments in the code. Our instrumentation covers the starts and the ends of crucial operations, such as, CVS operations, NFS operations, and database operations, also other structures, such as `if-then-else` blocks.

Results. We collect traces of a normal and of a failed run. We use the same machine as the MHM client and keep the workload pattern the same. Figure 6.11 shows the results of applying ORION. First, notice that the two top abnormal metrics are metrics related to I/O, i.e., `wchar` and `read_bytes` (written characters to disk and read bytes from disk, respectively). Next, we find abnormal code regions using `wchar` as our abnormal metric—ORION ranks equally four different code regions as the figure shows. Notice that none of the pinpointed regions perform any operation that makes use of NFS, the root cause. However, when we look at the code (Figure 6.11), we notice that these regions are short and are always executed together inside a loop, so they can be grouped into one region. We also notice that, right after this (grouped) region, there is a code region that makes use of the NFS, i.e., the `Checking-the-existence-of-lock-file` region. This region performs I/O to access a file that is mounted using NFS and is affected by the injected fault.

The reason the NFS region is not ranked as an abnormal region (in fact is ranked 4) is that measurement inaccuracies emerge from asynchronous profiling. These code regions (demarcated by our instrumentation points) are small compared to the frequency of metric collection. Hence, it becomes difficult to accurately map the metric collection points to within an instrumented code region. In these cases, ORION pinpoints to the user not only the abnormal code region but also one region before and one region after the abnormal one. Hence, a design decision in ORION is that when asynchronous profiling is used and the instrumented code region is “small”, ORION pinpoints to the user not only the abnormal code region but also one region before and one region after the abnormal one. This strategy works well here since the code region that is affected by the fault is right after the region that ORION selects as abnormal.

6.4.5 Summary of Cases 5–7

Case 5. This bug (HBASE-3098 report) manifests itself as an application hang in HBase version 0.90. ORION results suggest using the `vsize` metric as the abnormal metric to find the abnormal code regions. ORION ranks the following classes as the top-3 most abnormal: `HRegionServer`, `ZooKeeperWatcher`, `HBaseServer`. The fixing patch for this bug (suggested by the developers) contains changes to 12 Java classes, one of this being `HRegionServer`. We noticed that the the `Hmaster` class, which is ranked as number 4 in ORION, is also part of the patch. Two buggy code regions showed up in top 5 results. There are a total of 131 classes that the developer would need to analyze to fix this bug. ORION reduces the search space substantially to a handful of classes.

Case 6. This bug (HBASE-6305 report) manifests itself as a hang in the `TestLocalHBaseCluster` test in Hbase version 0.94.3. According to the bug report, the origin of the fault appears to be a stack overflow error. Using ORION we find the following top-3 abnormal code regions: `HBaseServer`, `HRegionServer`, `ServerName`. The suggested patch fixes only two classes—one of those is `HRegionServer` which was ranked by ORION as the second most abnormal class. There is a total 155 classes that the developer would need to analyze in order to fix this bug without ORION.

Case 7. This bug (HBASE-7578) was one of the most difficult to debug since it only manifested occasionally (in the `TestCatalogTracker` test case) as a hang. The top-3 abnormal classes given by ORION did not contain the origin of the bug, i.e., they were not changed by the suggested patch. However, ORION was able to show `CatalogTracker` which was part of the patch, in its top-5 results. Also notice that the developer would have had to analyze more than 44 classes to fix this bug without the use of ORION.

6.4.6 Performance

We measure the time ORION spends in *bug localization*. This involves creating the hyper-sphere based on normal-behavior traces, finding the abnormal metric and, subsequently, the abnormal code region. Table 6.3 shows our measurements. The localization time depends on the amount of traces and metrics that are collected in a bug case. The amount of traces depends on the number of instrumented functions and their call frequency.

Table 6.3.: Performance measurements.

Case No.	**Bug-report ID**	**Bug localization time (min)**
1	Hadoop-3067	4.06
2	HBase-2097	15.32
3	StationsStat	31.04
4	MHM	0.48
5	HBase-3098	4.90
6	HBase-6305	10.35
7	HBase-7578	2.10

6.5 Discussion

We discuss practical limitations of this work:

- We observe that, as Orion drills down deeper looking for problematic code regions (class→subclasses→method), it provides less accurate results. An example is the Hadoop bug in which we are able to identify the abnormal code region at a Java class granularity but not at a method granularity.

- Applications that do not provide code-region delimiters would require manual effort from the developer (like with the MHM system) to indicate what are good instrumentation points. For most applications, however, Orion automatically annotates entry and exit points of methods.

- A trade-off of our asynchronous profiling approach is the difficulty of mapping metric samples to code regions accurately, when the code region is short relative to the time it takes to sample metrics. However, this comes with the advantage of minimal perturbation of the application. For asynchronous profiling, Orion provides code regions that are adjacent to whatever code region it finds as abnormal. For example, if the bug arises only when a particular package is updated, she could only instrument that package to look for the bug.

- Although in bug cases 1, 2 ,5–7 we collect metrics from a single node (due to the nature of the test cases), our approach also works for multi-node metrics. In such a case, Orion would have to label metrics from different nodes and treat them as different metrics (or features). Cases 3 and 4 are fully distributed and do not need this special treatment.

In this work, we proposed Orion to perform root cause analysis for failures in distributed applications. From a comprehensive set of monitored metrics, Orion pinpoints the metric and a window that is most highly affected by a failure and subsequently highlights the code region that is associated with the problem's origin. Our algorithm models the application behavior through pairwise correlations of multiple metrics, and when failure occurs, it finds the correlations (and hence the metrics) that deviate from normality. Our case studies with different distributed applications show the utility of the tool—Orion can localize the origin of real-world failures at a granularity of metrics and code regions in few minutes.

Appendix

Table 6.4.: Some of the used metrics.

Hardware Metrics	**OS Metrics**
L1_DCM	minor_faults
L2_DCM	major_faults
L2_ICM	user_cpu_time
L1_TCM	sys_cpu_time
L2_TCM	num_threads
CA_SHR	virt_mem_size
CA_CLN	rss_mem_size
CA_ITV	processor
TLB_DM	stack_size
TLB_IM	read_chars
L1_LDM	write_chars
L1_STM	read_bytes
L2_LDM	write_bytes
L2_STM	canceled_write_bytes
HW_INT	num_file_desc
BR_CN	nicRcvBytes
BR_TKN	nicRcvPckts
BR_NTK	nicSentBytes
BR_MSP	nicSentPckts
BR_PRC	IPInTruncatedPkts
TOT_IIS	IPInOctets
TOT_INS	IPOutOctets
VEC_DP	**Application Metrics**
FP_INS	servlet_processingTime
LD_INS	servlet_maxTime
SR_INS	servlet_requestCount
BR_INS	servlet_errorCount
VEC_INS	datasource_maxWait
RES_STL	datasource_numIdle
TOT_CYC	datasource_maxActive
L1_DCH	datasource_numActive
VEC_SP	**Middleware Metrics**
L1_DCA	request_handler_bytesSent
L2_DCA	request_handler_bytesReceived
L2_DCR	request_handler_requestCount
L2_DCW	request_handler_maxTime
L1_ICH	request_handler_processingTime
L2_ICH	request_handler_errorCount
L1_ICA	cache_hits
L2_ICA	cache_accesses
L2_TCH	number_threads

7. CONCLUSION

With the increasing scale and complexity of distributed systems, characterization and detection of software bugs is becoming increasingly challenging. Due to the market pressure, often software is pushed into a production environment with little or no failure characterization. Detection mechanisms have a greater impact, if characterization data, is used to build the detection techniques. Additionally, providing diagnostic-context for the programmers helps in the manual process of debugging.

Our work on characterization and detection of both programming and configuration bugs give key insights for providing reliability to web applications and their underlying infrastructure. For programming bugs, we characterized silent and non-silent failures, using a fault-injection methodology, in a three-tier web service application. An important lesson learned is that a caller method should flag a returned *null* as an incorrect operation and indicate a failure notification to the end user. An application developer should log the failure case, thus causing a non-silent error. While this happens with some methods, further validation that the returned object type is correct and its values are reasonable, is not done natively in the studied application. Such validation should also be a part of careful application development. The validation can be created through static analysis of source code, but for increased reliability, there is a need of application-specific checks inserted by the developer similar to the ones we show in our work.

For configuration bugs, our analysis using bug-repositories, in four orthogonal dimensions: *problem-type, problem-time* , *problem-manifestation*, and *problem-culprit* revealed key findings for Java EE application servers: (1). More than one-third of problems reported by users are configuration-related, implying that configuration-perspective in software development requires considerable attention. (2). Inter-version problems are majorly parameter related, thus requiring more consistency in configuration-design from version-to-version. (3). Intra-version configuration-problems have an almost equal share for each of the *parameter, compatibility, missing-component* subtypes. (4). Code refactoring or re-implementation results in an increased number of *compatibility* issues. Our subsequent configuration robustness testing tool, `ConfGauge` helps in choosing a server that will be prone to lesser configuration problems.

Besides detecting programming mistakes by developers and configuration mistakes by operators, we also make contributions in detecting duplicate client requests that can cause performance problems in heavily loaded web sites. Duplicate requests cause an artificially high load on servers and can corrupt server and client state. The two root causes of duplicate client browser requests are incorrect handling of certain HTML tags by browsers, when tag values are empty, and unintentional inclusion of same JavaScript more than one time. Since there is no obvious manifestation of duplicate web requests, this problem has received little attention in the literature. Our work, GRIFFIN addresses this problem by capturing a signature, in form of a signal for each web request and using an autocorrelation-based algorithm to detect unneeded web requests from clients. GRIFFIN detects with an accuracy of 78% and with a low overhead of 29%. More importantly, GRIFFIN did not give any false-positives, a characteristic that is taken into consideration when deploying it in environments where subsequent remedial actions are taken.

In addition to detecting performance problems, automated solutions for finding the root cause of performance problems is addressed by our work called, ORION. By capturing metrics across system layers, ORION provides diagnostic-context. It associates these metrics to code regions and builds a normal application profile based on pair-wise metric correlation. Our correlation-based analysis technique first pinpoints the metrics that show abnormal temporal patterns and deviate from the normal application profile. ORION then determines the code-region responsible for the abnormal deviation. We present the top three code regions to the developer as the diagnostic-context.

8. FUTURE WORK

Our observations and insights from this work create the investigation of the following important research questions for the future:

- ***Do data races give both silent and non-silent manifestations? If yes, what is the typical failure distribution?*** Our current work does not apply to failures as a result of data races. We hypothesize that, given a data race happens, it can give either a silent or a non-silent manifestation. Further, each of the silent and non-silent data races could possibly be harmful or benign. A harmful data-race in an enterprise application can result in an unintentional financial loss to an entity, while a benign data race does not cause a financial loss, though it still results in an incorrect functionality of the application. An investigation of this hypothesis for concurrency bugs can provide important characterization data that can subsequently lead to more accurate detection schemes for concurrency bugs.

- ***Given source code or even bytecode, can useful detection rules be extracted statically?*** Currently, for application-specific checks, we require a normal run of an application workload to extract the normal length. Using control-flow and data-flow analysis on source code can help to build rules and eliminate the dependency on a normal run of workload. Another related question for the future is, *can comments within source code help in detection of configuration bugs?* A typical Java EE application has several configurations at different layers in the software stack, e.g., application, application server, JVM and OS. Programmers often write important information as free text within program source code. An investigation on whether rules can be learned, from comments, to detect misconfigurations can help improve configuration robustness.

- ***Can the process of finding the configuration bug to be injected be automated?*** Currently, parsing through the bug reports to find the string value to inject for a given type of configuration is a manual process. In the future, automatically finding a string value which is known to be a common cause of the configuration bug can significantly reduce the time for which a configuration injecting tool runs.

- ***Can sampling help to reduce the runtime overhead of* GRIFFIN*?*** Currently, we intercept all function calls and track all the function call depths for the given runtime (PHP). Another parameter that can reduce the current overhead of 29% will be to sample the intercepted functions with the design goal of keeping the false-positives at 0%.

- ***How can the workload changes be incorporated into performance debugging?*** Significant changes in workload patterns can raise false-alarms in ORION. Currently, the changes to workload are handled by using pair-wise correlations of metrics with the hope that the correlation patterns of metrics stays the same even if the workload changes. In the future, detecting workload changes and retraining ORION for the "new" normal workload will help to reduce the false-positives.

- ***How to manage the scalability challenge in the performance debugging algorithms?*** As the size of the monitored application increases with multiples processes and threads across a large number of hosts, computing pair-wise metric correlations across time windows of multiple sizes gets increasingly expensive. To deal with this in the future, either the input data to the algorithm should be aggregated, or approaches to finding an optimum window-size should be designed.

LIST OF REFERENCES

LIST OF REFERENCES

[1] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, “Dynamic instrumentation of production systems,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247415.1247417

[2] F. C. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston, and B. Chen, “Architecture of systemtap: a linux trace/probe tool,” 2005.

[3] “DayTrader - a sample Java EE benchmark application,” https://cwiki.apache.org/GMOxDOC20/daytrader.html.

[4] J. Duraes and H. Madeira, “Emulation of software faults: A field data study and a practical approach,” *Software Engineering, IEEE Transactions on*, vol. 32, no. 11, pp. 849 –867, 2006.

[5] “GlassFish - an application server for Java EE platform,” http://glassfish.java.net.

[6] “Jboss.” [Online]. Available: http://www.jboss.org/

[7] “Empty image src can destroy your site,” http://www.nczonline.net/blog/2009/11/30/empty-image-src-can-destroy-your-site/.

[8] “Empty SRC And URL() Values Can Cause Duplicate Page Requests,” http://www.bennadel.com/blog /2236-Empty-SRC-And-URL-Values-Can-Cause -Duplicate-Page-Requests.htm.

[9] Steve W, “Monkey Code,” http://code.alittlegoofy.com/2008/12/i-found-something-peculiar-about.html.

[10] S. Souders, “High-performance web sites,” *Commun. ACM*, vol. 51, no. 12, pp. 36–41, Dec. 2008. [Online]. Available: http://doi.acm.org/10.1145/1409360.1409374

[11] D. Cotroneo, S. Orlando, and S. Russo, “Failure classification and analysis of the java virtual machine,” in *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, 2006, p. 17.

[12] J. Li, G. Huang, J. Zou, and H. Mei, “Failure analysis of open source j2ee application servers,” in *Quality Software, 2007. QSIC ’07. Seventh International Conference on*, 2007, pp. 198 –208.

[13] H. Madeira, D. Costa, and M. Vieira, “On the emulation of software faults by software fault injection,” in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, 2000, pp. 417 –426.

[14] M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia, "Assessing and improving the effectiveness of logs for the analysis of software faults," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, 28 2010-july 1 2010, pp. 457 –466.

[15] C. Shelton, P. Koopman, and K. Devale, "Robustness testing of the microsoft win32 api," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, 2000, pp. 261 –270.

[16] W. Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Z. Yang, "Characterization of linux kernel behavior under errors," in *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, june 2003, pp. 459 – 468.

[17] S. Bagchi, Y. Liu, K. Whisnant, Z. Kalbarczyk, R. Iyer, Y. Levendel, and L. Votta, "A framework for database audit and control flow checking for a wireless telephone network controller," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, july 2001, pp. 225 –234.

[18] M. 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*, 2002, pp. 595 – 604.

[19] L. M. Silva, "Comparing error detection techniques for web applications: An experimental study," in *Proceedings of the 2008 Seventh IEEE International Symposium on Network Computing and Applications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 144–151.

[20] E. Kiciman and A. Fox, "Detecting application-level failures in component-based internet services," *Neural Networks, IEEE Transactions on*, vol. 16, no. 5, pp. 1027 –1041, 2005.

[21] M. Vieira, N. Laranjeiro, and H. Madeira, "Assessing robustness of web-services infrastructures," in *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, june 2007, pp. 131 –136.

[22] N. Laranjeiro, M. Vieira, and H. Madeira, "Robustness validation in service-oriented architectures," in *Architecting Dependable Systems VI*, ser. Lecture Notes in Computer Science, R. de Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, and M. ter Beek, Eds., vol. 5835. Springer Berlin / Heidelberg, 2009, pp. 98–123.

[23] M. Torchiano, F. Ricca, and A. Marchetto, "Are web applications more defect-prone than desktop applications?" *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 13, pp. 151–166, 2011, 10.1007/s10009-010-0182-6. [Online]. Available: http://dx.doi.org/10.1007/s10009-010-0182-6

[24] S. Chandra and P. Chen, "Whither generic recovery from application faults? a fault study using open-source software," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, 2000, pp. 97 –106.

[25] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 159–172.

[26] L. Keller, P. Upadhyaya, and G. Candea, "Conferr: A tool for assessing resilience to human configuration errors," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, june 2008, pp. 157 –166.

[27] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the USENIX Annual Technical Conference*, 2011, pp. 28–28.

[28] S. Zhang, "Confdiagnoser: an automated configuration error diagnosis tool for java software," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1438–1440.

[29] Y.-Y. Su, M. Attariyan, and J. Flinn, "Autobash: improving configuration management with operating system causality analysis," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 237–250, Oct. 2007.

[30] J. Li, G. Huang, J. Zou, and H. Mei, "Failure analysis of open source j2ee application servers," in *Quality Software, 2007. QSIC '07. Seventh International Conference on*, 2007, pp. 198–208.

[31] "Scale your Applications on the Web," https://www.openshift.com/developers/scaling.

[32] F. Oliveira, A. Tjang, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Barricade: defending systems against operator mistakes," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 83–96.

[33] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 193–202.

[34] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 17–17.

[35] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration debugging as search: finding the needle in the haystack," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 6–6.

[36] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–11.

[37] V. Cerf and R. Kahn, "A protocol for packet network intercommunication," *Communications, IEEE Transactions on*, vol. 22, no. 5, pp. 637–648, May 1974.

[38] N. T. Spring and D. Wetherall, “A protocol-independent technique for eliminating redundant network traffic,” *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 4, pp. 87–95, Aug. 2000. [Online]. Available: http://doi.acm.org/10.1145/347057.347408

[39] E. Vallés and P. Rosso, “Detection of near-duplicate user generated contents: The sms spam collection,” in *Proceedings of the 3rd International Workshop on Search and Mining User-generated Contents*, ser. SMUC ’11. New York, NY, USA: ACM, 2011, pp. 27–34. [Online]. Available: http://doi.acm.org/10.1145/2065023.2065031

[40] B. Coskun and P. Giura, “Mitigating sms spam by online detection of repetitive near-duplicate messages,” in *Communications (ICC), 2012 IEEE International Conference on*, June 2012, pp. 999–1004.

[41] M. Himmel, R. Hoffman, and M. Mall, “System and method for preventing duplicate transactions in an internet browser/internet server environment,” May 22 2001, uS Patent 6,237,035. [Online]. Available: http://www.google.com/patents/US6237035

[42] J. Mogul and J. C. Mogul, “Trace-based analysis of duplicate suppression in http,” http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-99-2.html, 1999.

[43] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, “Capturing, indexing, clustering, and retrieving system history,” in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP ’05. New York, NY, USA: ACM, 2005, pp. 105–118. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095821

[44] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting large-scale system problems by mining console logs,” in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP ’09. New York, NY, USA: ACM, 2009, pp. 117–132. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629587

[45] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, “Fingerprinting the datacenter: Automated classification of performance crises,” in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys ’10. New York, NY, USA: ACM, 2010, pp. 111–124. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755926

[46] Q. Fu, J.-G. Lou, Q.-W. Lin, R. Ding, D. Zhang, Z. Ye, and T. Xie, “Performance issue diagnosis for online service systems,” in *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on.* IEEE, 2012, pp. 273–278.

[47] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, “Friday: global comprehension for distributed replay,” in *NSDI*, 2007.

[48] D. Geels, G. Altekar, S. Shenker, and I. Stoica, “Replay debugging for distributed applications,” in *USENIX ATC*, 2006.

[49] M. Attariyan, M. Chow, and J. Flinn, “X-ray: Automating root-cause diagnosis of performance anomalies in production software,” in *Proc. OSDI*, 2012.

[50] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: finding liveness bugs in systems code," in *Proc. NSDI*, 2007, pp. 18–18.

[51] M. S. Musuvathi, D. Park, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, "Cmc: A pragmatic approach to model checking real code," in *Proc. OSDI*, 2002.

[52] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3s: debugging deployed distributed systems," in *Proc. NSDI*, 2008, pp. 423–437.

[53] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proc. ICDM*, 2009, pp. 149–158.

[54] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proc. NSDI*, 2012, pp. 26–26.

[55] S. Sabato, E. Yom-Tov, A. Tsherniak, and S. Rosset, "Analyzing system logs: a new view of what's important," in *Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, ser. Proc. SYSML'07, 2007, pp. 6:1–6:7.

[56] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. SOSP*, 2009, pp. 117–132.

[57] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *OSDI*, 2004.

[58] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based faliure and evolution management," in *Proc. NSDI*, 2004, pp. 23–23.

[59] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *Proc. NSDI*, 2011, pp. 4–4.

[60] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises," in *Proc. EuroSys*, 2010, pp. 111–124.

[61] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *Proc. SOSP*, 2005, pp. 105–118.

[62] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *Proc. ICDCS*, 2009, pp. 623–630.

[63] S. Zhang, I. Cohen, J. Symons, and A. Fox, "Ensembles of models for automated diagnosis of system performance problems," in *DSN*, 2005, pp. 644–653.

[64] M. Jiang, M. Munawar, T. Reidemeister, and P. Ward, "Dependency-aware fault diagnosis with metric-correlation models in enterprise software systems," in *Network and Service Management (CNSM), 2010 International Conference on*, 2010, pp. 134 –141.

[65] A. Chester, J. Xue, L. He, and S. Jarvis, "A system for dynamic server allocation in application server clusters," in *Parallel and Distributed Processing with Applications, 2008. ISPA '08. International Symposium on*, 2008, pp. 130 –139.

[66] E. Altman, M. Arnold, S. Fink, and N. Mitchell, "Performance analysis of idle programs," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 739–753.

[67] B. Jacobs, P. Muller, and F. Piessens, "Sound reasoning about unchecked exceptions," in *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, 2007, pp. 113 –122.

[68] F. Arshad and S. Bagchi, "Dangers and joys of stock trading on the web: Failure characterization of a three-tier web service," in *Reliable Distributed Systems, 2011. SRDS 2011. 30th IEEE International Symposium on*, oct. 2011, pp. 1 –10.

[69] "The grinder, a java load testing framework," http://grinder.sourceforge.net/.

[70] "The dacapo benchmark suite," http://www.dacapobench.org/.

[71] "Apache Derby - an open source relational database," http://db.apache.org/derby.

[72] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, ser. USITS'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–1.

[73] P. Thibodeau, "Amazon cloud outage was triggered by configuration error," April 2011. [Online]. Available: http://www.computerworld.com/s/article/9216303/Amazon_cloud_outage_was_triggered_by_configuration_error

[74] F. Foo, "Human error triggered nab software corruption," November 2010. [Online]. Available: http://www.theaustralian.com.au/australian-it/human-error-triggered-nab-software-corruption/story-e6frgakx-1225962953523

[75] N. Eddy, "Human error caused google glitch," February 2009. [Online]. Available: http://www.eweek.com/c/a/Enterprise-Applications/Human-Error-Caused-Google-Glitch/

[76] B. Hale, "Why every it practitioner should care about network change and configuration management," February 2012. [Online]. Available: http://web.swcdn.net/creative/pdf/Whitepapers/Why_Every_IT_Practitioner_Should_Care_About_NCCM.pdf

[77] "Java ee jsrs." [Online]. Available: http://jcp.org/en/jsr/platform?listBy=3&listByType=platform

[78] "Oracle jrockit jvm." [Online]. Available: http://www.oracle.com/technetwork/middleware/jrockit/overview/index.html

[79] "Glassfish bug repository." [Online]. Available: http://java.net/jira/browse/GLASSFISH

[80] "Jboss bug repository." [Online]. Available: https://issues.jboss.org/browse/JBAS

[81] "How do i migrate my application from as5 or as6 to as7." [Online]. Available: https://docs.jboss.org/author/display/AS7/How+do+I+migrate+my+application+from+AS5+or+AS6+to+AS7

[82] "Jbas-1115: bad path to included xsd gets built in wsdlfilepublisher." [Online]. Available: https://issues.jboss.org/browse/JBAS-1115

[83] "Geronimo-3921: getcontextroot() returns forward slash rather than empty string for apps deployed to root context." [Online]. Available: https://issues.apache.org/jira/browse/GERONIMO-3921

[84] "Glassfish-6822: Windows: package-appclient must not show forward slash." [Online]. Available: https://java.net/jira/browse/GLASSFISH-6822

[85] "Glassfish-16039: Jms, problem with destination sources having jndi-name with forward-slash." [Online]. Available: https://java.net/jira/browse/GLASSFISH-16039

[86] "Openejb-1709: Tomee webapps (see rest-example) doesn't work under windows (path - problem with backslash '́)." [Online]. Available: https://issues.apache.org/jira/browse/OPENEJB-1709

[87] "Resteasy-656: Context path without trailing slash doesn't work with applicationpath('/')." [Online]. Available: https://issues.jboss.org/browse/RESTEASY-656

[88] "Opensso-544: missing slash in resource names." [Online]. Available: https://java.net/jira/browse/OPENSSO-544

[89] "Sailfin-1556: Slash in sip uri gives error." [Online]. Available: https://java.net/jira/browse/SAILFIN-1556

[90] "Javaserverfaces-1146: Htmlresponsewriter renders unnecessary '/' symbols." [Online]. Available: https://java.net/jira/browse/JAVASERVERFACES-1146

[91] "Glassfish-2778: Treat empty-string virtual-servers attribute in ¡application-ref¿ as identical to virtual-servers attribute missing." [Online]. Available: https://java.net/jira/browse/GLASSFISH-2778

[92] "As7-5550: servlet filter mapping empty string not working." [Online]. Available: https://issues.jboss.org/browse/AS7-5550

[93] "Cargo api." [Online]. Available: http://cargo.codehaus.org

[94] "Specjenterprise2010." [Online]. Available: http://www.spec.org/jEnterprise2010

[95] "Glassfish verifier: verify-domain-xml." [Online]. Available: http://docs.oracle.com/cd/E19879-01/820-4337/beadq/index.html

[96] "Suggestimate: Detect duplicate jira issues." [Online]. Available: http://www.plugenta.com/suggestimate/jira/

[97] "Heroku," https://www.heroku.com/.

[98] M. McLennan and R. Kennell, "Hubzero: A platform for dissemination and collaboration in computational science and engineering," *Computing in Science & Engineering*, vol. 12, no. 2, pp. 48–53, 2010. [Online]. Available: http://scitation.aip.org/content/aip/journal/cise/12/2/10.1109/MCSE.2010.41

[99] "HTML 4.01 Specification," http://www.w3.org/TR/html4/.

[100] T. J. Hacker, R. Eigenmann, S. Bagchi, A. Irfanoglu, S. Pujol, A. Catlin, and E. Rathje, "The neeshub cyberinfrastructure for earthquake engineering," *Computing in Science and Engg.*, vol. 13, no. 4, pp. 67–78, Jul. 2011. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2011.70

[101] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 2:1–2:49, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2110356.2110358

[102] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[103] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746

[104] "Systemtap call-depth feature request," https://sourceware.org/bugzilla/show_bug.cgi?id=16472.

[105] W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S.* Springer Publishing Company, Incorporated, 2010.

[106] R. Bordawekar, U. Bondhugula, and R. Rao, "Believe it or not!: mult-core cpus can match gpu performance for a flop-intensive application!" in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques.* ACM, 2010, pp. 537–538.

[107] N. Huber, M. S. Hromalik-Pouchet, T. D. Carozzi, M. P. Gough, and A. M. Buckley, "Parallel processing speed increase of the one-bit auto-correlation function in hardware," *Microprocess. Microsyst.*, vol. 35, no. 3, pp. 297–307, May 2011. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2011.01.001

[108] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[109] "Apache MPM prefork," http://httpd.apache.org/docs/2.2/mod/prefork.html.

[110] "HUBzero powered websites," https://hubzero.org/sites.

[111] "The GNU Project Debugger," http://www.gnu.org/software/gdb/.

[112] "Using JConsole to Monitor Applications," http://www.oracle.com/technetwork/articles/java/jconsole-1564139.html.

[113] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," *SIGPLAN Not.*, vol. 39, no. 4, pp. 49–57, Apr. 2004.

[114] ".NET Memory Profiler," http://memprofiler.com/.

[115] "JProfiler," http://www.ej-technologies.com.

[116] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: a pervasive network tracing framework," in *Proc. NSDI*, 2007, pp. 20–20.

[117] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem diagnosis in large-scale computing environments," in *SC*, 2006.

[118] "Apache HBase," http://hbase.apache.org/.

[119] "Apache Hadoop Project," http://hadoop.apache.org/.

[120] "Branch mispredition for unsorted array," http://stackoverflow.com/questions/11227809/.

[121] "PAPI," http://icl.cs.utk.edu/PAPI/.

[122] K. Bare, S. Kavulya, and P. Narasimhan, "Hardware performance counter-based problem diagnosis for e-commerce systems," in *Proc. NOMS*, 2010, pp. 551–558.

VITA

VITA

Fahad Arshad received his BSc degree in Electrical Engineering from the University of Engineering and Technology (UET), Lahore Pakistan, in 2006. He began working under the supervision of Professor. Saurabh Bagchi in Fall 2006 as a graduate student in the School of Electrical and Computer Engineering at Purdue University in West Lafayette, Indiana USA. His research interest lies in developing algorithms for software testing, error detection and failure diagnosis in distributed systems. His work has appeared at the top dependability conferences DSN, ICAC, SRDS, ISSRE and Middleware, and he has been awarded grants to attend ICAC 2014, DSN 2014, DSN 2012, ICNP 2012. He is also a student member of IEEE and Usenix.

PUBLICATIONS

PUBLICATIONS

CONFERENCE PAPERS

1. "*Is Your Web Server Suffering from Undue Stress due to Duplicate Requests?*, **Fahad A. Arshad**, Amiya K. Maji, Sidharth Mudgal, Saurabh Bagchi. In the 11th International Conference on Autonomic Computing (ICAC '14), 6 pages, Philadelphia, Pennsylvania USA, June 18 - 20, 2014.
2. "*pSigene: Webcrawling to Generalize SQL Injection Signatures*, Gaspar Modelo-Howard, **Fahad A. Arshad**, Christopher Gutierrez, Saurabh Bagchi. In the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 12 pages, Atlanta, Georgia USA, June 23 - 26, 2014.
3. "*Characterizing Configuration Problems in Java EE Application Servers: An Empirical Study with GlassFish and JBoss*, **Fahad A. Arshad**, Rebecca J. Krause, Saurabh Bagchi. Accepted to appear at the the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE), 10 pages, Pasadena, CA, USA, November 4-7, 2013.
4. "*Automatic Problem Localization in Distributed Applications via Multi-dimensional Metric Profiling*, Ignacio Laguna, Subrata Mitra, **Fahad A. Arshad**, Nawanol Theera-Ampornpunt, Zongyang Zhu, Saurabh Bagchi, Samuel P. Midkiff, Mike Kistler and Ahmed Gheith. Accepted to appear at the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS), 10 pages, Braga, Portugal, October 1-3, 2013.
5. "*An Empirical Study of the Robustness of Inter-component Communication in Android*, Amiya K. Maji, **Fahad A. Arshad**, Saurabh Bagchi and Jan S. Reller-

meyer. In the 42th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 12 pages, Boston, MA, June 25-28, 2012.

6. “*Dangers and Joys of Stock Trading on the Web: Failure Characterization of a Three-Tier Web Service*, **Fahad A. Arshad** and Saurabh Bagchi. In 30th IEEE Symposium on Reliable Distributed Systems (SRDS), 10 pages, Madrid, Spain, Oct 5-7, 2011.

7. “*How To Keep Your Head Above Water While Detecting Errors*, Ignacio Laguna, **Fahad A. Arshad**, David M. Grothe, and Saurabh Bagchi. In ACM/IFIP/USENIX 10th International Middleware Conference, November 30-December 4, 2009, Urbana-Champaign, Illinois.

8. “*Distributed Diagnosis of Failures in a Three Tier E-Commerce System*, Gunjan Khanna, Ignacio Laguna, **Fahad A. Arshad**, and Saurabh Bagchi. In 26th IEEE International Symposium on Reliable Distributed Systems (SRDS-2007), pp. 185-198, Beijing, CHINA, October 10-12, 2007.

9. “*Stateful Detection in High Throughput Distributed Systems*, Gunjan Khanna, Ignacio Laguna, **Fahad A. Arshad**, and Saurabh Bagchi. In 26th IEEE International Symposium on Reliable Distributed Systems (SRDS-2007), pp. 275-287, Beijing, CHINA, October 10-12, 2007.

WORKSHOP PAPERS

1. “*Lilliput meets Brobdingnagian: Data Center Systems Management through Mobile Devices*, Saurabh Bagchi, **Fahad Arshad**, Jan Rellermeyer, Thomas Osiecki, Michael Kistler, Ahmed Gheith. In Third International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (DCDV) June 24, 2013 Budapest, Hungary.

2. “*To Cloud or Not to Cloud: A Study of Trade-offs between In-house and Outsourced Virtual Private Network*, **Fahad A. Arshad**, Gaspar Modelo-Howard

and Saurabh Bagchi. In 7th IEEE Workshop on Secure Network Protocols (NPSec) 30-October 2012, Austin, Texas.