Practical End-to-End Performance Evaluation of Backend Software Applications

by

TULI NIVAS

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2013

Copyright © by Tuli Nivas 2013

All Rights Reserved

ACKNOWLEDGEMENTS

First and foremost I would like to thank my advisor Dr. Csallner. Without his constant guidance this work would have never got done. His knowledge in the field of software engineering and dedication towards his students is unparalleled. His suggestions to the technical challenges I faced during this work, the brainstorming sessions and his meticulous reviewing and editing of papers and this dissertation document has been an invaluable help. I would also like to thank my committee members - Dr. Walker, Dr. Elmasri and Dr. Khalili for taking the time to review this document and provide critique and advice. I want to especially thank Dr. Khalili for being so accommodating and considerate and for stepping in at the last minute to be part of my committee.

I would like to take this opportunity to thank the pLab and EI architecture teams at Sabre Holdings. They are some of the most innovative and smart people I have worked with. Parts of this dissertation are a result of the excellent team work and rapport I share with them. A special shout out to Alan Walker and Ross Darrow! They went out of their way to discuss issues and provide me with books and licenses to software that helped me complete this work. They provided me with brilliant ideas and their recommendations were crucial in getting this research done.

Last but not the least I want to express my deepest gratitude to my family and friends. I cannot thank my parents - Dr. Pratibha Srivastava and Shri Nivas and sister - Shruti Rawat enough, for always believing in me, supporting and encouraging me and for their never ending love. Their positive outlook in everything and faith in me has always motivated me to achieve my goals. My brother-in-law Vikram Rawat

was my sounding board. Discussing different ideas and solutions with him and his years of industry experience were a big help in putting all the pieces together for this work. I can never express enough appreciation to my family for always being there when I need them - through the highs and the lows. Huge thanks to all my friends too for being part of my life and cheering me on in my every endeavor.

November 15, 2013

ABSTRACT

Practical End-to-End Performance Evaluation of Backend Software Applications

Tuli Nivas, Ph.D.

The University of Texas at Arlington, 2013

Supervising Professor: Christoph Csallner

This dissertation makes contributions to four areas of performance testing - the test process itself, monitoring, automation and end-to-end performance evaluation of backend applications.

The first contribution deals with the testing process. Performance testing is a key element of industrial software development, but we still encountered the following two problems. (1) While testing textbooks prescribe writing tests against performance goals, we find that it is impractical to gather from business analysts performance goals that are detailed enough for finding subtle performance bugs. (2) Once performance tests are conducted, we were asked questions such as - how do you make sure that the results collected during testing are a good indication of how the code will function in production? To enable practitioners to address these problems, we introduce two additional performance testing process components, which we call release certification and test data correlation. Our key idea to address problem (1) is to run two different versions of the same subject application side-by-side in the same test environment and (2) is to correlate the performance measurements of the test and production environments. Running individual soak, load and stress tests to

assess different aspects of an application is a tedious process. So we introduce a new test - impulse test that not only combines the characteristics of multiple existing tests but also enables testing for engineering aspects such throttles, alerts and timeouts.

The second contribution is in the area of instrumentation. Monitoring the performance of distributed applications is an important task in practical software engineering. Current monitoring tools are often limited in the range of computing platforms they support, which limits their utility in several business monitoring scenarios. Current monitoring tools can also impose a significant performance overhead. We describe our in house built EI Enterprise Instrumentation monitoring tool that addresses these issues. We compare EI with a state-of-the-art monitoring tool on a real online shopping application and describe our experience with EI as well as feedback we received from EI users.

The third component of the study deals with test automation and scripting. Scarcity of commercially available testing tools that could support all native or application specific message formats as well as those that cater to non GUI or non-web based backend applications led to creating our own customized traffic generator and scripts. This study provides (1) the general design principles for a test script that can be used to generate traffic for any request format as well as (2) specific factors to keep in mind when creating a script that will work in a test environment that uses a mock. It will also address the (3) design and properties of a test harness. It provides a simple framework that can be easily used to complete an end-to-end testing process: pre test, traffic generation and post test activities.

Last but not the least the dissertation proposes a solution to assess performance for backend applications. State space models specifically Markov models are used extensively to predict anomalies, capacity and throughput in computer systems. In most cases existing solutions depend on underlying architecture and historical data

to create the models and then detect states that differ from the created profile. In this study we will use Markov models to proactively predict the performance of a software system by using client side input parameters and application attributes without the need to know the underlying architecture. The goal is to find pressure points/bottlenecks for the application when it interacts with other components. Pressure points are any application resource that can become exhausted thereby restricting or degrading service level performance such as CPU, memory, disk, network and so on.

All solutions proposed in this study have been implemented on real world travel applications, in most cases an airline shopping application. The data gathered and all measurements/plots in this study are from travel related pieces of code that are live and used by customers today.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

ABSTRACT v

LIST OF ILLUSTRATIONS xiii

LIST OF TABLES xxi

Chapter Page

1. INTRODUCTION 1

1.1 Performance Certification and Correlation 2

1.2 Designing Performace Tests 5

1.3 Monitoring and Instrumentation 7

1.4 Test Script Design and Automation 8

1.5 End-to-End Performance Evaluation of Applications using State Space Models 9

2. MANAGING PERFORMANCE TESTING WITH RELEASE CERTIFICATION AND DATA CORRELATION 12

2.1 Introduction 12

2.1.1 Solution Overview 16

2.2 Release Certification 18

2.2.1 Example: Certification for Airline A 20

2.3 Performance Testing For Release Certification 23

2.4 Designing an Impulse Test 25

2.4.1 Initial Peak Load 27

2.4.2 Traffic Surge 27

2.4.3 Traffic Drop 28
2.4.4 Subsequent Peak Load 28
2.5 Benefits of Impulse Test 28
2.5.1 Ignoring the Implementation of Throttles and Timeouts 30
2.5.2 Insufficient Alerts 31
2.6 Performance Measurements Using Impulse Tests 33
2.7 Impulse Test vs. Conventional Tests 39
2.8 Guidelines for designing thresholds, timeouts, and alerts 43
2.8.1 Thresholds 44
2.8.2 Timeouts 47
2.8.3 Alerting 47
2.9 Test Data Correlation 48
2.9.1 Example: Correlation for Airline A 50
2.10 Related Work 51
3. SOFTWARE TESTABILITY 53
3.1 Introduction 53
3.1.1 Example: Insufficient Instrumentation of a Commercial Hotel Booking Application 54
3.2 Software Testability 56
3.3 Characteristics of Software Applications that Affect Testability 59
3.3.1 What to Instrument? 62
3.3.2 NOFEP 64
3.3.3 Standard Instrumentation 68
3.4 Characteristics External to Software Application that Affect Testability 70
3.5 Environmental Characteristics that Affect Software Testability 74

4. ENTERPRISE INSTRUMENTATION: A LIGHTWEIGHT CROSS-PLATFORM LOW-OVERHEAD MONITORING TOOL FOR DISTRIBUTED SOFTWARE APPLCIATIONS 79
4.1 Introduction 79
4.2 Background: DynaTrace 82
4.3 Overview of Enterprise Instrumentation (EI) 83
4.4 EI Log Types and Configuration 86
4.4.1 Log Types 86
4.4.2 Configuration 87
4.5 EI Implementation 88
4.6 EI API Implementation Details 89
4.6.1 Summary Customer Metrics (SCM) 89
4.6.2 JVM Metrics 93
4.6.3 Billing 96
4.7 Experience in the Sabre Production Environment 99
4.8 Comparison With a Third-Party Monitoring Tool: DynaTrace 103
4.8.1 Cross-Platform Functionality 105
4.8.2 Runtime Overhead 106
4.9 Related Work 108
5. TEST HARNESS AND SCRIPT DESIGN PRINCIPLES FOR AUTOMATED TESTING 112
5.1 Introduction 112
5.2 Test Harness and Script Design Priciples 115
5.2.1 Test Harness Definitions 115
5.2.2 Test Script Design 118
5.3 Test Harness Example 128

5.4 Related Work 130
6. DESIGN OF EXPERIMENTS (DOE) FOR PERFORMANCE EVALUATION OF SOFTWARE APPLICATIONS 132
6.1 Introduction 132
6.2 Background - Terminology of DOE 136
6.2.1 One Factor Experimental Design 137
6.2.2 Factorial Experimental Design 138
6.2.3 Orthogonal Main Effect Designs - OMEDs and Taguchi Method 141
6.3 Design of Experiments for Performance Evaluation of Software Applications 145
6.3.1 Orthogonal Array DOE Setup for Evaluating Performance of Airline Shopping Application 150
7. STATE SPACE MODELS - MARKOV MODELS FOR PERFORMANCE EVALUATION OF SOFTWARE APPLICATIONS 158
7.1 Introduction 158
7.2 Background - Markov Models 158
7.2.1 Discrete Time Markov Chains 159
7.2.2 Calculating State Transition Probabilities 162
7.3 State Space Models for Evaluating Performance of Software Applications 163
7.4 Related Work 165
8. EVALUATING PRESSURE POINTS IN A COMPUTER SYSTEM FOR END TO END PERFORMANCE EVALUATION 169
8.1 Introduction 169
8.2 Background - What is a Pressure Point or a Bottleneck? 173
8.3 Pressure Points in Computer Systems 177
8.4 Characteristics and Behaviors of Software Application Pressure Points 182

8.5 End-to-End Performance Evaluation of Software Applications 184
8.5.1 Creating a Node Cluster 185
8.5.2 Creating the State Space Model 188
8.5.3 Finding Pressure Point States in the State Space Model 192
8.6 Simulation Setup 193
8.7 Simulation Results 200
8.8 Proposed Solution vs. Conventional Performance Tests 205
8.9 Performance Prediction Using SSMs 209
REFERENCES 220
BIOGRAPHICAL STATEMENT 228

LIST OF ILLUSTRATIONS

Figure Page

1.1 Multi-tier online-shopping application from the airline travel industry 3

1.2 Booking transactions in four related travel shopping applications (air, hotel, car, and cruise) over eleven months. 6

2.1 Performance bug in an airline ticket sales back-end application hosted by Sabre. CPU usage (thin lines) is similar between release 8.08 and release 10, but virtual memory (dotted) and residual memory (broad lines) differ significantly. This bug was found with fine-grained performance goals for virtual and residual memory. These performance goals did not appear in any SLA but were derived from the performance of release 8.08. 15

2.2 Performance testing workflow extended with release certification and data correlation components. 17

2.3 Shopping Application Data Flow Path Showing Backend Applications - Availability and Schedule Finder along with the Database Server Pool. The Server Pool of Shopping Application is Made Up of Shopping Historical (HIST), Shopping Domestic (IS) and Shopping International (MIP) Components, Each having Separate Releases. 21

2.4 Shopping Application Setup in Test Environment with Two Identical Test Paths - One for Current Release and Second for New Release. . . 23

2.5 Booking transactions in four related travel shopping applications (air, hotel, car, and cruise) over eleven months. 25

2.6 Impulse test design. A long period of peak load is followed by a relatively short but drastic traffic surge, followed by a brief drop and a subsequent long pre-surge peak load; TPS = transactions per second. . 26
2.7 Impulse test design. Impulse test with incremental traffic increase after the surge to the pre-surge peak level. 29
2.8 Impulse test design. Impulse test with two traffic surges. 29
2.9 The Customer Insight application communicates with ICE for session authentication, USG for user communication, and Provider for user specific data. 30
2.10 Simplified architecture of Sabre Sonic Web (SSW). Users communicate with SSW via a firewall and SSW gets user specific information from the SSW database. 33
2.11 The application throttles requests during the traffic surge as expected. 34
2.12 The application produces and logs alerts for exceeded thresholds and exceptions during the traffic surge as expected. 35
2.13 Application TPS decreases during the traffic surge as expected, but does not recover after the traffic surge to before-surge levels, which indicates a performance bug; TPS = transactions per second. 36
2.14 Response times for Release 4.2.2(dotted lines) vs. Response times for Release 3.21.3 (solid lines). 36
2.15 Heap usage increased from ASv2 release 3.21.3 (red) to release 4.2.2 (blue). 37
2.16 Both GC times and frequency increase from ASv2 release 3.21.3 (red) to release 4.2.2 (blue); GC = garbage collection. 38
2.17 The overall availability of the entire online shopping application (shown in percent) has increased in recent years. 39

2.18 New release of shopping application (represented by the red line) shows an increase in response times for Expedia (EXPD) transactions as compared to the current release (represented by the green line) 40
2.19 The scheduler application's residual memory for the new release(represented by the red solid line) shows an increase as compared to the old release (represented by the green solid line) 41
2.20 Traffic Profile for a Conventional Load Test for an Airline Shopping Application.Red solid lines represent the new release and the green solid lines represent the current release 42
2.21 Elapsed Time Measurement for Conventional Load Test for New (represeneted by the color green) vs. Current (represeneted by the color red) Airline Shopping Release 43
2.22 Certification Numbers for Elapsed Time After a Conventional Load Test for an Airline Shopping Application 43
2.23 Production Like Traffic Profile for an Impulse Test for an Airline Shopping Application. Red solid lines represent the new release and the green solid lines represent the current release 44
2.24 Elapsed Time Measurement for Impulse Test for New (represented by the color red) vs. Current (represented by the color green) Airline Shopping Release 45
2.25 Certification Numbers for Elapsed Time After an Impulse Test for an Airline Shopping Application 45
2.26 Example of how to look for string patterns in Application Logs and then set Alerts. 48
3.1 Data Flow for Hotel Booking Application 55
3.2 Test Logs for Hotel Booking Application 55

3.3 Software Testability Enablers 58
3.4 Characteristics of Software Applications that Affect Testability 59
3.5 NOFEP Architecture 64
3.6 Time taken to create connections with existing NOFEP architecture . 66
3.7 CPU usage of a NOFEP server with existing architecture 66
3.8 Load average of a NOFEP server with existing architecture 67
3.9 New NOFEP architecture 68
3.10 Time taken to create connections using the new NOFEP architecture 69
3.11 CPU usage of a NOFEP server running the new architecture 69
3.12 Load average of a NOFEP server with the new architecture 70
3.13 Characteristics External to Software Application that Affect Testability 71
3.14 Example of an Automated Test Harness in a Test Environment with Backend Simulators/Mocks 73
3.15 Environmental Characteristics that Affect Software Testability 75
3.16 Factors to Consider When Setting Up a Test Environment 76
4.1 Design excerpt of a real distributed travel shopping application. The upper right shows key components of our EI monitoring tool, including various instances of the instrumentation library (EI API), the shared Pub/Sub communication medium, and the Consolidated Logging Repository (CLR). 81
4.2 Overview of the main EI components - 1. EI API, 2. Pub/Sub, 3. CLR and 4. Monitoring GUI collecting various monitoring data, i.e., application metrics and audit, security, and billing data. NAS = network-attached storage; pub/sub = publish-subscribe. 84
4.3 Logical View of Summary Customer Metrics Collected Using EI API . 90
4.4 Logical View of Application Using EI API to Collect JVM Data . . . 94

4.5 Logical View of Application Collecting Billing Information 98
4.6 Yearly historical process data on a given server collected in CLR to identify performance trends. The graph shows the total number of processes running on the server on a particular day for each month from September 2010 to September 2011. 101
4.7 End-to-end system response time increases as TPS is increasing in the shopping application of Figure 4.1. 102
4.8 CPU usage on one of the two shopping servers. The corresponding graph of the second server is very similar. CPU usage is increasing with the TPS increase of Figure 4.7. 102
4.9 Fine-grained performance data collected in CLR: Response times, elapsed times, up-times, and failed transactions. 103
4.10 Example fine-grained error statistics collected in CLR. 104
4.11 Logical View of User Actions Done During Testing 104
5.1 Step by Step Visualization of the Validation Activity of the Test Harness118
5.2 General Test Script Design for Environments without Mocks. Additional Components Needed for Environments with Mocks are shown in Figure 5.6 121
5.3 Sample Directory Structure for Easy Test Script Development and Management 122
5.4 Sample Test Scenario Configuration File that Clearly Describes Detailed Test Scenario, Interactive Console Features, Request and Log Directories. Configuration Files Simplify the Test Script Development and Ease Maintainability 123

5.5 Sample Post Test Report for a Script. Automated Generation of Such Reports Helps Identify the Types of Errors and Application Behavior during Testing 125
5.6 Test Script Design for Environments with Mocks. Basic Test Script Design Principles - replaying traffic, reading configuration file and creating reports are based According to Figure 5.2 127
5.7 Actual vs. Expected Test Results 128
5.8 Sequence of Activities for Test Harness and Script 130
6.1 Execution Steps for the Taguchi Method of Design of Experiments . . 143
6.2 Multi-Tier Architecture of Software Applications 146
6.3 Percentage Distribution of Traffic Types for Airline Shopping Application 151
6.4 Data Distribution of Incoming Transactions for Four Airline Shopping Workload Types. The X-axis represents the logarithmic scale of base 10 and Y-axis represents the kernel density function of the data 152
6.5 The Frequency Histogram for Airline Shopping Application TPS . . . 153
6.6 The Taguchi Othogonal Array Selector 155
7.1 Example of a discrete-time Markov chain referring to Equation (7.6) . 161
7.2 State Space Model for Rainy and Sunny Days in a Week 163
7.3 State Space Model for Rainy and Sunny Days in a Week with Transition Probabilities 163
8.1 Different Systems Work Together to Enable Self Service Checkin for Travelers 170
8.2 Performance bottlenecks for an application can be found at each communication link (incoming and outgoing) with other components . . . 171
8.3 The Pipes in Hydraulics are Similar to Software Application's Capacity Measurements 174

8.4 Steps for Detecting Bottlenecks in a System 175
8.5 Example of a Pressure Point Due to Queueing 175
8.6 Example Showing a Critical Bottleneck 176
8.7 Different Paths Through the System can Cause Different Pressure Points 177
8.8 An example of an airline ticketing system 179
8.9 Context Diagram for Interline Electronic Ticketing (IET)/TKTHUB . 180
8.10 Sequence of steps for pay me later option for payment web service application 181
8.11 nodeInCluster Algorithm Decides Whether to Add a Node to an Existing Cluster or Form a New Cluster 187
8.12 createSSM Algorithm to Build a State Space Model For a Software Application 189
8.13 Resulting SSM for the createSSM Algorithm Example 192
8.14 findPressurePoint Algorithm to Identify Performance Issues for Software Applications 195
8.15 Simple Representation of Shopping Application With Backend Components - Availability and Schedule Finder along with the Database Server Pool. 197
8.16 Simulation Setup for Shopping Application and Backend Components Shown in Figure 8.15 in extendsim. 197
8.17 The Different Simulation Factors and Their Levels Can be Setup as Attributes in extendim 198
8.18 Simulation Setup for Airline Shopping Application and findPressurePoint Algorithm to Identify Pressure Point States 199
8.19 Response Time Plot for Simulation Data with Response Times Greater Than the SLA Marked as Pressure Point 200

8.20 CPU Usage Plot for Simulation Data 201
8.21 Memory Usage Plot for Simulation Data 202
8.22 Disk Usage Plot for Simulation Data 202
8.23 Response Time Plot for Test Run on Real World Airline Shopping Application Using Proposed Solution vs. Regular Performance Test. The Combination of Multiple Factors Pushes the Application Over It's Response Time SLAs Before an Increase in TPS in a Regular Performance Test Does.Pressure Points are Detected Using Proposed Solution but Not Using a Regular Test. 206
8.24 CPU and Memory Usage Plot for Test Run on Real World Airline Shopping Application Using Proposed Solution vs. Regular Performance Test. The OS Utilization is Higher for the Proposed Solution Since It Pushes the Application Outside It's Normal Processing Boundaries. . . 207
8.25 Queue Depth Utilization Percentage for Real World Airline Shopping Application Workload Types Using Proposed Solution and Using the Queue Depth Factor Levels from Chapter 6 208
8.26 Errors Including Timeouts and Throttled Transactions for the Shopping Application Using the Proposed Solution (LHS) vs. Errors for a Regular Test (RHS) 209

LIST OF TABLES

Table Page

2.1 Release certification for different applications that are part of an airline shopping back-end application data flow path. 19

2.2 Release certification criteria for an airline shopping application; d = day; CPU/shop = CPU usage for every call made to shopping application. 22

2.3 Correlating production and test for Airline A. 50

4.1 JVM Metrics Collected Using EI API 97

4.2 Increase in response time for method calls when using DynaTrace compared to our EI tool. 106

4.3 Increase in response time when delivering pages, using DynaTrace as compared to our EI tool. 107

6.1 ANOVA for One Factor Design. df represents the Degrees of Freedom, SS the Sum of Squares, MS is the Mean Squares and F is the Calculated F-value 139

6.2 ANOVA for Three Factor Design.df represents the Degrees of Freedom, SS the Sum of Squares and MS is the Mean Squares 141

6.3 Layout of L9 Orthogonal Array 144

6.4 TPS and Workload Type Levels for Performance Evaluation DOE . . 154

6.5 Queue Size and Timeout Levels for Performance Evaluation DOE . . 154

6.6 Throttle and Payload Size Levels for Performance Evaluation DOE . . 154

6.7 DB and Application Thread Pool Levels for Performance Evaluation DOE 154

6.8 L50 Standard Orthogonal Array to Decide Combinations of Levels for Factors 155
8.1 Sample State Space Data for Travel Shopping Application 190
8.2 Transition Probabilities of the State Space Model Generated as a Result of the Simulation 203
8.3 State Transition Probabilities for Example State Table 8.1 212
8.4 State Transition Probabilities for Four Test Cases Run From the Taguchi Orthogonal Array for an Airline Shopping Application 214
8.5 Canonical State Transition Probability Matrix for Table 8.4 215
8.6 Sample Initial State Transition Probability Matrix 218

CHAPTER 1

INTRODUCTION

From simple day to day tasks to the more complicated functionalities from every aspect of our life is slowly becoming computerized. Every industry is dependent on computers to provide services to their customers. A company's or an organization's key strategy to retain its online customers is to maintain a production environment that reliably caters to their needs and works without fault. Adding more functionality to attract customers leads to more complex code and infrastructure. There are usually different kinds of components such as routers, load balancers, message queues, middleware APIs along with different kinds of hardware and operating systems that make up the end-to-end production environment. A smooth operation of such a distributed system requires a clear understanding of not only how to design it but also how to successfully test it before deployment.

The chapters will concentrate on the non-functional or performance testing aspect of software applications. This implies that we will look into how to improve the process and how to run complete tests that save time and resources by combining characteristics of various performance tests in one. We will also look at the process of certification and correlation that helps with deploying releases in production that perform better or comparable to currently deployed releases. We will also look at the various engineering aspects of the solution design that need to be part of the testing process for proper performance validation. Since automation is such an important part of testing, we will also look into how to design an automation framework and characteristics of test scripts that will help with the complete testing process of an

application. Application Instrumentation is another aspect that will be studied. We will provide details of an instrumentation technique that can be used for distributed systems. And lastly, we will look at how a state space model can be used to detect bottlenecks in an application. Most of the performance tests that exist today rely on load as an input parameter to then find issues with the software. Load is just one characteristic on which the application's performance is dependent. We will study an airline shopping application and see how it is affected by different input parameters and how these parameters can be changed to make the application function outside normal boundaries thereby revealing its bottlenecks in one go. Usually multiple tests are run in order to find different issues. The goal of the state space model will be to discover the performance bottlenecks without spending time on setting up, executing and then analyzing data from multiple test runs. The work described in the following chapters except the algorithms and processes involving state space models, was accomplished as part of a team at Sabre Holdings Inc.

1.1 Performance Certification and Correlation

To make our discussion concrete, Figure 1.1 shows a high level view of an actual airline shopping application. The end user accesses the system either through a web based interface such as the Travelocity.com web site or via a terminal. For example, travel agents currently use terminal access to search for flights with the required origin and destination city and dates. The terminal user requests go through an application called NOFEP (NO Front End Processor) and then to a component called Liberty, which in turn forwards the shopping request to the TPF (Transaction Processing Facility). The TPF system is fed from both hosted as well as non-hosted airline data through an intermediary database. Web user requests go through a load balancer, then a shopping application which determines if the request is domestic or

international and puts them in a queue accordingly. These requests are then picked up by ASv2 (Availability Service version 2) and DSSv2 (Dynamic Scheduler version 2) in order to find all available schedules for the particular origin/destination city pair. These applications also talk to a database that stores all hosted as well as non hosted airline data.

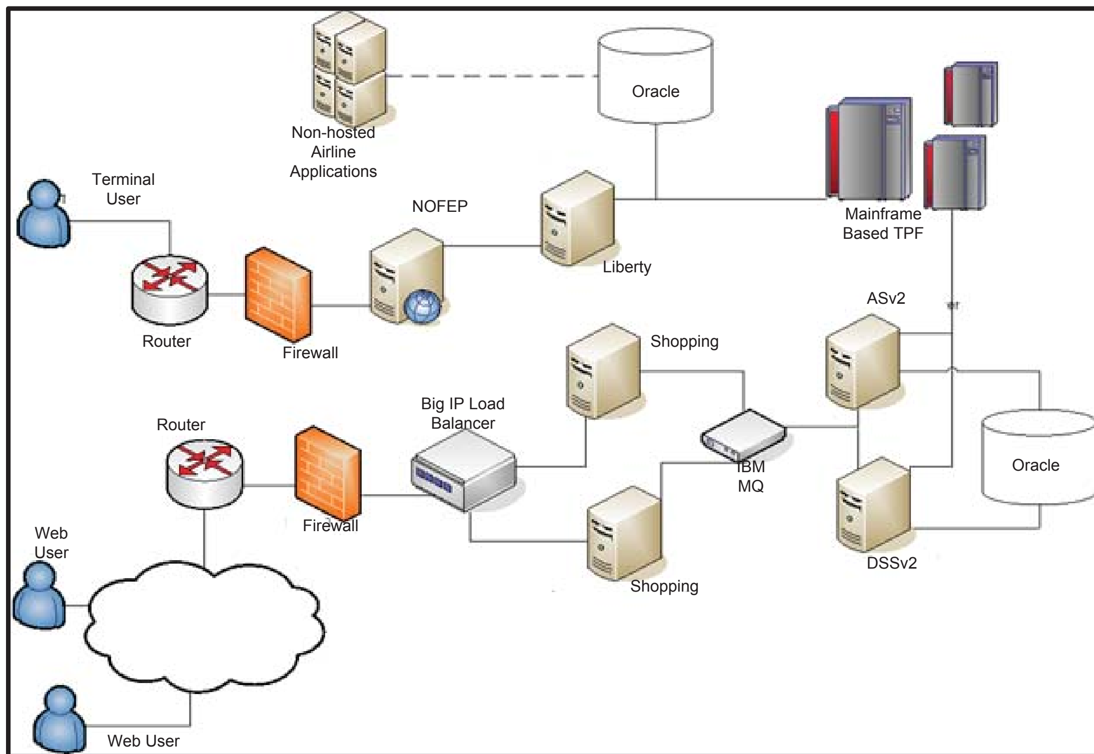


Figure 1.1. Multi-tier online-shopping application from the airline travel industry.

Two major aspects form the basis of running performance tests before the application goes live: (1) assessing its behavior standalone (2) as well as when it is integrated with other components. Standalone performance is usually measured by looking at the response time and throughput metrics and resource utilization data at both the system as well as the process level. Performance testing is a key element

of industrial software development. While basic performance testing concepts are well understood, it is less clear how to manage performance tests in practice. i.e., we have encountered the following two problems. (1) While testing textbooks prescribe writing tests against performance goals, we find that it is impractical to gather from business requirements, performance goals that are detailed enough for finding subtle performance bugs. (2) Once performance tests are conducted, we were asked questions such as the following, which we found hard to answer. How can you be confident that the executed tests assess the performance of the software truthfully? How can you be certain that the software under test performs better than the code that is already in production? How do you make sure that the results collected during testing are a good indication of how the code will function in production?

To enable practitioners to address these problems, we introduce two additional performance testing process components, which we call release certification and test data correlation. Our key idea to address problem (1) is to run two different versions of the same subject application side-by-side in the same test environment. This allows us to use the performance profile of the previous version as the detailed performance specification of the version under test. Our key idea to address the questions of (2) is to correlate the performance measurements of the test and production environments. We also report on a prototype implementation of these process components in our production environment in the travel industry along with the way the performance tests are designed. This work was done as part of a two member team where I helped design the new process along with developing the proof of concept. This work has been published in [4]

1.2 Designing Performace Tests

Figure 1.2 shows a real world traffic profile typically consists of peaks, troughs and some relatively constant transaction levels. In the development lifecycle, due to different reasons, sometimes more emphasis/time is given to the development phase as compared to the testing phase. It is also important to keep in mind that in most cases a particular application when deployed in the production environment will not be working standalone. It will be interacting with other applications, various middleware components, load balancers, the database(if any), customer facing interfaces and maybe even backend mainframe/TPF systems (as is the case with most of the real life examples considered during this work). This implies that both functional and non-functional testing needs to cover multiple scenarios. These scenarios need to include not only transaction paths between different components but also need to cover the engineering aspects of the architecture. Engineering aspects include throttling, timeouts, event based alerting and thresholds. Without having tested these factors, there is no way to validate that an application will perform as expected after deployment. However, conventional performance tests such as load and soak tests are designed to send constant levels of traffic throughout the test duration. Stress or spike tests use short periods of traffic surges. None of these conventional tests individually emulates the incoming request levels in a production environment. The other drawback of running these tests individually is that with every new test case, the environment and application has to be refreshed thereby resetting the state of the application under test, load has to be generated and then performance measurements have to be taken afresh. This reduces the chances of finding bugs related to incapability of the application to handle varying traffic patterns. A load or a soak test might miss performance issues associated with higher levels of traffic or improper settings of timeouts and thresholds. A spike test (which uses short bursts of traffic)

run individually might miss bottlenecks due to memory leaks or garbage collection in case of Java applications, which are usually only uncovered in a long running test.

Figure 1.2. Booking transactions in four related travel shopping applications (air, hotel, car, and cruise) over eleven months..

So to address the problem of evaluating the performance plus the mechanisms built into an application that are supposed to protect it from unusual traffic levels, we propose impulse testing, which is designed to combine traffic that more closely imitates real world application traffic with the characteristics and benefits of load, soak, spike and stress tests. Once again this work was done as part of a two member team where I helped architect the new test design and also implemented it on various application profiles to assess its benefits.

1.3 Monitoring and Instrumentation

Monitoring distributed applications is challenging as it tries to satisfy two main conflicting goals. (1) On the one hand, software engineers want to collect data that is as detailed and comprehensive as possible, as having fine-grained runtime data for the entire system is useful in software analysis and maintenance tasks. (2) On the other hand, collecting any monitoring data is an overhead that can be very expensive, as the monitoring tool consumes computing and communication resources of the very machines and networks it monitors. Any monitoring tool therefore tries to both maximize the utility of the data it collects and minimize the overhead it incurs. In addition to these conflicting two main goals, existing monitoring tools also try to optimize other goals, such as ease of deployment and providing an integrated solution. These additional goals however conflict with the two main goals, as both ease of deployment and a highly integrated tool can lead to collecting data that is less useful while incurring a higher runtime overhead.

Specifically, we found that many existing monitoring tools utilize generic program instrumentation facilities to collect data. This is convenient in program deployment, as it can automatically instrument different kinds of applications. However it also limits the applicability of the approach to the platforms it supports. Practical business applications are often implemented on a wide mix of hardware and software platforms as well as communication protocols, which increases the risk that some part of the system is not supported by such a generic instrumentation approach. In addition, such a general approach also leads to bloat, as it may insert monitoring probes that are not needed for the specific application, which causes unnecessary overhead.

To provide an integrated solution, many existing monitoring tools also utilize their own custom communication techniques to propagate monitoring data from the monitored entities to repositories or analysis nodes. Such an integrated solution may

be useful in some situations, but it can also lead to undesired overhead. Such custom communication techniques often run on the very nodes that are monitored and therefore consume resources that could otherwise be used by the monitored application. Examples of such custom communication techniques include hierarchies of filtering nodes that are co-located with the monitored application. We describe our in house built lightweight instrumentation and monitoring tool EI (Enterprise Instrumentation), which provides detailed monitoring data on a wide range of platforms and communication networks yet imposes very little overhead on the monitored machines. Specifically, EI provides an API that can be called from a variety of platforms, including Java, .Net, C++ and C environments. The API is simple and customized to avoid the overhead of more generic instrumentation approaches.

To propagate data from the monitored application to analysis nodes, EI leverages the well-known advantages of publish/subscribe systems and message queues, which separate processing logic (i.e., monitoring activities) from communication concerns. A key observation is that this separation of concerns allows us to separate the places at which the concerns are handled. This separation can preserve computing resources on the monitored machines for the application and thus yield better end-to-end performance. This work was accomplished working with the EI architecture team where I ran different tests and gathered data to compare the EI API tool with dynatrace.

1.4 Test Script Design and Automation

Scarcity of commercially available testing tools that could support all native or application specific message formats as well as those that cater to non GUI or non web based backend applications leads to creating your own customized traffic generators or scripts. Also the test environment setup may differ from one system to another

some may use simulators or mocks to stub out complex software, others may just be a scaled down (in terms of number of servers) replica of the production environment. So what are the factors that need to be considered when creating scripts that can be used for native request formats and for non GUI or web based applications? How do we design a script that is easy to maintain and extend when new test scenarios are added to accurately assess the performance of an application? We provide (1) the general design principles for a test script that can be used to generate traffic for any request format as well as (2) specific factors to keep in mind when creating a script that will work in a test environment that uses a mock. In addition to this the core activities of testing include not only traffic generation but also setting up the environment, verifying that both the hardware and software configurations are accurate prior to sending traffic and creating a report at the end of the test. Therefore the test script needs to be part of a complete harness that accomplishes these tasks. We will address the (3) design and properties of such a harness. It provides a simple framework that can be easily used to complete an end to end testing process -pre test, traffic generation and post test activities. This automation framework was built as part of a two member team where I architected the solution and helped code part of it for implementation. This work was published in [1].

1.5 End-to-End Performance Evaluation of Applications using State Space Models

Last but not the least we will finally look at how to identify performance pressure points in an application when it is interacting with different backend or upstream systems. As mentioned earlier there are different types of tests that can be performed on an application to verify its functionality and performance. Both functional and non-functional testing is of utmost importance before the application is deployed in production in order to create a stable and reliable operating environment. Usually

when it comes to software testing a majority of the effort is directed towards load testing of client facing applications. Existing tests that are performed in order to evaluate the performance of a software application are dependent on one varying input factor - load. Tests are defined as load, soak, stress, capacity and so on based on how often the incoming transactions are per second are varied and what level of load is driven through the application. With his kind of an approach, a number of tests have to run in sequence to find the various bottlenecks in the application. Varying just load levels may not even uncover all the issues.

In general there are numerous factors that affect how the application will perform after being deployed in production. We will be using state space models in particular Markov models and experimental designs to study the effect of changing input parameters that influence a software application's performance. The purpose of this is to operate the application outside its normal boundaries and reveal all possible issues that could affect it in production. The objective is to evaluate and proactively predict the performance and profile of a software system that is dependent on multiple varying input parameters in a large scale distributed system. The applications considered would not be client interfacing web based applications but ones that interact with multiple components to process incoming transactions. The goal is to find pressure points for the application when it interacts with other systems and/or middleware components and database. Pressure points are any application resource that can become exhausted thereby restricting or degrading service level performance. Examples of pressure points include: CPU, memory, disk, network, code loops, locks, file handles, stack/heap settings, buffers, threads, connections and so on.

We will be taking an airline shopping application as the pilot application to build its SSM and observe how the workings change, when subjected to different changing input parameters. Numerous factors have direct or indirect influence on the

application's performance. There are also different parameters that can be used to measure the changes. The SSM will define the state of the application in terms of it's resource utilization numbers, it's throughput as well as the it's processing times. The goal will be to subject the application to various input parameters and then observe how the state of the application changes. Any state of the application where the processing time goes over the predefined SLA and an application crash will be defined as a state that is a potential pressure point. The combination of the input parameters will be noted and that will define the scenarios that could potentially cause the application performance to degrade when deployed in production. Knowing the cases under which the code is not performing well, will help the development team to proactively write and put checks in place to avoid these scenarios and to instrument and implement alerts that will notify the operations team of any performance degradation immediately.

CHAPTER 2

MANAGING PERFORMANCE TESTING WITH RELEASE CERTIFICATION AND DATA CORRELATION

2.1 Introduction

For a commercial computing environment that processes transactions 24*7, 365 days a year, software and hardware stability and reliability are crucial. Any disruption of service could have catastrophic effects on the company in terms of revenue earned, brand credibility, and customer loyalty. Bottlenecks or other issues that could degrade the performance of a deployed application should thus ideally be found and resolved in the test environment. Performance testing becomes crucial in such situations because you want to find and resolve issues that could in anyway degrade the performance of an application when deployed in production.

A particular piece of code deployed in production, let's call it the existing application release goes through several iterations of improvements and maintenance. Every time the code is modified, let's call it the new application release, it has to be tested before it is deployed in production. Following are the conventional steps used to run performance tests for any software application [2].

1. Understand Business/Application Needs: Performance testing is different from functional testing. Understanding the business drivers and the service level agreements behind a developed application helps in identifying the objectives of performance testing.

2. Identify Acceptance Criteria: Here we identify the performance targets. We have to know if a test passed or failed and hence we need to know what kind of data or metrics need to be collected in order to make that decision.
3. Identify Test Environment: Having a test environment that mirrors production would be ideal. But since that is not practical there are certain factors other than assembling production like hardware that need to be considered when building a test environment. These factors include among others network, operating system, software licenses, and data collection tools.
4. Plan and Design Tests: Here we analyze workloads, create scripts, and build test data. There are several kinds of performance tests apart from load and soak tests and this is the time when the decision on what kind of tests need to be run is made.
5. Execute Tests: The test environment, scripts, data monitoring and collection tools are validated and the performance tests are run.
6. Analyze Results, Report, and Retest: This last step is one of the most important steps for any performance testing project. It is very easy for the stakeholders to misinterpret data and hence it becomes essential to analyze metrics correctly and use the right statistics to report results. Using these results, if the decision is made to tune parameters or change code, it is vital to re-run performance tests.

While an application is being used on production servers, developers often already work on subsequent versions. Before deploying such a new version to production servers, the new version has to be tested, to minimize the risk that it will not cause service disruption. The conventional wisdom on performance testing is to specify performance goals such as response times and test the application against these goals. Following is a typically textbook description of this process: "The objective of perfor-

mance testing is to validate the software 'speed' against the business need for 'speed' as documented in the software requirements. Software 'speed' is generally defined as some combination of response time and workload during peak load times." [2, page 129]

In practice, we have found it hard to impossible to gather a performance specification that is detailed enough for finding subtle performance bugs. Performance goals are typically maintained in service level agreements (SLAs), which are created by business units. SLAs contain some valuable high-level information, such as end-to-end response times. But missing in SLAs are fine-grained performance goals that are expressed in terms of low-level technical performance metrics, because business units that formulate SLAs are not familiar with such technical metrics. Examples of such fine grained metrics include CPU time, garbage collection time, virtual memory cache misses and hits, virtual memory usage, and system exceptions.

So how do we make certain that the new application release will behave similarly or better than the existing release? How do we know that the performance test cases and scenarios are emulating real world production scenarios? Keeping track of SLAs for standard metrics such as response times is a good starting point. For example, recording the 95th percentile response time of 3 seconds when the maximum acceptable response time is 5 seconds is good. But business units do not keep track of more granular metrics and their corresponding SLAs that can impact the performance of an application. For example, if the cache hit rate for a database decreases from 99.9% to 99.8%, it basically implies a 10% increase in the physical I/O requests and could be detrimental to how well the database performs. The miss rate is the number of I/O requests that cannot be satisfied from the cache and therefore result in a physical read.

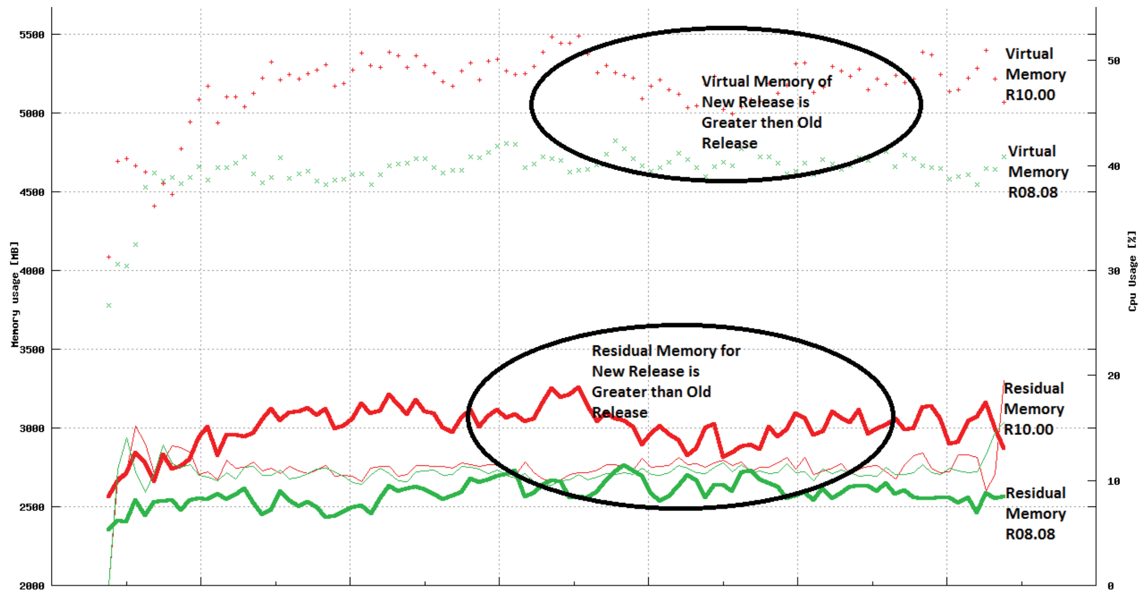


Figure 2.1. Performance bug in an airline ticket sales back-end application hosted by Sabre. CPU usage (thin lines) is similar between release 8.08 and release 10, but virtual memory (dotted) and residual memory (broad lines) differ significantly. This bug was found with fine-grained performance goals for virtual and residual memory. These performance goals did not appear in any SLA but were derived from the performance of release 8.08..

Figure 2.1 shows an example from our experience of testing an airline sales application. The example highlights the importance of having such fine-grained performance goals. The CPU usage behavior for the two releases being compared is similar, but there is a significant increase in the virtual and residual memory of the new release. This behavior can cause the system to crash, which means we would like to catch and fix this problem during performance testing.

Another missing feature from the conventional performance testing steps is data correlation. It is quite common to see applications breakdown when they go live even after rigorous functional and non-functional testing. The primary reason for that is the failure to correlate the results compiled during testing to the actual numbers seen in production. Although fine-grained technical performance goals are

important, we have found that results obtained from measuring performance metrics in the test environment cannot be used directly to make predictions about the eventual performance in the production environment. The reason is that the behaviors of the test and production environments typically diverge at some point. To address this problem, we are experimenting with a technique for test data correlation. This has allowed us to pinpoint problems in the test environment and in the test cases used to emulate production usage of the software under test. This extra validation step ensures that the test environment is setup properly, the test cases being executed are emulating production scenarios, and the workload being used during tests closely mirrors real live traffic.

2.1.1 Solution Overview

We propose to take a traditional performance testing workflow and extend it, by adding two components, release certification and test data correlation. Figure 2.2 shows an example resulting workflow, which we have been using over the last 2.5 years for performance testing of several large-scale applications, including an airline sales application. The certification process compares the performance of the new application release to the old and determines if the measures metrics are better or comparable to the old release. If so, the new release is good to go to production. If not, it does not necessarily mean that it cannot go live. There is a range of acceptable results between good and bad. If the measured metrics fall into that range, the release will still be deployed to production. If the results are not better/comparable or acceptable, that is when the release will not go live. More details on the process are given in the following section.

The process is general enough to be utilized for any end-to-end enterprise system. It is not dependent on any particular hardware or software and can be followed

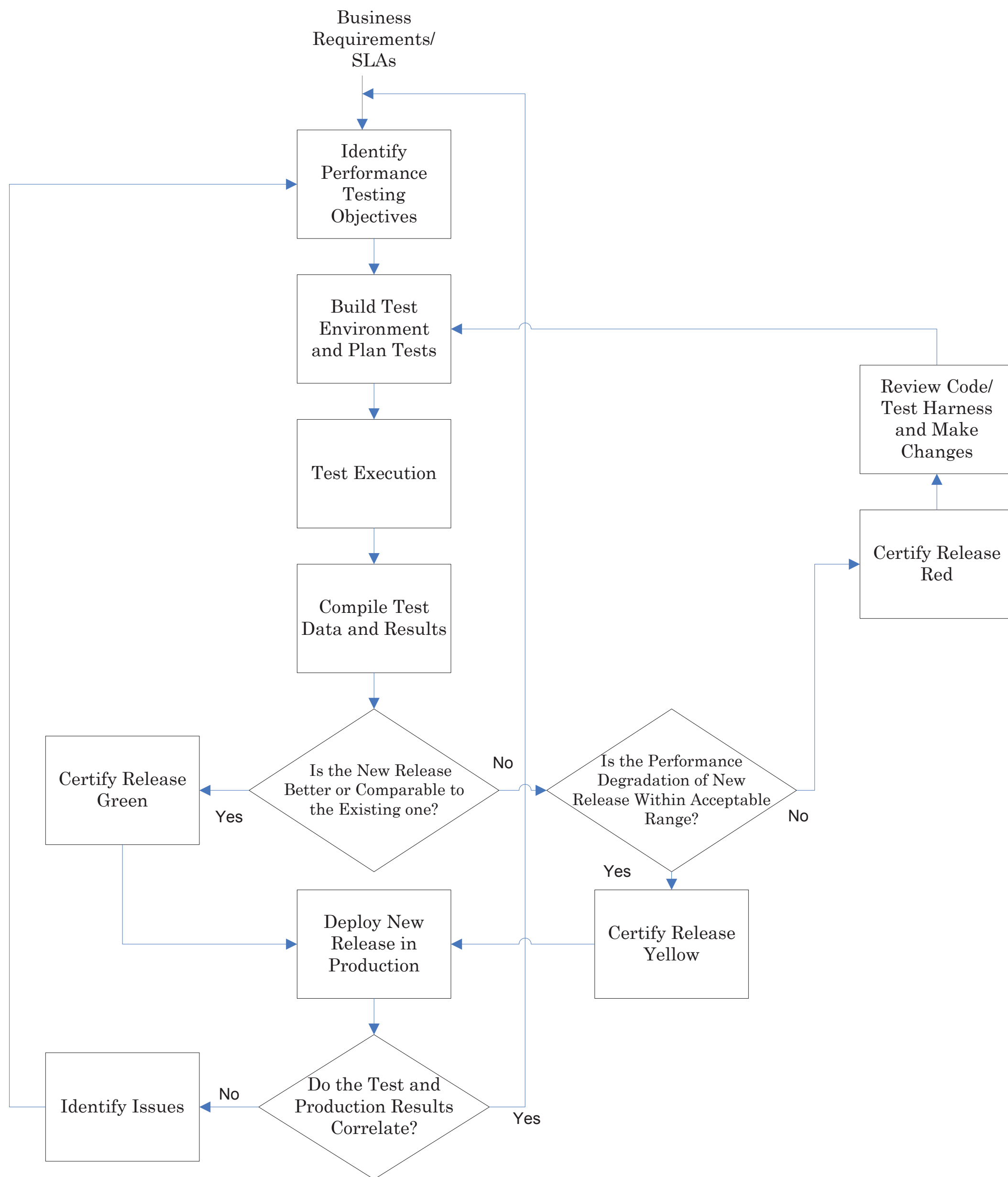


Figure 2.2. Performance testing workflow extended with release certification and data correlation components..

to get reliable performance test numbers before an application is deployed in production.

The two steps added to the traditional performance testing cycle provide the test engineers the ability to verify and validate that their test cases/scenarios and data being used during testing is proper. They are able to gauge if the testing process is providing useful results and the system under test will perform as expected when deployed.

Ideally we would like to have a test environment that matches production in every aspect but this is often not feasible. The test environment is not only a scaled down version but also has components that do not match production capacity like the network bandwidth or hardware types. This not only impedes how thoroughly a system can be tested but also limits the test scenarios that can be executed.

Test certification and correlation together help in setting up a better environment and make adjustments to the way the testing process is being implemented. Certification provides numbers to show how one release differs from the next. It gives test engineers as well as the project managers and system owners data to support their decision of whether to deploy the new application release. Correlation on the other hand provides a certain degree of confidence in how well the performance tests emulate real world production scenarios.

The next sections will describe these two additional steps in detail.

2.2 Release Certification

Release certification is the process of gathering all application and system metrics needed to review the performance of that specific application and setting targets for them. All metrics for two releases of the same application are considered and then the performance targets are classified as red, green or yellow, as seen in Table 2.1. We picked the colors in analogy to traffic lights. Green is a go, with yellow you move cautiously and red means stop. For us these categories encode the range of perfor-

mance targets from unacceptable to acceptable. The numbers for each category are determined together by the developer and performance testers, taking into account the SLAs and the performance measurements of previous releases.

We follow this process even if we add a new feature or traffic profile. In production, the new release will have to process both the existing and new traffic profiles. So when the workload analysis is done, we need to establish the percentage of the new traffic as part of the total and then use that as the input to the new release.

Establishing a baseline is a related approach [3]. As in release certification, a broad set of key performance indicators can be used, such as response time, processor capacity, memory usage, disk capacity, and network bandwidth. The key difference is that in release certification we test the existing and new releases at the same time, each time a performance test is being run. We found that establishing a baseline and later using it as a reference for a new release is not practical, as requirements change constantly. For example, in a web-facing application, the amount and mix of traffic an application is expected to handle can change significantly in a short time frame. Such changes can quickly, within a few months or even weeks, render a once valuable baseline obsolete.

Table 2.1. Release certification for different applications that are part of an airline shopping back-end application data flow path.

Service	**Release**	**Status**
Availability	2012.10.01	Certified GREEN
Schedule Finder	2012.10.01	Certified RED
Shopping HIST	2012.10.01	Certified Yellow
Shopping IS	2012.10.01	Certified GREEN
Shopping MIP	2012.10.01	Certified GREEN

Table 2.1 shows how the different releases for applications that form the part of the airline shopping path were certified after release 2012.10 was performance tested. To implement release certification, we need a test environment in which both the existing and the new application release can be tested side by side on the same traffic which will be described in detail in the next section.

2.2.1 Example: Certification for Airline A

To illustrate the certification process, consider our example of an online shopping website for airline A. Users on the website are able to search for flights between a source and destination city, get schedules and availability data. They are also able to specify how many solutions they want returned for their particular request. The incoming client request XMLs are guided to a pool of web servers and are then load balanced to go a pool of shopping application servers. These shopping servers talk to backend services to get availability and schedules and there is an Oracle database at the backend. Figure 2.3 shows a simplified version of the shopping data flow path in production.

Table 2.2 is the certification criteria table for this sample Airline A shopping application. The first column specifies the metric being measured and the next three columns are the criteria for the new release, expressed not as an absolute value but as a percentage of the measurement of the previous release. In order to properly compare memory utilization and CPU usage, performance tests are run for at least 48 hours and therefore these metrics are compared in terms of usage per day. For example, if we have a SLA for elapsed time (the time taken by a piece of code to execute a particular transaction) set as maximum 4 seconds, anything less than 4 seconds is good. Historical data indicated that even if the piece of code is taking about 5 seconds to process transactions it does not affect the application, so that is a warning but

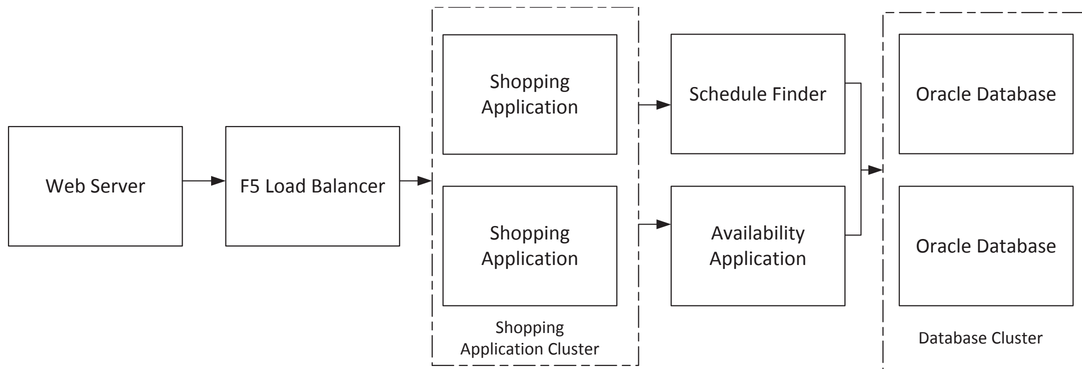


Figure 2.3. Shopping Application Data Flow Path Showing Backend Applications - Availability and Schedule Finder along with the Database Server Pool. The Server Pool of Shopping Application is Made Up of Shopping Historical (HIST), Shopping Domestic (IS) and Shopping International (MIP) Components, Each having Separate Releases..

still an acceptable number. However, anything over 5 seconds is unacceptable and the application will not be deployed in production to prevent performance related problems. The metrics are measured during the entire duration of the test and the differences in the numbers are also calculated for the entire test duration. For an airline shopping application number of solutions is an important metric (as seen in the last row of table 2.2). For example if the end user requests 50 flight combinations for his/her source and destination cities and the application returns only 10, that implies that the quality of the solution is not as expected. This needs to be kept track of from release to release in order to maintain the quality of the solution being returned. Similarly it is important to track the number of failed transactions. An increase in the number of any specific error from release to release implies a bug and needs to be fixed before the application can be deployed to production.

Setting up a test environment with two identical data flow paths for the two releases enables us to test the existing and new releases side by side. In the test

Table 2.2. Release certification criteria for an airline shopping application; d = day; CPU/shop = CPU usage for every call made to shopping application.

Metric	**Green**	**Yellow**	**Red**
Client response time[s]	< 10	10-15	> 15
Increase in CPU per shop	$< 5\%$	5-10%	$> 10\%$
CPU/shop absolute level[ms]	< 30	30-35	> 35
Increase in elapsed time[s]	$< 5\%$	5-10%	$> 10\%$
Elapsed time absolute level	< 4	4-5	> 5
Increase in CPU time	$< 5\%$	5-10%	$> 10\%$
Server CPU utilization	$< 60\%$	60-70%	$> 70\%$
Memory utilization growth	$< 20\%$/d	20-30%/d	>30%/d
Release specific core dumps	0	0	> 0
Other core dumps	0	0	> 0
Increase in #failures	$< 5\%$	5-10%	$> 10\%$
Solutions	$< 5\%$	5-10%	$> 10\%$

environment a similar scaled down (in terms of the number of servers) environment needs to be setup as shown in Figure 2.4. Note that two test paths, absolutely identical to the production environment are build. The difference being that one path will be configured with the existing production release and the other path will have the new release going into production. This will enable us to run tests through the two releases at the same time. This helps avoid chances of environment changes affecting results and also permits comparison of the existing and new releases with the same traffic profile.

Running tests in an environment as shown in Figure 2.4 will help us in comparing system and application metrics side by side and also help in determining if the new application release falls under the red, green or yellow certification category.

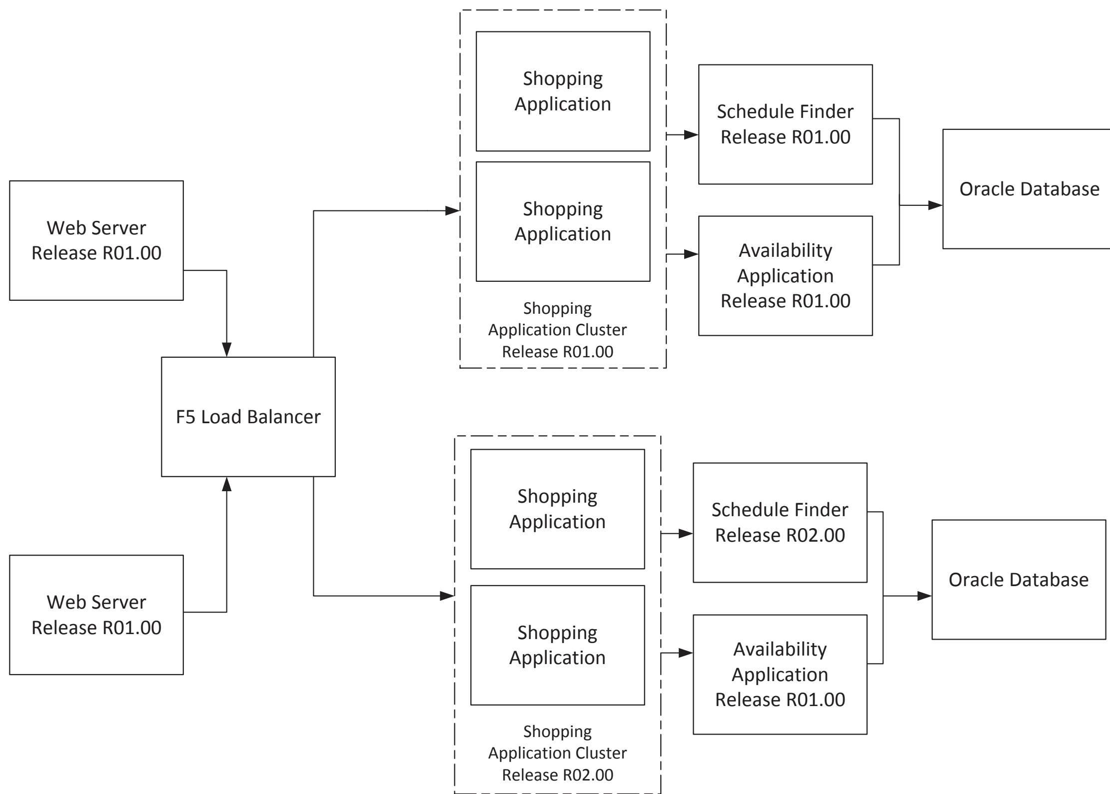


Figure 2.4. Shopping Application Setup in Test Environment with Two Identical Test Paths - One for Current Release and Second for New Release..

2.3 Performance Testing For Release Certification

Testing applications standalone and running only one kind of test or a series in sequence, does not account for situations where the application might have to handle scenarios like sudden surges or drops in traffic over or below the peak levels of workload. Incoming transaction levels fluctuate widely in a real world production environment. Tests need to be designed in a way that not only assess the application's performance but also ascertain that it can process these traffic swings and that it will trigger alerts when a particular threshold for processes or resource utilization have exceeded. Tests need to determine that the application will recover and return to

normal behavior when the traffic levels drop after a surge. When the application is integrated with other components in a system it is expected to be available 24X7 and any failure to do that will lead to system instability. So we propose a test that will be able to combine most of these scenarios into one 48 hour test. We call it an impulse test and it is designed as shown in Figure 2.6. Impulse test is a combination of a load, soak and stress test and can have multiple peaks and troughs and helps verify that the application is architected well to cope with the real world traffic and will perform as expected.

As shown in Figure 2.5, a real world traffic profile typically consists of peaks, troughs and some relatively constant transaction levels. However, conventional performance tests such as load and soak tests are designed to send constant levels of traffic throughout the test duration. Stress or spike tests use short periods of traffic surges. None of these conventional tests individually emulates the incoming request levels in a production environment. The other drawback of running these tests individually is that with every new test case, the environment and application has to be refreshed thereby resetting the state of the application under test, load has to be generated and then performance measurements have to be taken afresh. This reduces the chances of finding bugs related to incapability of the application to handle varying traffic patterns. A load or a soak test might miss performance issues associated with higher levels of traffic or improper settings of timeouts and thresholds. A spike test (which uses short bursts of traffic) run individually might miss bottlenecks due to memory leaks or garbage collection in case of Java applications, which are usually only uncovered in a long running test.

So to address the problem of evaluating the performance plus the mechanisms built into an application that are supposed to protect it from unusual traffic levels, we propose impulse testing, which is designed to combine traffic that more closely

Figure 2.5. Booking transactions in four related travel shopping applications (air, hotel, car, and cruise) over eleven months..

imitates real world application traffic with the characteristics and benefits of load, soak, spike and stress tests.

2.4 Designing an Impulse Test

In order to evaluate the performance of an application, it is not only necessary to assess the inherent functionalities but also its design. As is clear from the examples in the last section, the application maybe tested standalone successfully standalone using conventional tests like load or soak but it still could cause instability of the entire system when it is integrated with other components. So unless and until its behavior is not checked under real production like traffic level fluctuations, its evaluation is incomplete. Tests need to be custom designed to emulate production workload levels.

Keeping in mind the variations in real scenarios as shown in Figure 2.5, we design an impulse test. An impulse test is a combination of a load, soak and stress test and can have multiple peaks and troughs and helps verify that the application is architected well to cope with the real world traffic and will perform as expected. As mentioned earlier it can be designed in different ways but the main characteristics of the test are as shown in Figure 2.6.

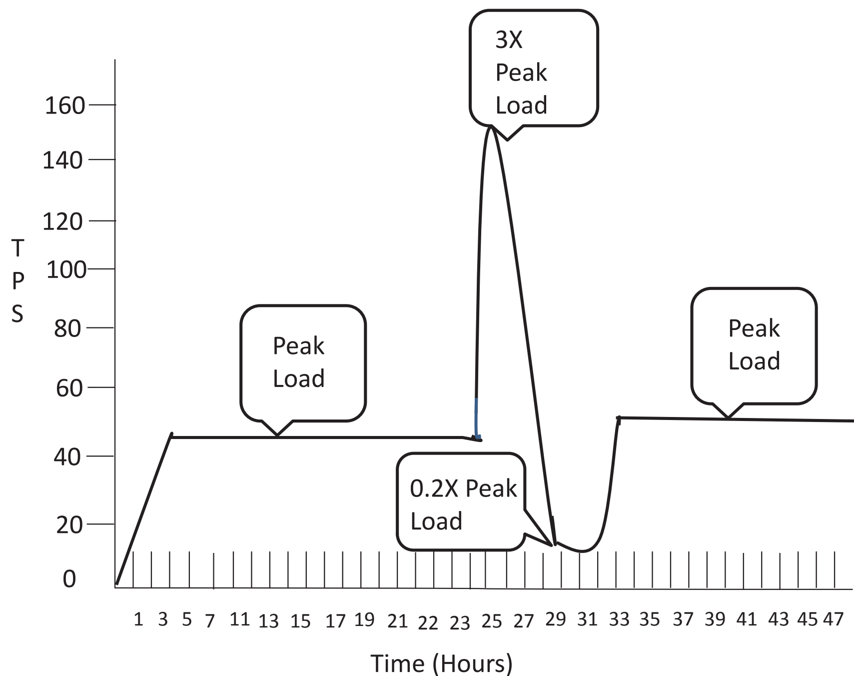


Figure 2.6. Impulse test design. A long period of peak load is followed by a relatively short but drastic traffic surge, followed by a brief drop and a subsequent long pre-surge peak load; TPS = transactions per second..

The impulse test is run for at least 48 hours. This enables not only sufficient time to generate the different scenarios but also will enable the detection of many memory leaks in the applications. The test will also emulate the real world scenario of varying traffic levels. A system will never be subjected to constant levels of traffic

for the entire duration that it is live. There will always be peaks and dips and until and unless these cases are not simulated during performance tests, it will be difficult to assess how the application will perform in such situations when it is deployed. The main design aspects of this test are:

2.4.1 Initial Peak Load

Determine the peak workload level that the application is expected to handle and then run traffic through it at these maximum levels for at least 24 hours. This will help us establish the performance and record various metrics for this duration.

2.4.2 Traffic Surge

Then the next relatively short period of time such as two hours is when the inbound traffic is increased to a level that is significantly higher than the expected maximum load, say three times the peak levels. This is to simulate times when there might be a sudden unexpected surge in incoming traffic. For example if a system handles airline traffic, during a particular airline fare sale there will be an increase in the volume of workload coming in because of customer trying to take advantage of lower air fares and all logging in during the same sale window. The application should behave gracefully even if the load surge is significantly steeper or higher than expected. To prepare for such cases, we need to test if the application will throttle properly, if it will send out alerts when the processing levels or even resource utilization metrics go over certain values, if it has timeout values set appropriately so that the end user is not left waiting indefinitely for a response and the constrained resources are not spent on requests that cannot be fulfilled. This will also check if the application is logging these kind of critical conditions effectively and correctly.

2.4.3 Traffic Drop

After the traffic surge, the next step is to decrease the traffic levels and in this case we are reducing it to at least 0.2 times the peak levels. This decrease in traffic levels is variable and can be changed to better suit the production traffic troughs. This will help in determining if the application recovers properly after the traffic surge, if it is able to process the transactions successfully without throttling or timing them.

2.4.4 Subsequent Peak Load

The last step is to send in peak volumes of workload for the rest of the test duration. This will establish that after the surges and dips the application is still able to consume transactions at peak levels and respond successfully. The application performance should now resemble the peak-level performance that the application exhibited before the surge.

The time durations for the peaks and troughs as well as the frequency at which they occur can be changed to imitate more closely the traffic fluctuations of a particular application as shown in Figures 2.7 and 2.8. Impulse tests can basically be designed according to the needs of each individual application and match the traffic profiles for it as closely as possible. The test can be used for both synthetically generated traffic or even when playing back production requests. Even when requests are captured from production there is a need to play them back in a pattern that matches real live traffic.

2.5 Benefits of Impulse Test

The purpose of running performance tests is twofold, (1) assess the performance of an application standalone and (2) assess its performance when integrated with other

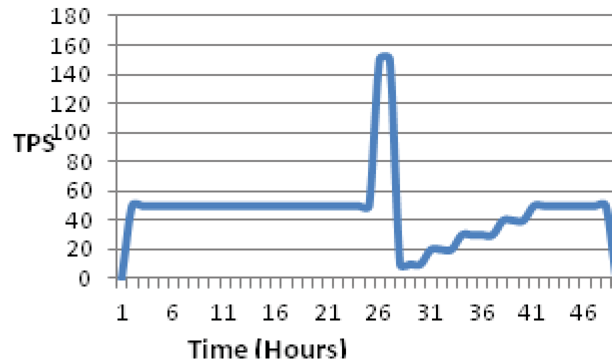


Figure 2.7. Impulse test design. Impulse test with incremental traffic increase after the surge to the pre-surge peak level..

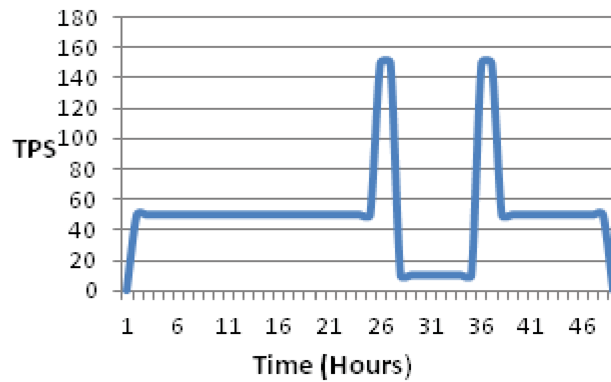


Figure 2.8. Impulse test design. Impulse test with two traffic surges..

components. Impulse tests are designed to achieve both these objectives by running just one test.

To keep up with end user demands many commercial systems are becoming multi-tiered and distributed. In order to retain the customer base and revenue, organizations have to ensure that their production environments are stable and over 99.999% available. It is therefore critical to ensure that each individual application integrates smoothly with other system components. With many applications being developed by different teams in parallel, guaranteeing that they will behave as expected when they go live is increasingly difficult.

Ensuring a smooth integration requires knowledge of the entire system when designing applications. Sometimes developers are either not aware of the entire system architecture or they do not have the proper knowledge of the kind of mechanisms that need to be implemented such as setting throttle limits or thresholds, timeouts or alerting levels, in order to ensure that an individual application will function even when its peer components are unavailable. Running individual tests to validate the working of each of these aspects is time consuming but cannot be an after thought. This is where an impulse test can prove most beneficial. Following are some of the real world examples that were experienced, which reinforce the importance of testing these features during the performance testing phase.

2.5.1 Ignoring the Implementation of Throttles and Timeouts

Figure 2.9 shows a very high level view of part of the customer insight application. The following briefly describes the main functions of the application.

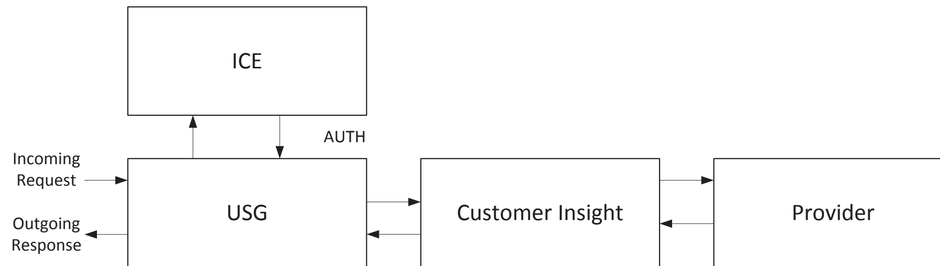


Figure 2.9. The Customer Insight application communicates with ICE for session authentication, USG for user communication, and Provider for user specific data..

The Universal Service Gateway (USG) is the application that creates sessions for users with incoming shopping requests. It passes the requests to a component called ICE that authorizes and authenticates users, creating valid sessions for billing.

Once the session has been created USG passes the requests to the customer insight application to get information about promotion codes or discount codes associated with the user account, in order to evaluate the total billing cost. The customer insight application also talks to a provider component that manages user specific data.

Before a new airline migration, the customer insight application was tested individually with a simulator imitating the provider behavior and all response times recorded were in the acceptable range. When the new airline migrated its traffic to this system, after two hours of service, the customer insight application crashed, causing users to experience unavailable system errors. Even though the application was tested for performance standalone, it was not tested for a scenario where the backend application, which in this case is the provider, becomes unresponsive. The provider was basically overwhelmed with the additional traffic and stopped responding within acceptable time ranges. The customer insight application was depleted of all its user threads waiting for a response because its timeout settings were not implemented properly. The core application's thread pool size was equal to the size of the thread pool that was used to access the provider application and when each of these threads was occupied waiting for responses, the core thread pool was depleted. With incoming requests piling up, the system crashed. This is an example where the application was tested standalone for performance like response times and constant throughputs, but was not tested to see if its throttle limits and timeouts setting were designed properly

2.5.2 Insufficient Alerts

The traffic profile for impulse tests match the profile of the incoming requests in production, and therefore it is easy to validate that the right alerts have been setup for the application. The following two examples show how a live application can go into a severity if alerts, are not setup properly. The following two examples describe

what happens when alerts for resource utilization as well as application logs are not setup respectively. Impulse tests can help with validating both kinds of alerts.

1. Example 1: It is common practice to set up alerts for resource usage utilizations like CPU and memory, but disk usage is often ignored. Setting alerts for disk usage going over a particular threshold will avoid cases where the server crashes due to logs filling up the disk. Even when logging levels are set to INFO, which implies minimal logging during normal processing, it does write extensive logs when exceptions occur. A logging system coupled with the application implies there are no provisions to automate log rotations and archives. So when exceptions occur the logs grow quickly and eventually fill up the entire disk. Setting no alerts for disk usage going over a particular threshold may lead to no indication that the server was about to crash during peak load levels.

2. Example 2: The other example is on not setting alerts for application logs. Figure 2.10 shows a simplified view of the architecture of an application called Sabre Sonic Web (SSW), which depending on the type of incoming user/airline, gets the user profile specifications from a database and displays a customized air shopping web page.

   During a database refresh activity a particular customer was deleted from the database. This caused authentication errors for that particular user every time the user tried to access the airline shopping webpage. This incident was logged in the application logs, but no alerts were set for this, and hence there was no way for the operations team to know that a particular customer was receiving authentication failure errors. It was only after customer complaints and deep investigation of application logs and errors that the root cause was discovered.

   These examples show how important it is to design self-protecting mechanisms like thresholds, throttles and timeouts, in addition to setting the right alerts at

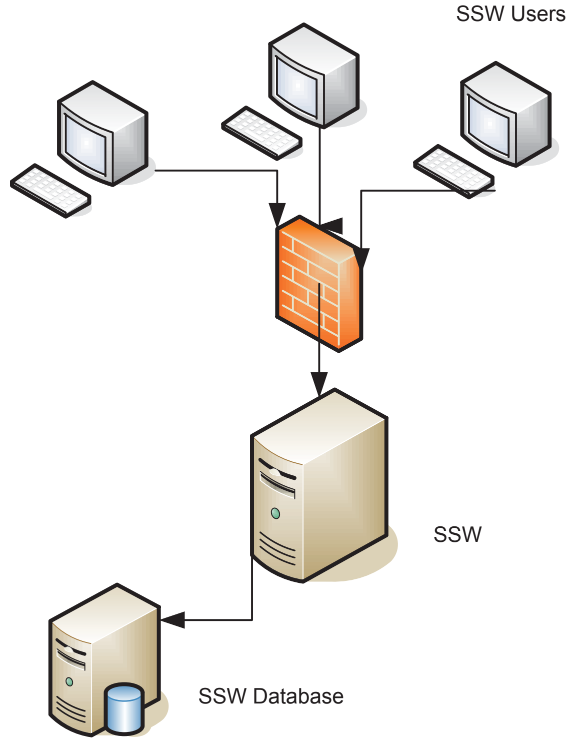


Figure 2.10. Simplified architecture of Sabre Sonic Web (SSW). Users communicate with SSW via a firewall and SSW gets user specific information from the SSW database..

the right places. Learning from these experiences in our production environment led to the design and implementation of impulse tests, which can not only assess the functionality of these mechanisms but also the individual performance of the application. These tests have proved valuable in uncovering situations described in the above examples and fixing them before the application goes live

2.6 Performance Measurements Using Impulse Tests

As mentioned earlier the purpose of running performance tests is twofold, (1) assess the performance of an application standalone and (2) assess its performance when

integrated with other components. Impulse tests are designed to achieve both these objectives by running just one test.

Taking ASv2, the availability application shown in Figure 1, and following the design of the impulse test, constant peak traffic is sent during the beginning of the test, followed by a surge in incoming requests and then the load level is dropped to 0.2X and then back to the peak level. The expected behavior of the application is to process the peak level loads and then throttle incoming requests when the load is increased three times. During this time the throttled requests would timeout. When the TPS (transaction per second) rate is dropped and brought back to processing levels, the application is expected to resume normal behavior. In this example we are comparing the performance of the latest release of ASv2, release 4.2.2 with the previous release 3.21.3.

Figures 2.11 and 2.12 show how the application is throttling requests and throwing proper alerts during the traffic surge duration and when exceptions occur or a particular threshold level is reached, respectively.

Instance	RST	Last Sample								Transactions						DB Name	Build	Last Call
		#Txn	Avg Elapsed	Avg CPU	Errors	Service Errors	Throttled	Con-current	#SQL reads	Total	Failed	Throttled	Avg Elapsed (secs)	Avg CPU (secs)	#SQL reads			
pifhlc100	32	587	0.081	0.042	0	0	0	0	261	3,787	13	0	0.067	0.021	6094	FUNCC2_MIPA	atsev2.2011.10.01	00:24
pimlc302	26	618	0.068	0.023	0	0	0	0	8	119,026	1,187	82	0.170	0.143	74510	FUNCC2_MIPA	atsev2.2011.10.01	00:42
pimlc303	25	658	0.060	0.019	0	0	0	0	7	224,940	1,597	72	0.119	0.090	134231	FUNCC1_MIPA	atsev2.2011.10.01	00:43
pimlc304	29	660	0.065	0.020	0	0	0	1	7	220,643	1,604	61	0.161	0.106	892164	FUNCC2_MIPA	atsev2.2011.10.01	00:42
	112	2,523 42.050/sec	0.068	0.026	0	0	0	1	0	568,396	4,401	215	0.146	0.107	0			

Figure 2.11. The application throttles requests during the traffic surge as expected..

Running an impulse test we observe that the application is working as expected during the initial peak phase and the traffic surge phase. But the test is also designed to evaluate the performance of the application standalone. For the new release of ASv2, we observe an increase in response times and a decrease in transaction process-

Error statistics

Error Code	Count
5053 - NO_COMBINABLE_FARES_FOR_CLASS	44131
5167 - UNABLE_TO_PRICE_ISSUE_SEPARATE_TICKETS	4
9501 - LMT_ISSUE_SEP_TKTS_EXCEED_NUM_DEPT_ARR	21
9502 - LMT_ISSUE_SEP_TKTS_INTL_SURFACE_RESTR	21
6002 - INVALID_INPUT	5
5011 - NO_FARE_FOR_CLASS_USED	45709
9997 - REQUEST_TIMEOUT	463
5005 - PRICING_REST_BY_GOV	144
5033 - NO_RULES_FOR_PSGR_TYPE_OR_CLASS	138
6050 - NO_FLIGHTS_FOUND	38
6119 - NO_PRIVATE_FARES_VALID_FOR_PASSENGER	10993
6120 - NO_PUBLIC_FARES_VALID_FOR_PASSENGER	5895
9006 - null	47
6124 - NO_CORPORATE_NEG_FARES_EXISTS	556
SUMMARY	
	108165

Figure 2.12. The application produces and logs alerts for exceeded thresholds and exceptions during the traffic surge as expected..

ing that indicates a performance issue with the application as shown in Figures 2.13 and 2.14.

This kind of behavior where the application stops processing incoming transactions after wide fluctuations in traffic levels can be easily discovered by an impulse test because the application under test is being subjected to similar traffic patterns as in production without resetting its state. Individual conventional tests like load, soak or stress tests do not emulate real world production traffic variations. With every new test, setting up a fresh environment, generating new load, restarting and monitoring the application and then measuring performance resets the application

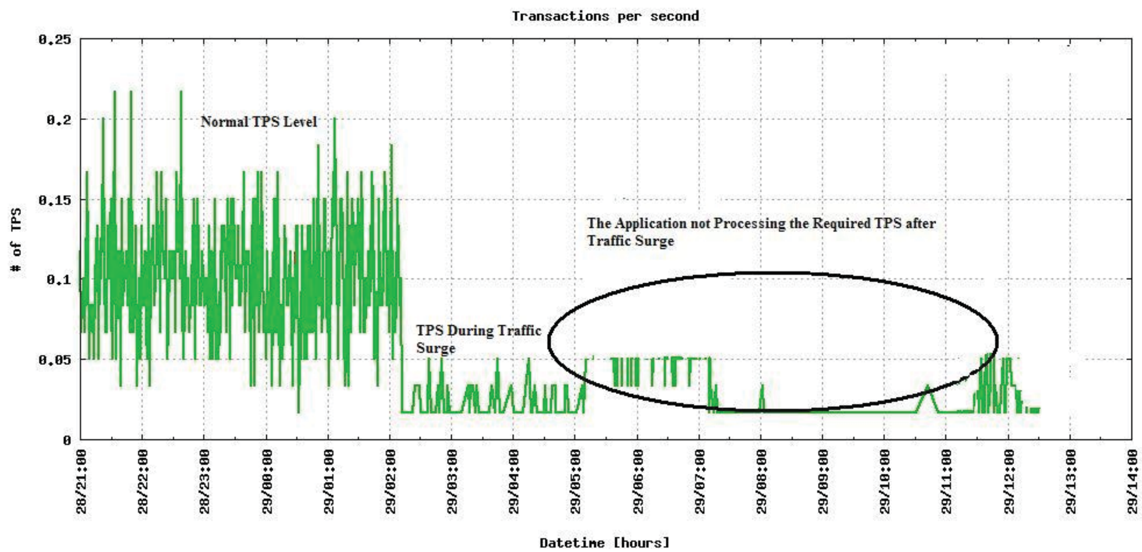


Figure 2.13. Application TPS decreases during the traffic surge as expected, but does not recover after the traffic surge to before-surge levels, which indicates a performance bug; TPS = transactions per second..

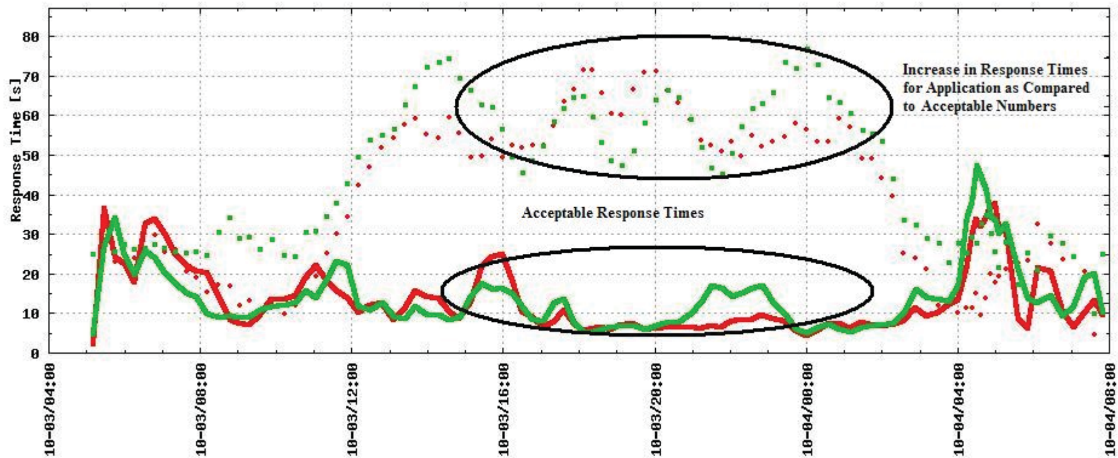


Figure 2.14. Response times for Release 4.2.2(dotted lines) vs. Response times for Release 3.21.3 (solid lines)..

state with every new test case thereby reducing the chances of finding traffic pattern induced performance bugs.

For our example once we see that response times have increased, we can actually look at some of the finer grained metrics like the JVM heap (Figure 2.15) and garbage collection times (Figure 2.16) to investigate the cause of the performance bottleneck. Looking at the heap usage behavior of the new release, it is evident that the application is using more heap memory to process transactions than in the previous release. With no added functionality to the new release, this is a concern. Also the new release has more frequent and bigger garbage collection times vs. the old release. So the application is spending more time in processing normal traffic loads and after the traffic surge is continuously running out of memory and is being stopped more often due to garbage collection pauses and therefore is unable to process incoming transactions. This ultimately leads to higher response times and lower TPS.

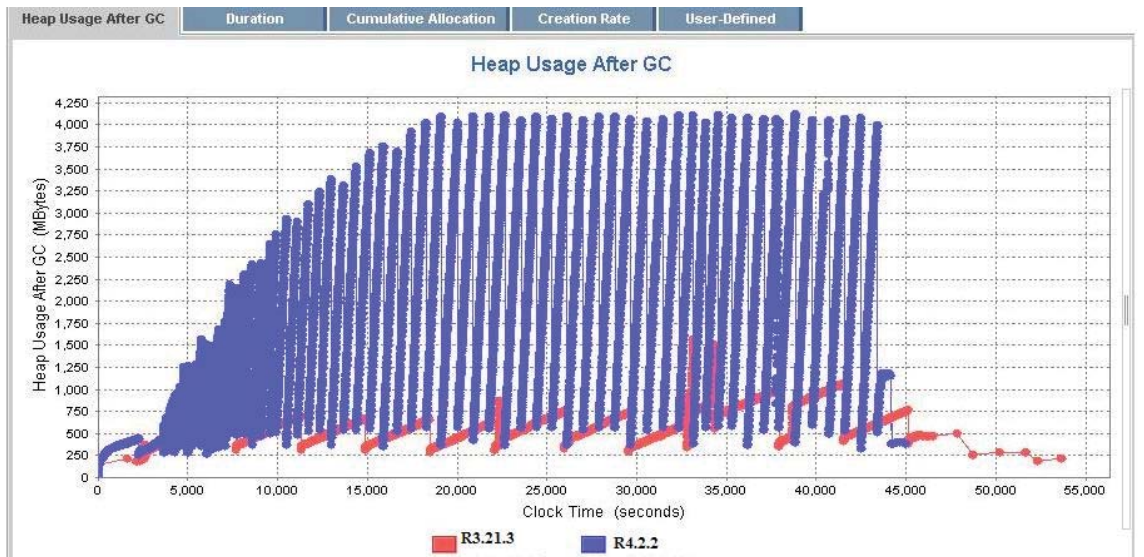


Figure 2.15. Heap usage increased from ASv2 release 3.21.3 (red) to release 4.2.2 (blue)..

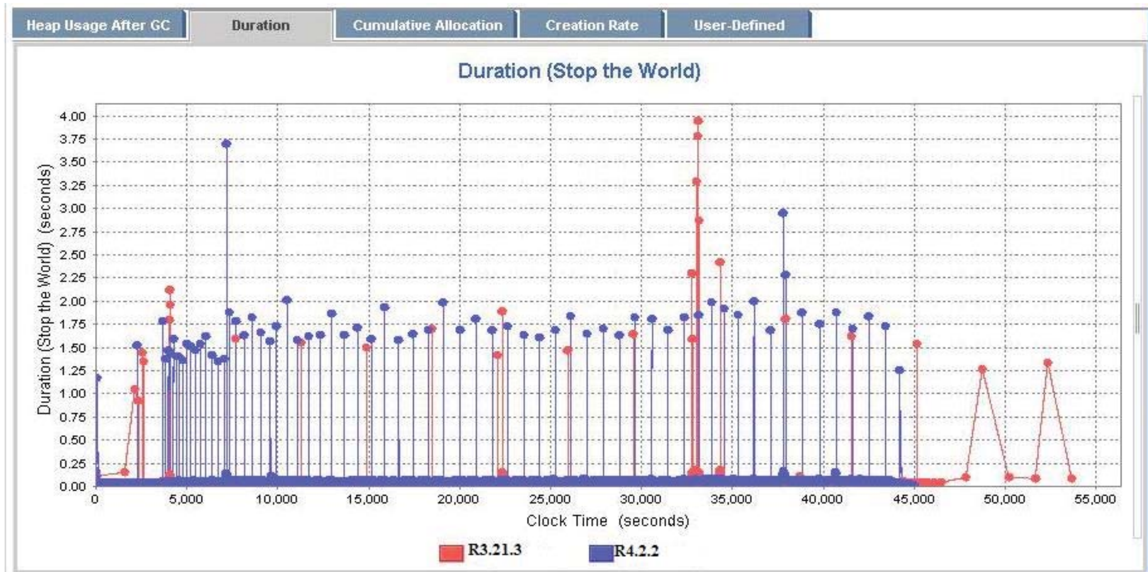


Figure 2.16. Both GC times and frequency increase from ASv2 release 3.21.3 (red) to release 4.2.2 (blue); GC = garbage collection..

Conventional tests where traffic levels do not emulate the fluctuating behavior of a real world production environment are not sufficient for identifying memory leaks or insufficient alerting or threshold/throttle mechanisms. These issues that invariably affect the system availability and stability are more visible when a more realistic production scenario is simulated. The impulse test can be designed to check that the application, once live, will not only perform well standalone but also integrate seamlessly with the other components. If issues are observed during the traffic surge or trough durations, it indicates that the application is unstable and these problems need to be resolved before it can go live. Running impulse tests has led to discovering not only the performance issues but also design issues with thresholds, timeouts and alerting in our test environment, which in turn has led to an increase in the availability of our systems.

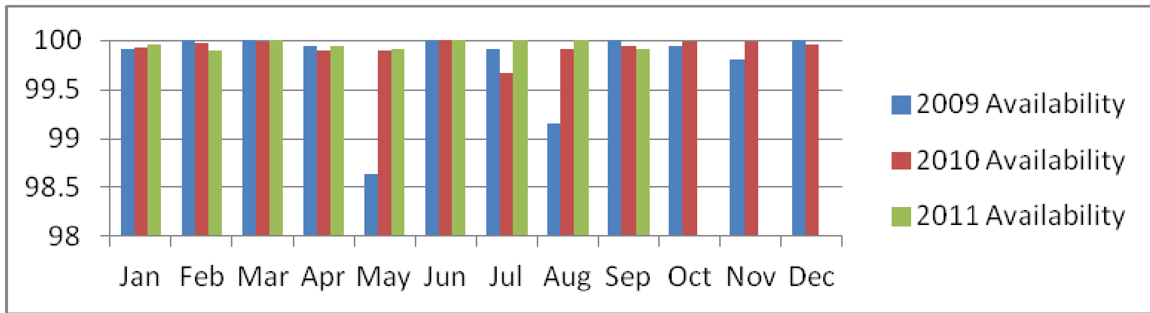


Figure 2.17. The overall availability of the entire online shopping application (shown in percent) has increased in recent years..

Figure 2.17 shows the availability percentage numbers for the airline services system of which ASv2 and most of the airline shopping applications are part of, for the last three years. Designing impulse tests to match production traffic patterns as described in this chapter has played a big role in improving the availability and stability of the environment.

2.7 Impulse Test vs. Conventional Tests

As mentioned earlier, the impulse test is a combination of various conventional tests. It saves time since it needs to be run once but it can still discover performance as well as engineering design bottlenecks that would otherwise have to be uncovered by running individual load, soak or stress tests. Testing is an important phase of the software development lifecycle right before the application/s go live. It is critical to discover all performance and system issues before production deployment but it is also important to accomplish these tasks within a short timeframe. Impulse tests realize both these objectives.

As seen in Figure 2.3 the shopping application cluster is made up of applications that take care of domestic, international and historical travel requests. Using impulse

tests and setting up the environment as shown in Figure 2.4, we can uncover issues for various applications in an end-to-end setup. This not only helps with performance evaluation but also validation of throttle, timeouts and alerts between the interacting applications. From the impulse test results Figure 2.18 shows that the client side response time for a particular incoming shopping transaction (Expedia) has increased for the new release.

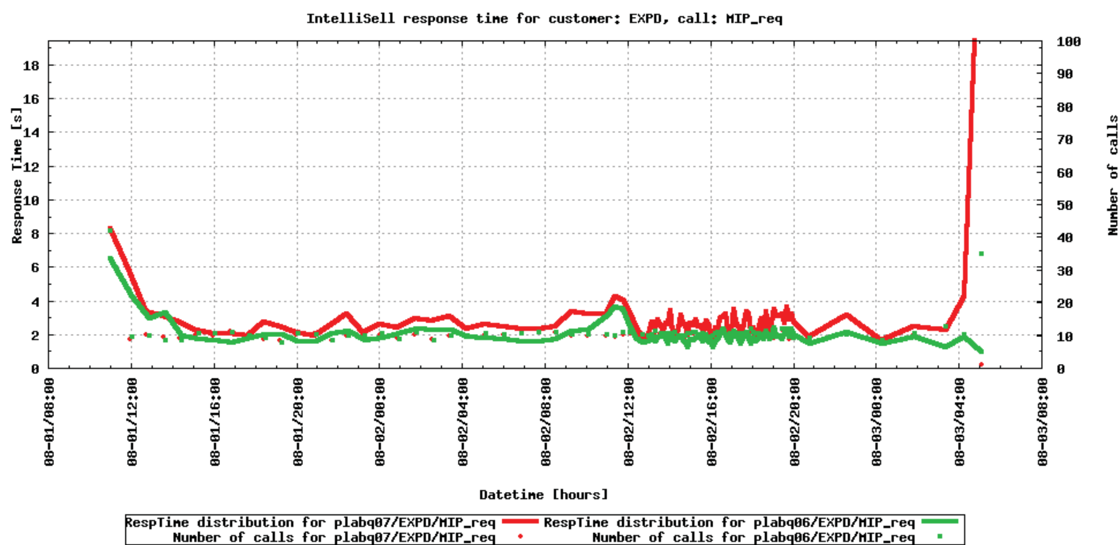


Figure 2.18. New release of shopping application (represented by the red line) shows an increase in response times for Expedia (EXPD) transactions as compared to the current release (represented by the green line).

Figure 2.19 shows that the new release of the scheduler application from Figure 2.3 has an increase in it's residual memory.

One of the advantages of running an impulse test that emulates production traffic profiles is that, it is able to uncover issues that might not be easily identifiable by running conventional tests. As an example take the case of a conventional 48 hour load test. Here the incoming traffic is incrementally increased and then is kept a

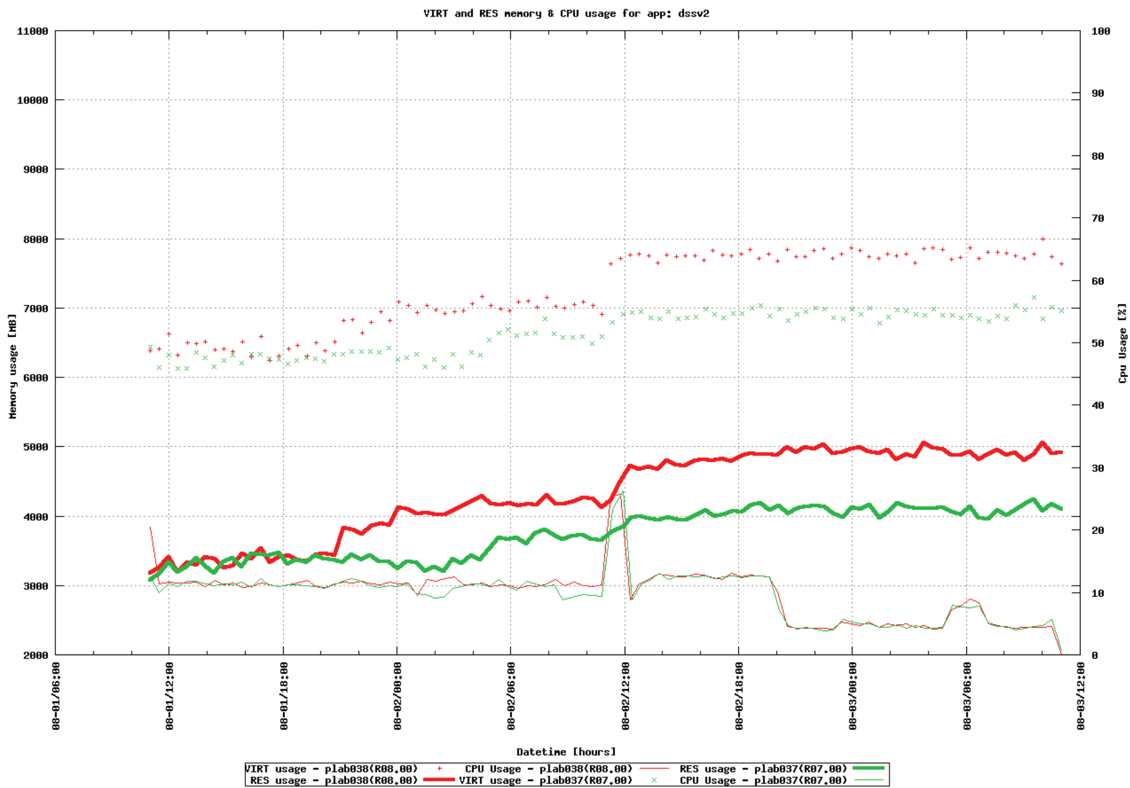


Figure 2.19. The scheduler application's residual memory for the new release(represented by the red solid line) shows an increase as compared to the old release (represented by the green solid line).

constant level for the entire duration of the test. The traffic profile for this load test is as shown in Figure 2.20.

The elapsed time measurement is taken during the test duration for both the current and the new release. The graphical representation of the elapsed time is shown in Figure 2.21.The difference between the measurements for the two releases is not visible from the graph and hence the certification numbers are shown in Figure 2.22.

As can be seen from Figure 2.22, the elapsed times for the new releases fall under the acceptable criteria and therefore are green. When the same releases are

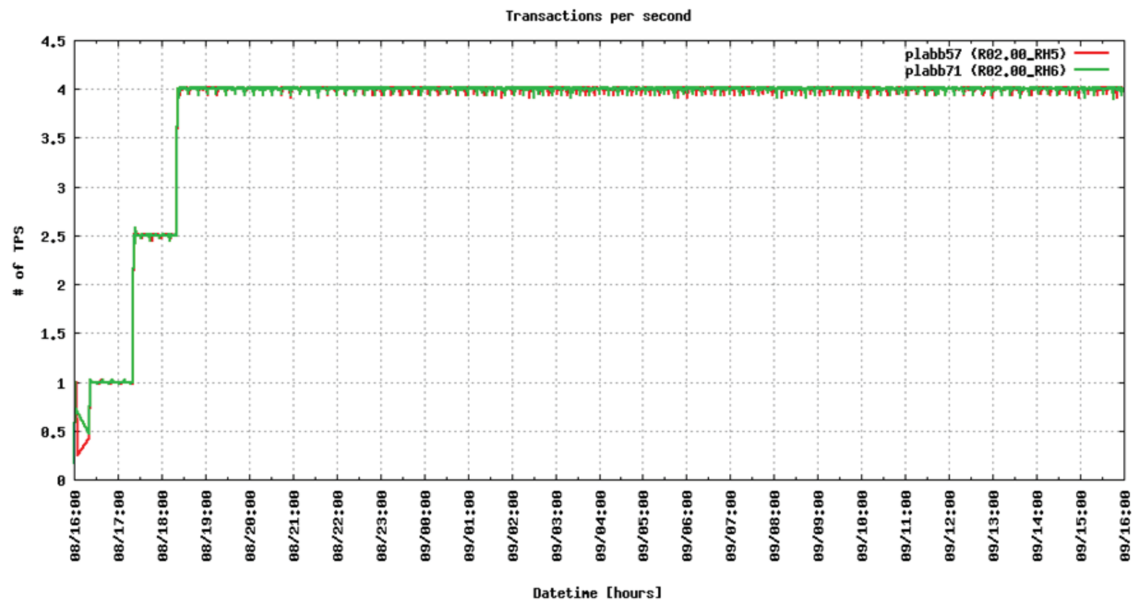


Figure 2.20. Traffic Profile for a Conventional Load Test for an Airline Shopping Application.Red solid lines represent the new release and the green solid lines represent the current release.

tested using an impulse test, the results differ. The impulse test uses a production like traffic profile as shown in Figure 2.23

Once again the elapsed time measurement is taken during the test duration for both the current and the new release. The graphical representation of the elapsed time is shown in Figure 2.24 and the certification numbers are shown in Figure 2.25.

As can be seen from Figure 2.25 there is a major difference in the elapsed times for the new vs. the current release. This difference would have never been discovered if we were not using a production like traffic profile. So instead of running individual conventional tests that take longer to discover performance issues, the impulse test can uncover performance bottlenecks in one run.

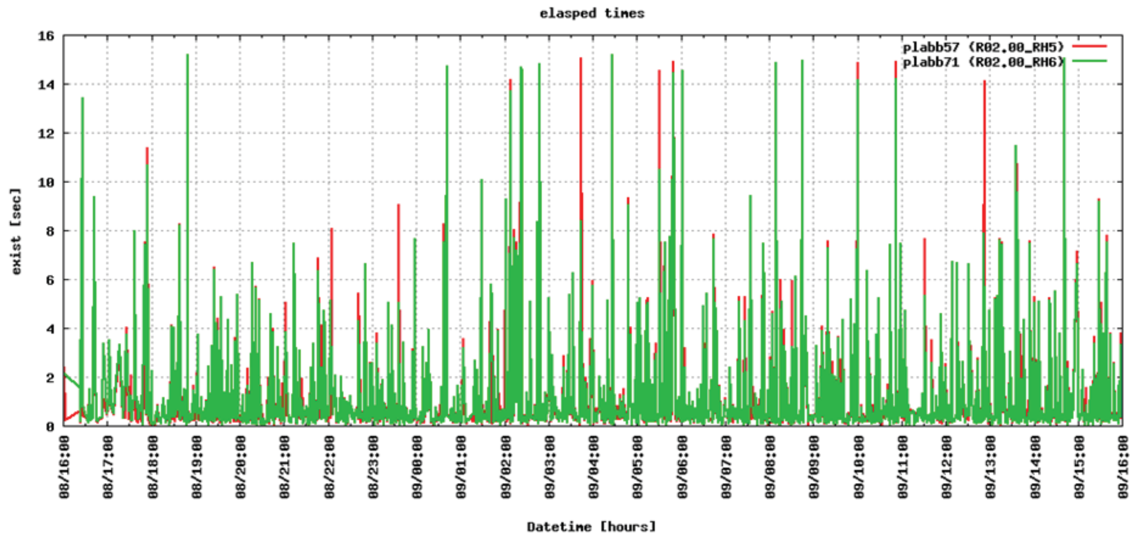


Figure 2.21. Elapsed Time Measurement for Conventional Load Test for New (represeneted by the color green) vs. Current (represeneted by the color red) Airline Shopping Release.

Exist times Show graph				
Parameter	R02.00_RH5	R02.00_RH6	diff	diff 2
Averages	1.20794	1.22693	+1.57%	+1.53%
95th percentile Averages	0.80113	0.82624	+3.13%	+3.05%
95th percentile Value	5.33133	5.35833		
Standard deviation	2.19096	2.17794		
Min value	0.00000	0.00000		
Max value	15.00300	15.18100		

Figure 2.22. Certification Numbers for Elapsed Time After a Conventional Load Test for an Airline Shopping Application.

2.8 Guidelines for designing thresholds, timeouts, and alerts

In this section, we will give a brief overview of how to implement thresholds, timeouts and alerts in applications. These are tenets learned from experience of managing the online application described in the previous section.

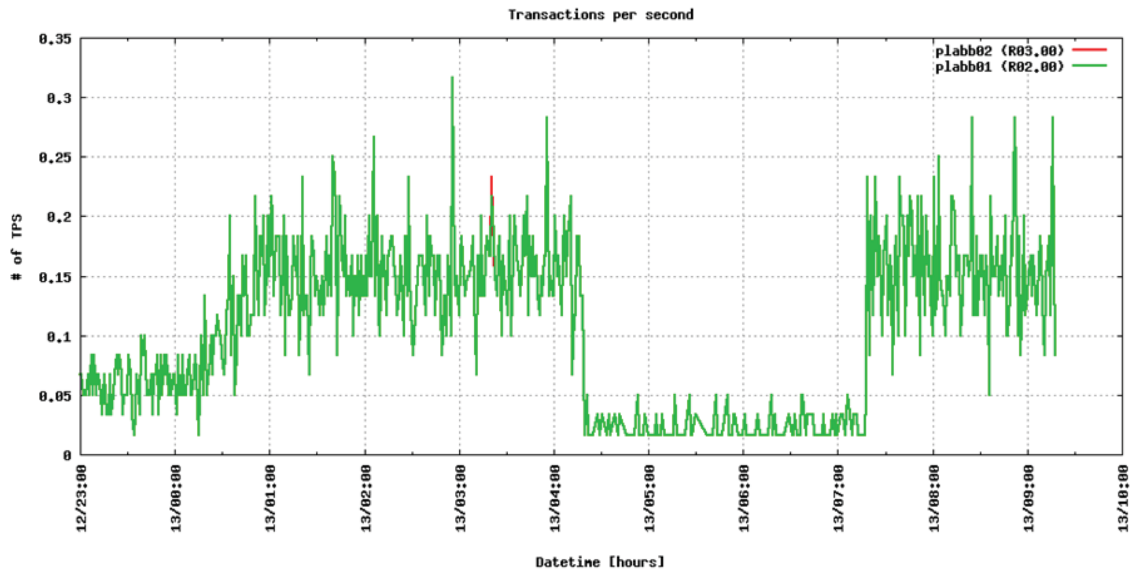


Figure 2.23. Production Like Traffic Profile for an Impulse Test for an Airline Shopping Application. Red solid lines represent the new release and the green solid lines represent the current release .

2.8.1 Thresholds

The goal of process or connection thresholds is to provide a configuration point within the application that permits the application to protect itself against external or internal processing issues. An example of this protection is a client call to a server process that does not respond, resulting in a spin-up of process threads or depletion of pooled threads within the calling client. This section outlines threshold control points within application modules and further details related logging and monitoring activity used to manage these control points. The combination of these provides a real-time view of application stability during both normal traffic processing as well as during traffic surges.

An individual application needs to protect itself from a traffic surge and hence establish control points that can be defined using the following rules of thumb.

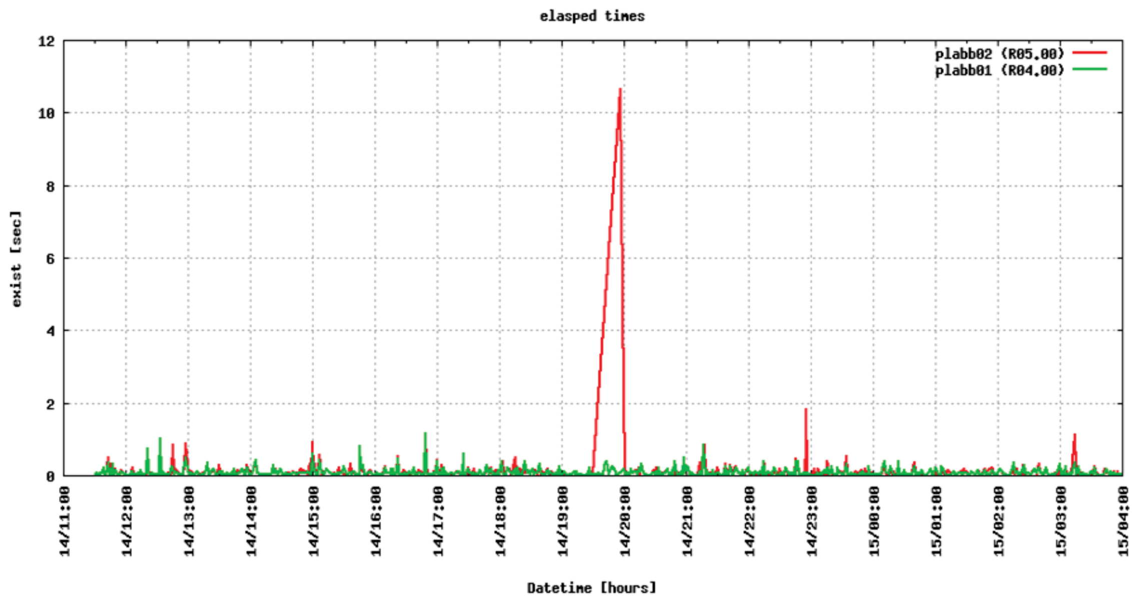


Figure 2.24. Elapsed Time Measurement for Impulse Test for New (represented by the color red) vs. Current (represented by the color green) Airline Shopping Release.

Exist times Show graph				
Parameter	R04.00	R05.00	diff	diff 2
Averages	0.07875	0.10445	+32.64%	+25.13%
95th percentile Averages	0.05967	0.06312	+5.77%	+2.49%
95th percentile Value	0.25600	0.30450		
Standard deviation	0.11127	0.42278		
Min value	0.00000	0.00000		
Max value	1.14100	10.63500		

Figure 2.25. Certification Numbers for Elapsed Time After an Impulse Test for an Airline Shopping Application.

- No single connection, process or I/O interface activity should deplete the main application process of resources to the point at which a module is unable to respond to input transaction requests.

- Each connection, process or I/O Interface should be monitored in real-time to provide an early warning mechanism for the operational coverage teams if processing stability goes out of normal range.

Threshold controls can be implemented either logically or physically within the application.

- Logical Control - A logical control can be described as a gating mechanism within the application. It can be implemented to permit only a given number of processes into a controlled area within the application. If the area is saturated, the next calling process is refused entry and continues down a non-successful path.
- Physical Control - A physical control consists of an independent set of instances of a controlled resource (i.e. threads, memory, and connections) and is utilized through a process handoff activity. If the physical allocation of the controlled resource is depleted, the next calling process hand off is refused and the calling process continues down a non-successful path.

The following should be kept in mind when implementing thresholds.

- The calling process in either of these threshold control mechanisms should never be permanently blocked. However it may be placed under a timed wait and retry condition. If such a retry condition is desired, the retry wait period must be accounted for in the allocation of process threads controlling the calling process driving requests against the controlled resource.
- Timeouts play a key role on a threshold configuration quantity at both entry to the controlled resource and process longevity within the controlled resource sub-service area.

2.8.2 Timeouts

Timeouts allow efficient usage of constrained resources. So instead of the end user waiting indefinitely for a response and the request waiting indefinitely in the queue to be processed, a timeout will let the request expire after a set value. This in turn prevents the system from getting overwhelmed with waiting requests and the customer from expecting a response when there is a possibility that the request will not be processed at all or will take longer than expected. While timeouts alleviate the waiting condition for a downstream server they may also cause usage of upstream servers to go up. Hence it is vital during performance tests to ensure that the waiting times set by the timeout value do not lead to the depletion of upstream resources. Timeouts typically apply in the following situations.

- Process activity that leaves the immediate application and uses a sub-service module to drive a transaction reply (client to server relationship).
- Process activity that leaves the immediate host and uses a remote logic component on another host to satisfy a processing need.
- Process activity that enters a sub-service pool of process threads to perform a parallel or serial long running activity.

While timeouts facilitate the release of a wait condition (client call to remote server), they also have the ability to drive up the use of a constrained upstream resource (input connection pool to this server). Therefore the use of timeouts must be closely observed to determine that no wait period has the ability to deplete upstream resource allocation.

2.8.3 Alerting

It is imperative that an application triggers an alert if a critical event occurs. When a system is properly configured to trigger alerts on critical events, it helps

improve its operability and availability. Timely alerts with clear instructions can have a significant impact on mean time to recovery (MTTR) during outages. The event could be anything from system resource utilization metrics going over a particular threshold to an interfacing component becoming unresponsive. There can also be monitoring and alerting set for specific processes or application logs. Figure 2.26 shows an example of setting up alerts for a memory exception that gets logged in the application logs.

Text Log File

Server	Log File Name (with full path)	Alert Pattern or String	Ignore Pattern or String	Severity	Procedure
SSWHLP321 SSWHLP322 SSWHLP323 SSWHLP324 SSWHLP221 SSWHLP222 SSWHLP223 SSWHLP224 SSWHLP225 SSWHLP226	/logs/prod/KM/<sswhlpXXX>/inst1/qtripNG_KM_<date>-<seqnum>.log /logs/prod/B6/<sswhlpXXX>/inst2/qtripNG_B6_<date>-<seqnum>.log /logs/prod/B6/<sswhlpXXX>/inst3/qtripNG_B6_<date>-<seqnum>.log /logs/prod/B6/<sswhlpXXX>/inst4/qtripNG_B6_<date>-<seqnum>.log	java.lang.OutOfMemoryError		2 (if more than one server is experiencing this problem)	1

Figure 2.26. Example of how to look for string patterns in Application Logs and then set Alerts..

Setting up such a trigger is vital to maintain the operability as well as the reliability of the entire system. These alerts can initiate the timely correction of these situations and often prevent the end user from experiencing service deterioration. It will also prevent the issues from propagating to other tiers and thereby causing the system to become unstable.

2.9 Test Data Correlation

The decision to deploy a particular release in production depends on the results from performance tests, which therefore should be reliable, giving stakeholders the confidence to trust the numbers. Even though performance engineers and application

teams try to create a test environment that mirrors production, it usually does have certain differences. That is, the test environment is typically a scaled down version of the production environment. For example, the storage area network used for databases in test might be slower than the one being used in production or the network configurations might be different. For this reason it is common practice to set a 5% plus or minus acceptable range between the test and production numbers. In this step, we compare results from the test environment with actual performance numbers seen in production. This enables improving the test harness, test cases, data collection procedures, scripts and workload analysis. Correlation needs to be done for each metric that is collected during tests.

The process of test data correlation is to take the 95% percentile value for each measured performance metric during a successful test and then compare that with the 95% percentile value of the same metric in production after the new release has been deployed. During testing, scenarios of varying load along with peak traffic are simulated during a 48 hour interval, so resource utilization numbers measured during tests might be higher than those observed in production. This along with incorporating the differences associated with the test environment as compared to production is the reason why a plus minus 5% range between the numbers is considered acceptable. The duration over which the numbers are collected in production also needs to be closely monitored. Performance tests are run for short durations as compared to the life of an application when deployed in production. The traffic profile used during testing which might contain new types of transactions might become live only after a few days of application deployment. For this reason the metrics in production should be collected after a few days of the application going live. This will give us a basis to compare the performance of the same release under the same traffic profile in the two environments.

Table 2.3. Correlating production and test for Airline A.

Release	**pLab Obs**	**PROD Avg.**	**% Diff from PROD Range**
2008.10.06	1.32	0.61	+114.77
2009.00.00	0.66	0.64	+0.00
2009.01.01	0.81	0.45	+55.24
2009.02.02	0.51	0.39	+21.07
2009.03.01	0.93	0.47	+77.73
2009.04.02	0.85	0.42	+90.10
2009.05.00	0.53	0.41	+0.00
2009.06.00	0.50	0.54	+0.00
2009.07.02	0.59	0.50	+0.00
2009.08.00	0.56	0.55	+0.00
2009.09.01	0.60	0.58	+0.00
2009.10.01	0.66	0.57	+2.81
2009.11.02	0.55	0.58	+0.00
2009.12.01	0.80	0.56	+31.03

2.9.1 Example: Correlation for Airline A

Table 2.3 shows the correlation of exist times even called elapsed time (the time taken by a piece of software to execute a particular transaction) for the sample flight shopping application between test and production. The difference between exist times numbers recorded in test are massively different from the ones that were recorded from the release in production. This indicates that changes have to be made either to the environment setup, data collection technique, test scenarios or even the workload being used. Since the test results are not comparable to production numbers, the tests being run are not reliable. After changed are made to better the system from release 2009.05.00 onwards correlation data looks more favorable. This exercise ensures that proper tests are being run using the right traffic profiles.

2.10 Related Work

This chapter extends [4] where the authors had briefly described the importance of adding certification and correlation to a standard software lifecycle. In this chapter the additional steps are illustrated in detail with the help of real world examples. It also introduces the setup of the test environment that is used for the certification process along with a test called impulse test. Traffic levels in a production environment are neither constant nor contain only traffic spikes and surges. Real traffic loads vary widely and contain both peaks and troughs. This chapter describes a new performance test called impulse test that is designed to imitate real live traffic more closely and includes the characteristics of several conventional performance tests. The design of this test consists of a period of peak load, followed by a short load surge, a load drop, and a subsequent peak load. Impulse tests are designed to uncover not only individual performance bugs but also the end-to-end application behavior, including its throttling, timeouts and alerting mechanisms. Impulse test combines the features of a stress, load and soak test in one, saving time of running each of these tests individually but still being able to identify bottlenecks. The test environment is setup with two identical paths for the application under test and then the impulse test is run simultaneously on both paths with the same input requests to record the performance numbers for the applications deployed in both paths under the same conditions. These performance numbers are then used for certification and correlation.

A lot of work has been done in the field of performance testing and measurement [5, 6, 7, 8]. But the majority of the research is devoted to web testing and load drivers that can drive traffic to web applications. [9] discusses what metrics should be collected during testing to measure performance, which are limited to client side response times and errors. It also explains how Little's Law can be used to determine

if test results are close to numbers calculated by the law. How well a particular piece of code will perform is dependent on more than just how much throughput it can sustain or how fast it processes transactions.

[3] describes testing processes and automation taking web applications as examples. [10] describes the importance of workload analysis and gathering of test requirements and explains how test cases and traffic profiles can be created. [11] talks about performance testing in distributed computing environments and how performance test cases can be derived from system architecture design. In addition to this emphasis also needs to be given to a procedure to confirm that the test cases, data collection and workload used during testing are emulating real world scenarios. The certification and correlation phases described in this chapter, added to the general performance testing guideline will help in determining all factors, system and application that affect the performance of a particular piece of code. These two steps will not only be able to provide the necessary performance evaluation process but also help in validating that the right set of activities are being followed during testing.

CHAPTER 3

SOFTWARE TESTABILITY

3.1 Introduction

Every major industry today (banking, travel, social media and gaming, entertainment, retail, etc.) is dependent on computers in the backend to sustain their business. With more and more users online it is necessary for businesses to deploy reliable and stable computer systems that can be accessed 24x7 and provide all needed services in a timely fashion. With end users being spread out all over the world accessing systems at different time zones, it leads to the need of computer systems to be complex and distributed but at the same time to have uptimes of higher than 99.999%. Managing such environments and maintaining their high availability and dependability can be an arduous task. Different teams are responsible for different components and pieces of software that will eventually make the whole end-to-end system. Each team has its own coding and programming language standards, operating system and hardware requirements, documentation, risk and release management practices. Once development is complete each one of these components needs to integrate with other pieces of software and work seamlessly to provide services to the end user. It is the responsibility of each individual team to validate and test their software before deploying in production but often this kind of individual testing does not give an accurate assessment of the performance of the entire system. Most of the research today on testability is dedicated to finding ways in which to make the software code more efficient and on how to add probes that would enable the ease the creation of test cases and testing the code. When the end goal is the smooth working of an en-

tire end-to-end production environment that can at any given point of time be made of several different components ensuring functional soundness of individual pieces of software, though vital, is only part of the effort. Design standardization across the entire organization for engineering aspects like timeouts, throttles and alerts when a particular metric goes over a specified threshold or establishing the use of a common instrumentation strategy across all development teams or uniformity in the data/-metrics that are being collected when the application is live, creation of a consistent test harness and standard processes for setting up a test environment all eventually lead to, only the simplicity and ease of testing but also help in establishing a stable and robust production environment. These common software testability practices help with not only uncovering performance issues but also the timely resolution of problems. When there is a common instrumentation strategy for example and a standard list of the metrics that need to be collected, it helps testers with the discovery of bottlenecks but also helps the operations team when the application goes live in production. An example of this can be seen in the next sub-section.

3.1.1 Example: Insufficient Instrumentation of a Commercial Hotel Booking Application

Figure 3.1 shows the setup used for performance testing of a hotel booking application. The load driver on the left generates XML requests to check hotel prices and places the requests in a queue. The requests are picked up by the application that in turn uses a backend simulator to get the prices for specific hotels.

The load driver then picks up the responses from the queue and produces a final report as shown in Figure 3.2.

The Load driver is the only component in this system that is creating any sort of logs or reports. It is definitely essential to have good logging and reporting on the

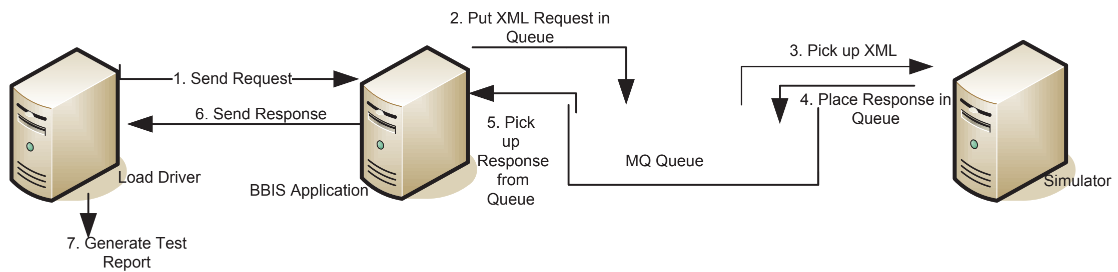


Figure 3.1. Data Flow for Hotel Booking Application.

SCENARIO COMPLETED
totalToProcess = 180000
totalWasSent = 180000
totalWasReceived = 164494
totalNotReceived = 15506

TEST RESULT
Message name : PriceXML
sent - 180000
received - 164494
not received - 15506
mean - 32357.39510863618 ms
sigma - 67059.83464903958 ms
max - 344609.0 ms
min - 9.0 ms

Figure 3.2. Test Logs for Hotel Booking Application.

load driver side in order to gauge the client side behavior of any system. But there are certain major gaps in the information that has been generated at the end of the test. The report gives a detailed analysis of the response times, which is excellent but does not provide complete data about the XMLs. By looking at the totalWASSent, totalWasReceived and the totalNotReceived numbers, we know that out of the total number of XMLs generated some were lost but with simply the load driver side logs to

look at, it is impossible to identify the root cause for these missing messages. There is also no way to investigate the cause of the huge response times - the delays could be on the queue, application, simulator or even the network side. There is also no record of the percentage of errors or the types of errors that might have occurred during testing that might have caused the number of lost XMLs.

This system is an example of insufficient logging at the load driver end as well as lack of any kind of logging at the application side which would ultimately lead to inconclusive testing. Logging is just one of the attributes that facilitates software testability. Performance or non-functional testing is a critical and a multi-faceted task with the goal being to catch all performance bottlenecks during the testing phase. With so many variants like hardware, software, operating systems, different timeouts and throttling levels for each application, alerting thresholds and diverse instrumentation practices, it becomes tricky to accomplish this task. Hence it is of utmost importance to educate development teams and standardize certain features that are part of software testability that will ultimately not only facilitate testing but also the help reduce the costs associated with monitoring a distributed production system. Adopting common practices reduces cost by avoiding one-off solutions developed by application teams. It also allows teams to focus on product differentiation and not spend time and resources on common utilitarian functions. In addition it also contributes to operational stability and improves application time to market.

3.2 Software Testability

Software testability covers various areas of software engineering. It is a process by which one can assess how good or bad a modified piece of software would perform. Testability is the inherent property of a piece of software and the degree

to which the artifact makes it easy to assess its performance and functionality in the context of a given test (test goals, methods, scenarios and resources). Certain software characteristics make it easier or harder to test it and to analyze the results. These software attributes define software testability. Hence designing for testability should not only help improve the probability of finding the system's defects during test executions but also help with the deployment of a fully functional and stable production environment along with its monitoring and problem resolution. For a system to be testable, it must be possible to control each of the inputs into the system and thereby the internal state and then, to observe each output of the system including any side-effects. Testing is typically done using a test harness - which could be an automated, in order to run tests for the piece of code. This could include functional testing tools as well as performance testing tools. Software Testability is an important concept and designing for testability reduces the cost associated with development and increases reliability of any testing effort both for ad hoc development as well as for teams/organizations with a high level of process maturity. Building software and other system components with good testability leads to simplification of test operations, reduction of test costs and increase in software quality which in turn helps with establishing a stable end-to-end system. Several characteristics enable the ease of software application testing. These factors can be categorized as internal, external and environmental as shown in Figure 3.3. These factors as described in more detail later in the chapter, and have been collected from experience working in the industry and the published work such as [12], also discussed in this chapter.

A few of these factors such as controllability, operability, coding standards and observability are well studied as guidelines or high level abstractions and listed in [13], [14] and [15]. As is evident from Figure 3.3, coding standards and development of efficient applications is just one of the many features that enable software testability.

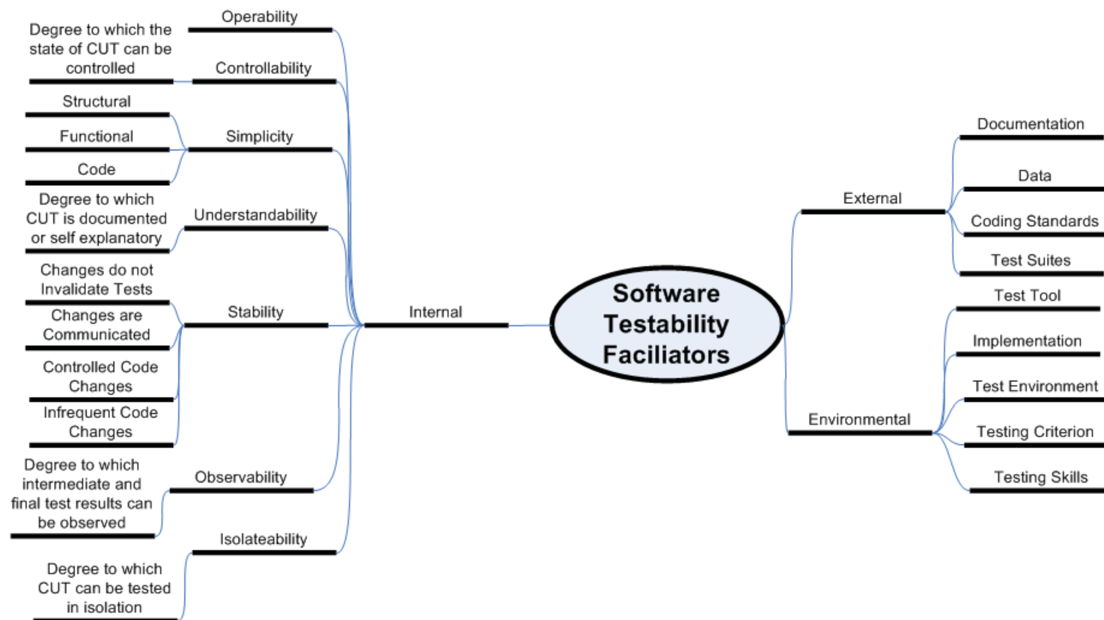


Figure 3.3. Software Testability Enablers.

A lot of research work has been dedicated to this one aspect and hence in this chapter we will look at some of other tenets that are most often ignored but nonetheless are vital in not only reducing testing efforts but also improving the overall operations of the production environment and reducing cost when managing large, distributed systems. The two factors selected and described from each of the categories - internal, external and environmental are selected from experience that are known to be critical aspects to improve testability but are not often discussed in this detail. The following sections give practical examples of how implementing these software testability facilitators help with the overall improvement of software quality, ease of testing as well as reduction in development and operational costs.

3.3 Characteristics of Software Applications that Affect Testability

The first category we will discuss deals with the internal features of the software application that affect testability as shown in Figure 3.4. The application scenarios mentioned here are from existing travel legacy applications that still need to be supported. A few of these challenges can be avoided by redesigning and rearchitecting these legacy application solutions but that is a costly endeavor and hence need to be supported as is.

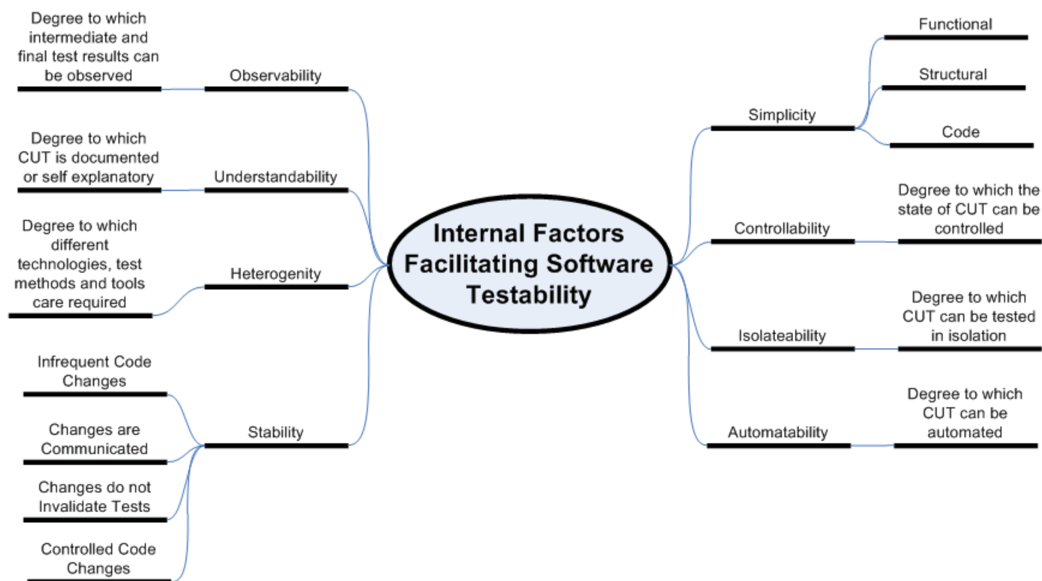


Figure 3.4. Characteristics of Software Applications that Affect Testability.

1. Controllability

    - GUI Based Applications

    Challenge: Ease of use and understandability by the end user has led to many applications being GUI based. Where these GUIs lead the customers to effortlessly navigate through a particular application, running consistent automated

performance tests for such applications is most difficult. The main reason for this is that a slight change in the GUI - be it changing the position of a button or adding a new menu, basically makes the test script unusable. Some of the other reasons that make it harder to test GUI based applications are:

- The business as well as the database logic is highly coupled with the GUI
- They use their own libraries to draw Gantt charts and other elements on the canvas. These elements are often dynamic - the content changes based on the backend database.
- The existing testing tools including Load Runner are able to create static test scripts where the user clicks are recorded for a particular instance of the GUI, but stop working when the GUI content changes.
- These applications do not have an API for testing
- Only possible approach to testing is to send GUI events (like mouse presses, key presses etc.) to the application.

Solution: One way to overcome the issue of writing reusable test scripts for GUI based applications is to provide a scriptable interface for all key functions. This could be an API in a standard programming language, a shell interface, a messaging interface or an HTTP interface. This is extremely important for GUI interfaces because as discussed above they are difficult to script - the scripts often depend on the screen position of various objects and are therefore, quite fragile. A "fake" interface that invokes the same function as a button click does go a long way towards making your scripts more reliable. Consider the example in Figure 5. This is a JBoss based application that has a front end GUI. In order to circumvent the GUI and create good test scripts that will exercise all the functionalities of the application and business logic, the application team will create an API which would in turn let the load driver to

invoke actions/functionalities without the need to clicks on the buttons on the GUI. This can be achieved by utilizing RMI calls that mimic user clicks on the GUI. Do keep in mind that the API is part of a normal thick client GUI and has all its parts except the GUI itself. This enables the creation of full client sessions emulating real clients.

2. Observability

   - Instrumentation and Data

   Instrumentation is one of the most important features that will not only help in identifying issues but also help in resolving them. Using a standard instrumentation technique which is exposed to the applications via an API, for the entire organization helps ease not only problem identification and resolution during performance testing but also once the application is deployed in production. Through consistent instrumentation, operational visibility is improved thereby leading to improved stability. Common monitoring tools provide a method of quickly observing both application performance as well as an end-to-end view for both operations and application teams. When teams adopt common instrumentation tools, the data including application metrics, billing, security and errors can be stored in a common repository without difficulty. From this repository business object reports can be used to derive accurate forecasting and capacity data. Operations team also have a consistent and reliable tool to help them with their day to day work. In addition, capturing all data in common repository allows for reducing the infrastructure footprint by eliminating redundant data storage used by different application teams.

3.3.1 What to Instrument?

This sub-section deals with what metrics are important to measure. As discussed in the related work section the most common metrics collected during performance assessment are response times, throughput and then server specific utilization data like CPU and memory. Though vital these metrics do not cover the total performance measurement of a particular application. I.e., without the right kind of metrics as shown in the example in Figure 3.1 and Figure 3.2, it is impossible to validate the performance of an application, identify performance issues or assess how a particular piece of software will behave when it takes real customer traffic in the production environment. The following is a list of a few significant metrics that are often overlooked but have shown to be important from experience, grouped by the location at which they are collected – the application side, server on which the application is running and also from the database side (if any). These metrics help with identifying both performance bottlenecks as well as measuring application behavior. Application side metrics:

- Failed transaction count
- Type of alerts and errors
- Heap memory
- Threads
- Inbound and outbound connections
- Elapsed and cpu times

As seen from example for Figure 3.1, it is of utmost importance to measure the number of failed transaction count and the alerts and errors. This helps with tracking the functional performance of an application from one release to another. Memory utilization metric and thread usage help in ensuring the health

of the application at any given point of time. Metrics that measure connections and the processing times for the transactions help troubleshoot performance issues for the specific application. Server side metrics:

- File descriptors
- Context switching
- Process/threads created per second
- Paging (swapping memory to or from disk)
- Process CPU and memory
- Load average

Just as with the application side metrics the operating system metrics help keep track of the health of the server on which the application is running. Performance issues with the server can also cause issues with the application, so it is critical to have the hardware monitors in place. Measuring file descriptors helps knowing how many more threads for a process can be sustained on the server. Context switching, CPU, load average and memory all help with tracking the performance profile of the application. Huge number of context switching, paging and memory swaps are clear indicators of a performance bottleneck. Database side metrics:

- Connections
- Threads
- Types of alerts and errors
- Server utilization metrics

Similarly instrumenting the database ensures the end to end stability of the system. The database server utilization and thread pool numbers as well as the exceptions and the alerts thrown are all needed to monitor the system at any given point of time. The data that is collected during testing as well as

when the application goes live in production should include both key metrics and critical events. A common instrumentation solution described in the next chapter ensures that both these aspects of performance measurement are taken care of. Accurate logging of key system metrics and errors, both internal and external (responses to downline requests) helps with the following:

- improves visibility into the working of the system
- simplifies troubleshooting and
- enables predictive alerting

3.3.2 NOFEP

In order to further explain the importance of standardizing the collection of these finer metrics, consider the following application NOFEP (No Open Front End Processor) whose architecture is shown in figure 3.5

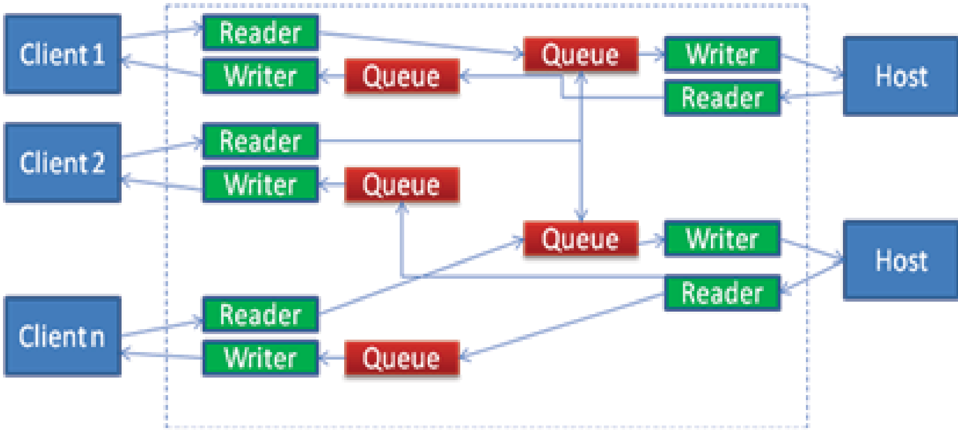


Figure 3.5. NOFEP Architecture.

This application is a front end application that accepts tens of thousands of client connections to route messages from the clients to the hosts. It is capable of processing thousands of TPS. NOFEP uses two threads per connection - one for reading and one for writing, using blocking calls. The application uses send() and recv() and blocks them until a send/receive is complete. Using asynchronous send/recv would be a performance issue because then the application would have to iterate over all open connection looking for I/O ready sockets. The problem with this kind of design is that the server struggles to scale beyond 20K connections due to the large number of threads that are being created. Even opening of the connections is slow due to the thread creation time. With 30K connections open, the server can spend 50% or more of CPU time in the kernel due to scheduling. The Figures 3.6, 3.7 and 3.8 show the number of connections that can be opened during a specific time duration, the server CPU usage and the load average on the server during the time the connections were being established.

These issues with thread contention and scheduling would never be discovered if there was no context switching data or load average data to look at. Once the performance bottleneck and its cause has been identified, it is easy to redesign the application and resolve the issue. Standardization of collecting finer metrics like threads and context switching and making them part of the process of designing for software testability, not only assists with discovering performance bottlenecks during testing but also helps with improving the architecture and hence performance of an application, as in this case. In order to get over this thread contention problem the application was redesigned using the epoll system to manage connections. Epoll allows specification of a set of connections which the operating system places in a table. Then one can block until a connection

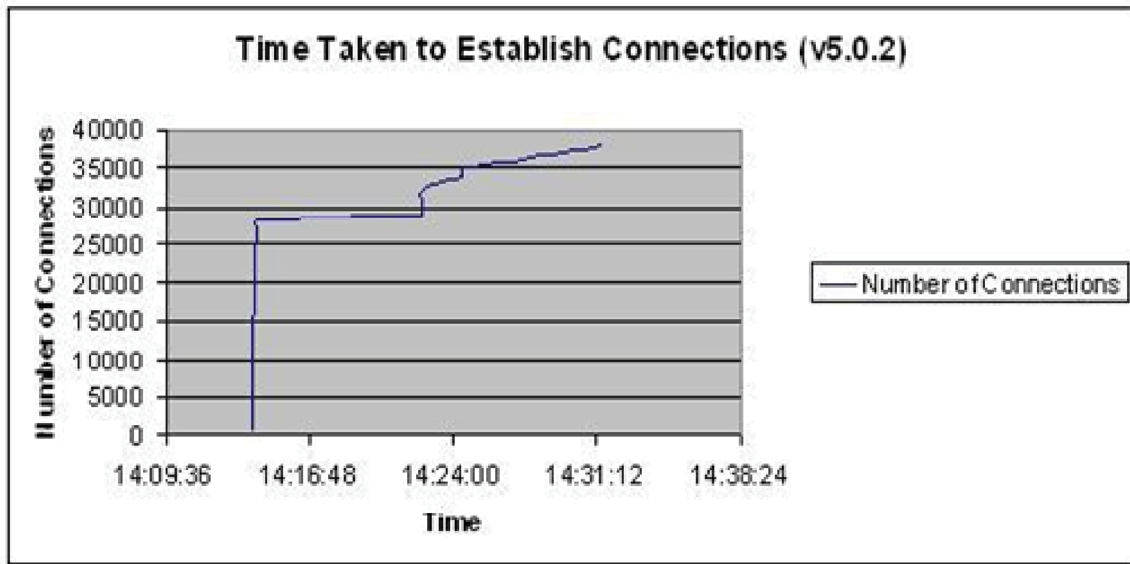


Figure 3.6. Time taken to create connections with existing NOFEP architecture.

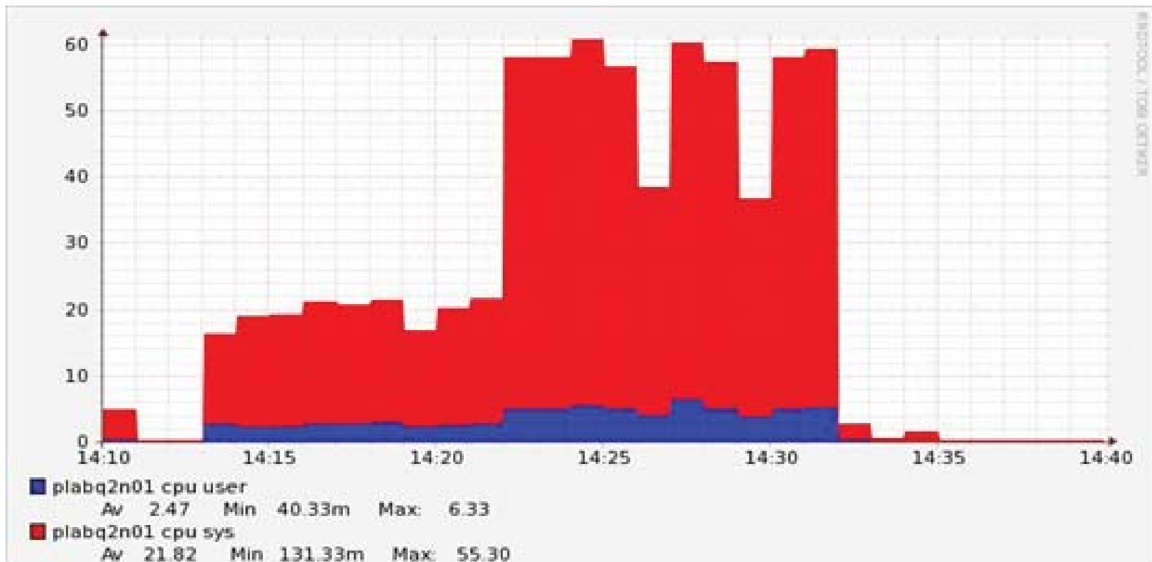


Figure 3.7. CPU usage of a NOFEP server with existing architecture.

is ready to read/write. This allows one thread to handle many connections as shown in Figure 3.9.

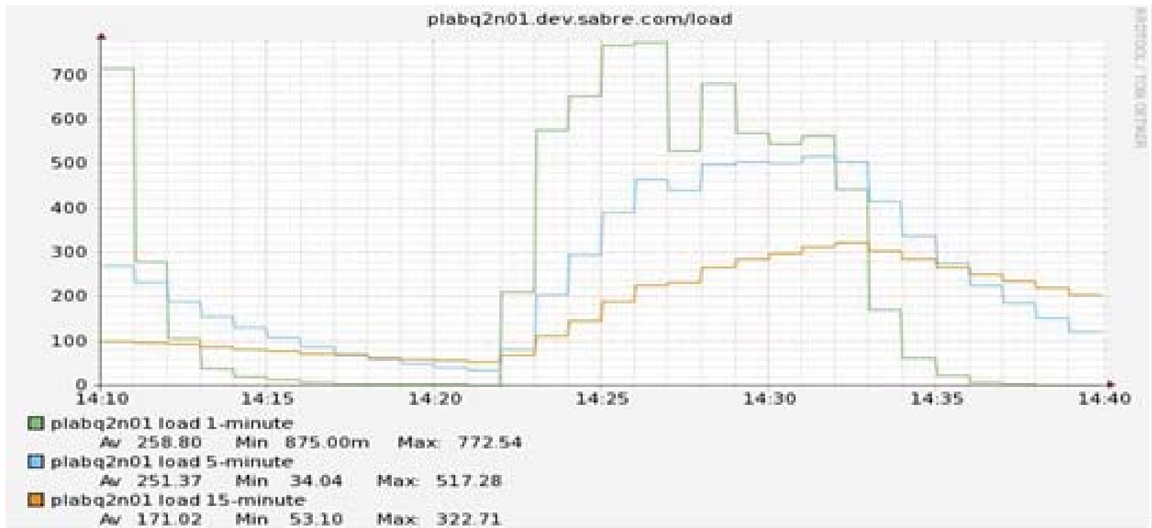


Figure 3.8. Load average of a NOFEP server with existing architecture.

The application now uses a pool of reader and writer threads to transfer messages from the client to the host. And the improvement in resource utilization and the time to create the connections is immediately visible along with the increase in the number of connections that the application can now handle as can be seen in Figure 3.10.

The CPU utilization numbers as well as the load average on the NOFEP server has gone down when using the new NOFEP architecture as shown in Figures 3.11 and 3.12.

This example just goes to show that collecting more fine grained metrics helps with better evaluation of the application's performance. Using these metrics would not only help with detecting performance bottlenecks but like in our case, also help with improving the architecture and hence increase the productivity of the application.

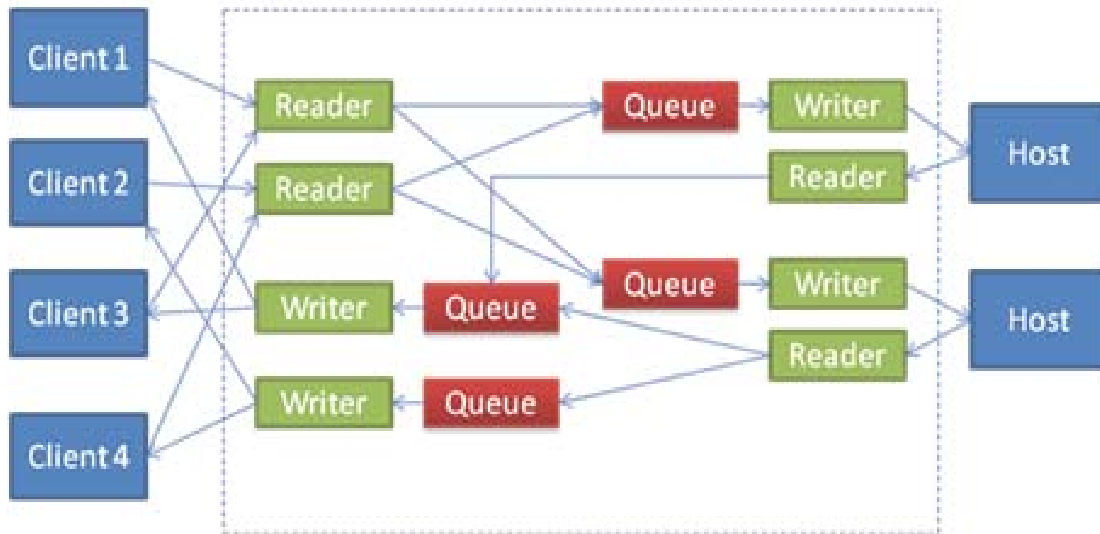


Figure 3.9. New NOFEP architecture.

3.3.3 Standard Instrumentation

Instrumentation is one of the most important features that will not only help in identifying issues but also help in resolving them. The next chapter will describe a standard instrumentation tool built keeping in mind the characteristics of software testability. Using a standard instrumentation technique which is exposed to the applications via an API called enterprise instrumentation API, for the entire organization helps ease not only problem identification and resolution during performance testing but also once the application is deployed in production. Through consistent instrumentation, operational visibility is improved thereby leading to improved stability. Common monitoring tools provide a method of quickly observing both application performance as well as

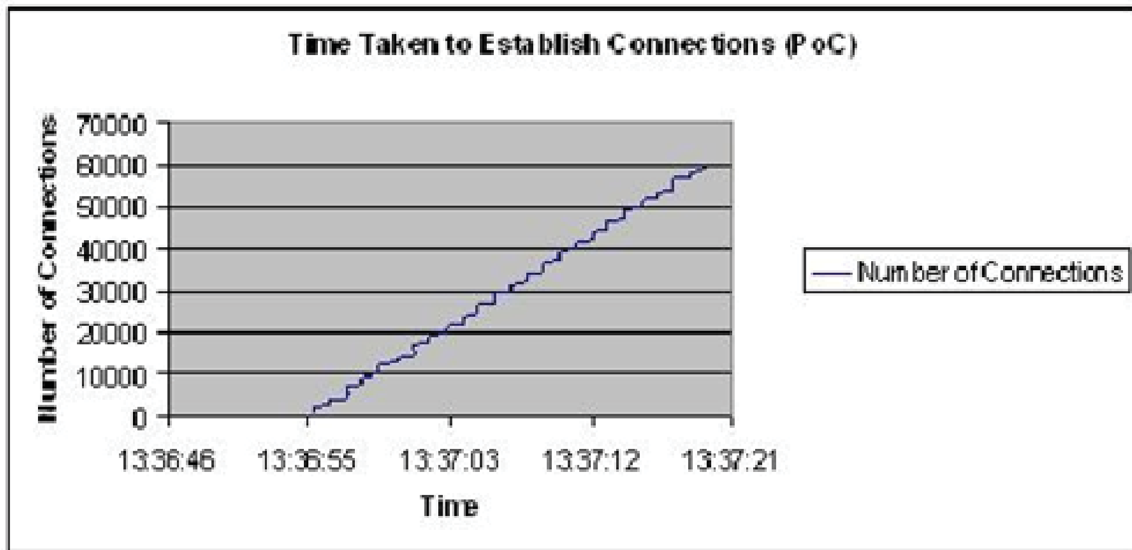


Figure 3.10. Time taken to create connections using the new NOFEP architecture.

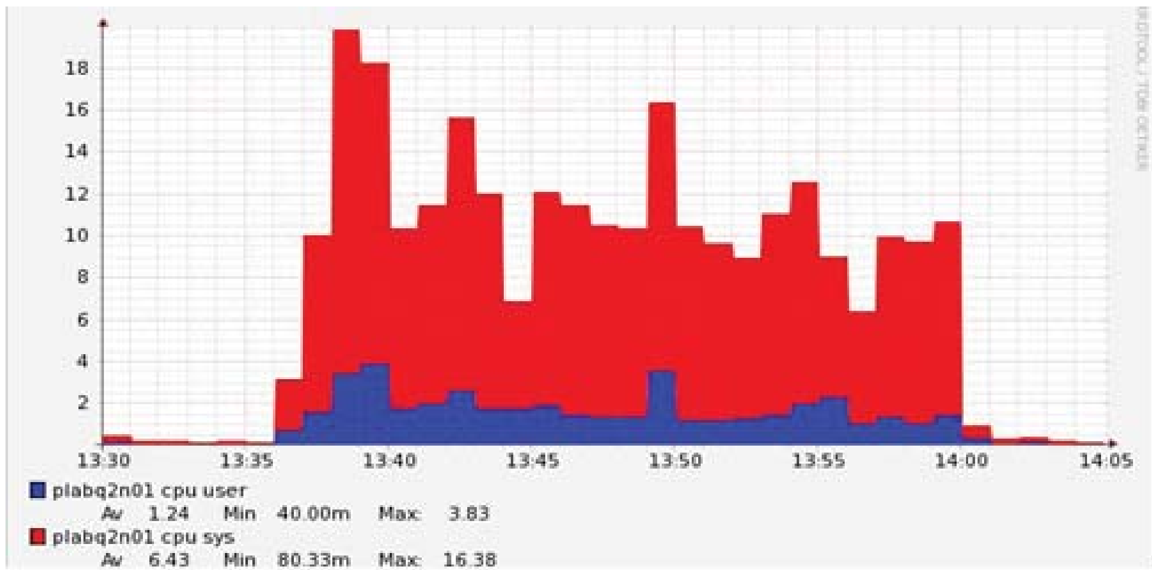


Figure 3.11. CPU usage of a NOFEP server running the new architecture.

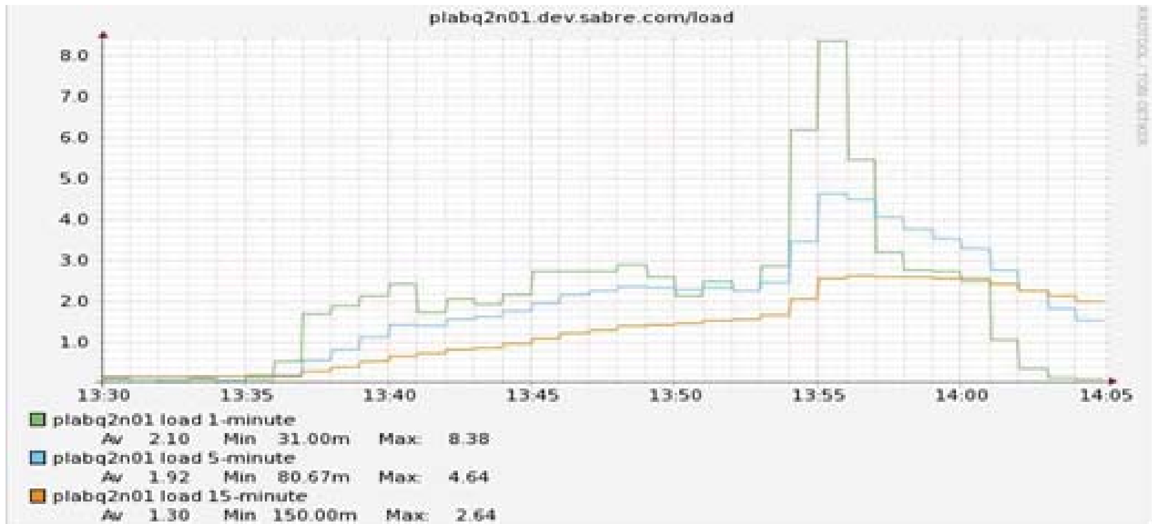


Figure 3.12. Load average of a NOFEP server with the new architecture.

an end-to-end view for both operations and application teams. When teams adopt common instrumentation tools, the data including application metrics, billing, security and errors can be stored in a common repository. From this repository business object reports can be used to derive accurate forecasting and capacity data. Operation teams also have a consistent and reliable tool to help them with their day to day work. In addition, capturing all data in a common consolidated repository allows for reducing the infrastructure footprint by eliminating redundant data storage used by different application teams.

3.4 Characteristics External to Software Application that Affect Testability

In this next section we will look at two factors external to the software application that can have direct or indirect affect the testability of the application but are often ignored. A list of these factors [12] is shown in Figure 3.13. One of these

features that will be discussed is coding standards. Even though there are numerous papers and research done on ways to improve code quality in order to ease testing, little has been said about incorporating engineering tenets in the code design. We will discuss a couple of those in this section.

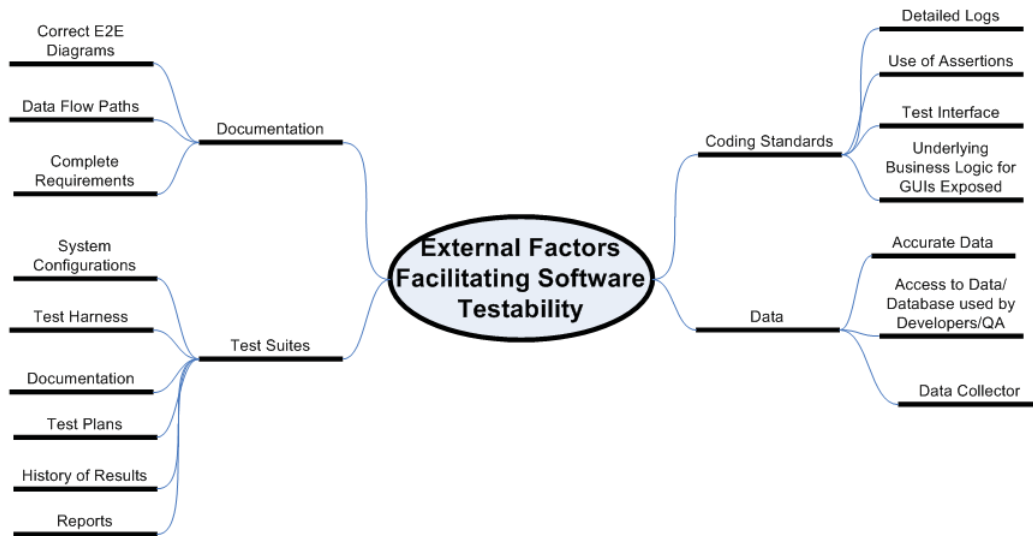


Figure 3.13. Characteristics External to Software Application that Affect Testability.

1. Test Suites

   Creating the right test harness and scripts thereby developing the correct test suite is of utmost importance in performance testing. In order to accomplish this task it is necessary for application teams to design their systems such that it can be isolated from various external dependencies. The mock/simulator implementation for every external dependency can have various levels of sophistication:

   - Simple mock with a constant and uniform response,
   - Record and replay, or

- Simulating specific data conditions

A sample of an automated test harness in an environment using a mock is shown in Figure 3.14. Usually simulators/mocks are used in the test environment to emulate the processing of complex backend applications or third party software. In cases of record and playback at the test script side in an environment using a simulator it is important to synchronize the test script and the simulator. The main reason for that is to make certain that both the traffic generator and the mock use the same delay between transactions so that actual results being generated during the test match the expected test results. The test entries are taken from the application logs in production and contains the delays recorded between the transactions. The test scenario and entries file contains these delays as recorded in production. Both the simulator and the test script also need to read the same configuration file that describes the test scenario and test entries. There is an additional component used in this kind of setup - expected results generator. The results generator reads the configuration file, parses the scenario and creates graphs for the various metrics plotting the expected behavior. These expected results are stored in a database. While the test script is running, it records the responses from the test environment and stores the test data in a database. Either during the test or after full completion the comparison and reporting component can be used to see how well the test is emulating the expected scenario.

2. Coding Standards

Every developer is habituated to writing code his/her way even when using the same programming language. This often leads to being innovative during the testing phase to detect bottlenecks. Enforcing coding standards may reduce the effort spent during testing and also increase the quality of the reports.

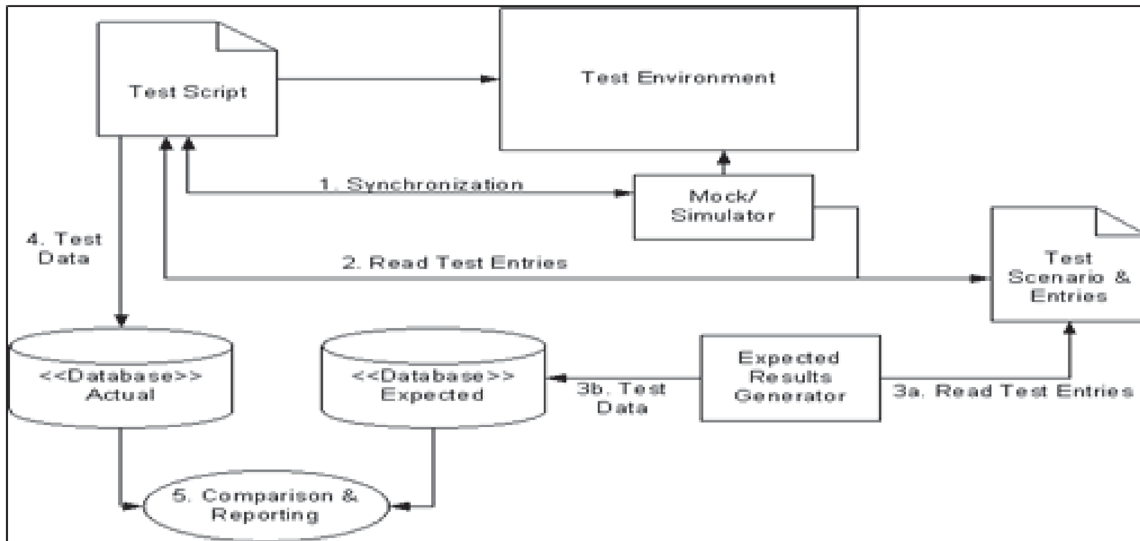


Figure 3.14. Example of an Automated Test Harness in a Test Environment with Backend Simulators/Mocks.

For example assertions can be used at run time when available in the development language and can be used through an interface or a test tool. Most of the work when it comes to software testability is centered on coding standards. So in this chapter we will look (at a very high level) at two engineering aspects - timeouts and throttles that should be part of the application design and code. These features help with the overall stability and reliability of the end-to-end system and are discussed in detail in Chapter 2.

(a) Timeouts Timeout is a feature/setting that enables applications to handle a lack of response (or extended latency) by logging and returning an error to the end user that initiated the request. All timeouts should be associated with some kind of post timeout action. Timeouts should be set for:

- The application's connection(s) to a database(s) or directory(ies)
- Responding to an incoming transaction

- Communication between its different tiers. This includes Web/App and App/App transactions
- Receiving a response to a connection the application makes to an external resource

(b) Throttles Throttling allows workload to be adjusted without causing any major failure or performance degradation. It is critical that all applications are designed with throttling controls for smooth operation even during times of unexpected peak loads. All throttle events should be logged and instrumented with alerts and/or alarms. Throttles should be employed for

- Applications to protect themselves from unexpectedly high rate of incoming requests
- Databases to set a ceiling on the number of connections that can be opened in order to protect itself from overload

3.5 Environmental Characteristics that Affect Software Testability

Last but not the least are the environmental software testability enablers as shown in Figure 3.15. Establishing a standard way of setting up the test environment is vital because it introduces common best practices that are then utilized for every application, organization wide. Every application is tested and measured using similar processes.

1. Test Environment

   Setting up an environment that mirrors production is ideal for assessing the behavior of any application. But due to the differences in hardware and/or network configurations, it is not always possible to setup an environment that replicates production. When it comes to setting up a test environment, it is not

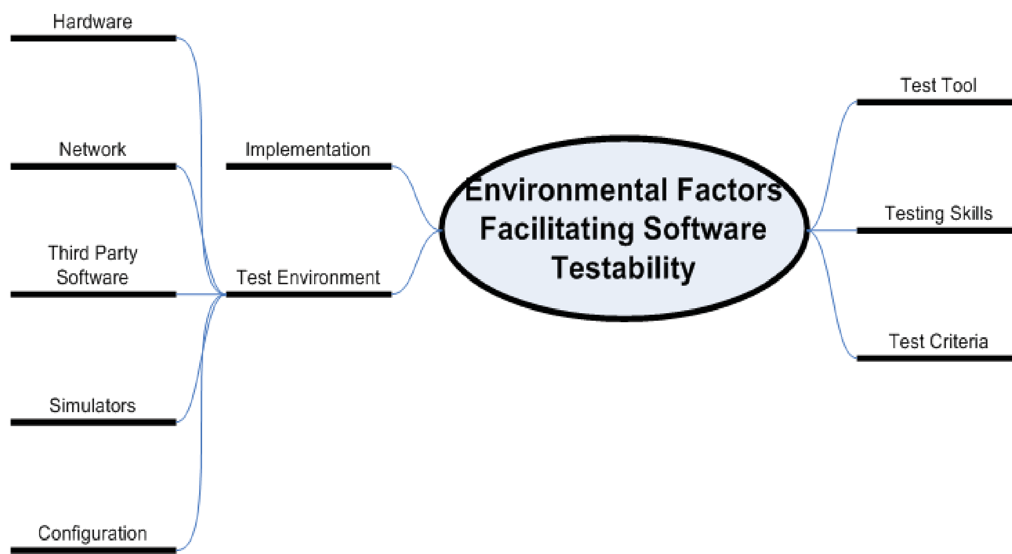


Figure 3.15. Environmental Characteristics that Affect Software Testability.

always just the type of hardware that needs to be considered. Figure 3.16 shows some of the factors that need to be considered when creating a test environment. Differences in the network configurations and even operating systems or software versions have to be taking into account. For example using the same jdk version in test as the application will use in production plays a big part is accurately assessing its performance. Other factors that should be considered are the tools that will be used to monitor the environment and collect data during testing, interactions between various components that make up the environment and even backend processes like cleanup batch jobs or end of the day report tasks that might run on a server in production but not in a test environment. Establishing common methods and processes of setting up test environments not only facilitates the ease of testing different applications but also provides a common benchmark to create a test bed that is as similar to production as possible.

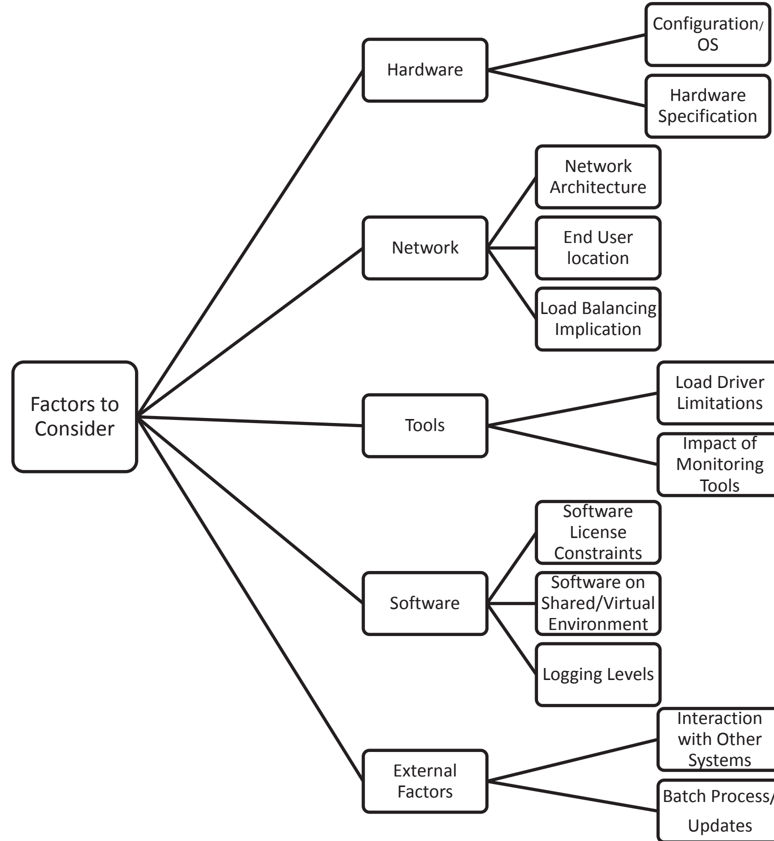


Figure 3.16. Factors to Consider When Setting Up a Test Environment.

2. Test Tool

   A good test tool or test harness is needed to run valid performance tests that emulate real world scenarios. Load drivers and test scripts play a vital part in performance testing, and hence they have to be created carefully. The same principles that are followed to develop a good piece of software need to be used to create test scripts. Test tools are a vital part of the success of a performance testing project. They drive the traffic, they simulate the test cases - both destructive as well as happy path scenarios and they record the client side

experience metrics like response times. Just like any other software, test scripts and tools should have the following qualities:

- Readability: It is important to have readable scripts that incorporate clean logic and are easy to understand. It makes it easier for a new tester to take over and work with them.
- Extensibility: This allows for script changes to be made and more test scenarios to be added in order to assess an application's performance.
- User friendliness: Test scripts do not need to have a GUI interface to run, but even when a command line is being utilized it should be simple and easy to understand and use.
- Efficiency: For many cases in performance testing it is required to simulate thousands of concurrent users or transactions at the same time, which might require running multiple instances of the script. Therefore it is important for these scripts to be efficient and light weight and use minimal server resources.
- Reusability: A production environment is usually made up of numerous applications and components that work together to process user requests. This leads to the importance of reusability - to be able to reuse pieces of a test script code to build scripts for other applications.
- Atomicity: Test scripts need to have a modular design, where each module is dedicated to a implementing a specific logic or test case. This speeds up the modification and enhancement process.
- Correctness: It is essential that the test scripts are able to simulate all documented test scenarios correctly in order to ensure successful performance testing.

In addition to satisfying the above mentioned requirements and generating traffic, a test tool needs to perform certain other functionalities. It needs to keep track of user side metrics like errors percentage, response times and throughput. If need be it should also be capable of recording and playing back production traffic. A more detailed description of test scripts/tools will be provided in Chapter 5.

CHAPTER 4

ENTERPRISE INSTRUMENTATION: A LIGHTWEIGHT CROSS-PLATFORM LOW-OVERHEAD MONITORING TOOL FOR DISTRIBUTED SOFTWARE APPLCIATIONS

4.1 Introduction

Monitoring distributed software applications is a crucial component of practical software engineering [13, 14]. Companies in every major industry rely on custom distributed applications to sustain their business. To serve their users around the world these applications are expected to function 24x7. Managing such applications and maintaining their high availability requires close monitoring. Figure 4.1 gives an overview of a real distributed software application. The figure shows the production environment of our online travel shopping applications. The system is composed of various components, including a load balancer (F5), individual applications such as the BBIS hotel booking application, message queues (MQ), databases, the backend mainframe system (TPF), as well as an interface for terminal users, who are usually travel agents. The deployed software is written in different languages and is running on different operating systems. Communication between the various components uses different protocols—HTTP, TCP/IP, CORBA, and middleware messaging.

Monitoring distributed applications is challenging as it tries to satisfy two main conflicting goals. (1) On the one hand, software engineers want to collect data that is as detailed and comprehensive as possible, as having fine-grained runtime data for the entire system is useful in software analysis and maintenance tasks. (2) On the other hand, collecting any monitoring data is an overhead that can be very expensive,

as the monitoring tool consumes computing and communication resources of the very machines and networks it monitors. Any monitoring tool therefore tries to both maximize the utility of the data it collects and minimize the overhead it incurs.

In addition to these conflicting two main goals, existing monitoring tools also try to optimize other goals, such as ease of deployment and providing an integrated solution. These additional goals however conflict with the two main goals, as both ease of deployment and a highly integrated tool can lead to collecting data that is less useful while incurring a higher runtime overhead.

Specifically, we found that many existing monitoring tools utilize generic program instrumentation facilities to collect data. This is convenient in program deployment, as it can automatically instrument different kinds of applications. However it also limits the applicability of the approach to the platforms it supports. Practical business applications are often implemented on a wide mix of hardware and software platforms as well as communication protocols, which increases the risk that some part of the system is not supported by such a generic instrumentation approach. In addition, such a general approach also leads to bloat, as it may insert monitoring probes that are not needed for the specific application, which causes unnecessary overhead.

To provide an integrated solution, many existing monitoring tools also utilize their own custom communication techniques to propagate monitoring data from the monitored entities to repositories or analysis nodes. Such an integrated solution may be useful in some situations, but it can also lead to undesired overhead. Such custom communication techniques often run on the very nodes that are monitored and therefore consume resources that could otherwise be used by the monitored application. Examples of such custom communication techniques include hierarchies of filtering nodes that are co-located with the monitored application.

In this chapter we describe our in house built lightweight instrumentation and monitoring tool EI (Enterprise Instrumentation), which provides detailed monitoring data on a wide range of platforms and communication networks yet imposes very little overhead on the monitored machines. Specifically, EI provides an API that can be called from a variety of platforms, including Java, .Net, C++ and C environments. The API is simple and customized to avoid the overhead of more generic instrumentation approaches.

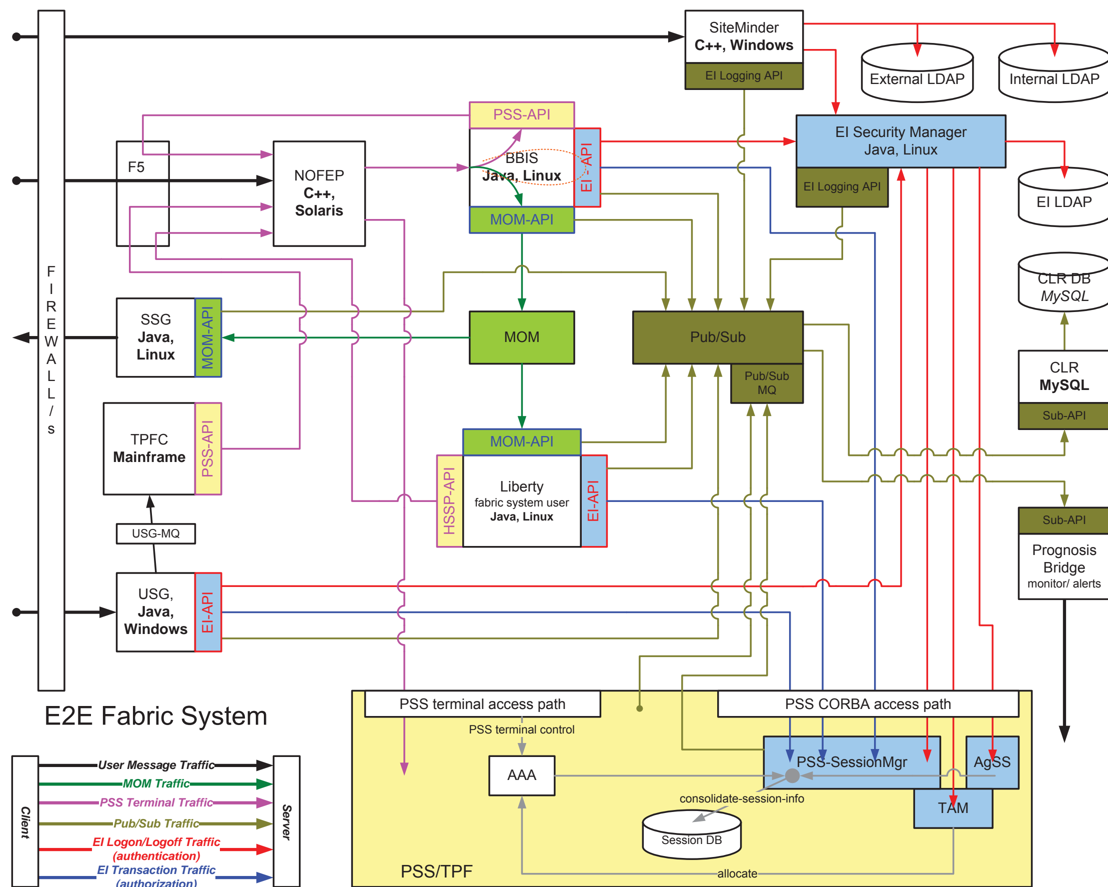


Figure 4.1. Design excerpt of a real distributed travel shopping application. The upper right shows key components of our EI monitoring tool, including various instances of the instrumentation library (EI API), the shared Pub/Sub communication medium, and the Consolidated Logging Repository (CLR)..

To propagate data from the monitored application to analysis nodes, EI leverages the well-known advantages of publish/subscribe systems and message queues, which separate processing logic (i.e., monitoring activities) from communication concerns. A key observation is that this separation of concerns allows us to separate the places at which the concerns are handled. This separation can preserve computing resources on the monitored machines for the application and thus yield better end-to-end performance.

The development of EI was motivated by a study of existing monitoring tools. We describe the result of this study for a well-known representative example tool, DynaTrace, and describe how EI compares with DynaTrace in a practical example scenario. DynaTrace was chosen as a comparison tool over other methods like NetLogger, Tealeaf and ClickTale because of it's use in the industry and proven reliability. Some of it's current customers are Macy's, Linkedin, Zappos and so on. DynaTrace was chosen as a viable solution because of the assurance that it would be supported and maintained as compared to exiting open source solution that rely on users to fix bugs and maintain the code. Specifically, we found that existing solutions often fail to support one or more of the platforms employed in a business application or a communication protocol. In addition to supporting a wide range of platforms, EI also imposes a very low overhead.

4.2 Background: DynaTrace

Several monitoring tools have been described in the literature. A representative example tool is DynaTrace, which is widely available. DynaTrace injects into the various software components of a distributed application light-weight agents. These Agents place sensors on component boundaries and on rich clients and browsers for full end-to-end monitoring.

After injection, these agents collect data at the OS and application level, such as payload, response times, class and method information, and SQL statements. DynaTrace can tag and track individual transactions across individual components and machines, which allows engineers to keep track of each incoming transaction and the path it traverses.

DynaTrace has its own communication mechanism, which consists of a hierarchy of data processing and communication nodes. That is, the agents forward sensor data to the DynaTrace data collector and server nodes for further analysis.

4.3 Overview of Enterprise Instrumentation (EI)

Figure 4.2 shows the four main components of the Enterprise Instrumentation (EI) tool.

1. The EI API is the entry point that collects data from the monitored application.
2. Pub/Sub is the communication medium EI uses to propagate data from API instances to the repository.
3. The main EI data sink is the Consolidated Logging Repository CLR.
4. The monitoring GUI provides a dashboard of EI and the monitored application.

The EI API can be added to an application by including in the application the EI libraries. The application can then use EI by using the libraries to create a EI logger (or collector) object and sending messages through the logger (or collector). EI sends these messages to their destination as configured by the logging mechanism, which could be via a file, a JMS Pub/Sub system, or other means.

- EI API

  The EI API is written in Java, C, and C++ to accommodate the different programming languages used in the production environment. It can be called from different platforms including .Net, Java, C and C++. It has the ability to

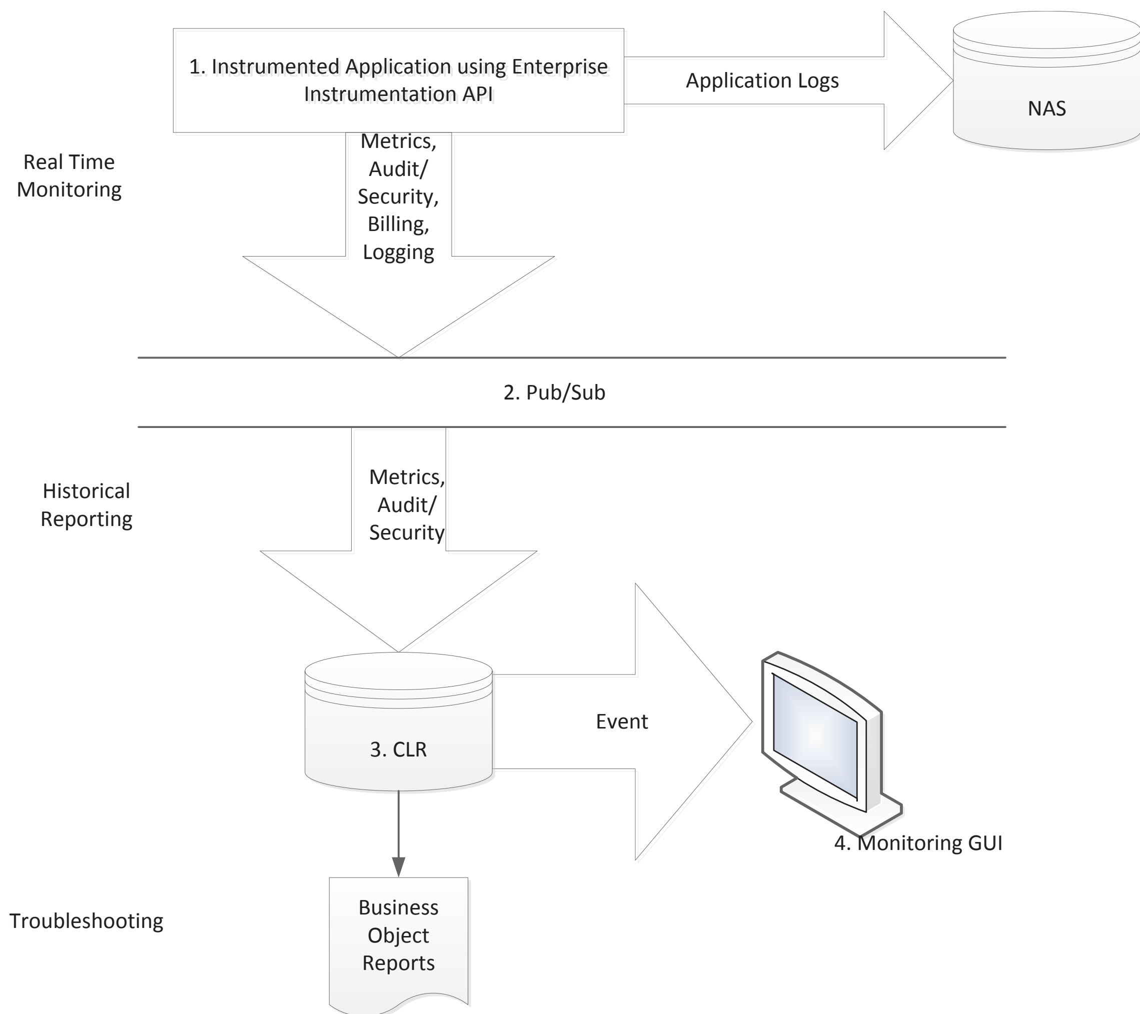


Figure 4.2. Overview of the main EI components - 1. EI API, 2. Pub/Sub, 3. CLR and 4. Monitoring GUI collecting various monitoring data, i.e., application metrics and audit, security, and billing data. NAS = network-attached storage; pub/sub = publish-subscribe..

track metrics for incoming transactions based on their unique identifiers. One of the major advantages of using the EI API to measure metrics, is that it can be used to gather information for not only application or client side or server utilization data but also data that is internal to the production system. For example the billing data metrics that are collected by the API are not directly

gathered from incoming transactions but EOD (End of Day) files that keep track of the ours of usage of a particular service/product by an airline and/or travel agent. More details are in section 4.6.3.

The API is used to collect required metrics for performance evaluation in a standard format, which in turn eases problem identification and resolution during testing and production use.

- CLR

  When teams adopt a common monitoring tool such as EI, the monitoring data such as application metrics, billing, security, and errors can be stored in a common repository. This repository can be used for reporting and forecasting.

- Pub/Sub

  This solution uses network attached storage (NAS) to store application logs. For redundancy purposes the application logs are stored in NAS along with being sent to CLR through pub/sub.

  EI uses a publish-subscribe (pub/sub) technique [15, 16] to send all instrumented application data to the repository. This middleware component is deployed on dedicated servers thereby eliminating loss of data due to a hardware failure on the application servers. The advantage of using a publish/subscribe technique is that it ensures that the application processing does not stop due to contentions or queue fill ups. This is due to the fact that the application is the publisher and hence simply sends out the various metrics to the appropriate topic/queue on the pub/sub server and then continues with its transaction processing. The subscriber associated with the topic is then responsible for writing the data to CLR.

- Monitoring GUI

EI also has a monitoring GUI, which serves as a dashboard for teams to ensure that all applications and systems are working as expected. As can be seen from Figure 4.2, the solution not only provides a capability to collect and view metrics while the application is live but also provides means to store and view historical data.

4.4 EI Log Types and Configuration

An application can generate five types of logging data with EI: application, audit, billing, metrics, and security. All of this data can be collected and retained in a centralized repository via the following steps.

1. Capture application measurements
2. Summarize measurements into time intervals (i.e., minute, hour, and day)
3. Consolidate data that is older than the retention period for that summary level

As new data is added to the CLR centralized repository and summarized each day, users can analyze and report the CLR data using Business Objects or ad-hoc SQL queries.

4.4.1 Log Types

- Application: Application-specific logging events, such as error messages and details about individual transactions can be generated at various levels of detail. For example, this fine-grained logging allows tagging and end-to-end tracking of individual transactions across applications and machines, which collects data that is very similar to the end-to-end transaction tracking provided in DynaTrace.
- Audit: Logging events that show application usage and critical changes in application state at various levels of detail for historical research into the usage

patterns of the application can also be generated. The Audit logging type is used to report application usage events that may be of interest during application audits. This information ranges from changes to critical data elements managed by an application to critical, infrequently used functions within an application.

Audit and Security logging are different flavors of the same logging type. The distinction between Security and Audit logging types provides an application with the opportunity to log these events at different levels as well as differentiate general audit events from those with security implications.

- Billing: Logging events that show application usage or critical changes in application state for charging customers for the use of the application can also be generated using the EI tool. The data elements generated by this type of logging is usually specific to the billing model of the application.
- Metrics: The normal usage for logging metrics is for XML-formatted summary metric records that are published every minute. For example, for a Java application these metrics include data on garbage collection and general JVM behavior.
- Security: Logging events that show application security functions such as authentications, authorizations for specific requests, and session destruction for historical research into the security aspects of the application can also be generated using the EI tool.

4.4.2 Configuration

All basic configuration elements for logging can be provided through either file- or LDAP-based configuration. The logging client API always expects to find

a file-based configuration to provide the connection information required to retrieve configuration settings from an LDAP environment.

File-based configuration can be implemented quickly and does not depend on external LDAP components. Our EI implementation uses log4J and log4cplus for the Java and C/C++ EI libraries respectively.

LDAP can be used to centralize the set of logging configuration settings for all application instances and modifying the logging levels for the different types at runtime. If the specified LDAP repository is unavailable at application instance startup, EI falls back to the properties of the file-based configuration.

4.5 EI Implementation

The EIAPI is added to an application by including the EIAPI jars and the third-party support libraries in a Java project, creating a logger or collector) object, and sending messages through the logger (or collector). These are sent as configured via a log4j mechanism to their destination - anything from a file to a JMS (Java Message Service) Pub/Sub system. When using EI, each application calls the EI API to send instrumentation data to the common repository. Applications can call the API with the transaction identifier before it is transformed, thereby keeping track of each transaction during the entire end-to-end processing, including message transformations.

EI benefits from using a mature communication channel (i.e., publish/subscribe) that is backed up by reliable message queues. This communication channel provides guarantees even in the case of hardware or software failures. That is, even if a node crashes, monitoring data is still queued in the communication channel and will be available once the node becomes available again.

The EI API also has a failover mechanism. When any appender error occurs, there is a risk of losing a record that could not be published. To avoid this, the API has implemented a failover logging mechanism. EI API provides its own ErrorHandler called EIAPIErrorHandler. This handler performs the following three operations.

- Logs to system.err when the corresponding appender fails for the first time. This output can be redirected to either the terminal screen or sent out as a file attachment in an email for notification purposes.
- Delegates the record that its appender was unable to send to a secondary appender.
- Informs any registered AppenderErrorListeners about the error. This way the application can extend EIAPIErrorHandler with any custom failover logic.

In our EI implementation we use a publish/subscribe tool from TIBCO called Enterprise Message Service (EMS), which implements Java Message Service (JMS). JMS provides asynchronous communication, which the EI API uses to connect the EI API with the publish/subscribe communication channel.

4.6 EI API Implementation Details

This section describes some of the metrics in detail that can be collected using the EI API. The metrics described here fall under the Application EI API metrics category and the last section talks about the Billing metric briefly. The examples in the following section are specific to Java applications but there are corresponding C and C++ EI API codes that are available.

4.6.1 Summary Customer Metrics (SCM)

Summary Customer Metrics (SCM) logging provides an application with the ability to log information about the operation of the application. This is used to

monitor the performance of the application and tune it to peak efficiency. Figure 4.3 depicts the logical view of how SCM can be collected using the EI API.

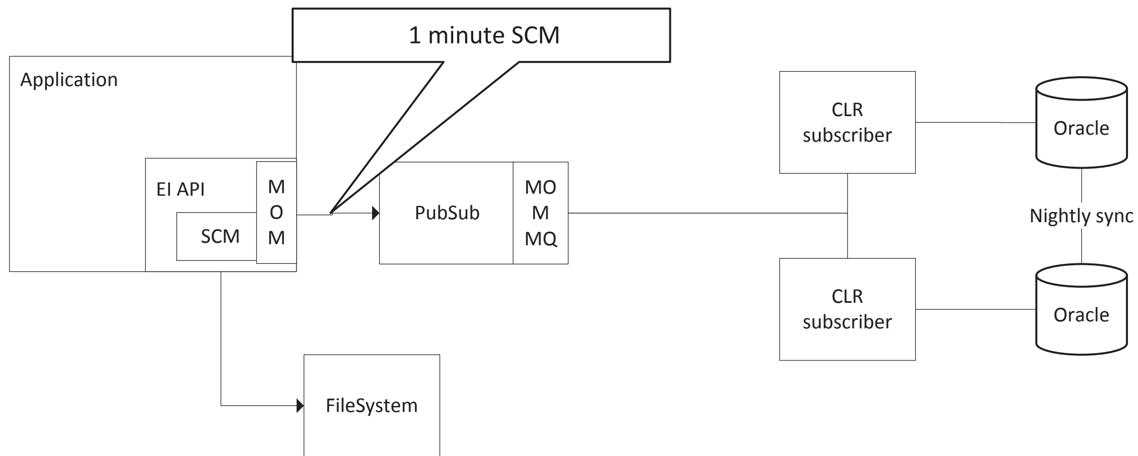


Figure 4.3. Logical View of Summary Customer Metrics Collected Using EI API.

Information can be logged in a way such that the structure of metrics within a transaction and its larger context can be preserved. If an incoming transaction spawns multiple sub-transactions, the metrics for each of these transactions will be stored in hierarchical way to preserve the parent-children structure. The SCM message structure enables flexibility by allowing the data collection point to be defined as any type of entity rather than just a service or a user.

The log header contains all the information that is common across all records in the message. The authentication identifier (AthID) and serverMethodName are

added at the data collection point for every transaction as they can change over the processing duration of the transaction. Any number of log entries may be in a record, and log entries may have sub entries, which would indicate a set of metrics associated by a parent/child relationship. An example might be a service associated with a subservice call. All log entries are identified with a minimum a name and type, but additional attributes may be added as well to further define the data collection point.

A SummaryCustomerMetricsLogger is created and all transaction activity is processed through it. The logger is stateless so that it can be employed in a multi-threaded environment. SummaryCustomerMetricsLogger logger = new SummaryCustomerMetricsLogger(); Once a logger is created, transactions may be logged to it. Transactions are begun and named, and then actions are performed through the logger on the transaction. Log log = logger.begin(transactionName); A child transaction is begun be referencing the parent log, a name, and a child type. Log log = logger.begin(log, transactionName, transactionType);

Values are placed into transactions to record (and update) performance metrics, critical events, and any other statistical value. logger.addStatistic(log, keyName, value); Custom statistics can be created and applied to a key as well. StatisticsContainer container = new StatisticsContainer(keyName, count, total, minimum, maximum, sumOfSquares); logger.addStatistic(log, keyName, container);

4.6.1.1 Transaction Details

The metrics for a transaction are collected and placed into a TransactionDetails object. Transactions require setting the identity of the application performing logging. The identity consists of three parts: the Node, the Application Name, and the Instance. The Application Identity needs to uniquely name the running application. The Instance identity identifies the unique instance. If there is more than one

instance of the application running on the same machine, then the Instance Identity differentiates those instances. They need to be non-empty strings. The Node Identity is not usually set. The node identity defines the machine on which the application runs. By default the Node Identity is the host's fully qualified domain name.

A transaction record is created when transactions are logged. Each of these transaction records is sent to the TransactionSummaryAppender. It then produces a summary record of all of the log events that have the same origin. The summary record is generated as a stamped XML document and the following is a sample of the kind of document that gets created:

```
2012-01-23 12:34:56.789 TransactionSummaryAppender-PublishThread
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<log>
  <version>1.3.2</version>
  <timestamp>2012-01-12 12:34:56.789 -0500</timestamp>
  <node-id>MYCOMPUTER</node-id>
  <app-id>MyApp</app-id>
  <instance-id>MyInst</instance-id>
  <transaction-summary level="20000">
    <statistic key="TransactionName/END" count="1" />
    <statistic key=" TransactionName/DURATION" count="1" total="78"
              average="78" min="78" max="78" stddev="0.0"
                 sumSquares="6084" />
    <statistic key=" TransactionName/BEGIN" count="1"></statistic>
  </transaction-summary>
</log>
```

The timestamp and thread name are delivered according to the log4j settings. The message itself is generated without line breaks. It is represented here with line breaks for readability. The transaction-summary tag holds all of the statistics. Each event such as the beginning or ending of a transaction is represented as a statistic. The same statistic is counted as well as totaled and averaged. The minimum, maximum, standard deviation, and sum-of-squares are also computed for each event statistic that is being collected. A few of the architectural goals for this particular metric type are as follows:

- Collects metrics broken out by service and customer
- Allows applications that are already collecting metrics data to simply report that data
- Provides standard metrics collection
  - count, total, average, min, max and standard deviation
- Support hierarchical transactions
  - Nested transactions,
  - attributes and
  - updating values at various levels

4.6.2 JVM Metrics

The objective is to develop an Appender that collects the JVM metrics. Java has a set of platform MBeans (Managed Beans) for monitoring and management of the Java virtual machine, which can be used to collect this information. It needs to collect these metrics in a configurable interval and then summarize them, so it can be logged as a Metrics Record. It can also be easily integrated with Application Metrics. Figure 4.4 depicts the logical view of how an application can use the EI API to collect JVM data.

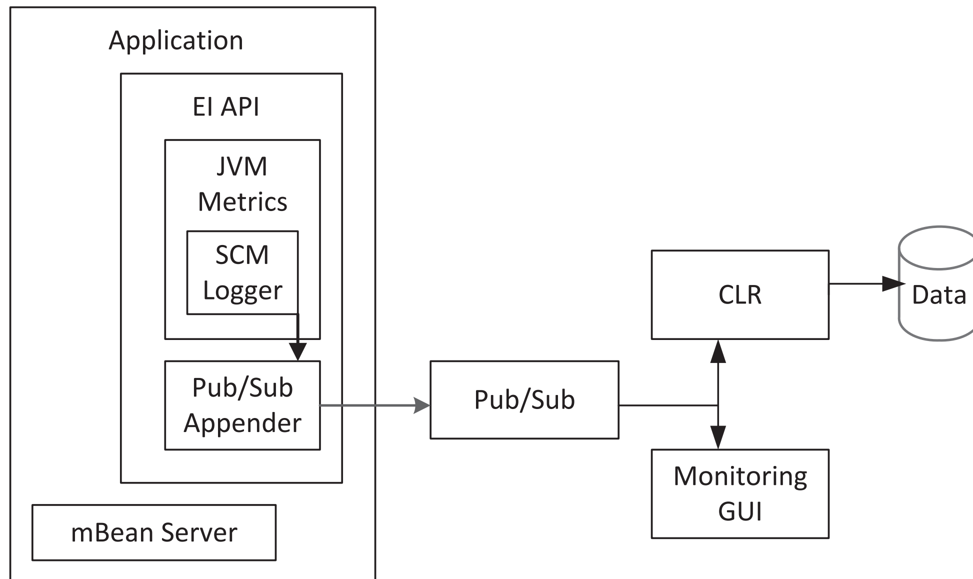


Figure 4.4. Logical View of Application Using EI API to Collect JVM Data.

MBean mappings are much more versatile because JMX poller resolves those at runtime. Basically, an MBean mapping is an expression that can contain static and dynamic parts. A static part is any constant text you want. A dynamic part on the other hand is a character sequence contained in a pair of special brackets which at runtime will result in a different value for various MBeans. An example of a dynamic expression part would be: "$objectName.name". Every mapping expression is resolved to a static text (even its dynamic parts) before being logged as an SCM record. Both MBean expressions and attribute expressions are placed in the generated SCM records. Sometimes it might be useful to use a custom name for them instead of a complicated expression as a metric name. For example, it might be useful to log "java.lang:type=Memory.HeapMemoryUsage.Max" as just "MEM.MaxHeapUsage".

The EI API allows these customizations for name mappings of both MBeans and their attributes.

4.6.2.1 Polling Interval

The JMX Poller allows for a granular configuration of the polling interval. It is required to set a global interval for the Poller but intervals can also be set on a per MBean or even per attribute basis. All interval configurations may be combined in a single configuration. This results in a tree of intervals. Some nodes in the tree may be empty. Others will contain a valid interval. Obviously, every attribute will get polled at only one interval. To determine this interval, EI API uses the "deepest node is the most important" approach. For example, this tree of intervals:

- Global interval (60)
  - First MBean (30)
    - First attribute (15)
    - Second attribute
  - Second MBean
    - First attribute (20)
    - Second attribute

will cause the EI API to use the following polling intervals:

1. First MBean.First attribute - every 15 seconds
2. First MBean.Second attribute - every 30 seconds
3. Second MBean.First attribute - every 20 seconds
4. Second MBean.Second attribute - every 60 seconds

The default configuration is to poll some default JVM metrics every few seconds. The poller can be configured programmatically as follows:

```java
// Most of configuration is built using the builder pattern.
// Every configuration object is immutable once created
MBean mem = new MBean.Builder("MEMORY", new
    ObjectName("java.lang:type=Memory"))
        .addAttributes("ObjectPendingFinalizationCount")
        .addAttributes(new Attribute("Verbose", "V"))
        .setMapping("MEM")
        .setInterval(30)
        .build();
MBean pool = new MBean.Builder("MEMORY_POOL ",
        new ObjectName("java.lang:type=MemoryPool,name=*"))
        .addAttributes(new Attribute("PeakUsage.Max", "PeakUsage.Max", 15))
        .build();
JMXPollerConfiguration configuration = new JMXPollerConfiguration.Builder()
        .setEnabled(true)
        .setInterval(60)
        .addMBean(mem).addMBean(pool)
        .build();
LogManager.instance().getJMXPoller().configure(configuration);
```

A list of the JVM metrics that can be collected using the EI API are as follows:

4.6.3 Billing

The Billing logging type is used to report application usage events for which customers may be charged usage fees for the application. The data elements generated by this type of logging are usually specific to the billing model of the application. These include, but are not limited to, service information and information about the

Table 4.1. JVM Metrics Collected Using EI API

mBean Name	**Attribute Name**	**Interval [s]**
java.lang:type=Memory	HeapMemoryUsage.Init	60
java.lang:type=Memory	HeapMemoryUsage.Used	60
java.lang:type=Memory	HeapMemoryUsage.Committed	60
java.lang:type=Memory	HeapMemoryUsage.Max	60
java.lang:type=Memory	NonHeapMemoryUsage.Init	60
java.lang:type=Memory	NonHeapMemoryUsage.Used	60
java.lang:type=Memory	NonHeapMemoryUsage.Committed	60
java.lang:type=Memory	NonHeapMemoryUsage.Max	60
java.lang:type=Memory	ObjectPendingFinalizationCount	60
java.lang:type=MemoryPool,name=*	CollectionUsage.Init	60
java.lang:type=MemoryPool,name=*	CollectionUsage.Used	60
java.lang:type=MemoryPool,name=*	CollectionUsage.Committed	60
java.lang:type=MemoryPool,name=*	CollectionUsage.Max	60
java.lang:type=MemoryPool,name=*	PeakUsage.Init	60
java.lang:type=MemoryPool,name=*	PeakUsage.Used	60
java.lang:type=MemoryPool,name=*	PeakUsage.Committed	60
java.lang:type=MemoryPool,name=*	PeakUsage.Max	60
java.lang:type=MemoryPool,name=*	Usage.Init	60
java.lang:type=MemoryPool,name=*	Usage.Used	60
java.lang:type=MemoryPool,name=*	Usage.Committed	60
java.lang:type=MemoryPool,name=*	Usage.Committed	60
java.lang:type=GarbageCollector,name=*	CollectionCount	60
java.lang:type=GarbageCollector,name=*	CollectionTime	60
java.lang:type=ClassLoading	LoadedClassCount	60
java.lang:type=ClassLoading	TotalLoadedClassCount	60

application invoking a billable service. Figure 4.5 shows the logical view of how the billing data is collected for an application. Billing provides metrics categorized by service for a customer. To identify the customer, the customer PCC (Pseudo City Code) can be obtained from a user supplied session object. Billing logging is currently used for Open Systems billing for non-TPF requests coming in through the web. Billing is available using Summary Customer Metrics.

The EIAPI has added Customer Attributes to Billing. There are two ways to add these attributes. In the Service Builder method, note that we support adding custom attributes on the Billing Client level. Adding attributes on Service level is

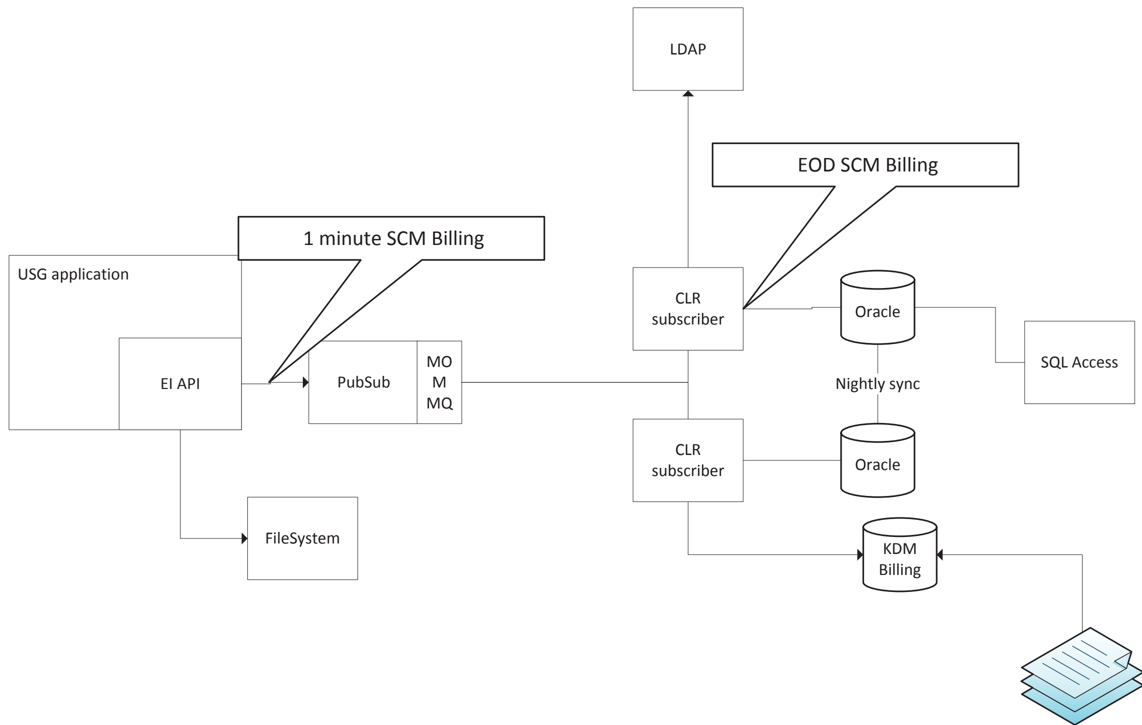


Figure 4.5. Logical View of Application Collecting Billing Information.

not yet supported. This is an example of adding attributes using UserData plus the Billing API.

```
/* Create new user data with custom attributes */
UserData userData = new UserData(new
    DefaultSessionImpl("ID","Type",Level.DEBUG,Level.DEBUG,Level.DEBUG,Level.DEBUG));
userData.addAdditionalAttribute("ApplicationId","App_ID1");
userData.addAdditionalAttribute("CustomerID",String.valueOf(new
    Long(1)));
userData.addAdditionalAttribute("UserID",String.valueOf(new
    Integer(3)));
```

```
    userData.addAdditionalAttribute("ServiceID",String.valueOf(new
        CustomServiceId()));
    billingsCollector.appendService("SomeService","1.0", userData);
```

This is an example of adding billing attributes at a service level:

```
/* Create attributes with the service builder */
BillingServiceUsageBuilder serviceBuilder =
  new BillingServiceUsageBuilder("S1Name", "S1Version")
               .begin(userData1).usingCpuFor(2L, 4L)
               .with("IPAddress", "192.168.1.0")
               .with("User", "BritishAirways")
               .commit()
               .begin(userData2)
               .with("IPAddress", "192.168.222.111")
               .with("User", "AirCanada")
               .usingCpuFor(4L, 6L)
               .commit();
       BillingsCollector billingsCollector = collectors.get();
       billingsCollector.appendService(serviceBuilder.build());
```

4.7 Experience in the Sabre Production Environment

Once we have a uniform way of collecting data from every application it is easy to display that information and monitor the performance of an application during performance testing and production. Since all monitoring data are stored in a database, it is easy to view historical data for applications at the server level and then trend the data as seen for our shopping application example in Figure 4.6.

The graph shows data for an entire year, from September 2010 to September 2011. It indicates the number of processes, of both the system and the application running on a particular server (swshlc113). The numbers below the graph indicate the average, minimum, and maximum number of system processes currently (at the time of the snapshot) running and blocked. The system total indicates the number of processes running on a particular day of the month. This kind of data is important for identifying trends for a given system. Figure 4.6 shows that the number of system processes running at any given point of time are pretty constant at 2.2K.At regular intervals approximately every 3 months the number of processes goes down which could indicate maintenance window for the server. The time interval of interest here is when the number suddenly goes up to more than 6k. This is the time frame that needs to be investigated. The increase could be contributed to a new application/process being installed on the server could be the result of a malfunctioning process. The instant availability of data to the application as well as the operations team facilitates in finding out root cause of such discrepancies in the performance trends.

The different types of data logged by EI help in identifying trends and patterns for potential problems. It can also be used in problem localization, for example, in classifying a problem as a user error or an application problem.

Thresholds and alerts can be setup on these performance metrics for better visibility. This is especially true for client side and resource metrics. Alerts set on response times and CPU usage on the shopping application behavior shown in Figures 4.7 and 4.8 alert our operations and application teams of potential performance issues. For example Figure 4.7 is showing an increase in the application elapsed times as the incoming transaction per second (TPS) rate is increasing. Similarly Figure 4.8 is showing that the user CPU time is constantly increasing till it reaches a 100while the CPU idle usage is decreasing as time progresses. Both these characteristics are

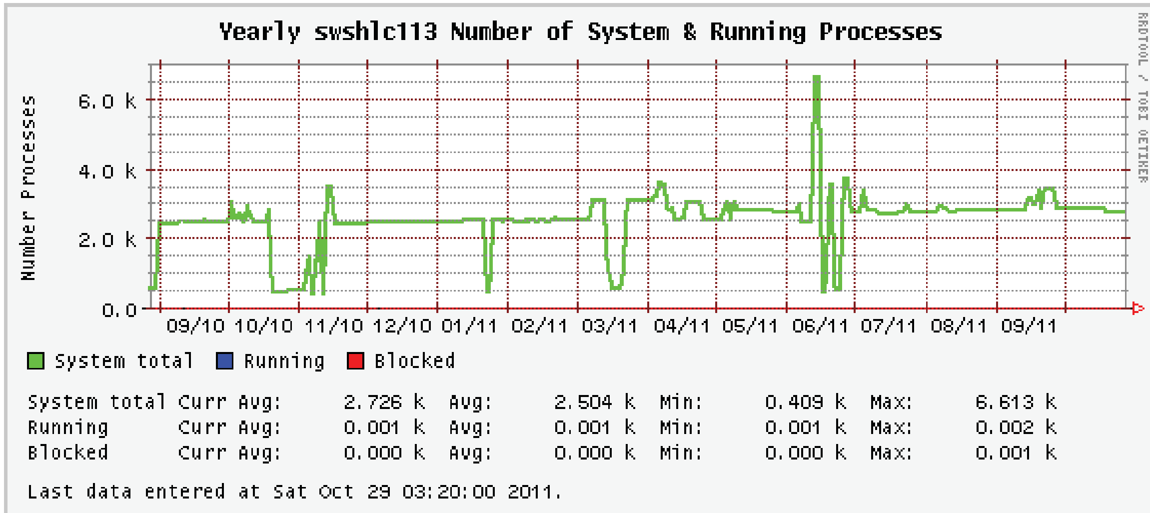


Figure 4.6. Yearly historical process data on a given server collected in CLR to identify performance trends. The graph shows the total number of processes running on the server on a particular day for each month from September 2010 to September 2011..

a cause of concern. It indicates that the application is not performing as expected and is using up the entire CPU to process the incoming traffic rate. Alerts could be set up for these major metrics - elapsed/processing times and CPU usage and the teams - application as well as operations will be instantly notified when a particular threshold for these metrics is reached. Threshold levels for metrics that affect the client can be set keeping in mind user defined SLAs where as thresholds for server side metrics like CPU can be determined using operational standards.

The screen-shots of Figures 4.9 and 4.10 are examples of the kinds of metrics EI can collect. Fine-grained metrics are needed to evaluate application performance. For example, Figure 4.9 displays information about the application process (uptime, server name, port, release number, etc.), operating system resource information (free memory, used virtual memory, system load, etc.), and client side metrics (total requests sent to the application, response times, failed transactions, etc.). During pro-

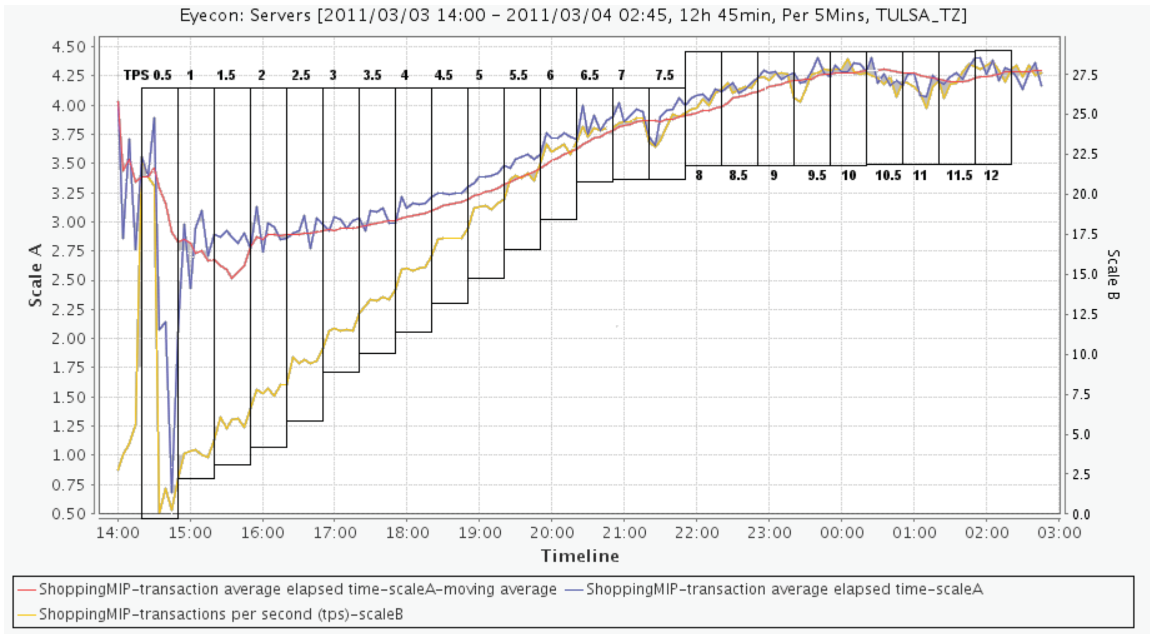


Figure 4.7. End-to-end system response time increases as TPS is increasing in the shopping application of Figure 4.1. .

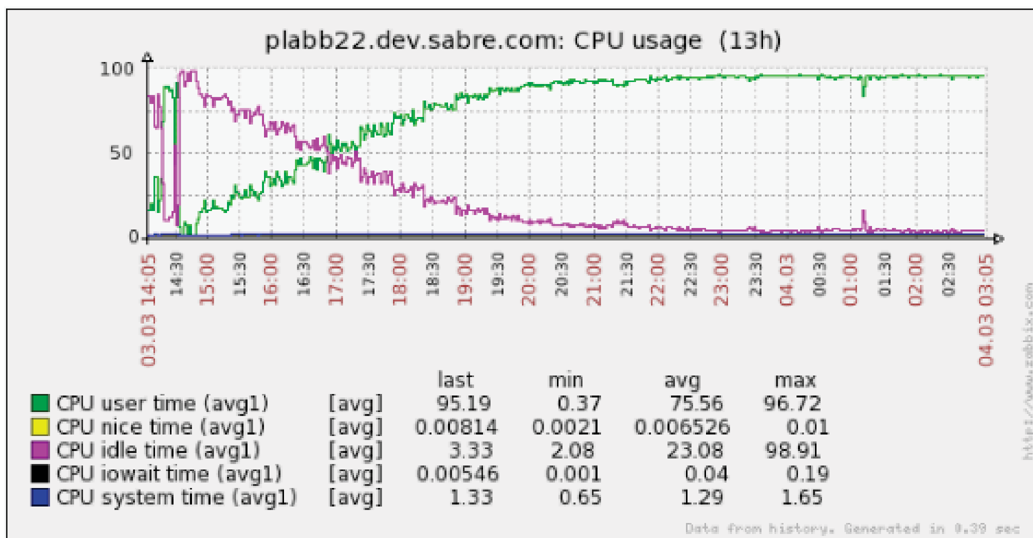
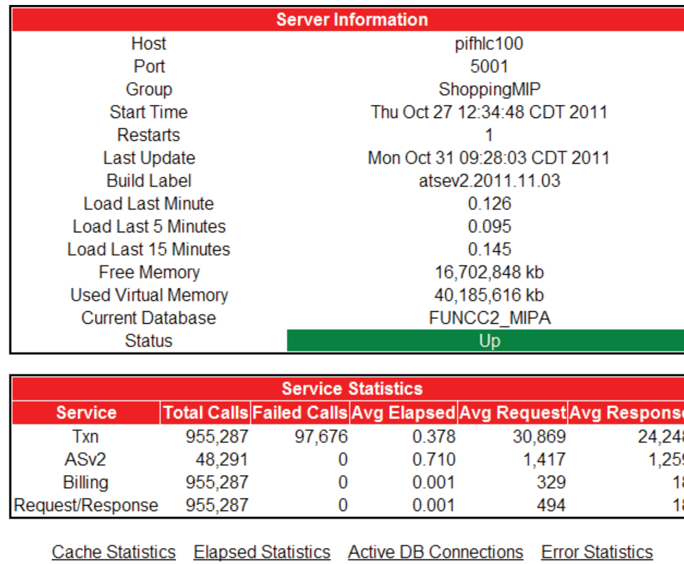


Figure 4.8. CPU usage on one of the two shopping servers. The corresponding graph of the second server is very similar. CPU usage is increasing with the TPS increase of Figure 4.7..



Server Information	
Host	pifhlc100
Port	5001
Group	ShoppingMIP
Start Time	Thu Oct 27 12:34:48 CDT 2011
Restarts	1
Last Update	Mon Oct 31 09:28:03 CDT 2011
Build Label	atsev2.2011.11.03
Load Last Minute	0.126
Load Last 5 Minutes	0.095
Load Last 15 Minutes	0.145
Free Memory	16,702,848 kb
Used Virtual Memory	40,185,616 kb
Current Database	FUNCC2_MIPA
Status	Up

Service Statistics					
Service	Total Calls	Failed Calls	Avg Elapsed	Avg Request	Avg Response
Txn	955,287	97,676	0.378	30,869	24,248
ASv2	48,291	0	0.710	1,417	1,259
Billing	955,287	0	0.001	329	18
Request/Response	955,287	0	0.001	494	18

Cache Statistics Elapsed Statistics Active DB Connections Error Statistics

Figure 4.9. Fine-grained performance data collected in CLR: Response times, elapsed times, up-times, and failed transactions..

duction incidents when an operations team member that is not knowledgeable about the workings of the application needs to inform the on-call and/or development team, this kind of information comes in handy. Its shows all the major application deployment related details as well as processing and client side metrics. The operations team can quickly access this data and get the right team on call to investigate issues.

Similarly, Figure 4.10 shows the types of errors that have occurred during the application runtime. Such an error report makes troubleshooting easier. For example, error code 5011 and the description NO_FARE_FOR_CLASS_USED indicates that fares are missing for a particular passenger booking class. The issue can be fixed by updating the appropriate fare table.

4.8 Comparison With a Third-Party Monitoring Tool: DynaTrace

We compared our EI monitoring tool with DynaTrace in March 2011. The tests were run on part of the system as shown in figure 4.1. The BBIS hotel shopping

Error statistics

Error Code	Count
5053 - NO_COMBINABLE_FARES_FOR_CLASS	44131
5167 - UNABLE_TO_PRICE_ISSUE_SEPARATE_TICKETS	4
9501 - LMT_ISSUE_SEP_TKTS_EXCEED_NUM_DEPT_ARR	21
9502 - LMT_ISSUE_SEP_TKTS_INTL_SURFACE_RESTR	21
6002 - INVALID_INPUT	5
5011 - NO_FARE_FOR_CLASS_USED	45709
9997 - REQUEST_TIMEOUT	463
5005 - PRICING_REST_BY_GOV	144
5033 - NO_RULES_FOR_PSGR_TYPE_OR_CLASS	138
6050 - NO_FLIGHTS_FOUND	38
6119 - NO_PRIVATE_FARES_VALID_FOR_PASSENGER	10993
6120 - NO_PUBLIC_FARES_VALID_FOR_PASSENGER	5895
9006 - null	47
6124 - NO_CORPORATE_NEG_FARES_EXISTS	556
SUMMARY	
	108165

Figure 4.10. Example fine-grained error statistics collected in CLR..

application was taken as the pilot application for these comparisons and was setup in the test environment which is part of the end to end system shown in Figure 4.1.

Both instrumentation techniques were used to measure the application side method call times and page loads. The DynaTrace server, client and agents were installed on the BBIS application server. To measure the EI API times , a version of BBIS that makes use of the API was used. Simple use cases as shown in Table 4.2 and Table 4.3. Figure 4.11 shows the logical view of activity flow that was followed and the method calls and page loads that were measured. The logical view of the actions that were tested are shown in Figure 4.11 We focus our comparison on two

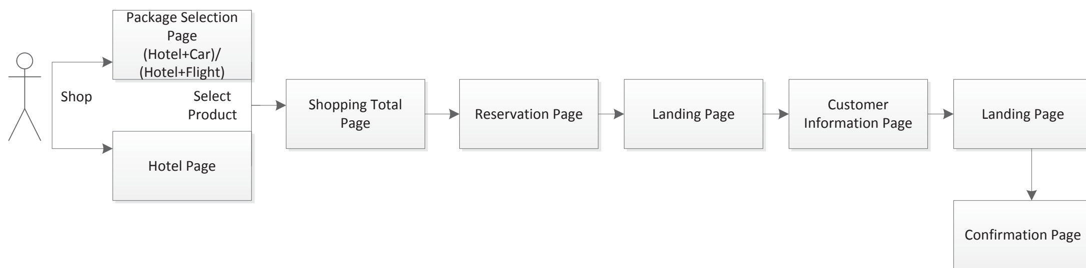


Figure 4.11. Logical View of User Actions Done During Testing.

key aspects, the ability of the tool to work with different communication protocols and platforms and the runtime overhead.

4.8.1 Cross-Platform Functionality

Solutions such as DynaTrace are typically easy to install and use, as long as they are used with a limited set of standard platforms and languages such as Java, .Net, C/C++, and Cobol. If the system to be monitored contains software running on specialized platforms such as legacy TPF mainframe systems then using such solutions becomes very complex or even impossible.

For example, DynaTrace provides an extension point via an instrumentation API. However, this API is not mature enough to, for example, defer the instrumentation start call, which is needed when significant processing must occur before calling Start(). For example, our travel application in several cases first parses the header of an incoming request to determine which application it needs to be sent to. However DynaTrace does not allow calling Start() after performing such parsing tasks.

DynaTrace's support for different communication protocols is also limited, as it cannot track individual transactions if the message or transaction is transformed during processing. However such transformations are performed routinely by existing systems. For example, an incoming HTTP request could be transformed into an XML format and then fed to the next processing unit, and transformed back to HTTP in the reply message. In such cases, the extra transaction tag DynaTrace adds to the header is lost, together with all associated instrumentation data. The third-party components performing such transformations are not aware of such additional tags and therefore drop them during transformations.

4.8.2 Runtime Overhead

In this section we will compare the response times for the different method calls as well as the time it takes to render the web pages between DynaTrace and the EI API. The reason to choose response time for comparison is that this the metric that is most affected by changing the technique in how application data is being accumulated and archived in the repository. Both these methods - DynaTrace as well as the EI API have a very small footprint and therefore a minimal overhead is added to the OS resource utilization metrics.

Table 4.2. Increase in response time for method calls when using DynaTrace compared to our EI tool.

Test case	**Request**	**DynaTrace [s]**	**EI [s]**	**Diff**
Shopping	url_1	4.62	3.24	-42%
BookRoomRun	url_2	4.09	3.10	-32%
BookRoomRun	url_3	10.25	9.24	-11%
Shopping	url_4	6.47	6.10	-6%
Shopping	url_5	6.19	5.89	-5%
BookRoomRun	url_6	8.58	8.64	1%

As will be seen by the data one major drawback is the additional processing overhead associated with DynaTrace. Using a hotel shopping application several tests were run to determine the overhead added due to DynaTrace as compared to the in house built enterprise solution. DynaTrace was configured to collect the same metrics in the same granularity and frequency as the in house built instrumentation technique.

Tables 4.2 and 4.3 show the difference in the response times for specific method calls as well as the times taken to load particular customer pages. The comparison was done on two levels - gathering metrics for page calls as a customer browses through the application to shop for and then book a hotel room; and the different method

calls that were made during this process. In both cases, for a majority of the time DynaTrace added an overhead which resulted in an increase in the response times as compared to EI.

The test cases in Table 4.2 refers to the activities/method calls being performed whereas the Request column refers to the different page URLs that are being used to process those activities. For simplicity purposes, the pages have been names URL_1, URL_2 and so on. The DynaTrace and EI columns show the time taken by the method calls when using these two different instrumentation techniques. Similarly in Table 4.3, Test Case refers to the kind of activity being performed, Transaction refers to the various web pages that have to be loaded based on a particular activity and the DynaTrace and EI columns refer to the time taken to load the different pages when these two instrumentation techniques are used.

Table 4.3. Increase in response time when delivering pages, using DynaTrace as compared to our EI tool.

Test case	**Transaction**	**DynaTrace [s]**	**EI [s]**	**Diff**
BookRoomRun	Pkg selection	4.09	3.10	-32%
Shopping	Shopping Total	27.68	24.25	-14%
BookRoomRun	Confirmation	6.68	5.90	-13%
BookRoomRun	Customer Info	6.50	6.04	-8%
Shopping	Product Page	6.47	6.10	-6%
Shopping	Landing Page 1	12.37	11.76	-5%
BookRoomRun	Landing Page 2	23.86	23.44	-2%
BookRoomRun	Reserv. Total	52.44	52.83	1%
BookRoomRun	Product Select	9.04	10.26	12%

The higher overhead of DynaTrace can presumably be attributed to the fact that there is a two step extra processing required, where the sensors collect the data at every hop and then send it to the agents which in turn forward it to the metrics

database. These additional steps, an overhead, that cause a slight increase in the response times. By using the EI API the metric data push is part of the regular application processing, a one step process. Once the data has been pushed on to the Pub/Sub topic, it is not the functionality of the application or any of it's components to push it to the data repository. This prevents additional time to be recorded to the end to end response time.

4.9 Related Work

Monitoring of computer systems is essential in not only keeping track of any potential bottlenecks that might occur while live but also in establishing traffic trends and patterns. Several articles and research work have focused on varying forms of monitoring systems. [17] discusses a system management system called PARMON to monitor large homogenous and heterogeneous clusters of workstations. It monitors system utilization data and presents it in a graphical view for better understanding. [18] is another framework called Pulsar that monitors UNIX hardware resources. The frequency of monitoring can be configured. A set of Unix commands are sent at the configured interval to collect resource utilization information. X window is used to send out alerts/alarms when utilization crosses a particular threshold. Another flexible monitoring framework called Provider-side Metric Collector (MeCo) is described in [19]. The MeCo component is deployed within the provider host and gathers data pertaining to the service usage from an observed server platform. The data is propagated to a Measurement Service for evaluation of the relevant SLAs. Deployment of the MeCo Framework comprises positioning relevant MeCo objects within the service platform, configuring the MOM subsystem and activating the Measurement Service.[20] talks about a framework called ZeliGrid that can be used with Grid applications and works using the resource constraint values to maintain system avail-

ability for applications with specific performance requirements. The author of [21] describes a set of tools called Falcon that support three distinct tasks - (1)captures performance data/information at the application level, (2)analyzes this performance data either by using programs or users and (3) takes steering decisions or actions depending on the data analysis. The framework achieves low latency by only capturing application attributes that are required for program steering. Sensors are used to collect the application data. The Enterprise Instrumentation tool described in this chapter relates to these papers in terms of the motivation. All the above mentioned monitoring tools intend to capture performance data to ensure that the resources and applications are working without incident. EI and the frameworks described in [19] and [21] have application monitoring as the common goal. EI is an API based approach whereas [21] uses sensors.

The Grid Monitoring Architecture GMA is a very general reference architecture [22]. EI and all related tools fit this architecture. EI can be further classified as a level-2 monitoring system, consisting of data sensors, producers, re-publishers (a communication channel), and consumers [23]. In the following we discuss closely related monitoring systems, which are either also level-2 systems or level-3 systems (where the latter have a hierarchy of republishing nodes).

[24] talks about a monitoring framework that takes the programming code as an input and then decides where to add monitoring and measurement. A specification language is used to enter questions in standard format and then the framework decides where to add instrumentation in the code. This automation tool is useful for functional testing where programmers instead of using profilers can use this framework to automate the process of adding instrumentation points. This is different from the EI tool described in the chapter where the API helps collect performance data not only on an application level but also client side and billing information. The authors

of [25] and [26] talk about monitoring in the Cloud. In the cloud environment where there are numerous layers' services and dependencies, it is difficult to keep track of SLAs. [25] proposes describing SLAs and their dependencies in term of a Web Service Modeling Ontology to build a knowledgebase whereas [26] describes a multi layer monitoring system that can be reconfigured in terms of the monitoring interval as well as parameters. The EI tool differs from these frameworks in terms of what it can instrument and measure. Customized paramerts as well as instrumentation points outside the application code can be defined to be measured.

GridEye provides a SOA-based architecture that conforms to the GMA Grid Monitoring Architecture [27]. Sensors deployed across the system collect information in real time which is stored by the producer component and stored in a local database. This information is used by consumers and a performance forecasting component.

SCALEA-G utilizes two sets of sensors, one for application data and one for system metrics [28]. The data is stored in buffers and a XML-based instrumentation request language is used to for interaction between the information requester and the instrumentation service.

NetLogger collects event logs across a distributed system using an API similar to EI [29]. It also has an interface to monitor these events and tools for visualizing log data. However NetLogger sends all logs to the same port of the same machine, whereas EI separates monitoring from communication concerns and leverages a mature publish/subscribe communication channel.

GLIMPSE is another publish-subscribe based architecture for collecting system information [30]. It is a model-driven approach where events are defined by consumers and then translated to a GLIMPSE specific event language using model driven transformations.

Zhang et al. performed a detailed comparison of the related monitoring systems Globus Monitoring and Distributed System (MDS), Relational-Grid Monitoring Architecture (R-GMA), and Hawkeye [31]. However, their work analyzes the performance of individual components on a synthetic workload, whereas we compared end-to-end performance in a practical example scenario. R-GMA makes use of a publisher-subscriber model and Hawkeye uses agents on systems to collect data. After running several tests to evaluate the scalability of the different components in these frameworks, the authors observed that each of these was comparable in terms of scalability. Caching helps with speeding up the data collection.

CHAPTER 5

TEST HARNESS AND SCRIPT DESIGN PRINCIPLES FOR AUTOMATED TESTING

5.1 Introduction

Retaining customer loyalty in today's competitive world is directly related to revenue and hence a huge priority for any industry. This in turn signifies the importance of deploying reliable and stable applications that in no way cause any kind of service disruption. To ensure this is the case when applications are run in the production environment, good functional as well as non functional testing becomes important. Manual testing is still very much alive and is used in some shape or form to run functional tests across every industry. Manual testing is a good way test the front end web or GUI user interface.

But for non functional testing where in most cases hundreds or even thousands of transactions per second need to be simulated for applications with no GUI or web interface, using manual testers is not an option. Performance (non functional) testing also requires emulating production like scenarios and testing every application that is part of the end-to-end data flow path, instead of just the user interface. In a computing environment each user request passes through several software components, including user-facing servers, load-balancers, middle-ware systems, database servers, and back-end applications. Any of these components could modify a user request in a subtle way. For example an incoming request that goes through an enterprise service bus or a message based middleware system gets transformed when the message header gets modified by adding application specific parameters to match its native format.

There are several commercially available and open source load drivers that are widely used for purposes of performance testing such as Load Runner, SoapUI, SOATest and JMeter. Load Runner and JMeter enable record and playback of web user transactions. SoapUI and SOATest enable the creation of test scripts that use SOAP XMLs. But an organization might have its own customized format of incoming requests that may not be supported by existing tools or may be using protocols like CORBA that may not be supported by current test tools. Recording production logs for these kinds of situations and playing them back against the applications in a test environment is not possible with the currently available test tools. In a complex computing environment where input messages are transformed and may use a different protocol to bridge the communication between applications, customized test tools and scripts have to be developed from scratch. Since load drivers and test scripts play such a vital part in performance testing, they have to be created carefully.

The same principles that are followed to develop a good piece of software need to be used to create test scripts. Test scripts are a vital part of the success of a performance testing project. They drive the traffic, they simulate the test cases - both destructive as well as happy path scenarios and they record the client side experience metrics like response times. Just like any other software, test scripts should have the following qualities:

1. Readability: It is important to have readable scripts that incorporate clean logic and are easy to understand. It makes it easier for a new tester to take over and work with them.
2. Extensibility: This allows for script changes to be made and more test scenarios to be added in order to assess an application's performance.

3. User friendliness: Test scripts do not need to have a GUI interface to run, but even when a command line is being utilized it should be simple and easy to understand and use.
4. Efficiency: For many cases in performance testing it is required to simulate thousands of concurrent users or transactions at the same time, which might require running multiple instances of the script. Therefore it is important for these scripts to be efficient and light weight and use minimal server resources.
5. Reusability: A production environment is usually made up of numerous applications and components that work together to process user requests. This leads to the importance of reusability - to be able to reuse pieces of a test script code to build scripts for other applications.
6. Atomicity: Test scripts need to have a modular design, where each module is dedicated to a implementing a specific logic or test case. This speeds up the modification and enhancement process.
7. Correctness: It is essential that the test scripts are able to simulate all documented test scenarios correctly in order to ensure successful performance testing.

In addition to satisfying the above mentioned requirements and generating traffic, a test script needs to perform certain other functionalities. Just like a manual tester records the software behavior corresponding to each click of the button or action, the test script needs to do the same. It needs to keep track of user side metrics like errors percentage, response times and throughput. If need be it should also be capable of recording and playing back traffic. In addition to this there are some pre and post test activities like setting up the environment and creating a report after the test is complete, that should also be automated in order to successfully create a complete framework for testers. The next section will describe the factors that should

be considered when building such a complete test harness along with the design for developing test scripts.

5.2 Test Harness and Script Design Priciples

For complete automation of testing we need to build a harness around the environment that incorporates test scripts. Scripts will work properly once the pre and post test work is done efficiently. In this section we will describe attributes of a good test harness as well as a test script. When customized test tools are being created they have to fulfill certain requirements in addition to generating traffic and simulating the needed test scenarios for the test environment. This section describes the complete solution needed to be developed in order to run a successful test. The test tools and scripts need to execute both pre and post test activities, in order to create a complete framework for testing purposes.

One of the major advantages of creating a test script over manual testing is the ease of repeating tests. It also saves time and increases efficiency, but if tasks like starting up the test environment, validating it, running a single user validation test, collecting data after the test and so on are still being done manually it defeats the purpose of automating the testing process. If an organization or a particular team is serious about investing the time and effort to automate testing, all the processes mentioned above have to be part of that automation activity.

5.2.1 Test Harness Definitions

Some of the terms that will be used to describe the test harness are defined here:

- Instance: Every individual component of the data flow path or application is called an instance. It can be the application itself, the load balancer or even the database.
- Test Environment: A test environment is made up of a number of instances.
- Configuration File: The XML configuration file is a definition of the test environment. It describes each instance in terms of the server name/IP, port, application directory, users and related instances. Providing the correct details in this configuration file is of utmost importance since the test harness will read this file and then perform the associated activities like starting up the environment and validating it. We have used XML to define the test environment but a different language or any user defined structure can easily be used for the same purpose.
- Activity Script: In addition to the environment definition file, there will also be scripts that will allow the test harness to perform certain tasks - like using secure shell to log on to a particular server and then change directory to the log files and either backup or remove (as specified) the files.

5.2.1.1 Test Harness Activities

Before a test can be run, it is vital to ensure that the environment is setup properly and all instances in the environment have been started and are running. This can be verified by sending either single user traffic for a few minutes to make sure that each of the instances are receiving the requests and processing them or a single transaction that will go through each of the instances and make certain that they are running correctly. Some of the basic functionalities that the test harness must accomplish are as follows:

- Clean: The clean command will read the environment configuration file, secure shell login to each instance server and remove or backup the log files using an activity script.
- Validate: This is the step where the test harness tool reads in the configuration file and parses it to retrieve the test environment instance definitions. It then locates the servers in the environment dedicated to the instances and the associated activity scripts. It then logs in to each of the instance servers and confirms that the server and instance are both ready to receive traffic. The validation process could consist of the following but is not limited to checking that:
  1. The proper directory structure for the test exists
  2. The instance specific configuration files exist and are correct
  3. The disk space on the server is sufficient to run a test
  4. Instance specific data collectors are running

  Activity scripts for each of these actions can be created to automate the validation process. Failure of any of these scripts implies that a server or an instance is not setup correctly. The test harness should report the error and abort. The process of validation works as shown in Figure 5.1.
- Start-Environment: This command also used the configuration file as an input and is used to bring up the instances that make up the test environment in the specified sequence. There might be a case where an instance's startup is dependent on another application being up and running. This sequence is identified in the configuration file and an activity script is used to point to the proper directory and execute the startup command for a particular instance.
- Ping: This is the option that will enable sending a single user or a single transaction through the environment to validate its setup. The XML defines the user credentials used to logon to the server, the request as well as the pattern

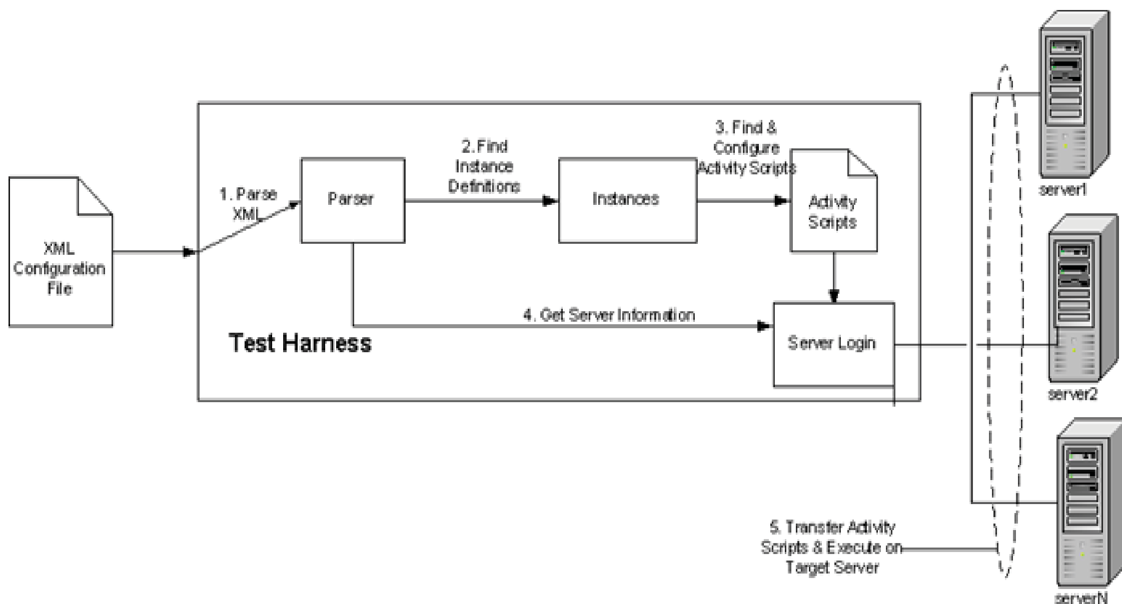


Figure 5.1. Step by Step Visualization of the Validation Activity of the Test Harness.

to look for in the response. If the response does not match the defined pattern, the harness will record the error message and abort.

- Start-Test: This command will pick up the associated activity scripts and execute them to start a performance test.
- Stop-Test: This will stop the test that is currently being run in the environment.
- Stop-Environment: This will take the configuration file as an input and an activity script to stop all the instances that make up the test environment

5.2.2 Test Script Design

Test scripts are part of the automated test harness. It is extremely critical to create good scripts in order to successfully run a performance test. As mentioned in section 1, creating test scripts is very similar to writing code for an application. Before an application undergoes functional or performance assessment, testers need

to document all the scenarios that will be executed. These become the requirements and use cases for the test script to fulfill. The development of an application is never static - maintenance and enhancements to the functionalities are done constantly to improve its performance and capabilities. This implies that test scripts assessing the performance of an application also need to be modified constantly enabling the addition of new test cases. Incorporating the qualities mentioned in section 1 will ease this process

Good documentation and modularity helps with the modification of test scripts. There is some documentation available on how scripts should be written in order to simulate real user behavior. Adding delays between transactions going through the test environment is one of the most common practice to emulate user actions. Most of the work is concentrated on applications that have a web interface where users are simulated to imitate the number of peak users at any given point of time when the application goes live. Almost all test tools available in the market are also able to record user actions for a particular web application and play them. Running tests for backend applications that process the transactions is also important. Web transactions are usually transformed before they reach the backend applications and may use protocols other than HTML for processing traffic. In such cases test tools where you can write a script using customized protocols or traffic formats are limited. Recording production logs for such traffic and then playing them back in the test environment is also to some degree restrictive when using commercially available traffic drivers. In such situations testers have to write their own scripts that will drive traffic.

The other factor to keep in mind when writing test scripts is the use of simulators or mocks. Usually in a test environment simulators are used to stub out applications that might be external to an organization's network or might be a third party product.

These stubs are referred to in this chapter as mocks. In such cases test scripts need to be synchronized with these simulators in order to execute the test cases where traffic playback is involved

5.2.2.1 General Test Script Design Principles for Environments without Mocks

One efficient way to create a test script that is bound to be changed constantly is to design it as shown in Figure 5.2. The test script is capable of collecting logs in any format from applications running in production and using them as requests for playing back the traffic to the same applications in test environment. These requests are collected in a text file and the script can then add headers or footers as necessary before sending them to the target server. The script uses transactions per second (TPS) method for setting the transmission rate with possible limits to the number of open connections.

The first step in this design is to collect the requests from production that will be replayed during testing. Usually there are multiple servers running in production for the same application. Depending on the configuration, each of these servers might be processing a particular kind of request or requests are simply being load balanced among these boxes. Either way the script that collects the input requests needs to round robin among all the servers. This ensures that data is being sampled uniformly among all available servers. This also makes certain that all input traffic types are collected because there may be certain scenarios where specific incoming transactions are routed to particular servers. Performance tests are run from anywhere between 48 - 72 hours or longer. For this reason the request collector needs to record enough traffic to last that test duration. The traffic can be in any format as long as the application under test can recognize and process them. The test scenario, traffic format and part of the test environment are described using a configuration file. The test script needs

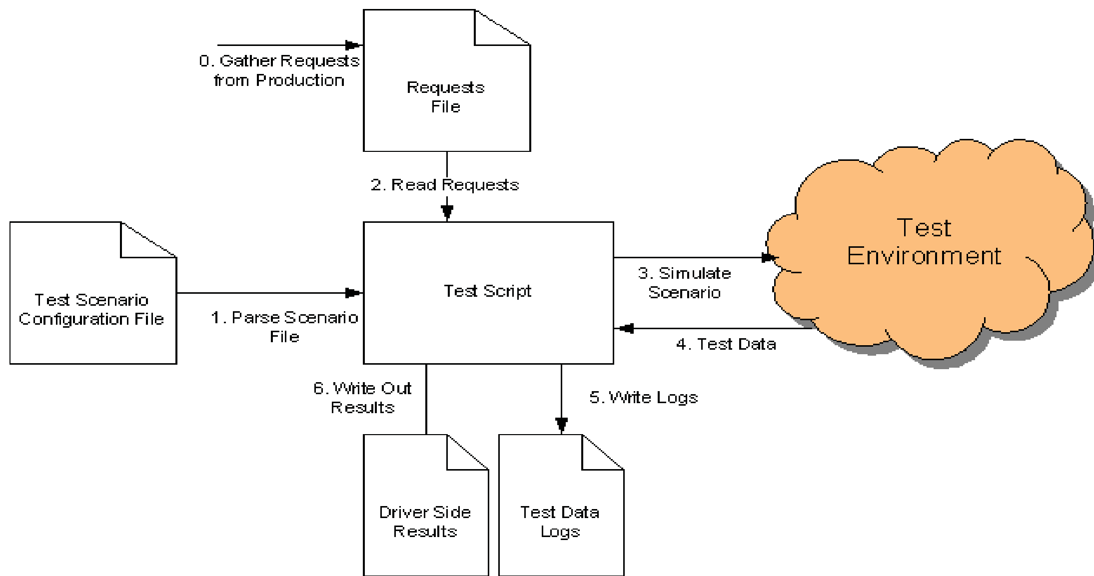


Figure 5.2. General Test Script Design for Environments without Mocks. Additional Components Needed for Environments with Mocks are shown in Figure 5.6.

to be general enough to process any kind of acceptable format. After reading the input request format from the configuration file the test script decides which internal module to call for processing. Once the test script reads the test scenario details - TPS levels to set at the start of the test, TPS increments during the test, the test duration, server or servers to send traffic to, where to read the input requests from and where to write the test logs to: it simulates the scenario. The script logs the responses sent from the test environment along with the time the environment took to process the transaction. Once the test is complete, the script spits out a summary report that details the TPS during the test duration, the transaction processing times as well as the number of the erroneous responses and types if errors that occurred during testing.

Factors that should be considered when creating test scripts are described as follows:

1. Clear Directory Structure

   Just like a clear directory structure needs to be created for a piece of software, the different input and output files for the test script need to be well structured also. This enables easy and comprehensive understanding of the test script and all its associated parts which in turn helps in managing the development of test scripts. A sample directory structure is shown in Figure 5.3.

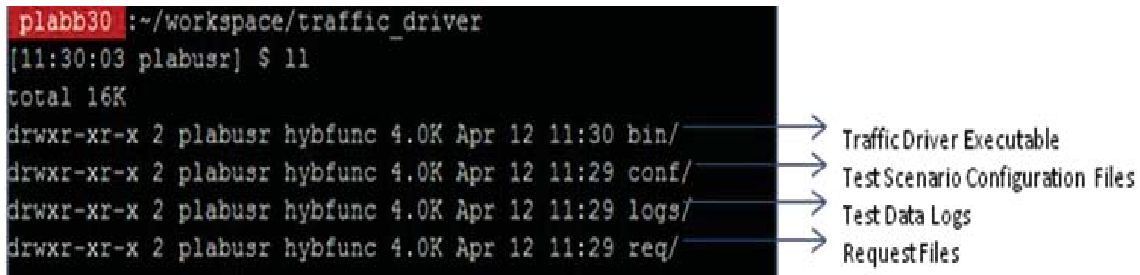


Figure 5.3. Sample Directory Structure for Easy Test Script Development and Management.

2. Test Script Configuration File

   The configuration file is the core piece of the test script. This is the file that describes the test scenario in detail. A sample of the configuration file is shown in Figure 5.4. The file details the format of the input requests, the directory from where the script can read the requests and the directory where the logs will be written to. It also describes the TPS levels, user ramp ups and ramp downs, duration of the test, the server or group of servers that are to receive the traffic, the port on which to send traffic and if applicable the size of the

responses. A console port to see the steps being executed by the script are also defined in the configuration file.

```
 // comments or disabled lines start with the double slashes
 // system call gets run immediately after it is parsed
 // Used to run a pre-test script or gather the entries to send

 RequestFile req/A01.txt          Test Requests Location
// skiplines 6000
 filemode BYLINES              // BYLINES,ALTLINES,BREAK
 quantity 3000                 // read in up to xx requests from input file
 LogFile  log/A01.txt          Test Logs Location
 csvfile  log/A01.csv
 headermode OTASOAP            // NONE,RHTTP,V2HTTP,OTASOAP
 TPSdesired 1                  // 0.5 tps default 0.25 min 49.9 maximum
 StartDelay 0                  // wait this many seconds before starting traffic
 maxruntime 3600 // this will allow the program to run for at most 1 hour
 MaxResponseSize 12000  Test Scenario Definition  lue to trim responses
 MaxRequestSize 200                               lue to trim requests in log (50% front 50% tail)
 limitrequests 15000           // per server: for unlimited run set to 0 otherwise control number of requests sent
 responsemode OTA              // ALLPASS, OTA, XMLV2,
// Setting responsemode overrides defaults set by headermode if needed

// ramping feature     TPS Ramp
 rampdelay 60          Features
 softstart 2                      Initial TPS add after first rampdelay seconds occurs
 rampinterval 30
 rampquantity 20               // 20 times 0.5 is 10 TPS added for a total peak of 13 TPS
 rampsize 0.50          Console
 interactive 0          Features  do we wait for console input or are we scripted
 consoleport 6000                 where do we listen
 findnextport 10               // if in use, look at the next xx ports to find an available one
 serverport 53601              // port MUST come first
 samerequests 1           Instance Definitions  ver identical traffic
 server server02 server03                       dress list
```

Figure 5.4. Sample Test Scenario Configuration File that Clearly Describes Detailed Test Scenario, Interactive Console Features, Request and Log Directories. Configuration Files Simplify the Test Script Development and Ease Maintainability.

3. Test Script Console

   The script can be run disconnected in the background so as to not need a constant console. Screen or any other console virtualization tool can be used for that. But the script can also be run using a console port to control and monitor the progress. The console serves the purpose of controlling the test as it is running. For example functions like increasing or decreasing the TPS

by 10%, increasing the frequency the report status, changing the request file or resetting the TPS levels while a test is running, is not possible is the script is running in the background. But if needed a console can be opened on a port specified in the configuration file and control the test.

4. Self Tuning Capability

   If the test script is not being run in the interactive console mode, the self tuning functionality needs to be added in order to adjust the TPS levels during tests. There are times when the applications under test might be encountering a processing error and therefore the TPS rate increases during tests. Or there might be a case where due to specific huge requests, the applications take longer to process requests and therefore the TPS decreases. In such cases the test script should have the capability to self tune and adjust the TPS to the target level.

5. Test Script Logs

   Both basic statistics and detailed logging should be supported by the script. The logs at a minimum should keep track of the name of the server where the request was sent, the timestamp at which the request was sent, TPS, the request and response size, the time it took to process the request and the status of the processing. The test script is basically simulating client side actions and logging this data will give a fair idea of how the application will perform in terms of end user experience when it goes live. Logging in a comma delimited file will make it easier to open up the data in an Excel or another tool and view.

6. Post Test Report

   As stated earlier the test script also needs to create a report at the end of the test. Figure 5.5 shows a sample output from the script after a test. Note that apart from recording the numbers of errors that occurred during the test, it is also critical to record the kind of errors that occurred. For this reason the

script not only receives and logs the responses from the test environment, but also needs to parse it to see if it was an erroneous response. Keeping track of the types of errors during tests helps in investigating the offending transaction or even component.

```
18:38:46,223 INFO  Final statistics:
SUMMARY OF THE PERFORMANCE STATISTICS:
###############################################
---------------------------------------------
Total number of requests              : 144000
Total number of success responses      : 137066
---------------------------------------------
Total number of receiving failures      : 6934
- including number of SSG_ERR responses : 3
 - SSG_ERR_SUPPLIER_TIMEOUT            : 3
- including number of timeouts          : 6931
---------------------------------------------
Actual TPS rate                       : 9.8
Average success response duration [ms]  : 10279
Average SSG_ERR response duration [ms]  : 39220

---------------------------------------------

###################################################
```

Figure 5.5. Sample Post Test Report for a Script. Automated Generation of Such Reports Helps Identify the Types of Errors and Application Behavior during Testing.

Developing the main script as general as possible and then customizing it using configuration files is good practice in order to facilitate the addition of new test cases with little to no effort. Following this design pattern and accommodating the pre and post test activities creates a complete harness that will enable the proper automation

of the testing process. This facilitates the creation of a useful alternative to manual testing.

Testers ensure that the environment is setup correctly, run traffic through the environment, record the types of errors and bugs that occurred during testing and then create a report at the end of testing. The test harness design described in this section incorporates and automates all these activities and provides a complete testing framework.

5.2.2.2 General Test Script Design Principles for Environments with Mocks

Usually simulators/mocks are used in the test environment to emulate the processing of complex backend applications or third party software. In cases of record and playback at the test script side in an environment using a simulator it is important to synchronize the test script and the simulator as shown in Figure 5.6. The main reason for that is to make certain that both the traffic generator and the mock use the same delay between transactions so that actual results being generated during the test match the expected test results. The test entries are taken from the application logs in production and contains the delays recorded between the transactions. The test scenario and entries file contains these delays as recorded in production. Both the simulator and the test script also need to read the same configuration file that describes the test scenario and test entries.

There is an additional component used in this kind of setup - expected results generator. The results generator reads the configuration file, parses the scenario and creates graphs for the various metrics plotting the expected behavior. These expected results are stored in a database. While the test script is running (keeping in mind the design factors mentioned in the previous section), it records the responses from the test environment and stores the test data in a database. Either during the test

or after full completion the comparison and reporting component can be used to see how well the test is emulating the expected scenario.

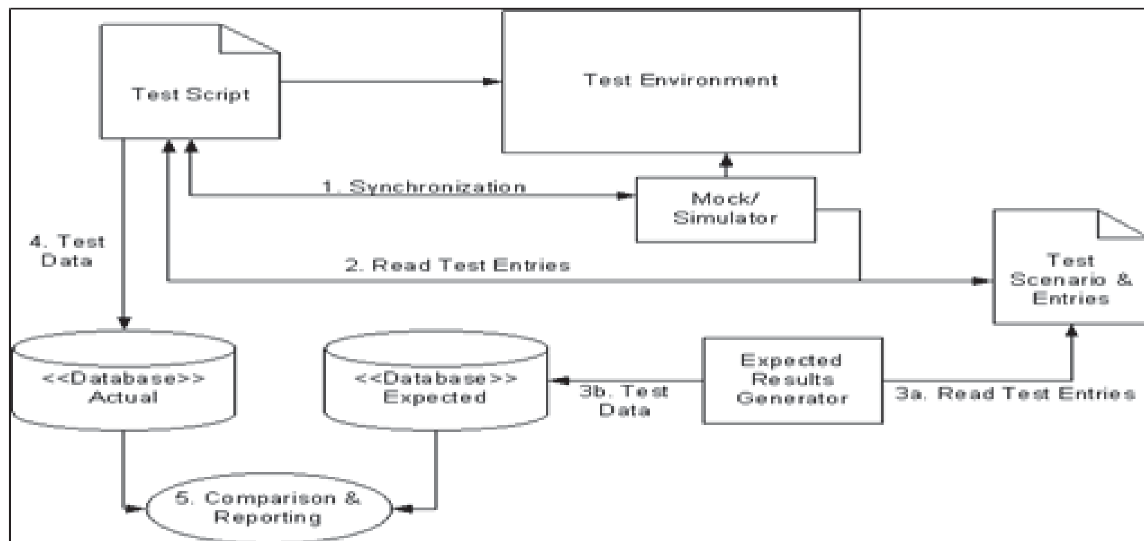


Figure 5.6. Test Script Design for Environments with Mocks. Basic Test Script Design Principles - replaying traffic, reading configuration file and creating reports are based According to Figure 5.2.

Due to the synchronization between the test script and the mock, results generated while a particular test is running should match the expected results. This is an indication that the script and the applications in the test environment are performing as expected. Let's take the example of a simple test scenario where the test is being run for 30 minutes. During this time a total of 11000 connections are being opened initially and then after 5 minutes additional 3000 are opened every 30 seconds and a constant TPS level of 600 is maintained for the entire duration of the test. The graph in Figure 5.7 shows the expected results plotted against the actual results compiled during testing. Discrepancies between the expected and actual results as shown in the figure implies that there is some kind of difference between how the real envi-

ronment would have behaved with the test entries and scenario as compared to the test environment. This will encourage the test team to investigate and identify the culprit component in the test environment - be it hardware, software or configuration related thereby improving the overall the test environment setup and increasing the confidence in the results.

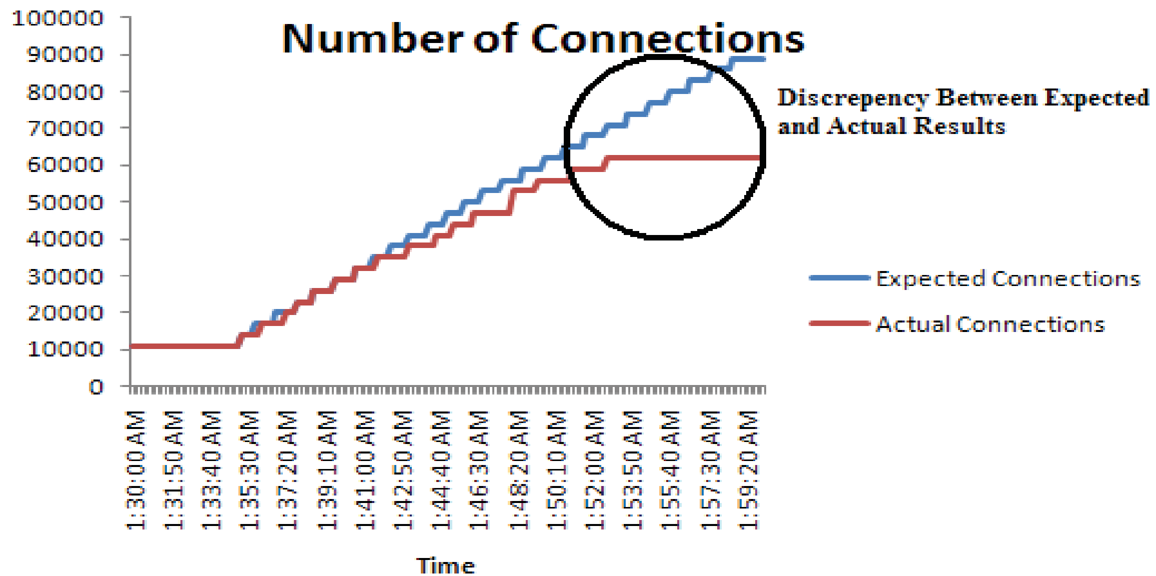


Figure 5.7. Actual vs. Expected Test Results.

5.3 Test Harness Example

Let's take a look at how a test harness and script created using the design principles mentioned in the chapter will help in automated testing of backend applications. Consider a flight shopping application as shown where the incoming client request XMLs are guided to a pool of web servers and are then load balanced to go a pool of shopping servers. These shopping servers transform the XMLs adding specific

headers to the requests and put them in a MOM (Message Oriented Middleware) queue. These messages are then picked up by the scheduling application that talks to an Oracle database. The scheduling application finds the available schedules for the origin and destination cities and dates specified in the input request and puts the response in the MOM queue. The shopping servers will then pick up the responses and send them back to the client.

It is difficult to find a test tool that will be able to read the format of the transformed XMLs and then generate thousands of TPS in order to rigorously test these backend applications that do not have a GUI or web interface. The test harness and script described in the chapter can easily accomplish the task. An XML configuration file can be created to explain the test environment and all the instances. The harness will then be able to read this configuration file, enable remote login to each instance server and achieve the tasks mentioned in section 2.1.1. Figure 5.8 shows how the sequence of activities followed by the harness and script.

The other advantage of using an automated test harness that starts and validates a test environment and then initiates the test script that drives traffic, is repeatability. It is of utmost importance to make certain that an application is thoroughly tested before it is deployed in production. Newer releases of the application are created to add more functionality. So how do we make sure that the new release is performing the same or better than the previous release? With the configuration file describing the test environment as well as the test scenario that is ultimately read by the harness, recreating test situations is easy. Results of two releases can be compared knowing that all activities followed to run a particular test were the same for both the releases. Figure 9 shows a sample report created by the test script after two releases were tested using the harness.

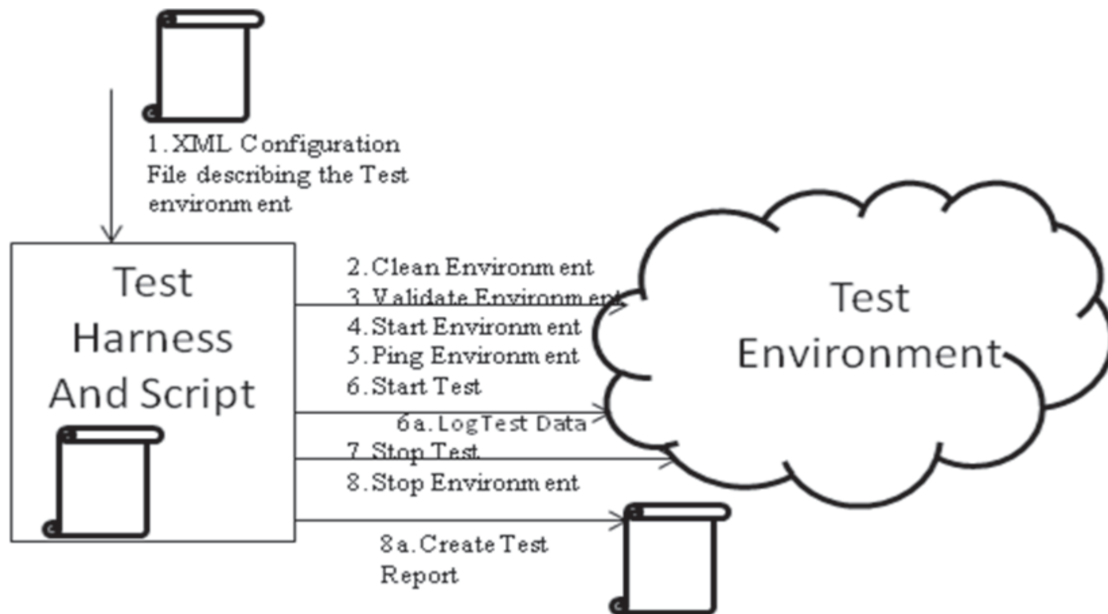


Figure 5.8. Sequence of Activities for Test Harness and Script.

5.4 Related Work

The academia as well as the industry is interested in finding novel ways to efficiently automate testing. Manual testing is tedious and not suitable for uncovering performance issues in complex computing environments. Automation provides repeatable and predictable test executions. But more often than not the focus is directed on improvising the methods to create test script that interact with web or GUI interfaces. This is extremely valuable since having good test scripts that emulate user behavior leads to well organized and competent testing of software. [32] talks about how using XML to create simple and flexible scripts and third party validation and parsing tools reduces development. [33] describes how using python can benefit testers by creating real time dynamic test scripts and a script interpreter that executes these scripts within 10 ms. [34], [35] take up the task of writing test scripts

for GUI based applications based on the REST and UML models respectively. They describe an automated way to create test scripts that can be easily changed as the GUI interface changes. [36] presents methods and algorithms to create automated tests cases and test data for hybrid systems. Automated testing is not just bound to creating and running scripts that simulate the various test cases. In order to create a fully automated testing framework, pre and post test activities also need to be included. [37], [38] describe the importance of record and playback during testing. [37] illustrates the method of selective record and playback between a sub system and the rest of the application. [38] introduces a personal virtual computer recorder model that enables the capture of user interactions while operating a desktop while [39] describes one of the ways to automate usability evaluation by tracing user interactions and playing them back. The authors of [40] observe that steps like changing configurations, creating post test reports along with data analysis need to be part of the automation process along with the test scripts. This paper addresses these pre and post test activities and describes a test harness that will be able to completely automate the testing process along with providing factors to consider during the design of extensible and easy to maintain test scripts.

CHAPTER 6

DESIGN OF EXPERIMENTS (DOE) FOR PERFORMANCE EVALUATION OF SOFTWARE APPLICATIONS

6.1 Introduction

So far in this dissertation we've gone over a few areas of performance testing such as the process, the tests, creation of an automated test harness and scripts and the monitoring and instrumentation of a piece of code and proposed solutions that improve the existing methods. When evaluating each of these proposed techniques, tests were run on real world travel applications. Tests can be interpreted as experiments. The general process of running a performance test is to list out the input parameters, setup the environment for the code under test, collect the data during the test and then analyze it and derive conclusions from this data. These steps are similar to how an experiment is defined. Experiments are run in all fields of study and are run on a given set of elements. Data is collected and observations are made while the experiment is running in order to arrive at a particular conclusion. And so performance tests run in order to validate how well a particular piece of code is working, are nothing but experiments. Design of Experiments is a specific field of study that deals with how to construct experiments/tests when there are multiple factors being used. ANOVA or analysis of variance is then used to determine which factor most or least affected the result of the experiment. In order to save time and still run useful experiments when there are multiple factors to consider, certain guidelines help with constructing good experiments. The following is a list of such guidelines [41] :

1. Identification of The Problem: For an IT company the mission is to continually improve products and processes. In our study the objective is to improve the process of finding bottlenecks in applications located in a tier other than the front end tier. This includes finding pressure points that might be introduced due to the interactions with other pieces of code. The problem in this case is to identify the factors that could cause bottlenecks to occur in a computer system and experiment with techniques to discover and reduce these bottlenecks.
2. Formally Defining the Problem: This next step takes care of the requirement gathering phase - what needs to be simulated and how it will be done, what are the factors to consider that could have direct or indirect effect on the system.
3. Identifying Points of Instrumentation: The end result of designing an experiment for software applications is to reduce the possibility of any disruption to service available to the end user. So components/metrics are chosen to be instrumented if they have a direct impact on the quality of the service/product. So in our study where software applications are being simulated and experiments are being designed to find bottlenecks, the measurable metric of importance will be the end to end response time. The end to end response time is the time that the customer sees as the total time it took for the system as a whole to process his/her request.
4. Identifying the Input Parameters/Factors to the Experiment: Just like the points of instrumentation in the system, factors (input variables) used in the experiment have to be measurable and/or distinguishable. The factors must be such that they can be controlled. The design of experiment methods will help in the determination of what combination of factors should be used to run the experiments and therefore it is essential to have distinguishable and controllable input parameters. Everything that affects the value of the response variable

is a potential factor to be used in the experiment when it comes to software applications.

5. Step-by-Step Process Planning of The Experiment: The step-by-step planning of how to run the experiment is termed as layout of the design. The system under test is called the experiment unit. Ideally we would want the experiment unit to be affected only by the input parameters that were chosen for the experiment but more often than not there are unknown factors that could affect the experiment unit. These unknown factors could be environment related or could be noise. So to reduce the effect of these factors, randomization of the experiment is recommended. This basically implies a randomization of the combination of factors that are being used as input. Unfortunately complete randomization is not always possible but restricted randomization is. The restriction on randomization is called restriction error.
6. Development of The Mathematical Model: The mathematical model (wherever possible) could help with the analysis of the data that is collected during the experiment. The model should be derived in such a way that it contains all information about the input parameters used in the experiment along with their specific values/levels. The mathematical model should also take into consideration the restriction error, if it was used during the experiment.
7. Assessment of The Experiment Design: This next step involves using various techniques to derive the appropriate analysis of variance or ANOVA table. The detailed description of what the table contains will be discussed in a later section in this chapter. The ANOVA table helps identify the effects of the different input parameters that were used during the experiment on the data that was collected.

8. Redesigning The Experiment: if issues are found in the original design of experiment, this is the time to redesign it. The new layout can be compared to the original and then the better solution can be chosen to run the actual experiments.
9. Data Collection during The Experiment: Data collected during the experiment is of utmost importance. It is looking at this data that the stakeholders can make an informed decision about the experiment unit. Therefore it is critical to know what data has to be collected in order to properly assess the experiment unit. In the case of the software application, as mentioned in an earlier point - response times are collected because they have a direct impact on how the end user perceives the service/product.
10. Data Analysis: Another essential step when running an experiment is data analysis. It is not sufficient to simply collect the data during the experiment but it is also important to analyze it correctly. Outliers should be kept in mind or ignored based on the historical data for the experiment unit. A visual display of the data in the form of graphs is usually the best method in presenting information to various stakeholders.
11. Implementation: Last but not the least is the process of implementing the recommendations gathered from the experiment to the actual system. Therefore it is critical to develop a realistic type of experiment, ensure proper running of it and collecting, analyzing and then presenting dependable data.

The main objective to study DOE methods in our study is to use them to establish the input test cases that would help evaluate the performance of a backend software application. We will be using an airline shopping application that interacts with different components to process incoming transactions. The goal is to assess it's performance not only as it relates to TPS, as existing performance tests do, but to

evaluate it based on other factors that might have a direct or an indirect impact on how well it performs. Since we will be dealing with multiple factors, each of them having multiple levels/values, we want to choose the best DOE method to run the tests. At a very high level, following are the steps we want to follow:

1. Choose factors that could affect the application's performance
2. Looking at the production data, choose the appropriate levels for each of the factors
3. Select the most suitable and practical DOE method to setup our test cases. These test cases will be used to run tests on the pilot application to assess it's performance.
4. Choose a pilot, which in our case will be an airline shopping application
5. Propose a solution that will help evaluate performance and detect pressure points using state space models

The following sections contain common knowledge on Design of Experiments available in various text books like [41]

6.2 Background - Terminology of DOE

This section explains the terminology used for experimental designs:

- Factor: An input parameter that will be used to run an experiment on a particular unit is termed as a factor. In case of performance evaluation of software applications, that we are interested in, factors would be parameters like TPS (transactions per second), queue lengths and so on.
- Factor Levels: A particular value of a factor is termed as it's level. A factor could have any number of levels. So if we take the example of TPS as a factor, it's levels would be the exact numeric values - 20, 50, 100 and so on.

- Scenario: An experiment scenario can be defined as a combination and permutation of all the factors and their corresponding levels.
- Simulation: The simulation is itself a model of some real-world system, process, or entity.
- Experimental Unit: The system under test is termed as the experimental unit. Data and observations are captured for this unit during tests. In our study this is the airline shopping application.

6.2.1 One Factor Experimental Design

This section will introduce how DOE works when there is a single factor with different levels to be implemented on an experimental unit/s. The next sections will detail how multiple factors affect the layout of the experiment. The section will also introduce the notation [41] that will be used going forward.

The different levels for a particular factor will be described using the subscript i, that ranges from 1 to I. This basically implies that there a total of I different levels for a specific factor. An experiment unit will be described by the index e, that ranges from 1 to E. This implies that there are E different experiment units and therefore the total number of experiments that will be run are IE. In our case where we will be implementing the DOE method on an airline shopping application so J = 1. A balanced experiment is one where each of the I different levels are applied to the same number of experiment units. The notation y_{ij} indicates that this is the output measurement when level i of a particular factor was applied to the eth experiment unit.

So now in order to create a mathematical model for the experiment, we assume that the output measurement y_{ie} is the results of the model as shown in Equation (6.1)

$$y_{ie} = \mu + A_i + \varepsilon_{e(i)} \tag{6.1}$$

where μ is the overall mean of the process, A_i is the differential effect due to the ith level of a fixed factor A and $\varepsilon_{e(i)}$ is a random error component. We will assume that

$$\sum_{i=1}^{I} A_i = 0 \tag{6.2}$$

$$\varepsilon_{e(i)} \; Normal(0, \sigma^2) \tag{6.3}$$

The assumption that $\sum_{i=1}^{I} A_i = 0$ can be made without loss of generality because if $\sum_{i=1}^{I} A_i$ were equal to some value C, we could replace μ with $\mu + C$ and $\sum_{i=1}^{I} A_i$ would then be equal to 0. Likewise we can assume that $\varepsilon_{e(i)}$ has mean 0.

The mean of all the measurements taken during the experiment is nothing but the natural estimate of μ. The differential effect of level i for factor A from the mean is A_i. So to calculate the natural estimate for A_i, we will need the mean of all the data collected during the experiment for the ith level of factor A and then subtracting the overall mean will give us the natural estimate for A_i.

$$\bar{y_i}. = \sum_{e=1}^{E} y_{ie}/(E) \tag{6.4}$$

$$\bar{y}_{..} = \sum_{i=1}^{I}\sum_{e=1}^{E} y_{ie}/(IE) \tag{6.5}$$

So in order to infer the effect of A_i on the experimental unit, the analysis of variance table can be used as shown in Table 6.1

6.2.2 Factorial Experimental Design

In the last section we saw how the mathematical model can be derived for a single factor A with levels from 1 to I represented by the subscript i. In more practical

Table 6.1. ANOVA for One Factor Design. df represents the Degrees of Freedom, SS the Sum of Squares, MS is the Mean Squares and F is the Calculated F-value

Source	**df**	**SS**	**MS**	**F**
A_i	I - 1	$E\sum_{i=1}^{I}(\bar{y_{i.}} - \bar{y_{..}})^2$	SS(A)/(I-1)	MS(A)/MS(ε)
$\varepsilon_{e(i)}$	I(E-1)	$\sum_{i=1}^{I}\sum_{e=1}^{J}(y_{ie} - \bar{y_{i.}})^2$	SS(ε)/[I(E-1)]	
Total	IE-1	$\sum_{i=1}^{I}\sum_{e=1}^{E}(y_{ie} - \bar{y_{..}})^2$		

scenarios, there is usually more than one factor that is applied to the experiment unit. This section will briefly describe these scenarios and the design of experiments for them. Designs with more than one factor are termed as factorial designs and each level of these factors is applied to the experimental unit. For terminology purposes just as in the previous section, the various factors will be represented with upper case letters. The different levels of the factors will be labeled with the subscript i, ranging from 1 to I. Similarly the levels for the second factor will be denoted by j, ranging from 1 to J. For the one factor model seen in the previous section, the model was written as

$$y_{ie} = \mu + A_i + \epsilon_{e(i)} \tag{6.6}$$

where μ is the overall mean and A_i is the differential effect of factor A with a restriction $\sum_{i=1}^{I} A_i = 0$. $\epsilon_{e(i)}$ is the random error term assumed to be $Normal(0, \sigma^2)$.

So now in order to extend equation (6.6) from one factor to multiple factors, it is important to understand how multiple factors affect the experimental unit. Since equation (6.6) is applicable for a single factor, the measurements are taken for each specific level for factor A. In order to assess the effect of the factor on the end result, the mean levels of the factor were used for the ANOVA calculation. Now that we have multiple factors, instead of individual means the interaction among the different factors is more important. This interaction is termed as random factor. In order to

utilize the random factor, it is assumed that factor A is distributed $Normal(0, \sigma_A^2)$ where σ_A^2 is independent of σ^2 and is called the error term. If the effect of a random factor is significant, it's variance is calculated i.e. estimate σ_A^2. The individual means of a random factors are not useful.

So now if two different factors A_i and B_j are included in an experiment, their interaction term AB_{ij} will also be included. Thus for a two factor design, the model becomes:

$$y_{ijk} = \mu + A_i + B_j + AB_{ij} + \epsilon k(ij) \tag{6.7}$$

where i ranges from 1 to I indicating I different levels for factor A, j ranges from 1 to J indicating J different levels for factor B and k ranges from 1 to K indicating that each of the ij combinations of factors A and B occurs K times. Keeping this notation in mind, a three factor model can be written as:

$$y_{ijkl} = \mu + A_i + B_j + AB_{ij} + C_k + AC_{ik} + BC_{jk} + ABC_{ijk} + \epsilon_{l(ijk)} \tag{6.8}$$

The extension to more than three factors follows the same pattern as (6.7) and (6.8).

6.2.2.1 ANOVA Table for Factorial Designs

The natural estimate for A_i would be $\hat{A}_i = \bar{y}_i.. - \bar{y}...$, if there were two factors A_i and B_j and there were K repetitions for the various factor level combinations. The sum of squares for A, SS(A) would be $JK \sum_{i=1}^{I} \hat{A}_i^2$ which implies this is the sum of squares of the natural estimates of A_i for all data. If I is the estimated average minus one, the df would be I - 1. Similarly for B_j, SS(B) will be $IK \sum_{j=1}^{J} \hat{B}_j^2$ and the corresponding df(B) will be J - 1.

Table 6.2. ANOVA for Three Factor Design.df represents the Degrees of Freedom, SS the Sum of Squares and MS is the Mean Squares

Source	**df**	**SS**	**MS**
A_i	I - 1	$JKL\sum_{i=1}^{I}(\bar{y_{i...}} - \bar{y_{....}})^2$	SS(A)/(I-1)
B_j	(J-1)	$IKLsum_{j=1}^{J}(\bar{y_{.j.}} - \bar{y_{....}})^2$	SS(B)/(J-1)
AB_{ij}	(I - 1)(J-1)	$KL\sum_{i=1}^{I}\sum_{j=1}^{J}(\bar{y_{ij..}} - \bar{y_{i..}} - \bar{y_{.j..}} + \bar{y_{....}})^2$	SS(AB)/(I-1)(J-1)
C_k	K - 1	$IJL\sum_{k=1}^{K}(\bar{y_{..k.}} - \bar{y_{....}})^2$	SS(C)/(K-1)
AC_{ik}	(I-1)(K-1)	$JL\sum_{i=1}^{I}\sum_{k=1}^{K}(\bar{y_{i.k.}} - \bar{y_{i...}} - \bar{y_{..k.}} + \bar{y_{....}})^2$	SS(AC)/(I - 1)(K - 1)
BC_{jk}	(J-1)(K-1)	$IL\sum_{j=1}^{J}\sum_{k=1}^{K}(\bar{y_{.jk.}} - \bar{y_{.j..}} - \bar{y_{..k.}} + \bar{y_{....}})^2$	SS(BC)/(J - 1)(K - 1)
ABC_{ijk}	(I - 1)(J - 1)(K - 1)	$L\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K}(\bar{y_{ijk.}} - \bar{y_{ij..}} - \bar{y_{i.k.}} -$ $\bar{y_{.jk.}} + \bar{y_{i...}} + \bar{y_{.j..}} + \bar{y_{..k.}} - \bar{y_{....}})^2$	SS(ABC)/df(ABC)
$\epsilon_{l(ijk)}$	IJK(L - 1)	$\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K}\sum_{l=1}^{L}(y_{ijkl} - \bar{y_{ijk.}})^2$	$SS(\epsilon)/IJK(L-1)$
Total	IJKL - 1	$\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K}\sum_{l=1}^{L}(y_{ijkl} - \bar{y_{....}})$	

The sum of squares for the natural estimate for all data, SS(AB) can be calculated as $K\sum_{i=1}^{I}\sum_{j=1}^{J}(\bar{y_{ij.}} - \bar{y_{i..}} - \bar{y_{.j.}} + \bar{y_{...}})^2$. And therefore the number of averages estimated minus one for the overall average minus the df for A and the df for B, will the df for the interaction between the two factors and can be represented as follows:

$$(IJ - 1_{-}(I-1) - (J-1) = IJ_I - J + 1 = (I-1)(J-1)$$

Using the same logic the differential effect of the combination of the levels i, j and k minus all the three main effects and all three factor interactions will be used to calculate the factor interaction for ABC_{ijk}. The SS(ABC) is the square of all terms summed for all the data and the df can be represented as: $df(ABC) = (IJK-1)-(I-1)-(J-1)-(K-1)-(I-1)(J-1)-(I-1)(K-1)-(J-1)(K-1) = (I-1)(J-1)(K-1)$.

The ANOVA for three factor designs is shown in table 6.2.

6.2.3 Orthogonal Main Effect Designs - OMEDs and Taguchi Method

The factorial design method works well when all factors in the experiment have the same number of levels. When there are situations where the number levels maybe

different for different factors, orthogonal main effect designs work better. When using these techniques it is not required to run the same number of experiments for each of the levels for each factor. As seen in the previous sections full factorial designs use all possible combinations for the factor levels to create the complete experiment layout. In cases where a large number of factors are applied to the experimental unit under evaluation, factorial methods result in a significantly large number of experiments. When using orthogonal techniques a subset of these experiments are selected without losing the overall effect. The process of choosing the subset of experiments from the larger set that guarantees similar results is called partial fraction experiment. Even though orthogonal methods are widely used, there are no standard guidelines on how to apply these methods to the experimental unit or how to analyze the data that is collected during the experiments. Taguchi created some general policies on how to select the number of experiments to run for multiple factor, multiple level experiments.

An orthogonal array created by Taguchi is used as a benchmark to select the right subset of experiments to run. Even though the number of experiments is reduced from the factorial design, the selection intends to still produce data that will help analyze the total effect of the all the factors on the experimental unit. The process of designing experiments using the Taguchi method is described in the flowchart below [42]:

The Taguchi orthogonal array has columns and rows as shown in Figure 6.6 taken from [42] where the columns represent the number of levels and the rows represent the number of factors. The experiment array can be selected from this general orthogonal array by looking at the column and row that corresponds to the experiment's factor and level numbers. The subscripts in the Figure 6.6 are the number of experiments that need to be run. Each of these numbers has a standard

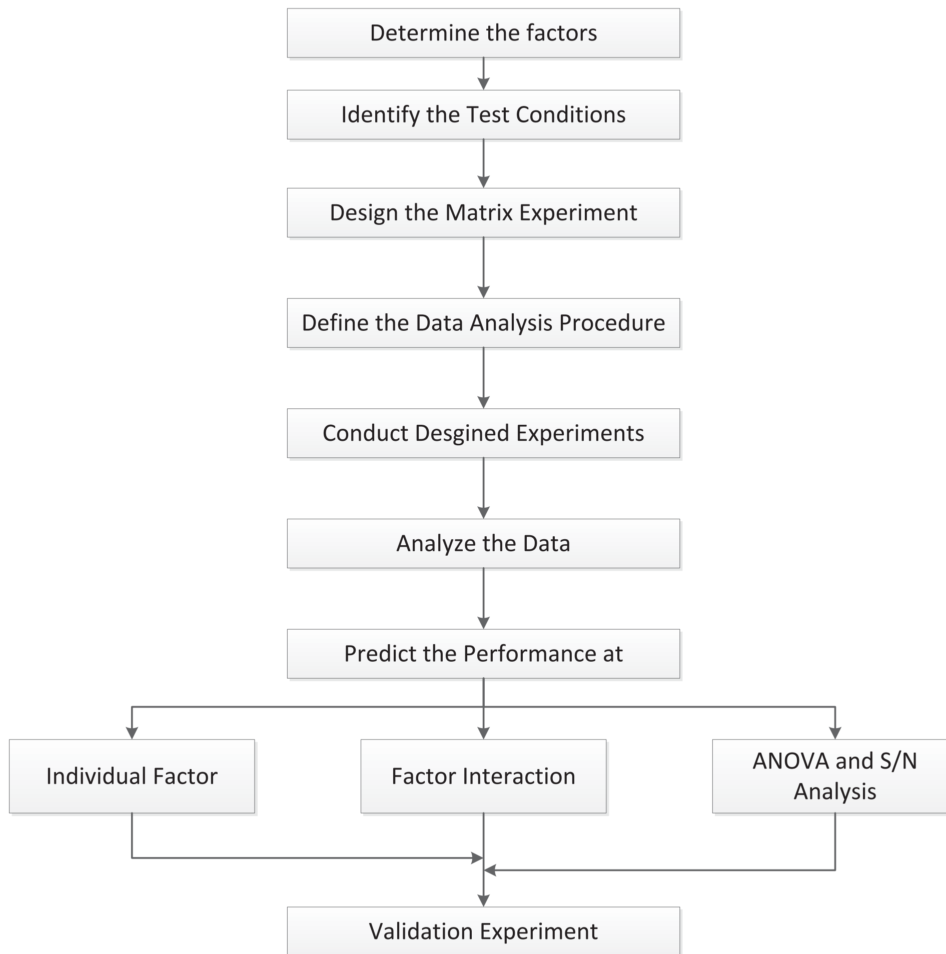


Figure 6.1. Execution Steps for the Taguchi Method of Design of Experiments.

predefined array associated with it. Experiments are then run by looking at the factor and level combinations from these standard arrays.

Let's take a look at an example from [42] on how to use the Taguchi orthogonal array. Assume that we need to run an experiment that has four factors and each of these factors consists of three levels. The following equation can be used to calculate the minimum number of experiments that need to be run.

$$N = 1 + \sum_{i=1}^{NV} (L_i - 1) \tag{6.9}$$

where NV is the total number of variables and L is the total number of levels for each factor.

So using the Equation (6.9) the number of experiments to be run can be looked up from Figure 6.6. The Taguchi array points to L9 for this particular setup. The L9 array is shown in Table 6.3

Table 6.3. Layout of L9 Orthogonal Array

Experiment Number	**Variable 1**	**Variable 2**	**Variable 3**	**Variable 4**	**Performance Value**
1	1	1	1	1	p1
2	1	2	2	2	p2
3	1	3	3	3	p3
4	2	1	2	3	p4
5	2	2	3	1	p5
6	2	3	1	2	p6
7	3	1	3	2	p7
8	3	2	1	3	p8
9	3	3	2	1	p9

The rows in this array specify the combination of the levels to use for each of the factors for each of the experiments. So for example, experiment number two will be run using level one for factor/variable one and then level two values for the remaining factors. A total of nine experiments will be run, data and observations will be gathered for each experiment and then analysis will be performed on the resulting data set to draw conclusions about the experiments. Just like in factorial design methods, analysis of variance ANOVA on the data collected at the end of the experiments can be used to analyze the effect of the factors on the results. Also redesigning and re-selecting factors for the experiments to optimize results is part of this process.

6.3 Design of Experiments for Performance Evaluation of Software Applications

In this study we are looking at non client interfacing software applications and how we can evaluate their performance. There are numerous conventional tests like the load, soak, stress, capacity, destructive and so on that can be run for this purpose. All these existing tests that are performed in order to evaluate the performance of a software application are dependent one varying input factor - load or transactions per second(TPS). These tests are defined based on varying the level of load driven through the application. With this kind of an approach, a number of tests have to run in sequence to find the various bottlenecks in the application. Varying just load levels may not even uncover all the issues. In general numerous factors have direct or indirect influence on the application's performance. There is an obvious correlation between TPS and the performance of the application but as seen in the previous chapters there are multiple other factors that can affect how well a piece of software performs. For middle tier applications (non client interfacing) as shown in Figure 6.2, interact with both front-end and back-end components and databases, engineering artifacts like timeout and throttles becomes all the more important.

So when we design an experiment/test we need to consider all factors that might affect the application for a proper assessment. And this is where the knowledge from the previous sections in this chapter comes in handy. There are numerous factors that can affect the performance of an application. These factors could occur any place in the setup:

- Server/hardware itself
- Application code
- A middleware component
- Resource - operating system or configurable resource like database connection pool

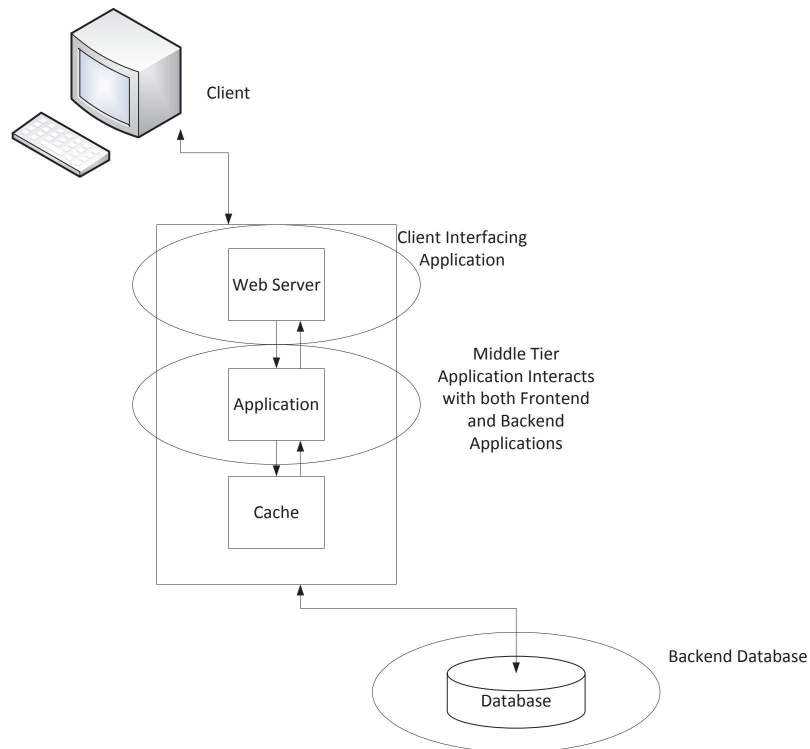


Figure 6.2. Multi-Tier Architecture of Software Applications.

And some of the factors that could affect application performance are :

1. OS resource exhaustion - lowering kernel parameters to load applications and simulate such scenarios
2. Timeouts from downstream applications
3. Increase in I/O wait times due to NAS or file systems
4. Asynchronous logging - running the application when disk is full
5. F5 health check scenarios - e.g. stopping one of the instances of tomcat and then checking if F5 has identified that an application is down
6. Sending extremely large requests to the applications so that MOM queues come into play
7. Application not draining ”responses” from MOM Queue response side

8. Database connections exceeded (usually a jdbc pool full)
9. Database exceeds SGA and PGA limits (running out of memory)
10. JVM out of threads
11. JVM out of memory
12. Malformed request formats (Data in XML that was not expected/expected so data patches have to be added)
13. Bad data on file (fares, rules, taxes etc)
14. Database not responsive (could be listener down) (deadlocks and how the app. responds to this)
15. Slowdown or loss of connection
16. Delayed responses (responses being sent in chunks)
17. Testing concurrency race conditions by slowing down of Java threads at random places
18. Database sessions exceeded (in database) - (parameters in init.ora file)
19. Database cursors exceeded (in database) - (parameters in init.ora file)
20. Database table space full (throw an error, retain the transaction and log an error)
21. Database temp space full (throw an error, retain the transaction and log an error)
22. User credential failure - fallback account could be used
23. Embedded applications - managing memory footprint (Active MQ embedded with Application on the same VM)
24. User credentials - authorization and authentication scenarios
25. Network packet loss —- Important in the database layer
26. URL not responding in cases of web applications like SSW
27. MOM and or database issues - queues filling up or database timeouts

28. Network packet corruption
29. Running out of file descriptors which leads to no tcp/ip connections - test to see that file descriptors are set properly
30. Running another instance of the same application on the same server
31. Wrong user starting up the application -test user privileges
32. Killing the application using kill -9 (killing parent or a child process by this hard kill and then issues when starting up the application, consistency check by application when started up again)
33. Ownership and permissions changes while the application is running
34. JVM not responding
35. Server temp space full - database and OS server (Oracle should store transaction to play back if running out of temp space) how the app. reacts to such issues is important

In addition to these, specifically for the airline shopping application, factors like time of day, rules for domestic and international shopping requests and number of sub-transactions for a single incoming transaction will also affect its performance. We've seen that there are different techniques available to setup experiments with factors and their levels in order to accomplish the goal of assessing application performance. Instead of using all the factors listed above to design experiments in order to evaluate performance, we will use the top eight factors that have caused severities in production for the airline application. The data has been collected by looking at the severity causes for the application over the last two years. The highest priority factors that have affected software performance are as follows:

1. Incoming transactions per second (TPS)
2. Workload/Traffic types
3. Timeout values

4. Throttle levels
5. Database connection thread pool
6. Queue size
7. Application connection thread pool
8. Payload size

When designing the experiment each of these factors is determined by five levels. If we went with factorial DOE, we would have to run $5^8 = 390625$ experiments or tests with the different combinations of levels for each of the factors. Since this is not feasible, we will be using Taguchi's orthogonal arrays to setup the tests. ANOVA will be used to determine the effect each of these factors on the application's performance. The client response time will be used to measure the performance of the software. As will be seen in later chapters, the state of the application is measured by different parameters - response time, CPU, memory, disk and number of core dumps. Out of these, response time and core dumps are the only parameters that have direct impact to the end user. Operating system resource utilization going over a pre-determined threshold may cause the server to become unstable but as long as the application is processing incoming transactions within SLA (Service Level Agreement) defined times, the end user does not get affected. It's only when the application crashes or the response times go over the SLA, is the customer affected and that is treated as unacceptable application behavior.

Another reason to go with the Taguchi orthogonal arrays was because of the uneven number of levels for the factors that have been chosen in our case. These arrays are effective even when the number of the levels among the various factors are not balanced.

6.3.1 Orthogonal Array DOE Setup for Evaluating Performance of Airline Shopping Application

In order to choose the right orthogonal array to run the tests for performance assessment of the airline shopping application, it is required to now determine the levels for each of the eight factors - TPS, workload type, timeout values, throttle levels, database and application thread pools, queue size and payload size. The Taguchi orthogonal array will then help us decide the number of tests/experiments to run and the exact combination of the levels to use for each of the tests. In our setup we are selecting the levels for the various factors using production data. The historical production traffic profile contains both peak as well as normal traffic levels. We are selecting the values for the levels using these profiles. If stress test kind of scenarios need to be tested, higher values for the levels can be chosen.

In order to find the levels for TPS to use in the tests, we will first look at the percentage distribution of the different payload types that the shopping application processes in one day. Figure 6.3 shows a pie chart with the percentages for the various traffic types/payloads that go through the application.

We will pick the top four traffic types that the application processes any given day for the tests and find the data distribution for them. According to Figure 6.3, the top four payloads are: INTLWPI1, LFSTREAM, ATSEILF1 and ATSEILF2. The data distribution for the traffic types is shown in Figure 6.4. All of these distributions are multimodal.

The levels for TPS are chosen after creating a frequency histogram for the different distributions. The histogram for the TPS of the airline shopping application is shown in Figure 6.5. The top five highest frequency TPS numbers are then chosen for the tests.

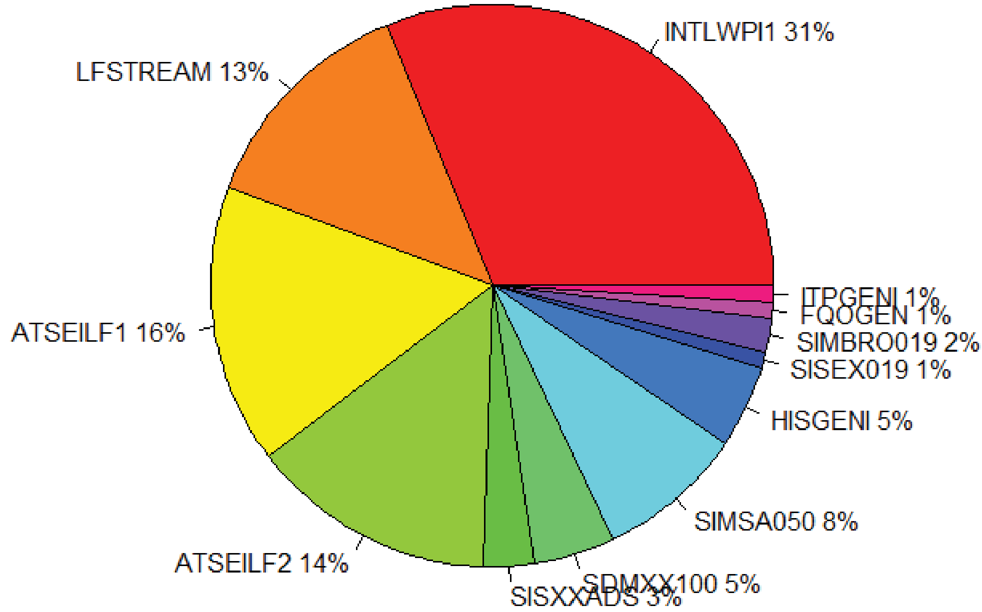


Figure 6.3. Percentage Distribution of Traffic Types for Airline Shopping Application.

So after looking at the production traffic profile and distribution, the levels for TPS and the payload types are chosen as shown in Table 6.4.

Just like choosing the TPS and payload type levels, the levels for the other factors are selected looking at the factor values in the production environment. The queue size and application timeout levels are shown in Table 6.5, throttle and payload size levels in Table 6.6 and the database and application thread pool levels in Table 6.7.

Once the factors and their corresponding levels have been chosen, it is now time to look at the Taguchi orthogonal array to decide how many experiment need to be

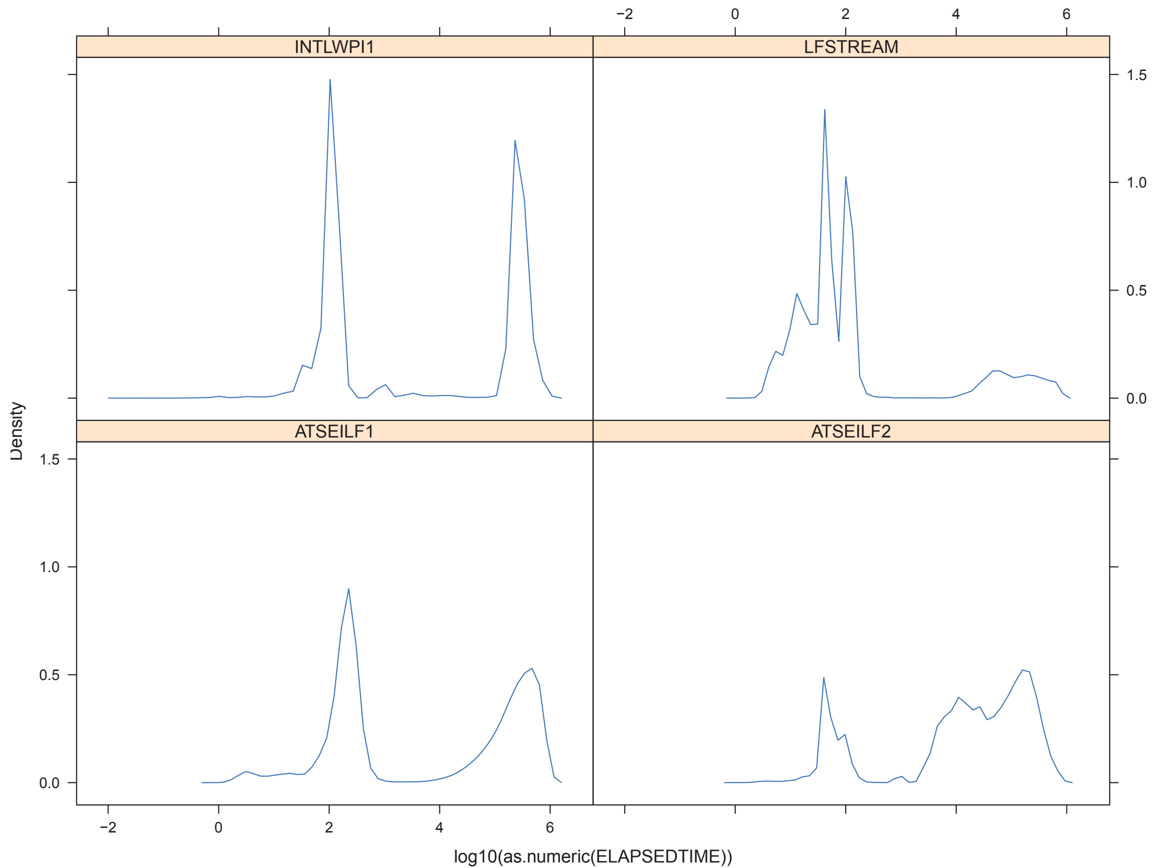


Figure 6.4. Data Distribution of Incoming Transactions for Four Airline Shopping Workload Types. The X-axis represents the logarithmic scale of base 10 and Y-axis represents the kernel density function of the data.

run. The orthogonal array where the rows and columns depict the factors and levels respectively is shown in Figure 6.6 taken from [42].

We have eight factors each with five levels (with the exception of payload types that has four). Looking up the corresponding row and column for our setup, L50 is the standard array that will be used to specify the combinations of the levels for the various factors that will ultimately be used to run the tests. The L50 standard array is as shown in Table 6.8. The L50 is defined for twelve factors with five levels

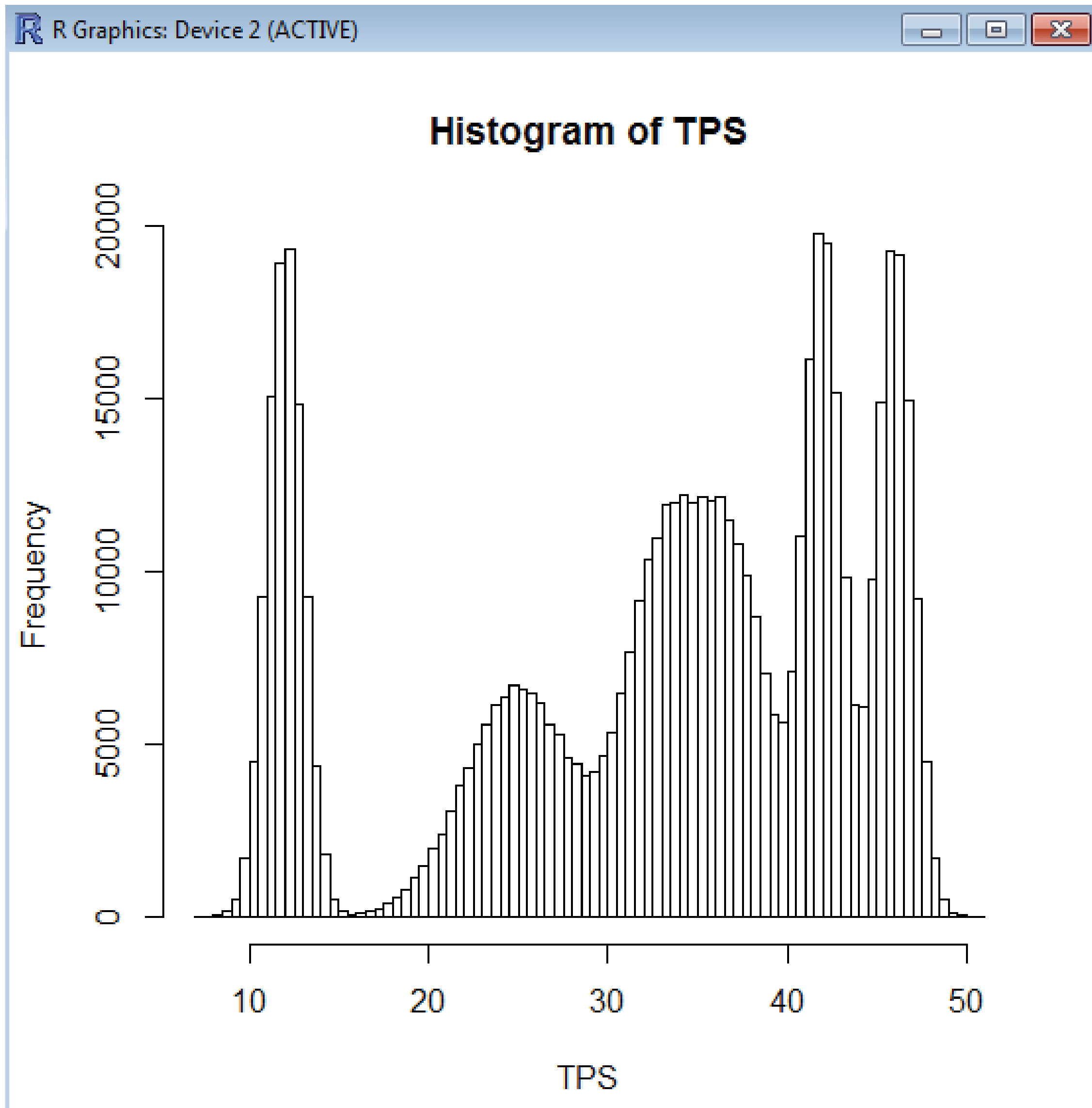


Figure 6.5. The Frequency Histogram for Airline Shopping Application TPS.

each but since in our scenario we have only eight factors, we will use the first eight columns of the array.

This is step one of setting up the tests to evaluate performance of a backend software application. In the next chapter, we will learn about Markov models and then put everything together to run tests for an airline shopping application. The

Table 6.4. TPS and Workload Type Levels for Performance Evaluation DOE

TPS	
Levels	15
	25
	35
	42
	48

Workload Type	
Levels	INTLWPI1
	LFSTREAM
	ATSEILF1
	ATSEILF2

Table 6.5. Queue Size and Timeout Levels for Performance Evaluation DOE

Queue Size	
Levels	500
	1000
	1050
	2000
	2050

Timeout Value [seconds]	
Levels	10
	12
	15
	18
	20

Table 6.6. Throttle and Payload Size Levels for Performance Evaluation DOE

Throttle Threshold	
Levels	20
	25
	30
	45
	50

Payload Size [Kb]	
Levels	20
	35
	45
	65
	75

Table 6.7. DB and Application Thread Pool Levels for Performance Evaluation DOE

DB Threads	
Levels	100
	200
	300
	400
	500

App. Threads	
Levels	100
	200
	300
	400
	500

Number of Levels	Number of Parameters (P)																													
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	L4	L4	L8	L8	L8	L8	L12	L12	L12	L12	L16	L16	L16	L16	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32	L32
3	L9	L9	L9	L18	L18	L18	L18	L27	L27	L27	L27	L27	L36	L36	L36	L36	L36	L36	L36	L36	L36	L36								
4	L'16	L'16	L'16	L'16	L'32	L'32	L'32	L'32	L'32																					
5	L25	L25	L25	L25	L25	L50	L50	L50	L50	L50	L50																			

Figure 6.6. The Taguchi Othogonal Array Selector.

test cases defined using L50 in this chapter will be used as an input for the simulation setup described in Chapter 8 section 8.6.

Table 6.8: L50 Standard Orthogonal Array to Decide Combinations of Levels for Factors

Experiment No.	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**F10**	**F11**	**F12**
1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	2	2	2	2	2	2	2	2	2	2
3	1	1	3	3	3	3	3	3	3	3	3	3
4	1	1	4	4	4	4	4	4	4	4	4	4
5	1	1	5	5	5	5	5	5	5	5	5	5
6	1	2	1	2	3	4	5	1	2	3	4	5
7	1	2	2	3	4	5	1	2	3	4	5	1
8	1	2	3	4	5	1	2	3	4	5	1	2
9	1	2	4	5	1	2	3	4	5	1	2	3
10	1	2	5	1	2	3	4	5	1	2	3	4
11	1	3	1	3	5	2	4	4	1	3	5	2
12	1	3	2	4	1	3	5	5	2	4	1	3
13	1	3	3	5	2	4	1	1	3	5	2	4

Continued on Next Page…

Table 6.8 – Continued

Experiment No.	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
14	1	3	4	1	3	5	2	2	4	1	3	5
15	1	3	5	2	4	1	3	3	5	2	4	1
16	1	4	1	4	2	5	3	5	3	1	4	2
17	1	4	2	5	3	1	4	1	4	2	5	3
18	1	4	3	1	4	2	5	2	5	3	1	4
19	1	4	4	2	5	3	1	3	1	4	2	5
20	1	4	5	3	1	4	2	4	2	5	3	1
21	1	5	1	5	4	3	2	4	3	2	1	5
22	1	5	2	1	5	4	3	5	4	3	2	1
23	1	5	3	2	1	5	4	1	5	4	3	2
24	1	5	4	3	2	1	5	2	1	5	4	3
25	1	5	5	4	3	2	1	3	2	1	5	4
26	2	1	1	1	4	5	4	3	2	5	2	3
27	2	1	2	2	5	1	5	4	3	1	3	4
28	2	1	3	3	1	2	1	5	4	2	4	5
29	2	1	4	4	2	3	2	1	5	3	5	1
30	2	1	5	5	3	4	3	2	1	4	1	2
31	2	2	1	2	1	3	3	2	4	5	5	4
32	2	2	2	3	2	4	4	3	5	1	1	5
33	2	2	3	4	3	5	5	4	1	2	2	1

Continued on Next Page...

Table 6.8 – Continued

Experiment No.	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
34	2	2	4	5	4	1	1	5	2	3	3	2
35	2	2	5	1	5	2	2	1	3	4	4	3
36	2	3	1	3	3	1	2	5	5	4	2	4
37	2	3	2	4	4	2	3	1	1	5	3	5
38	2	3	3	5	5	3	4	2	2	1	4	1
39	2	3	4	1	1	4	5	3	3	2	5	2
40	2	3	5	2	2	5	1	4	4	3	1	3
41	2	4	1	4	5	4	1	2	5	2	3	3
42	2	4	2	5	1	5	2	3	1	3	4	4
43	2	4	3	1	2	1	3	4	2	4	5	5
44	2	4	4	2	3	2	4	5	3	5	1	1
45	2	4	5	3	4	3	5	1	4	1	2	2
46	2	5	1	5	2	2	5	3	4	4	3	1
47	2	5	2	1	3	3	1	4	5	5	4	2
48	2	5	3	2	4	4	2	5	1	1	5	3
49	2	5	4	3	5	5	3	1	2	2	1	4
50	2	5	5	4	1	1	4	2	3	3	2	5

CHAPTER 7

STATE SPACE MODELS - MARKOV MODELS FOR PERFORMANCE EVALUATION OF SOFTWARE APPLICATIONS

7.1 Introduction

Dynamic computer systems like the ones were are considering in this study can be represented in an efficient manner by Markov chains/graphs. They provide a means to mathematically describe these systems as well as have standard associated algorithms and methods to analyze them. These processes are especially useful in deriving performance and dependability measures for these systems. A Markov chain comprises of states and the changing from one state of the system to another is called a transition. There is a unique probability associated with changing from one state to another and that is termed as the transition probability. Therefore in general a Markov process can be defined as consisting of set of states along with a transition matrix that lists the probabilities of transitioning from one state to another. The following sections contain some background knowledge on Markov models. This common literature is available in numerous text books like [43].

7.2 Background - Markov Models

The fundamentals of queuing theory also form the basic concepts of Markov models. Both fields of study share the underlying mathematical models and theorems. Any Markov process can be defined by defining the underlying stochastic process. A stochastic process helps with defining a relationship between elements in a possibly

infinite set of random elements. A series of random experiments can then be run on the set and results analyzed to make decisions on the set as a whole.

Definition: A stochastic process can be described as $X_t : t \in T$ where each random element X_t is indexed by a time parameter $t \in T$ such that $T \subseteq R_+ = [0, \infty)$. The state space of the stochastic process can thus be defined as the set of all possible values X_t (for each $t \in T$).

When dealing with a discrete parameter set T, the stochastic process is called a discrete parameter process and T is represented by a subset of $N_0 = 0, 1, ...$; otherwise the process is called a continuous parameter process. The state space of the resulting stochastic process when T is either discrete or continuous can also be discrete or continuous respectively.

Definition: A stochastic process (discrete or continuous) $X_t : t \in T$ can be termed a Markov process (discrete or continuous if for all $0 = t_0 < t_1 < ... < t_n < t_{n+1}$ and all $s_i \in S$ the conditional CDF (Cumulative Distribution Function) of $X_{t_{n+1}}$ depends only on the last previous value X_{t_n} and not on the earlier values $X_{t_0}, X_{t_1}, ...X_{t_{n-1}}$:

$$P(X_{t_{n+1}} \leq s_{n+1} | X_{t_n} = s_n, X_{t_{n-1}} = s_{n-1}, ...X_{t_0} = s_0) = P(X_{t_{n+1}} \leq s_{n+1} | X_{t_n} = s_n) \tag{7.1}$$

Definition: A time homogeneous Markov process is one where it's conditional CDF of $X_{t_{n+1}}$ does not depend on the observation time, that is $t_0 = 0$

$$P(X_{t_{n+1}} \leq s_{n+1} | X_{t_n} = s_n) = P(X_{t_{n+1} - t_n} \leq s_{n+1} | X_0 = s_n) \tag{7.2}$$

7.2.1 Discrete Time Markov Chains

Equation (7.2) describes an important property of a Markov chain called the Markov property. This property basically states that the entire history of a Markov

chain is summarized in the current state X_{t_n}. This implies that given the present state of the chain, a future state is conditionally independent of the past state/s.

Definition: A discrete time markov chain (DTMC) can be defined to consist of an underlying stochastic process $X + 0, X_1, ..., X_{n=1}, ...$ at the consecutive points of observation 0,1,...,n + 1 if the following Markov property holds for all $n \in N_0$ and all $s_i \in S$:

$$P(X_{n+1} = s_{n+1}|X_n = s_n, X_{n-1} = s_{n-1}, ..., X_0 = S_0) = P(X_{n+1} = s_{n+1}|X_n = S_n) \tag{7.3}$$

Given an initial state s_0, DTMC evolves over time according to the transition probabilities. The right hand side of the equation (7.3) defines the conditional pmf (probability mass function) of transitions from state s_n at time step n to state s_{n+1} at time step (n+1). So if S = 0,1,2,.., the conditional pmf of the process's one step transition from state i to state j at time n can be written as:

$$p_{ij}^{(1)}(n) = P(X_{n+1} = s_{n+1} = j|X_n = s_n = i) \tag{7.4}$$

In the case of a homogeneous process, the conditional pmf is independent of time steps and therefore equation (7.4) reduces to:

$$p_{ij} = p_{ij}^{(1)} = p_{ij}^{(1)}(n) = P(X_{n+1} = j|X_n = i) = P(X_1 = j|X_0 = i), \forall n \in T \tag{7.5}$$

Keeping the above definitions in mind, if we start from an initial state i, the probability that the DTMC will go to some state j (including the possibility of j = i), follows that $\sum_i p_{ij} = 1$ where $0 \leq p_{ij} \leq 1$. The resulting one step transition probabilities p_{ij} for the model can be written as a matrix P:

$$P = [p_{ij}] = \begin{pmatrix} p_{00} & p_{01} & p_{02} & \cdots \\ p_{10} & p_{11} & p_{12} & \cdots \\ p_{20} & p_{21} & p_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Generally a directed graph is used to visually represent a finite-state DTMC, where i of the chain is depicted by a vertex and a one step transition from state i to j by an edge marked with the one step transition probability p_{ij}.

Example: As an example from [43] consider the transition probability matrix shown below with state space S = {0,1}.

$$P = \begin{pmatrix} 0.75 & 0.25 \\ 0.5 & 0.5 \end{pmatrix} \tag{7.6}$$

According to the matrix (7.6), an event at any given point of time can transition the DTMC from state 0 to state 1 with a probability of 0.25. Similarly a state transition can occur from state 1 to state 0 with a probability of 0.5 at the next time step. The probability that the DTMC stays in state 0 or state 1 at any given time step is 0.75 and 0.5 respectively. These state transitions for the DTMC can be represented as a directed graph as shown in Figure 7.1.

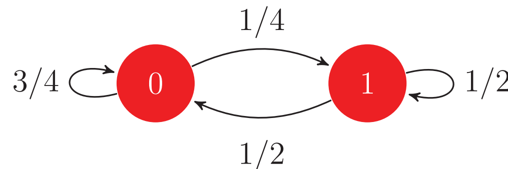


Figure 7.1. Example of a discrete-time Markov chain referring to Equation (7.6).

A n-step transition probability can be derived by repeatedly applying one-step transitions. If we describe the probability that the Markov process $p_{ij}^{(n)}(k,l)$ transitions from state i at time k to state j at time l in exactly n = l - k steps, the following can be derived:

$$p_{ij}^{(n)}(k,l) = P(X_l = j | X_k = i), 0 \leq k \leq l \tag{7.7}$$

The theorem of total probability can be applied to any given state i and any given time values k and l such that $\sum_j p_i j^{(n)}(k,l) = 1$ where $0 \leq p_i j^{(n)}(k,l) \leq 1$. Keeping this in mind and combining it with the Markov property, the n-step transition probability can be calculated by recursively applying the one-step transition probabilities. The state transitions of Markov process can be split into sub transitions. For example a transition from state i at time k to state j at time l divided into transitions from state i at time k to an intermediate state h at time m and from this independent state h to state j at time l, where $k<m<l$ and $n = l - k$. This condition leads to the well known system of Chapman-Kolmogorov equations:

$$p_i j^{(n)}(k,l) = \sum_{(h \in S)} p_{ih}^{(m-k)}(k,m) p_{hj}^{(l-m)}(m,l), 0 \leq k<m<l \tag{7.8}$$

In this equation (7.8), the Markov property, is the product of the terms on the right hand side. Just as in the one-step transition probability case, the n-step transition probability can also similarly be reduced since $p_i j^{(n)} = p_i j^{(n)}(k,l)$ depends only on the difference $n = l - k$ and not the actual values of k and l:

$$p_i j^{(n)} = P(X_{k+n} = j | X_k = i) = P(X_n = j | X_0 = i), \forall k \in T \tag{7.9}$$

7.2.2 Calculating State Transition Probabilities

So now that we've seen how a Markov process can be mathematically defined and how the states transition from one to another based on a probability, in this sub section we will see how to calculate these transition probabilities. We will extend the example from section 7.2 about rainy and sunny days. So let's assume we are trying to build a state space model for the days of the week. The SSM has two states - Rainy and Sunny. Figure 7.2 shows a graphical version of the SSM.

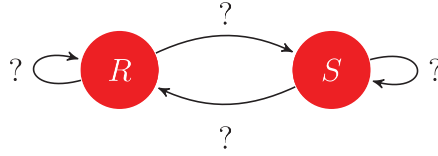


Figure 7.2. State Space Model for Rainy and Sunny Days in a Week.

Let's assume that the following sequence for the days of the week:

R S S S R S R

then the following transition probabilities can be calculated looking at the sequence:

$P(R|S) = 2/2 = 1$,

$P(R|R) = 0$,

$P(S|R) = 2/4 = 0.5$ and

$P(S|S) = 2/4 = 0.5$

and the state space model can can now be redrawn with the respective transition probabilities as shown in Figure 7.3:

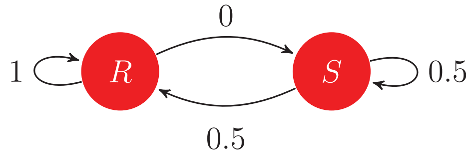


Figure 7.3. State Space Model for Rainy and Sunny Days in a Week with Transition Probabilities.

7.3 State Space Models for Evaluating Performance of Software Applications

The information in the previous sections can be applied to software applications. A piece of code performing a particular functionality can be considered as going through different states in order to successfully process incoming transactions. The

software's state can be defined in terms of its resource utilization metrics and the time it takes to process a particular transaction. The state will be represented by a vector of numeric values - each a measure of the various resource utilization metrics and the response times. So the state of an application at any point of time t will be represented as $S_t = <S_1t, S_2t, ...S_nt>$ where n is the total number of numeric values used to measure the state of the application. For any application running in a production environment - these values come from monitors already in place that instrument the code at all times. For the travel applications that are being considered in this study, we are using the Enterprise Instrumentation API to get these metrics.

State space models especially Markov chains/graphs are a popular choice for finding anomalies in computer system networks and also for predicting throughput and capacity. There most common usage is to mine for data when large volumes of information is available and it becomes difficult for a manual inspection of that data to identify anomalies. A similar thought process can be applied to software applications. Any production deployed application is instrumented extensively to monitor it's operation and health. It is absolutely acceptable for an application to be generating Gigabytes of data in a day. Though measures like file rotation are always put in place to avoid the risk of running out of disk space on a server to ensure that the application processing is never affected. To correctly predict the performance of a middle tier application (not client interfacing application), it becomes an even more tedious job. In order to fulfill a task these applications need to interact with other pieces of code, middleware components, use different communication protocols, transform payloads, talk to databases and so on to process transactions. In a live production environment, the input to a system as well as it's output is always unpredictable. All applications are thoroughly assessed using performance and functional

tests - the environments in which these tests are performed are usually scaled down versions of the production environment.

7.4 Related Work

Numerous research papers exist in the area of finding pressure points or bottlenecks in application using different techniques. Markov models or chains are a popular choice for this purpose as well as to predict performance for standalone applications. One of the disadvantages of using of a Markov chain is that it can contain an infinite number of states when simulating a real world application. Tremendous amount of work has been done to reduce the state map of applications. This becomes even more critical when dealing with big multifaceted pieces of code. Using hidden Markov models (HMM) is one way to achieve the goal. The authors of [44] using HMM to create the first state of the model and then split the new states that are created later by splitting existing states. HMMs are Markov chains where the state is not directly observable [45]. There are several other approaches to split states one of which is successive state splitting (SSS) and is used by authors in [46]. Another technique to restrict the number of states created in a model is to aggregate states where new states are not created if no new state representative factors are found.

Research work has also been done to automate the creation of Markov models. [47] describes a way to dynamically create a Markov chain where meeting certain criteria leads to duplicating the states and their transitions. This technique can lead to a larger number of states but the state representatives remain the same until a new state representative factor is identified. Another algorithm proposed in [48] is called the Augmented Markov Model (AMM). AMM is a method where new states are created only when the incoming data is identified as new. The transition probabilities

then adjusted based on the new states. After the model is created dynamically in this manner, it is not updated in the prediction phase.

As stated earlier one of the major applications of Markov chains is to detect anomalies in large volumes of data. Two of the most commonly used techniques are supervised and unsupervised and are described in [49]. Supervised methods make use of machine learning algorithms on a set of pre-classified data. The data is then categorized as either normal or as an anomaly. The objective of the supervised algorithms is to achieve a high recall percentage of both of these classifications. When using these algorithms there is a possibility of imbalance occurring in the two categories. One of the ways to alleviate the imbalance is by using a different sampling scheme to change the data distribution. Over [50] and under [51] sampling are the two most commonly used techniques for this. The authors in [52] combine the two methods and propose SMOTE - an optimized technique that combines both over and under sampling. Then a boosting method is applied by the authors in [53] to further improve the technique by applying optimization to the weighted classifier. There are several cost and scoring algorithms that further try to minimize cost and errors of these techniques [54, 55, 56]

The authors of [57] use Markov models to detect an anomaly - cyber attack in a computer system. The Markov techniques are used to profile the operations of the system by looking at the historical data. The paper presents a learning and inference algorithms that takes into account the observations of the system as time goes on and then computes the probability of how close the observations match the normal profile of the system. The lower the probability of the greater the chance that the observed activity is an anomaly, which in this cases implies a cyber attack. [58] and [59] propose anomaly detection methods using hidden Markov models and Bayesian networks respectively. The techniques involve a training phase where ordering of

events is computed. When new activities are observed an inference phase occurs which decides if an anomaly has been detected or not.

The authors of [60] talk about evaluating software performance using it's architecture. The underlying assumption is that the a particular architecture effects the quality of the software. The quality attributes are nothing but performance attributes of the application and this paper uses Discrete Time Markov Models to compare different architectural styles and then use response time to assess the performance of the software. The batch-sequential, parallel , fault tolerant and call and return architecture styles are modeled using DTMC. The model enables to count the number of times a particular state is visited in the model which in turn enables the calculation of the response time at each of these states in the software system. Since response time is a quality attribute that specifies how well a piece of software is performing, this process is able to assess performance.

[61] proposes a Markov model and queuing network based method to evaluate the system reliability, performance and availability. The solution takes into account different parameters such as software architecture, non functional attributes of components that make up the system, fault recovery strategies and the hardware. Specifically the model takes into account attributes like restarts, retries, reboots and repairs in the system. The model is created using deployment information for the software, machine specifications, architecture and client workload data. Just like in [60], the processing time is calculated using the number of times a particular node in the model is visited, restart overhead and availability of the underlying hardware.

[62] and [63] use DTMCs to create a vector space of formal languages and propose a metric based solution that identifies the total variation of new observations as compared to the language. Each new observation is a new state that is marked by a real value according to the design specifications of the implementer and his/her

perception of the state's impact on the overall performance on the software. The states are marked good or bad depending on the cost associated with the event that created the state. The DTMC then tries to disable as many bad states as possible, in order to enhance the performance of the system. [64] proposes a semi-Markov model to evaluate performance of a software application which is an extension of the solution proposed in [62] and [63].

The solution proposed in this study (described in the next chapter) differs from these existing methods in that, instead of assessing how well the software is working it actually tries to identify bottlenecks in the system that may degrade performance. Current Markov models used for performance evaluation, depend on either the underlying architecture/hardware or predefined costs are associated with the states being created in order to mark them good or bad. The model is created based on historical data and any major change in the new states being created as compared to the historical data, causes the model to mark the states as an anomaly. In our proposed solution the bottlenecks are not detected by creating the model beforehand. Client side input (traffic) along with attributes of the software itself as well as it's interacting components is taken into account to detect pressure points. The method is not dependent on the server it is running on and/or the architecture and so is more flexible to be adapted for different kinds of applications. The other major difference is that in our proposed solution we are not using the application's historical data/profile to detect pressure points. We've seen from the previous chapters, that the application's profile changes from one release to another because of various reasons - new functionality has been added or it supports a new traffic type and so on. So a deviation of historical data is not always the best way to detect an anomaly. In our solution, the pressure points are being identified based on the varying combination of input traffic and other configuration and component parameters.

CHAPTER 8

EVALUATING PRESSURE POINTS IN A COMPUTER SYSTEM FOR END TO END PERFORMANCE EVALUATION

8.1 Introduction

Previous chapters have covered the different aspects of performance testing. The major goal of testing software before it is deployed to avoid any issues that might impact the end customer. Any kind of incident that can cause disruption of service is detrimental to business. This is especially true for the travel industry. Computer systems that cater to airlines, business and leisure travelers, hotel suppliers and cruise companies need to be up and running 24X7, 365 days a year. There is no acceptable downtime for these systems. Airlines in various parts of the world are dependent on their day to day operations on these computer systems. Travelers need to access flight/hotel/car/rail and cruise information at any time of the day. Flights departing on time, passengers being able to checkin, security checks, crew control, air traffic control, landing gate assignments and more are all dependent on the availability of these systems. It therefore becomes even more critical to establish stable and reliable systems in production and minimize any disruption to services that cater to the different needs of airlines and passengers alike.

In order to provide any service to the end customer - be it an airline operations team, an airline gate agent, a travel agent or a traveler - multiple systems have to work together to process any single kind of transaction. Figure 8.1 shows an example of a part of such a system. The figure is a simplified illustration of how the self service checkin process happens.

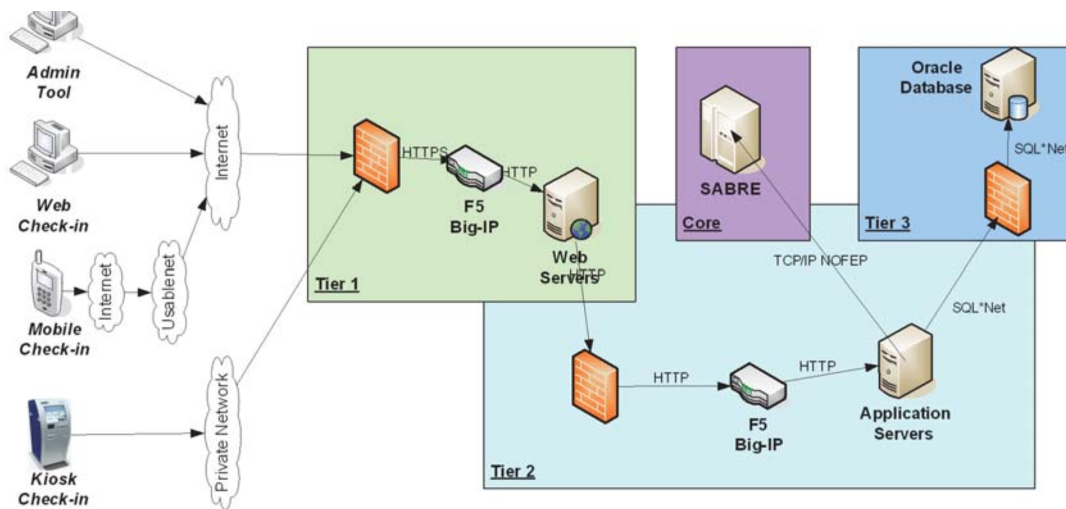


Figure 8.1. Different Systems Work Together to Enable Self Service Checkin for Travelers.

A passenger has an option to checkin using his/her mobile device, a web interface or a kiosk at the airport. The incoming transactions from any of these interfaces needs to go through a tier one firewall and a load balancer as well as a web server that contains the business logic to either reject or forward the transaction to the tier two systems. Tier two contains the main application servers with an instance of each web, mobile and kiosk to process the incoming transactions from the specific interface respectively. The load balancer in tier two will direct the traffic to the least busy server. The application servers talk to the database to store the incoming transactions and then forward it to the main TPF/mainframe system to process the passenger checkin request. It is in the mainframe system that the PNR (Passenger Name record) is retrieved, security checks are done and the seat map algorithm assigns the seat and prints the boarding pass.

So as can be seen from this example to seamlessly checkin a passenger for a particular flight, several components and systems come into play. Figure 8.2 shows

that every incoming and outgoing communication link to and from a system is a potential pressure point.

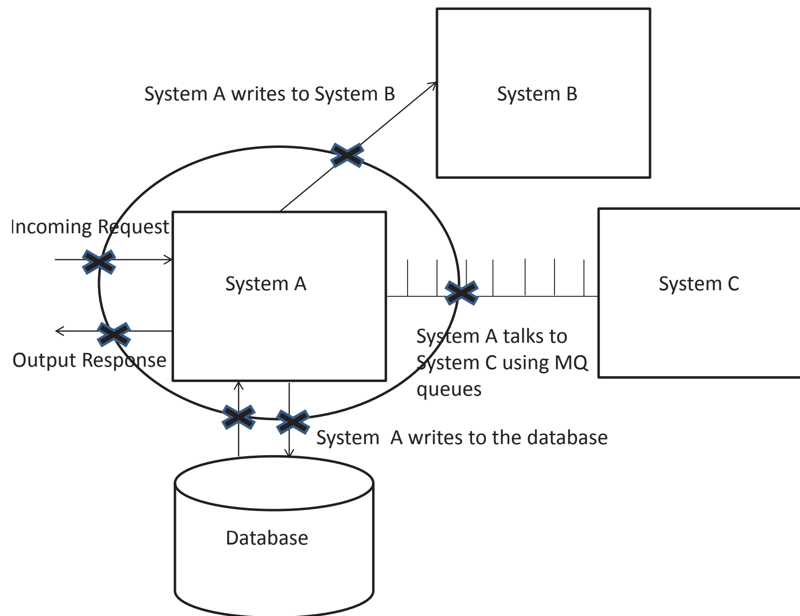


Figure 8.2. Performance bottlenecks for an application can be found at each communication link (incoming and outgoing) with other components.

The situation amplifies when we have multiple systems interacting with each other to process a particular transaction. It is therefore critical to not only test applications standalone but to also determine how well they operate in conjunction with other applications in order to fulfill a business functionality. More often than not due to test environment or resource and/or time constraints, performance testing becomes an after thought. Even when testing is targeted, it is usually executed on the application standalone. Conventional tests like soak, load, capacity and stress tests that rely on varying incoming traffic levels to find performance bottlenecks, fail to push the boundaries of the system under test. This chapter will describe how this can be avoided and how the system under test can be pushed to operate in scenarios

that are outside the normal processing boundaries. The next section will throw light on how to identify pressure points in systems.

As mentioned in Chapter 6, the goal of this part of the dissertation is to evaluate the performance of a backend software application. This implies we want to assess performance based on factors that take into account the workings of the application under test as well as the applications it is interacting with and any other middleware component that it is using to successfully complete processing of an incoming transaction. We have already chosen a pilot - airline shopping application and we have already done part of the work, which is to identify the factors and their levels that will make up the test cases in Chapter 6. In Chapter 7, we've studied about state space models specifically Markov models and now in this chapter we will see how we can put together this knowledge and come up with algorithms that will enable the assessment of application performance along with detection of pressure points/bottlenecks. So to enumerate what has been accomplished so far:

1. Factors were picked that could affect the application's performance
2. Looking at the production data, appropriate levels were chosen for each of the factors
3. Since we ended up with 8 factors with 5 levels each, we used the Taguchi Orthogonal Array DOE method to setup our test cases
4. The L50 array in the standard Taguchi orthogonal array corresponded to our factors and levels
5. So 50 test cases were setup according to the L50 array
6. An airline shopping application was chosen as the pilot application

So we have the input(test cases) ready for the performance tests and we've seen how Markov models work. Now in this chapter we will see what are pressure points and

then utilize the knowledge learnt in Chapter 7 to create algorithms using SSMs that will help evaluate a backend application's performance and detect bottlenecks.

8.2 Background - What is a Pressure Point or a Bottleneck?

This particular section will describe the characteristics of pressure points and their analysis. This is common text book knowledge and can be found in any software performance analysis book like [65]. A pressure point is another name for a bottleneck. In a distributed multi-tiered environment these bottlenecks could be either hardware or software related. In this chapter we will be concentrating on software related pressure points. After the rigorous functional and non functional testing it is rare to find performance issues in the software code standalone. Problems occur when multiple pieces of code, middleware components, databases and different communication protocols interact in order to complete a business functionality.

The basic concepts involved with hydrodynamics are also applicable to software systems. So in order to study what a pressure point is, we will look at examples from hydraulics. The capacity of a software application can be compared to a series of pipes of varying capacity. The output from one pipe is the input to another which is similar to two interacting pieces of code. For example in Figure 8.3, the pipe with the most restrictive flow (number two) is the bottleneck for the entire system. It is this limited capacity of pipe number two establishes the total amount water that can flow through the system which is finally the capacity for the end to end system.

So in order to identify a bottleneck, the simple logic shown in Figure 8.4 from [65] can be applied. It helps identify which pipe/system is able to or unable to handle the incoming input capacity.

This same concept can be applied to software applications. More often than not bottlenecks occur because one system is processing faster than another and is either

Figure 8.3. The Pipes in Hydraulics are Similar to Software Application's Capacity Measurements.

waiting for a response from a slow upstream or downstream application or is flooding the slower system. So any increase in end user response times and/or timeouts due to processing slowdown at any tier is a bottleneck. A bottleneck can occur because of increase in volume and/or an event that could indirectly cause capacity issues. And this is one of the major shortcomings of using existing performance tests that rely on finding bottlenecks by varying load. Keeping in mind the similarity between software and the study of hydraulics, the following list describes some of the most common characteristics of a pressure point.

1. Pressure Points Are Not Always Caused Because of Load:

   Figure 8.5 is another representation of the same kind of bottleneck as shown in Figure 8.3. In this figure, water is moving at a velocity of 1 in the larger diameter side of the pipe as compared to the smaller diameter side of the pipe (velocity of 4). The water in the larger diameter pipe has to wait till it gets to the smaller diameter part before it can flow at a higher velocity because of the principle of conservation of mass. This characteristic of the water to wait and flow at a lower velocity is the formation of a queue. This is an example of a queuing bottleneck. In software applications an indicator of this kind of a pressure point is when the code picking up messages is slower than the code putting in messages on the queue.

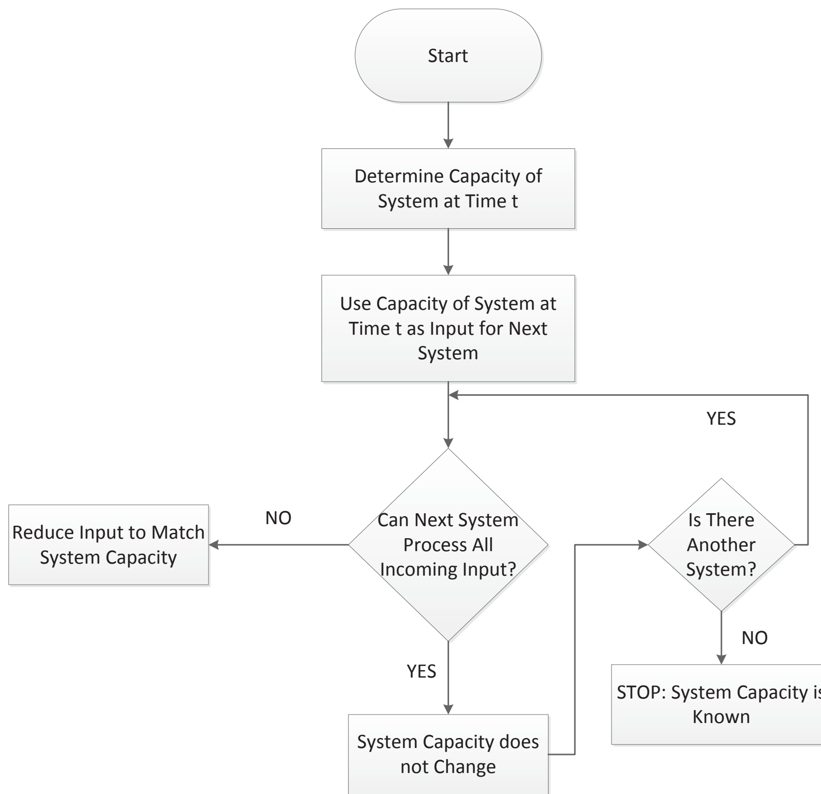


Figure 8.4. Steps for Detecting Bottlenecks in a System.

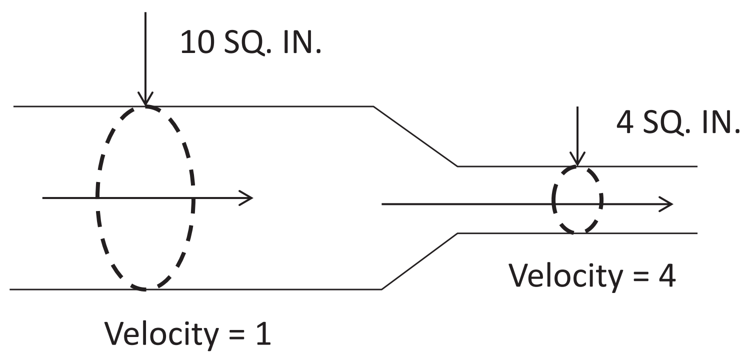


Figure 8.5. Example of a Pressure Point Due to Queueing.

2. Pressure Point Indicators Can be Observed Anywhere in the System:

   Critical bottleneck is another concept in hydrodynamics. It is defined as the one bottleneck if not resolved that dictates the characteristics of a system flow.

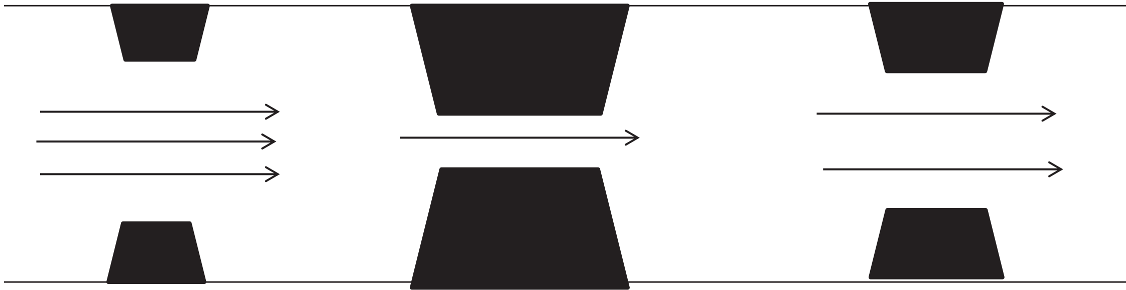


Figure 8.6. Example Showing a Critical Bottleneck.

   As can be seen from Figure 8.6, there are three sets of restrictions that are constraining the flow of water. Restriction number 2 is the critical bottleneck, because it is obstructing the water flow the most. This basically implies that removing the other restrictions will not help with the overall water flow unless and until the critical bottleneck is removed. This is also applicable to software systems. If there was one component that is taking longer to process incoming transactions, adding capacity to the other components in the system will not fix the issue, unless and until the slow component is fixed.

3. Different Paths Through the System Can Cause Different Pressure Points:

   Consider Figure 8.7 which is an example of a closed hydraulic system. The figure shows that one system could have multiple paths for the water to flow through.

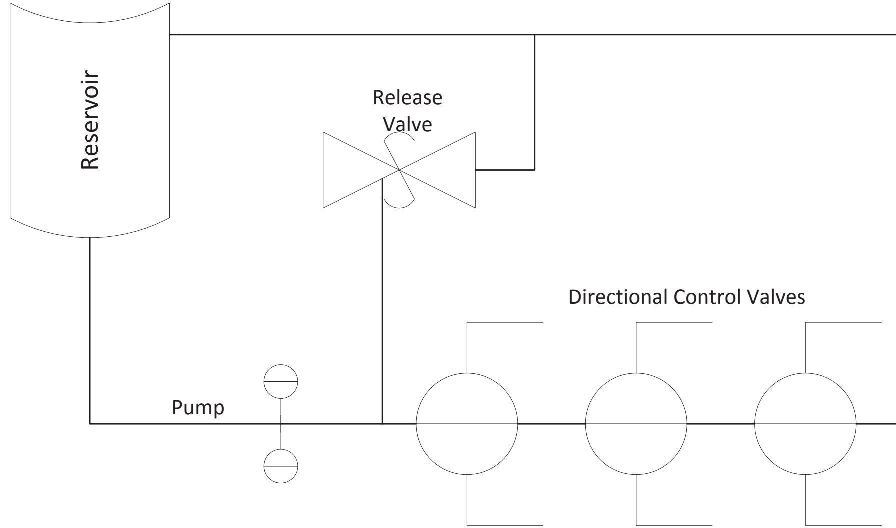


Figure 8.7. Different Paths Through the System can Cause Different Pressure Points.

The more paths there are in a system for the water to flow through, the more difficult it becomes to detect the bottlenecks plus the more points where bottlenecks could occur. This theory can be applied to software systems too. The more interacting applications there are in a system, the more places where pressure points could occur and the more complex it becomes to detect and troubleshoot them.

8.3 Pressure Points in Computer Systems

Identifying bottlenecks in hydraulics is very similar to finding bottlenecks in a piece of software code. In the system depicted in Figure 8.7, the various components like the pipe, a pump, valves and a reservoir can be compared to the computer network and other hardware items like web servers, routers, load balancers, application servers and so on. Just as shown in Figure 8.7, water can flow through different paths, in a computer system the incoming customer request goes through various systems and

paths to get processed. Each one of these paths could be a potential pressure point. In order to prevent disruption of service to the customer it therefore becomes necessary to study each and every path going in and out of a particular system. Figure 8.2 illustrates just this notion.

To understand the criticality of performing this task and how important it is in troubleshooting issues let's take a look at two real world travel applications where service disruption occurred not due to the standalone application itself but because of slow down of a backend interacting application and a database issue.

1. Example 1: IET - Interline Electronic Ticketing:

   This example describes the airline ticketing system. Multiple applications work in conjunction to perform the one task of issuing airline tickets to passengers. Figure 8.8 shows the various components apart from the software applications themselves that make up this piece of functionality.

   Consider just part of this system - Ticketing Hub (IET/TKTHUB) The TKTHUB System consists of one independent ServiceMix instance running in its own JVM. Data is input into the system from PSS (Passenger Service System), CTS (Common Translation Service) and WNP (External Reservation System) via various message queues provided by a set of MOM servers. Data is streamed out to internal and external customers via other MQs depending upon the destination. All data is persistently stored in an Oracle database cluster. The following context diagram 8.9 provides a high level view of all external interfaces to/from the TKTHUB ServiceMix instances.

   A new release of an interline electronic ticketing application is deployed in production after all functional and load tests have been successfully completed. The interline ticketing application sends transactions out to various suppliers to get data to process tickets for passengers that have booked flights on differ-

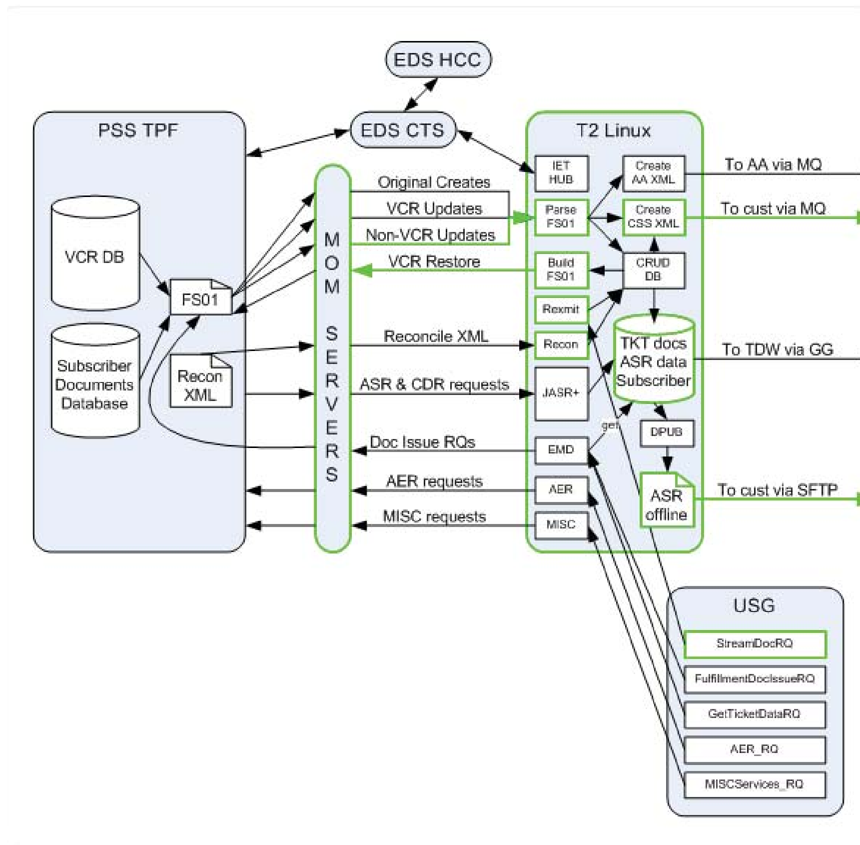


Figure 8.8. An example of an airline ticketing system.

ent airlines for the same itinerary. These messages are placed on a middleware IBM MQ queue and once the responses are received, the application reads the messages off of the queue and sends it out to the PSS system for ticketing purposes.

The application passes all average and peak load tests and is deployed in production but after its successful run for a few days, customers start observing long delays and ultimately timeouts for requests. These timeouts are not re-

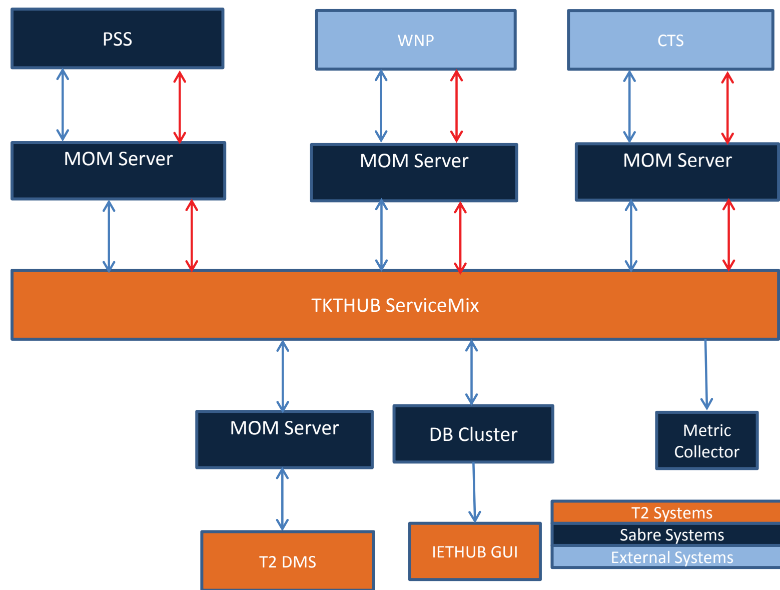


Figure 8.9. Context Diagram for Interline Electronic Ticketing (IET)/TKTHUB.

lated to an issue within the application itself but because of the slow down in a backend system like T2 DMS (Ticketing Data Mangement Service).

2. Example 2: PWS: Payment Web Services:

   Another example of a system - payment web services, that has been fully tested and deployed in production starts seeing timeouts for a particular mode of payment - pay me later, for a single airline - Vietnam airlines. Figure 8.10 shows the logical view of the sequence of steps needed to fulfill the pay me later option.

   For the pay me later mode of payment, the application is looking at the database tables to validate that the transaction does not previously exist. In the test environment where data volumes in the tables is small, all test cases dealing with this mode of payment passed. In production where these tables are huge, full table scans take time and transactions start to timeout. In the real world scenario, where the ticketing and payment processes are asynchronous, customers

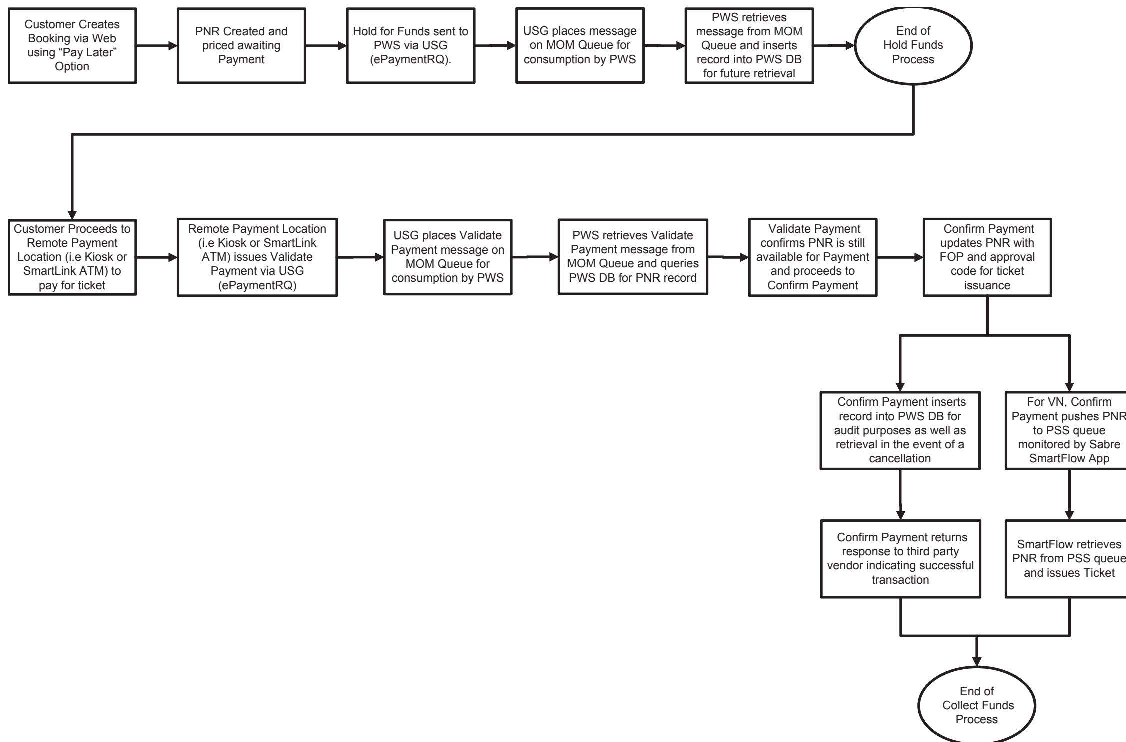


Figure 8.10. Sequence of steps for pay me later option for payment web service application.

are being issued tickets without payments being received. This has massive impact on business.

Software applications do not work standalone in production and are not dependent on just incoming TPS (transactions per second). They interact with various other components in the system and are impacted by different parameters. It is therefore of utmost importance to thoroughly test the software as well as how it communicates with other systems before it is deployed. It is essential to find out under what conditions the code will fail, what would be the impact of such a failure in order to design a fault tolerant system. A stable working environment is directly related to how well the business does in terms of cost and customer loyalty.

8.4 Characteristics and Behaviors of Software Application Pressure Points

The main objective is to evaluate and proactively predict the performance and profile of a software system that is dependent on multiple varying input parameters in a large scale distributed system. The applications considered would not be client interfacing web based applications but ones that interact with multiple components to process incoming transactions as shown in Figure 8.2. The purpose is to find pressure points for the application as well as all in communication links with other systems and/or middleware components and database. Pressure points are any application resource that can become exhausted thereby restricting or degrading service level performance. Examples of pressure points include: CPU, memory, disk, network, code loops, locks, file handles, stack/heap settings, buffers, threads, connections and so on. Performance is measured in terms of different SLAs.

A potential pressure point could be anywhere in the system. In order to troubleshoot, the first action is to identify where the bottleneck exists. It could be any of the following:

- Server/hardware itself
- Application code
- A middleware component
- Resource - operating system or configurable resource like database connection pool

Once we have identified the point of the bottleneck, we need to identify the behavior. In this study we will concentrate on application and resource utilization bottlenecks. Keeping in mind the similarities between the bottleneck characteristics between the two fields of study - hydraulics and software engineering, we can enumerate the following behaviors:

- Is the resource being shared?

1. Is the bandwidth between the two components of interest being shared?
2. Is one component utilizing more resources/bandwidth than the other?
3. What is the method of control? Is it peer to peer, token based or so on?
4. Is there data collision?
5. Is one more component faster/slower in processing than the other?
6. What happens if the sender is blocked or down?
7. What happens of the receiver is blocked or down?
8. Is the connection link between the two components broken?
9. What are different kinds of data flow routing methods?
   - Redirect e.g. Big-IP/F5
   - Load balancing e.g. Round robin routing of data
   - Allocation e.g. MQ based data routing
   - Blocking e.g. Firewall with rules for incoming transactions

These behaviors can be measured in terms of

1. the underlying resource utilization numbers
   - CPU
   - Memory
   - Disk
   - I/O
2. pool sizes for thread pools in terms of their
   - minimum value
   - maximum value
   - start value at the time of application initialization
   - increment level
   - number of active threads at a particular time
3. Service Level Agreement (SLA) numbers e.g. client side response times

8.5 End-to-End Performance Evaluation of Software Applications

Now that we've seen the characteristics of pressure points and identified how to detect them by looking at the different behaviors, we will now describe a method to discover bottlenecks in an application. We will use the concepts of state space models, specifically Markov models and experimental design algorithms as described in the previous two chapters. Markov models have been used as a solution to detect pressure points in the application because they provide an efficient way to identify anomalies (which in our case are the pressure points). As seen from section 7.4 in Chapter 7, a lot of work has been done to detect anomalies using Markov models. To summarize once again how this proposed solution differs from the existing work in that instead of assessing how well the software is working it actually tries to identify bottlenecks in the system that may degrade performance. Current Markov models used for performance evaluation, depend on either the underlying architecture/hardware or predefined costs are associated with the states being created in order to mark them good or bad. The model is created based on historical data and any major change in the new states being created as compared to the historical data, causes the model to mark the states as an anomaly. In our proposed solution the bottlenecks are not detected by creating the model beforehand. Client side input (traffic) along with attributes of the software itself as well as it's interacting components is taken into account to detect pressure points. The method is not dependent on the server it is running on and/or the architecture and so is more flexible to be adapted for different kinds of applications. The other major difference is that in our proposed solution we are not using the application's historical data/profile to detect pressure points. We've seen from the previous chapters, that the application's profile changes from one release to another because of various reasons - new functionality has been added or it supports a new traffic type and so on. So a deviation of historical data is not always the best way

to detect an anomaly. In our solution, the pressure points are being identified based on the varying combination of input traffic and other configuration and component parameters.

As described at length in Chapter 6, the DOE techniques specifically the Taguchi Orthogonal arrays have been used to run tests on the software application with varying combinations of levels for the selected factors. The following sub-section will describe the different algorithms that will help us create the application state model and identify states that can be termed as pressure points. Just as a reminder, in the cases of the airline shopping application (our pilot software application) states are deemed as pressure points/bottlenecks only if they exist for durations greater than 15 minutes.

8.5.1 Creating a Node Cluster

A state in the Markov model can also be called a node. In order to restrict the number of states/nodes that the application can transition to in the state space model (SSM), we will assume that each node belongs to a cluster. A cluster is a set of real world events/states. At any given point of time, t, if there exists clusters, each one of these clusters will be represented by a node termed $N_{centroid}$. This particular node is the centroid for the cluster and therefore will be used when calculating where a new state will be added to the model.

An event at time t, creates a new state/node S_{new}. The algorithm nodeInCluster will be used to place this new node/state into either one of the existing clusters in the model, SSM or form a new cluster. This decision is based on how close or far the new state is to the nodes in an existing cluster. The nearest neighbor clustering algorithm will be used to make this decision. A threshold will be chosen to decide

how close or far the new states are to the existing states in the clusters. The nearest neighbor clustering algorithm used in nodeInCluster is explained next.

The goal of the nearest neighbor clustering algorithm is to pair an instance x with a corresponding label y to create $<x, y>$. In our case the instance is the state S_{new} and the lables are the existing clusters in the SSM. So for any new instance without a label the algorithm needs to categorize instances with their labels. Assume that instance x is a member of the set X, while label y is a member of set Y. Thus the algorithm is nothing but a function $F : X \rightarrow Y$. So to draw a parallel to our case, the algorithm will try and place S_{new} in one of the clusters that already exits(depending on how close S_{new} is to the $N_{centroid}$ of the clusters) or create a new cluster.

We will be using the Euclidean distance function to calculate how close or how far S_{new} is from the existing $N_{centroid}$ of the different clusters. The Euclidean function to find the distance between two points x and y is as follows:

$$d(x, y) = \|xy\| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ...(x_n - y_n)^2} = \sqrt{(\sum_{i=1}^{n}(x_i - y_i)^2)}$$

Here the Cartesian coordinates for the points x and y are $(x_1, x_2, ...x_n$ and $(y_1, y_2, ...y_n$ respectively. This simple nearest neighbor clustering method will be used in our state space model to add new application output states to the closest cluster. This will prevent the resulting model from consisting of infinite number of states.

- Algorithm 1 - nodeInCluster:

Assume that at time t - 1, there exists a cluster C_{t-1} and there is a state space model/Markov model M for the application. It will become clear when looking at the createSSM algorithm next how the clusters are created. At time t, an event E_t occurs that changes the state of the application and a new state is created S_{new}. The centroid node in M that represents the cluster C_{t-1} is denoted by $N_{centroid}$. The decision to whether create a new cluster C_t at time t and add the new state to that new cluster or to add it to the existing cluster can be decided by the nodeInCluster algorithm

shown in Figure 8.11. According to the nearest neighbor clustering algorithm, the closer two nodes are to each other the smaller the distance between them. Keeping this in mind, an appropriate threshold - th is chosen when determining if a node should be added to an existing cluster or if it should form a new cluster.

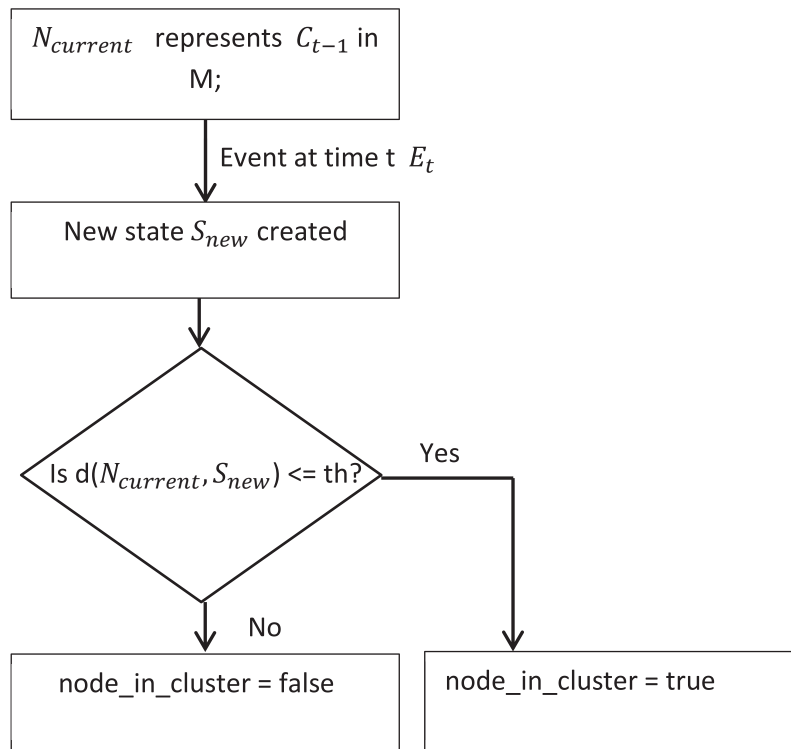


Figure 8.11. nodeInCluster Algorithm Decides Whether to Add a Node to an Existing Cluster or Form a New Cluster.

8.5.2 Creating the State Space Model

In this sub-section, we will look at an algorithm that will help us facilitate the creation of the state space model for our pilot application. The algorithm is general enough to be used for any software code.

- Algorithm 2 - createSSM:

So now that we've seen how the algorithm nodeInCluster determines where a new state/node should be put:

1. added to an existing cluster (if it exists) depending on how similar the new state is to the states in that cluster; calculated using the nearest neighbor clustering algorithm
2. or a new cluster created with the new state

let's see how the state space model gets created. Keeping in mind all the background information described in the previous chapter on Markov models on how the transition probabilities are determined and how new events create new states, createSSM will describe the process of building a state space model(SSM) for any software application. Once again assume that at time t - 1, there exists a cluster C_{t-1} and there is a state space model/Markov model M for the application. At time t, an event E_t occurs that changes the state of the application and a new output state is created S_{new}. The current node in M that represents the cluster C_{t-1} is denoted by $N_{centroid}$.

Figure 8.12 represents the steps in the algorithm. The output of the nodeInCluster algorithm is used to determine where to place the new state S_{new}. Once this new state is placed in a cluster, the size of the cluster is calculated. A new edge is created from the $N_{centroid}$ to S_{new}, if it doesn't exist. This implies a state transition and therefore the transition probability has to be calculated. The transition probability in this case is the number of times there is a transition from $N_{centroid}$ to S_{new} divided by the size of the cluster represented by $N_{centroid}$.

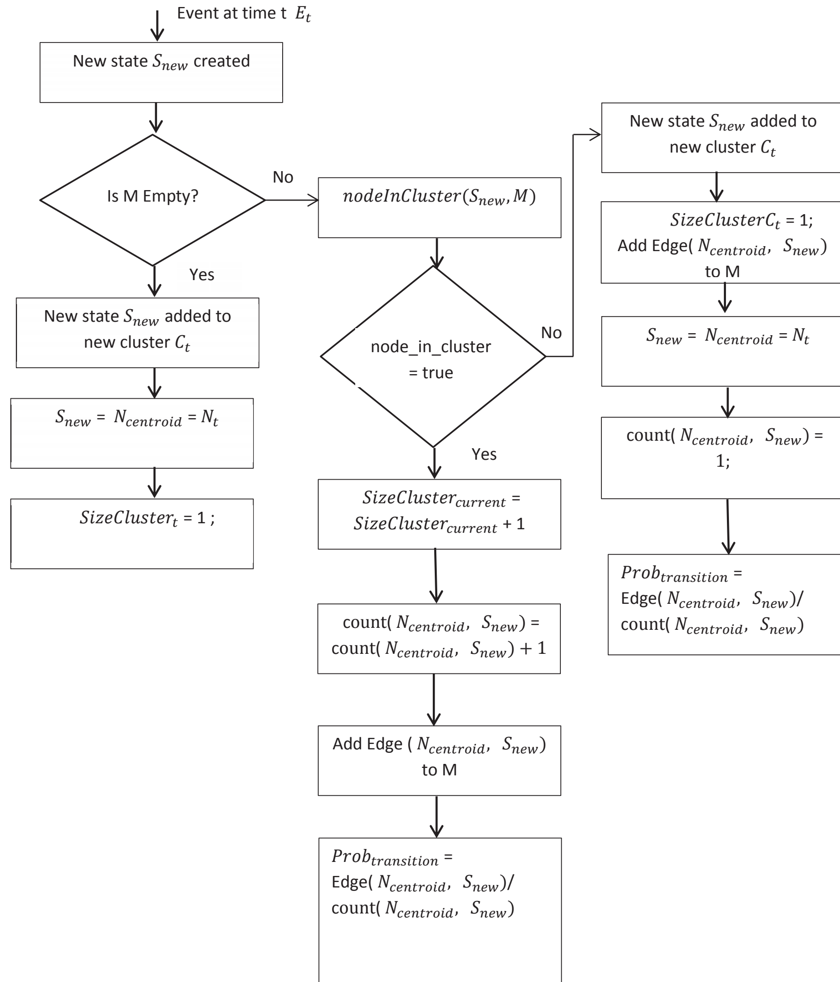


Figure 8.12. createSSM Algorithm to Build a State Space Model For a Software Application.

8.5.2.1 createSSM Example

Before we move onto the algorithm that will help us determine if the new state created in our model is a pressure point or not, let's take an example of how the createSSM algorithm works. We will take an example of a travel shopping application where the state of the application is described by the measure of it's response time,

CPU, memory, disk, I/O usage and number of core dumps. So keeping the same notation as mentioned in the previous chapter: $S_t = <S_1t, S_2t, ...S_nt>$ will basically imply that at any given time t, the state of the shopping application can be represented as $S_t = <respone - time_t, CPU_t, memory_t, disk_t, io_t, count(core - dumps)_t>$. According to the nearest neighbor clustering algorithm, the smaller the distance between the neighbors there is a greater possibility of them being in the same cluster. Keeping this in mind, we will choose a threshold value of 0.2 to compare distances between nodes and then decide which cluster they fall into.

Let's consider the states of the application as shown in Table 8.1:

Table 8.1. Sample State Space Data for Travel Shopping Application

Time	**State Space Vector**
Time 1	$<5.6, 2, 2, 0.5, 0.1, 0>$
Time 2	$<5.6, 2, 3, 0.7, 0.4, 0>$
Time 3	$<5.6, 2.1, 3, 0.7, 0.4, 0>$
Time 4	$<5.6, 2, 3, 0.5, 0.1, 0>$
Time 5	$<7, 2.2, 3, 0.7, 0.8, 0>$
Time 6	$<7, 2, 3, 0.7, 0.8, 0>$
Time 7	$<12, 3.6, 4, 2.1, 1.8, 0>$
Time 8	$<12.5, 3.8, 4, 2.5, 2, 0>$
Time 9	$<5.6, 2.1, 3, 0.7, 0.4, 0>$

So following the createSSM algorithm, the following steps will be implemented:

1. At time 1 the model is empty, so the state vector $<5.6, 2, 2, 0.5, 0.1, 0>$ will be the first node in the model, M. $SizeCluster_1 = 1$.
2. At time 2, a new event creates a new state vector $S_{new} = <5.6, 2, 3, 0.7, 0.4, 0>$. The distance between the $N_{centroid}$ and this new state will be calculated and compared to the threshold we've chosen. The distance between the two nodes needs to be calculated in order to either add the new state to the existing

cluster or create a new cluster. $d(N_{centroid}, S_{new}) = 1.063$. This is greater than the threshold chosen and hence the new state will not be added to the existing cluster but will be added to the model, M as a new node. $SizeCluster_2 = 1$ and $Prob_{transition}(N_1, N_2) = 1/1$

3. At time 3 there is a new state vector $<5.6, 2.1, 3, 0.7, 0.4, 0>$. The distance between the two existing clusters and the third node will be calculated to determine which cluster it will be a part of. $d(N_1, S_{new}) = 1.0676$ and $d(N_2, S_{new}) = 0.1$ which is less than our chosen threshold of 0.2 and hence this new state vector will be added to the cluster of N_2. So now $SizeCluster_2 = 2$ and $Prob_{transition}(N_2, N_2) = 1/2$

4. Similar processing will be done at each of the steps when a new state is created. $d(N_1, S_{new}) = 1$ and $d(N_2, S_{new}) = 0.3605$. Since both distances are greater than the threshold of 0.2, a new node will be created in M. $SizeCluster_3 = 1$ and $Prob_{transition}(N_2, N_3) = 1/2$.

5. New state vector $= <7, 2.2, 3, 0.7, 0.8, 0>$. $d(N_1, S_{new}) = 1.878$, $d(N_2, S_{new}) = 1.469$ and $d(N_3, S_{new}) = 1.5905$. A new node N_4 is added to M. $SizeCluster_4 = 1$ and $Prob_{transition}(N_3, N_4) = 1/1$.

6. New state vector $= <7, 2, 3, 0.7, 0.8, 0>$. $d(N_1, S_{new}) = 1.868$, $d(N_2, S_{new}) = 1.456$, $d(N_3, S_{new}) = 1.577$ and $d(N_4, S_{new}) = 0.2$. This new state is added to the cluster of node N_4. $SizeCluster_4 = 2$ and $Prob_{transition}(N_4, N_4) = 1/2$

7. New state vector $= <12, 3.6, 4, 2.1, 1.8, 0>$. $d(N_1, S_{new}) = 7.278$, $d(N_2, S_{new}) = 6.959$, $d(N_3, S_{new}) = 7.068$ and $d(N_4, S_{new}) = 5.5605$, so a new node N_5 will be added to M. $SizeCluster_5 = 1$ and the transition probability $Prob_{transition}(N_4, N_5) = 1/2$.

8. Distances for the new state vector $<12.5, 3.8, 4, 2.5, 2, 0>$ are as follows: $d(N_1, S_{new}) = 7.903$, $d(N_2, S_{new}) = 7.592$, $d(N_3, S_{new}) = 7.711$, $d(N_4, S_{new}) = 6.204$ and

$d(N_5, S_{new}) = 0.6999$, so a new node N_6 is added to M. $SizeCluster_6 = 1$ and the transition probability $Prob_{transition}(N_5, N_6) = 1/1$.

9. At time step 9, the new state vector is $<5.6, 2.1, 3, 0.7, 0.4, 0>$. $d(N_1, S_{new}) = 1.067$, $d(N_2, S_{new}) = 0.1$, $d(N_3, S_{new}) = 0.374$, $d(N_4, S_{new}) = 1.459$, $d(N_5, S_{new}) = 6.937$ and $d(N_6, S_{new}) = 7.569$, so the new state is added to the cluster N_2. $SizeCluster_2 = 3$ and $Prob_{transition}(N_6, N_2) = 1/1$.

Figure 8.13 is the end result of the above mentioned steps at each time interval.

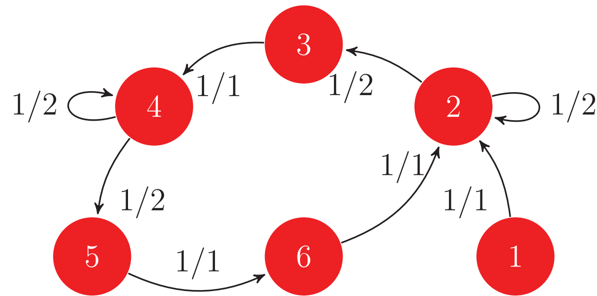


Figure 8.13. Resulting SSM for the createSSM Algorithm Example.

8.5.3 Finding Pressure Point States in the State Space Model

This sub-section will describe the algorithm that helps identify pressure point states in a software application.

- Algorithm 3 - findPressurePoint:

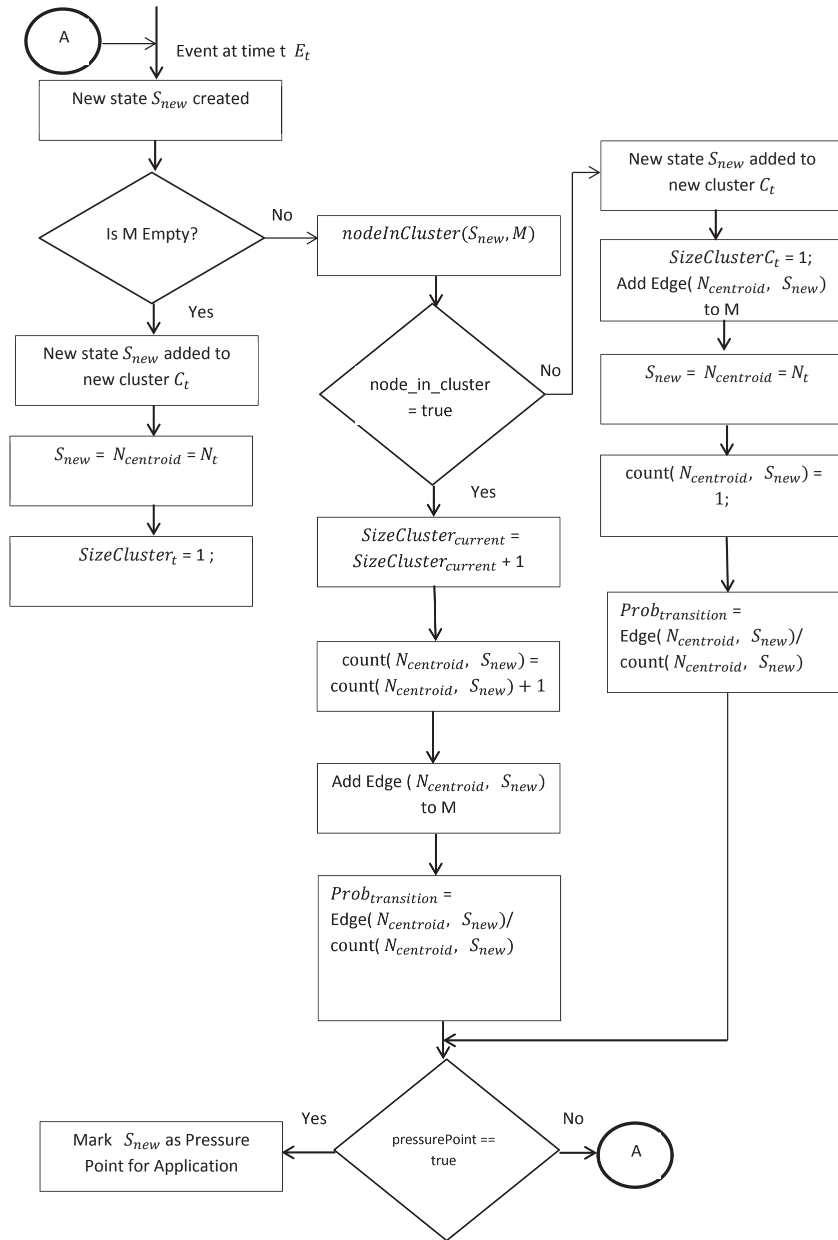
This algorithm is an add on to the createSSM algorithm that actually creates the Markov model. As mentioned earlier, the state of the application is measured using different parameter metrics - response time, OS resource utilization measures and the number of core dumps. Out of these only two effect the end user directly - response time and core dumps. Response times going over a particular SLA or the application crashing are signs of performance issues. So out of all the metrics that

are used to measure the state of the application at any given point of time t, the findPressurePoint algorithm tries to find states where the response time is over the SLA and/or a core dump has been encountered. As has been mentioned in the previous chapters the profile of the incoming transactions is such that there might be traffic peaks and troughs, so in order to process the peaks, the application could go over the specific response time SLA. Such a situation is not necessarily a pressure point but a simple spike in the normal day-to-day processing. In order to characterize that the application is under performance degradation, the response times have to be over the SLA for a continued duration. From an airline operational point of view where application availability needs to be high, 15 minutes of application not meeting SLA thresholds has been deemed as a severity. For the shopping application, response times under 5 seconds are considered good, times between 5 and 8 seconds are considered acceptable and anything over 8 seconds for long durations is unacceptable. findPressurePoint algorithm will adhere to the same conditions. On the other hand a single core dump will be deemed a severity and the state will be tagged as a pressure point. So if and only if there is a core dump(which implies an application crash) or the response time measurements for the application output states is greater than the specified SLA for 15 minutes or greater, the states will be marked as pressure point. Figure 8.14 is a visual representation of the algorithm.

8.6 Simulation Setup

So to understand how a backend software application's performance can be evaluated using the proposed solution, let's look at the high level steps:

1. Factors were picked that could affect the application's performance: In our case we looked at the production severities from the last two years and picked the top 8 factors that caused them, namely - TPS, queue lengths, throttle values, time-



out values, transaction types, payload sizes, database and application thread pools.

2. Looking at the production data, appropriate levels were chosen for each of the factors: In order to make sure that we were using cases that emulate the

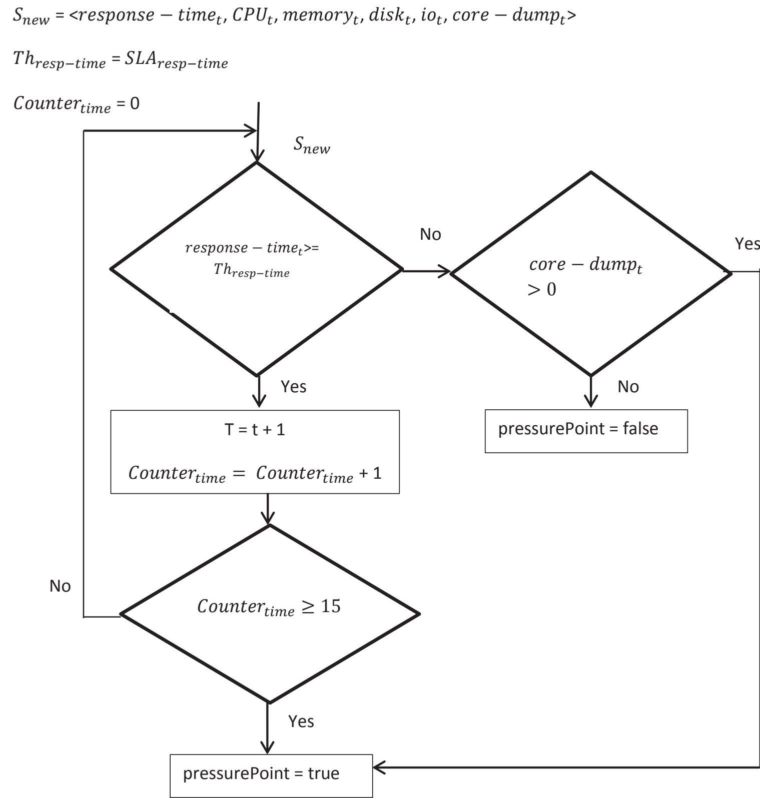


Figure 8.14. findPressurePoint Algorithm to Identify Performance Issues for Software Applications.

production environment, we chose the levels for each of our factors by looking at the production workload profile and configuration settings for factors. 5 levels were chosen for each of the factors, except for workload type that was set to 4 levels.

3. Since we ended up with 8 factors with 5 levels each, we used the Taguchi Orthogonal Array DOE method to setup our test cases: DOE techniques help in setting up test cases where there are multiple factors and levels. Instead of choosing factorial designs where we would have to setup 312,500 test cases, we

chose the Taguchi orthogonal method. It is a DOE method that reduces the number of test cases to be run without limiting the impact each one of the factors could have on the end result.

4. The L50 array in the standard Taguchi orthogonal array corresponded to our factors and levels: The standard array is shown in Figure 6.6 and looking up the rows and columns that correspond to the number of factors (8) and levels (5), we get L50.
5. So 50 test cases were setup according to the L50 array: The L50 array is as shown in Table 6.8, where each rows describes one test case with combination of the factor levels to use.
6. An airline shopping application was chosen as the pilot application
7. createSSM, nodeInCluster and findPressurePoint algorithms help in creating the state space model for our pilot application and identify the pressure points as the different test cases are run against it: Once we have the test scenarios ready and the application is setup with it's interacting components and database in our case, the different tests can now be run. Each input test case is an event and the SSM algorithms then capture the states of the application, create the state clusters and identify pressure points if the application is in a state where the response time is greater than the specified SLA for 15 minutes or more or if core dump occurs.

In order to run through these steps, a simulation tool called extendsim was used. extendsim is a commercially available tool that is used for high performance evaluation of computer systems. Step one was to simulate the working of the airline shopping application. Since this application is a non client interfacing application, it interacts with various other components in the system. A simple representation of the data flow path for the pilot application is shown in Figure 8.15.

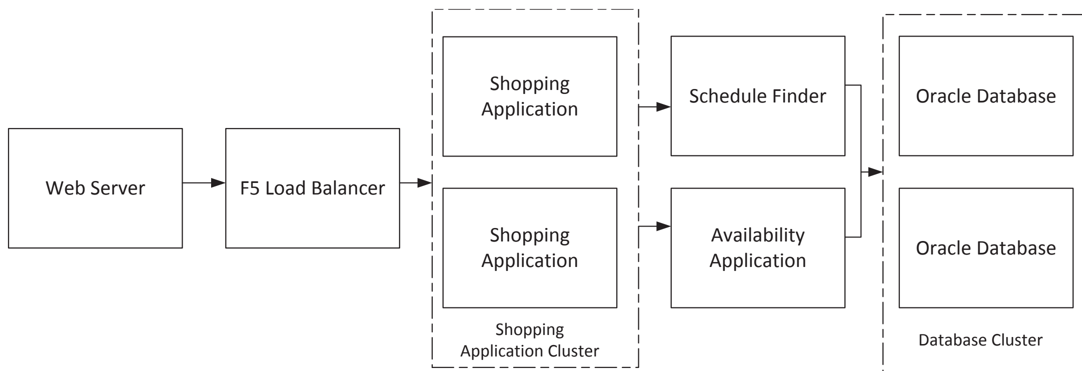


Figure 8.15. Simple Representation of Shopping Application With Backend Components - Availability and Schedule Finder along with the Database Server Pool..

Figure 8.15 shows a cluster of shopping application servers but for simulation purposes, we will use a single instance of the shopping, availability and schedule finder applications. Figure 8.16 is the environment as setup in extendsim.

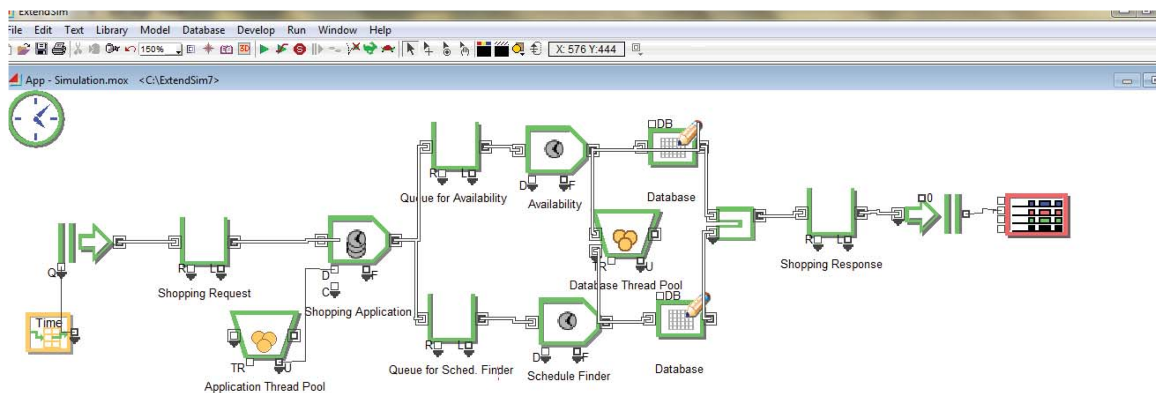


Figure 8.16. Simulation Setup for Shopping Application and Backend Components Shown in Figure 8.15 in extendsim..

Figure 8.16 corresponds to the data flow path for the airline shopping application shown in Figure 8.15. There is an input item that will simulate the incoming

traffic to the shopping application based on a lookup table at each discrete time step. One of the lookup tables is shown in Figure 8.17 for the different levels of the payload size and the processing times for each on the shopping application side. The second table shows the how the other factors can be setup in the simulation tool as attributes. When the simulation is run the values for the various attributes/factors will be picked up from these lookup tables.

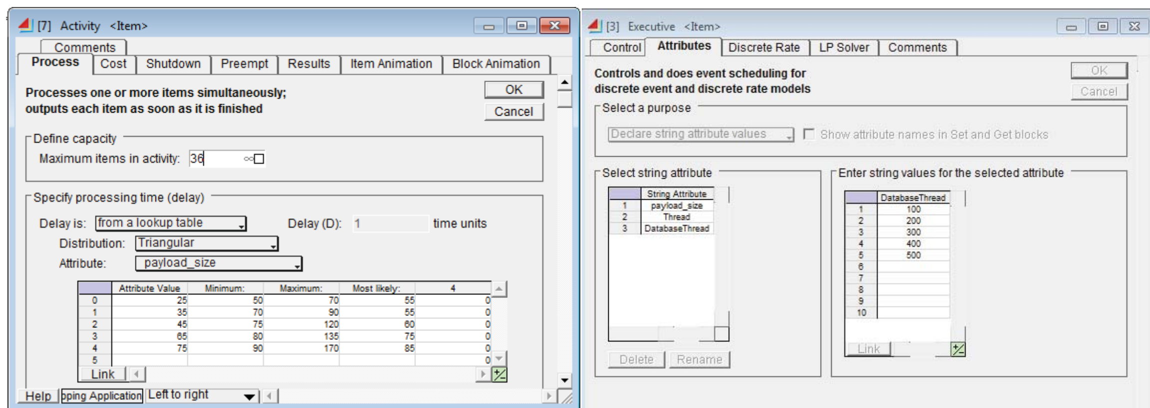


Figure 8.17. The Different Simulation Factors and Their Levels Can be Setup as Attributes in extendim.

Similar tables are setup for the remaining applications that form the simulation environment for the different factors being used in the experiments.

1. Incoming transactions per second (TPS)
2. Workload/Traffic types
3. Timeout values
4. Throttle levels
5. Database connection thread pool
6. Queue size
7. Application connection thread pool

8. Payload size

The resource items are added in the simulation setup to emulate the database and application side thread pools. The values for these correspond to the levels described in Chapter 6. Even though there are two database instances in the simulation setup, they both point to a single common shopping application database. The L50 array that contains the combinations of the levels for the different factors is stored in the database. Each set of the experiment is run corresponding to the records in the database. Once this application environment is setup the three algorithms described in the previous section are added to the simulation - nodeInCluster, createSSM and findPressurePoint are used to iterate through the states of the application at discrete time intervals. The setup is shown in Figure 8.18

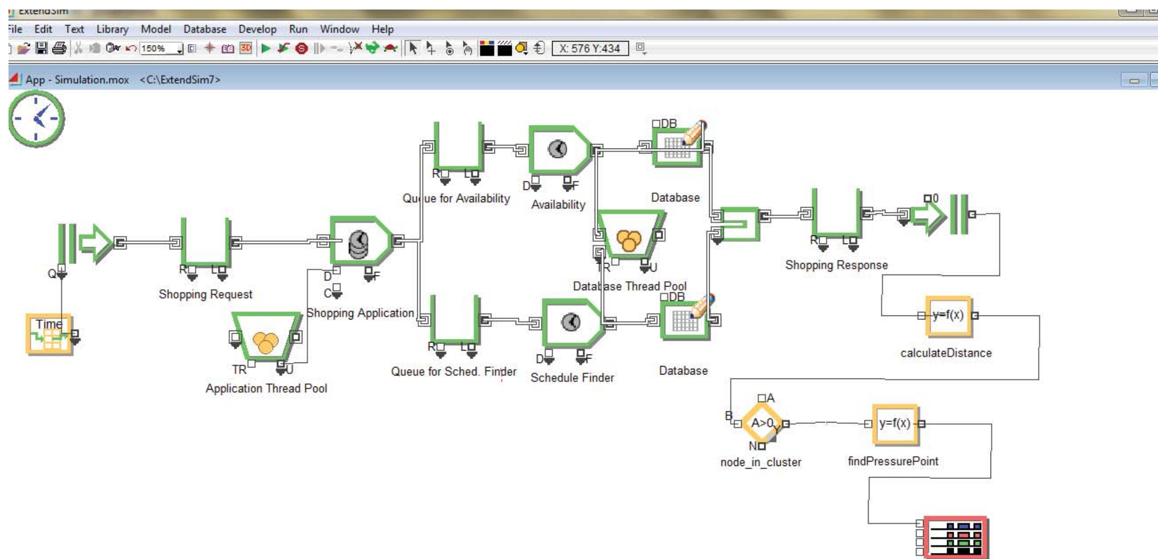


Figure 8.18. Simulation Setup for Airline Shopping Application and findPressurePoint Algorithm to Identify Pressure Point States.

8.7 Simulation Results

Once the simulation is run with the varying input parameters based on the L50 Taguchi orthogonal array, the output data is collected and the following graphs show the various plots for the output state parameters. Figure 8.19 shows the response times for the output states. States that have response times over the SLA for a minimal of 15 time steps are the ones that will be treated as the pressure points. During the tests run with the simulation setup, pressure point states were found and are marked as shown in Figure 8.19.

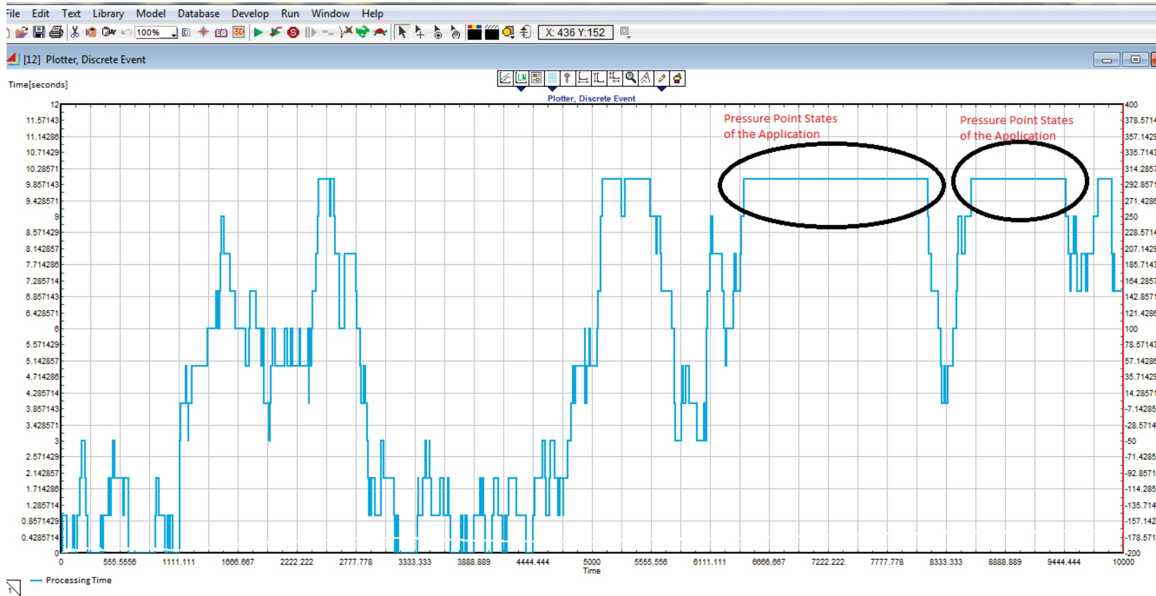


Figure 8.19. Response Time Plot for Simulation Data with Response Times Greater Than the SLA Marked as Pressure Point.

Figures 8.20, 8.21 and 8.22 are plots for the corresponding OS resource utilization measures for CPU, memory and Disk during the simulation. As can be seen from the plots, the application is CPU intensive. The variation in the application's

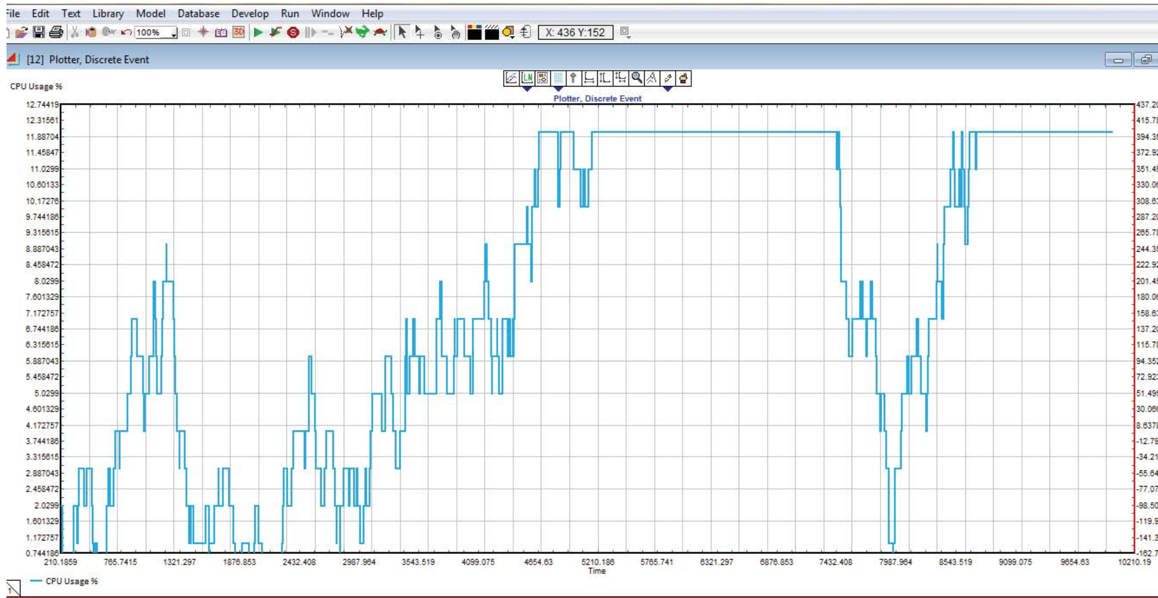


Figure 8.20. CPU Usage Plot for Simulation Data.

processing times causes variations in the CPU usage but not as much with memory and disk usage.

Table 8.2 shows the transition probability matrix for the different states that were created after the simulation. Each of these states represents a cluster of the most similar states.

Last but not the least we will look at the ANOVA calculations for the various factors and their effect on the response time of the application. To calculate the ANOVA we used R. The R statements and their outputs are shown below. The ANOVA, F and p-value calculations were performed for the different factors used in the tests. As a standard p values smaller than 5% or 0.05 imply greater significance. So as can be seen from the ANOVA calculation output - TPS, queue lengths, throttle levels, database thread pool and payload sizes were the most significant factors during the tests. We did not see any timeouts during the tests which implies that all incoming

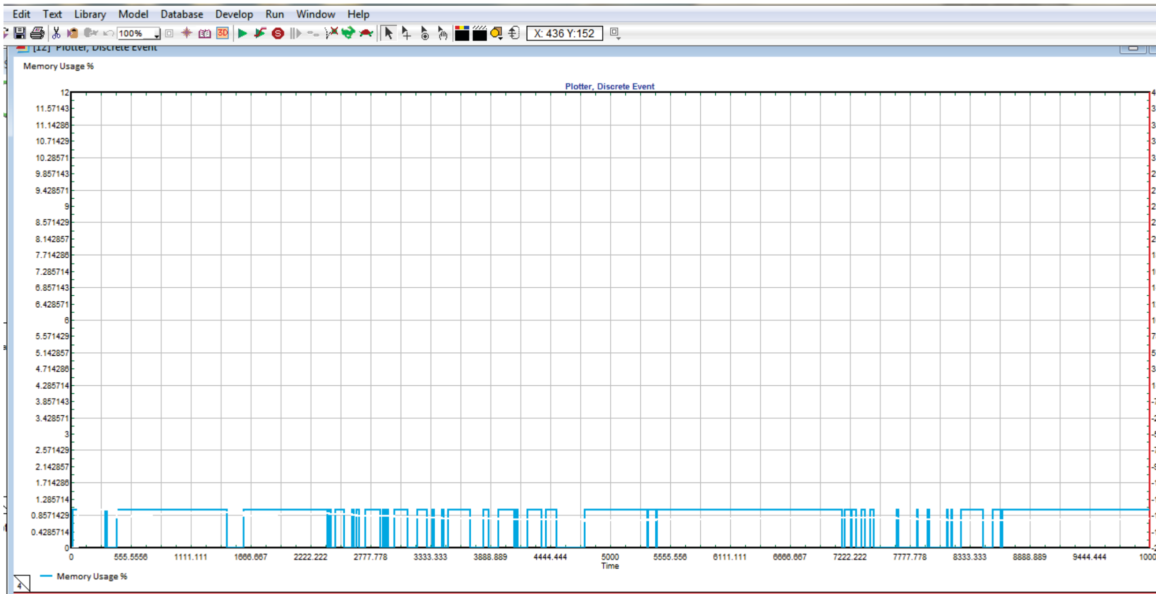


Figure 8.21. Memory Usage Plot for Simulation Data.

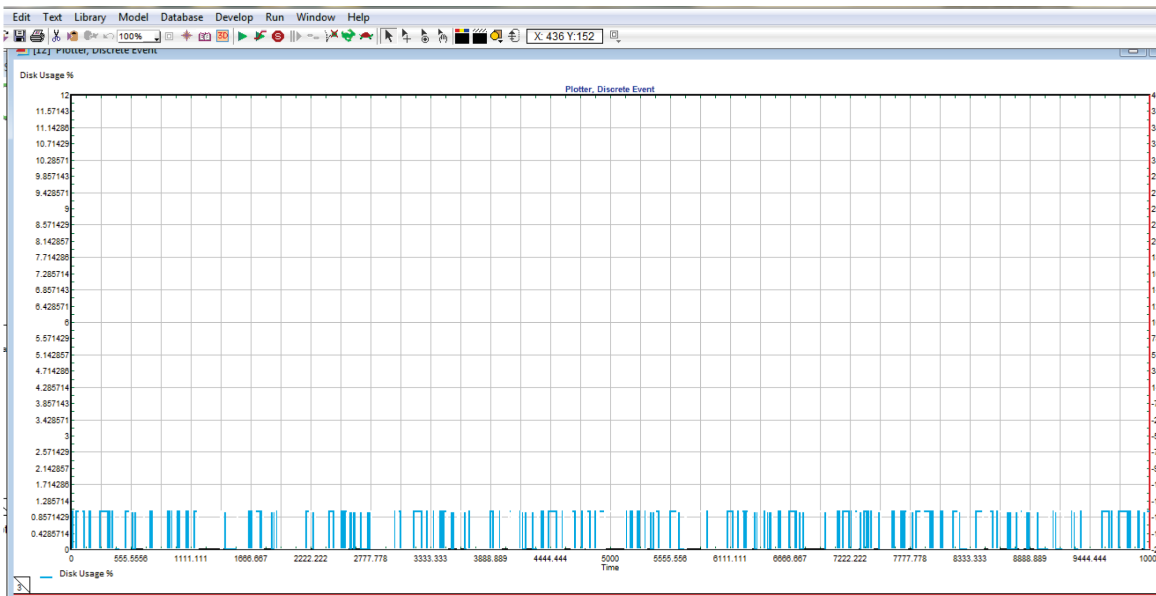


Figure 8.22. Disk Usage Plot for Simulation Data.

States	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
0	0.0	0.02	0.05	0.07	0.05	0.05	0.06	0.01	0.06	0.07	0.06	0.04	0.03	0.04	0.06	0.07	0.0	0.02	0.01	0.07	0.05	0.07	0.03
1	0.04	0.0	0.05	0.05	0.01	0.07	0.03	0.06	0.06	0.03	0.02	0.07	0.04	0.05	0.07	0.04	0.07	0.01	0.0	0.03	0.09	0.07	0.05
2	0.09	0.01	0.0	0.09	0.02	0.02	0.09	0.05	0.01	0.01	0.04	0.08	0.06	0.06	0.03	0.08	0.0	0.05	0.03	0.0	0.05	0.08	0.04
3	0.01	0.01	0.07	0.0	0.01	0.07	0.04	0.07	0.07	0.05	0.03	0.05	0.07	0.06	0.03	0.04	0.05	0.06	0.04	0.04	0.01	0.03	0.07
4	0.02	0.02	0.07	0.04	0.0	0.07	0.09	0.09	0.1	0.06	0.03	0.01	0.0	0.01	0.04	0.01	0.02	0.05	0.11	0.05	0.04	0.02	0.06
5	0.08	0.03	0.07	0.08	0.03	0.0	0.09	0.08	0.05	0.04	0.04	0.01	0.07	0.03	0.04	0.04	0.02	0.0	0.06	0.06	0.02	0.05	0.0
6	0.04	0.09	0.02	0.04	0.08	0.01	0.0	0.02	0.08	0.01	0.05	0.0	0.09	0.02	0.04	0.05	0.03	0.05	0.06	0.01	0.09	0.01	0.09
7	0.04	0.0	0.05	0.02	0.02	0.07	0.02	0.0	0.04	0.05	0.03	0.06	0.07	0.07	0.04	0.04	0.06	0.07	0.07	0.0	0.07	0.09	0.01
8	0.08	0.08	0.0	0.0	0.07	0.0	0.01	0.09	0.0	0.03	0.06	0.01	0.01	0.07	0.07	0.08	0.06	0.05	0.06	0.09	0.03	0.03	0.01
9	0.01	0.04	0.06	0.06	0.09	0.03	0.09	0.06	0.01	0.0	0.03	0.02	0.04	0.07	0.05	0.08	0.07	0.02	0.06	0.04	0.05	0.01	0.0
10	0.01	0.06	0.06	0.05	0.07	0.06	0.06	0.0	0.01	0.03	0.0	0.01	0.04	0.07	0.04	0.07	0.03	0.01	0.03	0.08	0.05	0.07	0.09
11	0.06	0.05	0.06	0.08	0.03	0.05	0.08	0.03	0.01	0.04	0.0	0.0	0.02	0.02	0.09	0.05	0.0	0.05	0.05	0.08	0.08	0.06	0.02
12	0.01	0.03	0.04	0.02	0.0	0.01	0.04	0.05	0.1	0.03	0.06	0.0	0.0	0.09	0.07	0.07	0.02	0.07	0.09	0.09	0.08	0.0	0.04
13	0.07	0.07	0.03	0.07	0.03	0.06	0.04	0.06	0.06	0.05	0.0	0.07	0.07	0.0	0.02	0.05	0.01	0.04	0.06	0.07	0.06	0.01	0.02
14	0.06	0.06	0.04	0.06	0.05	0.05	0.0	0.08	0.0	0.02	0.04	0.01	0.01	0.07	0.0	0.07	0.03	0.01	0.07	0.08	0.05	0.08	0.05
15	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	0.03	0.06	0.01	0.09	0.04	0.09	0.07	0.08	0.01	0.06	0.06	0.01	0.03	0.0	0.01	0.03	0.0	0.04	0.08	0.09	0.1	0.02	0.01
17	0.08	0.09	0.04	0.02	0.08	0.05	0.0	0.03	0.08	0.01	0.02	0.06	0.08	0.02	0.02	0.02	0.06	0.0	0.02	0.04	0.09	0.06	0.03
18	0.02	0.03	0.03	0.01	0.07	0.1	0.1	0.05	0.05	0.02	0.03	0.03	0.01	0.09	0.05	0.04	0.0	0.09	0.0	0.05	0.03	0.07	0.03
19	0.04	0.1	0.1	0.02	0.11	0.02	0.0	0.01	0.02	0.11	0.05	0.01	0.1	0.07	0.05	0.04	0.0	0.02	0.04	0.0	0.03	0.04	0.01
20	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.8	0.0	0.0
21	0.01	0.05	0.08	0.09	0.08	0.03	0.01	0.07	0.01	0.06	0.04	0.06	0.05	0.02	0.06	0.05	0.07	0.03	0.01	0.06	0.02	0.0	0.05
22	0.06	0.05	0.07	0.03	0.08	0.06	0.02	0.07	0.08	0.01	0.02	0.01	0.07	0.05	0.09	0.05	0.01	0.06	0.06	0.05	0.0	0.02	0.0

Table 8.2. Transition Probabilities of the State Space Model Generated as a Result of the Simulation

transactions were processed by the application. In turn this also means that the application was able to handle all incoming traffic. This is the reason why the timeout values and the application thread pool did not play a significant role in the output (response times).

```
> ANOVACalc <- read.table("C:/temp/sim-data.txt", sep=",",
    header=T,quote="")
> head(ANOVACalc)
      respTime tps    queuelen throttle.level timeout.value db.threads
          app.threads payload.size payload.type
1       9.10    48      1050            50            20        400
    300           45     INTLWPI1
2       3.00    35      2000            40            30        500
    200           45     INTLWPI1
3       9.50    48      2050            35            20        200
    100           20     LFSTREAM
```

```
4       4.23    15       2050           50           10        100
   400          35    INTLWPI1
5       4.44    35       2050           50           15        400
   200          35    LFSTREAM
6       9.50    25       1000           50           10        100
   200          75    ATSEILF2
```

```
> aovObject <- aov(ANOVACalc\$respTime ~ ANOVACalc\$tps +
   ANOVACalc\$queueLen + ANOVACalc\$throttle.level +
   ANOVACalc\$timeout.value + ANOVACalc\$db.threads +
   ANOVACalc\$app.threads + ANOVACalc\$payload.size)
> summary(aovObject)
                       Df Sum Sq Mean Sq F value Pr(>F)
ANOVACalc\$tps           4   4421   4421  27.577 6.48e-07 ***
ANOVACalc\$queueLen      4   3935    328   2.046  0.0256 *
ANOVACalc\$throttle.level    4   1120    428   2.186  0.0131
ANOVACalc\$timeout.value 3    237    237   1.477  0.2267
ANOVACalc\$db.threads    4    687    357   2.033  0.0221
ANOVACalc\$app.threads   4    943  2.569   0.357  0.550
ANOVACalc\$payload.size  3    833 10.921   3.520  0.0018
Residuals             122  19556    160
---
Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1    1
```

8.8 Proposed Solution vs. Conventional Performance Tests

As seen from the previous chapters the performance of an application is dependent on numerous factors, especially if it's a middle tier non client interfacing application. This is due to the fact that it interacts with other applications, the database, middleware components like queues and it could also be using techniques to transform messages and/or different protocols to communicate. Keeping this in mind the proposed solution uses multiple input factors to validate the performance of the application. In conventional performance tests as well as the impulse test described in Chapter 2 rely on varying incoming transaction rates to evaluate the application. The impulse tests are an improvement on the existing conventional tests in that, they emulate production traffic profiles but still the piece of code under test is being verified using two factors - TPS and workload types. Transaction rate is obviously an important variable but since we need to learn how the software will behave standalone as well as when integrated with other components, it becomes important to test it against different factors.

The reason for running performance tests is to identify and fix bottlenecks in advance, before the software is deployed in production. This implies putting the application through different scenarios that might occur in production and then evaluating how well it behaves under those scenarios. There is no way to test the code for all combinations and permutations of cases that might occur but a tester's responsibility is to run tests that cover as many cases as possible that cover both the happy path as well failure paths. The objective is to make the application work outside it's normal processing boundaries. In order to do so we have to utilize factors that affect the processing of the application. TPS is just one of those factors and therefore it is possible to miss cases that could cause service disruptions in production if we chose just this one factor. So even though it will be impossible to run each and every sce-

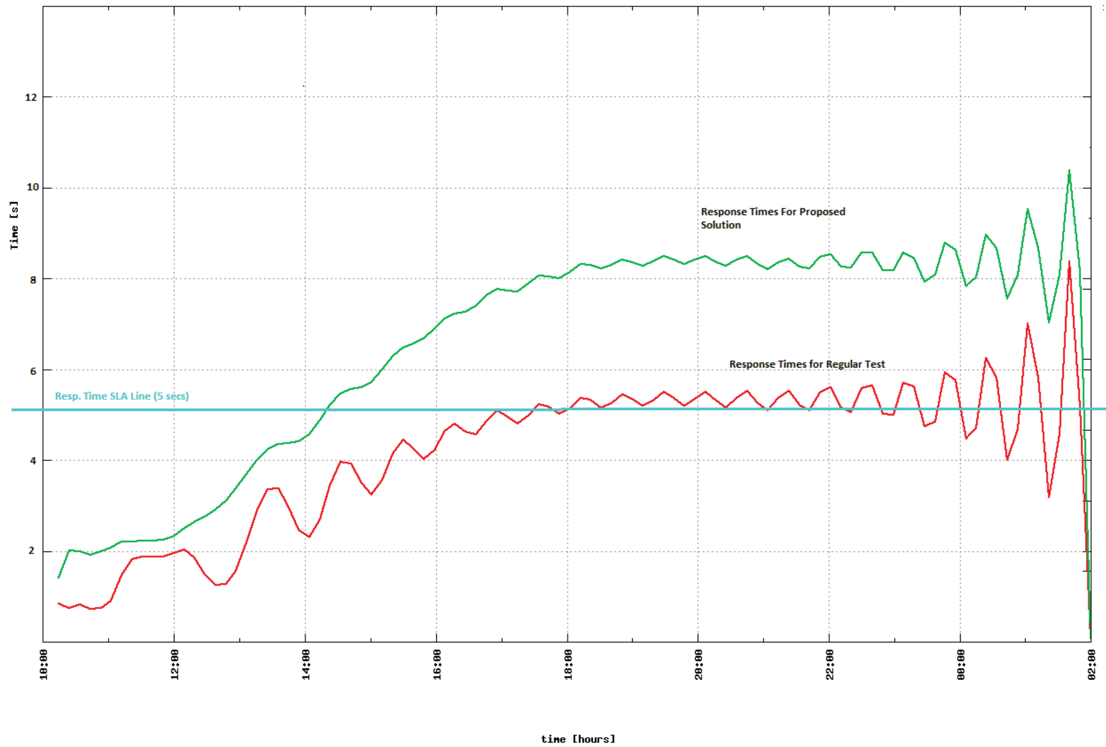


Figure 8.23. Response Time Plot for Test Run on Real World Airline Shopping Application Using Proposed Solution vs. Regular Performance Test. The Combination of Multiple Factors Pushes the Application Over It's Response Time SLAs Before an Increase in TPS in a Regular Performance Test Does.Pressure Points are Detected Using Proposed Solution but Not Using a Regular Test..

nario with each and every factor that could possibly have any effect on the workings of the application, the proposed solution uses 8 such factors. These factors cover the client side, the application configuration, the interacting application processing times as well as the middleware component configuration. Taking all these variables as parameters in the tests cover the end to end performance evaluation of the application.

To compare the results from the proposed solution to those of a conventional performance test that uses TPS and workload types as it's input factors, we will use

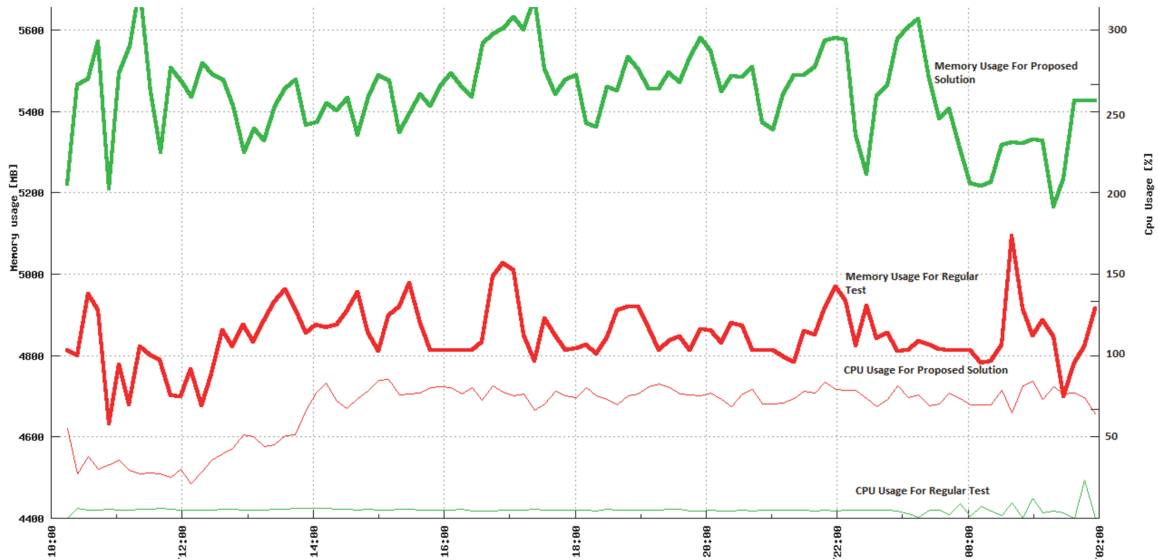


Figure 8.24. CPU and Memory Usage Plot for Test Run on Real World Airline Shopping Application Using Proposed Solution vs. Regular Performance Test. The OS Utilization is Higher for the Proposed Solution Since It Pushes the Application Outside It's Normal Processing Boundaries..

response times and CPU and memory usage data. Instead of running the comparison using simulations, both the proposed solution and the conventional performance test are implemented on the real world shopping application. Due to time constraints instead of running all 50 test scenarios and using all 8 factors, this test uses TPS, workload types and sizes, queue depth, throttle and timeout factors and their corresponding levels as described in Chapter 6. As compared to this, the regular performance test will simply use the varying TPS factor levels and workload types. The test was run for 16 hours and some of the common performance metrics were measured. Figure 8.23 shows the processing time graph for the application.

As can be seen from the graph the proposed solution pushes the application outside it's normal processing boundaries. The combination of factors being used, cause the application to go over it's SLA way before the TPS pushes it over the SLA when

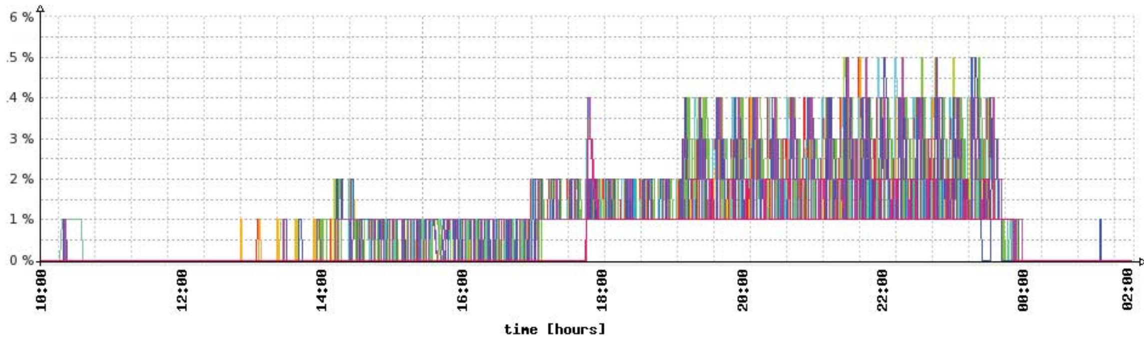


Figure 8.25. Queue Depth Utilization Percentage for Real World Airline Shopping Application Workload Types Using Proposed Solution and Using the Queue Depth Factor Levels from Chapter 6.

running a regular performance test. In the case of the conventional test, no pressure points will be found since the response time spikes are not sustained for a continuous duration of 15 minutes or more. Whereas the proposed DOE solution shows, that the application will hit a pressure point trying to process incoming transactions because of other combination of factors. Similarly Figure 8.24 shows the CPU and memory usage plot.

As expected since the code is processing under scenarios that require it to use more OS utilization is the case of the proposed DOE solution as compared to the regular test. Figure 8.25 shows the percentage utilization of the queue depth for the proposed solution as the queue depth factor is varied. When conventional tests are run in this manner, they fail to identify pressure points that could occur in production when factors outside the application change - like processing times for interacting applications. These scenarios can only be discovered if we are running tests that take them into consideration.

As can be seen from Figure 8.26 when the application uses the proposed solution, it had to throttle incoming transaction plus some of the transactions timed-

out. While the application when under a conventional performance test is processing transactions normally with a few timeout transactions at the end of the test. Using multiple factors to run performance tests helps to detect bottlenecks that are related to other parameters too as compared to simply TPS related pressure point states. As can be seen from this data, a conventional test depending on what values have been chosen for the various factors of the interacting components, could miss finding pressure points. Service disruption due to performance related issues in the production environment need to be avoided at all costs since they not only cause customer dissatisfaction but could also lead to revenue loss.

Errors

Code	picli311		Name
5053	2549		NO_COMBINABLE_FARES_FOR_CLASS
5111	349		TRANSACTION_THRESHOLD_REACHED
6002	1273		INVALID_INPUT
6201	2		NO_VALID_FLIGHT_DATE_FOUND
6203	2		EMPTY_ALT_DATES_PAIRS_MAP
9997	2305		REQUEST_TIMEOUT
9998	21		SYSTEM_EXCEPTION
Total	5094		

Code		picli314	Name
5011		4182	NO_FARE_FOR_CLASS_USED
5033		33	NO_RULES_FOR_PSGR_TYPE_OR_CLASS
6002		1463	INVALID_INPUT
6119		640	NO_PRIVATE_FARES_VALID_FOR_PASSENGER
6120		5	NO_PUBLIC_FARES_VALID_FOR_PASSENGER
6124		19	NO_CORPORATE_NEG_FARES_EXISTS
9502		33	LMT_ISSUE_SEP_TKTS_INTL_SURFACE_RESTR
9997		145	REQUEST_TIMEOUT
Total		6520	

Figure 8.26. Errors Including Timeouts and Throttled Transactions for the Shopping Application Using the Proposed Solution (LHS) vs. Errors for a Regular Test (RHS).

8.9 Performance Prediction Using SSMs

Now one of the major advantages of using SSMs - in our case the Markov models is that it can predict a future state. This is in addition to the fact that these models can be used to detect anomalies. Once we've build the initial state space model and have the state transition probabilities, it is now possible to predict the probability of being in a pressure point state, given the current state of the application.

A lot of work has been done in the field of performance evaluation using such models. [66] proposes a new metric called Performance Nonscalability likelihood (PNL) that is then used to predict if an increase in the workload will introduce a bottleneck in the system. The paper studies the workload - different types of incoming transactions and then profiles the application for each one of them, measuring it's processing times, database access times and so on. PNL measures performance degradation at a particular load level. A stochastic process is used to compute the states of the application which are then marked 1 if performance is unacceptable or 0 if it is acceptable. The state probability distribution is generated and then PNL is used to predict performance given a particular workload level. The authors of [67] use Markov chains to evaluate the performance of large scale stateful web services. The paper proposes techniques to build test cases and datasets that emulate production scenarios. These techniques rely on the incoming traffic to the production system and then applying data sanitization processes that strip the data of sensitive information but still preserve their stateful nature. Markov chains are used for data mining the large volume of logs from production and then identifying load patterns. The assumption is made that the incoming traffic profile remains the same over long durations of time. Once the load patterns are discovered, they are played back in the test environment.

[68] propose algorithms to automatically generate test suites for telecommunication systems but can be generalized for other applications. One of the algorithms uses Markov chains to generate the state space model for the application under test by utilizing the incoming traffic to the application. A similar approach is presented in [69]. The traffic profile for the application is used to create the state space model using Markov chains. Each one of these states is then used as a test case to load test the application. The path of execution from initial state to any particular state

represented by the Markov model is verification of it's proper run. Any deviation in the execution path points to a potential problem in the system.

A number of solutions exist to run tests where multiple factors can affect the performance the application. [70] proposes a fractional factorial solution where factors affecting the performance are first analyzed to find their interaction effects, and then only factors that have a major impact on the application's performance are used. Two levels for each of the chosen factors are selected and test cases generated using the fraction factorial DOE method. The cases study names the factors as A, B, C and so on, so it is hard to tell what factors were really chosen. The authors of [71] propose to use distributed continuous quality assurance DCQA processes to divide the QA regression test cases into smaller tasks and then distributing these tasks to be run by the end users and development teams that are distributed rather than running them in one lab. The special DCQA environment has been created for this that intelligently distributes tasks. All factors that are used to decide the breaking up of the tasks are assumed to be interacting and DOE methods are used to find their main effects. This helps in reducing the large number of tasks into smaller ones. The case studies provided in the paper take into account hardware and OS configuration related factors and software SLAs for response times and memory footprint. Another paper [72] uses DOE General Linear Model method to tune storage systems. The paper takes into account three factors that affect storage performance - cache partition size, LUN strip size and cache segment size with three levels to find their interaction and effect on performance. All other factors are kept constant.

All these solutions that use Markov models to build the state space model for the application and then predict bottlenecks are all based on incoming workload. As we've seen from several real world application examples in this document, performance degradation of an application can be due to several combinations of other

factors. The solution that we've proposed, takes into consideration not only the client side metrics, but also the application specific configuration parameters, middleware component factors and interacting application factors that all have an effect on the overall performance. The workload profile for an application changes over time, as new features are added to releases and/or new transaction types come in, so the existing solutions have to be changed continuously. The DOE orthogonal array method used in our solution takes into account all the factors that help build the state space model and have more than 2 (fractional factorial methods) or 3 (GLM method) proposed in the papers. The combination of using these factors as well as the state space model helps us in finding bottlenecks that may be related to the interacting application or the middleware component or the database and not just workload related pressure points. The application states will also help us in predicting performance.

So now to see how performance prediction can work in our case, let's once again take the table of events 8.1. The sequence of state transitions probabilities can be summarized as follows:

Table 8.3. State Transition Probabilities for Example State Table 8.1

States	**1**	**2**	**3**	**4**	**5**	**6**
1	0	1	0	0	0	0
2	0	0.5	0.5	0	0	0
3	0	0	0	1	0	0
4	0	0	0	0.5	0.5	0
5	0	0	0	0	0	1
6	0	1	0	0	0	0

According to the transition probabilities, the sequence of states is 1 2 2 3 4 4 5 6. Once we have this sequence and the transition probabilities between the states, we can use the Markov model property to predict the transition probabilities at a future

time step. So say at time n we are in state 3, using the following Markov property we can predict the probability of transitioning from state 3 to all other states at any time step in the future.

$$\pi^{(1)} = \pi^{(0)} P \tag{8.1}$$

$$\pi^{(2)} = \pi^{(1)} P = \pi^{(0)} P^2 \tag{8.2}$$

$$\pi^{(n)} = ... = \pi^{(0)} P^n \tag{8.3}$$

where π is the initial state vector and P is the state transition probability matrix and n denotes the time step.

So in this example if at step n we are in state 3 and we wanted to find out the state transitions at time step n = 4, we would have the following:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}^4$$

$$= \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0.125 & 0.125 & 0.5 & 0.25 & 0 \\ 0 & 0.0625 & 0.0625 & 0.375 & 0.25 & 0.25 \\ 0 & 0.5 & 0 & 0.125 & 0.125 & 0.25 \\ 0 & 0.5 & 0.25 & 0.0625 & 0.0625 & 0.125 \\ 0 & 0.25 & 0.25 & 0.5 & 0 & 0 \\ 0 & 0.125 & 0.125 & 0.5 & 0.25 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0.5 & 0 & 0.125 & 0.125 & 0.25 \end{bmatrix}$$

So at time n + 4, the application has a 50% probability of being in state 2, 12.5% probability to be in either state 4 or 5 and a 25% probability to be in state 6.

Apart from being able to predict the state transition probabilities at a given time in the future, there are a few other statistical observations that can be made using Markov chains. In our case since we are using Markov models to describe the performance profile of an application, using the transition probabilities we will be able to tell how many time steps later from each state will the application get back to the bottleneck state. Taking the same experiment setup that we described in section 8.6 and to keep the example manageable we'll simply run four test cases, generate the states and the state transition probabilities. And using the data we will now be able to predict not only the probability of being in any one of these states in the future but also the first time the pressure point state occurs again (assuming that we see one during the initial run) and also the recurrence probability of this bottleneck state at a future time step. The state transition probabilities for four test cases chosen from the orthogonal array built in Chapter 6, is as shown in Table 8.4.

Table 8.4. State Transition Probabilities for Four Test Cases Run From the Taguchi Orthogonal Array for an Airline Shopping Application

States	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	0.5	0	0	0	0.5	0	0	0
1	0.33	0	0.34	0	0.33	0	0	0	0
2	0	0.5	0	0.5	0	0	0	0	0
3	0	0	0.33	0	0.34	0	0	0	0.33
4	0	0	0	0	1	0	0	0	0
5	0.33	0	0	0	0.34	0	0.33	0	0
6	0	0	0	0	0	0.5	0	0.5	0
7	0	0	0	0	0.34	0	0.33	0	0.33
8	0	0	0	0.5	0	0	0	0.5	0

In this particular case we have one pressure point state - state 4. According to the Markov model definition this state is an absorbing state and the others can be termed transient states. So for any Markov chain that has r absorbing states and t transient states, the canonical representation of the chain's transition probability matrix can be written as: $\begin{pmatrix} Q & R \\ O & I \end{pmatrix}$

Here Q is a t X t matrix, I is a r X r identity matrix, R is a nonzero t X r matrix and O is a r X t zero matrix. The fundamental matrix $F = inv(I^{\circ}Q)$, which implies $F = I + Q + Q^2 + err$, where err is the error term. So now for our example we want to find the number of time steps it would take to get the bottleneck state (state 4) from any initial state. This is called the mean first passage time. In order to do that, we can rewrite the transition probability matrix for our case shown in Table 8.4,in it's canonical form:

Table 8.5. Canonical State Transition Probability Matrix for Table 8.4

States	**0**	**1**	**2**	**3**	**5**	**6**	**7**	**8**	**4**
0	0	0.5	0	0	0.5	0	0	0	0
1	0.33	0	0.34	0	0	0	0	0	0.33
2	0	0.5	0	0.5	0	0	0	0	0
3	0	0	0.33	0	0	0.34	0	0.33	0.33
5	0.33	0	0	0	0	0	0	0	0.34
6	0	0	0	0	0.5	0	0.5	0	0
7	0	0	0	0	0	0.33	0	0.34	0.33
8	0	0	0	0.5	0	0	0.5	0	0
4	0	0	0	0	0	0	0	0	1

From the canonical transition probability matrix, 8.5, the Q matrix can be calculated as:

$$Q = \begin{pmatrix} 0 & 0.5 & 0 & 0 & 0.50 & 0 & 0 & \\ 0.33 & 0 & 0.34 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.33 & 0 & 0 & 0.34 & 0 & 0.33 \\ 0.33 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.33 & 0 & 0.34 \\ 0 & 0 & 0 & 0.5 & 0 & 0 & 0.5 & 0 \end{pmatrix}$$

Given this the fundamental matrix F can be calculated:

$$F = \frac{1}{8}\begin{pmatrix} 14 & 9 & 4 & 3 & 9 & 4 & 3 & 2 \\ 6 & 14 & 6 & 4 & 4 & 2 & 2 & 2 \\ 4 & 9 & 14 & 9 & 3 & 2 & 3 & 4 \\ 2 & 4 & 6 & 14 & 2 & 2 & 4 & 6 \\ 6 & 4 & 2 & 2 & 14 & 6 & 4 & 2 \\ 4 & 3 & 3 & 3 & 9 & 14 & 9 & 4 \\ 2 & 2 & 2 & 4 & 4 & 4 & 14 & 6 \\ 2 & 3 & 4 & 9 & 3 & 4 & 9 & 14 \end{pmatrix}$$

The fundamental matrix, F represents the number of times the application stays in a particular state before it hits the bottleneck state (state 4 in our case). So these are the time steps at each state before the application hits state 4 for the first time. In order to find the number of time steps it takes to get to the bottleneck state, we will use one of the Markov model theorems that states that is the chain starts at state S_i, and t_i is the number of steps before the chain gets to a target state and t is a column vector such that it's ith entry is t_i, then the following is true: t = Fc. Here c

is another column vector with all entries being 1. So in our case

$$Fc = \begin{pmatrix} 6 \\ 5 \\ 6 \\ 5 \\ 5 \\ 6 \\ 5 \\ 6 \end{pmatrix}$$

From this we can conclude that if the initial starting state is 0, it will take us 6 time steps to get to the bottleneck state. If we start at state 2, it will take us 5 steps to get to our pressure point state and so on.

In addition to the two above stated statistical observations, Markov chains and their transition probabilities can be used to determine a sequence of states in the future. Let's assume that instead of having a single state pressure point node in the SSM, we now have a specific sequence of states that point towards a bottleneck. Let's assume that the sequence of states can be represented as x and has a length L and we are interested in finding out the probability of x occurring given a model M. The probability of x can be determined as:

$Pr(x) = Pr(x_L, x_{L-1}, ...x_1) = Pr(x_L|x_{L-1}, ...x_1).Pr(x_{L-1}|x_{L-1}, .., x_1).Pr(x_1)$

We know that for a Markov chain, the probability of each x_i depends only on the value of x_{i-1}. Therefore

$$Pr(x) = Pr(x_L|x_{L-1}).Pr(x_{L-1}|x_{L-2})...Pr(x_2|x_1).Pr(x_1) = Pr(x_1)\prod_{i=2}^{L} Pr(x_i|x_{i-1}) \quad (8.4)$$

Consider the following transition probability matrix for four states shown in Table 8.6:

Table 8.6. Sample Initial State Transition Probability Matrix

States	**0**	**1**	**2**	**3**
0	0.18	0.274	0.426	0.12
1	0.171	0.368	0.274	0.188
2	0.161	0.339	0.375	0.125
3	0.079	0.355	0.384	0.182

So now if we know that the state sequence 3 3 3 3 points to a bottleneck, we can find the probability of such a sequence occurring in the future by using the Equation (8.4) and calculating the following:

$Pr(3333) = Pr(3).Pr(3|3).Pr(3|3).Pr(3|3)$

Now from common probability knowledge, we know that given two events the conditional probabilities can be calculated as follows:

$P(A|B) = \frac{A \cap B}{P(B)} and$

$P(B) = \frac{n(B)}{n(A \cup B)}$

Where $P(A|B)$ is the probability of A given B, P(B) represents the probability of B, $n(A \cup B)$ is the count of the total number of elements in the set and n(B) is the total number of occurrences of B in $n(A \cup B)$. In our case the elements of the set are equivalent to the states of the application. Let's assume that when we run the different test cases in a particular order, we observe the following application state sequence

1 2 0 3 3 2 1

we can now find the probability of occurrence for the pressure point state 3 3 3 3 by calculating the individual probabilities and the conditional probabilities of the

specific states occurring in a particular order in the future by looking at the normal application state sequence 1 2 0 3 3 2 1:

$Pr(0) = \frac{1}{7}$

$Pr(1) = \frac{2}{7}$

$Pr(2) = \frac{2}{7}$

$Pr(3) = \frac{2}{7}$

$Pr(3|3) = \frac{0.182}{\frac{2}{7}}$

So using the conditional probability formaula shown in Equation (8.4) for a Markov chain sequence,

$Pr(3333) = \frac{2}{7} . \frac{0.182}{\frac{2}{7}} . \frac{0.182}{\frac{2}{7}} . \frac{0.182}{\frac{2}{7}}$

$= 0.07689$

Therefore there is a 7.689% chance that the application will run into a pressure point state in the future. It is these kind of statistical observations that can be used to proactively evaluate an application's performance over time. Since the underlying model was built using test cases that contained factors from production, it will give be a pretty good representation of what could happen when the application is live. If the model contains a bottleneck state or a sequence of states that could cause a bottleneck, it then becomes easy to detect under what scenario the issue happens and also predict the probability of it occurring again in the future. Since we are now able to calculate the probability of the application running into a pressure point at any given point of time, plus we know what combination of factor levels caused the issue, it becomes easier to work on a solution before the incident happens again.

REFERENCES

[1] T. Nivas, "Test harness and script design principles for automated testing of non-gui or web based applications," in *Proc. International Workshop on E2E Test Script Engineering*, 2011.

[2] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea, *Performance Testing Guidance for Web Applications*, ser. Patterns and Practices. Microsoft Press, Nov. 2007.

[3] T. Riley and A. Goucher, *Beautiful Testing*. O'Reilly, Oct. 2009.

[4] T. Nivas and C. Csallner, "Managing performance testing with release certification and data correlation," in *19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), Industry Track*. ACM, Sept. 2011.

[5] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005, pp. 105–118.

[6] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *Proc. 28th International Conference on Software Engineering (ICSE)*. ACM, May 2006, pp. 282–291.

[7] G. Everett and R. McLeod Jr., *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley, July 2007.

[8] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *Journal of Software Maintenance*, vol. 20, no. 4, pp. 249–267, 2008.

[9] R. K. Mansharamani, A. Khanapurkar, B. Mathew, and R. Subramanyan, "Performance testing: Far from steady state," in *Proc. 34th Annual IEEE International Computer Software and Applications Conference Workshops (COMPSACW).* IEEE, July 2010, pp. 341–346.

[10] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," *IEEE Transactions on Software Engineering (TSE)*, vol. 26, no. 12, pp. 1147–1156, Dec. 2000.

[11] G. Denaro, A. Polini, and W. Emmerich, "Early performance testing of distributed software applications," in *Proc. 4th ACM International Workshop on Software and Performance (WOSP).* ACM, Jan. 2004, pp. 94–103.

[12] G. Shah, "Software - testability," *DeveloperIQ*, Jan. 2009.

[13] J. Gao and M.-C. Shih, "A component testability model for verification and measurement," in *Proc. 29th Annual International Computer Software and Application Conference (COMPSAC).* IEEE, July 2005, pp. 211–218.

[14] Z. Balaton and G. Gombás, "Resource and job monitoring in the grid," in *Proc. 9th International Conference or Parallel Processing (Euro-Par).* Springer, Aug. 2003, pp. 404–411.

[15] B. M. Oki, M. Pflügl, A. Siegel, and D. Skeen, "The information bus - an architecture for extensible distributed systems," in *Proc. 14th ACM Symposium on Operating System Principles (SOSP).* ACM, Dec. 1993, pp. 58–68.

[16] A. Chan, "Transactional publish / subscribe: The proactive multicast of database changes," in *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD).* ACM, June 1998, p. 521.

[17] R. Buyya, "Parmon: A portable and scalable monitoring system for clusters," *Software—Practice & Experience (SP&E)*, vol. 30, no. 7, pp. 723–739, June 2000.

[18] R. A. Finkel, "Pulsar: An extensible tool for monitoring large Unix sites," *Software—Practice & Experience (SP&E)*, vol. 27, no. 10, pp. 1163–1176, Oct. 1997.

[19] S. E. Parkin and G. Morgan, "Toward reusable SLA monitoring capabilities," *Software—Practice & Experience (SP&E)*, vol. 42, no. 3, pp. 261–280, Mar. 2012.

[20] A. Sztajnberg, R. S. Granja, J. Cesário, and A. F. A. Monteiro, "An integration experience of a software architecture and a monitoring infrastructure to deploy applications with non-functional requirements in computing grids," *Software—Practice & Experience (SP&E)*, vol. 41, no. 1, pp. 103–127, Jan. 2011.

[21] W. Gu, G. Eisenhauer, K. Schwan, and J. S. Vetter, "Falcon: On-line monitoring for steering parallel programs," *Concurrency—Practice & Experience (CP&E)*, vol. 10, no. 9, pp. 699–736, Aug. 1998.

[22] R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, B. Tierney, and R. Wolski, "A grid monitoring architecture," Global Grid Forum, Tech. Rep. GWD-Perf-16-1, July 2001.

[23] S. Zanikolas and R. Sakellarioui, "A taxonomy of grid monitoring systems," *Future Generation Computer Systems*, vol. 21, no. 1, Jan. 2005.

[24] Y. Liao and D. Cohen, "A specificational approach to high level program monitoring and measuring," *IEEE Transactions on Software Engineering (TSE)*, vol. 18, no. 11, pp. 969–979, Nov. 1992.

[25] A. V. Dastjerdi, S. G. H. Tabatabaei, and R. Buyya, "A dependency-aware ontology-based approach for deploying service level agreement monitoring services in Cloud," *Software—Practice & Experience (SP&E)*, vol. 42, no. 4, pp. 501–518, Apr. 2012.

[26] G. Katsaros, G. Kousiouris, S. V. Gogouvitis, D. Kyriazis, A. Menychtas, and T. A. Varvarigou, "A self-adaptive hierarchical monitoring mechanism for

clouds," *Journal of Systems and Software (JSS)*, vol. 85, no. 5, pp. 1029–1041, May 2012.

[27] W. Fu and Q. Huang, "GridEye: A service-oriented grid monitoring system with improved forecasting algorithm," in *Proc. 5th International Conference on Grid and Cooperative Computing (GCC) Workshops*. IEEE, Oct. 2006, pp. 5–12.

[28] H. L. Truong and T. Fahringer, "SCALEA-G: A unified monitoring and performance analysis system for the grid," *Scientific Programming*, vol. 12, no. 4, pp. 225–237, Dec. 2004.

[29] D. Gunter and B. Tierney, "NetLogger: A toolkit for distributed system performance tuning and debugging," in *Proc. 8th IFIP/IEEE International Symposium on Integrated Network Management (IM)*. Kluwer, Mar. 2003, pp. 97–100.

[30] A. Bertolino, A. Calabrò, F. Lonetti, and A. Sabetta, "GLIMPSE: A generic and flexible monitoring infrastructure," in *Proc. 13th European Workshop on Dependable Computing (EWDC)*. ACM, May 2011, pp. 73–78.

[31] X. Zhang, J. L. Freschl, and J. M. Schopf, "A performance study of monitoring and information services for distributed systems," in *Proc. 12th International Symposium on High-Performance Distributed Computing (HPDC)*. IEEE, June 2003, pp. 270–282.

[32] D. Johnson and P. Roselli, "Using xml as a flexible portable test script language," in *Proc. Systens and Readiness Technology Conference*, 2003.

[33] b. L. Y. Y. Chonwu Jiang and C. Liui, "Study on real-time test script in automated test equipment," in *Proc. 8th International Conference on Maintainability and Safety*, 2009.

[34] Q. X. M. Grechanik and C. Fu, "Maintaining and evolving gui-directed test scripts," in *Proc. 31st International Conference on Software Engineering*, 2009.

[35] B. H. R. S. Marlon Vieira, Johanne leduc and J. Kaszmeir, "Automation of gui testing using a model-driven approach," in *Proc. International Workshop on Automation of Software Test*, 2006.

[36] J. P. Bahareh Badhan, martin Franzle and T. Teige, "Test automation for hybrid systems," in *Proc. 3rd International Workshop on Software Quality Assurance*, 2006.

[37] A. Orso and B. Kennedy, "Selective capture and replay of program executions," in *Proc. 3rd International Workshop in Dynamic Analysis*, 2005.

[38] D. P. S. P. Oren Laadan, Ricardo Baratto and J. Nieh, "A personal virtual computer recorder," in *Proc. 21st Symposium on Operating System Principles*, 2007.

[39] M. Ivory and M. Hearst, "The state of the art in automating usability evaluation of user interfaces," in *Proc. Computing Surveys*, 2001.

[40] R. W. Stefan Berner and R. K. Keller, "Observations and lessons learned from automated testing," in *Proc. 27th International Conference on Software Engineering*, 2005.

[41] T. J. Lorenzen and V. L. Anderson, *Design of Experiments: A No-Name Approach.*

[42] B. T. Stephanie Fraley, Mike Oom and J. Zalewski, *The Michigan Chemical Process Dynamics and Controls Open Text Book.* Available Online, 2007.

[43] H. d. M. Gunter Bolch, Stefan Greiner and K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications.*

[44] U. N. Bhat and G. K. Miller, *Elements of Applied Stochastic Processes*, 3rd ed. John Wiley and Sons, 2002.

[45] M. Black and Y. Yacoob, "Recognizing facial expressions in image sequences using local parameterized models of image motion."

[46] M. Ostendorf and H. Singer, "Hmm topology design using maximum likelihood successive state splitting."

[47] G. Cormack and R. Horspool, "Data compression using dynamic markov modeling."

[48] D. Goldberg and M. J. Mataric, "Coordinating mobile robot group behavior using a model of interaction dynamics," in *Proc. 3rd International Conference on Autonomous Agents*. IEEE, May 1993, pp. 1–5.

[49] J. S. Aleksandar Lazarevic and V. Kumar, "Pakdd 2004 tutorial: Data mining for analysis of rare events: A case study in security, financial and medical applications," in *Proc. Pacific-Asia Conference Knowledge Discovery and Data Mining*, 2004.

[50] C. Ling and L. C, "Data mining for direct marketing: Problems and solutions," in *Proc. International Conference of Knowledge Discovery and Data Mining*, 1998.

[51] K. M and M. S, "Addressing the curse of imbalanced training sets: One sided selection," in *Proc. International Conference of Machine Learning*, 1997.

[52] L. H. N. Chawla, K. Bowyer and P. Kegelmeyer, "Smote: Synthetic minority over-sampling technnique."

[53] N. Chawla and A. Lazarevic, "Smoteboost: Improving the prediction of minority class in boosting," in *Proc. Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2003.

[54] P. Domingos, "Metacost: A general method for making classifiers cost-sensitive," in *Proc. International Conference of Knowledge Discovery and Data Mining*, 1993.

[55] J. Z. W. Fan, S. Stolfo and P. Chan, "Adacost: Misclassification cost-sensitive boosting," in *Proc. International Conference of Machine Learning*, 1999.

[56] K. Ting, "A comparative study of cost-sensitive bossting algorithms," in *Proc. International Conference of Machine Learning*, 2000.

[57] Y. Z. N.Ye and C. Borror, "Robustness of the markov chain model for cyber attack detection," *IEEE Transactions on Reliability*, vol. 53, no. 1, pp. 116 – 123, Mar. 2004.

[58] Q. Z. N. Ye and M. Xu, "Probabilistic networks with undirected links for anomaly detection," in *Proc. SMC Information Assurance and Security Workshop*, 2000.

[59] W. DuMouchel, "Computer intrusion detection based on bayes factors for comparing command transition probabilities," *National Institute of Statistical Sciences [Available: http://www.niss.org/download-abletechreports.html]*.

[60] G. G. S.M. Sharafi and S. Emadi, "An analytical model for performance evaluation of software architectural styles," in *2nd International Conference on Technology and Engineering*, 2010.

[61] V. Sharma and K. Trivedi, "Reliability and performance of component based software systems with restarts, retries, reboots and repairs," in *17th International Symposium on Software Reliability*, 2006.

[62] X. Wang and A. Ray, "Signed real measure of regular languages," in *Proc.of American Control Conference*. IEEE, May 2002.

[63] A. Ray and S. Phoha, "A language measure of discrete event automata," in *International Federation of Automatic Control World Congress*. IEEE, July 2002.

[64] A. R. Hong Yiguang, K.S. Trivedi and S. Phoha, "Sofwtare performance analysis using a language measure," in *Proc.of American Control Conference*, 2003.

[65] R. Jain, *The Art of Computer Systems Performance Analysis.* John Wiley and Sons, Inc., 1991.

[66] E. Weyuker and A. Avritzer, "A metrics for predicting the performance of an application under growing workload," *IBM Systems Journal*, vol. 41, no. 1, pp. 45–54, Sept. 2002.

[67] C. S. K. G. H. S. Marcelo De Barros, Jing Shiau and J. Forsmann, "Web service wind tunnel: On performance testing large scale stateful web services," in *International Conference on Dependable Systems and Networks.* IEEE, June 2007, pp. 612 – 617.

[68] A. Avritzer and E. J. Weyuker, "The automatic generation of load test suites and the assessment of the resulting software," *IEEE Transactions on Software Engineering (TSE)*, vol. 21, no. 9, pp. 705–716, Sept. 1995.

[69] A. Avritzer and B. Larson, "Load testing software using deterministic state testing," in *Proc. International Symposium on Software Testing and Analysis (ISSTA).* ACM, June 1993, pp. 82 – 88.

[70] T. Berling and P. Runeson, "Efficient evaluation of multifactor dependent system performance using fractional factorial design," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 769 – 781, Sept. 2003.

[71] A. F. Karr and A. A. Porter, "Distributed performance testing using statitical modeling," in *Proc. International Workshop on Advances in Model based Testing.* ACM, July 2005, pp. 1 –7.

[72] D. P. B. Prabu Dorairaj and L. P. Kantam, "Performance tuning of storage system using design of experiments," in *Proc. 33rd International Computer Measurement Group Conference*, Dec. 2007.

BIOGRAPHICAL STATEMENT

Tuli Nivas received her Master of Computer Science degree from the University of Texas at Arlington. She is currently working with Sabre Holdings Inc., a leading travel IT company with headquarters in Southlake, Texas. She is part of the Enterprise Engineering group and her focus has been performance and system engineering. Tuli has designed, architected and helped implement various solutions for running performance tests, setting up monitoring and building automation for different airline solutions. Her interests lie in different technologies including cloud and virtualization for distributed systems.

Tuli is active in writing papers and attending performance engineering conferences and has also taught classes at Sabre to teach various groups about the importance and the process of proactive engineering.