1-1-2013

Model-Based Autonomic Performance Management of Distributed Enterprise Systems and Applications

Rajat Mehrotra

Model-based autonomic performance management of distributed enterprise systems and applications

By

Rajat Mehrotra

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

December 2013

Model-based autonomic performance management of distributed

enterprise systems and applications

By

Rajat Mehrotra

Approved:

Sherif Abdelwahed
(Major Professor)

Ioana Banicescu
(Committee Member)

Thomas H. Morris
(Committee Member)

Bryan A. Jones
(Committee Member)

James E. Fowler
(Graduate Coordinator)

Achille Messac
Dean
Bagley College of Engineering

Name: Rajat Mehrotra

Date of Degree: December 14, 2013

Institution: Mississippi State University

Major Field: Computer Engineering

Major Professor: Dr. Sherif Abdelwahed

Title of Study: Model-based autonomic performance management of distributed enterprise systems and applications

Pages of Study: 226

Candidate for Degree of Doctor of Philosophy

Distributed computing systems (DCS) host a wide variety of enterprise applications in dynamic and uncertain operating environments. These applications require stringent reliability, availability, and quality of service (QoS) guarantee to maintain their service level agreements (SLAs). Due to the growing size and complexity of DCS, an autonomic performance management system is required to maintain SLAs of these applications.

A model-based autonomic performance management structure is developed in this dissertation for applications hosted in DCS. A systematic application performance modeling approach is introduced in this dissertation to define the dependency relationships among the system parameters, which impact the application performance. The developed application performance model is used by a model-based predictive controller for managing multi-dimensional QoS objectives of the application. A distributed control structure is also developed to provide scalability for performance management and to eliminate the requirement of approximate behavior modeling in the hierarchical arrangement of DCS.

A distributed monitoring system is also introduced in this dissertation to keep track of computational resources utilization, application performance statistics, and scientific application execution in a DCS, with minimum latency and controllable resource overhead. The developed monitoring system is self-configuring, self-aware, and fault-tolerant. It can also be deployed for monitoring of DCS with heterogeneous computing systems.

A configurable autonomic performance management system is developed using model-integrated computing methodologies, which allow administrators to define the initial settings of the application, QoS objectives, system components' placement, and interaction among these components in a graphical domain specific modeling environment. This configurable performance management system facilitates reusability of the same components, algorithms, and application performance models in different deployment settings.

Key words: Performance Management, Distributed Monitoring, Distributed Control Structure, Model Integrated Computing, and Component Based Approach.

DEDICATION

To my family for their constant love and support.

ACKNOWLEDGEMENTS

I would like to thank each and every individual who helped and supported me directly or indirectly during my graduate education at Mississippi State University.

I would like to begin by expressing my gratitude to Dr. Sherif Abdelwahed, my advisor and mentor, for his constant support and encouragement, knowledge sharing, and helpful ideas in solving complex research problems. This dissertation would not have been possible without his able guidance and support. I would like to thank him again for giving me an opportunity to perform research in his research group.

I would like to thank Dr. Ioana Banicescu for her able guidance and consistent support on multiple occasions during the course work, project collaboration, and dissertation development. I would also like to thank Dr. Thomas H. Morris and Dr. Bryan A. Jones for providing me great insight and support by enriching my dissertation proposal through their valuable feedback. I am also thankful to the faculty and staff of the Department of Electrical and Computer Engineering and the NSF Center for Cloud and Autonomic Computing at MSU for their help and support during my graduate studies.

Next, I would like to thank Dr. Abhishek Dubey from ISIS, Vanderbilt University for constant guidance in solving research problems and offering me research opportunity in his group. I would also like to thank Dr. Asser N. Tantawi from IBM T. J. Watson Lab for his research ideas during formative years of this dissertation.

I would like to thank my research group members Rui Jia, Qian Chen, Ranjit Amgai, Jian Shi, Zimin Wang, and Srishti Srivastava for interesting discussions and valuable suggestions. I would also like to thank all of my friends in Starkville and India for their continuous supply of support and encouragement.

This dissertation has been supported by multiple organizations in parts – NSF SOD: CNS-0804230; NSF IIP-1034897; NSF IIP-1127978; Qatar National Research Foundation NPRP: 09-778-2299, and DoE SciDAC-II: DE-FC02-06ER41442 (Vanderbilt University).

Finally, I would like to express special thanks to my family for their unconditional love, support, encouragement, guidance, and patience that brought me so far in life.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES xi

LIST OF FIGURES xii

CHAPTER

1. INTRODUCTION 1

1.1 Motivation 1
1.2 Challenges 2
1.3 Research Hypothesis 3
1.4 Research Approach 4
1.5 Contributions of the Dissertation 6
1.6 Scope and Limitations 7
1.7 Dissertation Organization 8

2. PRELIMINARIES AND RELATED WORK 11

2.1 Autonomic Computing 12
2.1.1 Architecture of Autonomic Computing 13
2.1.2 Development of Autonomic Computing Systems 15
2.1.2.1 Artificial Intelligence (AI) Based Techniques 15
2.1.2.2 Control Theory Based Techniques 16
2.1.2.3 Bio-inspired Techniques 18
2.1.2.4 Dissertation Contribution 20
2.1.3 Autonomic Computing Projects 21
2.2 Introduction to Web Services 24
2.2.1 Web Service Architecture 24
2.2.2 Service-Level Agreement for Web Services 26
2.2.3 Web Service Used in Dissertation 27
2.3 Power Consumption in Computing Systems 28

2.3.1 Power Consumption Modeling 29
2.3.2 Power Management Techniques 31
2.3.2.1 CPU Power Management 31
2.3.2.2 RAM Power Management 33
2.3.2.3 Other Methods for Power Consumption Management 34
2.3.2.4 Power Management in Dissertation 35
2.4 Model Integrated Computing (MIC) 36
2.4.1 Domain Specific Modeling Language (DSML) 37
2.4.2 MIC Tools for Domain Specific Modeling 38
2.4.3 Code Generation using Model Interpreters 39
2.4.4 MIC in Dissertation 40
2.5 Component-based Performance Management Solutions 40
2.6 Summary 45

3. A PERFORMANCE MODELING APPROACH FOR A MULTI-TIER WEB SERVICE SYSTEM 46

3.1 Related Work 46
3.2 Preliminaries 48
3.2.1 Queuing Models for Multi-Tier Systems 48
3.2.2 Kalman Filters 51
3.3 Web Service System Setup 53
3.3.1 System Setup 53
3.3.2 Web Service Application 54
3.3.3 Monitoring Setup 55
3.4 System Modeling Approach 55
3.4.1 Power consumption 56
3.4.2 Request characteristics 59
3.4.3 Webserver characteristics 60
3.4.4 Impact of Maximum Usage of the Bottleneck Resource . . 67
3.4.5 Impact of Limited Usage of Bottleneck Resource 70
3.4.6 Kalman Filter Analysis 72
3.5 Summary 74

4. PERFORMANCE MANAGEMENT OF A WEB SERVICE DEPLOYMENT USING A MODEL-BASED CONTROL APPROACH 75

4.1 Preliminaries 76
4.1.1 Model-Predictive Control 76
4.1.2 Control Theory based Management of Computing Systems 77
4.2 Model-Based Online Predictive Controller 78
4.2.1 System Variables 79
4.2.2 Forecasting Environment Input 81

4.2.3 Controller Model 82
4.2.4 Operating Constraints 83
4.2.5 Control Algorithm and Performance Specification 84
4.3 Case Study: Power Consumption and QoS Management of a Web Service 85
4.3.1 Experiment settings 86
4.3.2 System State for Predictive Control 86
4.3.3 Experimental Results 88
4.4 Summary 93

5. A REAL-TIME AND FAULT-TOLERANT DISTRIBUTED MONITORING SYSTEM 96

5.1 RFDMon: Real-Time and Fault-Tolerant Distributed Monitoring System 96
5.2 Preliminaries 97
5.2.1 Publish-Subscribe Mechanism 97
5.2.2 Open Splice DDS 99
5.2.3 ARINC-653 101
5.2.4 Ruby on Rails 104
5.3 Other Distributed Monitoring Systems 105
5.4 RFDMon System Architecture 109
5.4.1 Sensors 109
5.4.2 Region 110
5.4.3 Local Manager 110
5.4.4 Regional Leader 111
5.4.5 Topics 111
5.4.6 Topic Managers 112
5.4.7 Global Membership Manager 113
5.5 Sensor Implementation 114
5.5.1 Resource Utilization Monitoring Sensors 116
5.5.2 Hardware Health Monitoring Sensors 119
5.5.3 Node Health Monitoring Sensors 119
5.5.4 Scientific Application Health Monitoring Sensor 119
5.5.5 Web Application Performance Monitoring Sensor 120
5.6 Experiments Related to Monitoring System 121
5.7 Summary 125

6. A DISTRIBUTED CONTROL APPROACH FOR PERFORMANCE MANAGEMENT OF A WEB SERVICE DEPLOYMENT 126

6.1 Related Work 126

6.1.1 Advanced Large Scale Control Algorithms and Their Applications . . . 127
6.1.2 Large Scale Control Management in Web Service Environment . . . 130
6.2 A Distributed Web Service Deployment . . . 133
6.3 Distributed Model Predictive Control Problem . . . 135
6.4 A Distributed Control Approach using Interaction Balance Principle 137
6.4.1 Decomposition of the Overall Problem into N Sub Problems 140
6.4.2 Optimizing the Subsystem Level problem using Model Predictive Control . . . 144
6.4.3 Solving the Coordinator Level problem using Conjugate Gradient Method . . . 144
6.5 Evaluation of the Distributed Control Approach . . . 146
6.5.1 Performance Parameters . . . 146
6.5.1.1 Load Distribution Among Subsystems . . . 146
6.5.1.2 Subsystem Resource Utilization . . . 146
6.5.1.3 SLA Parameters . . . 147
6.5.1.4 Utility Value . . . 147
6.5.2 Robustness Toward Failure . . . 147
6.5.3 Computational Overhead . . . 148
6.5.3.1 Interaction Between the Coordinator and the Subsystems . . . 148
6.5.3.2 Computational Overhead at each Subsystem . . . 148
6.5.4 Reducing the Computational Overhead in the Proposed Approach . . . 150
6.5.4.1 Reducing the Number of Interactions . . . 150
6.5.4.2 Reducing the computations at each subsystem . . . 150
6.6 Case Study: Performance Management of a Distributed Web Service Deployment . . . 151
6.6.1 System Model . . . 151
6.6.2 Forecasting Environment Input . . . 153
6.6.3 Experiment Setup . . . 153
6.6.4 Simulation-1: Comparing the Performance of Distributed Control Approach with a Centralized Control Approach . . . 155
6.6.5 Simulation-2: Impact of the Workload Arrival rate and Error Tolerance Value on Distributed Control Approach . . . 164
6.6.6 Simulation-3: Evaluation of Robustness Towards Subsystem Failure . . . 168
6.7 Summary . . . 174

7. APPLICATION OF THE DISTRIBUTED CONTROL APPROACH AS A COMPONENT BASED DEPLOYMENT . . . 175

7.1 Preliminaries . . . 175
7.1.1 Generic Modeling Environment . . . 175
7.1.2 Universal Data Model . . . 179
7.2 Component-Based Control Structure: Development, Deployment, and Configuration . . . 181
7.2.1 Development Phase . . . 182
7.2.1.1 Meta-Model Development . . . 182
7.2.1.2 Interface Definition File Creation . . . 184
7.2.1.3 Component Specification and Development . . . 185
7.2.1.4 Component Packaging as Module and Framework . . . 186
7.2.2 Target System Configuration Phase . . . 186
7.2.3 Deployment Phase . . . 187
7.2.3.1 Deployment Plan Generation . . . 187
7.2.3.2 Deployment Preparation . . . 188
7.3 Case Study: Distributed Control Structure Design and Deployment by using Component-based Approach . . . 188
7.3.1 Control Structure Component Design . . . 189
7.3.1.1 Environment Module . . . 191
7.3.1.2 System Module . . . 191
7.3.1.3 SLA Module . . . 191
7.3.1.4 Controller Module . . . 192
7.3.1.5 Actuator Module . . . 192
7.3.1.6 Regional Coordinator . . . 192
7.3.1.7 Membership Manager . . . 193
7.3.1.8 Local Manager . . . 193
7.3.2 GME Meta-Model Development . . . 193
7.3.3 Component Library Development . . . 194
7.3.4 Control Structure Domain Application Model . . . 196
7.3.5 Application Deployment Configuration . . . 196
7.3.6 Deployment Planner Module Development . . . 198
7.3.7 Deployment Plan Generation . . . 198
7.3.8 Control Structure Deployment . . . 201
7.4 Summary . . . 203

8. CONCLUSIONS AND FUTURE RESEARCH . . . 204

8.1 Conclusions . . . 204
8.2 Future Research Directions . . . 207
8.2.1 Extended Component Library . . . 207
8.2.2 Fault Diagnosis Module . . . 207
8.2.3 Multi-Level Distributed Control Approach . . . 208

REFERENCES . . . 209

APPENDIX

A. LIST OF PUBLICATIONS 223

A.1 Journal Publications 224
A.2 Book Chapters 224
A.3 Conference Publications 224
A.4 Technical Reports 225
A.5 Posters 226

LIST OF TABLES

3.1 Web Service System Setup Configuration 54

3.2 System Parameters. 57

5.1 List of Monitoring Sensors 117

LIST OF FIGURES

2.1 Autonomic Manager Control Loop (MAPE-K). 14

2.2 Service Oriented Architecture of Web Services. 25

2.3 Service Level Agreement Life Cycle. 26

2.4 Model Integrated Computing Architecture. 36

2.5 Model Driven Development. 38

2.6 Corba Component Model (CCM) description. 41

3.1 Queueing System Structure and Parameters. 49

3.2 Power Consumption on Nop03 vs (CPU frequency and Aggregate CPU core Utilization). 58

3.3 Work Factor Plot for Request Characteristic. 59

3.4 Http Workload based upon World Cup Soccer(WCS-98) Applied to the Web Server. 60

3.5 Web Server Behavior Modeling Experiment Setup from Section 3.4.3. . . . 61

3.6 Web Server Behavior Modeling Experiment Results. 63

3.7 Web Server Behavior Modeling Experiment Results (continued from Figure 3.6). 64

3.8 A Queuing Model for the Two-Tier System. 65

3.9 Offline Exponential Kalman Filter Output from Section 3.4.3. 68

3.10 Impact of Maximum Utilization of the Bottleneck Resource on Web Server Performance. 69

3.11 Web Server Performance while Limiting the use of Bottleneck Resource. . 71

3.12 Offline ExpoKF Analysis of the Results from Figure 3.11. 73

4.1 Main Components of a General Control System. 78

4.2 Elements of the Predictive Control Approach. 79

4.3 Predictive Control Algorithm for Calculating Value of Control Input. . . . 85

4.4 Predictive Controller System Setup 87

4.5 Web Server Performance Results With Predictive Controller. 90

4.6 Comparison of Web Server Performance with Controller and Without Controller from Section 3.4. 91

4.7 Online EKF output Corresponding for experiment with Predictive Controller. 92

4.8 Comparison of Power Consumption for Actual Vs Predicted one Through Power Consumption Model in Section 3.4.1. 94

4.9 Comparison of Actual Http Workload from the Clients Vs Predicted one Through the Estimator. 94

5.1 Publish Subscribe Architecture. 98

5.2 Data Distribution Services (DDS) Architecture. 99

5.3 Data Distribution Services (DDS) Entities. 100

5.4 ARINC-653 Architecture. 102

5.5 Rails and Model View Controller (MVC) Architecture Interaction. 105

5.6 Schema of Monitoring Database (PK = Primary Key, FK = Foreign key). . 106

5.7 Architecture of the “RFDMon” Monitoring System. 110

5.8 Class Diagram of the “RFDMon” Monitoring System. 115

5.9 Life Cycle of a CPU Utilization Sensor. 118

5.10 Architecture of the Scientific Application Health Monitoring Sensor. 120

5.11 CPU and RAM Utilization at various Nodes. 122

5.12 State Transition of the Nodes and Leaders of the Distributed Monitoring System. 123

5.13 CPU Utilization at node ddsnode1 during the Experiment. 124

6.1 Interaction Balance and Interaction Prediction Approach. 128

6.2 The i-th Subsystem and its Parameters. 128

6.3 The proposed Distributed-Control Approach 138

6.4 Predictive Control Algorithm at each Subsystem. 143

6.5 Subsystem Level Control Architecture. 152

6.6 Simulation Settings. 154

6.7 Workloads $W1$, $W2$, and $W3$ were used During the Simulations. 155

6.8 Centralized Control Algorithm for Calculating Values of Control Inputs. . . 157

6.9 Simulation-1: Workload Share Comparison Between the Developed Approach and the Centralized Approach. 158

6.10 Simulation-1: Comparison of Applied Frequency (Power Consumption) Between the Developed Approach and the Centralized Approach. 159

6.11 Simulation-1: Comparison of Response Time Between the Developed Approach and the Centralized Approach. 159

6.12 Simulation-1: Comparison of Queue Size Between the Developed Approach and the Centralized Approach. 160

6.13 Simulation-1: Global Queue Size (Q) and Total Load Share Processed by all the Nodes in the Distributed Control Approach. 161

6.14 Simulation-1: Number of Interactions Between Coordinator and Nodes. . . 162

6.15 Simulation-1: Comparison of System Utility Between the Developed Approach and Centralized Approach. 164

6.16 Simulation-2: Total Load Share Processed. 165

6.17 Simulation-2: Number of Interactions. 166

6.18 Simulation-2: Interaction Error Statistics. 167

6.19 Simulation-3: Workload Arrival rate. 169

6.20 Simulation-3: Workload Share Processed by Subsystems in the Cluster. . . 170

6.21 Simulation-3: Frequency Values used by the Nodes. 170

6.22 Simulation-3: Response Time at Subsystems. 171

6.23 Simulation-3: Queue Size at Subsystems. 172

6.24 Simulation-3: Global Queue Size (Q) and Total Load share Processed by Subsystems. 172

6.25 Simulation-3: Global Queue Size and Total Load share processed by Nodes. 173

7.1 GME Architecture. 176

7.2 Computing System Dynamics Meta-Model in GME. 178

7.3 Components of the UDM Framework. 180

7.4 Component-Based Control Structure Deployment Plan Generation Process. 183

7.5 Example of an Interface Definition Language (IDL) File. 185

7.6 Key Components of the Control Framework at each Node. 190

7.7 Controller Module Meta-Model in GME (see Figure 7.6). 194

7.8 Local Controller Meta-Model in GME. (Present Inside Controller Module Meta-Model in Figure 7.7). 195

7.9 Domain Application Model of the Environment Module. 197

7.10 Code Sample from UDM-C++ Interpreter to generate the Source code with Deployment Settings for the Distributed Management System Deployment. 199

7.11 Code Sample from UDM-C++ Interpreter... (Continued from Figure 7.10). 200

7.12 Hierarchical Arrangement of the Distributed Performance Management System Deployment. 202

CHAPTER 1

INTRODUCTION

1.1 Motivation

Distributed computing systems have been used for hosting e-commerce, social networking, internet search services, and information broadcasting applications during the past few decades. Additionally, these systems are now used for hosting cloud computing services (e.g., Amazon EC2[1], Windows Azure [42], and Google Apps[15]). The growing use of applications, hosted in distributed computing systems, increases the demand to ensure the availability, the reliability, and the quality of service (QoS) of applications in dynamic and uncertain operating environments. Effective management in such operating environments requires expert administrator knowledge to determine the capacity requirement and resource allocation based on incoming workload pattern and application behavior; however, manual administration leads to poor performance and frequent outages due to the extremely complex application dynamics and the large size of the deployment.

In general, distributed computing systems also impose consistency and synchronization requirements over multiple computing nodes while exchanging measurements related to system resource utilization and application performance statistics. This requires an aggregate picture of the distributed systems to be available for analyzing and providing feedback to compute control commands by the management systems. This aggregate view can

be constituted through an efficient distributed infrastructure monitoring system that is extensive enough to monitor system resource utilization, hardware health, and application performance with minimum latency.

Recently, research communities and academia have been successful in designing and developing large scale distributed systems, which address the availability, reliability, and QoS management issues of the hosted applications without any human intervention. These self-managing systems compute and execute management commands themselves in case of QoS objective violations. IBM's *Autonomic Computing* [4], Microsoft's *Dynamic Systems* [20], HP's *Adaptive Infrastructure* [17], and Intel's *Proactive Computing* [18] are some examples of ongoing industry efforts to realize these self-managing systems in future. A class of techniques investigated by both academia and industry to build such systems rely on the use of mathematical models, which represent the system's behavior. These techniques include artificial intelligence (e.g., prediction, search, planning, and reinforcement learning), feedback control theory, and formal models. Typically, these techniques take corrective actions to meet the stipulated system performance objectives by using the application measurements and the developed system model.

1.2 Challenges

Performance of a computing system and hosted application can be estimated with certain accuracy by utilizing models of the system resources and its service configurations. These models represent the relationship among environment inputs, control inputs, current system state, and their impact on system performance. Precise knowledge of relevant sys-

tem parameters, their mutual dependencies, and constraints are required to develop these system performance models; however, the development of an accurate performance model is difficult due to the dynamic nature of the operating environment, the complex relationships among system parameters (controllable and uncontrollable), and their indirect impact on the application performance.

Another major challenge in distributed computing systems is that a wide variety of applications are hosted in the distributed environments. These applications include stock trading, medical infrastructure, news portal, media streaming, and storage services. These applications are different from each other in the deployment architecture (data center or cloud environment), environment inputs (session-based, session-less, or different arrival pattern), monitored components (software or hardware), system performance model, and QoS requirements. Moreover, these applications are deployed with different operational settings and placement strategies for their components. Therefore, the development of a single performance management system to maintain the QoS of each of these applications in different deployment schemes is infeasible.

1.3 Research Hypothesis

The research hypothesis for this dissertation is formulated as follows. The QoS of distributed applications can be *maintained efficiently* in dynamic operating environments if a *performance model* of the application is applied in association with a *model-based distributed control approach* and an *efficient distributed monitoring system*.

This research hypothesis is based on the premise that well-established control theoretic approaches, coupled with model-integrated computing practices, can be applied for design, development, and maintenance of an autonomic performance management system. This performance management system can be further used for maintaining the QoS objectives of a general class of applications.

1.4 Research Approach

As a solution to the previously mentioned performance management issues in distributed application deployment, a generic autonomic performance management system needs to be designed. This generic performance management system can be configured as per the requirements of the deployed application and the operating environment. Additionally, this configurable management system should be applicable to a large set of applications to maintain a desired performance guarantee. This management system should also encourage reusability of the developed modules to increase the productivity of the development process and to minimize the duplication of effort.

This dissertation is aimed to develop state-of-the-art techniques for building a generic autonomic performance management system in order to maintain the QoS requirements of distributed applications with limited human intervention. Another objective of this dissertation is to develop component-based performance management systems that improve the ability to design, implement, maintain, and deploy an autonomic performance management systems for large scale enterprise applications. A number of results from earlier research

are used in developing this dissertation from following areas: Autonomic Computing, Data Distribution Services, and Model-Integrated Computing.

Feedback control-based approach is used to develop an autonomic performance management system, which can be deployed with the developed application performance model. In this case, the control changes can be verified on the developed performance model before applying them to the actual system.

Data distribution services (DDS) is a middleware communication standard for real-time systems. It overcomes the typical shortcomings of a traditional client-server model, where clients and servers are coupled together for data exchanges. DDS gives complete control of data to the application developers by providing an extensive list of QoS configurations in the data. It provides scalability in size of the distributed system for monitoring and ensures minimum resource overhead with maximum bandwidth utilization while exchanging the measurements among computing nodes.

Model-integrated computing techniques and tools are used in this dissertation to develop formal models for each component of the developed performance management system. These models are further utilized during the system development process to capture the service requirements, deployment configurations, and environment dynamics. Model-integrated computing tools are utilized to develop a semi-automatic procedure for generating the deployment plan for the performance management system according to the application deployment configurations.

1.5 Contributions of the Dissertation

This dissertation is focused on developing a model-based autonomic performance management system for enterprise applications hosted in distributed computing environments. The key research contributions of this dissertation are as follows.

A systematic approach is introduced to create performance models for distributed multi-tier enterprise applications as part of this research. This approach identifies the system parameters through extensive experimentation, defines the relationship among these parameters, and identifies the underlying model structure of the system. The developed performance model is further utilized by a model-based predictive controller for the autonomic performance management of the deployed application.

A distributed monitoring system "RFDMon" is developed for monitoring system resources (e.g., CPU, memory, disk, network channels), system health (e.g., temperature and voltage of motherboard and CPU core), application performance statistics (response time, queue size, and throughput), and scientific application execution state with minimum latency and controllable system resource overhead. This monitoring system can be deployed for monitoring of distributed infrastructure with heterogeneous computing systems.

A distributed control-based performance management approach is developed for managing a general class of applications hosted in distributed environment. In this approach, the infrastructure level performance management problem is solved by decomposing the overall control problem into various different local control problems of managing the application instances at each computing node through node level controllers, while satisfying the constraints posed by the controller working at immediate higher level. This approach

provides scalability in size of the infrastructure and eliminates the requirement of approximate behavior modeling of lower level controllers in case of hierarchical deployment.

A component-based distributed control approach is developed in this dissertation by using model-integrated computing methods. This component-based implementation can be applied to a large set of distributed applications after defining the initial settings of the application, the multi-dimensional QoS objectives, the application performance models, the components' placement, and interaction among these components. Component-based implementation eliminates the need of applications specific development of performance management systems.

1.6 Scope and Limitations

A component-based autonomic performance management system is developed during this research for maintaining the SLAs of distributed enterprise applications in dynamic operating environment. To develop this management system, this dissertation is focused on these specific areas:

- The development of performance models of the multi-tier enterprise applications that can be updated dynamically in the actual production environment.
- The development of a non-intrusive power consumption model that estimates power consumption of the multi-core system with high accuracy by using the hosted application characteristics and utilization of relevant system resource.
- The development of a model-based predictive controller that can utilize these application performance and power consumption models to compute the optimal values of control inputs for maintaining the SLAs of the hosted application with minimum operating cost.
- The development of a distributed monitoring system that can be utilized for comprehensive monitoring of the resource utilization and application performance statistics at each computing node in a distributed infrastructure. This monitoring system can

also be used for monitoring the execution state of a scientific application in a cluster or grid environment.

- The development of a distributed control approach that can utilize the measurements collected through the distributed monitoring system to maintain the SLAs of the web service applications instances at each computing while minimizing the operating cost of the entire infrastructure.
- The development of a configurable distributed control structure where an administrator can define the SLAs of the deployed distributed application, initial settings, and interaction among the components of the distributed control structure.

The applicability and performance of the contributions of this research are demonstrated by applying it on a distributed multi-tier web service deployment. The developed models, monitoring system, and control structures are tested in the academic lab environment for managing a multi-tier web service deployment with different settings. However, due to the scalable nature of the developed monitoring system and distributed control approach, the performance management system can be easily applied to the large scale distributed computing systems in production environment after making appropriate configurations. Moreover, the generic and customizable nature of the developed component-based management system makes it a candidate for other domains as well, such as cloud computing and big data analysis platforms.

1.7 Dissertation Organization

The remainder of the dissertation is organized as follows:

A detailed introduction of autonomic computing systems, web service architecture, and model-integrated computing are presented in Chapter 2. This chapter also highlights earlier research efforts of academia and industry in developing physical server power consump-

tion management techniques and component-based performance management solutions for distributed computing systems.

A systematic approach for developing performance models of multi-tier applications is presented in Chapter 3. This chapter covers the previous research efforts in developing application performance models, basic concepts of queueing theory, and Kalman filters to understand the developed approach. Moreover, the physical system power consumption and http request characteristic modeling efforts of the dissertation are highlighted in this chapter. A set of experiments are also discussed in this chapter to demonstrate the accuracy of the developed performance model and to investigate the impact of variation in system parameter values on the application performance.

Applicability of the developed performance model is presented in Chapter 4. A model-based predictive controller uses the developed application performance model to manage QoS objectives of the deployed application while minimizing the power consumption of the physical server. This chapter also presents earlier research efforts in managing performance of the deployed multi-tier applications using model-based control approaches.

Architecture of a real-time and fault-tolerant distributed monitoring system is presented in Chapter 5. This monitoring system was developed during this research in order to monitor the computing nodes of the distributed infrastructure with minimum latency at a pre-specified interval. Additionally, this chapter also provides a brief introduction of publish-subscribe communication mechanism, data distribution services, ARINC-653 concepts, and ruby on rails web service development framework. Moreover, some industry-wide

popular distributed monitoring products are compared with the monitoring system developed in this dissertation.

A novel distributed control structure for performance management of distributed applications is presented in Chapter 6. This chapter also discusses earlier research for managing applications deployed in distributed systems by using traditional control theoretic approaches. Experimental results are also presented to demonstrate the efficiency of the developed distributed control structure and its fault-tolerant properties.

A component-based implementation of the distributed control structure (developed in Chapter 6) is introduced in Chapter 7. This semi-automatic approach for the deployment of the distributed control structure uses model-integrated computing (MIC) tools. This chapter briefly discusses MIC tools ("GME" and "UDM") and their applicability in development of the component-based performance management system.

Finally, conclusions and the research contributions are summarized in Chapter 8. Additionally, a proposed direction for future research to extend this dissertation is discussed in this chapter.

CHAPTER 2

PRELIMINARIES AND RELATED WORK

Distributed computing systems are difficult to manage due to large infrastructure size, system complexity, dynamic operating environments, and multi-dimensional QoS objectives. Autonomic computing [98] empowers researchers to deal with these management challenges by employing formal mathematical techniques derived from biological systems. Autonomic computing aims to introduce the self-management characteristics (e.g., self-configuration, self-optimizing, self-protections, and self-healing) in computing systems to maintain performance objectives of the hosted application.

Distributed computing systems are used to host a wide range of applications for e-commerce, health infrastructure, telecommunication, real-time systems, parallel scientific computing, and networking applications. Each of these computing systems consist of a large number of components, which interact with each other for common system wide objectives. Therefore, various component-based performance management solutions have been developed for performance management of these computing systems.

This chapter introduces the basic concepts and earlier research efforts in developing an autonomic performance management system for applications hosted in a distributed environment. These concepts are described in following subsections.

2.1 Autonomic Computing

Autonomic computing systems are inspired from the self-management features of the human nervous system, which monitors a large number of parameters inside and outside the body, analyzes the impact of changes in these parameters, and counters these changes with an appropriate response if necessary. These parameters include glucose concentration in the blood, blood pressure, body temperature, heart rate, breathing, etc. Moreover, the nervous system also classifies these parameters into different categories of urgency based on their impact on the survivability of the organism [120]. Changes in these parameters are countered by the nervous system to keep their values in the normal range for survivability of the body. These monitoring, analysis, and reaction are part of self-management activities, which do not require any human notice or intervention.

The *Autonomic Computing* vision was presented by Kephart et. al in 2003 [98]. The authors of vision document argued that computing systems should manage themselves according to the system level objectives specified by the human administrators. In addition, systems should perform the regular maintenance and optimization tasks without any human intervention. The authors further added that these systems should also be able to integrate the new components in the system smoothly without any glitches. This vision emphasized that autonomic computing is the only viable way to tackle the rapidly increasing complexity and size of computing systems while keeping the total cost of the ownership at a manageable level with limited administrator skills. According to IBM White paper in 2002 [87], a computing system can be considered autonomic only after completing five stages of system management: *Basic, Managed, Predictive, Adaptive,* and *Autonomic*.

These stages correspond to the level of computing system for self-management characteristics of autonomic computing, such as *self-configuring, self-optimizing, self-protecting,* and *self-healing*.

In the past decade, list of autonomic characteristics has been extended with a large number of new properties [140]: self-anticipating, self-assembling, self-adapting, self-diagnosing, self-defining, self-governing, self-installing, self-reflecting, self-planning, self-learning, self-organizing, and self-simulating.

2.1.1 Architecture of Autonomic Computing

According to the vision document [98], autonomic computing systems are composed of multiple autonomic elements (AEs). An AE operates according to the pre-specified policies of the system and provides desired computational services to the other AEs connected to itself. Each of these AEs manages a system element, such as computational resources, networking resources, storage resources, etc. A typical autonomic control loop "MAPE-K" is shown in Figure 2.1 [2].

In the "MAPE-K" system described in [2], an autonomic computing functionality is achieved through control loops, which consist of Monitor, Analyze, Plan, Execute, and Knowledge components. In the MAPE-K autonomic loop, the managed system represent a system component (e.g., system resource, web server, cluster of computing nodes, and software) under observation. The managed system is coupled with the autonomic manager through measurement sensors and system actuators. *Sensors* monitor the system parameters and publish the measurements to an autonomic manager, while *Actuators* per-

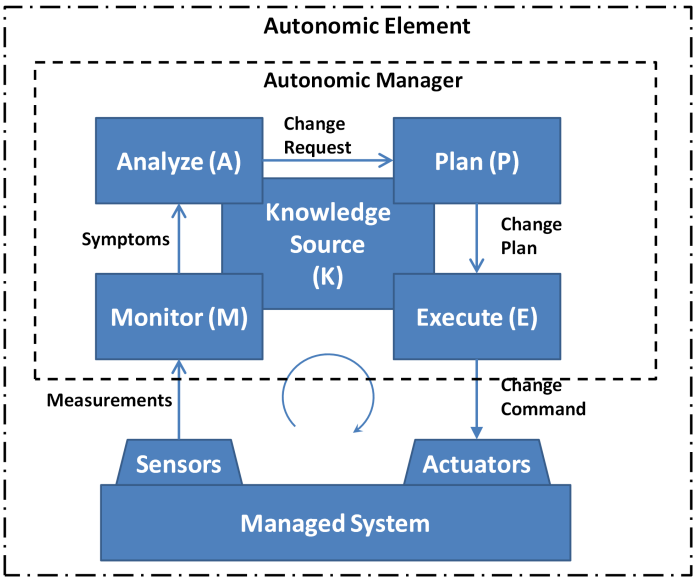


Figure 2.1

Autonomic Manager Control Loop (MAPE-K).

form changes in settings or configuration of a system component. The *Monitor* collects the measurements from the managed element through *Sensors*, filters the useful measurements, aggregates the measurements to determine the current symptom of the managed system, and forward it to the *Analyze* component. The *Analyze* component determines if the managed system is complying with the system performance objectives. In case of an objective violation (or violation expected in near future), the *Analyze* component determines the changes needed in the managed element and sends these change requests to the *Plan* component. The *Plan* component creates or selects the procedure to enact the changes in the managed system as suggested by the *Analyze* component. The *Plan* component generates a set of changes in the managed element and forwards it to the *Execute* component. The *Execute* component provides the mechanism to execute the changes on

the managed system by actuators. Additionally, it updates the stored system knowledge that will be used by the autonomic managers. The *Knowledge* component is a repository of usual system behavior, business policies, and system performance models.

2.1.2 Development of Autonomic Computing Systems

Autonomic computing systems are designed by using techniques derived from formal, mathematical, and biological systems. These techniques include artificial intelligence, feedback control, formal models, and bio-inspired algorithms. Major research efforts of academia and industry for the development of autonomic computing systems can be summarized in following categories:

2.1.2.1 Artificial Intelligence (AI) Based Techniques

Several autonomic computing technologies and approaches are based on concepts and techniques derived from AI for developing an automated decision making module. AI techniques can be used to design an intelligent agent that can understand, predict, and act based upon changes in the environment. This intelligent agent selects actions (based on previous history) that are expected to maximize agent's objectives [130]. A framework called *Clockwork* [128] was introduced in 2003 for predicting future loads by using an autoregressive integrated moving average (ARIMA) filter in a file sharing scenario over distributed computing systems. This research demonstrated the feasibility of using a predictive technique that uses past events to learn about future loads and to determine the appropriate assignment of incoming loads for load balancing in file sharing systems.

Automated Planning is a branch of artificial intelligence that investigates action sequences performed by intelligent agents aiming to drive themselves from an initial state to a final state while considering a predefined set of constraints. In static (or known) environments, planning is performed by using environment models and an agent's goal. In dynamic environments, planning is performed by using dynamic models of the environment, variable policies in possible situations, and possible plans to determine a single plan or a set of plans [129]. CHAMPS [96] and ABLE [61] are two planning-based autonomic computing systems, which provide support to system administrators for automating the configuration process in an IT infrastructure.

Reinforcement learning (RL) is another technique that is used frequently in AI, where an intelligent agent can associate its action with a corresponding reward. The use of RL for automatic learning of management policies are presented in [141, 142], where an underlying system aims for resource allocation in the computing systems. The authors have performed a series of approximations to reduce the state-space size to apply RL algorithms in trading applications.

2.1.2.2 Control Theory Based Techniques

One of the main objectives of an autonomic computing system is to optimize the system behavior through configuring the managed system in real time. Existing approaches from feedback-based control theory can be utilized to achieve these objectives due to the presence of a feedback loop in an autonomic computing architecture (see Figure 2.1). Performance management of distributed computing systems can be automated using control

theory concepts, which can provide substantial improvement compared to rule-based and heuristic-based approaches. Control theory provides a systematic way to solve dynamic resource provisioning problems by using basic control concepts and verify the applicability of these solutions before its actual application to the system deployed in the production environment [127]. For this purpose, a system model is required to identify the relationship between input and output variables of the system for a specific system setting.

Researchers have applied classical feedback control theory in developing performance management solutions for computational resource intensive applications. These approaches continuously monitor the application service level agreements (SLAs) and take appropriate actions in the case of (or possibility of) SLA violations in the near future. These techniques have been applied to applications related to different domains, such as CPU power management [103], task scheduling [64], file server load balancing [111], and bandwidth allocation in web servers [53, 48].

In more complex subsystems, inflexible feedback maps and pre-specified resource allocation plans do not accommodate the changing resource requirements due to an extremely dynamic operating environment. Additionally, classical feedback control is not suitable for non-linear and hybrid computing systems, which have complex dynamics and potentially use a finite set of control options. As a result, researchers have developed advanced state-space methods derived from model predictive control [112] and limited lookahead supervisory control [66] for managing these complex applications. These advanced approaches provide a framework, which solves multi-dimensional non-linear objective functions by using finite control input sets in a dynamic operating environment while maintaining the

application SLAs. A detailed survey of feedback control based autonomic computing approaches is presented in [86].

2.1.2.3 Bio-inspired Techniques

Autonomic computing systems are expected to maintain their operation even in the case of a changing workload, environment configurations, system goals, and hardware or software faults by monitoring various system resources and performance objectives. Additionally, these autonomic computing systems should be able to update their protection capabilities for future threats and be able to roll back to a previous state if these updates are not suitable for the safe system operation. Bio-inspired computing enables researches to look at biological organizations (natural models) for inspiration while adapting and improving it with help of the advancement in computing system technology [82]. Various performance management, intrusion detection, policy management, and network routing issues have been solved using bioinspired algorithms in computer science. These algorithms are inspired from natural elements (e.g., ant, bees, termites, flocks of birds, and human antibodies), which show self-adaptive and emergent nature by combining the interactions among thousands of individuals to achieve common system objectives.

The analogy between autonomic computing systems and biology has been used for the development of future unmanned aerial systems, where multiple pico-class swarm space crafts will be sent for asteroid missions from NASA [144]. Each one of these small swarms will carry the model of the work it is supposed to perform. Each of them will record the measurements and forward these measurements to the ruler element of the group. The

ruler element will choose the type of asteroid and measurement for further coordination among the swarms. Another bio-inspired approach was used in a hierarchical policy based management system (PBMS) for developing an autonomic communication system [58]. This system utilized the ability of a cell to recognize other similar cells to form a tissue and to maintain the equilibrium through self-management capabilities. The developed PBMS was applied to a communication system network, where multiple communication nodes (routers) join the infrastructure dynamically and comply with the overall infrastructure business policy while performing the assigned work of data forwarding.

Issues of intrusion detection in computing systems are also solved successfully through various bio-inspired approaches, which adaptively learn the differences between safe and unsafe system states or network traffic patterns. These approaches select the appropriate corrective measures for a system based on the biological rules in the nature. The natural immune system is inherently diverse, distributed, error tolerant, dynamic, adaptive, and self-protecting in behavior, which makes it suitable for designing an efficient network intrusion detection system [83]. The natural immune system based intrusion detection approaches were presented in [83, 60] where abnormal behavior were considered as intrusion into the system. Genetics-based machine learning approaches for intrusion detection were used for learning the classification of normal and abnormal behavior [137, 68]. A genetic algorithm and decision tree based approach was used in order to generate the rules for classifying the incoming network connections as normal or abnormal [137]. In this approach, genetic algorithm and decision trees were used to evolve the rules by matching the patterns of attribute values (e.g., source IP address, source IP port, destination IP address, destina-

tion IP port, and network protocol) as normal and abnormal patterns. A detailed survey of various bio-inspired intrusion detection techniques is available in [134].

2.1.2.4 Dissertation Contribution

In general, AI techniques require significant computation to learn the system performance model and to compute the appropriate control parameters for maintaining the desired performance guarantee of the computing systems in dynamic operating environments. These large computations also require availability of significant amount of computational resources to the management system. Therefore, AI techniques are not well-suited for applying in real-time production environment, where the management system is required to react on the changes in the operating environment in strict time constraints with limited amount of system resources available to the management system. Similarly, the natural models used in bio-inspired techniques, take a large number of iterations for deriving (emerging) the optimal solutions for the managed system. In addition, natural models are too specific to capture the performance behavior of a general class of applications for solving performance management issues. Moreover, it is difficult to accommodate the operating and state constraints in the AI and bio-inspired techniques while computing the control commands. In addition, the techniques utilized in management system, should also be able to take (or compute) control action at the same rate as rate of change in the operating environment; however, AI and bio-inspired techniques require a large computation time in a certain cases to compute the appropriate control command.

In contrast to the AI and bio-inspired techniques, the model-based feedback control techniques are well-suited for applying in real-time production environment. The primary benefits of using model-based feedback control approach is that the computational overhead and system resource utilization, during computation of control commands, can be managed (increased or decreased) by using application performance models of varying complexity. Furthermore, the state and operating constraints of the managed application and underlying system can also be accommodated in the developed control technique easily. Therefore, an autonomic model-based performance management system is developed in this dissertation by using model-based control theoretic approaches, which utilize a mathematical performance model of the managed web service. This performance management system has following autonomic computing characteristics:

1. *Self-configuring* with respect to the number of nodes under consideration for monitoring and managing the QoS objectives.
2. *Self-optimizing* with respect to the system performance by appropriately choosing the controlled parameters for maintaining the system performance according to desired QoS specifications.

2.1.3 Autonomic Computing Projects

In past decades, industry and academia have developed a number of advanced products and prototypes, which demonstrate some of the autonomic computing characteristics. Some of these widely known products are listed below:

1. *OceanStore* [151] (University of California, Berkeley): This project developed a global persistent data storage, where any computer can join the infrastructure through service providers. Service providers can purchase the storage as and when necessary. It creates local replica of stored data on servers that localizes access traffic to reduce network congestion. In addition to this, it pro-actively transfers data to maintain high availability in outages and malicious attacks. OceanStore demonstrates the self-healing and self-optimization features of autonomic computing.

2. *Computer Operations, Audit, and Security Technology* (COAST) [6] (Purdue University): It is a combination of multiple security projects that includes the development of an agent-based intrusion detection system, audit trails, vulnerability logs, and a security archive for self-protection.

3. *Automate* [54] (Rutgers university): This project is an enhancement of already existing grid middleware to support context aware grid applications, which have autonomic computing properties. In this project, applications are developed as composition of autonomic components over Open Grid Service Architecture [30]. This project focuses on self-configuring, self-optimizing, and self-adapting features.

4. *Autonomia* [71] (University of Arizona): This project provides various tools to application developers for specification of application attributes (e.g., performance, fault, and security) and software or hardware resources. Application developers can also configure online monitoring systems and middleware services to add autonomic properties in network services. This project demonstrates self-configuring, self-deploying, and self-healing properties by configuring management schemes.

5. *Storage Tank* [116] (IBM Research): This project designs a distributed storage management system that provides access to meta-data of the stored data first and then direct access to data using high speed networks. This project focuses on resource optimization, data consistency, policy based storage, checkpoint-based recovery, and self-healing in distributed storage systems.

6. *Q-Fabric* [122] (Georgia Institute of Technology): This project developed cooperating OS-level components that manage resource allocation to the multimedia applications for minimal jitter and best achievable image quality. It creates event channels to set up communication between the monitors and the management modules for resource management. This project focuses on self-organization and self-optmization features only.

7. *SMART (DB2)* [110] (IBM Research): This project focuses on making IBM DB2 database self-managing for reducing the total cost of ownership by minimizing administrator efforts during database design, deployment, maintenance, fault detection, and recovery. In long term, SMART will cooperate with other IBM components of the system (e.g., IBM Websphere Application Server [41]) in order to develop a complete autonomic system.

8. *Sabio* [123] (IBM Research): It is a knowledge management software that classifies large data files by using neural network and uses this classification for self-organization features in data files.

9. *SUN-N1* [35] (SUN Microsystems): SUN N1 range of products automate the provisioning of computation, networking, and storage resources as per the demands from the service and pre-specified business policies. These products support automatic

recovery from server failure. Additionally, these products perform auto-discovery of servers, monitor server health, maximize server utilization, and collect event logs.

10. *Neuromation* [25] (Edinburgh University): In this project, algorithms are developed to structure data based on the concepts and thoughts behind generating the data by using simple autonomous units. This structuring of data is independent from the application, which is using this data. In this project, each piece of the information is stored as thoughts; thoughts are connected together by using developed proprietary algorithms for storage.

11. *ebiquity* [11] (University of Maryland): In this project, computing systems are created with cooperation from dynamic, adaptive and autonomous components. These components become aware of each other for data communication, information exchange, and cooperation to complete their individual and global task.

12. *Auto Admin* [3] (Microsoft Research): The primary focus of this project is to create a large scale self-tuning and self-administering database that monitors the application's resource requirements in order to decrease the total cost of ownership. In this research, a database tracks application and auto tunes itself to complete workload requirements posed by the application. This project also develops database monitoring tools, which have minimum CPU and memory overhead.

The above mentioned research projects introduce various autonomic characteristics for specific type of systems and hosted applications. These projects can not be generalized or extended for maintaining a general class of systems even after reconfigurations. Therefore, in this dissertation, an autonomic performance management approach is developed in a very generic manner for maintaining the performance objectives of the deployed infrastructure and hosted applications by redefining and reconfiguring the developed performance management approach. This dissertation makes a number of novel contributions compared to the projects surveyed in this subsection:

- The developed performance management system can be customized according to the target application requirements and deployment architectures by using model-integrated computing practices.
- The developed performance management system can be extended by adding new measurement sensors, actuators, estimators, performance parameters, and optimization algorithms in the component library.

- The developed distributed control approach can be applied to a general class of applications hosted in distributed environment for managing QoS objectives.
- The developed performance management system provides scalability for managing QoS of applications hosted in distributed computing environment.
- The developed performance management system provides fault isolation among its components and reduces computational resource overhead through avionics operating system specifications.

2.2 Introduction to Web Services

According to World Wide Web Consortium (W3C) [39], a web service is a software system, which is represented by a uniform resource identifier (URI). Its public interface, interactions and bindings are well-defined to support interaction with other computing systems over the network. A web service interacts with other software systems in a well-defined manner as prescribed in its definition through Web Services Description Language (WSDL). WSDL version 2.0 [40] defines a standard language (XML based) for describing the functionality, concrete details, and compliance of the web service. WSDL separates the description of the web service from its implementation and location details.

2.2.1 Web Service Architecture

A number of computing systems (or applications) are using web services to interact with other computing systems (or applications) due to the standard interfaces and support of interoperability among computing system platforms. A web service is described with standard XML notation, which contains all the required details for interacting with the service, messaging format, transport protocols, and address. Only interface details are expressed while implementation details remain hidden, which makes web service indepen-

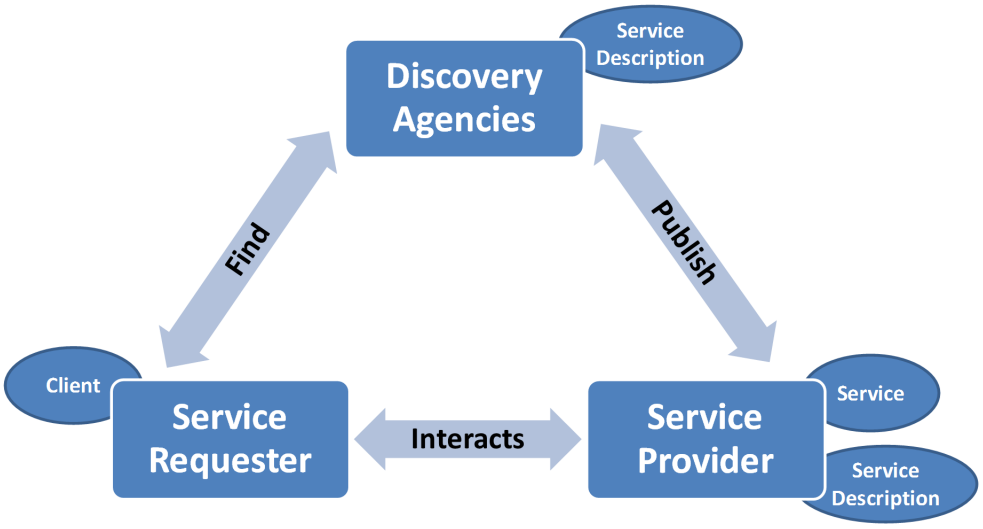


Figure 2.2

Service Oriented Architecture of Web Services.

dent of the underlying platform and programming language. Multiple web services can be combined together in a logical manner to create a complex application or business service.

A typical web service architecture is shown in Figure 2.2. In Figure 2.2, the nodes of the triangle represent *roles* and the edges represent *operations* performed by those roles. This web service architecture describes interaction among three roles: Service Provider, Service Requester, and Discovery Agent. According to Figure 2.2, the *Service Provider* hosts a internet (network) accessible web service and *publishes* its description to the *Service Requester* and *Discovery Agencies*. The service description consists of the structure definition, data types, message exchange patterns, and location of the *Service Provider*. The *Service Requester* performs the *find* operation to get the service definition locally or from the *Discovery Agencies*. The *Service Requester* uses this service definition to *bind* to the *Service Provider* and initiate *interaction* with the web service (software module). The

Service Requester and *Service Provider* communicates with each other through a sequence of one or more messages.

2.2.2 Service-Level Agreement for Web Services

In this highly competitive world, a Service-Level Agreement (SLA) is the primary feature to compare the QoS provided from different service providers to the customers they pay for. An SLA between a service provider and its customer can be defined as "expected performance behavior of the deployed web service with respect to measurable performance matrices under the pre-specified operating conditions." An SLA provides common mutual understanding and expectation of web service performance to service providers and their customers. In general, these SLA performance metrics include average response time, throughput, and availability of the service. Failure in meeting these SLA specifications may result in heavy penalties for the service provider. Moreover, the same service can be provided to different customers with different SLAs due to different pricing strategies.

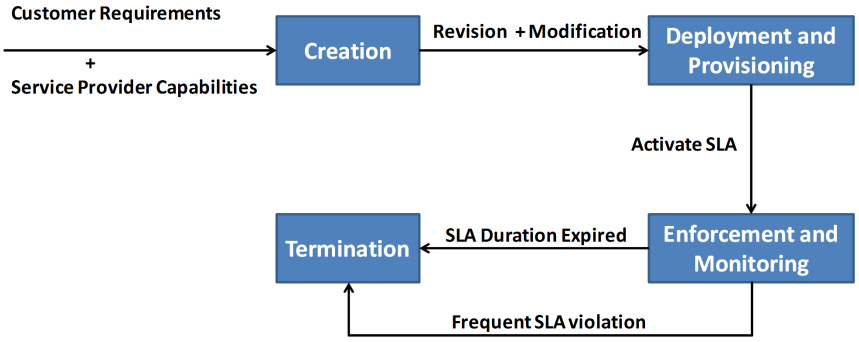


Figure 2.3

Service Level Agreement Life Cycle.

An SLA is important factor in negotiating a contract between service provider and customer during the deployment of the web service. It provides service-level assurance to the customers, a benchmark to provision system resources for a deployed web service, and a revenue model to the service providers. Each SLA passes through various stages of the SLA life cycle [57] as shown in Figure 2.3 [34].

Similar to a programming language, Service Level Agreement Language (SLAL) [81] provides a standard format to express the SLAs in terms of requirements, objectives, functions, penalties, and restrictions. SLAL is based on XML schema and is interpreted by both the service providers and the customers to deploy, monitor, and take appropriate actions on the web service.

2.2.3 Web Service Used in Dissertation

In this dissertation, a multi-tier web service, "Daytrader" [9] is utilized as a representative application. The Daytrader application is an online stock trading system that allows its users to login through a client utility or web browser to monitor their portfolios and buy or sell stocks. This application was developed at IBM research and has been utilized as performance benchmark application in the research community. Daytrader is built using J2EE technology, Java database connectivity (JDBC) for database access, and Java Beans for business logic. In this dissertation, daytrader is hosted on IBM's J2EE middleware, Web Sphere Application Server Community Edition (WASCE) [41]. More details on Daytrader will be provided in following chapters.

2.3 Power Consumption in Computing Systems

Most modern electronic components are built using the Complementary Metal Oxide Semiconductor (CMOS) technology. Advances made in the last decade have led to the increased clock rates and narrower feature length of the CMOS transistor. This in turn has allowed chip developers to stack more transistors on the die, which increases the available computational power. However, these advancements have come at the cost of increased power consumption.

At the level of a transistor, power consumption can be attributed to three factors: switching (dynamic) power consumption, leakage (static) current power consumption, and short circuit power consumption. These factors are applicable to all electronic systems of the computer, including the CPU, memory, and even the hard drive [1].

The working principle of a CMOS Field Effect Transistor (MOSFET) is based upon modulation of the electric charge stored by the capacitance between the gate and the body of the transistor [121]. This capacitor actually charges and discharges during one cycle (i.e., turning the switch first on and then off). Effectively, this causes an extra consumption of power, which is used for charging the capacitor. This power loss is also called switching or dynamic loss.

Leakage (static) current power consumption is due to the leakage current flowing through the transistor while being in the OFF state. Previously, static power consumption was negligible due to the low number of transistors per inch and high resistance of wires used on the chip. Currently, power loss due to the leakage current is about 40% of the total power

[1] In the hard drive, there are also some other mechanical factors that lead to increased power consumption, which are out of scope of this dissertation

consumption [125]. Lowering the voltage across the chip increases the leakage current by making transistors too leaky, which in turn increases the power consumption of the microprocessor [121]. Additionally, the high operating temperature of microprocessor increases the leakage current power consumption significantly.

A small amount of power consumption is present in CMOS due to the short circuit current on the short circuit path between the supply rails and the ground. This power consumption is considered as short circuit power consumption.

Dynamic power loss has been the main component of the total power loss for a long time in the past. Lately, the percentage of static power loss is increasing as feature sizes have been decreasing.

2.3.1 Power Consumption Modeling

A non-intrusive but accurate real time power consumption modeling effort is described in [76] to generate a power model with help of AC power measurements and user level utilization metrics. A microprocessor level power consumption estimation technique first examines the hardware performance counters and then uses relevant counters to estimate the power consumption through sampling based approaches [88]. Another approach for generating hard disk power consumption model is developed by extracting the performance information from the hard disk itself and by predicting the power model [157]. Additionally, the authors show that the modeling of idle periods is an important step in predicting the power consumption model of a hard disk.

An approach for power consumption modeling in an embedded multimedia application is introduced, where power consumption behavior is predicted by taking various image, speech, and video coding algorithms into account with supplied frequency and voltage [85]. A highly scalable power modeling approach is described in [91] for high performance computing systems by linearly extrapolating the power consumed in a single node to a complete large scale system by using various electrical equipment. A micro-architecture level temperature, voltage aware performance, and leakage current power modeling approach is introduced in [108]. This approach demonstrates variation of leakage current and energy consumption with varying temperature. Another approach for estimating power consumption in embedded systems is presented in [117], where power consumption is estimated during execution of a software application by considering the pipeline stall, inter-instructions effect, and cache misses. A power consumption model for smart phones is developed by utilizing measurements from built-in voltage sensors inside the battery and its discharge behavior [104]. Another approach of CMOS power short circuit dissipation is presented in [62], when short circuit power dissipation represents the significant amount of power consumption in a specific settings.

A computing system contains a large number of power consuming devices as discussed in the previous subsections. It is extremely difficult task to collect accurate measurements from all of these devices and to estimate their non-linear impact on the overall power consumption of the system for developing a performance management system. Therefore, a non intrusive power consumption modeling technique is used in this dissertation to estimate power consumption inside a computing system by using the system resource consumption

only. In this technique, an electronic Watt meter [38] is attached between the computing system power supply and electrical power socket to measure the power drawn by the computing system during its operation. Next, the computing system is put under a test by executing a set of experiments, which results in increased or decreased utilization of various system resources. According to the nature (computation, memory, or disk access) of experiments, corresponding system resource utilization measurement is recorded. Finally, a look-up table with near neighbor interpolation map is created with relevant resource utilization as the "key" and power consumption as the "value." This map is then validated by using a different set of tests with different utilization pattern of the same resource.

The primary benefit of the power modeling technique used in this research is that it is very simple but estimates the power consumption with high accuracy. The experiments discussed in next few chapters of this dissertation utilize this technique to model the power consumption of a multi-core physical server based on CPU core frequency and CPU utilization values. More details and observations of power modeling effort of this dissertation is described in Section 3.4.1.

2.3.2 Power Management Techniques

2.3.2.1 CPU Power Management

The main focus of academia and industry has been targeted at the power consumption of microprocessors. Three methods have been proposed till date to control the power consumption in microprocessors through application and system level techniques: dynamic

power switching (DPS), standby leakage management (SLM), and dynamic voltage and frequency scaling (DVFS).

The dynamic power switching (DPS) approach tries to maximize the system idle time that in turn forces a processor to make transition to idle or low power mode for reducing the power consumption [121]. The only concern is to keep track of the wakeup latency for the processor. The processor tries to finish the assigned tasks as quickly as possible so that the rest of the time can be considered as idle time of the processor. It reduces leakage current power consumption while increasing the dynamic power consumption due to the excessive mode switching of the processor.

The standby leakage management (SLM) technique is close to the strategy used in DPS by keeping the system in low power mode [121]. However, this strategy comes in to the effect when there is no application running in the system and when the processor needs to take care of its responsiveness only toward user related wake up events (e.g., GUI interaction, key press, or mouse clicks).

In contrast to the DPS, in the dynamic voltage and frequency scaling (DVFS) method, the voltage across the chip and the clock frequency of the transistor are varied (increased or decreased), to lower the power consumption and maintain the processing speed at the same time [121]. This method is helpful in preventing the processor from overheating, which can result in a system crash. However, the applied voltage should be kept at the level suggested by the manufacturer to keep the system stable for safe operation. DVFS reduces processor idle time by lowering the voltage or frequency, while continuing to pro-

cess the assigned task in a stipulated time with minimum possible power consumption. This approach reduces the dynamic power loss in the processors.

2.3.2.2 RAM Power Management

In general, the CPU is considered as the dominant component for power consumption in a computing system. However, recent research [70, 115] revealed that RAM can also be a significant contributor to the system power consumption. Therefore, RAM should also be a target for managing the power consumption, especially in the case of small computers. Currently, memory chips with multiple power modes (e.g., active, standby, nap, and power down) are available in the market and can be used for designing an efficient memory power management technique. The primary idea behind multiple modes of memory operation is that a different amount of power is consumed inside a memory in different states. Memory can execute a transaction only in the active state but can store the data in all of the states. The only concern while utilizing multiple modes is to consider the latency in time and power consumption while switching the modes. Similar to CPU power management, there are primarily two approaches for memory power management by using static and dynamic methods. In static method, memory is kept at a low power mode for the duration of system operation. In dynamic method, memory is placed in a low power mode when its idle time is more than the threshold time. Another approach of memory power management is described in [163], where multiple hardware components are combined on a single chip to create smaller and power efficient components.

2.3.2.3 Other Methods for Power Consumption Management

According to a detailed analysis and modeling of power consumption, miscellaneous components are also responsible for a large fraction (30% - 40%) of power consumption inside a computing system [76]. These components include disk, I/O peripherals, network cards, and power supplies. The primary contributors in power consumption are disk and power supplies. Device vendors have started implementing their power management protocols to ensure that these devices can run in a low performance mode. For example, hard disks typically have a timer that measures the time of inactivity and spins the drive down to save power.

An application level power management scheme is used in [146] to solve the cost-aware application placement problem. This scheme also designs an algorithm to minimize the migration cost while maintaining the performance requirements. Another two step approach of power aware processor scheduling is presented in [158]. This two step approach, first performs load balancing among the multiple processors and then applies DVFS to control the speed of the processors to minimize the power consumption.

A proactive thermal management approach in data centers is described in [106] to prevent heat imbalance in cooling the data center while minimizing the cooling cost. This approach optimizes the fan speed and air compressor duty cycle to prevent heat imbalance. Additionally, it reduces the risk of damage to the data center due to excessive heating in the data center. An approach of saving power consumption in servers is introduced in [126], where NAND flash based disk caches are extended to replace PCRAM.

2.3.2.4 Power Management in Dissertation

In this dissertation, power consumption of the physical server is considered as the primary component of the operating cost in application deployment. In addition, as computational resource intensive application (modified "Daytrader") is used in the research, CPU frequency and CPU utilization are considered as the major contributing factors in total power consumption (refer to the experiments in next chapters). Therefore, CPU power management techniques are used to control the power consumption due to CPU core frequency and CPU utilization. During this research, DVFS is used as the tuning option to change the CPU core frequency for maintaining the desired response time of the application and minimizing power consumption at same time. DVFS reduces CPU core frequency to avoid overheating of CPU core and dynamic power loss. Other CPU power management techniques, such as DPS and SLM are not used in this research because these techniques force processors to use the highest CPU core frequency to finish the assigned task as soon as possible and then switch to low power mode. Due to the use of highest CPU core frequency, these techniques increases the dynamic power consumption, chances of system crash, and temperature of the CPU core. Moreover, this increased temperature of the CPU core activates cooling mechanism (cooling fans), which further increases the total power consumption of the system.

RAM power management approaches are not utilized in this dissertation because the deployed application (modified "Daytrader") does not show any significant variation in RAM utilization with change in the incoming workload. However, the power management

technique used in this dissertation can also be extended for RAM power management by utilizing the RAM power consumption models built through experiments and simulations.

2.4 Model Integrated Computing (MIC)

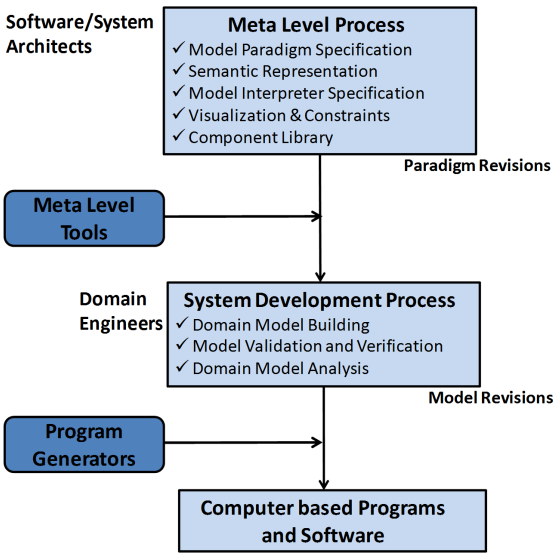


Figure 2.4

Model Integrated Computing Architecture.

Modern computing systems contain a large number of interconnected components, which increase the complexity of computing systems in functionality and interoperability with other systems. In addition to this, applications deployed on the system are tightly coupled with the physical system resources that add another level of complexity and possibility of unknown interactions between the application and the physical system. Therefore, it is recommended that the applications and the physical systems are studied together. This study can be further utilized by the researchers to study the impact of changes in one on

the other. *Model Integrated Computing* (MIC) [23] technology has been used successfully for over two decades to design complex software systems in the form of models. These models capture the complete requirement, architecture, operating environment, and interaction rules of the system in high level models per the researchers' understanding of the system. In general, these models are developed for an entire class of the problem domain instead of a specific implementation.

MIC presents a formal and compositional representation of the model, which is further utilized during the system development process. A typical system development work-flow using MIC is shown in Figure 2.4. MIC keeps the models in the heart of the system development life cycle similar to the model driven development approach shown in Figure 2.5. This model based system development life cycle contains various states that include model specification, verification, implementation, testing, production, and runtime monitoring. MIC contributes to the model driven development through three core elements: Domain Specific Modeling Language (DSML), MIC tools for DSML, and Code Generation using Model Interpreters.

2.4.1 Domain Specific Modeling Language (DSML)

MIC uses DSML to visually or textually represent the system under design. DSML represent semantics of the system using domain specific symbols in a declarative manner. A DSML can be defined by using concrete and abstract syntax, a semantic domain, and mapping between syntactic and semantic information [95]. This concrete syntax contains visual or textual notations (or symbols) to represent the system models while abstract syn-

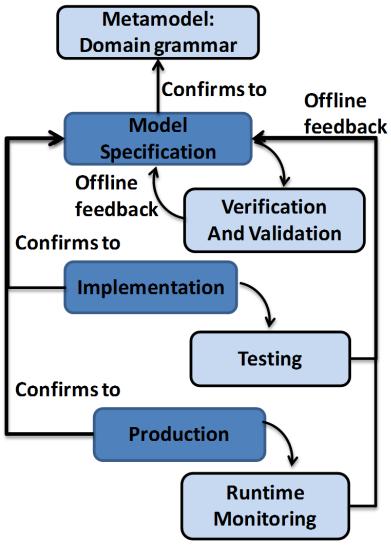


Figure 2.5

Model Driven Development.

tax represents the object relationship and integrity constraints of the DSML. The semantic domain mappings define the properties of the models created using the DSML. DSML is useful in creating a tailored modeling language for a specific problem domain.

2.4.2 MIC Tools for Domain Specific Modeling

MIC tools are utilized for generating, validating, and maintaining the domain specific system models. These tools are first used for creating DSML (or meta-models) for a class of application (problem) domain and then used as a domain specific modeling tool for a specific implementation of that domain. The meta-modeling process in MIC tools is based on UML [37]. MIC uses tools, which are developed at the Institute of Software Integrated System, Vanderbilt University. GME [14] is used for both meta-modeling and application modeling of a large number of engineering system in visual and text environment. A set

of meta-modeling and object constraints language (OCL) [29] concepts are built inside the GME to enable it to understand the UML-like meta-modeling language and generate target domain models. During the application domain modeling process, GME is used to create domain specific models, which represent the target domain and the deployment configurations. These domain specific models are stored in model databases. GME has been used successfully in many areas including real-time systems [73], fault diagnosis [52], fault mitigation [74], software health management [49, 114, 67], distributed real-time and embedded systems [135], QoS adaptive applications [156], shipboard power systems [136], and wireless sensor networks [147].

2.4.3 Code Generation using Model Interpreters

GME provides a set of generic APIs to access the created domain models and their various attributes. These APIs can be further used by the high level programming languages (C++) for developing software tools to access and manipulate the model objects and their attributes. For this purpose, two tool suites are developed at ISIS, Vanderbilt University, with different functionality: Universal Data Modeling (UDM) and Graph Rewriting and Transformation (GReAT).

UDM [36] is a model management tool suite that generates model specific meta-programmable APIs in a high level programming language (C++). These C++ APIs can be further used for developing software packages specific to that application domain. UDM can also be used to access the models developed by using other DSML instead of using GME exclusively [94].

GReAT [16] is a graphical model transformation tool suite that is used for transforming the models from one meta-model domain to another meta-model domain. GReAT utilizes graph transformation rules that convert the input meta-model to the target meta-model by starting from a small part of the meta-model and moving toward the entire meta-model. This tool suite is utilized frequently for developing model transformation tools and code generators [55] in model integrating computing research.

2.4.4 MIC in Dissertation

In this dissertation, MIC methodologies are utilized for developing the distributed control structure for performance management of a general class of applications hosted in a distributed environment. MIC techniques first create meta-models of various components of the distributed control structure, then create a application deployment specific application model of the control structure, and finally use code generation utilities for generating the deployment plan and deployment configurations of the control structure. More details on this will be presented in Chapter 7.

2.5 Component-based Performance Management Solutions

Distributed computing environments host a wide variety of applications. In general, application providers and customers have similar expectations from these applications, such as minimum response time, maximum throughput, high availability; however, each of these applications might have different performance metrics, configuration strategies, operating environment dependency, resource constraints, and management policies. Mostly, these applications are packaged with their own management solutions, which can manage these

stand-alone applications. However, it creates another problem of management complexity, when multiple applications of different nature are hosted in the same infrastructure under an administrator. In this case, the administrator has to monitor, and manage each application from their respective management solutions, which is extremely difficult, error prone, and inefficient. This monitoring, and managing of different applications can be made extremely easy and efficient if performed while considering the functionality of individual modules and components of these applications.

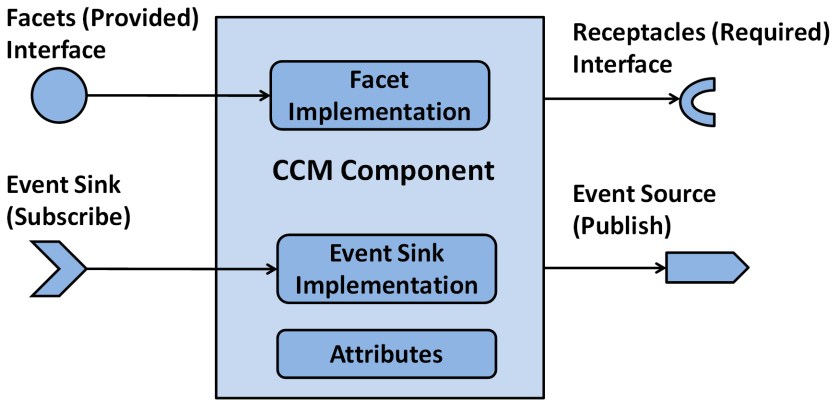


Figure 2.6

Corba Component Model (CCM) description.

Component-based systems provide flexibility to system developers to decompose a large complex system into smaller components that are created for a special functionality. Each of these components provides a well-defined special functionality to other components connected to it by means of a well-defined interface. The interface of these components is described as ports and attributes. Ports are communication end points that

interact with other components, while attributes are the data types that represent the state of the component. These small components can be reused, reconfigured, and extended in a flexible manner to combine as a large system. Typical examples of component models: Corba Component Model [150], Microsoft .NET model [21], Microsoft's Component Object Model (COM) [7], and Enterprise Java Beans Model [12]. These component models define the attributes of the component, its interaction with other components, its configuration rules, and its deployment strategy. CCM is a strong candidate compared to EJB, COM, and .NET component-based approaches for developing performance monitoring and management systems due to multiple benefits: CCM components can be developed in multiple high level programming languages (C++, Java), CCM has advanced support for QoS properties of components, and CCM components can be executed on multiple operating systems (Windows and Linux). In these management systems, a number of small monitoring and management components can be developed for each type of resource (CPU, RAM, etc.) and combined together to function as a complete management system in different application domains. Additionally, these small components can also be replaced (corrected or updated) in the future by only changing respective components (functions) of the management system while keeping the rest of the management system unchanged.

In this dissertation, the corba component model (CCM) is used to create various components that use CORBA as its middleware infrastructure. CCM specifications are defined by the Object Management Group (OMG) [27]. A typical component of CCM is shown in Figure 2.6. According to Figure 2.6, *Facet* interfaces are a set of services (functionalities) that a component provides to other components in a synchronous or asynchronous

manner. *Receptacle* interfaces are a set of services that this component needs from other components to complete a specific task. *Event* sink and *Event* source are the ports that this component uses to subscribe (listen) and publish (send) events respectively. *Attributes* represent the state of the component (configuration value of the component), and these can be set or queried by external components by using CORBA interfaces.

Recently, a number of research groups in the research industry and academia have utilized component-based management approaches to develop performance management frameworks for different application domains, such as web services, multimedia applications, network management, sensor network, real-time systems and workflow systems. A components-based integrated management framework (SCENE Admin) [107] was developed for web service platforms. In this framework, an administrator can select and configure the management system of the web service environment. Another component-based framework *PLASMA* [46] was developed for building self-adaptive multimedia applications. PLASMA uses hierarchical reconfiguration and dynamic architectural description language to create the management framework. A component-based orchestration management framework [59] was introduced for multi-domain service oriented architecture. This framework gives a unified and global abstract view of the workflows executing in a decentralized manner for effective monitoring and management.

A component-based framework [63] was designed for data mining and knowledge discovery in avionics systems by analyzing the measurements from different modules. This component-based framework results in decreased human intervention and processing for analyzing the measurements. A component-based framework [73] was developed for giv-

ing flexibility to developers in modeling real-time systems in a component-based fashion. Another component-based network management system (RMTool) [65] was designed for providing flexibility to network administrator in dynamically configuring the network resources of wireless sensor networks. This tool provides the capability of both efficient monitoring and effective management to the network administrator. A component based programming framework (ACCORD) [109] was presented for developing self-managed autonomic applications that can be executed in the distributed environment as composition of various small autonomic components.

The component-based approach is utilized in this dissertation for the development of a generic performance management system that can be applied to a class of applications hosted in a distributed environment. Existing component-based approaches, discussed above, are applicable to only specific type of applications. The component-based management system is developed in this dissertation in following steps: developing the meta-models of the standard components for each functionality (monitoring or control) of the typical management system, developing these components in high level programming language as per meta-model specifications, configuring the components as per the application requirements, and deploying the components in distributed environment according to the deployment plan of the application. This approach facilitates the re-usability of the developed monitoring sensors, control modules, control algorithms, and application performance models that increase the productivity of the researchers while decreasing the application specific development.

2.6 Summary

This chapter described the preliminary concepts used during the course of this dissertation. It also presented the previous research efforts by other research groups in the following areas: autonomic computing systems, web service development, power consumption modeling and management, model integrated computing, and component-based performance management systems.

CHAPTER 3

A PERFORMANCE MODELING APPROACH FOR A MULTI-TIER WEB SERVICE SYSTEM

This chapter presents research efforts to model a multi-tier web service system that is hosted in a virtualized environment. In this chapter, web service system configuration, web service performance models, environment workload models, and physical system power consumption models are presented. Additionally, previous research on modeling a web service, preliminary ideas of queueing models, and Kalman filters are also discussed.

3.1 Related Work

In the web service environments, incoming web service requests have a cyclic nature and correlated properties in computation and data access complexities. Therefore, model-based approaches are a well-suited option to predict the computational requirement of the future incident requests and to devise an efficient resource provisioning after observing the previous requests. The potential of a model-based resource provisioning method in a web services environment is already demonstrated in [72] for developing a simplified analytical models of server memory, storage I/O rate, storage response time, and service response time. This analytical model was utilized to capture the application performance in an informed policy-driven resource allocation for complex resource management challenges.

Furthermore, an approximate layered queuing model of a multi-tier web application system has been used by researchers at IBM Labs in [69] and [145] to capture the service dependencies, different performance characteristics, per-tier concurrency limits, and resource contention. This approximate layered queuing model was developed with help of the function approximation methods while serving an incoming request among multiple tiers of the hosted application.

The key characteristics of a web traffic access pattern and corresponding web server performance have been investigated in [138] with help of *auto regressive moving average* (ARMA) filter and $G/G/1$ queuing models, respectively. The web service performance is studied to analyze the waiting time behavior of the web server and the latency observed by the end user. Additionally, a cluster-based web services management technique is described in [119] to support a mixed http workload related to multiple services. This management approach dynamically allocates server resources for maximizing the pre-specified cluster level utility function with the help of a first principle based (response time) performance model of the cluster in the case of extremely dynamic workloads. Furthermore, machine learning and Bayesian estimation techniques (a Kalman Filter) have been used in [152] and [90], respectively, to derive the best allocation policy for CPU and memory resources in a multi-tier deployment with dynamic incoming workloads.

This dissertation presents a systematic performance modeling approach for a distributed multi-tier web service system. The proposed approach starts by experimentally identifying the system parameters impacting the performance of the web service, by defining the dependency relationship among these parameters, and then by using that relationship to

develop the model structure of the system. This performance modeling approach uses a mixture of a regression technique (offline) for estimating the power usage model and Bayesian techniques (online exponential Kalman Filter) for estimating the state of the web service system modeled as an equivalent processor sharing queue system.

3.2 Preliminaries

The developed performance modeling approach uses a multi-layered queuing model to capture the performance behavior of the web service in a multi-tier environment. An exponential Kalman filter is developed in this research that estimates the state of the system as an equivalent processor sharing queuing system. This section presents the key concepts of the multi-layered queuing system models and of the Kalman filters.

3.2.1 Queuing Models for Multi-Tier Systems

In general, a web request has to wait in a queue before it can enter inside a tier (e.g., application or database) and acquire computational resources. For example, if the number of maximum threads allowed in the application tier is capped to limit concurrency of the tier, a new request will wait until an already executing request releases a thread. Clearly, the total service time of the enterprise system is directly affected by the queuing policy at each tier. Therefore, an approximate queuing model can be used to capture the behavior of such systems. This queueing model can be used to measure the average number of web requests in the queue and the average time spent in the queue. Generalized structure of single-server and multi-server queueing systems with the relevant parameters are shown in Figure 3.1 [139]. Queueing models are represented in Kendall's notation [97]

as $A/B/C$. In Kendall's notation, A represents the statistical distribution of inter-arrival time in incoming requests λ. B represents the statistical distribution of service time S of the incoming requests. C represents the numbers of servers N processing the incoming requests. During this research, four different queuing models [102] are considered: $M/M/1$, $M/G/1\ FCFS$, $M/G/1\ PS$, and $M/G/1\ LPS(k)$.

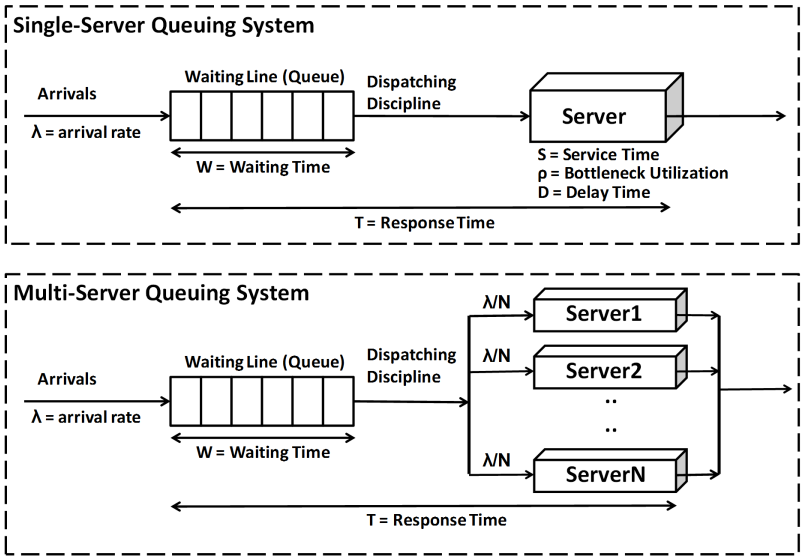


Figure 3.1

Queueing System Structure and Parameters.

$M/M/1$ is the most basic queuing model, where both inter-arrival time and service time, of the incoming requests, are exponentially distributed (M). In this case, single server queue (1) is considered.

The $M/G/1$ queue assumes that the inter-arrival time is exponentially distributed (M), but the service time has a general distribution (G). $M/G/1\ FCFS$ is a more realistic

model of web service behavior since web requests exhibit a wide variance in their service requirements. The scheduling discipline is *first come first served* ($FCFS$). Therefore, this model assumes that only one request is serviced at a time, hence restricting the concurrency of the tier to one.

$M/G/1$ PS is an $M/G/1$ queue with *processor sharing* (PS) scheduling. In this queueing model, a newly arrived request shares the computational resource, concurrently, with all of the preceding and existing requests. The computational resource is shared equally by all of the requests in a round-robin manner, though with an extremely short time slice. In other words, the concurrency level is basically unlimited. When a request is not using the resource, the request waits in the queue for its turn to get a slice of the resource. In this queueing model, the mean analysis of the system is rather simple. In fact, the closed form expression for the mean response time is similar to that of the $M/M/1$ queue. This queueing model can only be used to study a web server realistically if the total number of requests in the system do not increase above the maximum concurrency limit. *Thus, if the bottleneck resource utilization is light to moderate (less than* 1*), this queuing model can be used to model the web servers*.

$M/G/1$ $LPS(k)$ is an $M/G/1$ queueing model with *Limited Processor Sharing* (LPS) scheduling discipline. The parameter k models the concurrency level of the tier. In such a model, the first k requests in the queue share the computational resource and the rest of the requests wait in the queue. As requests complete their service and depart from the system, awaiting requests are admitted to be among the k concurrently sharing the resource in a processor sharing manner. When $k = 1$ the queue becomes an $FCFS$ queue, and when

$k = \infty$ it becomes a PS queue. The $LPS(k)$ is a realistic queuing model for systems, which have a limit on the maximum number of jobs that can execute concurrently. This limit is typically enforced on all web servers and databases. Even in an operating system, the maximum number of processes that can execute concurrently is limited (e.g., 32000 for Linux kernel 2.6). The analysis of the LPS queue is difficult, which makes online prediction for response time and other variables intractable [160, 159].

3.2.2 Kalman Filters

Kalman filter [89] (KF) is an *optimal recursive data processing algorithm*, which estimates the future states of a linear stochastic process in the presence of measurement noise. This filter is optimal in the sense that it minimizes the mean of squared error between the predicted and actual value of the states. It is typically used in a predict-and-update loop where information of the system dynamics, the current system state, the statistical characteristic of the system noise, and the measuring device dynamics are used to estimate the next system state. The available measurements and statistical description of the measurement noise are also used to update the state estimate. In general, the following two assumptions are made before applying the standard Kalman filter techniques to a process for estimation:

1. The system in consideration is described by a linear model. If the system is nonlinear, the system model is linearized at the current state and an extended Kalman Filter (EKF) can be used in this case.
2. The measurement and system noises are Gaussian and white, respectively. Here, *Whiteness* indicates that noise is not correlated with time and it has equal impact on all operating modes of the system.

Due to the simple approach with optimal results, the KF has been applied in wide areas of engineering applications, including motion tracking, radar vision, and navigation systems. Woodside, et al. [153] applied EKF techniques for the parameter estimation of a simple closed queueing network model. The population size was approximated by a continuous variable in order to fit the KEF mathematical framework. An EKF is used to track parameters, such as the think time and the processing time of a time-varying layered queueing system in [162]. The EKF uses average response time with three different resource utilization: a web processor, a database processor, and a disk. An EKF coupled with layered queueing models is also applied in [154] to control the number of allocated servers so as to maintain the average response time within a given range. Another implementation of an EKF is presented in [161], where an estimation methodology is sought for estimating the performance model parameters in various open or closed queueing network models. These parameters have lower and upper bounds on values. KF has also been applied to CPU resource (physical cpu share) provisioning in a virtualized environment, where virtual machines host application instances [90]. The feedback controllers based on KF continuously monitors the CPU utilization and accordingly updates the CPU allocation for estimated future workloads [90]. In this case, an average of 3% performance improvement was observed in extremely dynamic workload conditions over a three-tier Rubis benchmark web site deployed on a virtual Xen cluster.

In this dissertation, an *exponential Kalman filter* (ExpoKF) is developed to predict the computational nature of the incident http requests over a web server. This filter is exponential because it operates on the exponential transformation of the system state variables.

This transformation allows us to enforce the ≥ 0 constraint on the state variables. These constraints are not possible in typical Kalman filter implementations. ExpoKF predicts the *service time* S and *delay* D of a web request through observing the current *average response time* T of the incident request and *request arrival rate* ω on the web server. This filter uses an $M/G/1\ PS$ approximate queuing model as the system state equation and considers variation in S and D at the previous approximation to estimate the S and D at the next sample time. Further details are provided in Section 3.4.3.

3.3 Web Service System Setup

Multi-tier enterprise systems are composed of various tiers (server) that typically include web (http) tier, application tier, and database tier. Each of the tiers performs its function with respect to web requests and forwards the result to the next tier. In order to experiment and validate the developed web service performance model, a web service system setup is used in this dissertation. The representative setup consists of application and database tiers while Http tier is combined with the application tier. Details of the used setup are described in Table 3.1.

3.3.1 System Setup

Web service system configuration is summarized in Table 3.1. It also shows the virtual machines (VM) hosted on each of the physical machines and the roles played by those VMs. All VMs use the same version of Linux ($2.6.18 - 92.el5xen$). Client machines are used to generate http requests. Application servers host the open source version of IBM's J2EE middleware, *Web Sphere Application Server Community Edition* (WASCE) [41].

Table 3.1

Web Service System Setup Configuration

Name	Cores	Description	RAM	DVFS	VMs
Nop01	8	2 Quad core 1.9GHz AMD Opteron 2347 HE	8GB	No	Nop04, Nop07 (Monitoring server)
Nop02	4	2.0 GHz Intel Xeon E5405 processor	4GB	No	Nop05, Nop08 (Client Machines)
Nop03	8	2 Quad core 1.9GHz AMD Opteron 2350	8GB	Yes	Nop06, Nop09 (Application server)
Nop10	8	2 Quad core 1.9GHz AMD Opteron 2350	8GB	Yes	Nop11, Nop12 (Database Server)

Database machines host MySQL open source database. *Nop03* and *Nop10* both have DVFS capability that allows administrators to tune the entire physical node or its individual cores for a desired performance level. Xen Hypervisor [43] was used to create, configure, and manage physical resources (CPU and RAM) for the cluster of Virtual Machines (VMs) on these physical servers.

3.3.2 Web Service Application

Daytrader [9] is used as a representative application during this research. Business enterprise loads in extremely dynamic environment is emulated by modifying the the main trade scenario servlet in Daytrader to shift the processing load of a request from the database tier to the application tier. The Httperf [47] benchmarking client utility is used in all experiments to generate the http workload at a pre-specified rate. Httperf provides flexibility to generate various workload patterns (poisson, deterministic, and uniform) with numerous command line options for benchmarking. Httperf is modified to log the per-

formance measurements periodically while executing the experiment. At the end of each sample period, the modified version of *Httperf* logs the detailed performance statistics of the experiment in terms of *total numbers of requests sent, minimum response time, maximum response time, average response time, total number of errors with types*, and *response time* for each request.

3.3.3 Monitoring Setup

Specially developed python scripts and Xenmon [80] were used as monitoring sensors on all virtual and physical machines. These sensors monitor the CPU, the disk, and the RAM utilization throughout system execution. These measurements are reported after each sampling interval and at the end of the experiment. System time across all machines was synchronized using NTP. Web server (Daytrader) source code was modified to monitor the web server performance in *max threads active* in the web server, *response time measured* at application tier and at database tier for each incident request, and *average queue size* in the web server after each sampling interval. Measurement of power consumption in a physical node was performed with help of a real time watt meter [38].

3.4 System Modeling Approach

An accurate system model is necessary to run a computing system efficiently in an SLA complied environment. The developed model depicts the exact system behavior in terms of various performance objectives with changes in the operating environment and the controllable parameters. A number of experiments were performed during this dissertation to learn the performance behavior of the representative system. During these experiments,

the multi-tier system performance was analyzed with respect to the system utilization, various environment workload profiles, bottleneck resource utilization, and its impact on the system performance. Table 3.2 shows the list of parameters that have been identified after these experiments. This list contains three different types of parameters: *Control Inputs, State Variables,* and *Performance Variables*. Control inputs can be used at runtime during an experiment for tuning the system in order to achieve performance objectives. State variables describe the current state of system under observation. Performance variables are used to asses QoS objectives. Additionally, state variables are divided into two different categories: *Observable* and *Unobservable*. Observable variables can be measured directly through various sensors, system calls, or application related APIs, while unobservable variables cannot be measured directly; instead, they are estimated within a certain level of accuracy by using existing measurements through various estimation techniques at runtime (e.g., KF explained in section 3.2.2). Details of the modeling efforts are described in the following sub sections.

3.4.1 Power consumption

As a first step toward system model identification, the mutual relationship among physical CPU core utilization, CPU frequency, and power consumption of the physical server was identified. Figure 3.2 shows the power consumed on one of the physical server *Nop03* with respect to the aggregate CPU core usage and the CPU frequency. An extensive experiment was performed on the physical server *Nop03* with the help of a specially written script, which exhausted a physical CPU core through floating point operations in incre-

Table 3.2

System Parameters.

Control Inputs	**State Variables**	**Performance Variables**
CPU Frequency	CPU Utilization (Observable)	Average Response Time
Cap on Virtual machine resources.	Memory Utilization (Observable)	Power consumption in Watts
Load distribution percentage (in a cluster)	Service Time (Unobservable)	Percentage of Errors
Number of Service Threads	Queue waiting Time (Unobservable)	
Number of Virtual Machines in Cluster	Queue Size on each server (Unobservable)	
	Number of Live Threads (Observable)	
	Peak Threads available in a JAVA VM (Observable)	

ments of 10% utilization that was independent of the current CPU frequency. With multiple instances of this utility, all eight physical CPU cores of the *Nop03* server were loaded in an incremental manner for different discrete values of CPU frequencies. The CPU frequency across all of the physical cores (1 to 8) was kept the same during each step. The power consumption was measured with the help of a real time watt meter. During this experiment, the utility scripts perform floating point computations, which were neither memory intensive nor I/O intensive. Therefore, only CPU utilization and CPU core frequency were considered as the factors of power consumption. Based on this experiment, a regression model was created for power consumption at physical machine with respect to CPU core frequency and aggregate CPU utilization. After analyzing the results (and reconfirmation with several other experiments across other nodes), it was observed that the power con-

sumption model of a physical machine is non-linear because power consumption in these machines depends not only upon the CPU core frequency and utilization, but also depends non-linearly on other power consuming devices (e.g., memory, hard drive, CPU cooling fan, etc).

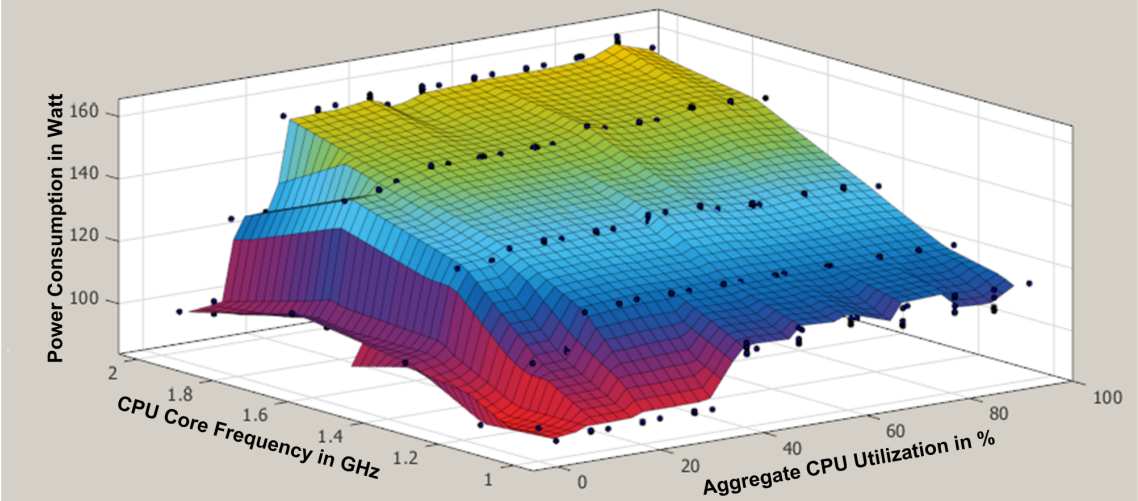


Figure 3.2

Power Consumption on Nop03 vs (CPU frequency and Aggregate CPU core Utilization).

As a result, a *look-up table with near neighbor interpolation* was found to be the best fit for aggregating the power consumption model of the physical machine. The combination of the CPU core frequency and aggregate CPU core utilization of the physical machine is used as the key for the lookup table to access the corresponding power consumption value. This aggregate power model was utilized further for the controlled experiments described in the next chapter for predicting the estimated power consumption by the physical server at a specific setting of the CPU core frequency and the aggregate physical CPU utilization.

3.4.2 Request characteristics

The Httperf benchmark application code was modified to allow the generation of client requests to the web server ($Nop06$) at a pre-specified rate as provided from a trace file. At $Nop06$, each request first performed certain fixed floating point computations on the application tier ($Nop06$) and then performed a random select query from the database tier ($Nop11$). client requests are characterized by using linear regression [118] as the number of CPU cycles needed to process the request at the web server.

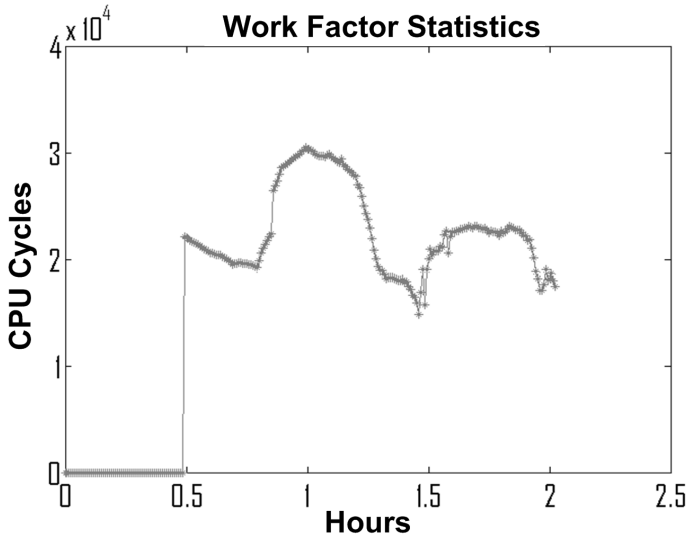


Figure 3.3

Work Factor Plot for Request Characteristic.

During any sampling interval T, if ρ is the virtual CPU utilization, f is the CPU frequency, c is the work factor of the request (defined in terms of CPU clock cycles), λ is the request rate, and ψ is system noise,

$$\rho f = \lambda c + \psi \tag{3.1}$$

The average work factor was computed to be $2.5X10^4$ CPU cycles with a coefficient of variation equal to 0.5. The variation in W shows the variation in the nature of final request based upon the chosen symbol for the database query. The result of the experiment is shown in Figure 3.3.

Due to the similar computational nature of all the requests incident on the web server in a given sample time, the total computation time for all the requests can be approximated, which in turn gives the average response time of the requests in a given sample time. We use this average response time information to check the status of the QoS objective (response time) in the web server.

3.4.3 Webserver characteristics

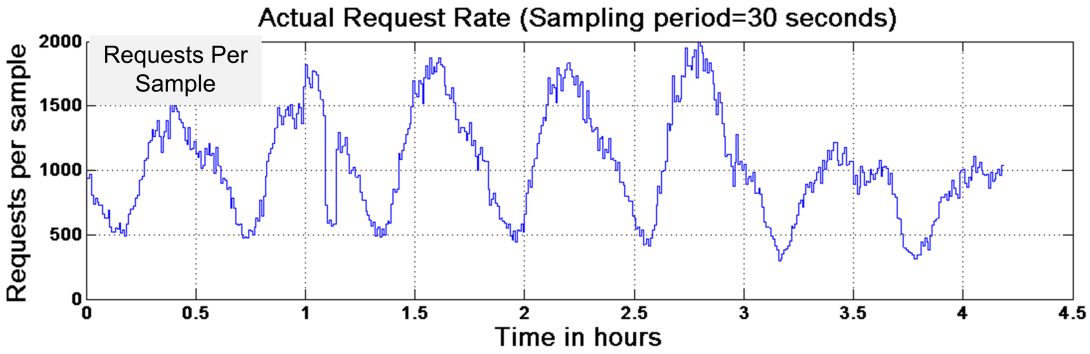


Figure 3.4

Http Workload based upon World Cup Soccer(WCS-98) Applied to the Web Server.

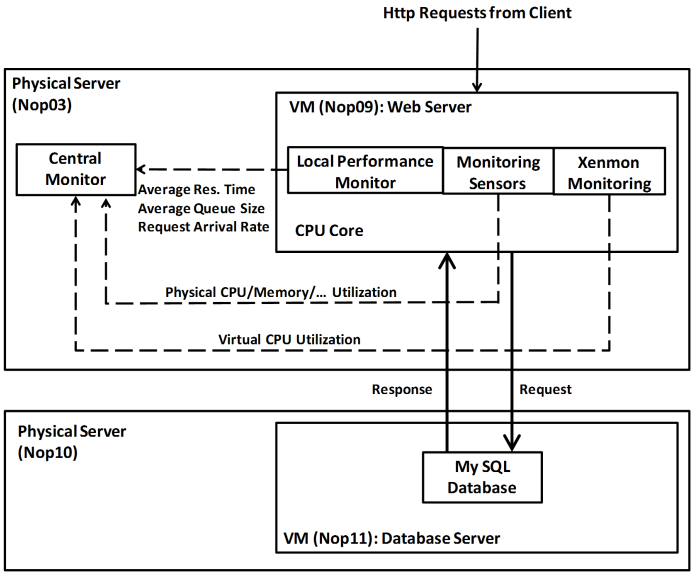


Figure 3.5

Web Server Behavior Modeling Experiment Setup from Section 3.4.3.

Experiment were performed to model the uncontrolled web server performance and to identify the bottleneck resource of the web server. A system resource is considered as the *bottleneck* if performance of the web server is affected due to limited availability of this resource. For example, CPU time slices, free Java threads, and RAM are necessary to process an incoming web request. In this experiment, *Nop09* was the virtual machine (physical machine *Nop03*) hosting the application tier of the *Daytrader* application (see Figure 3.5). The virtual CPU of *Nop09* was pinned to a single physical core and 50% of the physical core was assigned to *Nop09* as the maximum available computational resource. Physical memory was also limited to 1000MB for *Nop09*. *Nop11* was configured as a database tier by using similar CPU and memory related operating settings over the physical server *Nop10*. To simulate a real time load scenario, all CPU cores of the physical server *Nop03*

(except the CPU core hosting *Nop09*) were loaded approximately 50% with help of the utility scripts described in Section 3.4.1. The maximum concurrency limit of the web server was limited to 600 by configuring the `MAX_JAVA_threads` parameter in the IBM Websphere application server. All CPU cores in the physical server *Nop03* were operating at their maximum frequency of 2.0 Ghz. The incoming http client request trace (see Figure 3.4) of this experiment was based on user request traces from the 1998 World Cup Soccer(WCS-98) web site [56].

The response time and power consumption as measured from this experiment are shown in Figure 3.7 and Figure 3.6. These figures also show the CPU utilization at the web server and the aggregate CPU utilization of the physical machine. Notably, it was observed that the CPU utilization at the web server ($Nop09CPU$) and the aggregate CPU utilization of the physical machine ($Nop03CPU$) follow a trend that is similar to the rate of incoming http requests made to the application tier. The power consumption curve was almost flat. The web server response time is also correlated with the resident requests (system queue size) in the web server system.

To determine the bottleneck resource, a queuing approximation for a two tier system was used as shown in Figure 3.8. λ is the incoming throughput of requests to an application while ρ is the utilization of the bottleneck resource. S is the average service time on the bottleneck resource, D is the average delay, and T is the average response time of a request. The average waiting time for a request is $W = T - S - D$. We define a queuing model with the state vector $[S\ D]$ and the observation vector as $[T]$.

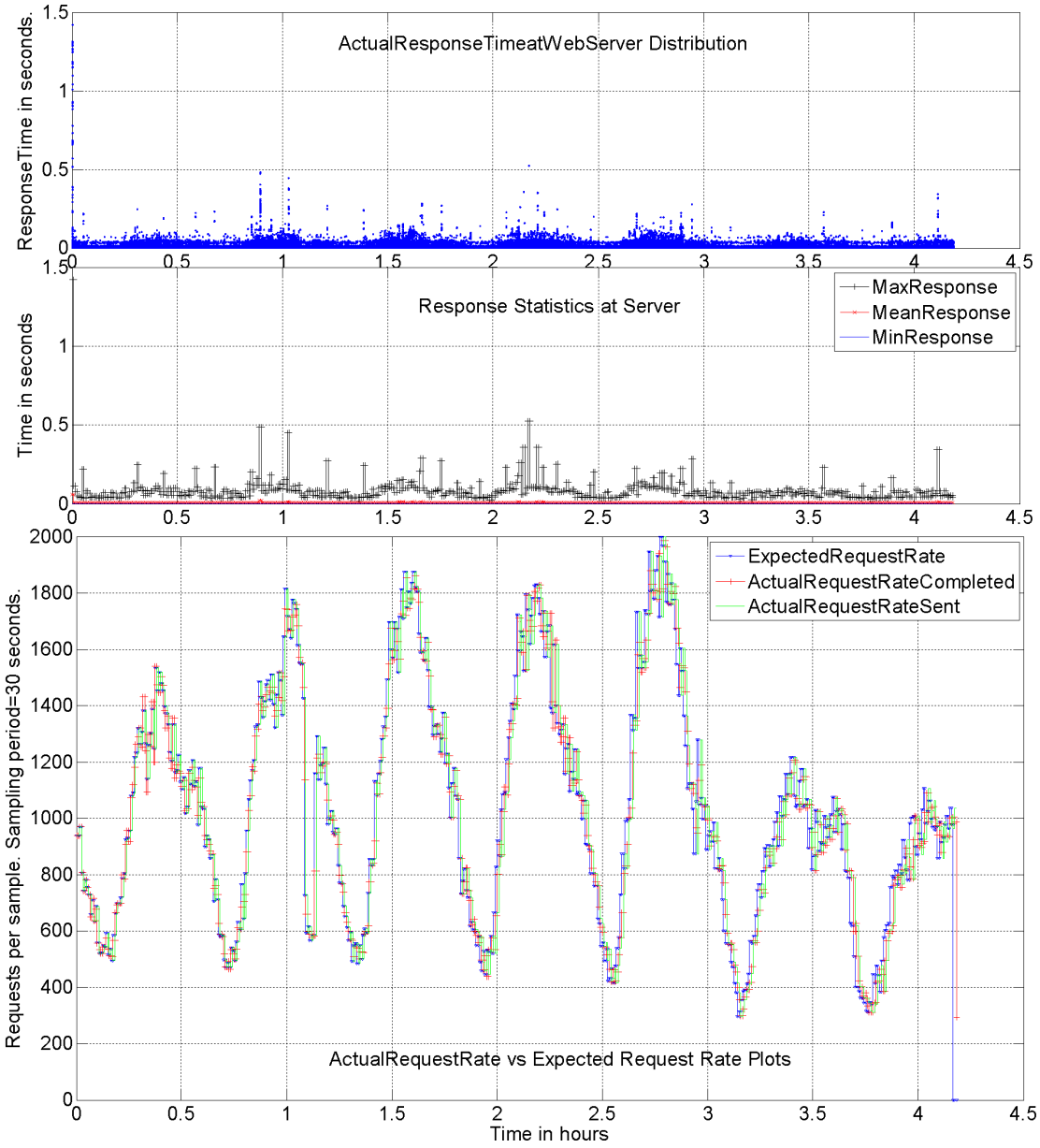


Figure 3.6

Web Server Behavior Modeling Experiment Results.

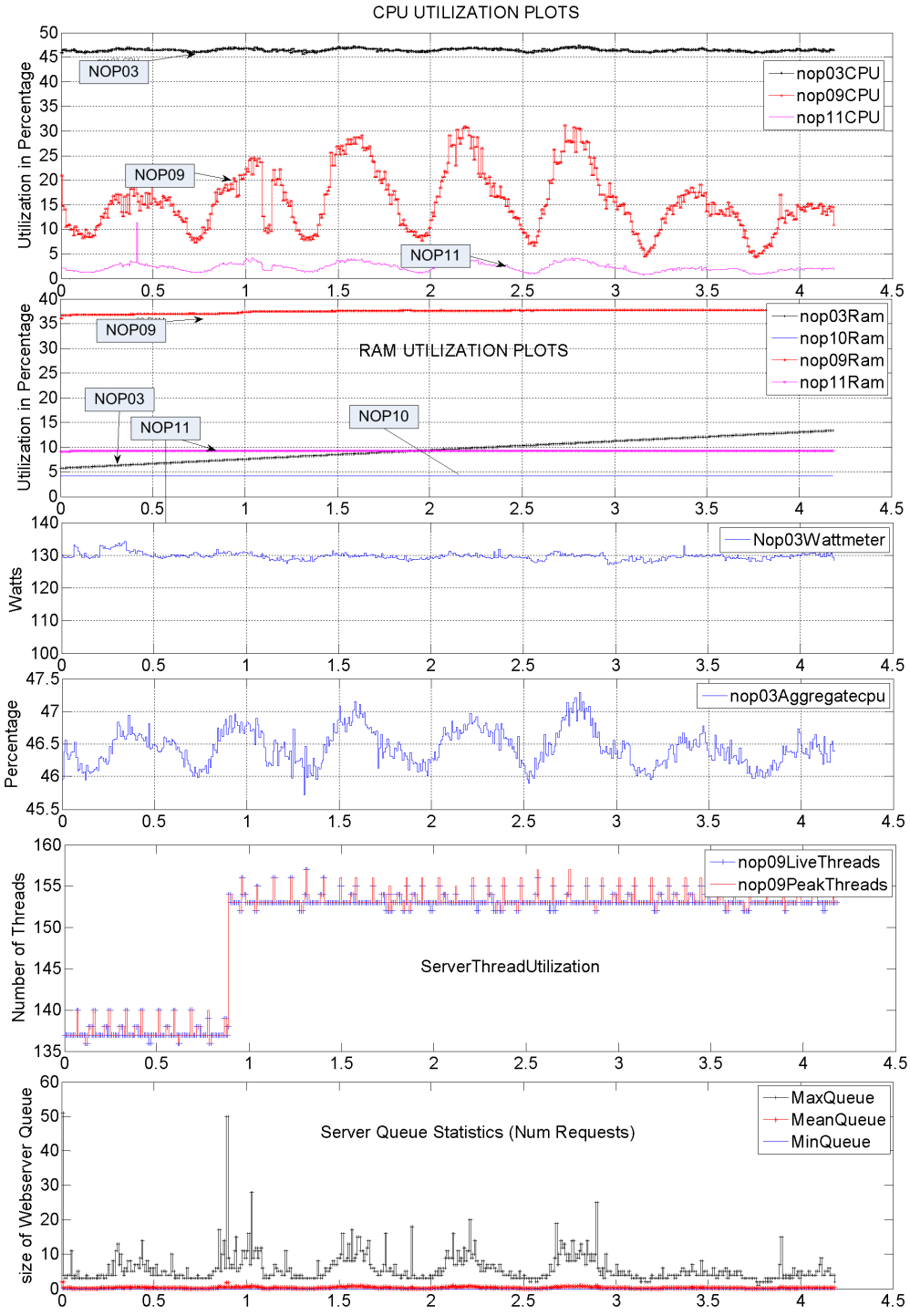


Figure 3.7

Web Server Behavior Modeling Experiment Results (continued from Figure 3.6).

An ExpoKF was used to estimate the system state. It is important to note that the system can be approximated as a $M/G/1/\infty$ PS queue if the system has no bottlenecks. In the presence of a bottleneck, the system resource utilization (not necessarily CPU) will approach 1 (or 100%). At that time, the system will change to the LPS queue model. However, as mentioned earlier, it is difficult to build a tractable model for LPS queuing systems. Hence, the operating regions of the system is identified, where the system changes the mode between two queuing models and the system is analyzed in the infinite PS queue region only.

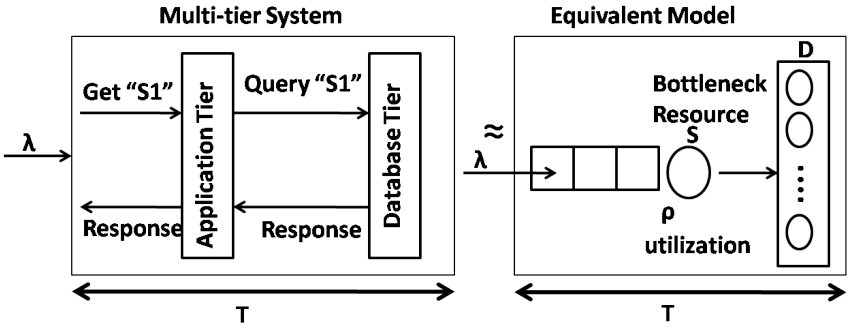


Figure 3.8

A Queuing Model for the Two-Tier System.

The ExpoKF equations were written in exponentially transformed variables $[x_1\ x_2]$, such that $S = e^{x_1}$ and $D = e^{x_2}$. Here, $x_1, x_2 \in \mathbb{R}$. Note that this transformation ensures $S, D \in \mathbb{R}+$: For a given time sample of observation k, Equation 3.2 defines the system update dynamics and Equation 3.3 defines the observation. $N(0, Q)$ and $V(0, R)$ are Gaussian processes measurement noises with mean zero and covariances of Q and R

respectively. One can verify that these equations described the behavior of a $M/G/1PS$ queue. Here, predicted bottleneck utilization $\hat{\rho}_k$ is given in Equation 3.4. Moreover, the ExpoKF does not update its state when the predicted bottleneck resource utilization becomes more than 1.

$$\begin{pmatrix} e^{\hat{x}_{1k}} \\ e^{\hat{x}_{2k}} \end{pmatrix} = \begin{pmatrix} e^{x_{1(k-1)}} \\ e^{x_{2(k-1)}} \end{pmatrix} + N(0, Q) \tag{3.2}$$

$$T = \frac{exp^{x_{1k}}}{1 - \lambda_k \, e^{x_{1k}}} + e^{x_{2k}} + V(0, R) \tag{3.3}$$

$$\hat{\rho}_k = \lambda_k \, e^{x_{1k}} \tag{3.4}$$

Figure 3.9 shows results of the off-line analysis of the logs with the help of the kalman filter. These logs were generated during the experiment in Section 3.4.3. Service time S and delay D are in millisecond while response time T is specified in seconds. According to Figure 3.9(sub-figure 1), the developed ExpoKF tracks service time S and delay D at the web server accurately with very low error variance as the experiment (Section 3.4.3) progresses. Additionally, in sub-figure 2, ExpoKF tracks the bottleneck resource utilization, which is similar to the CPU utilization of the system. However, it was noticed that sometime, the bottleneck utilization trend is not similar to the CPU utilization. In those cases, the number of available JAVA threads acted as the bottleneck. According to sub-figure 3, the predicted response time from the EKF, T_{pred}, and actual response time, T,

are also very close to each other, which indicates accuracy of the ExpoKF in capturing the response time dynamics of the web server system. This response time prediction is used in on-line fashion with a predictive control framework in the next chapter.

3.4.4 Impact of Maximum Usage of the Bottleneck Resource

This experiment was performed to observe the effect of the extremely high bottleneck resource usage on the web server performance. This test setup uses Daytrader application, which is a multi-threaded java based enterprise application hosted on *Web Sphere Application Server Community Edition (WASCE)* [41]. Daytrader receives incoming http requests on port number 8080. This Daytrader application serves the incoming http requests by creating a new child JAVA thread or through an existing JAVA thread if one is available in the pool of free threads. A newly arrived http request for the Daytrader application is handed over to the newly created or free child thread for further processing. This child thread rejoins the pool of free threads once it finishes the processing. A limit on the maximum number of JAVA threads can be imposed on the web server. This maximum limit can also be considered as the maximum concurrency limit of the web service system. In case of unavailability of a free thread due to already achieved maximum thread limit and empty pool of free threads, a newly arrived http request has to wait for thread availability, which impacts the application performance severely in terms of the response time. Therefore, the number of available threads was identified as the bottleneck resource of the system.

A number of experiments were performed with different settings for Max JAVA Threads in WASCE. This parameter sets the maximum number of threads that can be used for re-

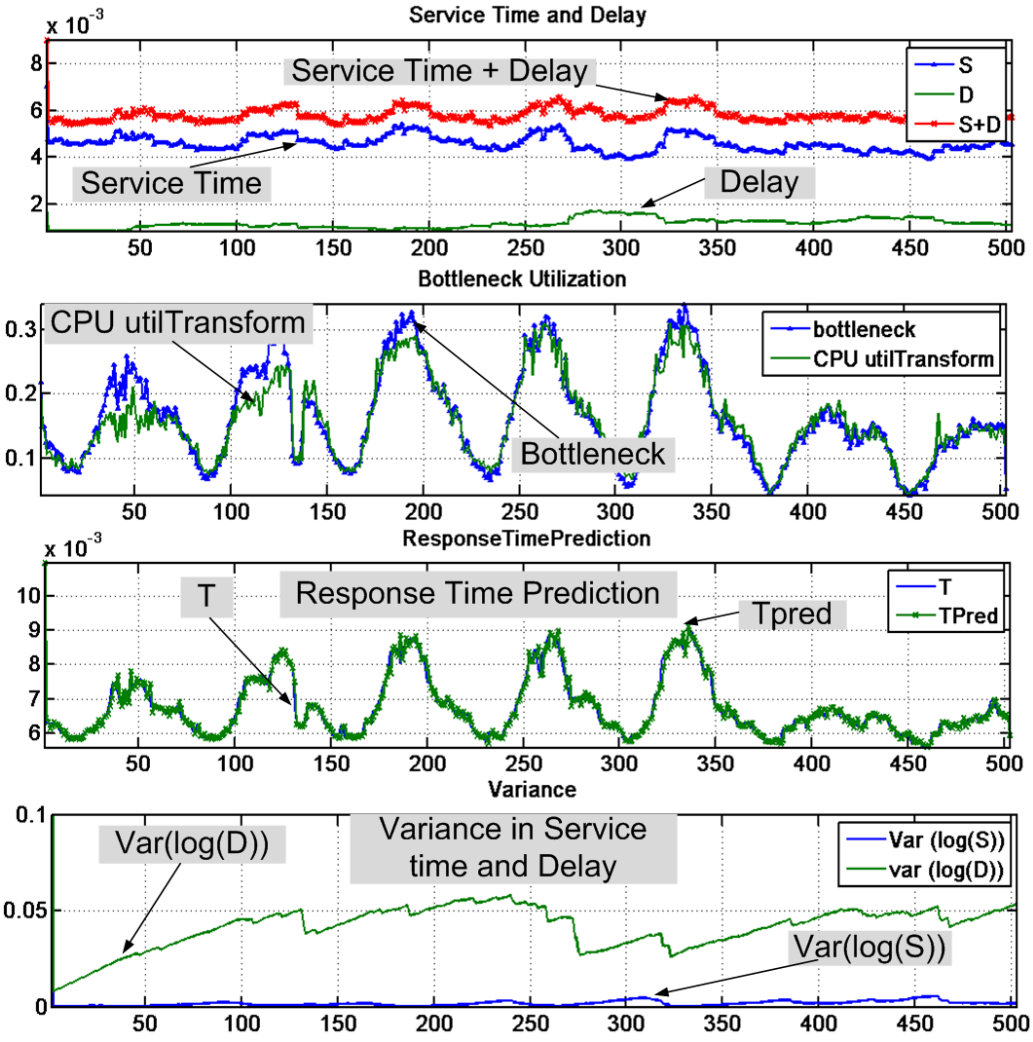


Figure 3.9

Offline Exponential Kalman Filter Output from Section 3.4.3.

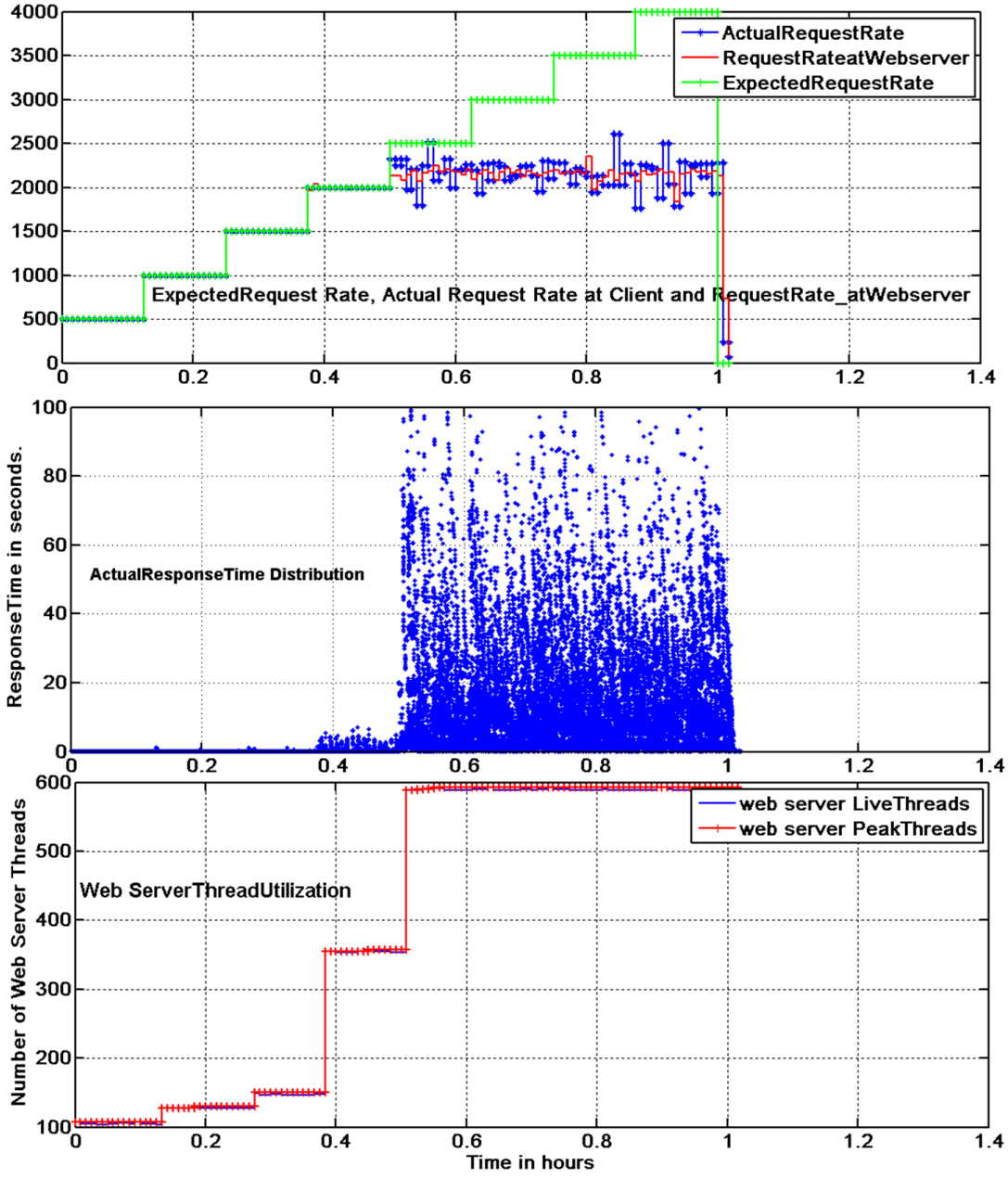


Figure 3.10

Impact of Maximum Utilization of the Bottleneck Resource on Web Server Performance.

quest processing. Based on the observation, there are typically 90 more system threads, which are not accounted for under this cap. The results from one of these experiments are shown in Figure 3.10, where max JAVA threads limit is set to 500. This figure shows that at the maximum utilization of the bottleneck resource, system performances deteriorates significantly and the response time from the web server becomes unpredictable. Furthermore, this is the region, where the system makes transition from a PS queue to an LPS queueing system.

Once the system reaches the max utilization of the bottleneck resources, it restricts entry for more requests into the system resulting in the maximum utilization of the incoming system queue, which in turn results in the rejection of the incoming client requests from the user. Therefore, to achieve predefined QoS specifications, the system should never be allowed to reach the maximum utilization of the bottleneck resource. Additionally, this boundary related to the maximum usage of the bottleneck resource can also be considered as a "Safe Limit" of the system's operation.

3.4.5 Impact of Limited Usage of Bottleneck Resource

In this experiment, the web server performance was observed when the bottleneck resource utilization varies from minimum to maximum and was restored back to the minimum value. This type of investigation provides knowledge about web server performance if bottleneck resource utilization is lowered from the maximum limit through a controller that maintains the QoS objective of the multi-tier system.

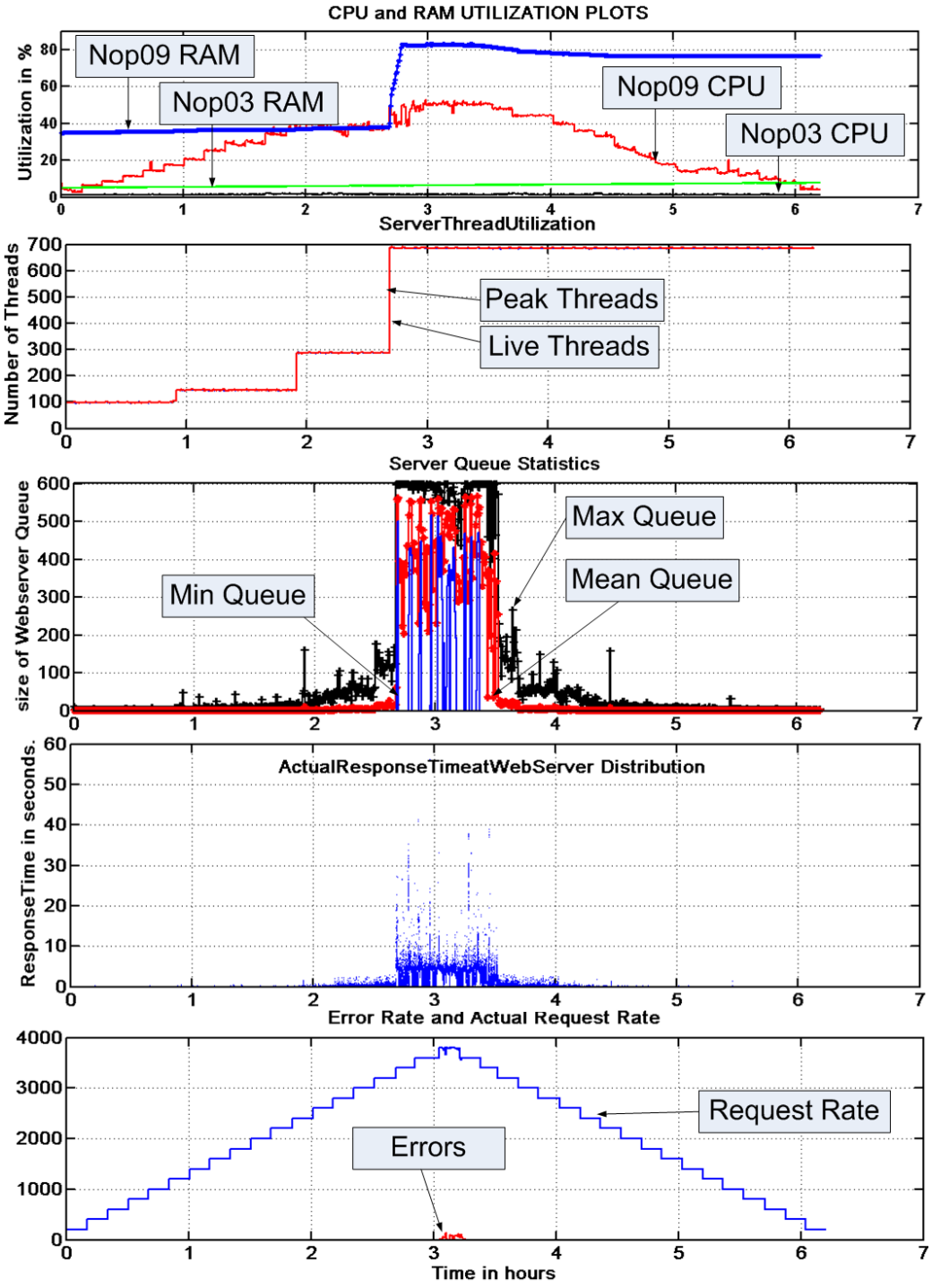


Figure 3.11

Web Server Performance while Limiting the use of Bottleneck Resource.

The configuration settings for this experiment were the same as Section 3.4.3. The MAX number of JAVA threads for this experiment were 600. The client request trace profile used for this experiment is shown in Figure 3.11, sub-figure 5. According to the results shown in Figure 3.11, system utilization (sub-figure 1) and response time (sub-figure 4) follow the trend of the applied client request profile (sub-figure 5). The sudden increase in the size of the web server queue (sub-figure 3) indicates the contention of the computational resources among all of the pending requests inside the system. The sudden increase in RAM utilization is due to the increase in the thread utilization of the system. Additionally, by comparing the request rate and response time plot in Figure 3.11, it is apparent that by lowering the system utilization and client load on the web server, a web server can be brought back to the state, where QoS objectives (queue size and response time) of the system are restored.

3.4.6 Kalman Filter Analysis

Results of the experiment were analyzed with the help of the ExpoKF described in Section 3.2.2 and the results of this analysis are shown in Figure 3.12. According to Figure 3.12, the ExpoKF tracks service time and delay at web server accurately with low variance as the experiment progresses. One can notice the regions where the bottleneck resource utilization approaches unity while the CPU utilization is less than one. Upon further investigation of those time samples, the maximum number of Java threads available in the web server were found to be the bottleneck. During the experiment, when predicted utilization of the bottleneck resource is more than one, the ExpoKF does not update its

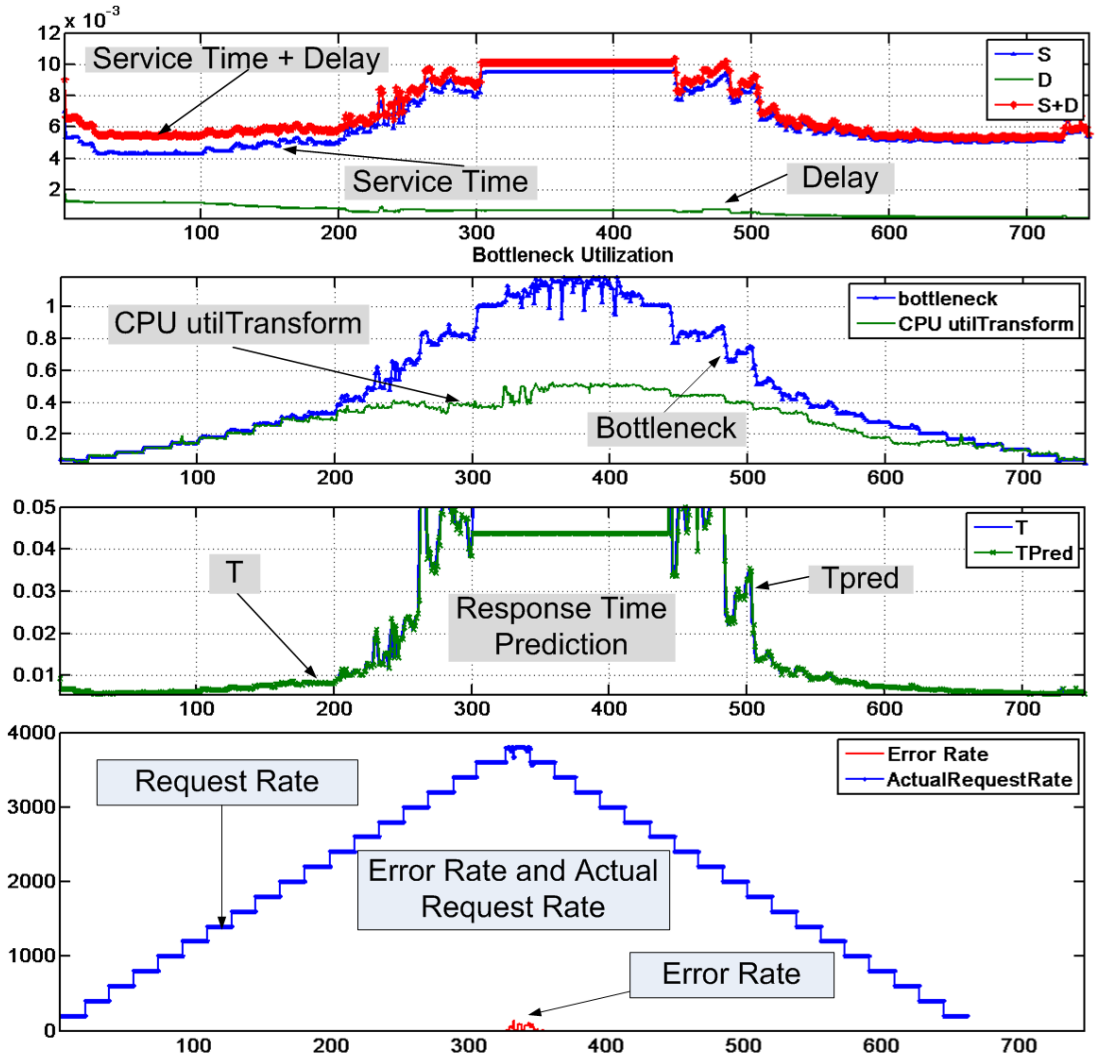


Figure 3.12

Offline ExpoKF Analysis of the Results from Figure 3.11.

states. In the later half samples of the experiment, the Kalman filter again starts tracking the bottleneck utilization and response time efficiently. Therefore, the goal of any successful controller design for performance optimization of the system will be to drive the system for operating in the stable region, where the bottleneck resource utilization is less than 1.

3.5 Summary

In this chapter, we presented a simple and novel approach to develop system models with low variance for multi-tier enterprise systems. This approach can be used for performance modeling of multi-tier enterprise applications by using regression analysis of system measurements corresponding to a discretized domain of operating settings. Performance of this modeling approach was demonstrated on a two-tier benchmark enterprise application "*Daytrader*." Experimental results show that the generated models can accurately predict the system behavior with very low variance.

CHAPTER 4

PERFORMANCE MANAGEMENT OF A WEB SERVICE DEPLOYMENT USING A MODEL-BASED CONTROL APPROACH

In this chapter, a model-based predictive controller is presented that uses the web service performance model and power consumption model developed in the previous chapter. This predictive controller utilizes the previously developed exponential Kalman filter (ExpoKF) for estimating the computational requirement of the incoming http requests, predicts aggregate response time of the incident requests, and uses the power consumption model to estimate the power consumption during the next time sample. This predictive controller optimizes the system behavior for QoS objectives through continuous monitoring of the underlying web service system and by choosing the optimal control input that maintains the QoS objectives of the system in next time sample.

This predictive controller is an advanced version of the *L0 Controller* proposed in [93]. Specific contributions of this dissertation in developing the advanced predictive controller are: utilizing the ExpoKF based system model for predicting the dynamic computational requirements of the incoming workload, using a more accurate power consumption model that utilizes CPU core frequency and CPU utilization in a virtual environment, and optimizing the performance of a typical multi-tier enterprise application *Daytrader*.

4.1 Preliminaries

In this section, model-predictive control basics and a typical control theory based performance management system are briefly presented.

4.1.1 Model-Predictive Control

Model-Predictive Control (MPC) is a model-based approach to control a system (chemical, power system, computer, etc.) by using an explicit system model. This system model is used at each control interval to predict the future system behaviour by applying a sequence of control adjustments on the system. These control adjustments are computed for optimizing the future system behavior and keeping the system within certain pre-specified constraints by maximizing a utility function in extremely dynamic operating environment. The MPC strategy is also known as Receding Horizon Control (RHC). In this strategy, an open loop optimization problem is solved at each control step for future prediction horizon and pre-specified constraints are applied to the system to compute the sequence of control adjustments. The first control adjustment from the sequence is applied to the system and this step is repeated again at the next control interval. The MPC strategy has several benefits over other advanced control technologies:

1. It contains various tuning parameters to control the computational overhead of the control strategy, such as the prediction horizon and an optimization algorithm library for utility function.
2. It handles operating and system state constraints explicitly in the control strategy.
3. The system model used by the MPC can be updated at run-time without making any changes in the MPC strategy.
4. The complexity of the developed system model can also affect the performance of the MPC strategy. The computational overhead of the developed MPC approach will

depend on the number of state variables, the environment inputs, and control input set used inside the system model.

In the past, MPC strategies have been applied to a variety of industrial control systems, such as electrical, chemical, and mechanical systems [124]. However, recently these strategies have also been applied for controlling computing systems [51, 93, 92, 48], Unmanned vehicle formations [75, 100], and electronic shipboard power systems [136]. MPC strategies are applied to a large number of systems, which have utility functions as linear unconstrained to non-linear constrained. However, in this chapter, only the non-linear constrained utility function is considered for developing the appropriate MPC strategy. More details of the utilized MPC strategy will be explained in future sections of this chapter.

4.1.2 Control Theory based Management of Computing Systems

Control theoretic approaches are utilized for automating the system management tasks by constantly observing the system performance parameters and by applying computed control commands (inputs). These control commands keep the system within the boundaries of permissible system states, which are defined through state constraints. A typical control system is shown in Figure 4.1. *System Set Point* is the desired system state, which a system tries to achieve during its operation. *Control Error* is the difference between the desired system set point and the measured output during the system operation. *Control Inputs* are the set of system configuration parameters that are applied to the system dynamically for changing the performance level of the system. *Controller Module* takes observation of the measured output and provides the optimal combination of different control inputs to achieve the desired set point. *Estimator Module* estimates the unknown parameters for

the system based upon the previous history by using statistical methods. *Target system* is the system in consideration, while *System Model* is the mathematical representation, which defines the relation between input and output variables of the system. *Learning Module* observes system output through the monitors and extracts system behavior information by using the statistical methods.

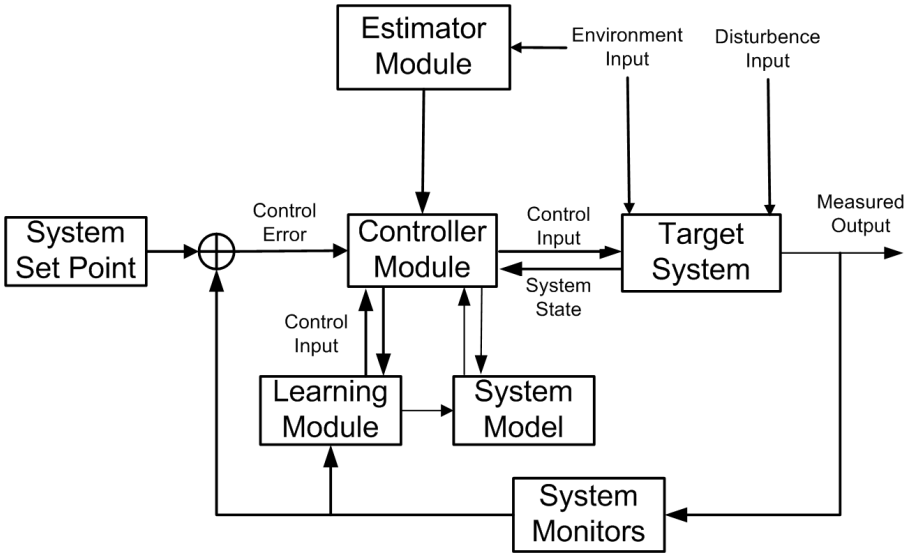


Figure 4.1

Main Components of a General Control System.

4.2 Model-Based Online Predictive Controller

Elements of the developed predictive control approach are shown in Figure 4.2. *Target System* represents the web service node that is monitored through various *Monitors*. These monitors collect measurements from the web service node and forward them to the Kalman filter (i.e. ExpoKF) at run-time. The ExpoKF estimates the computational nature

of the incoming http requests and corresponding system state in service time and response time. These estimated system states, future request estimates from *Request Forecaster*, are used by the *Controller Model*. The Controller model uses various *Control Algorithms* and desired system *Set Points* to calculate the optimal value of control input for next sample. This control input is applied to the web service system for the next sample.

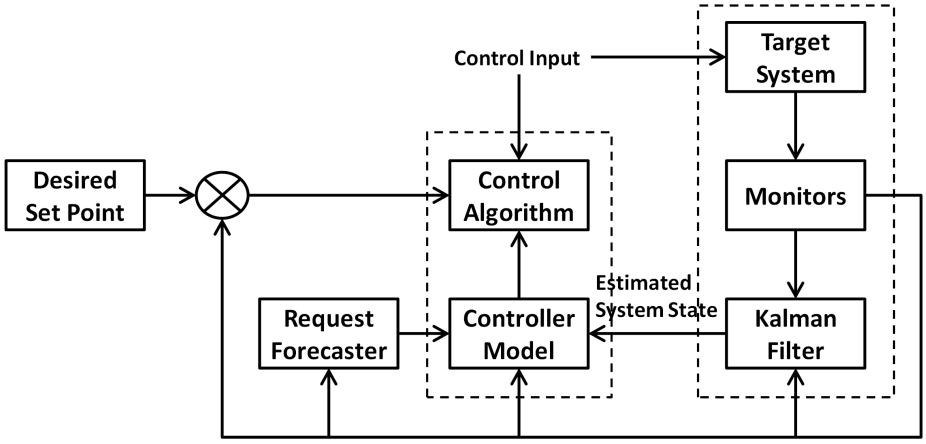


Figure 4.2

Elements of the Predictive Control Approach.

4.2.1 System Variables

Although there are a large number of system parameters listed in Table 3.2, a small set of the most important parameters is chosen for the predictive controller in order to show the performance of web service modeling approach described in Chapter 3. The chosen control input is the *CPU core frequency* due to its impact on the system performance as defined by response time of the web service and power consumption of the physical system.

System queue size and *response time* were chosen as the system state variables, while *power consumption* was chosen as the performance variable. The developed predictive controller tries to minimize the application queue size and the total response time as parts of the system operating cost function J (described later in this chapter) while keeping the power consumption at minimum. Experiments (reported in Chapter 3) indicate that the higher value of application queue represents contention in the computational resources required for the application while total response time indicates a system's capability to process the requests residing inside the system queue in a timely manner. Therefore, the application queue size and total response time are also chosen as a component in cost function.

The system model describes the dynamics of the active state processing element. We used a multi-layered queueing model to capture the dynamics of the system. These system dynamics can be represented as the following state space equation:

$$x(k+1) = \Phi(x(k), u(k), \omega(k)) \tag{4.1}$$

where $x(k) \subseteq \mathbb{R}^n$ is the system state at time sample k, the set of user controlled system inputs is $u(k) \in U \subseteq \mathbb{R}^m$ (e.g., CPU frequency at time k), and $\omega(k) \in \Omega \subseteq \mathbb{R}$ is the environment input at time k (e.g., incoming http requests toward the web service). The state update function Φ captures the relationship among the observed system state $x(k)$, the control inputs $u(k)$ that adjust system parameters, and the environment input $\omega(k)$, which can not be controlled.

Since the current value of the environment input $\omega(k)$ is not available until the next sampling instance $k+1$, the system dynamics can only be captured in state update function Φ by using a system model with estimated values of environment input $\hat{\omega}(k)$:

$$\hat{x}(k+1) = \Phi(x(k), u(k), \hat{\omega}(k)) \tag{4.2}$$

where $\hat{x}(k+1) \subseteq \mathbb{R}$ is the estimated system state at time sample $k+1$ while $\hat{\omega}(k)$ is the expected environment input at time sample k.

4.2.2 Forecasting Environment Input

The estimation of future environmental input $\hat{\omega}(k)$ to the system is crucial for estimating the future system states $\hat{x}(k+1)$ and system outputs. In this predictive control approach, an *autoregressive integrated moving average* (ARIMA) filter (similar to [93]) was used to estimate the environmental input $\hat{\omega}(k)$ by using function θ according to the following equation.

$$\hat{\omega}(k) = \theta(\omega(k-1, r)) = \gamma_1 \omega(k-1) + \gamma_2 \omega(k-2) + (1 - (\gamma_1 + \gamma_2))\bar{\omega}(k-3, r) \tag{4.3}$$

where γ_1 and γ_2 are user specified weights on the current and previous arrival rates for prediction and $\bar{\omega}(k-3, r)$ represents average value of environment input between time samples $(k-3)$ and $(k-3-r)$. $\omega(k-1, r)$ represents the array of environment inputs between sample time $(k-1)$ and $(k-1-r)$. More accurate forecasting can be performed by accommodating the previous estimation errors as specified in [93].

4.2.3 Controller Model

In order to combine the estimated power consumption and the predicted response time, the predictive controller uses a different internal system model. The controller model uses the estimated system state, predicted response time, and predicted power consumption to make the system decisions. The system state for this experiment, $x(k)$, at time k is defined as set of system queue $q(k)$ and response time $r(k)$, that is:

$$x(k) = [q(k)\, r(k)] \tag{4.4}$$

The queuing system model dynamics is given by the equations:

$$\hat{q}(k+1) = \left[q(k) + \hat{\omega}(k) - \frac{u(k)}{\hat{c}(k)\, u^m}\, T \right]^+ \tag{4.5}$$

and

$$\hat{r}(k+1) = (1 + \hat{q}(k+1))\, \frac{\hat{c}(k)\, u^m}{u(k+1)} \tag{4.6}$$

where $[a]^+ = max(0, a)$. $q(k)$ is the queue level of the system at time k, $\omega(k)$ is the arrival rate of http requests, $r(k)$ is the response time of the system, $\hat{r}(k+1)$ is expected response time of the system, $u(k) \in U$ is the frequency at time k (U is the finite set of all possible frequencies that the system can take), u^m is the maximum supported frequency in the system, $\hat{c}(k)$ is the predicted average service time (work factor in units of time) required per request at the maximum frequency. The online ExpoKF (see Section 3.4.3)

estimates the service time $\hat{S}(k)$ of the incoming http request at the current frequency $u(k)$, which is scaled against the maximum supported frequency of the system to calculate the work factor $\hat{c}(k)$ according to the following equation:

$$\hat{c}(k) = \hat{S}(k)\,\frac{u(k)}{u^m} \tag{4.7}$$

$E(k)$ is the system power consumption measured in watts at time k, which is calculated by using the system power consumption model developed in Section 3.4.1.

4.2.4 Operating Constraints

The web service system in consideration must follow the strict operating constraints of the system state and the control inputs. In general, these operating constraints are formulated in feasible (or valid) system state variables and the permissible control inputs at a particular system state. These constraints represent the operating state of the system for safe operation or extreme violation of QoS objectives. These constraints are imposed on the system in two ways: Firstly, by preventing the system transition to the unsafe states by reducing the control input set, which will not allow these transitions. Secondly, by forcing system to make a transition, which moves system back from these unsafe states to safe states through applying corrective control inputs. In this case, operating constraints were applied to the system state x and the control input u. These constraints include the feasible domain of the system states ($x(k) \in X$), the permissible control inputs ($u(k) \in U$) at each system state ($x(k)$), and the desired terminal state (x_s). The system in consideration must follow the admissible trajectory of the system states ($x(k) \in X$) by using permissible con-

trol inputs ($u(k) \in U$ defined by the function $Valid(x(k), u(k))$) to drive the system close to the desired terminal state x_s, while keeping the control, transient, and terminal costs to minimum. Here, function $Valid(x(k), u(k))$ returns 1 if control input $u(k)$ is permissible at system state $x(k)$, otherwise it returns 0.

4.2.5 Control Algorithm and Performance Specification

A limited look-ahead control algorithm (similar to [93]) is utilized by the developed model-predictive controller to solve the control optimization problem. According to this algorithm, starting from a time k_0, the controller solves an optimization problem defined over a predefined horizon H ($k = k_0 + 1, k_0 + 2, ..., k_0 + H$) and chooses the first input $u(k_0)$ that minimizes the total cost of operating the system, J, in the future during the prediction horizon.

The prediction horizon was limited to $H = 2$ because there is a computation cost associated with a longer prediction horizon and estimation error at each step also accumulates with subsequent steps in the horizon. Formally, cost function $J(k)$ can be specified as:

$$J(k) = \|x(k) - x_s\|_A + \|E(k)\|_B \tag{4.8}$$

The cost function J at time k, is the weighted conjunction of drift of the system state $x(k)$, ($x(k) = [q(k)\, r(k)]$) from the desired set point x_s, of the system state ($x_s = [q_s\, r_s]$ where q_s = desired maximum queue size, r_s = desired maximum response time) and power consumption $E(k)$ (desired power consumption is 0). A and B are user specified relative weights for the drift from the optimal system state x_s and power consumption $E(k)$,

respectively. The power consumption $E(k)$ is predicted with the help of the *lookup table* generated in Section 3.4.1 based on the current CPU core frequency and the aggregate CPU utilization of the physical server. Formally, the chosen control input can be calculated by using algorithm shown in Figure 4.3.

Input: System Stare $x(k)$, Look Ahead Horizon H,
Input: Control Input Set U, $U = [u_1, u_2, ..,]$,
1: $s_k := x(k)$, $Cost(x(k)) = 0$
2: **for all** k within prediction horizon of depth H **do**
3: Forecast environment parameters $\hat{\omega}$ for time $k+1$
4: $s_{k+1} := \phi$
5: **for all** $x \in s_k$ **do**
6: **for all** $u \in U$ **do**
7: $\hat{x} = \Phi(x, u, \hat{\omega})$ /* Estimate state at time $k+1$ */
8: $Cost(\hat{x}) = Cost(x) + J(\hat{x})$
9: $s_{k+1} := s_{k+1} \cup \{\hat{x}\}$
10: **end for**
11: **end for**
12: $k := k+1$
13: **end for**
14: Find $x_{min} \in s_N$ having minimum $Cost(x)$
15: $u^*(k) :=$ initial input leading from $x(k)$ to x_{min}
16: return $u^*(k)$ /* Apply $u^*(k)$ to the system for next control period */

Figure 4.3

Predictive Control Algorithm for Calculating Value of Control Input.

4.3 Case Study: Power Consumption and QoS Management of a Web Service

This section uses the concepts introduced in the earlier chapters for managing power consumption in a physical server while maintaining the predefined QoS objectives of minimum response time under a time varying dynamic workload for the Daytrader application

hosted in a virtualized environment (see Section 3.4.3). The following subsections will provide the details of this case study.

4.3.1 Experiment settings

Experimental settings and incoming request profile for this experiment were kept similar to Section 3.4.3 for direct comparison of the webserver performance with and without the controller deployment. A local performance monitor executing on the web server (*Nop09*) collected, processed, and reported performance data after every SAMPLE_TIME (30 seconds) to the controller, which is executing on the physical host machine (*Nop03*). The average queue size of the system is measured based on the total resident requests in the system at the previous sample, (plus) the total incident requests into the system, and (minus) the total completed requests from the system in the current sample duration. Schematic description of the predictive controller with web server deployment is described in Figure 4.4

4.3.2 System State for Predictive Control

The EKF described in the previous chapter was used to track the computational nature of the incoming http requests. The two main parameters received from the filter are the current service time S and predicted response time T_{pred}. These values are then plugged into the model described in Section 4.2.3. The power model described in Section 3.4.1 was used to estimate the system (physical node of webserver) power consumption. With the help of these system and power models, the predictive controller estimates the optimal configuration of the system in terms of CPU core frequency by using algorithm shown

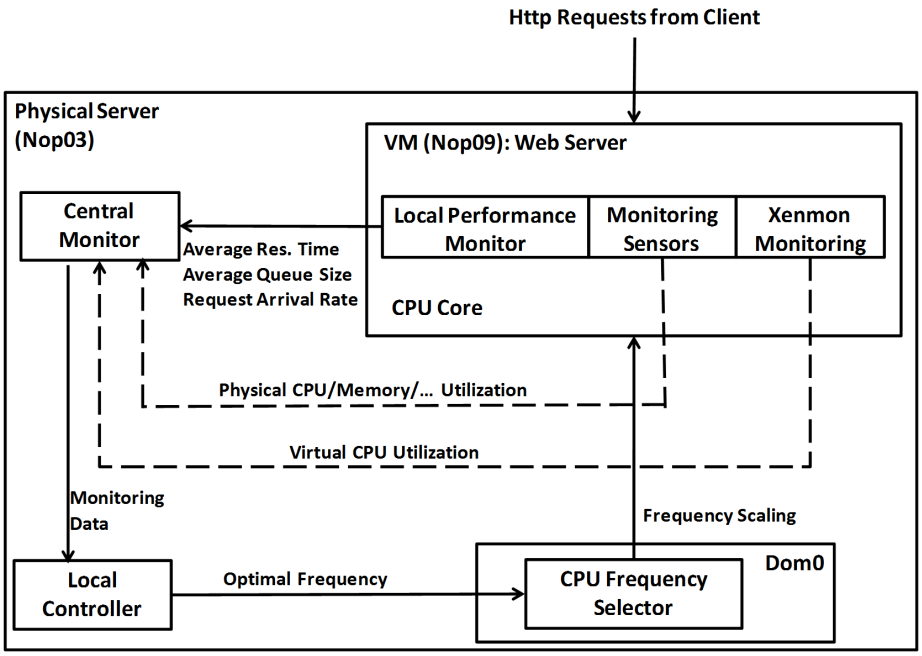


Figure 4.4

Predictive Controller System Setup

in Figure 4.3. Also, the performance of the online controller directly depends upon the accuracy of the KF estimations for parameters of the webserver performance model and the power consumption model of the physical system.

For this experiment, the optimal system state was chosen as $x_s = [q_s, r_s]$ with $q_s = 0$ and $r_s = 0$, which shows an inclination toward keeping both the system queue and the response time to a minimum. A and B (user specified relative weights for cost function) were chosen as 10000 and 1, respectively, to penalize the multi-tier system a lot more for increment in queue size and response time compared to the increment in power consumption. Additionally, look ahead horizon value H was 2 for the current experiment. Request fore-

casting parameters γ_1 and γ_2 were equal to 0.8 and 0.15, respectively, to put the maximum weight on the current arrival rate in estimating the future request arrival rate.

4.3.3 Experimental Results

Results from this experiment are presented in Figure 4.5 and Figure 4.6. The direct comparison of the results from the experiments in presence and absence of the controller for response time, system resources, and power consumption are presented in Figure 4.6. Additionally, the ExpoKF estimations for this experiments are shown in Figure 4.7, while the accuracy of the power consumption model is presented in Figure 4.8. The comparison between the actual numbers of http requestes towards the webserver and the numbers estimated by the ARIMA estimator is shown in Figure 4.9, which represents the accuracy of the ARIMA estimator.

The $Nop03$ CPU core frequencies during the experiment are shown in Figure 4.5 (sub figure 2) and the JAVA thread utilization of the web server is shown in sub-figure 3. Sub-figure 4 shows the queue size of the web server through the method described in Section 4.3.1. The most interesting plot in Figure 4.5 is sub-figure 2, which shows the change in frequency of the CPU core from the controller to achieve predefined QoS requirements based on the control steps taken by the controller. After direct comparison of sub-figure 2 and sub-figure 1 from Figure 4.5, we can see that $Nop03$ CPU core frequency is changed when incident request rate at the web server changes. Additionally, the controller chose 1.2 Ghz frequency for the CPU core until there was some sudden increase or decrease in the incident request rate. Furthermore, the controller does not change the frequency of the

core too frequently, even when the incident request rate is changing continuously. This less often change in CPU core frequency shows a minimal disturbance in the system operation due to the predictive controller.

The aggregate CPU utilization and memory utilization (sub-figure 4) of the application and the database tier are shown in Figure 4.6. These utilization plots show negligible memory and CPU utilization overhead due to the controller. However, the increment in CPU utilization while using controller is due to the lower value of CPU core frequencies. The power consumption plot for $Nop03$ is shown in Figure 4.6(sub-figure 3), while statistics (max and min) of observed response time at web server are shown in Figure 4.6 (sub-figure 1 and 2). These response time statistics show that even after applying lower frequency values by the predictive controller, the response time remains in a similar range as in the absence of a controller with highest frequency. It also shows that while managing to decrease power consumption, the controller does not affect QoS objectives of the web service system negatively.

According to Figure 4.7, the ExpoKF tracks the average response time T of the incident requests and bottleneck utilization with high accuracy. Additionally, the estimated service time of the incident requests by the EKF shows minimal variation. According to sub-figure 3, the predicted response time from the EKF T_{pred} and actual response time T are also very close to each other, which reflects accuracy of the EKF even in online deployment. Service time S and delay D are in millisecond while response time T is specified in seconds.

According to Figure 4.6 (sub-figure 3), the controlled experiment uses a lower frequency most of the times, which results into considerable amount of power saving (18%)

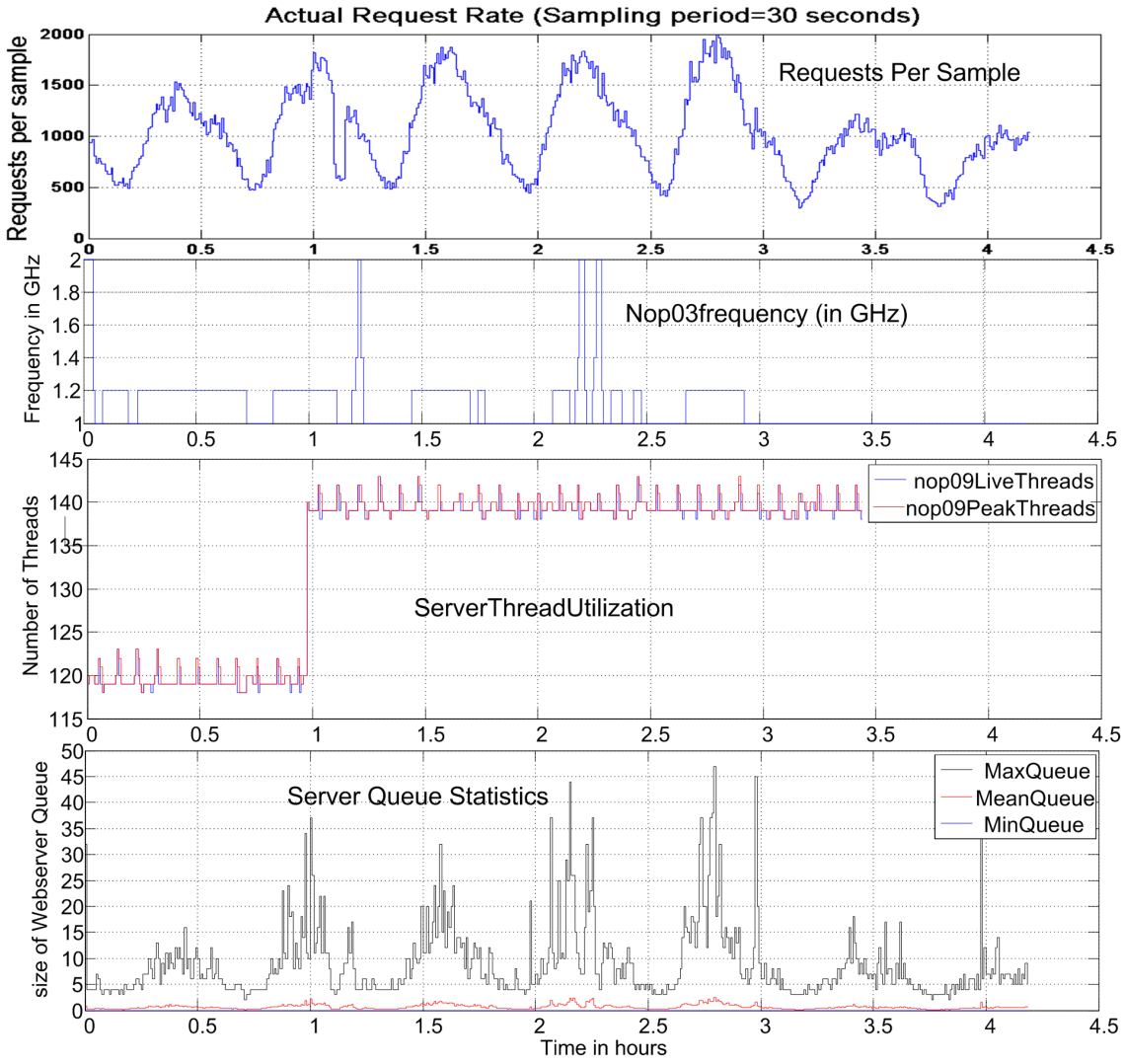


Figure 4.5

Web Server Performance Results With Predictive Controller.

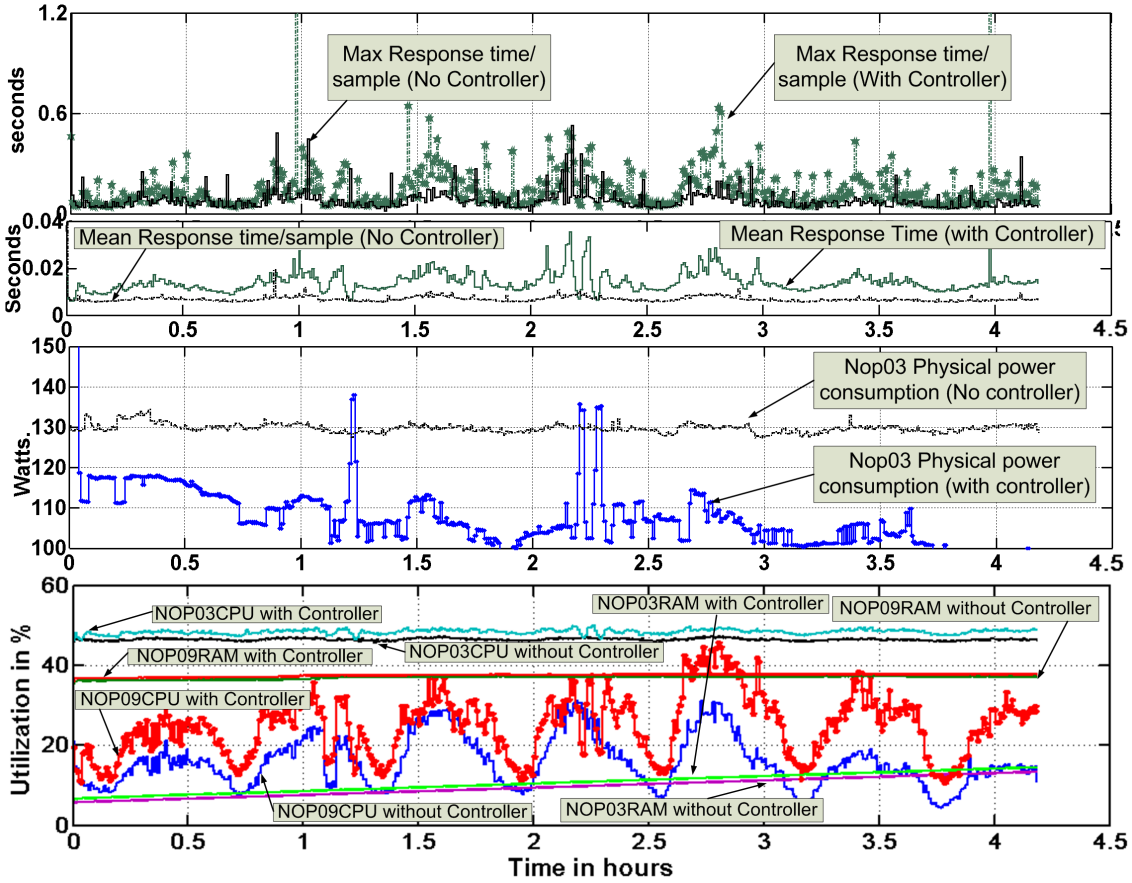


Figure 4.6

Comparison of Web Server Performance with Controller and Without Controller from Section 3.4.

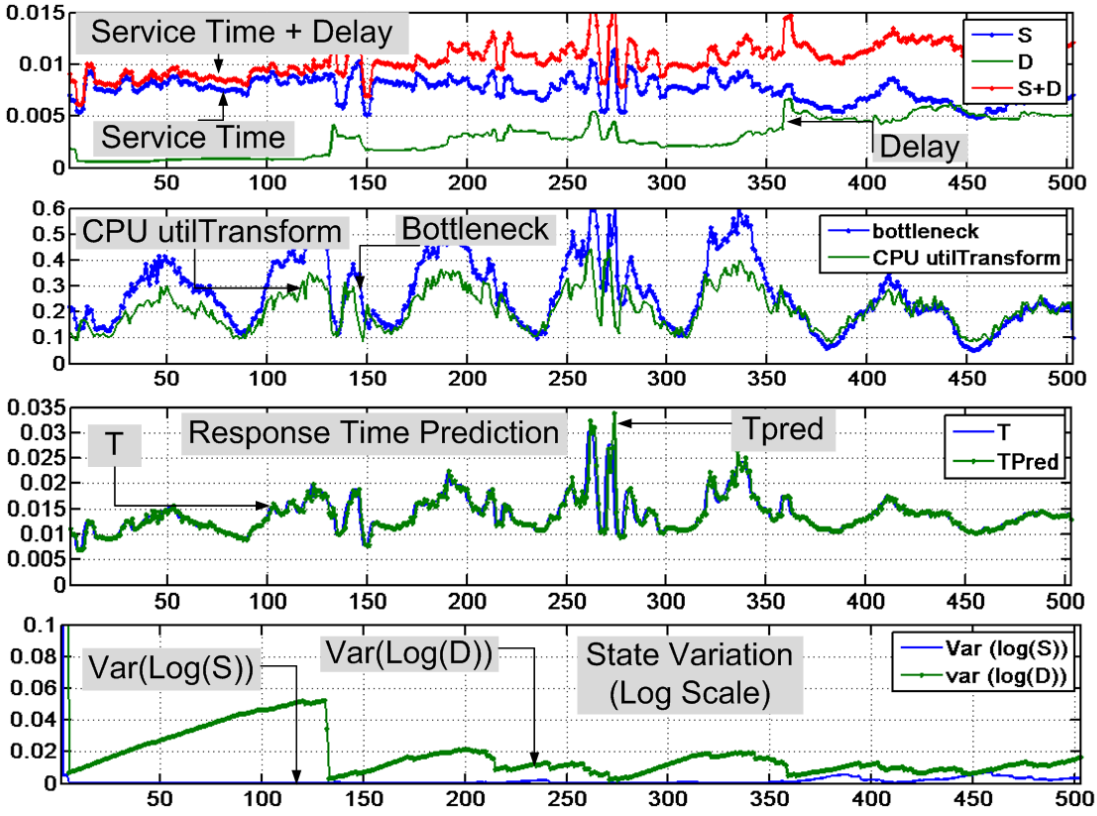


Figure 4.7

Online EKF output Corresponding for experiment with Predictive Controller.

over a period of four hours of experiment (Figure 4.6 sub-figure 3) compared to the baseline experiment shown in Section 3.4. The controller changes the frequency of the CPU core on very few occasions; however, the controller is able to identify the sudden increase in the incident request rate, which reflects adaptive nature of the controller in the case of dynamic environment conditions.

According to Figure 4.8, the power model developed in Section 3.4.1, estimates the power consumption in the physical machine *Nop03* with higher accuracy with only 5% average error in prediction. Additionally, the JAVA thread utilization (see Figure 4.5) is still less in case of controller, which indicates that even after slowing down the system, incident requests are getting served in time without much contention of computational resources. Furthermore, according to Figure 4.5, mean server queue statistics are also in the same range for controlled and uncontrolled experiments.

4.4 Summary

This chapter demonstrates that the system model developed in Chapter 3 can be combined with a predictive controller to maintain the system in a closed boundary of QoS in extremely dynamic and unpredictable operating environment. According to the results shown in this chapter, the developed system model, based on EKF, tracks the system performance online with high accuracy. Additionally, the proposed power consumption model of the system used by the controller predicts the overall physical server power consumption with 95% accuracy. Using this model, we showed that we can optimize system performance and achieve 18% reduction of power consumption in four hours of the experiment

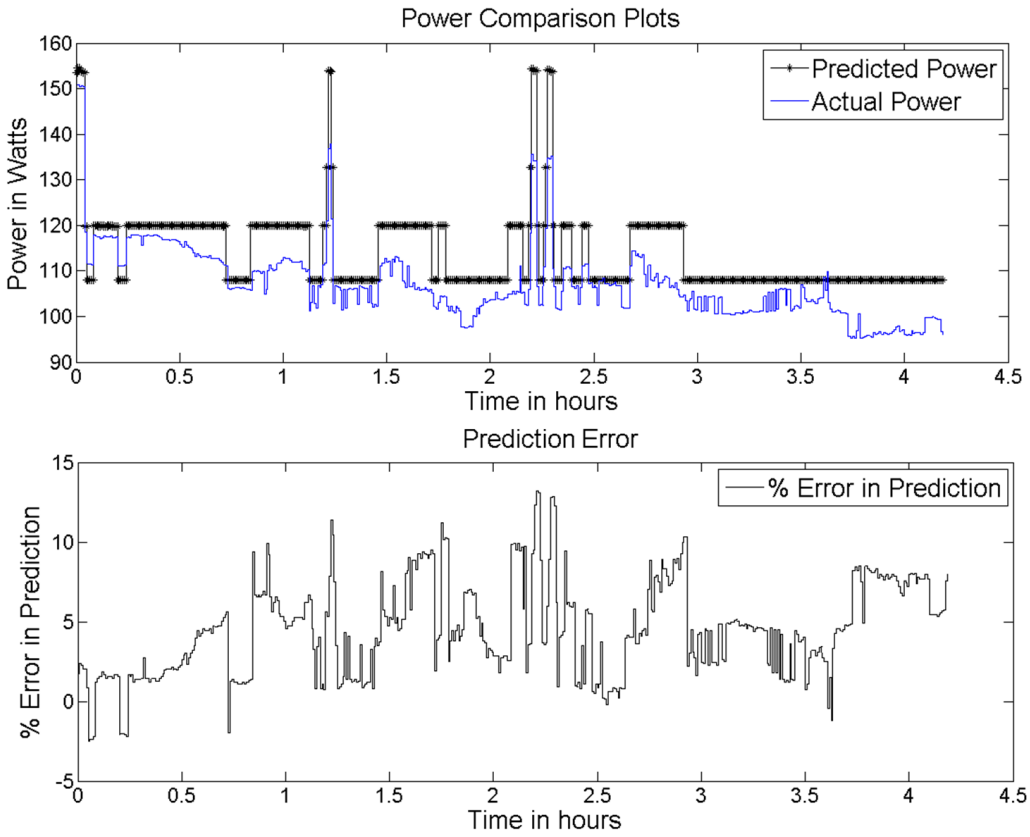


Figure 4.8

Comparison of Power Consumption for Actual Vs Predicted one Through Power Consumption Model in Section 3.4.1.

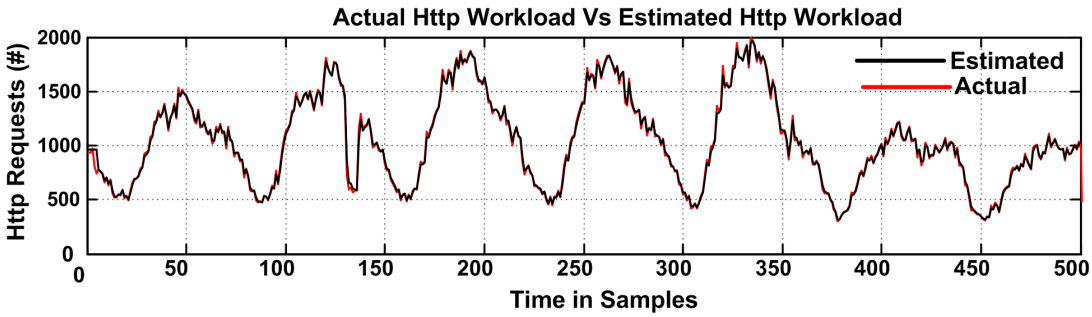


Figure 4.9

Comparison of Actual Http Workload from the Clients Vs Predicted one Through the Estimator.

in single physical server with negligible impact on the response time. Furthermore, the experimental results (CPU and RAM consumption with and without the controller) indicate that the proposed approach has low run-time overhead in terms of computational and memory resources.

CHAPTER 5

A REAL-TIME AND FAULT-TOLERANT DISTRIBUTED MONITORING SYSTEM

In this chapter, a distributed monitoring system, *"RFDMon,"* is introduced for an efficient exchange of measurements among the computing nodes of a distributed system. RFDMon was developed for comprehensive monitoring of the system resources, system health, and application performance statistics in a distributed deployment. RFDMon monitors computing systems in a distributed manner and reports each event to the system administrator with maximum accuracy and minimum resource overhead. It also introduces self-configuring properties in the distributed monitoring system. RFDMon uses ACM: ARINC-653 Component Framework [73] and Open splice [31], which is an open source implementation of Data Distribution Services [5].

5.1 RFDMon: Real-Time and Fault-Tolerant Distributed Monitoring System

The initial version of this monitoring system was developed at the ISIS, Vanderbilt University, for Fermi Lab, IL, to monitor scientific clusters in order to identify the location and cause of failures during the execution of scientific experiments. In this dissertation, "RFDMon" is extended to monitor the performance statistics of the deployed applications in distributed environment and to introduce self-configuration and self-healing autonomic properties for correcting faults in the monitoring system itself.

5.2 Preliminaries

The developed monitoring system *"RFDMon"* consists of two major modules: the *Distributed Sensors Framework* and the *Infrastructure Monitoring Database*. The distributed sensors framework utilizes data distribution services (DDS) middleware standard for communication among the nodes of the distributed infrastructure by using the publish-subscribe mechanism [78]. It uses Opensplice Community Edition DDS [31] and executes a sensor framework on top of the ARINC component framework [45]. The infrastructure monitoring database uses Ruby on Rails [33] to develop a web service, which is used to update the database with measurements and to display the stored data from the database on the administrator web browser. In this subsection, the key concepts of the publish-subscribe mechanism, DDS, ARINC-653, and Ruby on Rails development framework are briefly discussed.

5.2.1 Publish-Subscribe Mechanism

The publish-subscribe is a communication model for sharing data (information) in distributed infrastructures. Various nodes can communicate with each other by sending (publishing) data and receiving (subscribing) data anonymously through the communication channel as specified in the infrastructure. Publishers and subscribers need only the *name* and *definition* of the data in order to communicate. Publishers do not need any information about the *location* or *identity* of the subscribers, and vice versa. Publishers are responsible for collecting the date from the application, formatting it per the data definition, and sending it out of the computing node to all registered subscribers over the publish-subscribe

domain. Similarly, subscribers are responsible for receiving the data from the publish-subscribe domain, format it per the data definition, and present the formatted data to the intended applications (see Figure 5.1).

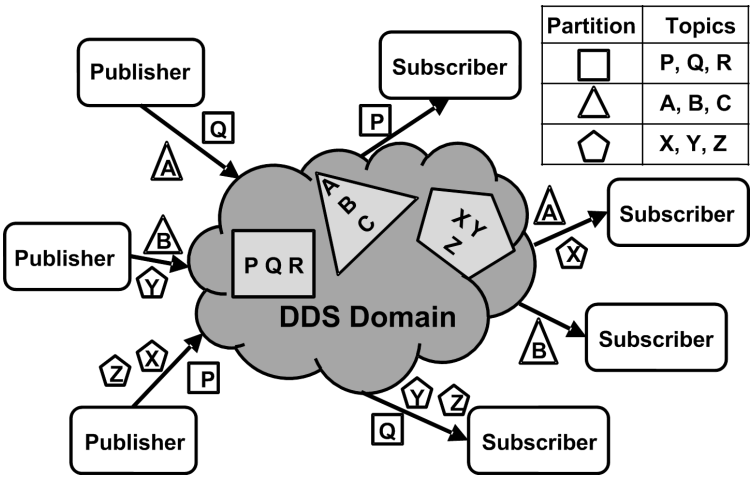


Figure 5.1

Publish Subscribe Architecture.

Publish-subscribe communication is performed through the DDS domain. A DDS domain can have multiple partitions. Each partition can contain multiple topics. Topics are published and subscribed to across the partitions. Partitioning is used to group similar types of topics together. It also provides flexibility to the application for receiving data from a set of data sources [78]. The publish-subscribe mechanism overcomes the typical short comings of the client-server model, where client and servers are coupled together for exchange of messages. In the case of the client-server model, a client should have information about the location of the server for the exchange of messages. However, in the case of

the publish-subscribe mechanism, clients should know only about the type and definition of the data published for the server or other nodes in the domain.

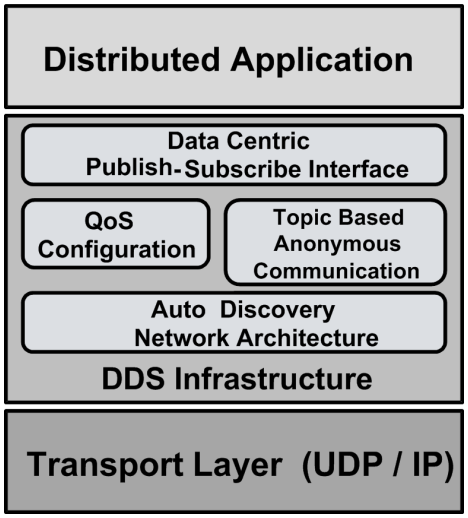


Figure 5.2

Data Distribution Services (DDS) Architecture.

5.2.2 Open Splice DDS

OpenSplice DDS [31] is an open source community edition version of the DDS specifications defined by the Object Management Group [27]. These specifications are primarily used for communication requirements of distributed real-time systems. Distributed systems use DDS as an interface for the "Data-Centric Publish-Subscribe" (DCPS) communication mechanism (see Figure 5.2). Data-Centric communication provides flexibility to specify QoS parameters for the data, depending upon the type, availability, and criticality of the data. These QoS parameters include the rate of publication, rate of subscription, data

validity period, etc. DCPS provides flexibility to the developers for defining the different QoS requirements for the data in order to take control of each message, and developers can concentrate only on the handling of data at each node, instead of on the transfer of data among the nodes. The publishers and the subscribers use the DDS domain to send and receive the data. DDS can be combined with any communication interface for communication among instances of a distributed application hosted in a distributed environment. The DDS domain handles all of the communications among the publishers and the subscribers according to the QoS specifications of the data. In the DDS communication mechanism, each message is associated with a special data type called the *topic*. The subscribers register to one or more topics of their interest. In DDS, a computing node can act simultaneously as a publisher and as a subscriber for different topics (see Figure 5.3).

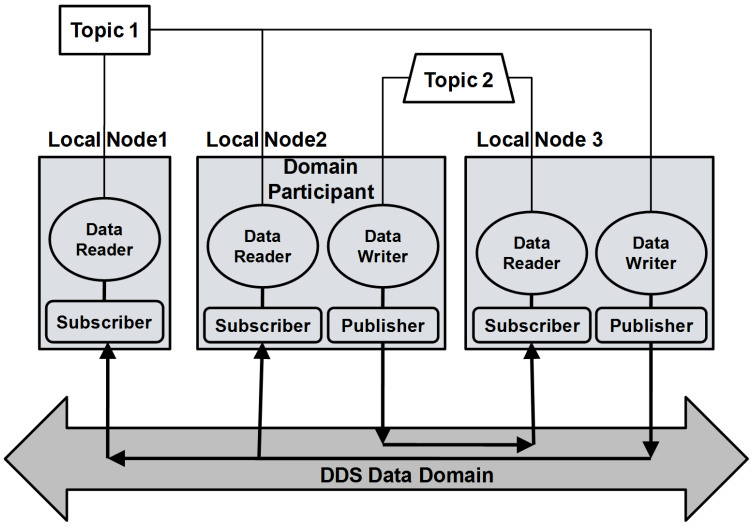


Figure 5.3

Data Distribution Services (DDS) Entities.

The primary benefit of using the DDS framework for the distributed monitoring system is that DDS is based upon the publish-subscribe mechanism that decouples the sender and receiver of the data. There is no single point of bottleneck or of failure in communication. Additionally, the DDS communication mechanism provides scalability for number of computing nodes and supports auto-discovery of the computing nodes in the distributed deployment. Moreover, DDS ensures data delivery with minimum overhead and efficient bandwidth utilization.

5.2.3 ARINC-653

ARINC-653 [45] software specification has been utilized in safety-critical real-time operating systems (RTOS) that are used in avionics systems and recommended for space missions. ARINC-653 specifications present a standard Application Executive (APEX) kernel and its associated services in order to ensure spatial and temporal separation among various applications and monitoring components in integrated modular avionics. ARINC-653 systems (see Figure 5.4) group multiple processes into spatially and temporally separated *partitions*. Multiple partitions (or one) are grouped to form a module (i.e. a processor), and one or more modules form a system. These partitions are allocated in a predetermined chunk of memory. ARINC-653 specifications provide two useful characteristics to the executing partitions: spatial partitioning and temporal partitioning.

Spatial partitioning [79] ensures that an ARINC-653 partition uses an exclusive region from the memory. This memory partitioning ensures that a faulty process from a partition does not corrupt the data structures of other processes executing in other partitions.

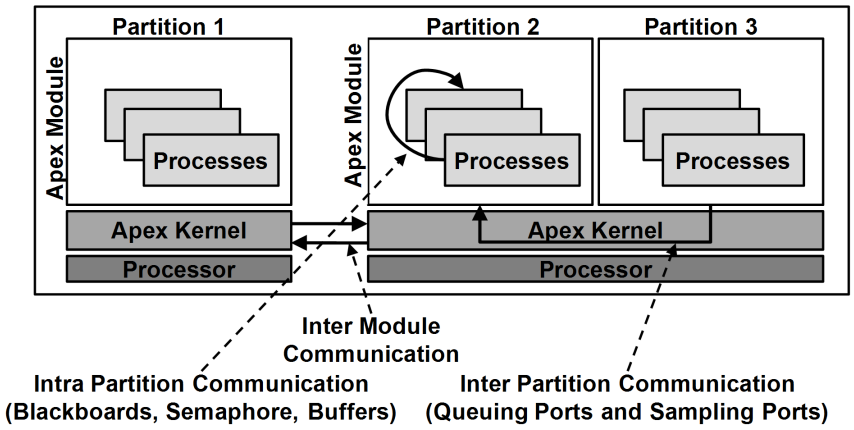


Figure 5.4

ARINC-653 Architecture.

This feature is utilized in avionics systems for separating the components of different criticality [73]. Memory management hardware guarantees memory protection by allowing a process to access only the part of the memory that belongs to the partition, which hosts the same process.

Temporal partitioning [79] ensures that multiple partitions use a common processing resource at different times through a fixed periodic schedule. This fixed periodic schedule is generated or supplied to RTOS in advance for the sharing of resources among the partitions. This deterministic scheduling scheme ensures that each partition gains access to the computational resources within its execution interval according to the scheduling scheme. Additionally, deterministic scheduling guarantees that the partition's execution will be interrupted as soon as the partition's execution interval is finished. The current partition will be moved to the dormant state; the next partition as per the scheduling scheme will be allowed to access the computing resources. All shared hardware resources are managed by

the partitioning OS to guarantee that the resources are freed as soon as the time slice for the partition expires.

ARINC-653 architecture ensures fault-containment through functional separation among applications and monitoring components. In this architecture, partitions and their process can only be created during the system initialization. Processes can not be created dynamically, when the system is in execution. Additionally, users can configure the real-time properties (priority, periodicity, duration, soft/hard deadline, etc.) of the processes and partitions during their creation. These partitions and processes are scheduled and strictly monitored for possible deadline violations. Processes of the same partition share data and communicate using intra-partition services. Intra-partition communication is performed by using buffers to provide a message passing queue and blackboards to read, write, and clear single data storage. Two different partitions communicate by using inter-partition services, which use ports and channels for the sampling and queueing of the messages. The synchronization of processes related to the same partition is performed through semaphores and events [79].

The ARINC-653 Emulation Library [73] (available to download from [22]) provides a UNIX based implementation of ARINC-653 interface specifications. The ARINC-653 Emulation Library is also responsible for providing temporal partitioning among partitions, which are implemented as Linux Processes. This library provides process and time management services as described in the ARINC-653 specifications.

5.2.4 Ruby on Rails

Rails [33] is a web application development framework that uses the Ruby programming language [32]. Rails uses Model View Controller (MVC) architecture for application development. MVC divides the responsibility of managing the web application in three components: *Model:* It contains the data definition and manipulation rules for the data. Model maintains the state of the application by storing data in the database. *Views:* View contains the presentation rules for the data and handles visualization requests made by the application according to the data present in the model. View never handles the data but interacts with users in various ways in order to help users visualizing the data. *Controller:* Controller contains rules for processing the user input, testing, updating, and maintaining the data. A user can change the data by using controller while views will update the visualization of the data to reflect the changes.

The sequence of events while accessing a rails application over a web interface are shown in Figure 5.5. In a rails application, an incoming client request is first sent to a router that finds the location of the application and parses the request to find the corresponding controller (method inside the controller) that will handle the incoming request. The method inside the controller can look into the data of request, can interact with the model if needed, and can invoke other methods according to the nature of the request. Finally, the method sends information to the view, which renders the browser of the client with the result.

In *"RFDMon,"* a web service is developed to display the measurements stored in the database, which are collected from the distributed infrastructure through various monitoring sensors. The schema information of the database is shown in Figure 5.6. This database

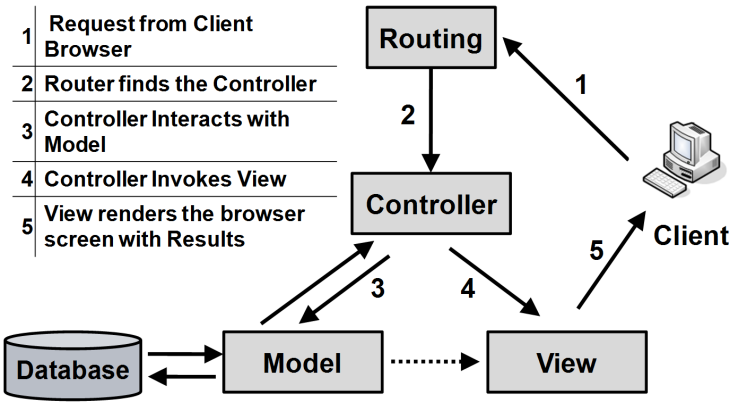


Figure 5.5

Rails and Model View Controller (MVC) Architecture Interaction.

contains information related to clusters, nodes in a cluster, node states, measurements from various sensors on each node, MPI and PBS related information for scientific applications, web application performance statistics, and process accounting logs.

5.3 Other Distributed Monitoring Systems

Various distributed monitoring systems have been developed by industry and research communities in the past. Ganglia [13], Nagios [24], Zenoss [44], and Nimsoft [26] are among the most popular enterprise products used for monitoring distributed systems.

Ganglia [13] is developed on the concept of a hierarchical federation of clusters. In this architecture, multiple nodes are grouped as a cluster, which is attached to a module, and then multiple clusters are again grouped under a monitoring module. Nodes and applications utilize a multi-cast based listen-announce protocol for sending their measurements to all of the other nodes. The primary advantage of Ganglia is the auto-discovery of the nodes, easy portability, manageability, and the aggregation of cluster measurements at each node.

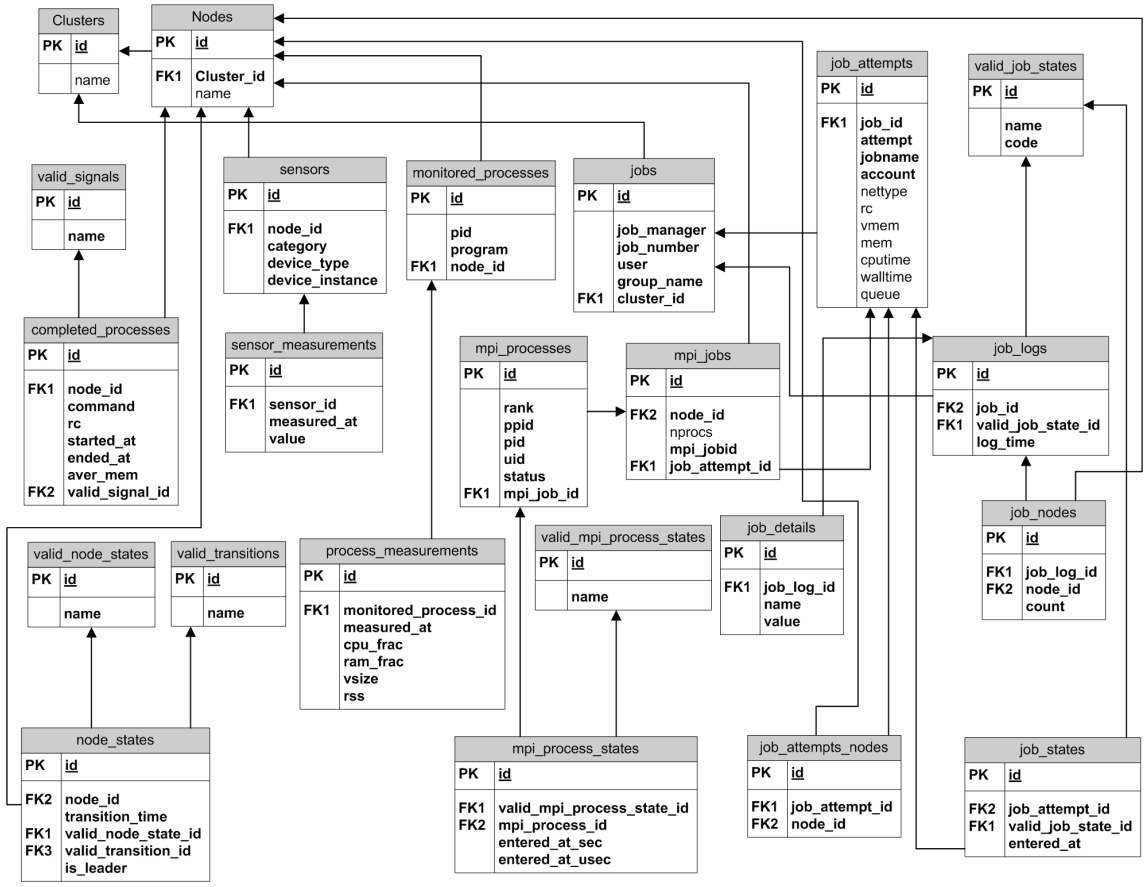


Figure 5.6

Schema of Monitoring Database (PK = Primary Key, FK = Foreign key).

Nagios [24] is developed on a plug-in based agent-server architecture, where agents can report the abnormal events from the computing nodes to the server node (administrators) through email, SMS, or instant messages. Nagios consists of three components- *Plug-in*: These small modules are placed on the computing nodes and configured to monitor a resource. Plug-ins forward the measurements to the *Nagios* server module over SNMP interface. *Scheduler*: This is the administrator component that checks the plug-ins and takes corrective actions if needed. *GUI*: This is a web-based interface that displays the measurements from the system with various buttons, sounds, and graphs.

Zenoss [44] is a model-based monitoring solution that has a comprehensive and flexible approach of monitoring with an extremely detailed GUI interface. It is an agentless monitoring approach, where the central monitoring server collects measurements from each node over the SNMP interface through ssh commands. In Zenoss, the computing nodes can be discovered automatically and monitored according to their types (Xen, VMWare, etc.). This characteristic ensures appropriate and complete monitoring by using pre-defined templates, thresholds, and event rules. *Nimsoft Monitoring Solution* [26](NMS) offers a light-weight, reliable, extensive, and GUI based monitoring of the entire infrastructure. NMS uses a message BUS for the exchange of messages among the applications residing in the infrastructure. These applications (or components) are configured with the help of a software component (HUB) and are attached to the message BUS. The monitoring activity is performed by small probes and the measurements are published to the message BUS by software components (*ROBOTS*) deployed over each managed device. NMS also provides

an *Alarm Server* for alarm monitoring and a *GUI portal* to visualize the comprehensive view of the system.

These distributed monitoring approaches are significantly scalable in the number of nodes, responsive to the changes at the nodes, and comprehensive in the number of parameters. However, these approaches do not support restrictions on the resources consumed by the monitoring framework, fault containment in the monitoring system, and expandability of the monitoring approach for new parameters in the already executing monitoring system. Additionally, these approaches are stand alone and are not easily extendible to associate with other modules that can perform fault diagnosis for the infrastructure at different granularity (application level, system level, and monitoring level). Furthermore, these monitoring approaches work in a client-server or host-agent manner (except NMS) that require the direct coupling of two entities, where one entity has to be aware of the location and identity of the other entity.

A distributed monitoring system "RFDMon" is developed in this dissertation to monitor system resources, hardware health, node availability, scientific application status, and application performance statistics in a comprehensive manner. RFDMon is easily scalable with the number of nodes because it is based upon the data centric publish-subscribe mechanism. Also, in the developed monitoring system, new sensors can be easily added to increase the number of monitored parameters. RFDMon is fault tolerant with respect to faults in the monitoring system due to partial outages. It can self-configure (Start, Stop, and Poll) the sensors and it can be applied in the distributed systems with heterogeneous nodes. The major benefit of using this monitoring system is that the total resource consumption by

the sensors can be limited by applying ARINC-653 scheduling policies. Moreover, due to the spatial isolation features of the ARINC-653 emulation library, the monitoring system will not corrupt the memory area or data structures of applications, which are executing on the node. Additionally, RFDMon has a small computational overhead, which can be further lowered by using ARINC-653 scheduling policies. The developed web service in monitoring system helps in visualizing the various resource utilization, hardware health, and process state on different computing nodes. Therefore, an administrator can easily find the location and possible causes of the fault in the system.

5.4 RFDMon System Architecture

The developed monitoring system "RFDMon" is based upon the data centric publish-subscribe communication mechanism. Modules (or processes) in the monitoring system are separated from each other through by using spatial locality as described in section 5.2. Architecture of "RFDMon" is shown in Figure 5.7. The developed monitoring system has following key concepts and components.

5.4.1 Sensors

Sensors are the primary components of the monitoring system. These are lightweight processes that monitor a device on the computing nodes and read it periodically or aperiodically to get the measurements. These sensors publish the measurements under a topic (described in next subsection) to the DDS domain. The monitoring system contains various types of sensors, which are described in Section 5.5.

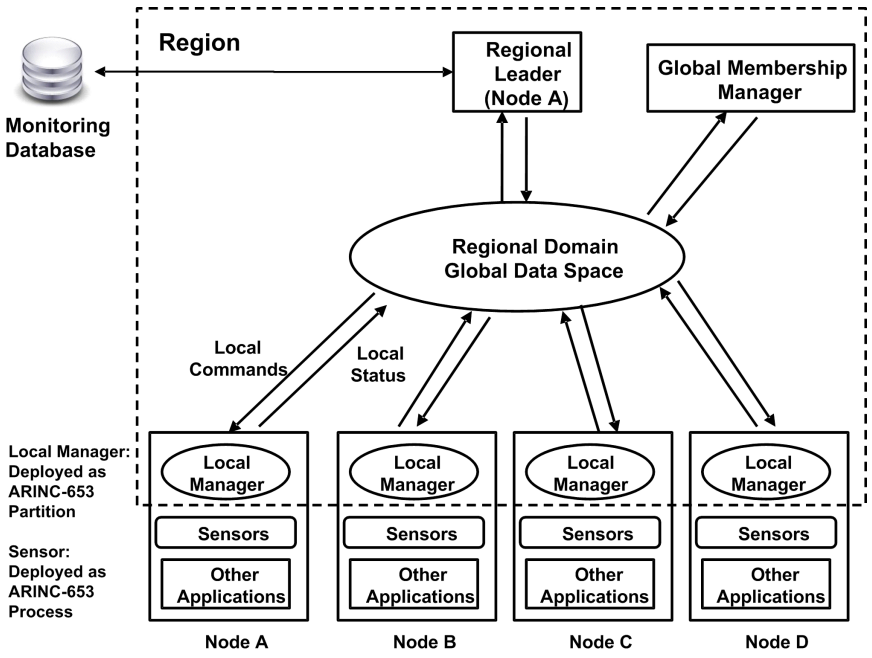


Figure 5.7

Architecture of the "RFDMon" Monitoring System.

5.4.2 Region

The developed monitoring system organizes the nodes in *Regions* (or clusters). Nodes can be homogeneous or heterogeneous. Nodes are combined only logically; these nodes can be located in a single server rack or on a single physical machine (in the case of virtualization). However, physical closeness is recommended to combine the nodes in a single region in order to minimize the unnecessary communication overhead in the network.

5.4.3 Local Manager

The *Local Manager* is a module that is executed as an agent on each computing node of the distributed infrastructure. These agents are executed on each node with the knowledge of its pre-defined region name. However, these agents are not provided with any informa-

tion related to the other nodes or the configuration of the region. The primary responsibility of the local manager is to set up a sensor framework on the node.

5.4.4 Regional Leader

Among multiple local manager nodes that belong to the same region, there is a local manager node which is selected as the *Regional Leader* for collecting sensor measurements from other local managers and updating the monitoring database. The regional leader is also responsible for updating the changes in the state (UP, $FRAMEWORK\_DOWN$, $DOWN$) of various local manager nodes. Each local manager is supplied with pre-defined URLs to the Ruby on Rails web service for database updates. This state or and measurement update is performed over an http interface by using *"libcurl"* library [8]. However, these URLs are used by only the local manager, which is selected as a regional leader. Once a regional leader terminates, a new leader will be selected for that particular region. The selection of the regional leader is completed by "Global Membership Manager" module as described in Section 5.4.7.

5.4.5 Topics

Topics are the primary unit of information exchange in the DDS domain. Details about the type of topic (structure definition) and key values (keylist) to identify the different instances of the topic are described in an interface definition language (IDL) file [28]. CORBA IDL files are used to promote the interoperability among the systems developed in different programming languages (e.g., C, C++, Java, etc.) that use the same interface.

Keys can represent an arbitrary number of fields in the topic. These topics are categorized in the following categories based upon their content.

- *MONITORING_INFO:* System resource and hardware health monitoring sensors publish measurements under the monitoring info topic.
- *HEARTBEAT:* The Heartbeat sensor uses this topic to publish heartbeat of computing node in the DDS domain to notify the monitoring system that the node is still attached to the monitoring system. All nodes, which are listening to the HEARTBEAT topic, can keep track of the existence of other nodes in the DDS domain through listening to these heartbeats.
- *NODE_HEALTH_INFO:* When a regional leader node (described in Section 5.4.4) detects changes in the state ($UP, DOWN, FRAMEWORK\_DOWN$) of other nodes through a change in their heartbeat, it publishes the NODE_HEALTH_INFO topic to notify all of the other nodes about changes in the status of the node.
- *LOCAL_COMMAND:* This topic is used by the regional leader to send the control commands to other local managers in order to start, stop, or poll the monitoring sensors for publishing the measurements.
- *GLOBAL_MEMBERSHIP_INFO:* This topic is used for communication between local managers and the global membership manager (described in Section 5.4.7) for selection of regional leader and for providing information related to the existence of the regional leader to other local managers.
- *PROCESS_ACCOUNTING_INFO:* The process accounting sensor reads process accounting records from the system and publishes these records under this topic.
- *MPI_PROCESS_INFO:* This topic is used to publish the execution state (STARTED, ENDED, KILLED) and MPI or PBS information of scientific applications and their parallel tasks executing on the computing nodes.
- *WEB_APPLICATION_INFO:* This topic is used to publish the performance statistics of the web applications in DDS domain. These performance statistics can include the average response time, heap memory usage, the number of JAVA threads, and the number of pending requests inside the application.

5.4.6 Topic Managers

Topic Managers are classes, which create a subscriber or publisher for a pre-defined topic. This publisher publishes the data received from various sensors under the same topic

name. The subscriber receives data from the DDS domain under the same topic name and delivers it to the underlying application for further processing.

5.4.7 Global Membership Manager

The Global Membership Manager (GMM) module is responsible to maintain the membership of each node for a specific region and for the selection of a regional leader. Once a local manager comes online on a node, it first contacts the GMM module with node's region name by using GLOBAL_MEMBERSHIP_INFO topic to get the information regarding the regional leader. GMM module replies with the name of regional leader (if a leader exists) or assign the new node as regional leader. The GMM module updates the leader information in a file ("REGIONAL_LEADER_MAP.txt") on disk in colon separated format (RegionName:LeaderName). When a local manager sends message to the GMM module that its regional leader is dead, the GMM module selects a new leader for that region and replies to the local manager with the leader name.

This leader re-selection functionality enables the fault tolerant nature in the monitoring system with respect to the regional leader, which ensures a periodic update of the infrastructure monitoring database with measurements, even in case of leader (or node) failure. The leader selection for the region is performed by a single GMM module, which ensures the presence of only one leader in a region. Because the leader selection or re-selection is performed by communication between only two nodes, this process is unaffected by the size of the region. The communication delay of the message exchange in the DDS domain is the only factor that can delay the leader selection process. Additionally, other more so-

phisticated algorithms can be easily plugged into the monitoring system by modifying the GMM module for leader selection.

The GMM module is executed through a wrapper executable *GMM_Monitor* as a child process. The GMM_Monitor keeps track of the execution state of the GMM module and starts a fresh instance of the GMM module if the previous instance terminates due to some error. The new instance of the GMM module receives updated data from "REGIONAL_LEADER_MAP.txt" file. This wrapper executable provides the fault tolerant abilities in the framework with respect to the GMM module.

The UML class diagram in Figure 5.8 shows the relation among all of these modules. The next section will describe the details regarding each sensor, which is executing at each local node.

5.5 Sensor Implementation

The developed monitoring system contains various software sensors to monitor system resources, network resources, node states, the scientific application execution state, and performance statistics of web applications (see Table 5.1). These sensors can be periodic or aperiodic depending upon the implementation and the type of resources. Periodic sensors are implemented for system resources and performance data while aperiodic sensors are used for the MPI process state and other system events that get triggered only on availability of the measurements. These sensors are executed as an ARINC-653 process on top of the ARINC-653 emulator [73]. Sensors are constructed with the following attributes:

- *Name:* Name of the sensor (e.g., UtilizationAggregatecpuScalar).
- *Source Device:* Name of the device to monitor for the measurements (e.g., "/proc/stat").

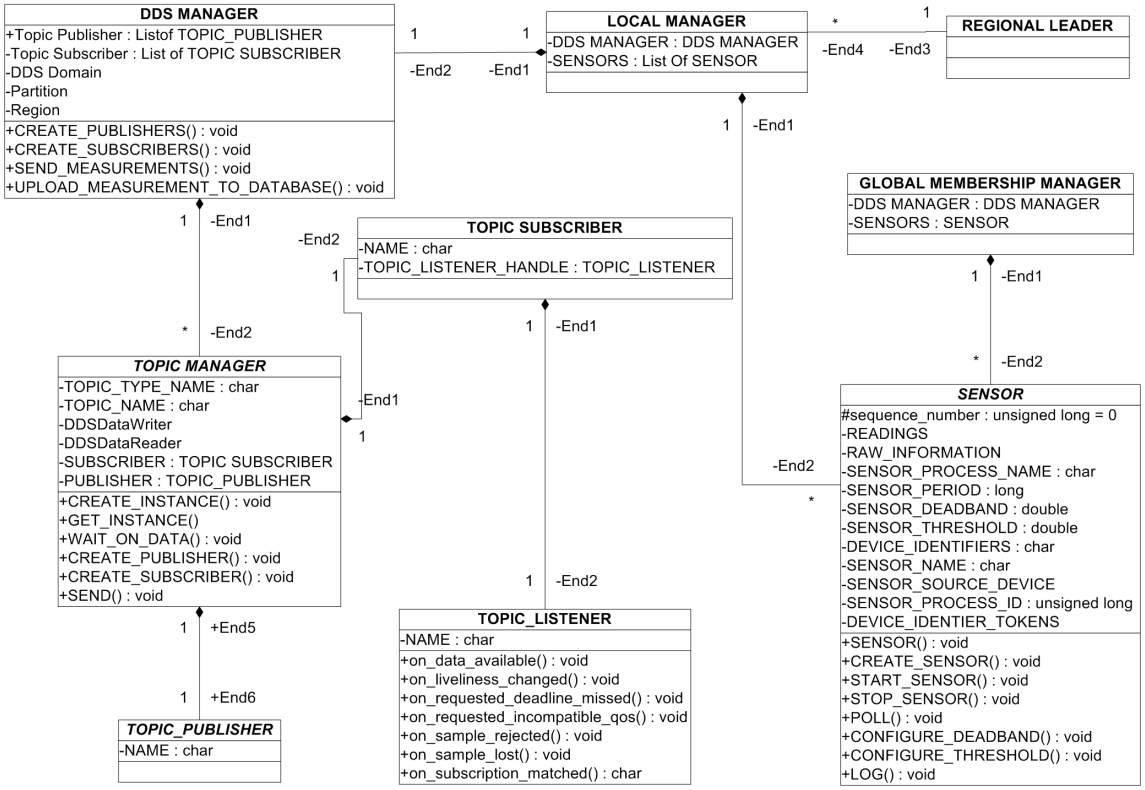


Figure 5.8

Class Diagram of the “RFDMon” Monitoring System.

- *Period:* Periodicity of the sensor (e.g., 10 seconds for periodic sensors and -1 for aperiodic sensors).
- *Deadline:* A sensor has to finish its measurement within a specified deadline. A HARD deadline violations are considered as an error. The HARD deadline violations are handled by the restarting the sensors while SOFT deadline violation are handled through warnings.
- *Priority:* The sensor priority indicates the priority of scheduling the sensor over other processes in the system. In general, normal (base) priority is assigned to the sensor.
- *Dead Band:* The sensor reports the measurements only if the difference between current measurement and the previous recorded measurement becomes greater than the specified sensor dead band. It reduces the number of sensor measurements in the DDS domain if sensor measurement is changing slightly.

Sensors support three types of commands for publishing the measurement: *START, STOP,* and *POLL*. The *START* command starts the already initialized sensor to start publishing the measurements. The *STOP* command is executed to stop the sensor thread from publishing the measurement while the *POLL* command is executed to get the previous measurement from the sensors. Sensors publish the data according to the predefined topic to the DDS domain (e.g., MONITORING_INFO). The life cycle and various functions of a typical sensor ("CPU utilization Sensor") is described in Figure 5.9. Sensors are categorized based upon their functionality as follows.

5.5.1 Resource Utilization Monitoring Sensors

These sensors monitor the utilization of the system resources: CPU, RAM, Disk, Swap, and Network. These sensors are periodic in nature, follow SOFT deadlines, contain normal priority, and monitors system devices (e.g., /proc/stat) in order to collect the measurements. These sensors publish the measurements under the MONITORING_INFO topic.

Table 5.1

List of Monitoring Sensors

Sensor Name	**Period (Seconds)**	**Description**
CPU Utilization	30	Aggregate utilization of all the CPU cores.
Swap Utilization	30	Swap space usage.
Ram Utilization	30	Memory usage.
Hard Disk Utilization	30	Disk usage.
CPU Fan Speed	30	Speed of CPU fan that helps keep the processor cool.
Motherboard Fan Speed	10	Speed of motherboard fan that helps keep the motherboard cool.
CPU Temperature	10	Temperature of the processors.
Motherboard Temperature	10	Temperature of the motherboard.
CPU Voltage	10	Voltage of the processor.
Motherboard Voltage	10	Voltage of the motherboard.
Network Utilization	10	Bandwidth utilization of each network card.
Network Connection	30	Number of TCP connections.
Heartbeat	30	Periodic liveness messages.
Process Accounting	30	Periodic sensor that publishes the commands executed on the system.
MPI Process Info	-1	Aperiodic sensor that reports the change in state of the MPI Processes.
Web Application Info	-1	Aperiodic sensor that reports the performance data of Web Application.

SENSOR LIFE CYCLE (pseudo code)

Define Sensor:

```
CPU_UTILIZATION_SENSOR* CPU_UTILIZATION_SENSOR::INSTANCE=0;
SENSOR_PROCESS_NAME = "CPU_UTILIZATION_SENSOR";
SENSOR_NAME = "UtilizationAggregatecpuScalar";
const std::string CPU_UTILIZATION_SENSOR::DEVICE_IDENTIFIERS = "cpu";
double CPU_UTILIZATION_SENSOR::SENSOR_THRESHOLD = 0;
double CPU_UTILIZATION_SENSOR::SENSOR_DEADBAND = 0.01;
SYSTEM_TIME_TYPE CPU_UTILIZATION_SENSOR::SENSOR_PERIOD = 10; //SECONDS
```

Create Sensor:

```
SENSOR_PROCESS_ID = APEX_HELPER_CREATE_PROCESS (
            SENSOR_PERIOD,   //SYSTEM_TIME_TYPE PERIOD,
            SENSOR_PERIOD,    //SYSTEM_TIME_TYPE TIME_CAPACITY,
            (SYSTEM_ADDRESS_TYPE)SENSE,  //ENTRY_POINT,
            0,   //STACK_SIZE_TYPE STACK_SIZE,
            90,   //PRIORITY_TYPE BASE_PRIORITY,
            SOFT,  //DEADLINE_TYPE DEADLINE,
            SENSOR_PROCESS_NAME,  //std::string PROCESS_NAME,
            RETURN_CODE );   //RETURN_CODE_TYPE *RETURN_CODE
```

Start Sensor:

```
START_SENSOR (RETURN_CODE_TYPE* RETURN_CODE)
{
    START(SENSOR_PROCESS_ID, RETURN_CODE);
}

SENSE()
{
  // Sensor Process Periodically enters into this function to read the device for measurements
  LOG();
)

LOG()
{
   // Log function process the measurements and publish the data as topic in DDS domain
   SEND_MONITORING_LOG(dds_data_packet);
}
```

Stop Sensor:

```
STOP_SENSOR(RETURN_CODE_TYPE* RETURN_CODE)
{
   STOP(SENSOR_PROCESS_ID, RETURN_CODE);
}
```

Figure 5.9

Life Cycle of a CPU Utilization Sensor.

5.5.2 Hardware Health Monitoring Sensors

These sensors monitor the system hardware components for the applied voltage and temperature. E.g, CPU Fan Speed, CPU Temperature, Motherboard Temperature, and Motherboard voltage. These sensors are periodic, follow SOFT deadlines, and contain normal priority. These sensors use Intelligent Platform Management Interface (IPMI) interface [19] to record the measurements. These sensors publish these recorded measurements under the MONITORING_INFO topic.

5.5.3 Node Health Monitoring Sensors

Each local manager executes a Heartbeat sensor that periodically publishes its own node's name to the DDS domain under the topic "HEARTBEAT" to inform other nodes about its existence in the monitoring system.

5.5.4 Scientific Application Health Monitoring Sensor

This sensor monitors the execution state (STARTED, KILLED, ENDED) of scientific applications and their various jobs executing in parallel environment. In this monitoring systems, a wrapper application (*SciAppManager*) is developed, which executes the actual scientific application (e.g., *SciAPP* in Figure 5.10) internally as a child process. "MPIrun command" is issued to execute the *SciAppManager* application from master nodes in the cluster (see Figure 5.10). The SciAppManager writes the execution state information of the scientific application in a POSIX message queue that exists on each node. The scientific application sensor reads messages from that POSIX queue and publishes messages to the DDS domain under MPI_PROCESS_INFO topic.

5.5.5 Web Application Performance Monitoring Sensor

This sensor works similarly to the scientific application health monitoring sensor as described in Figure 5.10. This sensor keeps track of performance statistics of the web application through the web server performance logs written into a POSIX message queue (different queue from *SciAppManager*). This sensor reads messages from the message queue and publishes the message to the DDS domain under the WEB_APPLICATION_INFO topic. In the developed monitoring system, a web application logs its performance data in a POSIX message queue that exists on each node. A generic structure of performance logs is defined in the sensor framework that includes average response time, heap memory usage, number of JAVA threads, and pending requests inside the web server.

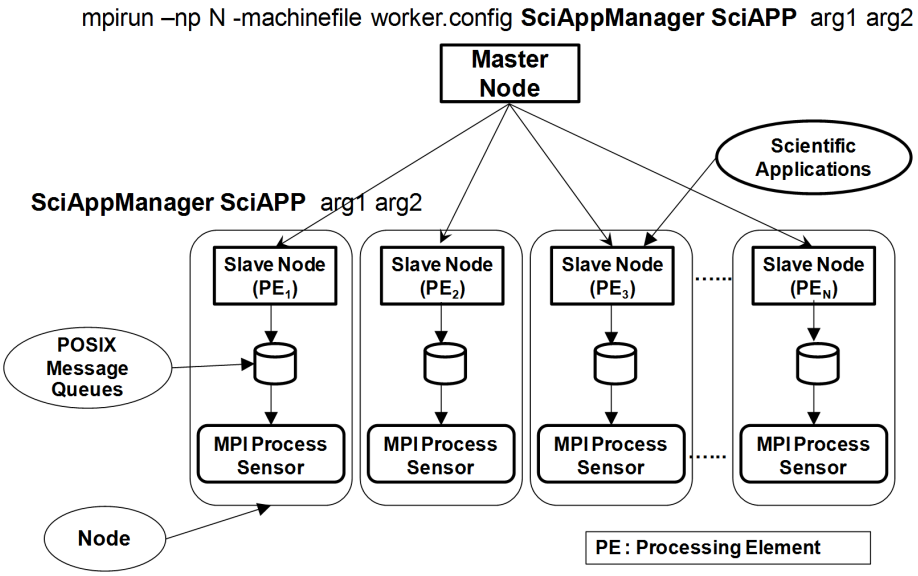


Figure 5.10

Architecture of the Scientific Application Health Monitoring Sensor.

5.6 Experiments Related to Monitoring System

A set of experiments have been performed to show the system resource utilization and fault-adaptive properties of the developed monitoring system. During these experiments, the monitoring system is deployed in a Linux environment ($2.6.18 - 274.7.1.el5xen$) that consists of five nodes ($ddshost1$, $ddsnode1$, $ddsnode2$, $ddsnode3$, and $ddsnode4$). The ruby on rails based web service and the MYSQL database are hosted on the $ddshost1$ node. These experiments have been performed to measure the computational overhead of executing the monitoring system and to demonstrate the fault-tolerant and self-configuring properties of the monitoring system in case of failures in the monitoring system itself.

In one of these experiments, monitoring system were started on all of the nodes ($ddshost1$, and $ddsnode1...4$) one by one with a random time interval. Once all the nodes started executing the monitoring system, the local manager on a few nodes were killed through "KILL" system call. During this experiment, the CPU and RAM consumption by the local manager at each node is monitored through "TOP" system command. Results from the experiment are shown in Figure 5.11, Figure 5.12, and Figure 5.13.

Figure 5.11 shows the CPU and RAM utilization at each node during the experiment. It is evident from Figure 5.11 that CPU utilization is mostly in the range of 0 to 1 percent, with occasional spikes. However, even in the case of spikes, CPU utilization is under ten percent. Similarly, RAM utilization by the monitoring framework is less than 2%. These results clearly indicate that overall resource overhead of the developed monitoring approach "RFDMon" is extremely low. As mentioned earlier, it is possible to limit even

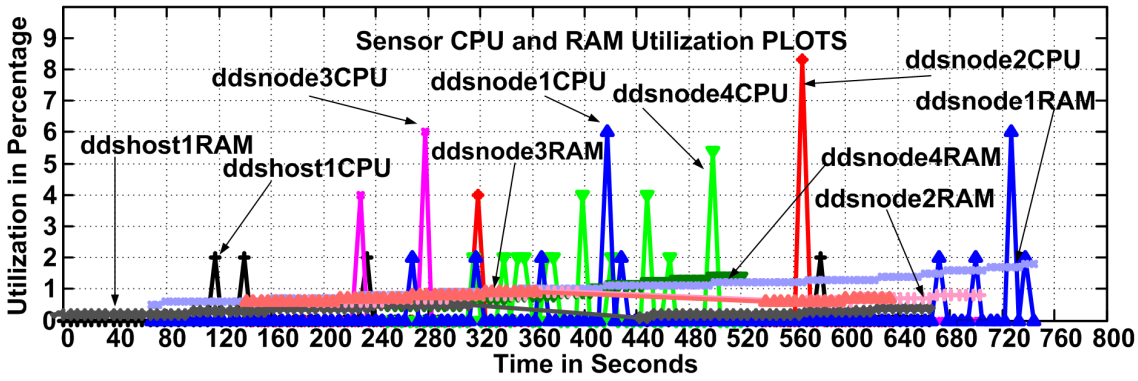


Figure 5.11

CPU and RAM Utilization at various Nodes.

this small resource usage by using the temporal partitioning practices from ARINC-653 specifications to allocate CPU resources.

The transition of various nodes between states UP and FRAMEWORK_DOWN is shown in Figure 5.12. According to Figure 5.12, $ddshost1$ was started first, then followed by $ddsnode1$, $ddsnode2$, $ddsnode3$, and $ddsnode4$. $ddshost1$ was selected as the regional leader in the beginning. Approximately, at time sample 310, the local manager of the host $ddshost1$ was killed; therefore, its state has been updated to FRAMEWORK_DOWN. Similarly, the states of $ddsnode2$ and $ddsnode3$ were also updated to FRAMEWORK_DOWN once their local managers were killed on time sample 390 and 410, respectively. The local manager at $ddshost1$ was again started at time sample 440; therefore its state was updated to UP at the same time. Figure 5.12 also represents the nodes, which were regional leaders during the experiment. According to Figure 5.12, initially $ddshost1$ was the leader of the region, while as soon as the local manager at $ddshost1$ was killed at time sample 310 (see Figure 5.12), $ddsnode4$ was elected as the new leader of the region according to the

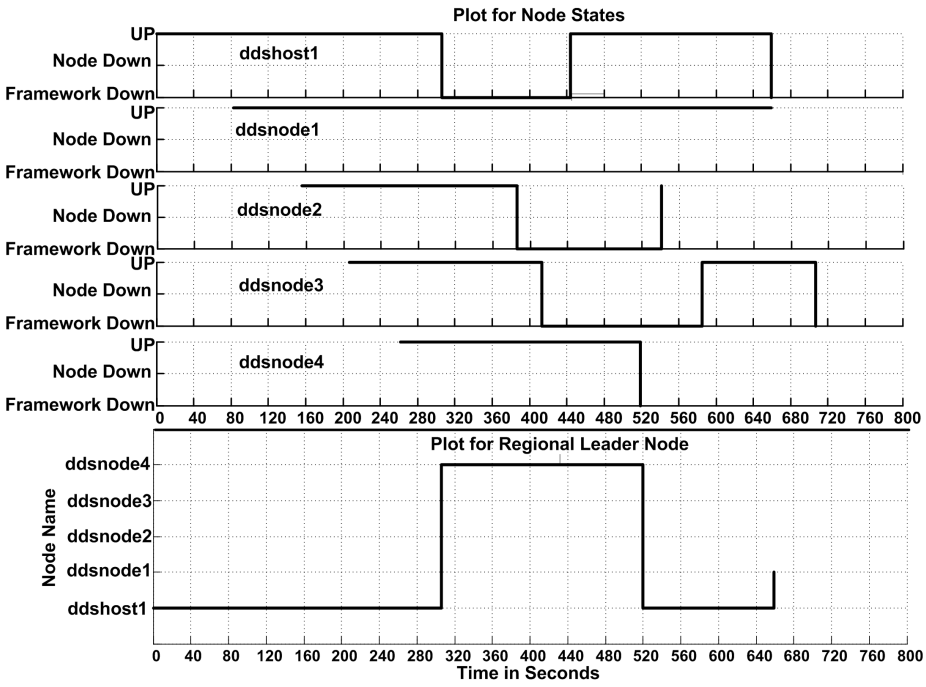


Figure 5.12

State Transition of the Nodes and Leaders of the Distributed Monitoring System.

procedure specified in Section 5.4.7. Similarly, when the local manager of the $ddsnode4$ was killed at time sample 520 (see Figure 5.12), $ddshost1$ was again elected as the leader of the region. From Figure 5.12, it is clearly evident that as soon as there was a fault in the monitoring system related to the regional leader, a new regional leader was elected instantly without any further delay. This specific feature of the monitoring system exhibits that it is robust with respect to failures of the regional leader and that it can adapt to the faults in the monitoring system instantly with minimum delay.

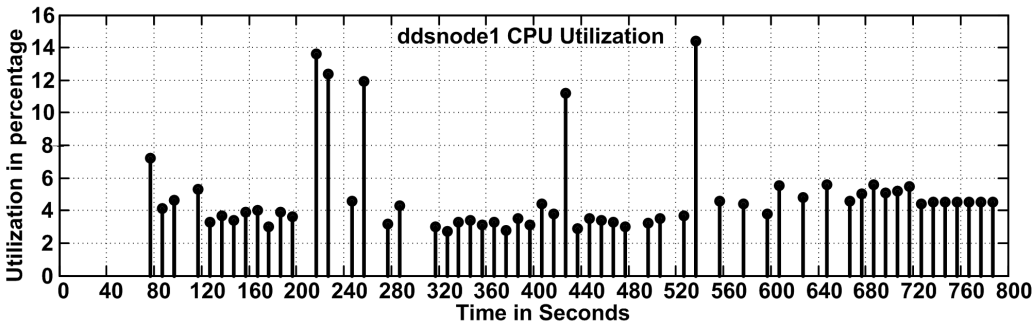


Figure 5.13

CPU Utilization at node ddsnode1 during the Experiment.

The sensor framework at $ddsnode1$ was allowed to execute during the complete experiment (see Figure 5.12), and no fault was introduced in this node. The primary purpose of executing this node continuously was to observe the impact of introducing faults in the framework over monitoring capabilities of the monitoring system. In the most ideal scenario, the entire set of measurements related to $ddsnode1$ should be reported to the centralized database without any interruption even in the presence of faults (leader re-election

and nodes going out of the monitoring system). Figure 5.13 shows that the CPU utilization of $ddsnode1$ from the monitoring database as reported by regional leader through CPU monitoring sensor from $ddsnode1$. According to Figure 5.13, monitoring data from $ddsnode1$ was collected successfully during the entire experiment. Even in the case of Regional Leader re-election at time samples 310 and 520 (see Figure 5.12), only one or two (max) measurement samples are missing from the database (see Figure 5.13. Henceforth, it is evident that there is a minimal impact of faults in the monitoring system itself over its monitoring functionality.

5.7 Summary

In this chapter, the design aspects of "RFDMon" and basic concepts of OpenSplice DDS, ARINC-653, and Ruby on Rails were described. Additionally, it was shown that "RFDMon" can efficiently and accurately monitor the system resource consumption, system health, application performance, and scientific application execution state in a comprehensive manner with minimum computational overhead. Furthermore, self-configuring and fault-tolerating properties of "RFDMon" were demonstrated through experiments.

CHAPTER 6

A DISTRIBUTED CONTROL APPROACH FOR PERFORMANCE MANAGEMENT OF A WEB SERVICE DEPLOYMENT

In this chapter, a distributed control-based performance management approach is presented that can manage a general class of web services deployed in distributed computing environments, where performance of the web service can be tuned by changing a finite set of control inputs. This approach is developed by using interaction balance principles that have been already applied to various performance management problems in the large scale engineering systems. In this chapter, the proposed approach is applied on a web service hosted in a distributed environment for power and response time management.

6.1 Related Work

Control-based methods have recently emerged as a promising way to automate certain system management tasks encountered in distributed computing systems. Algorithms have been developed for optimal control of large scale systems by decomposing the large systems into a number of interconnected subsystems. Thus, the system wide optimization problem is also divided into a number of interconnected subsystem optimization problems. These subsystems coordinate with each other through a coordinator by using interaction inputs and achieving the system wide performance objectives. The coordinator receives

the solution $Sol_i(k)$ ($\forall i \in N$) of the subsystem level problems and updates the interaction inputs $\beta_i(k)$ ($\forall i \in N$) as shown in Figure 6.1. The coordinator modifies the activities at the subsystems to find the optimal solution for the entire system. The definition and relevant parameters of a subsystem are shown in Figure 6.2. These "decomposition and coordination" strategies are primarily implemented in two ways: Interaction Balance (Goal Coordination) and Interaction Prediction (Model Coordination) [143] (shown in Figure 6.1). Both of these approaches have been applied successfully to a number of large scale systems, where subsystems are "coupled with each other in terms of both the subsystem dynamics and the system wide performance objectives". In the *interaction balance* method, the coordinator modifies the objective functions of the subsystems using interaction inputs $\beta_i(k)$ iteratively until the difference between actual interface inputs (interactions) and demanded by the subsystems become zero or small tolerable values. In the case of *model coordination*, the coordinator predicts the value of interface inputs $\hat{Z}_i(k)$ among subsystems, compares them with actual values $Z_i(k)$, and modifies the values of predicted interface inputs $\hat{Z}_i(k)$ again until the error in prediction of the interface inputs reaches zero or small tolerable values.

6.1.1 Advanced Large Scale Control Algorithms and Their Applications

In past, interaction balance and model coordination approaches have been utilized extensively to develop distributed control algorithms for variety of linear and non linear control system problems with applications in electrical systems, chemical plants, and manufacturing industry [143]. Designing a specific type of distributed control approach depends

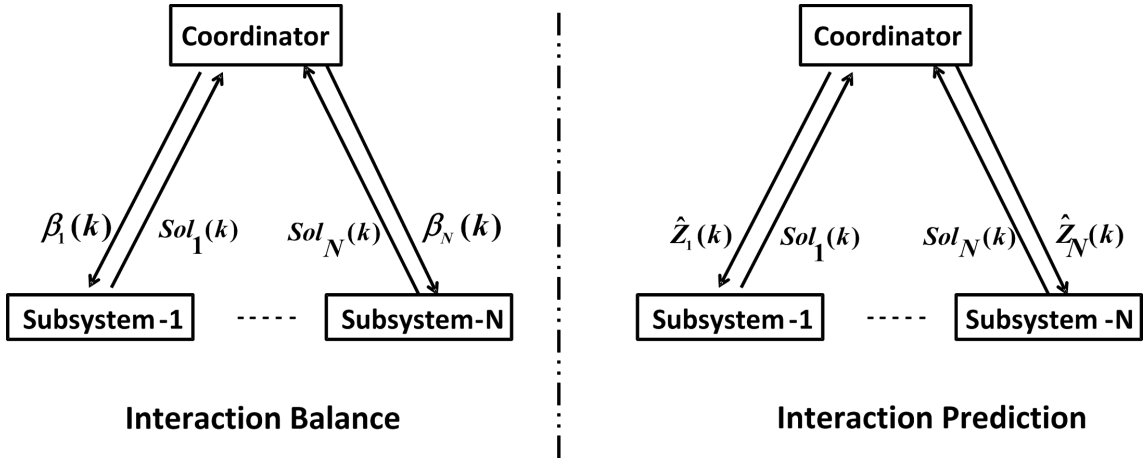
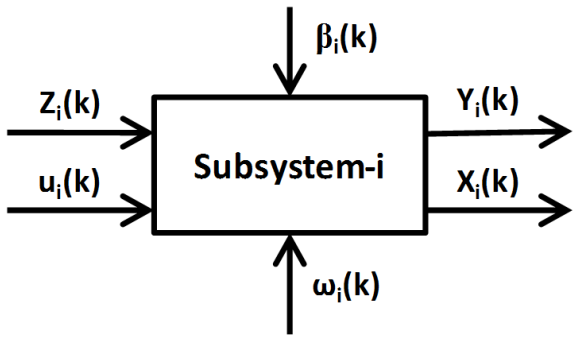


Figure 6.1

Interaction Balance and Interaction Prediction Approach.



$\omega_i(k)$	Environment Input to both global system and subsystem. It can not be controlled or influenced.
$u_i(k)$	Control Input, which can be controlled or modified.
$Z_i(k)$	Interface inputs supplied from other subsystems to i-th subsystem.
$Y_i(k)$	Output of the i-th subsystem
$X_i(k)$	Output or state of i-th subsystem, which act as input for other subsystems.
$\beta_i(k)$	Interaction Input provided from the coordinator.

Figure 6.2

The i-th Subsystem and its Parameters.

upon the type of interaction among the subsystems, the system dynamics, the nature of exchanged information among the subsystems, and the control techniques used inside the subsystem [99]. An asynchronous and parallel version of these control algorithms are presented in [50] for river pollution control and gas absorber tower problems by using interaction balance and interaction prediction approaches. These modified versions of the large scale control algorithms show substantial savings in computation time in the given conditions and the possible methods to reduce the communication delay. Another approach of faster convergence in interaction error vector are presented in [131, 132] by using the gradient of the subsystem state, the control input, and the interaction vector to calculate the Lagrange coefficients.

A hierarchical optimization approach by using neural network is applied to a nonlinear discrete large scale power control systems in [84], where the proposed neural network is decomposed into coordinator and subsystem level sub-networks. Also, the subsystem level system model equations and the operating constraints are embedded into the subsystem level neural network. This approach shows significant improvement in computational overhead compared to the traditional interaction balance approaches developed earlier. In a similar approach, a reinforcement learning (RL) based implementation of goal coordination algorithm is developed in [133] for intelligent coordination among a set of nonlinear subsystems. In this approach, a neuro-fuzzy based RL approach is developed for supervisory coordination at higher level of large scale control systems deployed in a hierarchical manner. The subsystem level controllers are also developed using neural network and

their control problem is solved using neuro-regulators. This approach shows a significant reduction in the number of interactions between the coordinator and the subsystems.

A multi agent planning system implementation of goal coordination approach is developed in [77] to solve a system wide complex optimization problem while each of these autonomous subsystems have their own set of goals. This approach extends goal coordination with resource coordination among the autonomous subsystems through a resource coordinator. The resource sharing is performed only when the utility (cost benefit analysis) of acquiring the resource by an agent is positive. This approach demonstrates that the satisfaction level of global objectives can be increased significantly if goal coordination approach is coupled with resource coordination approaches. An application of distributed control algorithm with receding horizon is described in [75], where subsystem are coupled together in state vectors for multi-vehicle formation. This approach demonstrates the requirement of frequent and quick updates of the receding horizon updates and compatibility constraints to reduce the error in assumptions made at a vehicle about location of other neighboring vehicles in the formation. Another similar implementation of decentralized receding horizon control is presented in [101] for coordination among multiple unmanned autonomous vehicles (UAVs) for making a flight formation and avoid collision.

6.1.2 Large Scale Control Management in Web Service Environment

Researchers from academia and industry have recently addressed the coordination issues among computing nodes of distributed web service deployment. These computing nodes are coupled in only objective functions and constraints. There is no interaction

among these subsystems for their individual system dynamics (state). A fully decentralized and cooperative control approach is developed in [149] for managing power consumption and SLAs of a distributed web service deployment. In this approach, the infrastructure level optimization problem is decomposed into a set of low subsystem level optimization problems. Each of the subsystem level controllers solve their subsystem level control problem by using a model-based control approach to achieve subsystem wide goals. A distributed model predictive control based approach is developed in [148], where subsystems are decoupled in system dynamics but coupled with linear constraints, additive disturbances, and infrastructure level cost functions.

A hierarchical control approach is presented in [93] to solve the performance management problem of a distributed web service in a hierarchical deployment. In this framework, a limited look-ahead based controller is applied at each level of the hierarchy to manage the interaction among lower level controllers by forecasting the operating environment parameters. This approach aggregates the control behavior of all of the lower level controllers (till the leaf subsystem in the hierarchy) for making the control decision at a higher level controller. Another hierarchical control approach is presented in [103] for utilizing the regression tree and neural network based approximation technique to generate performance models of a non linear application. This approach further utilizes these approximation methods for dynamically learning the controller behaviour and making optimal resource allocation decisions at multiple levels of the hierarchy with appropriate control inputs. An approach for developing self-managing systems is presented in [155], where multiple decentralized controllers form a dynamic overlay network for communication. These

controllers share their local information with their neighbors to approximate the overall system level state. These information are further used to optimize a CPU intensive application performance in a grid computing environment, which consists of a large number of virtual machines.

All of the previous research efforts address the optimal resource allocation problem of web service deployment in case of either low number of subsystems or a few levels in the hierarchy. Approximation efforts for modeling the lower level controller behavior becomes a major bottleneck in the case of increasing number of hierarchical levels. Similarly, communication overhead among the subsystems increases drastically with the increase in the number of subsystems. As a solution to these problems, a novel distributed control-based performance management approach is developed in this dissertation by utilizing interaction balance principles, which has already been applied for management of large scale electrical and manufacturing systems. The developed distributed control approach neither requires the approximate behavior modeling of lower level controllers at a higher level nor requires communication among the subsystems for generating the system wide view. Additionally, each subsystem level controller gives priority to both the local and global system objectives while computing the appropriate control inputs.

In this chapter, a distributed control structure is developed by utilizing the interaction balance (or goal coordination) based management approach for managing the web service systems hosted in a distributed environment, where each web service node is *decoupled* from other nodes in system dynamics, while *coupled* in terms of the overall infrastructure wide operating cost functions. This distributed control approach derives an optimal de-

ployment configuration of subsystems that maximizes the infrastructure profitability while keeping the system operation as per SLA guidelines.

6.2 A Distributed Web Service Deployment

The scope of the optimization problem considered here is a typical web service deployed over a cluster of N computing nodes, referred to as subsystems from now onwards. These N subsystems host instances of the web service for computational efficiency, design simplifications, ease of maintenance, and redundancy purposes. This large scale web service system dynamics can be presented as the following state space dynamics,

$$q(k+1) = \Phi(q(k), u(k), \omega(k)) \tag{6.1}$$

where $q(k)$ represents the queue size of the cluster, $u(k)$ represents the set of control inputs to the cluster, and $\omega(k)$ represents the environment input to the cluster at time sample k. The system model (state update function) Φ captures the relationship between the observed system queue state $q(k)$, the control inputs $u(k)$ that adjust system parameters, and the environment input $\omega(k)$, which can not be controlled.

Now, we can consider the overall system state of the cluster as a composition of N subsystems. For example, $q(k)$ = [$q_1(k)$, $q_2(k)$, ..., $q_i(k)$, ..., $q_N(k)$], $u(k)$ = [$u_1(k)$, $u_2(k)$, ..., $u_i(k)$, ..., $u_N(k)$], and $\omega(k)$ = [$\omega_1(k)$, $\omega_2(k)$, ..., $\omega_i(k)$, ...,$\omega_N(k)$]. Here, $q_i(k) \in X_i \subseteq \mathbb{R}$ is the system state at time sample k, the set of user controlled system inputs is $u_i(k) \in U_i \subseteq \mathbb{R}$, and $\omega_i(k) \in \Omega_i \subseteq \mathbb{R}$ is the environment input at time k. $\omega_i(k)$ represents a fraction $\alpha_i(k)$ of the workload $\omega(k)$ incident at subsystem i during the

time sample k, where $\sum_{i=1}^{N} \alpha_i(k) = 1$. X_i represents the set of feasible system states in the subsystem i, U_i represents the set of permissible control input for the node i, and Ω_i denotes the feasible values of environment inputs. Since the current value of environment input $\omega(k)$ cannot be measured until the next sampling instant $k+1$, the system dynamics can only be captured by using a system model with estimated values of environment input $\hat{\omega}(k)$ as

$$\hat{q}_i(k+1) = \Phi_i(q_i(k), u_i(k), \hat{\omega}(k), \alpha_i(k)) \tag{6.2}$$

Equation 6.2 for the subsystem i can be further described for the next queue size and corresponding response time

$$\hat{q}_i(k+1) = \left[q_i(k) + \hat{\omega}(k)\alpha_i(k) - \frac{u_i(k)}{u_i^m \, \hat{c}_i(k)} T \right]^+ \tag{6.3}$$

and

$$\hat{r}_i(k+1) = (1 + \hat{q}_i(k+1)) \, \frac{\hat{c}_i(k) \, u_i^m}{u_i(k+1)} \tag{6.4}$$

where $[a]^+ = \max(0, a)$, $q_i(k)$ is the queue level of the subsystem i at time sample k, $\hat{\omega}_i(k)$ is the expected arrival rate of http requests, $\hat{q}_i(k+1)$ is the expected queue size of the subsystem, $\hat{r}_i(k+1)$ is expected response time of the subsystem, u_i^m is the maximum supported CPU core frequency in the subsystem i, and $\hat{c}_i(k)$ is the predicted average service

time required per request at the maximum CPU core frequency in subsystem i. T is the sample time for the controller.

Next, the operating cost $J(k)$ of the web service cluster at time sample k can be represented as summation of the SLA violation penalty and energy cost of appropriate control inputs at each subsystem while in operation. It can be further represented as

$$J(k) = \sum_{i=1}^{N} J_i(k) = \sum_{i=1}^{N} \sum_{k=1}^{H} \|q_i(k+1) - q_s\|_{A_i} + \|r_i(k+1) - r_s\|_{B_i} + \left\| \frac{u_i(k)}{u_i^m} \right\|_{C_i} \tag{6.5}$$

where A_i, B_i, and C_i are positive weights for the subsystem i. q_s represents the recommend queue size of the subsystems and r_s represents the recommended response time according to SLA guidelines. Here, $k = 1, 2, .., H$ represents the sampling instants in the trajectory of the system operation. According to this quadratic cost function, subsystems are also penalized for the number of the requests remaining in the queue $q_i(k)$. Therefore, q_s is generally set to 0 for the complete depletion of the queue. Additionally, the power consumption of a computing system can be considered as a function of its control input (CPU core frequency in this case) as $E_i(k) = \left\| \frac{u(k)}{u_i^m} \right\|_{C_i}$.

6.3 Distributed Model Predictive Control Problem

The overall optimization problem for the web service deployment over N subsystems can be described as minimizing the overall cost function J described in Equation 6.5, while satisfying the system state and control input constraints of the subsystems.

According to Equation 6.5, the overall cost J is the sum of the operating cost J_i at each subsystem. However, the operating cost J_i at the subsystem i is indirectly coupled with other subsystems j ($\forall j \in N,\ j \neq i$) through the workload fraction $\alpha_i(k)$ (where $\omega_i(k) = \alpha_i(k)\,\omega(k)$) as presented in Equation 6.3. Therefore, changes in the value of $\alpha_i(k)$ at the subsystem i not only impacts the cost function $J_i(k)$ at subsystem i, but also the cost function at all other subsystems $J_j(k)$ ($\forall j \in N,\ j \neq i$). For example, a higher value of $\alpha_i(k)$ at subsystem i may result in a higher value of $q_i(k)$ and $r_i(k)$ according to system dynamics in Equation 6.3 and 6.4, which in turn results in increased $J_i(k)$. However, an increased value of $\alpha_i(k)$ at subsystem i results in lower values of $\alpha_j(k)$ at subsystem j ($\forall j \in N, j \neq i$), which decreases the operating cost $J_j(k)$ at subsystem j. As a result, variation in $\alpha_i(k)$ ($\forall i \in N$) can impact the overall cost function $J(k)$ positively or negatively.

In a typical distributed deployment of the web services, incoming http requests enter into the deployment through a common shared buffer (or queue) that exists at a "Dispatcher" module of the deployment (see Figure 6.3). This dispatcher module forwards the incoming requests to the subsystems, which host instances of the web service. Therefore, incoming http requests are first queued at the dispatcher in the *global queue* (Q) and then at the subsystem level in the *local queue* (q_i). Now, the requests queued in the global queue can be processed by any subsystem that has system resources available, while requests queued inside the local queue at the subsystem i ($\forall i \in N$) can only be processed by the subsystem i. So, these subsystems should be discouraged from increasing their local queue. This can be done by applying a higher penalty on local queue size as compared to

the global queue size in the cost function. According to Equation 6.3, a higher value of the work load fraction $\alpha_i(k)$ received at the subsystem i may increase the local queue size at the subsystem i. However, a higher value of $\alpha_i(k)$ ($\forall i \in N$) results in the reduced global queue at the "Coordinator".

Hence, the overall optimization problem can be again described as "finding an optimal set of control input u and workload fraction α for each subsystem i ($[u_i(k)\, \alpha_i(k)]\,, \forall i \in N$) to minimize the overall cost function $J(k)$, while satisfying the operational constraints of the subsystems and SLAs of the web service."

6.4 A Distributed Control Approach using Interaction Balance Principle

In this approach, a large scale web service deployment is considered, which consists of a cluster of N subsystems as shown in Figure 6.3. The incoming http requests from different clients arrive at a common entry point of the deployment described as a "Coordinator," which forwards these http requests to the subsystems through the "Dispatcher" module. According to Figure 6.3, "Coordinator" maintains a common global queue (Q) for the incoming http requests. This global queue information is available to all of the subsystems. During each time step k, the coordinator assigns a fraction $\alpha_i(k)$ of requests from the total requests ($Q(k) + \omega(k)$) to the subsystem i ($\forall i \in N$), where, $0 \leq \alpha_i(k) \leq 1$ and $\sum_{i=1}^{N} \alpha_i(k) = 1$. This fraction $\alpha_i(k)$ value is supplied from the subsystems to the coordinator, and it is dynamically updated after every time step based upon the requests made by each subsystem. Additionally, no http requests will be assigned to a failed subsystem.

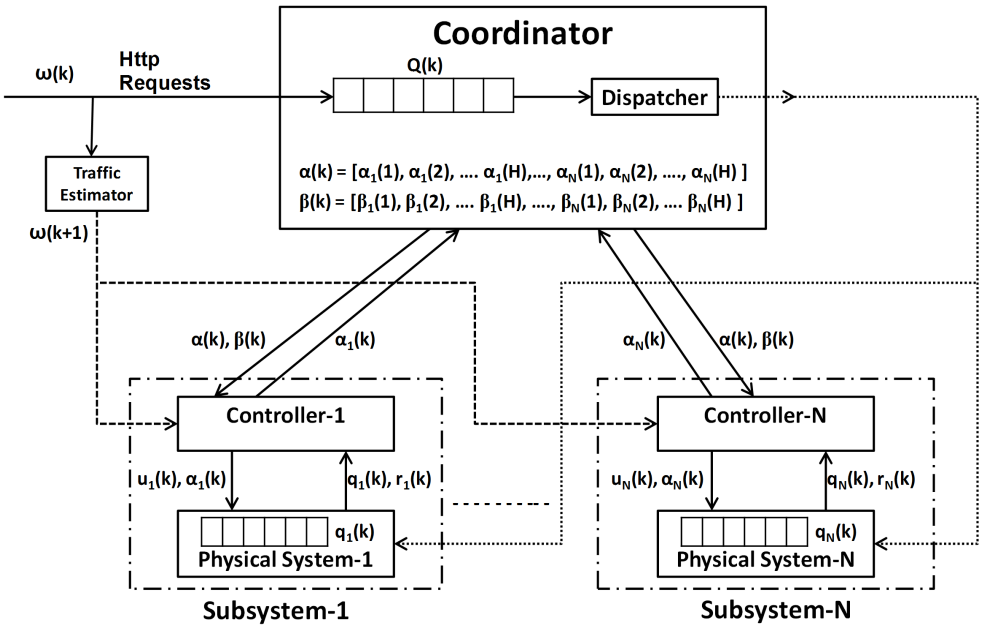


Figure 6.3

The proposed Distributed-Control Approach

The schematic structure of the distributed controllers executing at each subsystem is shown in Figure 6.5. An optimal control algorithm, which is based on the similar one developed in Chapter 4 (Section 4.2.5), is executed inside each subsystem. However, this control algorithm also considers the load fraction $\alpha_i(k)$ while determining the optimal values of the control inputs $u_i(k)$. The state update dynamics at the subsystem i can be described in following equation.

$$\hat{q}_i(k+1) = \Phi_i\left(Q(k), q_i(k), u_i(k), \hat{\omega}(k), \alpha_i(k)\right) \tag{6.6}$$

Equations for the next queue size, response time, and the global queue size as observed at subsystem i can be further described as

$$\hat{q}_i(k+1) = \left[q_i(k) + (Q(k) + \hat{\omega}(k))\, \alpha_i(k) - \frac{u_i(k)}{u_i^m\, \hat{c}_i(k)} T \right]^+ \tag{6.7}$$

$$\hat{r}_i(k+1) = (1 + \hat{q}_i(k+1))\, \frac{\hat{c}_i(k)\, u_i^m}{u_i(k+1)} \tag{6.8}$$

and

$$\hat{Q}(k+1) = \left[(Q(k) + \hat{\omega}(k))(1 - \alpha_i(k) - \sum_{j=1, j \neq i}^{N} \alpha_j^*(k)) \right]^+ \tag{6.9}$$

where $q_i(k)$ is the local queue size of the system i, $Q(k)$ is the global queue size at the coordinator and $\hat{\omega}(k)$ is the expected arrival rate of http requests at the cluster level. $\hat{q}_i(k+1)$ is the expected queue size of the subsystem i, $\hat{r}_i(k+1)$ is the expected response time of the system, and $u_i(k) \in U_i$ is the CPU core frequency at time sample k (U_i is the finite set of all possible CPU core frequencies that the subsystem can take). u_i^m is the maximum supported frequency in the subsystem i and $\hat{c}_i(k)$ is the predicted average service time (work factor in units of time) required per request at the maximum CPU core frequency. T is the sample time for the controller. $\alpha_i(k)$ is the fraction of total workload $\hat{\omega}(k)$ received at the subsystem i. The values of workload processed at subsystem j during time sample $k-1$ ($j \neq i$, $\alpha_j(k-1)$) are received from the coordinator and used as $\alpha_j^*(k)$ ($j \neq i$) in Equation 6.9 at subsystem i.

Next, the operating cost of this cluster of N subsystems hosting web service instances at time sample k for a look ahead horizon of H steps, can be represented as the summation

of SLA violation penalty and energy cost of appropriate control inputs at each subsystem and global queue size (Q) at the coordinator level. Therefore, operating cost can be further represented as

$$J(k) = \sum_{k=1}^{H} \left[\|Q(k+1) - Q_s\|_G + \sum_{i=1}^{N} \|q_i(k+1) - q_s\|_{A_i} + \|r_i(k+1) - r_s\|_{B_i} + \left\| \frac{u_i(k)}{u_i^m} \right\|_{C_i} \right] \tag{6.10}$$

where A_i, B_i, and C_i are positive weights for the subsystem i, while G is positive weight for the global queue of the cluster.

6.4.1 Decomposition of the Overall Problem into N Sub Problems

The web service deployment considered here consists of N subsystems, which are coupled only through the processed workload fractions (according to Equations 6.6 and 6.9) and cluster level cost function $J(k)$. The overall cost function $J(k)$ from Equation 6.10 can be further divided as sum of cost functions related to each subsystem $J_i(k)$ ($\forall i \in N$) according to following equation.

$$J(k) = \sum_{i=1}^{N} J_i(k) = \sum_{k=1}^{H} \sum_{i=1}^{N} \left[\|Q(k+1) - Q_s\|_{\frac{G}{N}} + \|q_i(k+1) - q_s\|_{A_i} + \|r_i(k+1) - r_s\|_{B_i} + \left\| \frac{u_i(k)}{u_i^m} \right\|_{C_i} \right] \tag{6.11}$$

In Equation 6.11, $\frac{G}{N}$ is a positive weight constant. Thus, it can be replaced by constant G_i again, and Equation 6.11 can be rewritten as follows:

$$J(k) = \sum_{i=1}^{N} J_i(k) = \sum_{i=1}^{N} \sum_{k=1}^{H} \left[\|Q(k+1) - Q_s\|_{G_i} + \|q_i(k+1) - q_s\|_{A_i} + \|r_i(k+1) - r_s\|_{B_i} + \left\| \frac{u_i(k)}{u_i^m} \right\|_{C_i} \right] \tag{6.12}$$

Now, the interaction variable $Z_i(k)$ at the subsystem i can be represented in workload fractions $\alpha_i(k)$ ($\forall i \in N$) received from other subsystems j ($\forall j \in N$, $j \neq i$) according to following equation:

$$Z_i(k) = \sum_{j=1, j \neq i}^{N} \alpha_j^*(k) \tag{6.13}$$

and

$$\alpha_i(k) = 1 - Z_i(k) \tag{6.14}$$

Therefore, the constraints at each subsystem i can be represented as:

$$\text{Constraints: } Z_i(k) - \sum_{j=1, j \neq i}^{N} \alpha_j^*(k) = 0 \tag{6.15}$$

or

$$1 - \alpha_i(k) - \sum_{j=1, j \neq i}^{N} \alpha_j^*(k) = 0 \tag{6.16}$$

Now the Lagrangian $L(k)$ for the cost function $J(k)$ can be written as sum of the Lagrangian of each subsystem L_i

$$L(k) = \sum_{i=1}^{N} L_i(k) \tag{6.17}$$

where the Lagrangian of each subsystem L_i can be represented as follows:

$$L_i(k) = J_i(k) + \sum_{k=1}^{H} \beta_i(k) \left(1 - \alpha_i(k) - \sum_{j=1, j \neq i}^{N} \alpha_j^*(k) \right) \tag{6.18}$$

where $\beta_i \in R^H$ is a vector corresponding to the subsystem i that is extracted from the Lagrange multipliers vector β received from the coordinator, where $\beta \in \mathbb{R}^{NH}$. The overall problem of minimizing cost function J can be decomposed into N subsystem level problems of minimizing L_i. Furthermore, the overall problem of minimizing the cost function J can be decomposed in to N subsystem level problems in the following form:

$$\min_{q,u,\alpha} L_i(k) = J_i(k) + \sum_{k=1}^{H} \beta_i(k) \left(1 - \alpha_i(k) - \sum_{j=1, j \neq i}^{N} \alpha_j^*(k) \right) \tag{6.19}$$

such that:

$$1 - \alpha_i(k) - \sum_{j=1, j \neq i}^{N} \alpha_j^*(k) = 0 \tag{6.20}$$

$$\hat{q}_i(k+1) = \Phi_i(Q(k), q_i(k), u_i(k), \omega(k), \alpha_i(k)) \tag{6.21}$$

$$Q(1) = 0 \tag{6.22}$$

$$q_i(1) = 0 \tag{6.23}$$

The problem at the coordinator level can be expressed by updating the value of Lagrange multipliers $\beta_i(k)$, so that the interaction error or sum-squared error SSE (see Equation 6.27) can become zero or minimized to a small tolerable value ϵ, where e is the interaction error vector defined in Equation 6.24.

Input: Subsystem queue $q_l(k)$, Global Queue $Q(k)$, Prediction Horizon H
Input: Control Input Set at Subsystem i, $U_l = [u_{l1}, u_{l2}, ..,]$,
Input: β and $\alpha_j^{*(l)}(k)$ received from the coordinator
Input: $\hat{\omega}$ received from the estimator
Input: Fraction Set at subsystem i, $F_l = [F_{l1}, F_{l2}, ..., F_{lR}]$
Input: Workload share processed at subsystem i, $\alpha_l^l = [\alpha_l^l(1), \alpha_l^l(2), .., \alpha_l^l(H)]$

Subsystem state $x(k) = [q_l(k)\ r_l(k)\ Q(k)]$
$s_k := x(k)$, $Cost(k) = 0$
for all fraction set $f \in F_l$ **do**
  $\alpha_l^l(1) = \alpha_l^l(2) = .. = \alpha_l^l(H) = f$ /* Same workload fraction at each step */
  for all k within prediction horizon of depth H **do**
    $s_{k+1} := \phi$
    for all $x \in s_k$ **do**
      for all $u \in U_l$ **do**
        $\hat{q}_l(k+1) = \left[q_l(k) + (Q(k) + \hat{\omega}(k))\ \alpha_l^l(k) - \frac{u_l(k)}{u_l^m\ \hat{c}_l(k)} T\right]^+$
        $\hat{r}_l(k+1) = (1 + \hat{q}_l(k+1))\ \frac{\hat{c}_l(k)\ u_l^m}{u_l(k+1)}$
        $\hat{Q}(k+1) = \left[(Q(k) + \hat{\omega}(k))(1 - \alpha_l^l(k) - \sum_{J=1, J\neq l}^{N} \alpha_j^{*(l)}(k))\right]^+$
        Compute $L_l(k+1)$ /* using Equation 6.18 */
        $Cost(k+1) = Cost(k) + L_l(k+1)$
        $\hat{x} = [\hat{q}_l(k+1)\ \hat{r}_l(k+1)\ \hat{Q}(k+1)]$
        $s_{k+1} := s_{k+1} \cup \{\hat{x}\}$
      end for
    end for
    $k := k + 1$
  end for
end for
Find $x_{min} \in s_N$ having minimum $Cost(k)$
Choose f with minimum $L_l(k)$, where $f = \alpha_l^{*(l)} = [\alpha_l^{*(l)}(k), .., \alpha_l^{*(l)}(k+H-1)]$
Choose $u_l^{*(l)} = [u_l^{*(l)}(k), ..., u_l^{*(l)}(k+H-1)]$ leading from $x(k)$ to x_{min} during f
return $[u_l^{*(l)}\ \alpha_l^{*(l)}]$

Figure 6.4

Predictive Control Algorithm at each Subsystem.

6.4.2 Optimizing the Subsystem Level problem using Model Predictive Control

At the subsystem level, the Lagrange L_i is minimized using subsystem dynamics (Equation 6.7, 6.8, and 6.9) with control input [$u_i(k)$, $\alpha_i(k)$] constraints. We create a uniform discretization F_i for the workload fraction α_i. For example, F_i=[0.05,0.1,...,0.95,1.0]. The optimal value of control inputs [$u_i(k)$, $\alpha_i(k)$] is computed that minimizes L_i using the following model predictive control based steps.

1. Use $\beta_i^l(k)$ and $\alpha_j^{*(l)}(k)$ (where $j \neq i$) as received from coordinator to compute the optimal sequence of $(\alpha_i^{*(l)}(k), u_i^{*(l)}(k))$ over the horizon $k \in [1, H]$ by using algorithm shown in Figure 6.4, which minimizes Lagrange $L_i(k)$ in Equation 6.18 through a tree search method [48]. Here, l indicates the iteration instance between the subsystem and the coordinator within time step k.
2. Forward the optimal values of $\alpha_i^{*(l)}$ to the coordinator.

6.4.3 Solving the Coordinator Level problem using Conjugate Gradient Method

At the coordinator level, the goal is to update the values of Lagrange multipliers β in order to decrease the sum-squared error SSE (see Equation 6.27), where interaction error vector e is defined as,

$$e_i^l(k) = 1 - \sum_{j=1}^{N} \alpha_j^{*(l)}(k) \tag{6.24}$$

$$e^l = \begin{pmatrix} e_1^l & e_2^l & \dots & e_i^l & \dots & e_N^l \end{pmatrix}^T \tag{6.25}$$

$$e_i^l = \begin{pmatrix} e_i^l(1) & e_i^l(2) & \dots & e_i^l(k) & \dots & e_i^l(H) \end{pmatrix}^T \tag{6.26}$$

and

$$SSE = \|e^l\|_2 \tag{6.27}$$

Interaction error vector e is used as gradient to modify the Lagrange multipliers $\beta(k)$ using conjugate gradient method [143] as per following set of equations.

$$\beta^{(l+1)}(k) = \beta^{(l)}(k) + \xi^l \, d^l(k) \tag{6.28}$$

where ξ^l represents the step length and d^l represents the search direction. $d^l(k)$ is calculated by using following set of equations with $d^0 = e^0$.

$$d^{l+1}(k) = -e^{l+1}(k) + \sigma^{l+1} \, d^l(k) \tag{6.29}$$

$$\sigma^{l+1} = \frac{\|e^{l+1}\|_2}{\|e^l\|_2} \tag{6.30}$$

where $\| \cdot \|_2$ denotes the (Cartesian) ℓ_2-norm.

The main steps of the algorithm at coordinator level are as follows:

1. Set the initial values of the Lagrange multipliers vector β and forward it to all of the subsystems.
2. The coordinator uses the values of α_i^* received from the subsystems to calculate the interaction vector e by using Equation 6.24 and SSE by using Equation 6.27.
3. If $SSE \leq \epsilon$, stop and send the corresponding $\alpha(k)$ to the Dispatcher and the subsystems (as Figure 6.3) for workload distribution among the subsystems, else go to the next step.
4. Calculate the values of Lagrange multipliers β for the next iteration by using Equation 6.28, 6.29, and 6.30. Send this updated value of β to the subsystems for solving the subsystem level optimization problem again. Increment l and jump to Step 2.

This exchange of information is shown in Figure 6.3.

6.5 Evaluation of the Distributed Control Approach

The proposed distributed control approach for the management of the distributed web service deployment can be evaluated based on following parameters.

6.5.1 Performance Parameters

6.5.1.1 Load Distribution Among Subsystems

In a typical web service deployment over a cluster of heterogeneous N subsystems, each executing subsystem i should be assigned a fraction α_i of the total incoming workload entering into the cluster. In decentralized approaches, where a central entity of the cluster is not determining the load fraction α_i (where $0 < \alpha_i < 1$ and $\sum_{i=1}^{N} \alpha_i = 1$) assigned to the subsystems, these fractions should be determined by the subsystems themselves in a cooperative manner. In this situation, these subsystems try to minimize their load share α_i to maximize their individual performance. However, in an ideal situation, these load shares should be close to their relative measure of available system resources p_i (i.e. $\alpha_i \approx p_i$) needed for processing the incoming workload. These system resources include CPU frequency, memory, network bandwidth, etc., according to the characteristics of incoming http requests..

6.5.1.2 Subsystem Resource Utilization

Another aspect of an ideal distributed management approach is to utilize resources of each of the active subsystems equally if they are homogeneous or proportionally if they are heterogeneous. For example, if the computational resource utilization of subsystem i is ρ_i, then $\rho_1 = \rho_2 = ... = \rho_i = ... = \rho_N$. Profitability of the deployment should

also be maximized by utilizing the resources equally compared to the cases, when a few subsystems are over utilized and others are under utilized.

6.5.1.3 SLA Parameters

The proposed approach should also be evaluated with respect to the pre-specified SLA parameters. For typical web services, these SLAs include response time r_i and http requests pending inside the systems q_i at a particular time step. For a distributed deployment of the web services, these SLA parameters should be observed at each individual subsystem and also as average value at the cluster level. The average response time can be computed as: $r_{avg} = \sum_i^N (r_i \alpha_i)$. Similarly, the average queue can be computed as: $q_{avg} = \frac{\sum_i^N (q_i)}{N}$.

6.5.1.4 Utility Value

The profitability of the distributed deployment should also be improved by using the proposed distributed control approach. Increase in profitability is possible by lowering the operating cost (power consumption) and minimizing the SLA violation penalty simultaneously. In this chapter, utility of the deployment configuration is defined as negative value of the operating cost (Equation 6.5) of the deployment. Increment in the operating cost decreases the utility value.

6.5.2 Robustness Toward Failure

This feature indicates the performance of the proposed approach with respect to the complete failure of the subsystems or their unavailability during a few time samples. In an ideal situation, the proposed management approach should be able to detect the node

failure in the cluster and process the extra workload by increasing its own load share similar to its relative processing power in the remaining active nodes of the cluster. In addition to this, the load share of the nodes should also decrease when a new node joins the cluster reducing the load share on the already active nodes.

6.5.3 Computational Overhead

6.5.3.1 Interaction Between the Coordinator and the Subsystems

In the proposed approach, the communication between the subsystems and the coordinator is used to find the optimal set of the load share α_i (where $\sum_{i=1}^{N} \alpha_i \approx 1$) to distribute the incoming workload among the subsystems. Therefore, the primary overhead of the proposed approach is the number of iterations I between the coordinator and the subsystems. Additionally, the communication delay in each of these interactions between the coordinator and the subsystem could also be a potential overhead. Another factor that can be observed during this interaction is the *interaction error vector* e, which is used for calculating SSE. This vector represents the amount of difference between the current configuration (sum of load fractions from all of the subsystems) and the optimal value of the configuration, when the sum of the load fractions from all of the subsystems is equal to 1. Therefore, the convergence rate of the interaction error (SSE) towards tolerance value ϵ should also be observed.

6.5.3.2 Computational Overhead at each Subsystem

In a control-based management approach, the computational overhead can be measured as the number of explored states in the controller while determining the optimal set of

control inputs. These future states are explored by the controller while generating the tree for all possible future paths. Given that N is the number of subsystems, H is lookahead horizon steps, U is the control input set, and F is the size of fraction set for α, we can compute the number of future states in case of exhaustive search strategy [48] as follows.

In the distributed control approach, the total number of states visited S_D in each subsystem for calculating the optimal set of load fraction (α_i) and control input (u_i) is given approximately as,

$$S_D = N(F^{H+1} + |U|^{H+1})\ I \tag{6.31}$$

where, I is the number of iterations performed between the coordinator and the subsystems. Similarly, in case of a centralized control approach using similar parameters, the total number of future states explored S_C at centralized controller to calculate optimal set of load fraction (α_i) and control input (u_i) is given approximately as,

$$S_C = (F + N\,|U|)^{H+1} \tag{6.32}$$

The ratio between the two cases defines the reduction R in the computation overhead achieved by the distributed control algorithm and is given by.

$$R = \frac{S_C}{S_D} \approx \frac{N^H}{2\ I} \tag{6.33}$$

For the above approximation, we assumed that $F \approx |N|$. Accordingly, the reduction R depends upon the number of iterations I needed to calculate the optimal set of control

inputs. However, the factor N^H should be significantly higher for a large scale system. For medium size systems the number of iterations is more significant. This factor can be decreased to an extent by using higher value of step length, higher value of interaction error tolerance ϵ, and better convergent techniques [132] for updating the Lagrange multipliers β at the "Coordinator".

6.5.4 Reducing the Computational Overhead in the Proposed Approach

6.5.4.1 Reducing the Number of Interactions

The number of interactions between the coordinator and the subsystems can be reduced by using a technique that drives the interaction vector toward the optimal value swiftly or reduces the interaction error SSE among the load shares desired by the subsystems. For this purpose, two different values of interaction error tolerance ($\epsilon = 0.05 \; or \; 0.02$) are used to study the convergence rate (see Section 6.6.5). Moreover, a faster convergent technique [132] was developed to use the gradient in system state, control input, and coordination parameter to estimate the value of the future interaction vector; however, to reduce the amount of information transfer between the coordinator and the subsystems, we have not applied the approach described in [132].

6.5.4.2 Reducing the computations at each subsystem

The look-ahead horizon H indicates the explored depth of control inputs in the future by the controller during the calculation of the next optimal control input. If the estimation of future environment inputs are accurate, increasing the value of the look ahead horizon H can significantly improve the controller performance. However, due to the stochastic

nature of the environment input, these future estimations are bounded to contain estimation errors, which will accumulate with increasing look ahead horizon size and will deteriorate the overall controller performance. Therefore, the look ahead horizon H should be chosen appropriately in order to balance the controller performance and computational overhead.

In addition to the above methods, the controller performance can also be increased by applying enhanced tree search techniques (greedy, pruning, heuristics, and A^*), which reduce the number of explored future states (computational overhead) compared to uniform cost tree search methods that explore all the possible states [48].

6.6 Case Study: Performance Management of a Distributed Web Service Deployment

The proposed interaction balance based management approach is simulated by using Matlab to manage the performance of a web service application that is hosted in a distributed environment over four computing nodes. Details of the simulations are described in following subsections.

6.6.1 System Model

The distributed deployment is configured as shown in Figure 6.3. All of the four nodes are connected to the *Coordinator* system that also hosts the *Dispatcher* module and controls a fraction of the incoming workload $\omega(k)$ distributed to the subsystems. A *Traffic Estimator* module was developed that uses an ARIMA filter (described in next subsection) to estimate the future arrival rate of the incoming workload. The Coordinator system contains a global queue (Q), which accepts the incoming http requests from the environment

and forwards them to the dispatcher module. The *Dispatcher* module forwards the http requests to the subsystems according to the pre-specified load fractions.

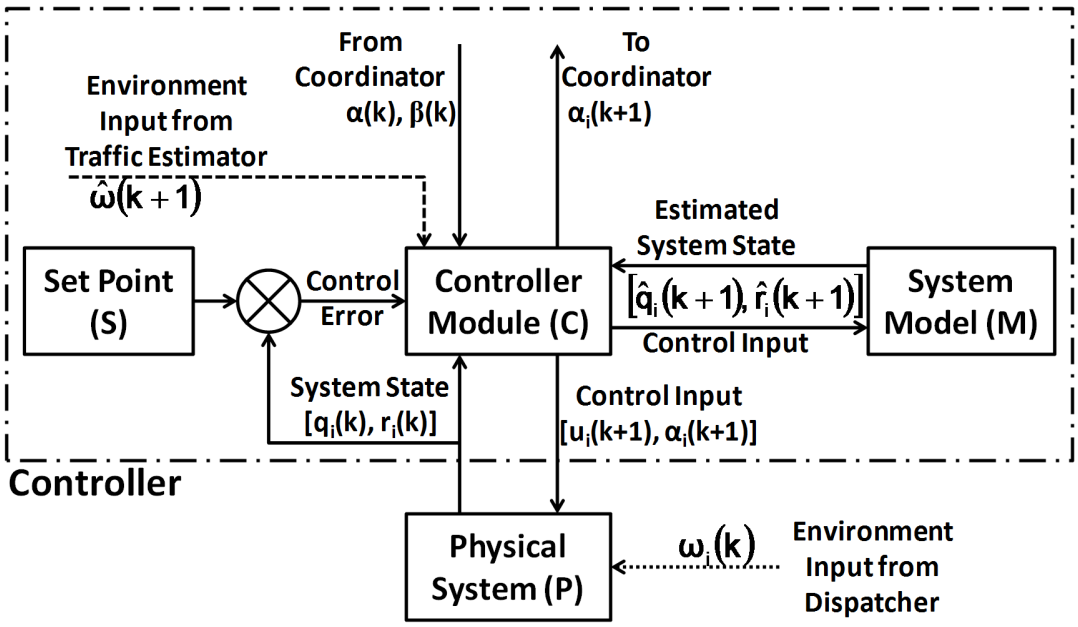


Figure 6.5

Subsystem Level Control Architecture.

The subsystem controller dynamics are shown in Figure 6.5. Each subsystem receives a fraction of cluster wide incoming http requests at time sample k. The subsystem forwards its current state $[q_i(k)\ \ r_i(k)]$ to the controller module. Each physical subsystem works under a finite set of control input frequencies $u_i \in U_i$ and a discretized value of the load share $\alpha_i(k)$. The Controller module receives current system state $[q_i(k)\ \ r_i(k)]$ from the physical subsystem, the future workload arrival rate $\hat{\omega}(k+1)$ from the traffic estimator, and Lagrange multipliers β from the coordinator. The controller module uses the physical

subsystem model and available control algorithms to obtain the optimal set of control inputs $[u_i(k+1)\ \alpha(k+1)]$ using Equation 6.18. The calculated interaction input (load share) $\alpha(k+1)$ is sent back to the coordinator to calculate the interaction error.

6.6.2 Forecasting Environment Input

We developed a single estimator module, which monitors the incoming workload $\omega(k)$ and publishes the future workload estimates $\hat{\omega}(k)$ to each of the subsystems. This estimator uses an ARIMA filter for estimation, which is similar to the estimator developed in Section 4.3. We can also implement this estimator inside each subsystem instead of using a centralized one to reduce the communication for the workload estimation.

6.6.3 Experiment Setup

A web service system consisting of four computing nodes is utilized to demonstrate the performance of the proposed approach. Simulation settings are shown in Figure 6.6. These four computing nodes operate on five different discrete frequencies as shown in Figure 6.6. Additionally, various coefficients used in cost functions are also shown in Figure 6.6. Nodes 1 and 2 are assigned a higher penalty for power consumption, while Nodes 3 and 4 are assigned a higher penalty for queue size and response time. These settings reflect heterogeneous computing nodes with different priorities to performance objectives (queue size, response time, and power consumption) in the cluster. Therefore, Nodes 1 and 2 are expected to minimize power consumption by using lower frequencies, while Nodes 3 and 4 are expected to minimize the node level queue size and response time by using higher frequencies. All of the nodes are assigned the same penalty value for the global

queue, so all of the nodes will give equal consideration to global queue while calculating the optimal values of control inputs. During these simulations, three different synthetic workloads (see Figure 6.7) of 500 time samples were used, which were based on 1998 Football World Cup web server traffic [56].

Node No.	Queue Weight (A)	Response Time Weight (B)	Power Consumption Weight (C)	Global Queue Weight (G)	SLA (q_s,r_s)	Frequencies (GHz.)
N1	1	1	25000	1	(0,0)	1.0, 1.4, 1.6, 1.8, 2.0
N2	1	1	30000	1	(0,0)	1.5, 2.1, 2.4, 2.7, 3.0
N3	25	25	1	1	(0,0)	1.0, 1.4, 1.6, 1.8, 2.0
N4	50	50	1	1	(0,0)	1.5, 2.1, 2.4, 2.7, 3.0

Figure 6.6

Simulation Settings.

Performance of the developed distributed-control based performance management approach is demonstrated in following three simulations with specific objectives as follows:

1. Compare the performance of the proposed distributed control approach over a centralized control based approach.
2. Investigate the impact of workload and interaction error tolerance value ϵ on the performance of the proposed approach in number of interactions between the two levels and the interaction error convergence.
3. Demonstrate the performance of the proposed approach in the case of subsystem failures to show the adaptivity of the proposed approach.

More details and results of the simulations are discussed in following subsections.

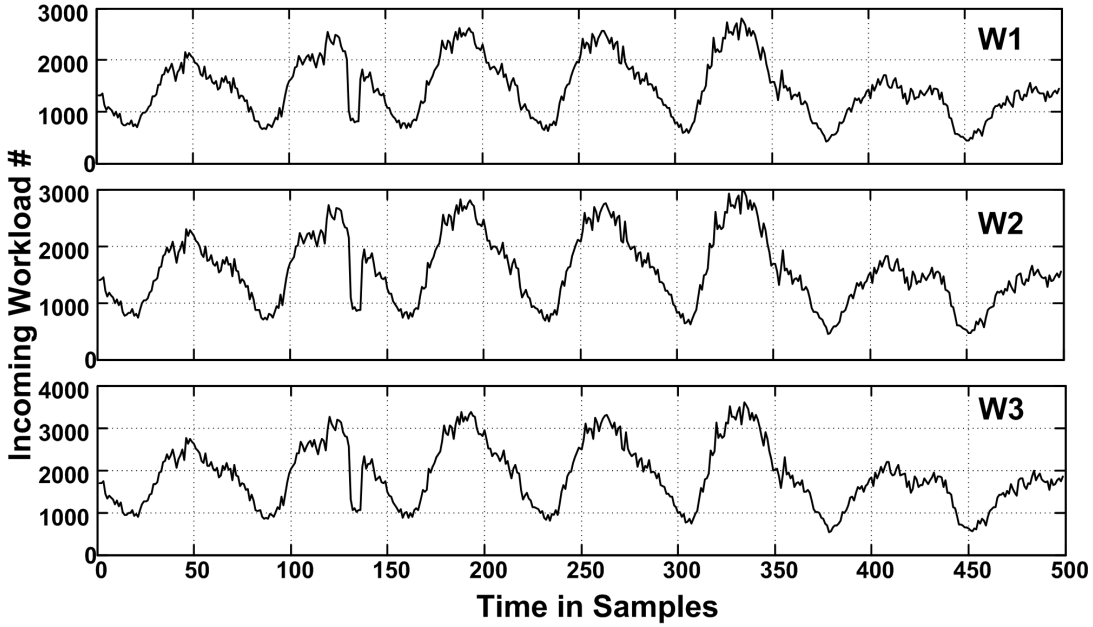


Figure 6.7

Workloads $W1$, $W2$, and $W3$ were used During the Simulations.

6.6.4 Simulation-1: Comparing the Performance of Distributed Control Approach with a Centralized Control Approach

This simulation was performed to compare the performance of the proposed distributed control approach directly with a centralized approach managing the same deployment. In this simulation, workload $W3$ (see Figure 6.7) is used as incoming http requests. Interaction error tolerance ϵ in this simulation is chosen as 0.05. The centralized control approach is expected to use model predictive control to compute the set of optimal control inputs u and α for each subsystem i $(u_i(k), \alpha_i(k))$ by using Equation 6.5. The concept of global queue is not utilized in centralized approach here, as in the case of a centralized controller, all of incoming workload is transferred to the subsystems for processing at each time sam-

ple. The cost optimization method that is used to compute the optimal control inputs in the model-based centralized control approach is presented in algorithm shown in Figure 6.8.

The performance of the proposed approach is compared to the centralized control approach with respect to the load share values (α_i), the applied CPU core frequency (u_i), the queue size (q_i), and the response time (r_i). The comparative plots are presented in Figure 6.9 to Figure 6.15.

Results from two separate simulations related to the proposed distributed control approach and a typical centralized control approach are compared with respect to the following performance metrics:

According to Figure 6.9, the proposed distributed control approach distributes the incoming workload among the subsystems more uniformly and closer to the subsystem's relative processing capabilities in the cluster. In case of the centralized approach, the incoming workload is distributed in random share values that minimize the overall cost function. Additionally, the distributed control approach utilizes all nodes fairly per their processing capability, instead of distributing a large fraction of workloads to a few nodes and a small fraction of workload to others as in the case of the centralized approach. Furthermore, the distributed control approach adapts to the changes in the workload arrival rate by changing the load distribution on the nodes in order to maintain their individual utility and total deployment utility simultaneously, while the centralized approach only considers the total utility of the system and distributes the incoming load among the subsystems accordingly.

According to Figure 6.10, the distributed control approach applies higher frequencies on Nodes 1 and 4 compared to the centralized approach, while the distributed approach ap-

Input: System queue $q(k) = [q_1(k), q_2(k), ..., q_N(k)]$, Prediction Horizon H
Input: Control Input Set U, $U = [U_1, U_2, .., U_N]$, /* where U_i represents control input set at Node i */
Input: $\hat{\omega}$ received from the estimator
Input: Fraction Set for workload distribution $F = [F_1, F_2, ..., F_N]$
Input: Fraction Set at subsystem i, $F_i = [F_{i1}, F_{i2}, ..., F_{iR}]$
Input: Workload share processed at subsystem i, $\alpha_i^l = [\alpha_i^l(1), \alpha_i^l(2), .., \alpha_i^l(H)]$

System state $x(k) = [q(k)\ r(k)]$, where $q(k) = [q_1(k), q_2(k), ..., q_N(k)]$
$s_k := \{x(k)\}$, $Cost(k) = 0$
for all k within prediction horizon of depth H **do**
  for all node $i \in N$ **do**
    for all fraction set $f \in F_i$ **do**
      $\alpha_i^l(1) = \alpha_i^l(2) = .. = \alpha_i^l(H) = f$, /* Same workload fraction at each step */
      $s_{k+1} := \phi$
      for all $x \in s_k$ **do**
        for all $u \in U_i$ **do**
          $\hat{q}_i(k+1) = \left[q_i(k) + \hat{\omega}(k)\ \alpha_i^l(k) - \frac{u_i(k)}{u_i^m\ \hat{c}_i(k)} T\right]^+$
          $\hat{r}_i(k+1) = (1 + \hat{q}_i(k+1))\ \frac{\hat{c}_i(k)\ u_i^m}{u_i(k+1)}$
        end for
      end for
      $Cost(k+1) = Cost(k) + J(k+1)$
      $\hat{x} = [\hat{q}_i(k+1)\ \hat{r}_i(k+1)]$
      $s_{k+1} := s_{k+1} \cup \{\hat{x}\}$
    end for
  end for
  Compute Cost $J(k+1)$ /* using Equation ?? */
  $k := k + 1$
end for
Find $\{\hat{x}\} \in s_N$ having minimum $Cost(k)$
Choose F with minimum $Cost(k)$, where $F = [F_1, F_2, ..., F_N]$
and $F_i = \alpha_i = [\alpha_i(k), .., \alpha_i(k+H-1)]$
Choose u with minimum $Cost(k)$, where $u = [u_1, u_2, ..., u_N]$
and $u_i = [u_i(k), ..., u_i(k+H-1)]$ leading from $x(k)$ to x_{min} during F_i
return $[u_i\ \alpha_i]$

Figure 6.8

Centralized Control Algorithm for Calculating Values of Control Inputs.

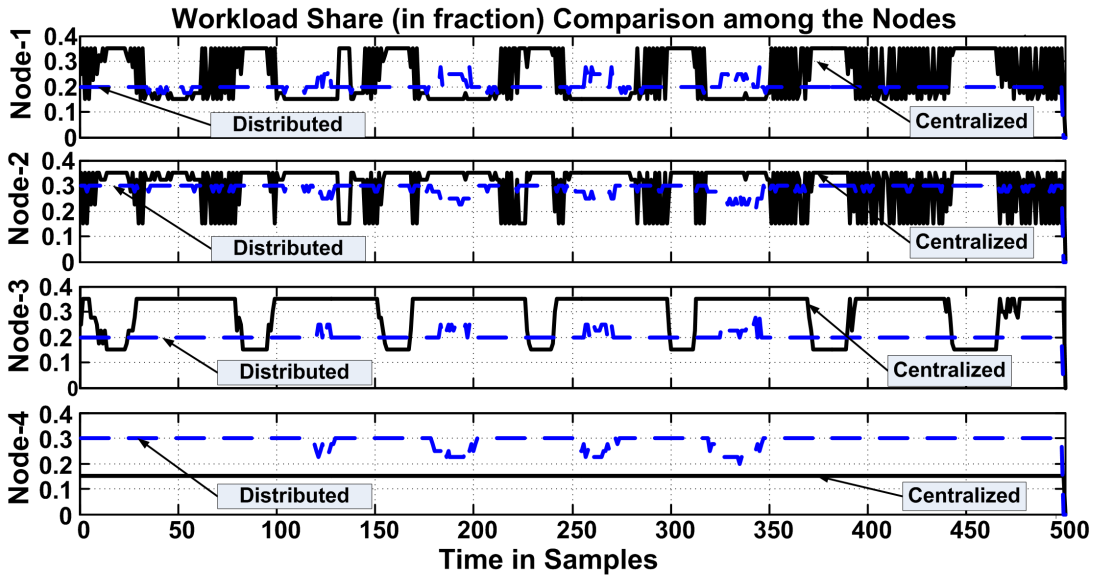


Figure 6.9

Simulation-1: Workload Share Comparison Between the Developed Approach and the Centralized Approach.

plies lower frequencies on Nodes 2 and 3 compared to the centralized approach. According to Figure 6.11, the distributed approach has lower values of response time on all the nodes except on a few occasions at Nodes 1 and 4. The primary reasons for higher values of response time in the distributed control approach is due to a higher workload processed at Nodes 1 and 4 (see Figure 6.9) at these time steps. However, the average response time across all nodes (Node 1 to 4) is still lower in the case of the distributed control approach as expected.

According to Figure 6.12, queue size at each subsystem follows similar trend as response time. According to Figure 6.13, global queue is the amount of workload remaining at the coordinator if the combined sum of load share fraction demanded by each node is less than "1". In distributed control approach, global queue size is mostly zero except in

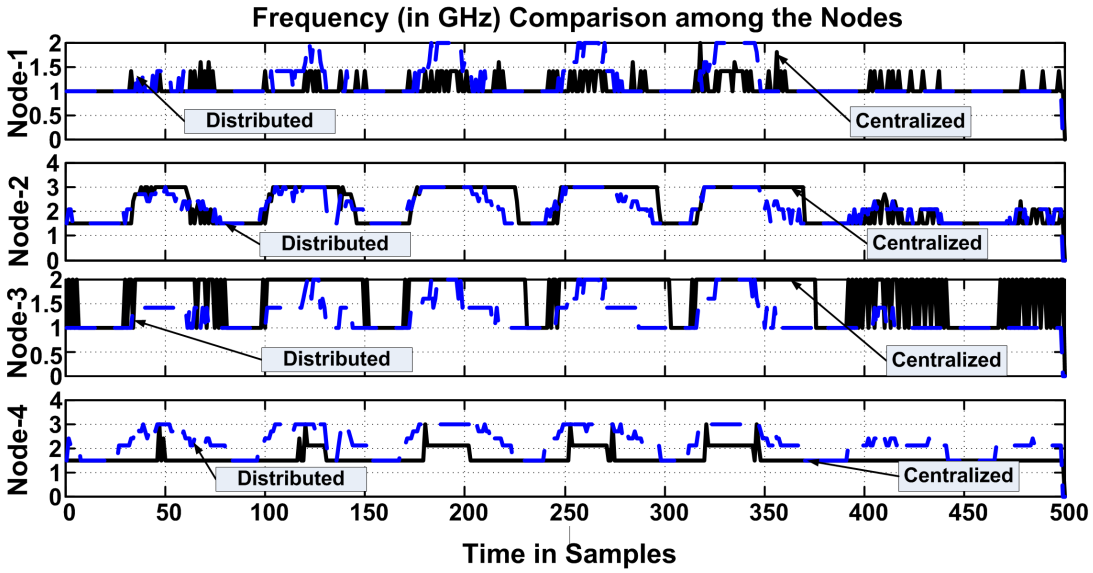


Figure 6.10

Simulation-1: Comparison of Applied Frequency (Power Consumption) Between the Developed Approach and the Centralized Approach.

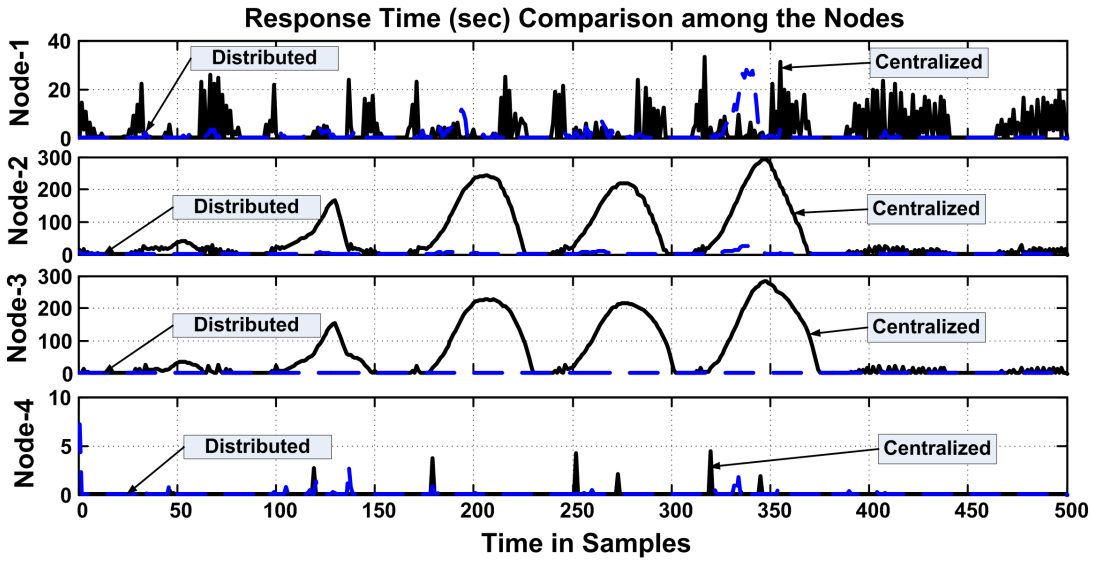


Figure 6.11

Simulation-1: Comparison of Response Time Between the Developed Approach and the Centralized Approach.

the cases of extremely high workload rate. However, this global queue size is still less than 5% of the arrived workload.

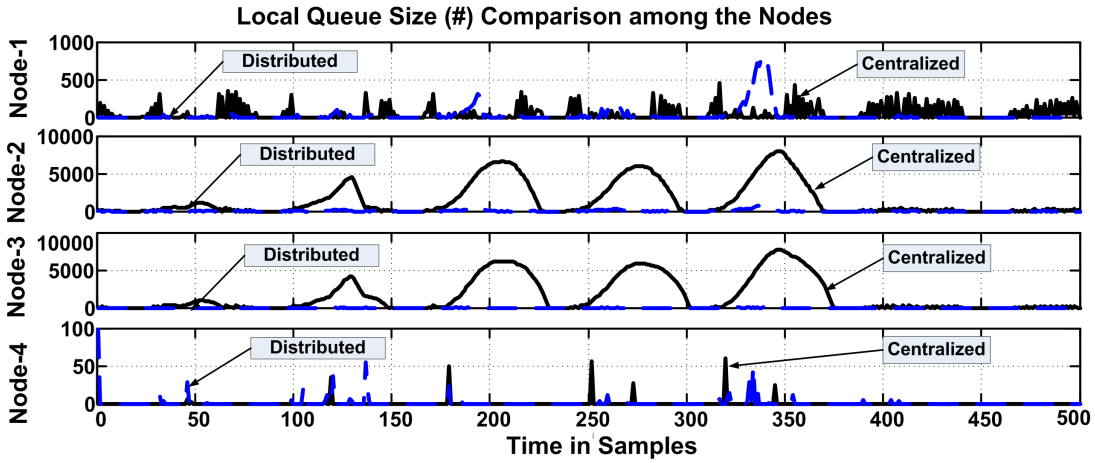


Figure 6.12

Simulation-1: Comparison of Queue Size Between the Developed Approach and the Centralized Approach.

The total load share is sum of the load fractions demanded by each node from the coordinator during a sample as shown in Figure 6.13. Ideally, this value should be equal to "1." However, in the case of an extremely high workload arrival, this sum can be lower than 1 (allowed value is 0.95) in this experiment. This lower than "1" value is useful to maintain the low value of the local queue size at each node, which has a higher priority compared to the global queue. Thus, the workload available in the global queue can be processed by the lightly loaded node in the future, instead of assigning to a extremely loaded node at the current sample time. In the distributed control approach, the total load share is mostly "1" except in the cases of an extremely high workload rate. In a few cases, total load share goes

more than "1" too (max allowed value is 1.05) indicating that nodes show an inclination toward processing a higher workload to maximize the utility of the cluster.

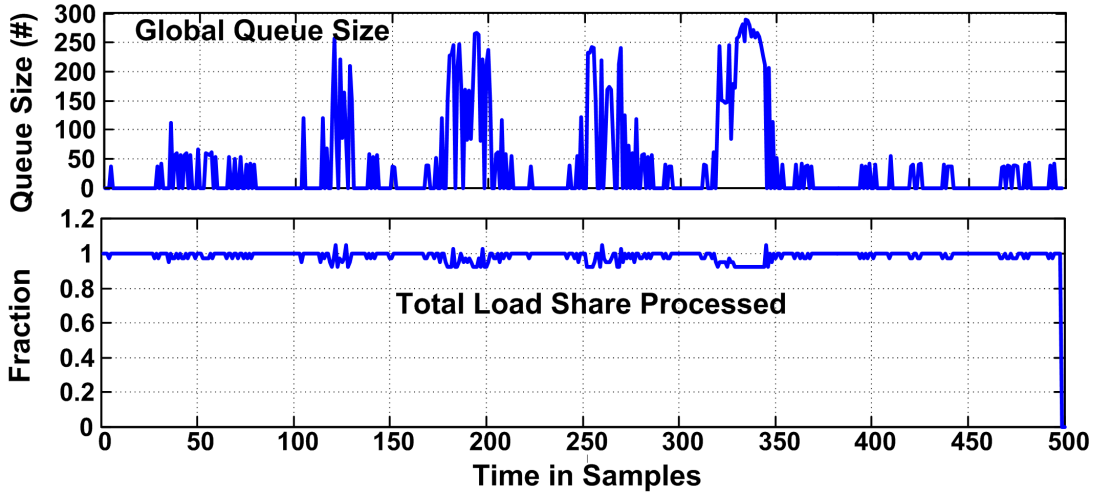


Figure 6.13

Simulation-1: Global Queue Size (Q) and Total Load Share Processed by all the Nodes in the Distributed Control Approach.

Number of iterations indicates the number of interactions that have occurred between the computing nodes and coordinator while computing the optimal solution to maximize both the operating cost of the deployment and of the computing node. According to Figure 6.14, this interaction vale is "1," most of the time. However, in the case of an extreme arrival rates it varies between "1" to "100" due to different relative priorities at each node for queue size, response time, and power consumption. In the case of low communication delay in interaction, the total time spent in multiple interactions can still be lower than the computation time required to calculate the optimal control inputs while using centralized approach due to large set of possible control input combinations.

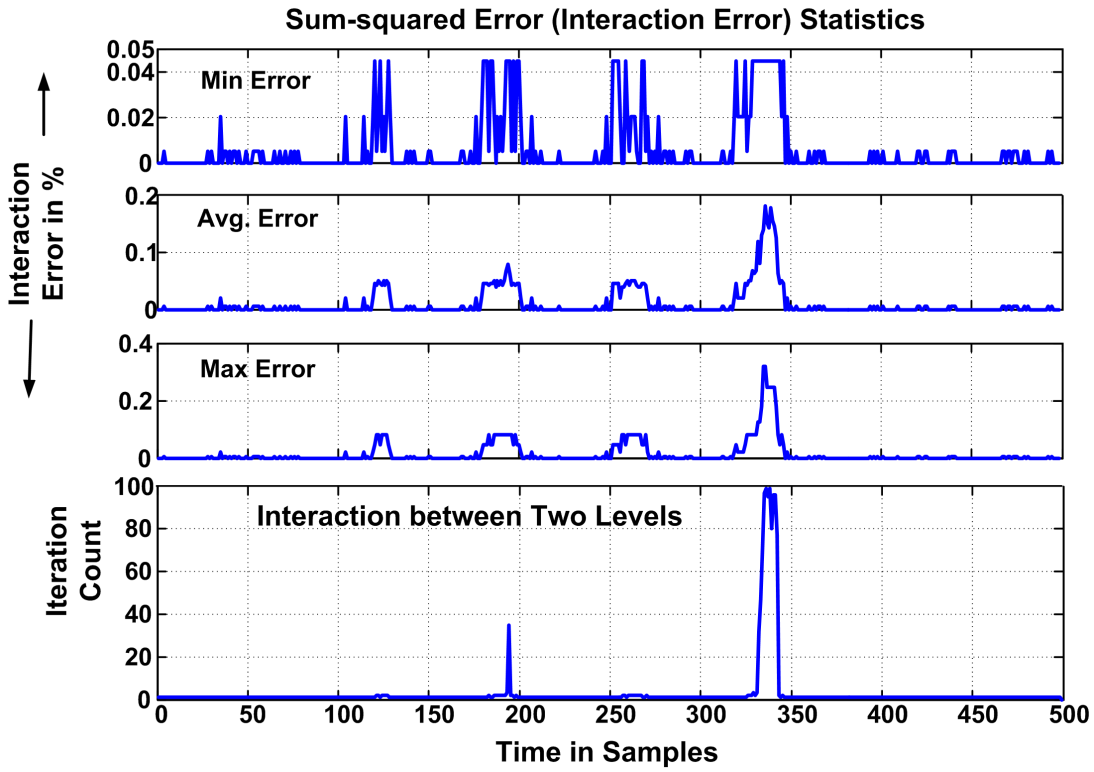


Figure 6.14

Simulation-1: Number of Interactions Between Coordinator and Nodes.

Figure 6.14 shows the minimum, average, and maximum value of interaction error at each sample. These error values indicate the difference between the optimal sum of the load share ("1") and the sum of the desired load shares by each node during the interaction at each sample time. This interaction error increases with the increase in the workload (see workload $W3$ in Figure 6.7) due to the conservative nature of nodes to minimize queue size, response time, and power consumption.

The total utility value for the centralized and the distributed control approach are computed offline by using a centralized cost function (Equation 6.5) and values of the queue size, response time, and frequencies from both of the simulations at each node. Here, utility is defined as negative of the cost of operation. The higher the cost of operation, the lower the utility. According to Figure 6.15, an extremely low value of utility (higher cost) in the case of the centralized approach is due to the higher queue size and the higher response time compared to the distributed approach at extremely high workload arrival instances. The utility difference plot is computed to show the improvement in utility values due to the developed distributed approach. The average improvement in the utility value $((C1 - C2)/|C2|)$ is almost equal to 0.98, which indicates a near 100% improvement. Additionally, the $CPUtime$ measured in Matlab in the developed distributed approach shows improvement of almost 95% with respect to the centralized approach. This improvement in computation time is due to smaller size of the control input set ($[U_i, \alpha_i]$) at each computing node in the distributed approach compared to the centralized approach ($[U, \alpha]$), which considers all the computing nodes together.

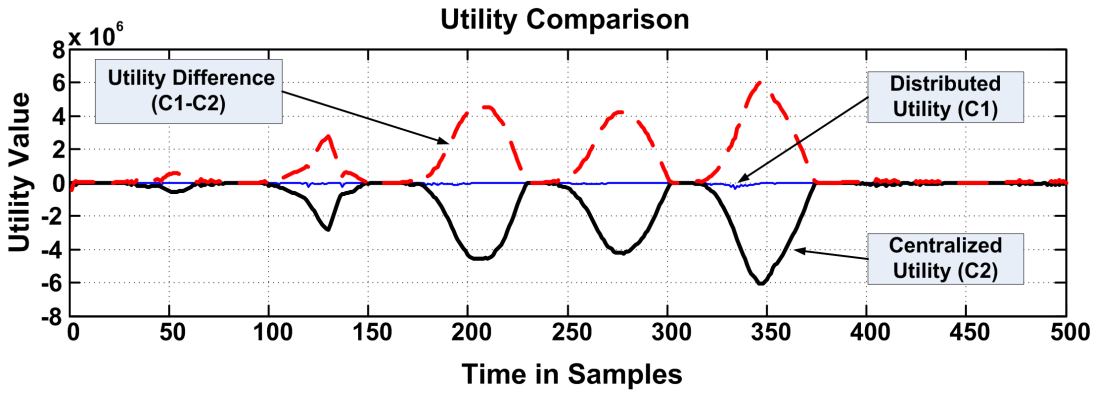


Figure 6.15

Simulation-1: Comparison of System Utility Between the Developed Approach and Centralized Approach.

6.6.5 Simulation-2: Impact of the Workload Arrival rate and Error Tolerance Value on Distributed Control Approach

This simulation was performed to investigate the impact of the workload arrival rate and interaction error tolerance value on the overhead in computing the appropriate control inputs that maximize the utility of the deployment. For this simulation, two http workloads, $W1$ and $W2$ (see in Figure 6.7) of different magnitudes are used. These workloads are chosen as they represent the boundary of the workload arrival rate that can be easily processed by the cluster of four computing nodes. Additionally, two different values of interaction error tolerance ϵ ($T1 = 0.02$ and $T2 = 0.05$) at the coordinator to compute the Lagrange multipliers β are considered. The primary purpose of choosing different values of interaction error tolerance is to analyze its impact on the number of interactions performed between the coordinator and the subsystems. Moreover, a lower value of interaction error tolerance ϵ will ensure that the maximum share of the incoming workload is

processed inside the cluster at the current time sample. Therefore, four different simulations of combination $W1-T1$, $W1-T2$, $W2-T1$, and $W2-T2$ are performed. The results of the experiments are shown in Figure 6.16, Figure 6.17, and Figure 6.18.

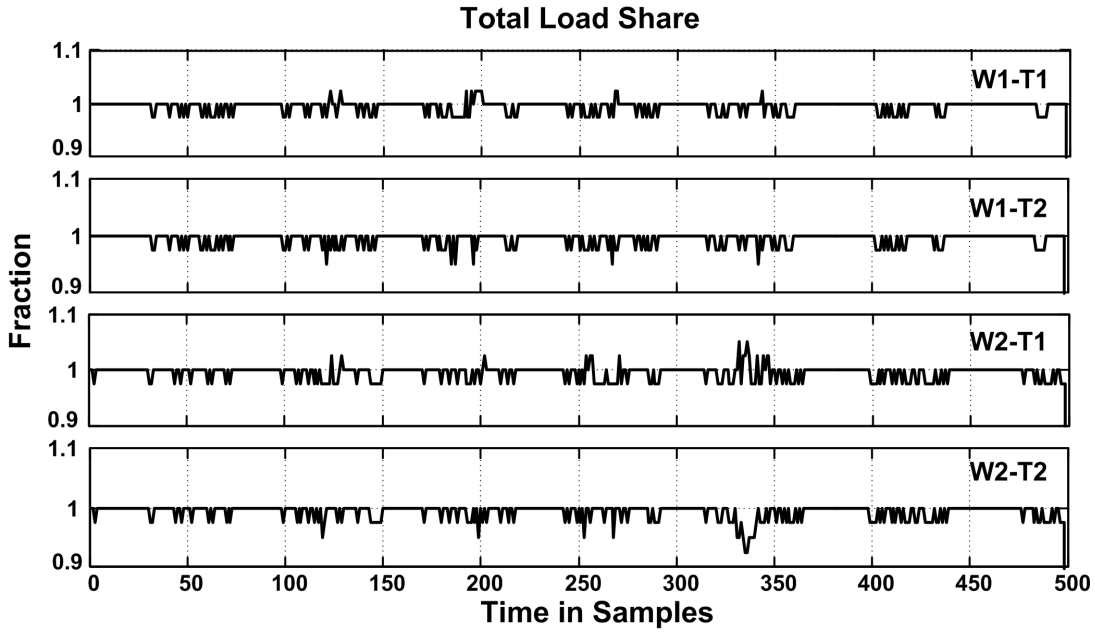


Figure 6.16

Simulation-2: Total Load Share Processed.

According to Figure 6.16, in the case of a low value of interaction error tolerance value $T1$, the total load share processed by the subsystems is higher compared to the case of higher error tolerance $T2$. Therefore, a lower value of error tolerance will ensure that the maximum amount of incoming http requests are distributed to the subsystems, which decreases global queue size substantially.

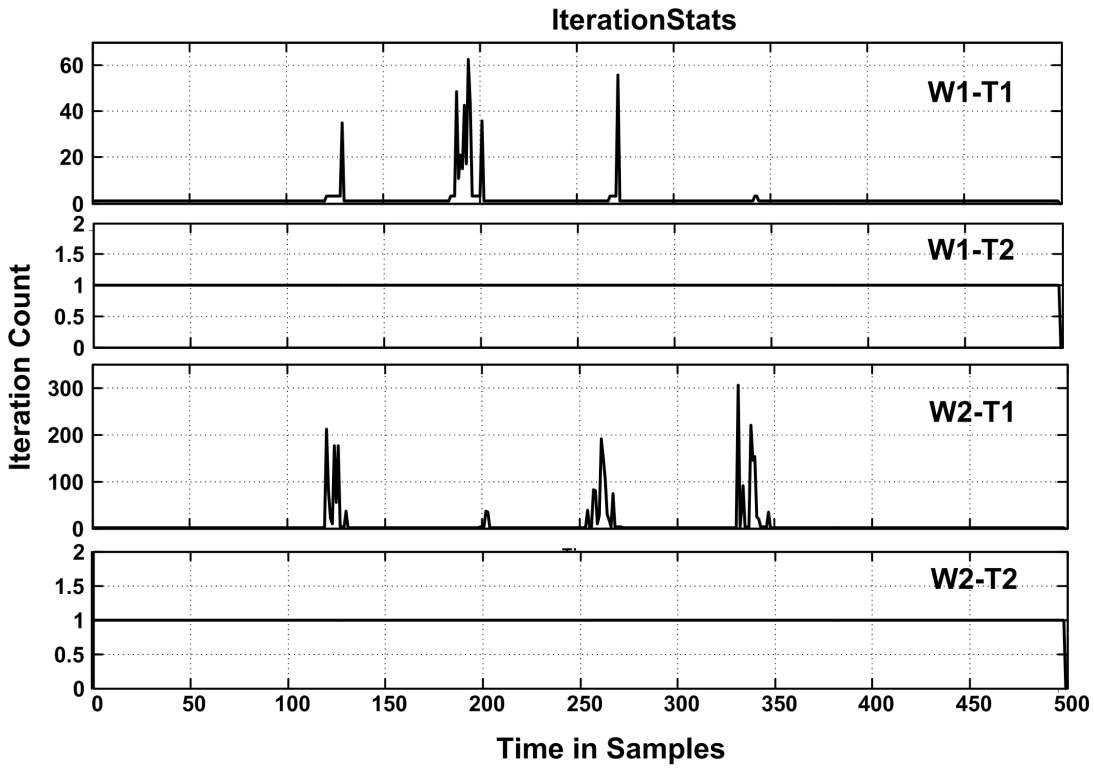


Figure 6.17

Simulation-2: Number of Interactions.

In the case of a low value of interaction error tolerance value $T1$, the number of interactions between computing nodes and the coordinator increases substantially to compute the optimal value of control inputs, even at the same workload arrival rate ($W1$ or $W2$) as shown in Figure 6.17. These results show that error tolerance value should be chosen carefully to minimize the number of interactions between the subsystem and the coordinator, which in turn increases the performance of the proposed approach.

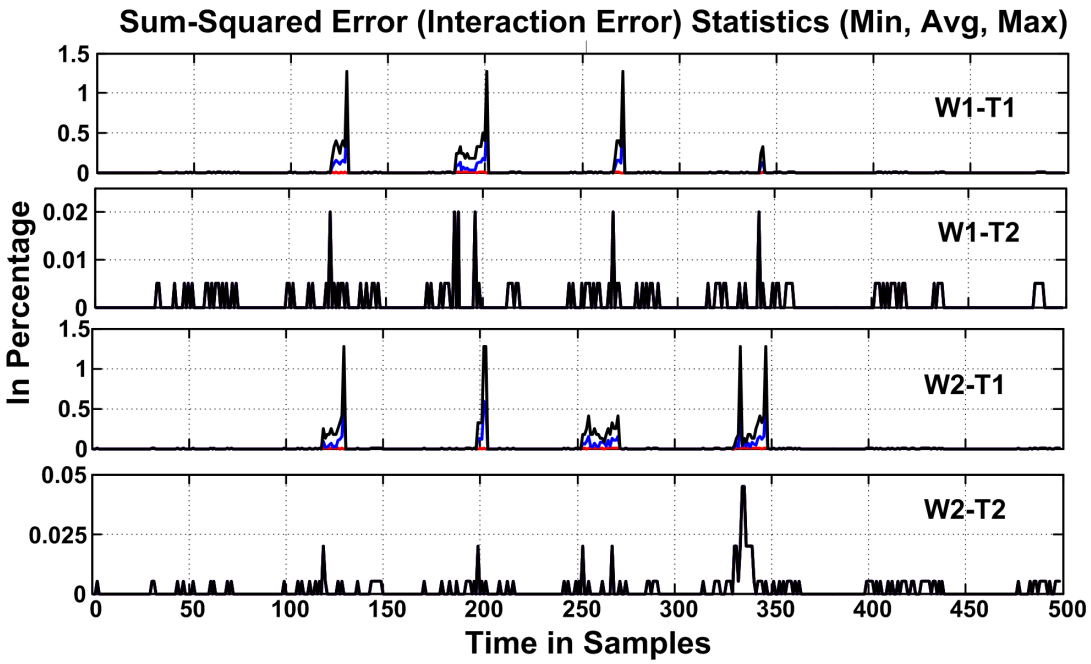


Figure 6.18

Simulation-2: Interaction Error Statistics.

The amount of interaction error (minimum, average, and maximum values during a time step) increases with the increase in the workload arrival rate as shown in Figure 6.18. This phenomenon can be explained by the relative priority of the computing nodes to vari-

ous performance parameters including response time, queue size, and power consumption. At high workload arrival rate, each computing node tries to maximize its utility function either by reducing or by increasing the workload fraction. This creates higher amount of interaction error at the initial few interactions between the subsystems and the coordinator. However, after a few interactions and with the updated Lagrange multipliers, this interaction error is minimized and optimal control inputs are determined.

According to this simulation, error tolerance value should be chosen carefully with respect to the incoming workload that increases the total share of the processed workload while keeping the number of interactions low between the subsystems and the coordinator.

6.6.6 Simulation-3: Evaluation of Robustness Towards Subsystem Failure

This simulation was performed to check the adaptivity feature of the proposed distributed control approach in case of a computing node failure or addition of a new computing node to the cluster. For this simulation, workload $W1$ (see Figure 6.19) is used. According to Figure 6.19, subsystem 2 fails between time steps 70 to 110, when the workload arrival rate is decreasing. Subsystem 1 fails between time steps 250 to 290, when workload arrival rate is increasing. Results of this simulation are show in Figure 6.20 to Figure 6.25.

The load share at Nodes 1, 3, and 4 change once subsystem 2 fails during time step $70 - 110$ as shown in Figure 6.20. The extra load share processed by Nodes 1, 3, and 4 are close to their relative processing capacity in the cluster. Similarly, when subsystem 2 comes back online and joins the cluster of subsystems, load shares are restored back to the

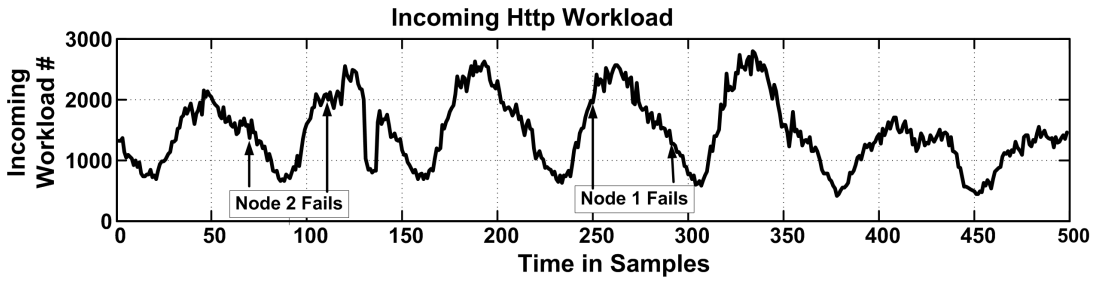


Figure 6.19

Simulation-3: Workload Arrival rate.

the original values. A similar trend is observed when subsystem 1 fails between time step 250 and 290.

According to Figure 6.21, after the failure of subsystem 2, subsystems 1, 3, and 4 do not change their frequencies initially as the workload is decreasing during time step $70 - 110$ (see Figure 6.19). However, after the failure of subsystem 1, subsystems 2, 3, and 4 change their frequency and execute at higher frequencies in order to maintain the deployment profitability during time step $250 - 290$. During the same time step, the response time is not increased at subsystems 3 and 4. However, the response time at subsystem 2 is increased because subsystem 2 does not increase its frequency due to subsystem 2's power-conservative nature.

After the failure of subsystem 1 during sample time $250 - 290$, subsystems 2, 3, and 4 increase their frequencies to the maximum value (see Figure 6.21) as the workload arrival rate is increasing significantly. Due to extreme fluctuations of the workload, response time levels are increasing in each subsystem; however, due to the relative priorities of the

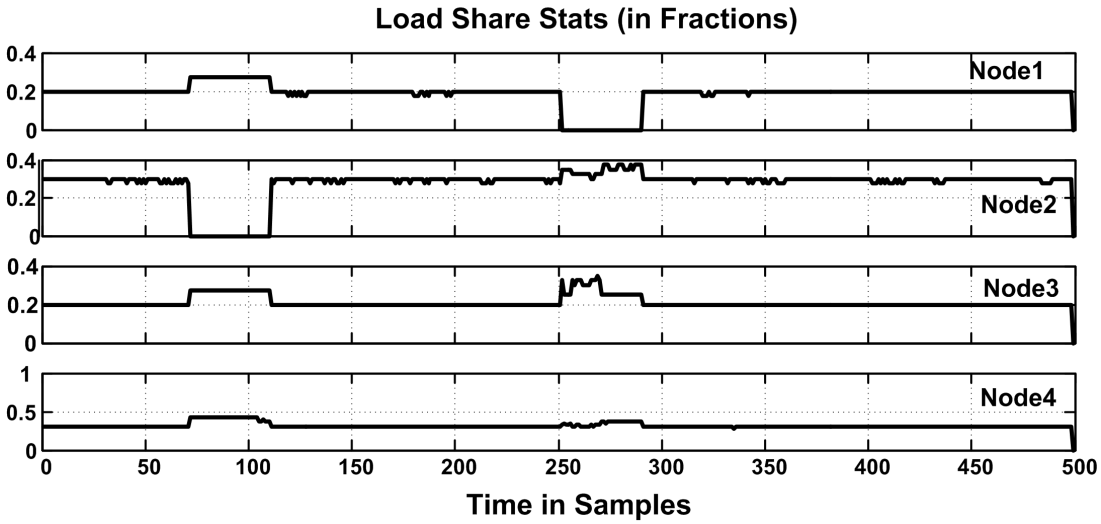


Figure 6.20

Simulation-3: Workload Share Processed by Subsystems in the Cluster.

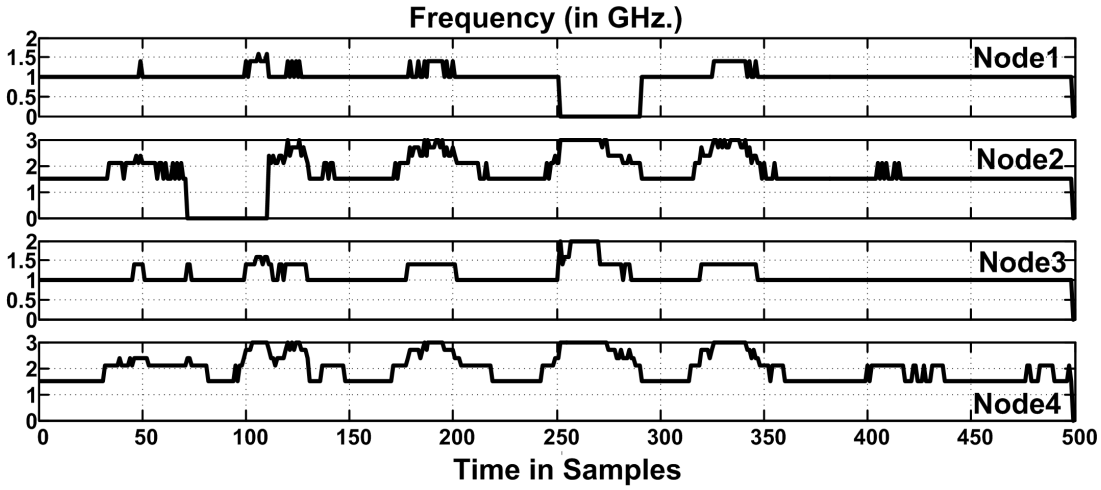


Figure 6.21

Simulation-3: Frequency Values used by the Nodes.

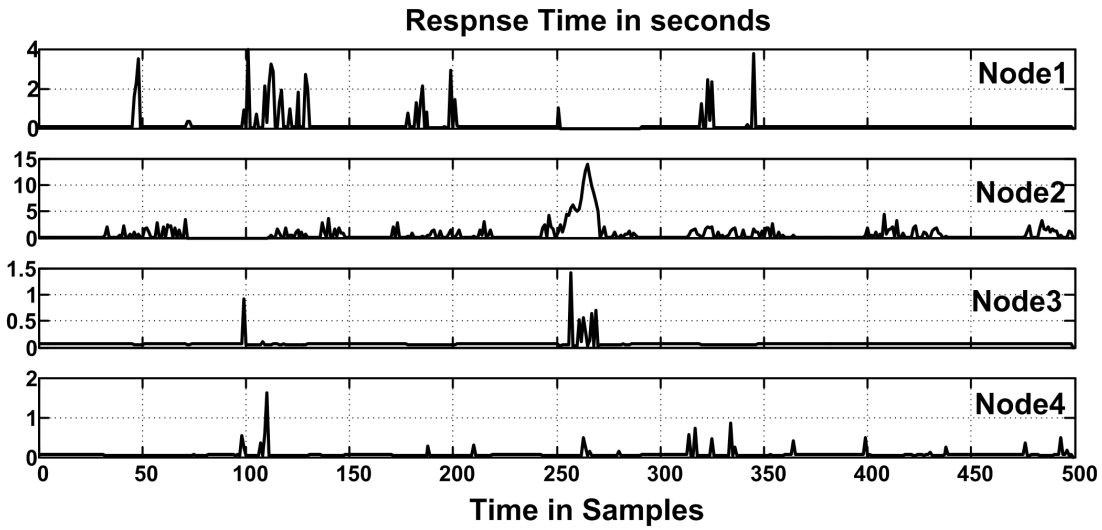


Figure 6.22

Simulation-3: Response Time at Subsystems.

response time and the queue size of each node, the increment in response time is lower on subsystems 3 and 4 while higher on subsystem 2.

According to Figure 6.23, the queue size shows a similar trend as the response time during the simulation. Figure 6.24 shows that the global queue size is increased at the time step 70 and 250 when subsystems 2 and 1 fail. However, the global queue size is back to a minimum as soon as the active subsystems adjust their load shares.

The number of interactions between the coordinator and the subsystems 1, 3, and 4 are unchanged (see Figure 6.25) when subsystem 2 fails during time step $70 - 110$. However, the number of interactions are increased when subsystem 1 fails during time step $250 - 290$ due to increase in the incoming workload.

According to Figure 6.25, maximum error in the interaction variable is increased extremely when subsystem 1 fails during time step $250 - 290$ compared to the time step

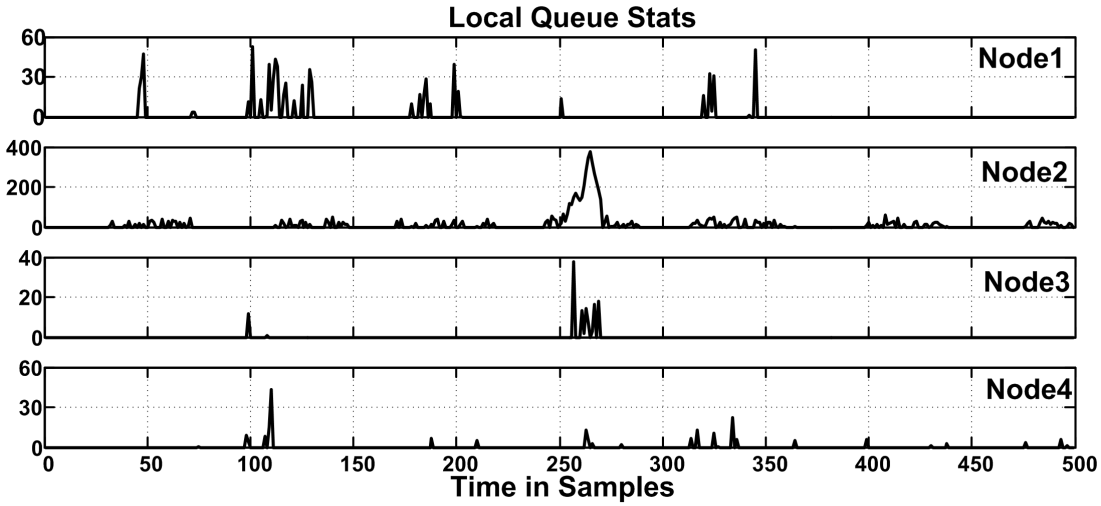


Figure 6.23

Simulation-3: Queue Size at Subsystems.

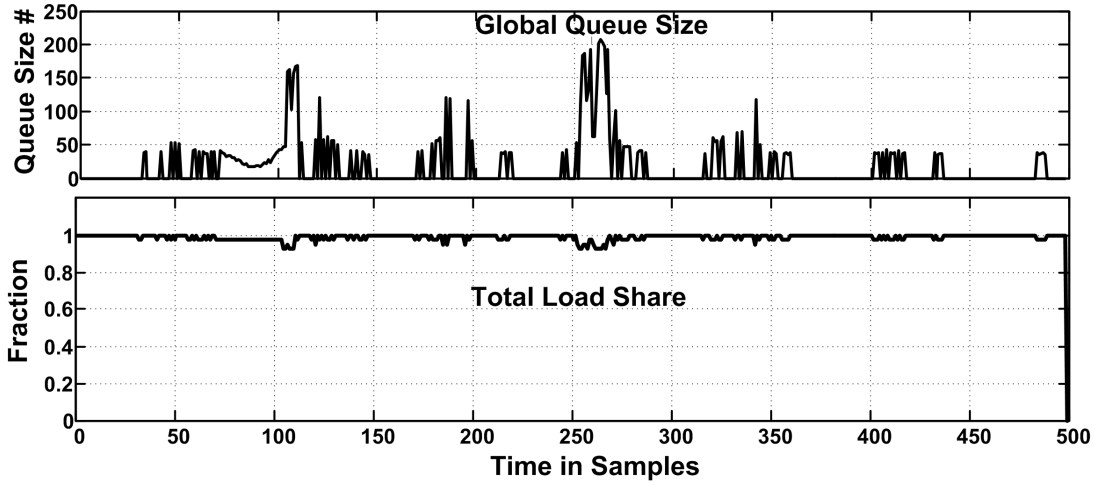


Figure 6.24

Simulation-3: Global Queue Size (Q) and Total Load share Processed by Subsystems.

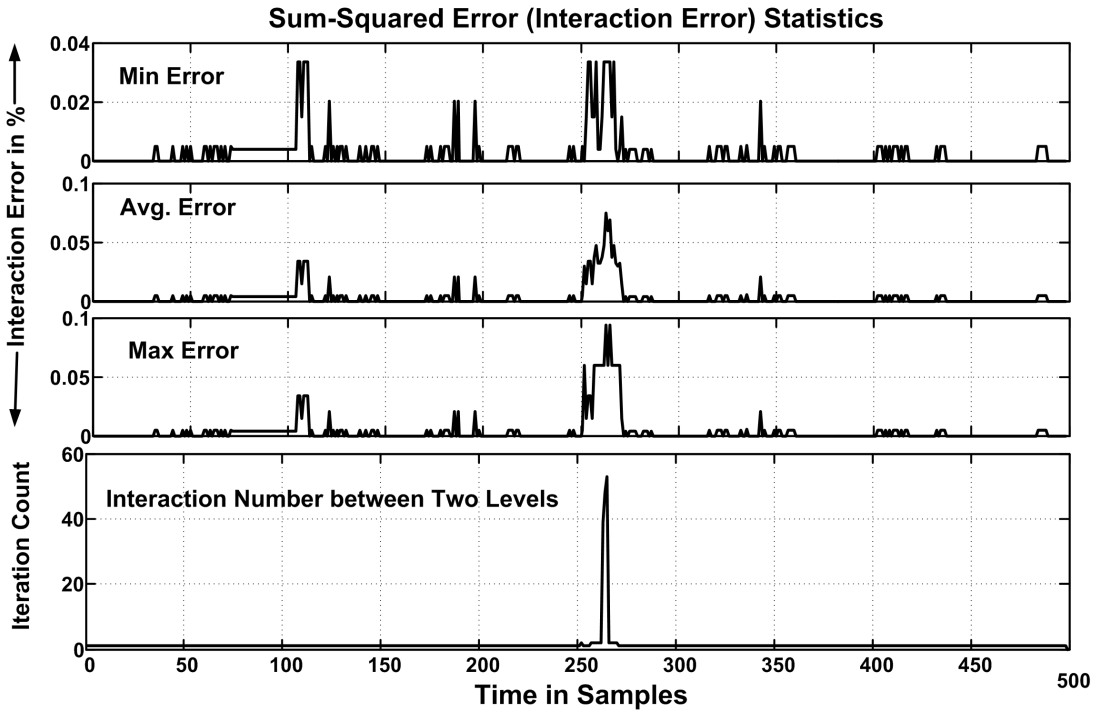


Figure 6.25

Simulation-3: Global Queue Size and Total Load share processed by Nodes.

$70 - 110$. It again shows the impact of the workload arrival rate on the interaction error convergence rate as discussed in Section 6.6.5.

6.7 Summary

In this chapter, a distributed control-based management approach is presented that utilizes the interaction balance principles used for control of large scale systems. The proposed approach efficiently manages the service level agreements of a web service deployment by minimizing the response time and power consumption simultaneously. According to the various simulations presented in this chapter, the developed distributed control approach maintains the SLAs of the deployed web service while increasing the utility measurement of the deployment by 98% and reducing computational overhead compared to a centralized control approach. Additionally, the proposed approach shows robustness towards nodes' failures in the cluster. We also investigated possible methods for improving the performance of the proposed management approach by reducing the overheads and increasing the convergence rate towards the optimal control inputs.

CHAPTER 7

APPLICATION OF THE DISTRIBUTED CONTROL APPROACH AS A COMPONENT BASED DEPLOYMENT

In this chapter, a component-based design of the developed distributed control system is presented, where system administrators can design and configure the distributed performance management system according to the specifications of the deployed application. This chapter also introduces the component modeling tool (GME) and a universal data model (UDM) packages that provide a programmable interface to access component models. Additionally, the development, deployment, and configuration strategy for developing a component-based control structure is introduced in this chapter. The proposed strategy is then applied for designing a distributed performance management system for a web service deployed in distributed environment.

7.1 Preliminaries

7.1.1 Generic Modeling Environment

The Generic Modeling Environment (GME) [14] is developed at Institute of Software Integrated System (ISIS), Vanderbilt University. *GME is a windows-based, domain-specific, model-integrated program synthesis tool for creating domain-specific and multi-aspect models of large-scale engineering systems* [14]. GME is designed to work with

various different engineering domains. GME uses concepts of hierarchy, multiple aspects, sets, references, and constraints to develop large scale complex system models [105]. Application model design in GME can be performed in the following two stages:

1. *Meta-Model Development:* At this stage, domain architects design the basic meta-model of the system, various modules inside the system, their connectivity, constraints, and visualization aspects.
2. *Application Model Development:* At this stage, domain engineers of the application can utilize the meta-model created in the previous step for defining the deployment plan or simulation configuration of the application. Domain engineers can utilize the meta-model just by knowing the connectivity pattern of the application, while incorrect connections can be restricted (or validated) through the constraints specified in the meta-model itself.

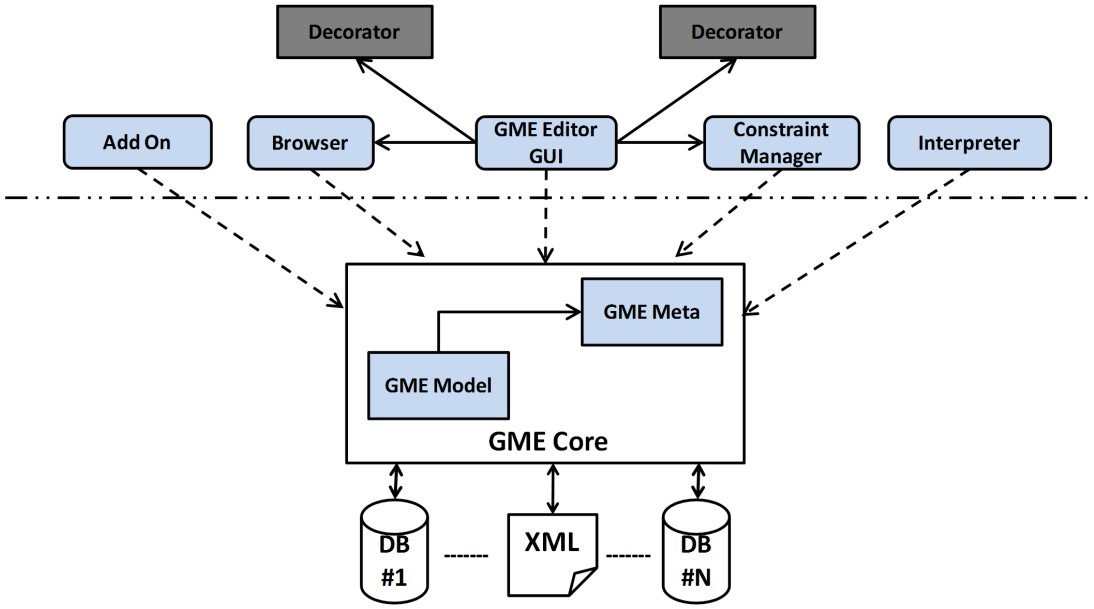


Figure 7.1

GME Architecture.

GME has a component based architecture as shown in Figure 7.1 [14]. According to Figure 7.1, GME architecture has three main components:

1. *Storage*: This layer contains multiple components related to the various storage formats. Currently, only binary and XML based storage are supported.

2. *GME Core*: This layer contains two components. *GME Meta*, which defines the modeling paradigm for target application domain and *GME Model*, which implements the modeling concepts of the given modeling paradigm for a specific application deployment setting.

3. *User Interface*: This layer represents the components of GME that interacts with users. *Add-Ons* represent different events during the editing process, while *Constraint Manager* and *Interpreter* are responsible for checking constraint validation and interpreting the model respectively. *GME Editor* and *Browser* are used for visualizing and creating the models.

GME contains multiple modeling concepts to create meta-model of engineering systems. These modeling concepts can be listed as follows:

- *Folder:* Folders contain different sections of the modeling project, which are related logically. Each project has at least one folder, which is named as the *Root* folder.
- *Models*: A model is an object that contains other objects or elements, which can be manipulated if required.
- *Atom*: An atom is the most basic and elementary object, which can not contain another object in it. It is used to represent small entities that cannot be divided further into smaller atoms.
- *Model Hierarchy*: It represents the containment relationship among the objects. According to this, an object must have one parent, which should be another model.
- *Aspects*: An aspect is defined by the parts of the model, either visible or hidden in it.
- *Connection*: A connection expresses the relationship between the objects within the same model.
- *References*: A reference expresses the relationship between objects at different levels or in different models. It is similar to the concept of pointers in programming languages, such as C or C++.
- *Set*: A set represents the relationship among a group of objects in the same folder or the same aspect. Each object can belong to one or multiple sets.
- *Attributes*: An attribute shows the property of an object as *test, integer, double, boolean* and *enumerated*.

- *Constraints*: Constraints represent the rules specific to the model composition and the project specifications.

For an application domain, the model paradigm is created as a composition of the above listed modeling concepts.

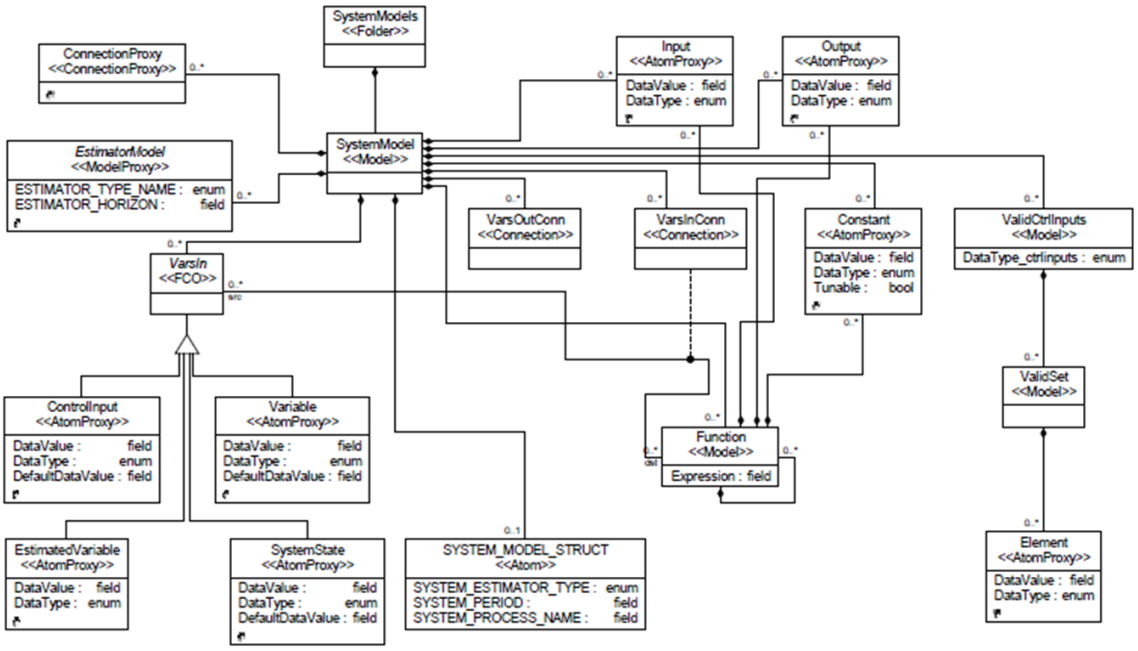


Figure 7.2

Computing System Dynamics Meta-Model in GME.

The domain-specific modeling toolkit *GME* [14, 105] is utilized earlier for creating application domain specific visual modeling environments. In GME, an application domain specific deployment model can be configured and developed by using the meta-level model specification for the same domain. The meta-modeling paradigm contains all the semantic, syntactic, and presentation information for the family of application. The GME can be

utilized by an administrator to create a different deployment settings for the application of the same family in a visual modeling environment. Furthermore, GME components are developed to interface with various simulation tools (MATLAB), configuration files (XML), or code generation utilities (Microsoft Visual Studio) for different programming languages (e.g., C++, Visual Basic, C#, Python etc.) per the desired output format.

In this dissertation, the meta-models of various components and the modules of the distributed control structure are developed by using GME. These modules are deployed in distributed manner to function as distributed control structure. An example of the computing system meta-model is presented in Figure 7.2.

7.1.2 Universal Data Model

The universal Data Model (UDM) package [36] provides programmable access to the GME domain models. The UDM package is used to generate a C++ programming interface from the GME meta-models. These programming interfaces are utilized by the domain engineers for interpreting the developed application domain models and generating either the dynamic linked libraries (*.dll and *.so) or the programs to generate the source code. UDM package is best suitable for cases when the data is either represented through object oriented design or through xml format. The primary components of the UDM package are described in Figure 7.3 [113].

The UDM framework consists of five main components:

1. *GME/UML Interpreter*: It is used to generate XML data files from the GME meta-models developed by domain architects.

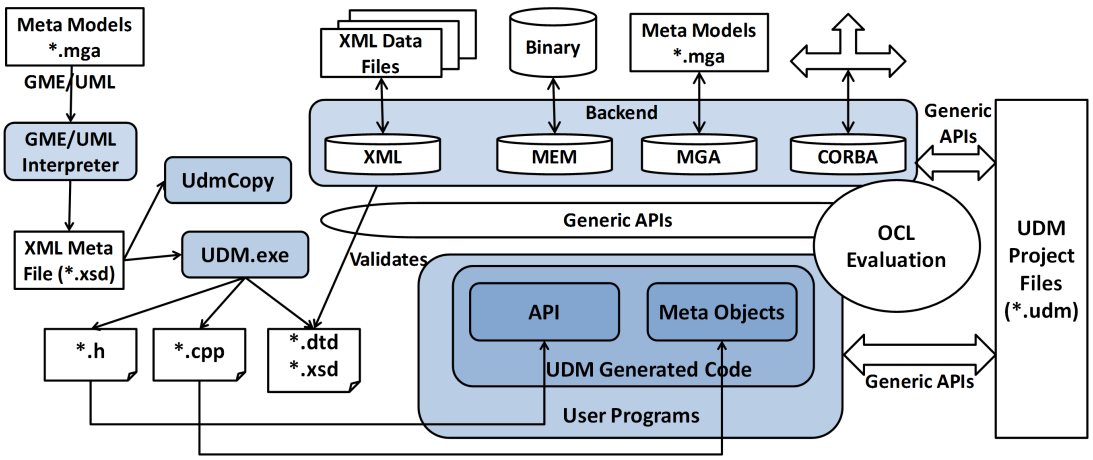


Figure 7.3

Components of the UDM Framework.

2. *UDM Programs*: These programs convert generated XML files to the meta-model dependent interface files of UDM. These files include C++ header file, C++ source file, XML document type definition file, and XML schema definition file.

3. *UDM Headers and Libraries*: These headers and libraries are linked to the user programs for utilizing UDM data types and utility programs.

4. *UDM Utility Programs*: These programs are used to manipulate the UDM data.

5. *UDM Back End*: UDM back end contains various generic APIs, which are used by the UDM programs. These generic APIs implement model objects in various forms: in memory (MEM), in XML dom tree (XML), as GME objects (MGA), and as generic binary objects (CORBA).

Programming APIs for GME meta-models are generated from the UDM package in the following steps.

1. A UML based meta-model is created in a GME/UML environment.

2. The UML meta-model information is converted to an XML file using the GME interpreter supplied with the GME/UML environment.

3. The generated XML file is passed as input to the *UDM.exe* program in order to generate the C++ header and the API files.

4. The users create a project in *Microsoft Visual Studio* and include these header and API files with other UDM header files.
5. The users develop an interpreter source file in C++ to read a specific domain model (objects) of the GME meta-model. Any changes in the GME meta-model requires the execution of all of the previous steps again.

In this dissertation, the UDM package is utilized for generating a C++ programmable interface to access the model objects in the GME application domain model related to the distributed control approach. These C++ interface files are further used to create and configure the control structure with deployment configurations related to each node.

7.2 Component-Based Control Structure: Development, Deployment, and Configuration

The primary idea of developing a component-based control structure is to divide a controller into very small reusable components that can be again assembled together by using interconnections and interaction rules to form the same controller. Furthermore, these components can be developed by using different approaches, rearrangements, and reconfigurations to create a controller with the same functionality. Moreover, small parts of the controller (assembly of a few small components) can also be reused for a different functionality by rearranging the components present inside it. However, the specifications of external ports, which are connected to other parts of the controller will remain unchanged.

For the deployment of the component-based control structure, each small component should be first developed, then connected with other components, and finally configured according to the deployment requirement. In the case of distributed controllers, interaction rules, network interfaces, and communication protocols should also be specified in the

controller at each node. A similar idea has already been implemented for deployment and configuration of component-based distributed applications [10].

A component-based control structure deployment strategy is proposed in this dissertation by combining GME and UDM for designing the distributed control structure deployment plan. This development, deployment, and configuration activity is shown in Figure 7.4. This entire process is completed in three different phases, which are described in following subsections.

7.2.1 Development Phase

In the development phase, the control structure is developed by using the meta-model specifications (provided from the system architects) related to the various components and components' assembly to perform a specific function. At this phase, these small components are developed in a generic manner to process different types of inputs. These components can be reconfigured (or rearranged) as a module to perform different functions. This development phase consists of following four tasks:

7.2.1.1 Meta-Model Development

The meta-model development for the control structure is performed by the domain architects of the application domain. This process is performed in a top down approach, where a large distributed control system functionality is decomposed hierarchically in their sub-functionalities executing at each subsystem. By decomposing the functionality, the control structure is also decomposed into an assembly of control functions and further each control function is decomposed into an assembly of small components upto the smallest

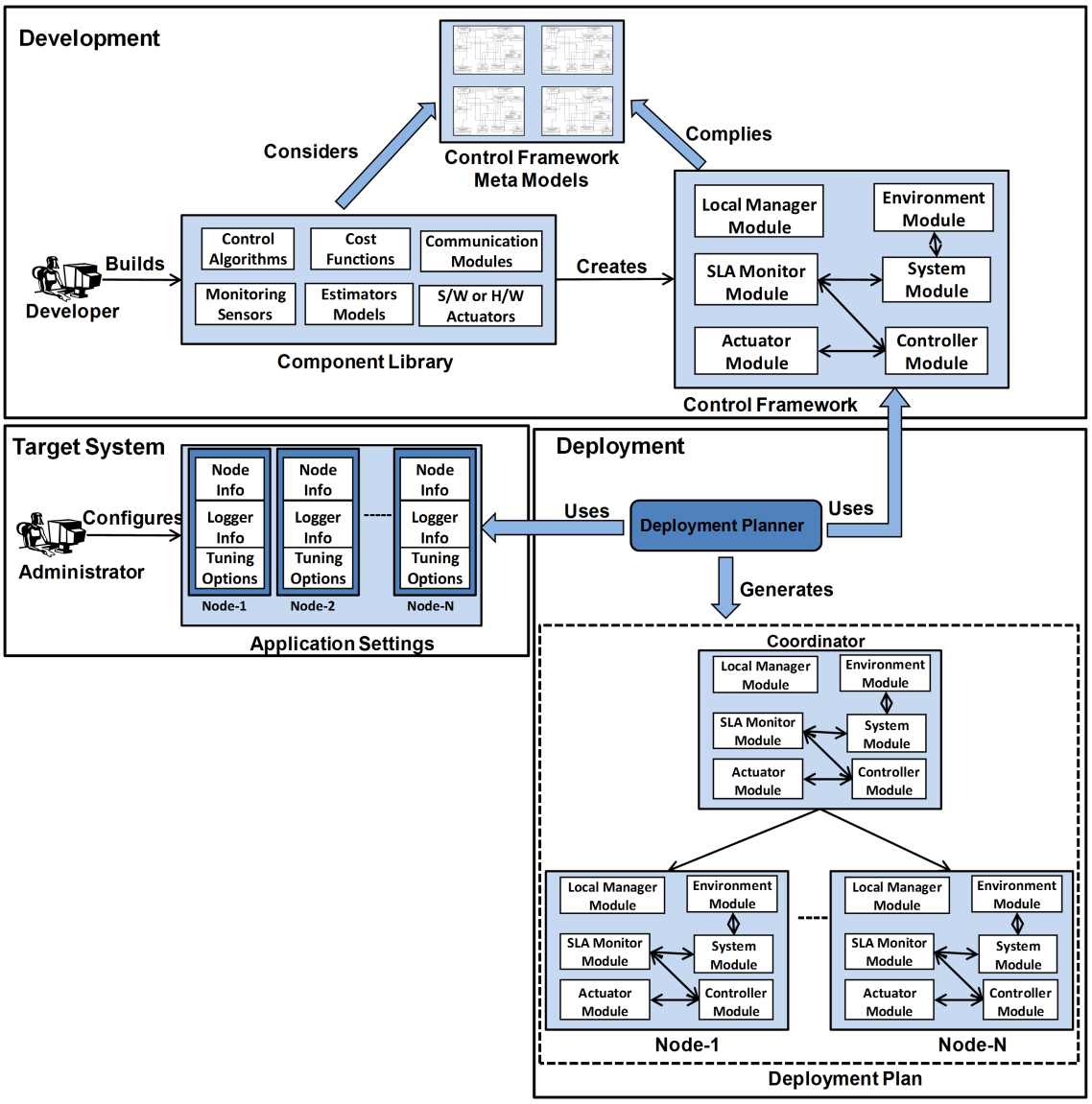


Figure 7.4

Component-Based Control Structure Deployment Plan Generation Process.

function possible. At this stage, the interaction rules and interconnects between different components are expressed in the meta-model specifications. In the case of distributed controllers, the network interface and communication protocol options are specified in a generic manner, independent from the actual method used in final deployment activity.

7.2.1.2 Interface Definition File Creation

The interface definition files [28] define the data types of different attributes of each component and the information variables shared between the components in a manner independent of the implementation language (e.g., C, C++, Java). These interface definition files are written in interface definition language (IDL). IDL files use various common data types (long, double, boolean, sequence, etc) and user defined data types (e.g., enum, struct). These data types are shared on an interconnection among multiple components (or nodes), where each side of the interconnect may or may not be implemented in the same programming language.

These IDL files are created manually by using meta-model of various components developed in the previous task. Additionally, IDL files contain implementation language specific bindings, which are generated by using IDL compilers. These IDL compilers translate the IDL file to the implementation language specific headers, which are included by the developers in order to implement a component in a specific programming language. A sample IDL file is shown in Figure 7.5.

```
00001:
00002: /* Sample IDL File and Struct  for Measurements from Sensors */
00003:
00004: module LQCD
00005: {
00006:     struct SENSOR_RECORD
00007:     {
00008:         unsigned long long UNIX_EPOCH;
00009:         string DEVICE_ID;
00010:         double VALUE;
00011:     };
00012:     struct MONITORING_INFO
00013:     {
00014:         string NODE_NAME;
00015:         string SENSOR_NAME;
00016:         string UNITS;
00017:         long long SEQUENCE_ID;
00018:         sequence<SENSOR_RECORD> RECORDS;
00019:     };
00020:     #pragma keylist MONITORING_INFO NODE_NAME SENSOR_NAME
00021: }
00022:
```

Figure 7.5

Example of an Interface Definition Language (IDL) File.

7.2.1.3 Component Specification and Development

In this task, system developers implement each component in a high level programming language according to the specifications in meta-models and by using the IDL files generated in the previous task in order to form a component library. A component can also be developed using multiple approaches according to the specifications provided from the meta-model. The control structure can choose among these implementations according to the QoS requirements from the deployed controller or node specific requirements. For example, an environment estimator can be developed using Kalman and ARIMA filters. During this task, different small components are also developed to perform very generic operations, such as sorting, searching, estimating, and filtering.

7.2.1.4 Component Packaging as Module and Framework

In this task, different components are grouped together and connected among themselves to perform a meaningful functionality. These connections and the assembly of components are performed in accordance to the meta-model specifications. These components can be grouped in two ways: Monolithic (compiled code) and assembly (group of components). In monolithic implementations, packages are created as dynamic linked libraries and cannot be divided further into components for rearrangement. In assembly implementations, each component inside the group can still be re-arranged or re-configured according to the requirement. These groups of components form a module, which is deployed at each computing node with a specific configuration according to the deployed application.

7.2.2 Target System Configuration Phase

The target system is considered as the application, which needs to be controlled through control structure at each node. In this phase, deployment information of each instance of the application at each computing node is configured in an "*Application Settings*" structure, which is developed by the system architects during the development of the meta-model of the control structure. This deployment information of the application include the physical node information (cluster name, node name), virtual node information (host VM node), system utilization logging information (type of system resource with their corresponding logging mechanism), application performance logs (performance log file or message queue), and system performance tuning options (CPU frequency, CPU cap, etc). In addition to this information, the communication methods available between the computing

nodes should also be specified in the application setting. These communication methods will be utilized by the control structure to communicate among the nodes.

7.2.3 Deployment Phase

After the development and packaging of the control structure, it is used for the deployment at each node according to the application settings by using a deployment plan. This deployment plan is generated by a "*Deployment Planner*" module as shown in Figure 7.4. This complete deployment activity is performed in two steps: Deployment Plan Generation and Deployment Preparation.

7.2.3.1 Deployment Plan Generation

During the deployment plan generation, application deployment settings and control structure is compared for matching and creating a configuration plan that specifies the placement of different modules of the control structure at each node and their interaction with the deployed application. Moreover, the deployment planer generates the configuration plan for each component in the control structure according to the specifications of the application settings. These settings include logs for measuring the system performance, control input options according to the available tuning options of the computing node, and communication module related information. In case of multiple implementations of a component in the component library, the most appropriate implementation is used for the deployment plan per application deployment setting and desired QoS specifications.

7.2.3.2 Deployment Preparation

After generating the deployment plan, the control structure code is moved to the actual computing node, configured, and launched in parallel to the application. This deployment activity takes place in the following steps:

- *Preconditioning of the Deployment:* In this step, the packaging of the control structure as different modules is performed according to the application deployment requirements. This packaging activity follows the constraints specified by the control structure meta-models.
- *Installation of the Control Structure:* At this step, the source code (or compiled binary) of the control structure module is moved to the actual computing node and placed inside the appropriate directory. In the case of the source codes, they need to be compiled after setting up the required environment variables.
- *Pre-configuration:* At this step, the default configurations of the computing nodes are performed according to the application deployment settings or control structure specifications. This activity can be performed manually or can be left for the control structure once it launches.
- *Control Structure Launch:* As a final step of the deployment, the control structure on each node is launched by using NFS mounted network scripts.

This component-based approach is applied for developing a distributed control structure for managing performance objectives of a distributed web service deployment.

7.3 Case Study: Distributed Control Structure Design and Deployment by using Component-based Approach

The distributed control structure developed in Chapter 6 is combined with the component-based approach developed in Section 7.2 of this chapter for managing distributed web services deployed in a virtualized environment. The distributed web service considered here, contains performance specifications similar to "Daytrader", which was used in Chapter 3 of this dissertation. This distributed control structure is designed as follows:

- Subsystem level controllers and regional coordinators are decomposed into multiple modules based on their functionality. These modules are further decomposed into small components with special functions. Additionally, some components are identified, which require multiple implementations. For example, estimator components are developed using ARIMA and Kalman filter approaches.
- Meta-models for each module of the controller and their components are developed by using GME. The interaction among these components as well as among modules are also specified at this step. In addition to this, meta-models for the application deployment configurations are also developed.
- Each of these components are developed in a high level programming language (preferably C++).
- The domain application model of each module from the control structure is developed by using GME as shown in Figure 7.6.
- Application deployment settings are configured in the domain application model of the deployment record structure.
- C++ code for APIs are generated by using GME meta-models with UDM packages. Additionally, the deployment planner module is also developed by using C++ header APIs generated from UDM. The deployment planner module uses these APIs to interpret the application models.
- The deployment planner module is executed to generate a deployment plan with the source code of various modules developed in the distributed control structure by using the control structure application model and application deployment settings.
- The distributed control structure is deployed after performing pre-deployment tasks and by using NFS mounted scripts.

All of these tasks are described in detail in the following subsections.

7.3.1 Control Structure Component Design

The subsystem controller described in Chapter 6 (Section 6.6.1) is decomposed into different modules based on their functionalities for developing a component based control structure. These modules and their different components are shown in Figure 7.6 and Figure 7.12.

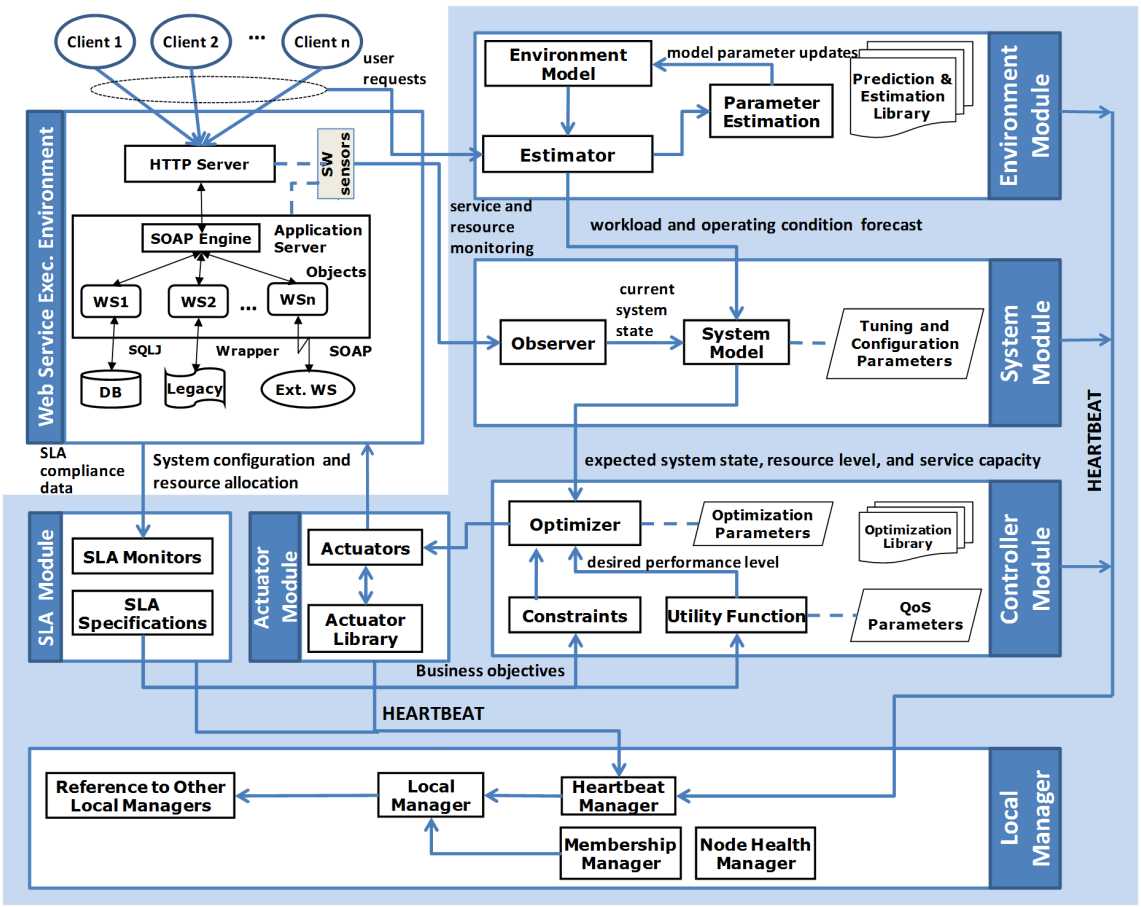


Figure 7.6

Key Components of the Control Framework at each Node.

7.3.1.1 Environment Module

The environment module captures the incoming workload from the end users toward the web service deployment at each node. This module receives the workload arrival rate by using environment input sensors and uses the environment model coupled with estimation libraries to estimate the expected environment input for the future. This estimated value is forwarded to the *System Module*.

7.3.1.2 System Module

The system module captures the dynamics of the deployed web service with respect to the system resources. The system module executes sensors related to system resources (CPU, disk, memory, etc.) and hardware health (motherboard, CPU fan, etc). This module captures the current system resource utilization level and future availability of the system resources. All of this information is transmitted to the *Controller Module*.

7.3.1.3 SLA Module

The SLA module keeps track of the SLA compliance for deployed application at the node. It contains information of the SLA between the service provider and service user in the distributed environment. It periodically sends the current SLA compliance to the controller module. Furthermore, this module is also responsible for updating the new SLA compliance policies on the controller module.

7.3.1.4 Controller Module

The controller module receives the SLA compliance information from the SLA module and system resource level from the system module. In the case of an SLA violation (or possible violation), the controller module computes the optimal control input from available control options at the node, which will ensure SLA compliance on the node and minimizes the cost of system operation through utility function. The controller module also contains a copy of the system model dynamics, system state, control input constraints, and search optimization library. The controller module forwards the updated control input value to the actuator module.

7.3.1.5 Actuator Module

The actuator module contains an actuator library that consists of various actuators developed for each type of tuning options available at the node. These tuning options include CPU core frequency, CPU cap share, load balance ratios, and virtual machine (on/off/pause/suspend) commands.

7.3.1.6 Regional Coordinator

The regional coordinator module is executed at one of the nodes in the region (or cluster) of N nodes. Regional coordinator is chosen as the leader of the region (see Section 5.4.4 for more details on the regional leader). Here, regional coordinator coordinates among the nodes to calculate the solution of the infrastructure level optimization problem as described in Section 6.4 of this dissertation.

7.3.1.7 Membership Manager

The membership manager module is executed to keep track of the regional coordinator of the region. Various features and functions of the membership manager are already described in Section 5.4.7 of this dissertation.

7.3.1.8 Local Manager

The local manager module is executed at each node for two purposes: First, it keeps track of all of the other modules with their HEARTBEAT information. If any of these modules terminate due to fault, local manager will restart another instance of that module as soon as it discovers the failure. Second, it maintains connection with the membership manager module to keep track of the regional coordinator.

7.3.2 GME Meta-Model Development

In this step, meta-models of different modules and their components are developed in GME. These meta-models contain information related to different components of the modules, input/output ports, communication modules, sensors started by the modules, user defined functions, module level constants, and the actuators if any. In addition to components, meta-models for various estimators, actuators, sensors, and communication models are also developed. For example, meta-models of the controller module and its component (local controller) are shown in Figure 7.7 and Figure 7.8. According to Figure 7.7, the controller module contains an instance of local controller, multiple sensors, multiple communication modules to interact with other modules, user defined functions, a copy of the current system state, and input/output ports. Similarly, according to Figure 7.8, the local

controller contains the controller information structure, the node deployment information, different utility information maps, SLA constraints, the system model reference, and the system state.

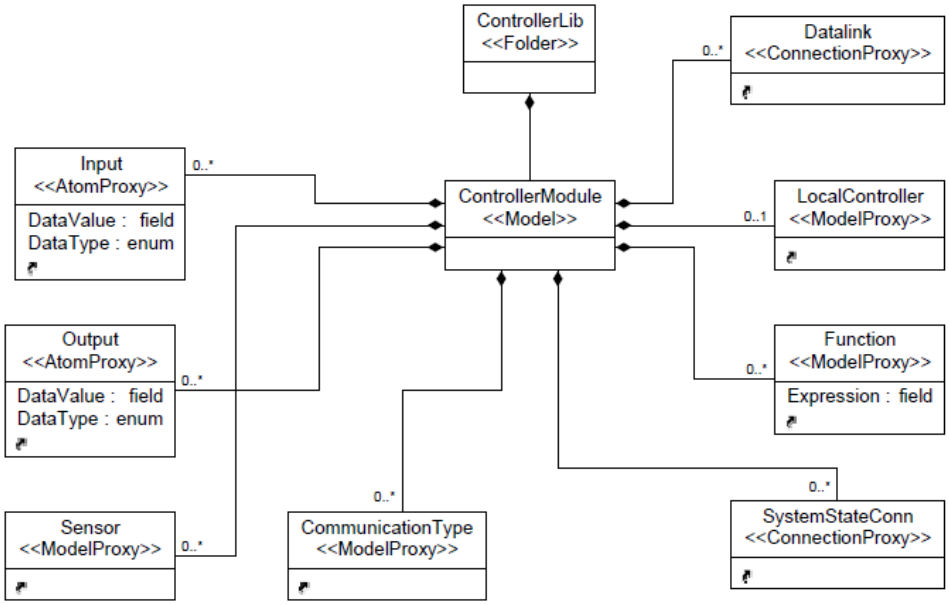


Figure 7.7

Controller Module Meta-Model in GME (see Figure 7.6).

7.3.3 Component Library Development

At this step, different components of the control structure are developed in a high level programming language (C++). These components are developed using specifications in their meta-models. Various types of monitoring sensors, actuators, communication modules, estimators, and optimization functions are implemented at this step. Estimators and optimization library components contain multiple implementations of the same function-

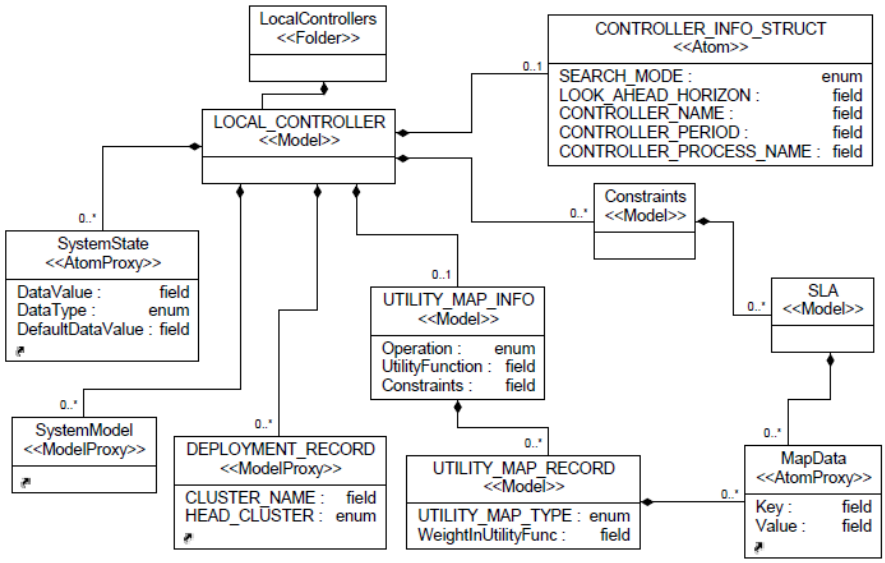


Figure 7.8

Local Controller Meta-Model in GME. (Present Inside Controller Module Meta-Model in Figure 7.7).

ality, which provide flexibility to the administrators to choose the suitable implementation according to the deployed application dynamics and computational requirements. For example, estimators contain both ARIMA and Kalman filter implementations, while the optimization function contains exhaustive, greedy, A*, and pruning tree search methods for computing optimal control inputs.

In this dissertation, a component library is developed in C++ by using a template-based approach that provides the flexibility of processing user defined data types. Additionally, the monitoring sensors and actuators are developed as an ARINC-653 process, while communication modules are developed as OpenSplice DDS communication modules as

described in Chapter 5. However, these components can be implemented through other approaches too in future and added to the component library.

7.3.4 Control Structure Domain Application Model

At this step, the application model from the control structure meta-model is created as shown in Figure 7.6. This application model is created to mimic a representative deployment of the control structure at each node, regional coordinator, and membership manager. This application model contains information regarding the types of sensors each module contains, communication module configuration for interaction among the modules, different components, and user defined functions of the modules. For example, the domain application model of the environment module is shown in Figure 7.9. The environment module contains a "Sensor" and an "Environment Model" as shown in Figure 7.9. The "Environment Model" further contains an "Environment Model Structure" and an "ARIMA" estimator. "ARIMA" estimator further contains an "Estimate" function, previous data ($P1$ and $P2$), and history data ($h1$).

7.3.5 Application Deployment Configuration

At this step, the deployment configuration of the web service is configured in the application model of the "Deployment Record" as shown in Figure 7.4 (see Application Settings). These settings include the computing node related information (cluster name, node name), and various logging information related to the system performance, system health, and application performance logs. These settings also include the various tuning options (CPU frequency, CPU share, etc.), their values or valid range, and their target resource

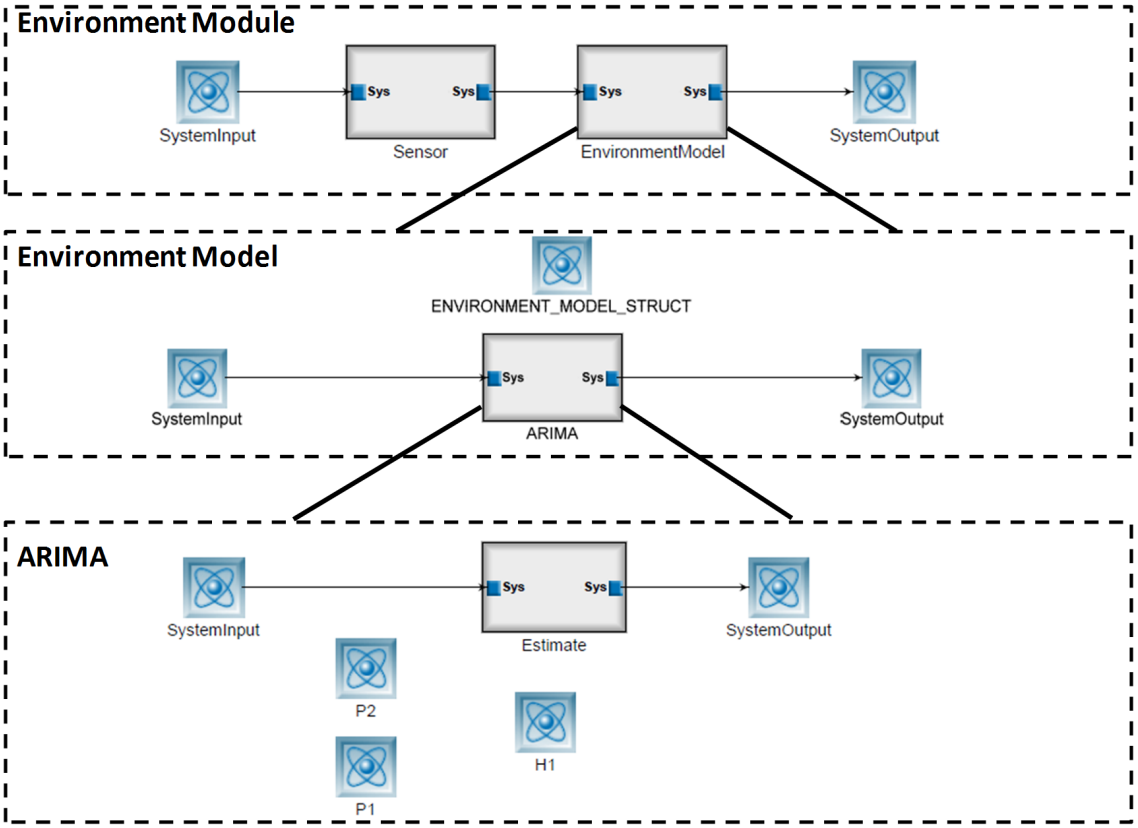


Figure 7.9

Domain Application Model of the Environment Module.

options (CPU core, VM node, etc). These application settings are used to configure and initialize various components of the control structure. This mapping of application settings with the control structure components is described in next few subsections.

7.3.6 Deployment Planner Module Development

The deployment planner module is developed in this dissertation by using the UDM package with Microsoft Visual Studio-2010 bindings. As described in Section 7.1.2, first C++ header and implementation APIs are generated through UDM package and meta-models of the control structure. These header and APIs are then included in a Microsoft visual studio project. These APIs provide programmable access to the application models of the distributed control structure and deployment configuration developed in previous Section 7.3.4 and Section 7.3.5, respectively. Another C++ source file, "*ControllerInterpreter*" is developed that interprets the application models using C++ APIs and UDM headers. This interpreter file can recursively read all of the object models, objects contained inside them, their different attributes, and the values of these attributes. A C++ code snippet of this interpreter source file is shown in Figure 7.10 and Figure 7.11. The procedure to generate the deployment plan is described in next subsection.

7.3.7 Deployment Plan Generation

During this step, application deployment settings are compared with the control structure application models and a deployment plan is generated, which specifies the placement of different modules of distributed control structure at each computing node and interaction

```
//////////////////////////////////////////////////////
// Main Function for Reading the Domain Application File  //
//////////////////////////////////////////////////////
int main(int argc, char *argv[])
{
  Udm::SmartDataNetwork dn(DDSController::diagram);   // INITIAL SETUP: UDM-specific
  dn.OpenExisting("DDSController_App.mga", "DDSController");  // Open the model
  DDSController::RootFolder rf = DDSController::RootFolder::Cast(dn.GetRootObject());  // Get the root folder
  cout << "Reading Global Deployment Record Info Start" << endl;
  // Get all of the Application Deployment from the root folder
  set<DDSController::ApplicationDeployment> applicationDeployment = rf.ApplicationDeployment_children();
  // Iterate over all of the ApplicationDeployments in the model
  if(applicationDeployment.size() > 0)
  {
    for(set<DDSController::ApplicationDeployment>::iterator it = applicationDeployment.begin(); it !=
applicationDeployment.end(); ++it)
    {
      DDSController::ApplicationDeployment TempApplicationDeployment = *it;
      CreateGlobalDeploymentRecord(rf, TempApplicationDeployment);
    }
  }
  cout << "Reading Global Deployment Record Info Finished" << endl;
  /* For each Node as configured from Deployment Configuration, Generate code for each Module Present in Root
Folder. Modules at each node will use their deployment Configuration from  map<std::string,
DDSController::DEPLOYMENT_RECORD> NodeDeploymentRecordMap as per their NodeName */
  cout << "Starting Code Generation" << endl;
  std::string DeploymentRecordDirectory("GlobalDeploymentRecord");
  ChangeDirectory(DeploymentRecordDirectory);
  for(map<std::string, DDSController::DEPLOYMENT_RECORD>::iterator itDeployment =
NodeDeploymentRecordMap.begin();   itDeployment != NodeDeploymentRecordMap.end(); ++itDeployment)
  {
    DDSController::DEPLOYMENT_RECORD TempDeploymentRecord = itDeployment->second;
    DDSController::APPLICATION_NODE ApplicationNode = TempDeploymentRecord.APPLICATION_NODE_child();
    std::string NodeName1 = ApplicationNode.NODE_NAME();
    cout << "!!!!! Node Name1 = " << NodeName1  << " !!!! " << endl;
    std::string NodeName = itDeployment->first;
    //Change Directory to ClusterName/NodeName
    std::string ClusterName = TempDeploymentRecord.CLUSTER_NAME();
    // cout << "Cluster Name" << ClusterName << endl;
    cout << " || Creating Code for Cluster Name: " << ClusterName << " , Node Name : " << NodeName << " ||" <<endl;
    ChangeDirectory(ClusterName);
    ChangeDirectory(NodeName);
    // Cretae Code for Membership Managers
    set<DDSController::MembershipManagers> membershipManagers = rf.MembershipManagers_children(); // Get
all of the state machines in the root folder
    // Iterate over all of the MembershipManagerss in the model
    if(membershipManagers.size() > 0)
    {
      for(set<DDSController::MembershipManagers>::iterator it = membershipManagers.begin(); it !=
membershipManagers.end(); ++it)
      {
        cout << "Creating Membership Manager at Node = " << NodeName << endl;
        DDSController::MembershipManagers TempMembershipManagers = *it;
        CreateCodeMembershipManagers(TempMembershipManagers, NodeName);
      }
    }
```

Figure 7.10

Code Sample from UDM-C++ Interpreter to generate the Source code with Deployment Settings for the Distributed Management System Deployment.

```
// Cretae Code for Local Managers
   set<DDSController::LocalManagers> localManagers = rf.LocalManagers_children(); // Get all of the
state machines in the root folder
   // Iterate over all of the Local Managers in the model
   if(localManagers.size() > 0)
   {
    for(set<DDSController::LocalManagers>::iterator it = localManagers.begin(); it !=
localManagers.end(); ++it)
    {
     cout << "Creating LocalManagers Manager at Node = " << NodeName << endl;
     DDSController::LocalManagers tempLocalManagers = *it;
     CreateCodeLocalManagers(tempLocalManagers, NodeName);
    }
   }
   // Cretae Code for Regional Coordinators
   set<DDSController::Coordinators> CoordinatorNodes = rf.Coordinators_children(); // Get all of the
state machines in the root folder
   // Iterate over all of the Regional Coordinators in the model
   if(CoordinatorNodes.size() > 0)
   {
    for(set<DDSController::Coordinators>::iterator it = CoordinatorNodes.begin(); it !=
CoordinatorNodes.end(); ++it)
    {
     cout << "Creating Coordinators at Node = " << NodeName << endl;
     DDSController::Coordinators TempCoordinators = *it;
     CreateCodeCoordinators(TempCoordinators, NodeName);
    }
   }
   std::string CdBackCommend("../..");
   ChangeDirectory(CdBackCommend);
  }
  std::string CdBackCommend("..");
  ChangeDirectory(CdBackCommend);
  std::string pwdCommand = "echo %cd%";
  system(pwdCommand.c_str());
  dn.CloseNoUpdate();
}
```

Figure 7.11

Code Sample from UDM-C++ Interpreter... (Continued from Figure 7.10).

of these modules with modules hosted at other nodes. This deployment plan is generated in following sequence:

1. *Application Deployment Settings*: Application deployment settings are read recursively for each node in each of the specified cluster and stored in the "Deployment Map" with "node name" as the key.

2. *Configuration of Control Structure Module for Deployment Settings*: For each node present in the "Deployment Map", each of the modules from the control structure is read and attributes from the "Deployment Map" are copied to the appropriate variables in control structure. For example, "Control Input" information for the node is copied from the "Deployment Map" to the controller module's "control input" data structure. Similarly, information regarding application performance logs is transferred from "Deployment Map" to the web application and SLA monitoring sensors in various modules.

3. *Configuration of Control Structure Modules*: Control structure modules at each node are configured according to the settings of the modules. At this point, if multiple implementations of the same component are present in the component library, the most suitable component implementation can be chosen per the application characteristic. For example, we chose the Kalman filter based implementation of the web application for estimating the web service system parameters, instead of the ARIMA filter based implementation.

4. *Code Generation*: The deployment planner generates the C++ source files that contain "Main" function for each of the modules in the control structure, and places them in folders arranged with the cluster name and the node name.

7.3.8 Control Structure Deployment

After generating the deployment plan with the configured code of the distributed control structure, it is deployed over each of the nodes that host an instance of the web service application. This deployment process is performed in following sequence:

1. *Installation*: At first, the source code (and compiled libraries) of the distributed control structure are moved to the respective computing nodes and placed under the appropriate directories.

2. *Pre-Configuration*: The computing nodes are configured according to the default configuration of the application deployment and control structure. In this dissertation, this step is used to set up appropriate DDS communication domains. In the case

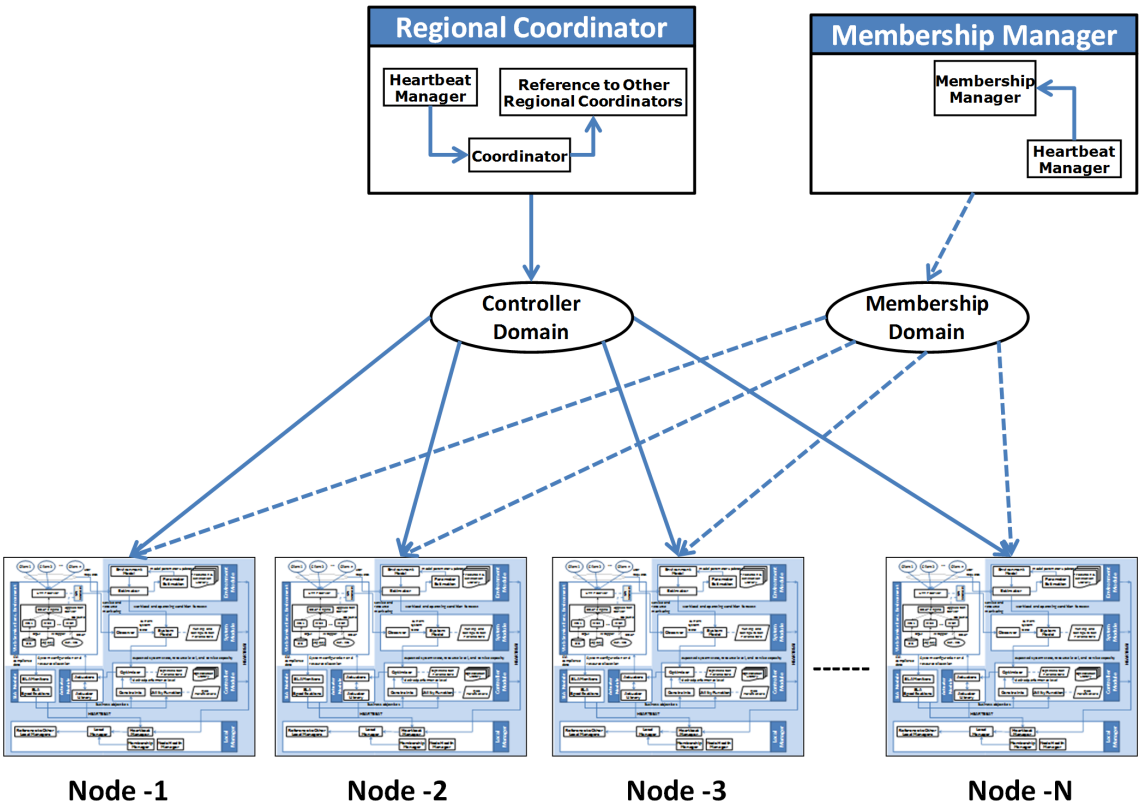


Figure 7.12

Hierarchical Arrangement of the Distributed Performance Management System Deployment.

of control structure related settings, this step can either be performed manually by the administrator or control structure modules can perform this procedure according to the initial values of the attributes.

3. *Control Structure Launch*: Finally, the distributed control structure is launched at each node by using NFS mounted scripts. After the launch, the control structure comes online, and configures itself for communication among modules at each node and with controllers executing at the other nodes. In addition to this, monitoring sensors start monitoring the different system parameters, application performance, and SLA compliance of the web service.

A schematic diagram of the developed distributed performance management system is shown in Figure 7.12. According to Figure 7.12, web service is deployed in a cluster of N computing nodes (Node-1 to Node-N). These computing nodes cooperate with each other through regional coordinator to compute the optimal values of control inputs according to the distributed control algorithm described in Chapter 6. Additionally, a membership manager is also deployed in the region to track the availability of regional coordinator according to the procedure described in Chapter 5.

7.4 Summary

In this chapter, a component-based design of the distributed control structure is presented based on model integrated computing practices. Detailed information of the generic modeling environment (GME) and the universal data model (UDM) package are also introduced in this chapter. A case study of applying the developed component-based approach to a distributed web service deployment is presented in detail that includes meta-model development, domain model development, deployment plant generation, and distributed control structure launch on each node.

CHAPTER 8

CONCLUSIONS AND FUTURE RESEARCH

8.1 Conclusions

In this dissertation, we developed a model-based autonomic performance management system for managing distributed enterprise systems and the underlying web service. Model-integrated computing practices are used together with component-based approach for the development of a generic distributed control-based performance management system. The developed performance management system can be applied to a wide range of enterprise applications hosted in a traditional data center environment or in a cloud computing infrastructure. This is done in following steps: developing standard components for each functionality (monitoring or control) of the typical management approach, configuring the components according to the application settings, and deploying the components in a distributed environment according to the deployment settings of the application. This component-based approach facilitates the reusability of the developed monitoring sensors, control modules, control algorithms, and application performance models that increases the productivity of the researchers, while decreasing the application specific development.

This dissertation introduces a systematic approach for developing performance models of a multi-tier web service within a certain accuracy. According to a set of experiments, the developed web service performance model tracks the performance behavior of the web

service with high accuracy in dynamic operating environment. Additionally, the computational nature of the incoming http requests is also computed by using an exponential Kalman filter. This chapter also proposes a lookup table-based physical server power consumption model that uses CPU core frequency and CPU utilization as the key to determine the corresponding power consumption value.

The developed web service performance model is utilized by a model predictive controller for minimizing the power consumption and maintaining the QoS specifications of the web service deployed in a virtualized environment. According to the experiments, the predictive controller can enhance the system performance and achieve an 18% power saving while maintaining the response time per SLA specifications. Additionally, the developed lookup table-based power consumption model predicts the overall power consumption of the physical server with 95% accuracy. Furthermore, the experiments demonstrate that the developed predictive-controller coupled with application performance model has low overhead for CPU and memory resources.

A real-time and fault-tolerant distributed monitoring system "RFDMon" is also introduced, which uses data centric publish-subscribe mechanism for sharing the measurements. This monitoring system utilizes avionics operating system specifications for fault isolation and restricting computational resource utilization. "RFDMon" can monitor the system resources, hardware health, computing node availability, scientific application execution state, and web service performance in a comprehensive manner. Moreover, this monitoring system provides scalability in number of monitored nodes because it is based upon data centric publish-subscribe mechanism, which is scalable in itself. In this mon-

itoring system, new sensors can be added dynamically without restarting the monitoring system. This monitoring system is fault-tolerant with respect to the faults due to partial outages. It can also self-configure (Start, Stop, and Poll) the sensors and can be applied in heterogeneous cluster environment.

A distributed control-based performance management approach is developed that can manage a general class of web services deployed in the distributed computing environment. The applicability of the developed management approach is demonstrated by applying it on the web service hosted in a distributed environment for power and response time management. According to the experimental results, this approach manages the web service within SLA requirements and minimizes power consumption. Additionally, this nodes provides scalability for increasing the number of levels in hierarchical arrangement of computing nodes hosting application instances. Moreover, this approach is fault-tolerant with respect to the computing node failures in the deployment.

The distributed performance management system is designed as a component-based performance management system by applying model integrated computing (MIC) methodologies. MIC tools (GME and UDM) are utilized for developing the components of the distributed control structure in a modular manner and generating the deployment plan of the distributed control structure. The distributed control structure components are configured, deployed, and launched according to the application deployment settings specified by the system administrator.

8.2 Future Research Directions

This dissertation has presented a comprehensive step by step approach for developing an autonomic performance management system for enterprise applications and web services hosted in a distributed environment. This performance management system is developed by using very general design concepts to extend its applicability for a wide range of systems and applications. We demonstrated the applicability of developed management system to a specific web service deployment. However, the research contributions of this dissertation can be further extended as follows:

8.2.1 Extended Component Library

In this dissertation, a large number of sensors, control algorithms, and estimators are developed as part of the component library. However, these component implementations are the ones applied to a limited class of web services utilized during the dissertation. This component library can be enriched by developing new sensors, control algorithms, and estimators to extend the coverage of the developed approach for a larger set of web services systems and other applications.

8.2.2 Fault Diagnosis Module

The developed monitoring system "RFDMon" helps in visualizing the system resource utilization, hardware health, and scientific application process state on various computing nodes. Therefore, an administrator can easily find the location and possible causes of the faults in the infrastructure. To make this fault identification and diagnosis procedure

autonomic, a fault diagnosis module can be developed that can detect or predict the faults in the infrastructure by observing and correlating measurements from various sensors.

8.2.3 Multi-Level Distributed Control Approach

The distributed control structure, developed in this dissertation, can be extended to a multi-level hierarchical control structure by using the same interaction balance approach, where computing nodes are arranged in a hierarchical manner consisting of multiple levels for efficient management of the application instances. This efficient management will be performed by decomposing the overall control problem of managing the infrastructure into various different control problems of managing different modules, and then further managing the application instances at each computing node. Node level controllers solve each subsystem problems while satisfying the constraints posed by the controller working at immediate higher level in the hierarchy.

REFERENCES

[1] "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/ec2/.

[2] *An architectural blueprint for autonomic computing*, Tech. Rep., http://www.ginkgo-networks.com/IMG/pdf/AC_Blueprint_White_Paper_V7.pdf [Mar 2012].

[3] "AutoAdmin," http://research.microsoft.com/en-us/projects/autoadmin/ [June 2013].

[4] "Autonomic Computing," http://www.research.ibm.com/autonomic/.

[5] "Catalog of OMG Data Distribution Service (DDS) Specifications," http://www.omg.org/technology/documents/dds_spec_catalog.htm [Nov 2010].

[6] "COAST (Computer Operations, Audit, and Security Technology)," http://www.cerias.purdue.edu/about/history/coast/ [Mar, 2012].

[7] "COM:Component Object Model Technologies," http://www.microsoft.com/com/default.mspx [July 2013].

[8] "CURL," http://curl.haxx.se/ [Nov 2011].

[9] "Daytrader," http://cwiki.apache.org/GMOxDOC20/daytrader.html.

[10] "Deployment and Configuration of Component-based Distributed Applications (DEPL)," http://www.omg.org/spec/DEPL/ [June 2013].

[11] "ebiquity," http://neuromation.blogspot.com/Neuro/neuro.htm [June 2013].

[12] "Enterprise Java Beans," http://www.oracle.com/technetwork/java/javaee/ejb/index.html [Mar 2012].

[13] "Ganglia," http://ganglia.sourceforge.net/ [Sep 2011].

[14] "GME: Generic Modeling Environment," http://www.isis.vanderbilt.edu/projects/GME.

[15] "Google Apps for Business," http://www.google.com/apps/intl/en/business/index.html.

[16] "GReAT: Graph Rewriting and Transformation," http://www.isis.vanderbilt.edu/tools/GReAT[June2013].

[17] "HP Adaptive Infrastructure," http://www.hpl.hp.com/research/.

[18] "Intel Proactive Computing," http://www.intel.com/content/www/us/en/research/research-areas/intel-labs-research-areas.htmls.

[19] "Intelligent Platform Management Interface (IPMI)," http://www.intel.com/design/servers/ipmi/ [Sep 2011].

[20] "Microsoft Dynamic Systems," http://research.microsoft.com/apps/dp/areas.aspx.

[21] "Microsoft .NET Framework," http://www.microsoft.com/net [Mar 2012].

[22] "Model-based Software Health Management," https://wiki.isis.vanderbilt.edu/mbshm/index.php/Main_Page [Nov 2011].

[23] "Model Integrated Computing," http://www.isis.vanderbilt.edu/research/MIC [July 2013].

[24] "Nagios," http://www.nagios.org/ [Sep 2011].

[25] "Neuromation: Smart thinking," http://neuromation.blogspot.com/Neuro/neuro.htm [June 2013].

[26] "Nimsoft Unified Manager," http://www.nimsoft.com/solutions/nimsoft-unified-manager [Nov 2011].

[27] "Object Management Group," http://www.omg.org/ [Nov 2010].

[28] "Object Management Group (OMG) IDL: Details," http://www.omg.org/gettingstarted/omg_idl.htm [June 2013].

[29] "OCL: Object Constraint Language 2.0," http://www.omg.org/spec/OCL/2.0/ [July 2013].

[30] "The Open Grid Services Architecture, Version 1.0," http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf [July 2013].

[31] "OpenSplice DDS Community Edition," http://www.prismtech.com/opensplice/opensplice-dds-community.

[32] "Ruby," http://www.ruby-lang.org/en/ [Sep 2011].

[33] "Ruby on Rails," http://rubyonrails.org/ [Sep 2011].

[34] "Service Level Agreement Life Cycle," http://publib.boulder.ibm.com/infocenter/sr/v6r3/index.jsp?topic=%2Fcom.ibm.sr.doc%2Frwsr_gep_sla_life_cycle.html [Mar 2012].

[35] "SUN N1 System Manager," http://docs.oracle.com/cd/E19957-01/819-5135/gbiij/index.html [June 2013].

[36] "UDM: Universal Data Model," http://www.isis.vanderbilt.edu/tools/UDM [June 2013].

[37] "UML: Unified Modeling Language," http://www.uml.org/ [July 2013].

[38] "Watts Up?," https://www.wattsupmeters.com/secure/products.php?pn=0&wai=456 [July 2013].

[39] "Web Services Architecture," http://www.w3.org/TR/ws-arch/ [Mar, 2012].

[40] "Web Services Description Language (WSDL) Version 2.0," http://www.w3.org/TR/wsdl20/ [Mar, 2012].

[41] "WebSphere Application Server Community Edition," http://www-01.ibm.com/software/webservers/appserv/community/.

[42] "Windows Azure," http://www.windowsazure.com/ [Mar, 2012].

[43] "XEN," http://www.xenproject.org/ [July 2013].

[44] "Zenoss," http://www.zenoss.com/ [Sep 2011].

[45] *Arinc specification 653-2 : Avionics application software standard interface part 1Required services*, Tech. Rep., Annapolis, MD, December 2005.

[46] *PLASMA: a component-based framework for building self-adaptive multimedia applications*. Proc. SPIE 5683, 2005.

[47] *httperf documentation*, Tech. Rep., HP, 2007.

[48] S. Abdelwahed, J. Bai, R. Su, and N. Kandasamy, "On the application of predictive control techniques for adaptive performance management of computing systems," *Network and Service Management, IEEE Transactions on*, vol. 6, no. 4, 2009, pp. 212–225.

[49] S. Abdelwahed, A. Dubey, G. Karsai, and N. Mahadevan, *Model-based Tools and Techniques for Real-Time System and Software Health Management*, CRC Press, to be published 2011.

[50] S. Abdelwahed, M. Hassan, and M. Sultan, "Parallel Asynchronous Algorithms for Optimal Control of Large-scale Dynamic Systems," *Optimal Control Applications and Methods*, vol. 18, July 1997.

[51] S. Abdelwahed, N. Kandasamy, and S. Neema, "Online Control for Self-Management in Computing Systems," *Proc. RTAS*, 2004, pp. 365–375.

[52] S. Abdelwahed, G. Karsai, N. Mahadevan, and S. Ofsthun, "Practical Implementation of Diagnosis Systems Using Timed Failure Propagation Graph Models," *Instrumentation and Measurement, IEEE Transactions on*, vol. 58, no. 2, feb. 2009, pp. 240 –247.

[53] T. Abdelzaher, K. Shin, and N. Bhatti, "Performance guarantees for Web server end-systems: a control-theoretical approach," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 1, Jan 2002, pp. 80–96.

[54] M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, and S. Hariri, "AutoMate: enabling autonomic applications on the grid," *Autonomic Computing Workshop. 2003. Proceedings of the*, june 2003, pp. 48 – 57.

[55] A. Agrawal, "Graph rewriting and transformation (GReAT): a solution for the model integrated computing (MIC) bottleneck," *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, 2003, pp. 364–368.

[56] M. Arlitt and T. Jin, *Workload characterization of the 1998 world cup web site*, Technical Report HPL-99-35R1, Hewlett-Packard Labs, September 1999.

[57] H. L. Asit Dan and G. Pacifici, "Web service differentiation with service level agreements," 2003, `http://www.ibm.com/developerworks/library/ws-slafram/[Mar2012]`.

[58] S. Balasubramaniam, K. Barrett, W. Donnelly, S. van der Meer, and J. Strassner, "Bio-inspired Policy Based Management (bioPBM) for Autonomic Bio-inspired Policy Based Management (bioPBM) for Autonomic," *Policies for Distributed Systems and Networks, 2006. Policy 2006. Seventh IEEE International Workshop on*, 2006, pp. 3 –12.

[59] F. Baude and V. Legrand, "A component-based orchestration management framework for multidomain SOA," *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, may 2011, pp. 1156 –1163.

[60] P. J. Bentley, "Towards an artificial immune system for network intrusion detection: an investigation of dynamic clonal selection," *Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress - Volume 02*, Washington, DC, USA, 2002, CEC '02, pp. 1015–1020, IEEE Computer Society.

[61] J. Bigus, D. Schlosnagle, J. Pilgrim, W. Mills, and Y. Diao, "ABLE: a toolkit for building multiagent autonomic systems - Agent Building and Learning Environment," *IBM Systems Journal*, vol. 41, 2002, p. 350.

[62] L. Bisdounis, S. Nikolaidis, O. Koufolavlou, and C. Goutis, "Modeling the CMOS short-circuit power dissipation," *Circuits and Systems, 1996. ISCAS '96., 'Connecting the World'., 1996 IEEE International Symposium on*, May 1996, vol. 4, pp. 469 –472 vol.4.

[63] M. B. Blake, A. B. Williams, and S. College, "A Component-Based Data Management and Knowledge Discovery Framework for Aviation Studies," .

[64] A. Cervin, J. Eker, B. Bernhardsson, and Karl-Erik, "Feedback–Feedforward Scheduling of Control Tasks," *Real-Time Syst.*, vol. 23, no. 1/2, 2002, pp. 25–53.

[65] H. Cha and I. Jung, "RMTool: Component-Based Network Management System for Wireless Sensor Networks," *Consumer Communications and Networking Conference, 2007. CCNC 2007. 4th IEEE*, jan. 2007, pp. 614 –618.

[66] S. L. Chung, S. Lafortune, and F. Lin, "Limited Lookahead Policies in Supervisory Control of Discrete Event Systems," *IEEE Trans. Autom. Control*, vol. 37, no. 12, Dec. 1992, pp. 1921–1935.

[67] A. Dabholkar, A. Dubey, A. Gokhale, N. Mahadevan, and G. Karsai, "Reliable Distributed Real-time and Embedded Systems Through Safe Middleware Adaptation," *31st International Symposium on Reliable Distributed Systems (SRDS 2012)*, Irvine, California, USA, 10/2012 2012, IEEE, IEEE.

[68] S. W. De Jong K.A. and G. D.F., "Machine Learning," 1993, pp. 161–188.

[69] Y. Diao, J. L. Hellerstein, S. Parekh, H. Shaikh, M. Surendra, and A. Tantawi, "Modeling Differentiated Services of Multi-Tier Web Applications," *MASCOTS*, vol. 0, 2006, pp. 314–326.

[70] B. Diniz, D. Guedes, W. Meira, Jr., and R. Bianchini, "Limiting the power consumption of main memory," *Proceedings of the 34th annual international symposium on Computer architecture*, New York, NY, USA, 2007, ISCA '07, pp. 290–301, ACM.

[71] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao, "Autonomia: an autonomic computing environment," *Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International*, april 2003, pp. 61 – 68.

[72] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat, "Model-based resource provisioning in a web service utility," *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, Berkeley, CA, USA, 2003, pp. 5–5, USENIX Association.

[73] A. Dubey, G. Karsai, and N. Mahadevan, "A component model for hard real-time systems: CCM with ARINC-653," *Software: Practice and Experience*, vol. 41, no. 12, 2011, pp. 1517–1550.

[74] A. Dubey, N. Mahadevan, and G. Karsai, "A Deliberative Reasoner for Model-Based Software Health Management," *The Eighth International Conference on Autonomic and Autonomous Systems*, St. Maarten, Netherlands Antilles, 03/2012 2012.

[75] W. B. Dunbar and R. M. Murray, "Distributed receding horizon control for multi-vehicle formation stabilization," *Automatica*, vol. 42, no. 4, 2006, pp. 549 – 558.

[76] D. Economou, S. Rivoire, and C. Kozyrakis, "Full-system power analysis and modeling for server environments," *In Workshop on Modeling Benchmarking and Simulation (MOBS*, 2006.

[77] G. Edwin and M. T. Cox, "COMAS: CoOrdination in MultiAgent Systems," 2001.

[78] B. F. Gerardo Pardo-Castellote and R. Warren, "An Introduction to DDS and Data-Centric Communications," 2005, `http://www.omg.org/news/whitepapers/Intro_To_DDS.pdf`.

[79] A. Goldberg and G. Horvath, "Software Fault Protection with ARINC 653," *Aerospace Conference, 2007 IEEE*, March 2007, pp. 1 –11.

[80] D. Gupta, R. Gardner, and L. Cherkasova, *XenMon: QoS Monitoring and Performance Profiling Tool*, Tech. Rep., HP Labs, 2005.

[81] A. D. R. P. K. Heiko Ludwig, Alexander Keller and R. Franck, "Web Service Level Agreement (WSLA) Language Specification," January 2001, `http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf/` [Mar 2012].

[82] M. G. Hinchey and R. Sterritt, "99% (Biological) Inspiration...," *Engineering of Autonomic and Autonomous Systems, 2007. EASe '07. Fourth IEEE International Workshop on*, 26-29 March 2007, pp. 187–195.

[83] F. S. Hofmeyr SA, "Immunity by Design: An artificial immune system.," *Genetic and Evolutionary Computation Conference (GECCO-1999).* 1999, Morgan Kaufmann.

[84] Z.-G. Hou, "A hierarchical optimization neural network for large-scale dynamic systems," *Automatica*, vol. 37, no. 12, 2001, pp. 1931 – 1940.

[85] Y. Hu, Q. Li, and C.-C. Kuo, "Run-time power consumption modeling for embedded multimedia systems," *Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on*, 2005, pp. 353 – 356.

[86] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing, degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, Aug. 2008, pp. 7:1–7:28.

[87] IBM Global Services, IBM *Global Services and autonomic computing*, Tech. Rep., http://www-03.ibm.com/autonomic/pdfs/wp-igs-autonomic.pdf, 2002.

[88] R. Joseph, M. Martonosidepartment, and E. Engineering, "Run-time power estimation in high performance microprocessors," *In International Symposium on Low Power Electronics and Design*, 2001, pp. 135–140.

[89] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Transactions of the ASME Journal of Basic Engineering*, , no. 82 (Series D), 1960, pp. 35–45.

[90] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters," *ICAC '09: Proceedings of the 6th international conference on Autonomic computing*, New York, NY, USA, 2009, pp. 117–126, ACM.

[91] S. Kamil, J. Shalf, and E. Strohmaier, "Power efficiency in high performance computing," *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1 –8.

[92] N. Kandasamy, S. Abdelwahed, and J. P. Hayes, "Self-Optimization in Computer Systems via Online Control: Application to Power Management," *Proc. IEEE Int'l Conf. Autonomic Computing*, 2004, pp. 54–62.

[93] N. Kandasamy, S. Abdelwahed, and M. Khandekar, "A hierarchical optimization framework for autonomic performance management of distributed computing systems," *Proc. 26th IEEE Int'l Conf. Distributed Computing Systems (ICDCS)*, 2006.

[94] G. Karsai, A. Ledeczi, S. Neema, and J. Sztipanovits, "The Model-Integrated Computing Toolsuite: Metaprogrammable Tools for Embedded Control System Design," *Proc. of the IEEE Joint Conference CCA, ISIC and CACSD, Munich, Germany*, October 2006, pp. 50–55.

[95] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, no. 1, 2003, pp. 145–164.

[96] A. Keller, J. Hellerstein, J. Wolf, K.-L. Wu, and V. Krishnan, "The CHAMPS system: change management with planning and scheduling," *Proc. IEEE/IFIP Network Operations and Management Symposium NOMS 2004*, 19–23 April 2004, vol. 1, pp. 395–408.

[97] D. G. Kendall, "Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain," *The Annals of Mathematical Statistics*, vol. 24, no. 3, 1953, pp. 338–354.

[98] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, January 2003, pp. 41–50.

[99] T. Keviczky, F. Borrelli, and G. Balas, "Hierarchical design of decentralized receding horizon controllers for decoupled systems," *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, 2004, vol. 2, pp. 1592–1597 Vol.2.

[100] T. Keviczky, F. Borrelli, and G. J. Balas, "Decentralized receding horizon control for large scale dynamically decoupled systems," *Automatica*, vol. 42, no. 12, 2006, pp. 2105 – 2115.

[101] T. Keviczky, F. Borrelli, K. Fregene, D. Godbole, and G. Balas, "Decentralized Receding Horizon Control and Coordination of Autonomous Vehicle Formations," *Control Systems Technology, IEEE Transactions on*, vol. 16, no. 1, 2008, pp. 19–33.

[102] L. Kleinrock, *Theory, Volume 1, Queueing Systems*, Wiley-Interscience, 1975.

[103] D. Kusic, N. Kandasamy, and G. Jiang, "Approximation Modeling for the Online Performance Management of Distributed Computing Systems," *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, 2007, p. 23.

[104] Z. Q. Z. W. R. P. D. Z. M. L. Zhang, B. Tiwana and L. Yang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, 2010.

[105] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The Generic Modeling Environment," *Workshop on Intelligent Signal Processing*, 2001.

[106] E. Lee, I. Kulkarni, D. Pompili, and M. Parashar, "Proactive thermal management in green datacenters," *The Journal of Supercomputing*, vol. 38, 2010, pp. 1–31, 10.1007/s11227-010-0453-8.

[107] J. Li, J. Han, Z. Li, and Y. Zhao, "SCENE admin: A component-based integrated management framework for web service platforms," *IT in Medicine and Education (ITME), 2011 International Symposium on*, dec. 2011, vol. 2, pp. 77 –81.

[108] W. Liao, L. He, and K. Lepak, "Temperature and supply Voltage aware performance and power modeling at microarchitecture level," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 7, 2005, pp. 1042 – 1053.

[109] H. Liu and M. Parashar, "Accord: a programming framework for autonomic applications," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 36, no. 3, may 2006, pp. 341 – 352.

[110] G. M. Lohman and S. S. Lightstone, "SMART: making DB2 (more) autonomic," *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*. 2002, pp. 877–879, VLDB Endowment.

[111] C. Lu, G. A. Alvarez, and J. Wilkes, "Aqueduct: Online Data Migration with Performance Guarantees," *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2002, p. 21, USENIX Association.

[112] J. M. Maciejowski, *Predictive Control with Constraints*, Prentice Hall, London, 2002.

[113] E. Magyari, A. Bakay, A. Lang, T. Paka, A. Vizhanyo, A. Agrawal, and G. Karsai, "UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages," *The 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003*, Anahiem, California, October 2003.

[114] N. Mahadevan, A. Dubey, and G. Karsai, "Application of Software Health Management Techniques," *To appear in the Proceedings of the 2011 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, New York, NY, USA, 2011, ACM, SEAMS '11, ACM.

[115] X. W. Matthias Eiblmaier, Rukun Mao, "Power Management for Main Memory with Access Latency Control," San Francisco, CA, USA., 2009, Febid '09, ACM.

[116] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "IBM Storage Tank– A heterogeneous scalable SAN file system," *IBM Syst. J.*, vol. 42, no. 2, 2003, pp. 250–267.

[117] M. R. Mostafa E. A. Ibrahim and H. A. H. Fahmy, "A Precise High-Level Power Consumption Model for Embedded Systems Software," *EURASIP Journal on Embedded Systems*, vol. 2011, 2010, pp. 1:1–1:14.

[118] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "CPU demand for web serving: Measurement analysis and dynamic estimation," *Performance Evaluation*, vol. 65, no. 6–7, June 2008, pp. 531–553.

[119] G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef, *Performance Management for Cluster Based Web Services*, Tech. Rep., IEEE Journal on Selected Areas in Communications, Volume 23, Issue, 2003.

[120] M. Parashar and S. Hariri, "Autonomic Computing: An Overview.," *UPP*, 2004, pp. 257–269.

[121] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, The Hardware/Software Interface, 4th Edition*, Morgan Kaufmann, 2008.

[122] C. Poellabauer, *Q-fabric: system support for continuous online quality management*, doctoral dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 2004, Adviser-Karsten Schwan.

[123] R. Pool, "Natutal Selection: A New Computer Program Classifies Documents Automatically," `http://domino.watson.ibm.com/comm/wwwr_thinkresearch.nsf/pages/selection200.html`, 2002.

[124] S. J. Qin and T. A. Badgwell, "A survey of industrial model predictive control technology," 2003.

[125] D. Rittman, "Power optimization within nanometer designs.," *System Design Frontier, Shanghai Hometown Microsystems Inc.*, May 2005.

[126] D. Roberts, T. Kgil, and T. Mudge, "Using non-volatile memory to save energy in servers," *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, 2009, pp. 743 –748.

[127] N. Roy, A. Dubey, and A. Gokhale, "Efficient Autoscaling in the Cloud using Predictive Models for Workload Forecasting," *To Appear in the 4th IEEE International Conference on Cloud Computing (Cloud 2011)*, Washington, DC, 07/2011 2011, IEEE, IEEE.

[128] L. W. Russell, S. P. Morgan, and E. G. Chron, "Clockwork: A new movement in autonomic systems," *IBM Syst. J.*, vol. 42, no. 1, 2003, pp. 77–84.

[129] S. Russell and P. Norvig, *Artifical Intelligence A Modern Approach*, Prenctice Hall, 1995.

[130] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2 edition, Prentice Hall, 2003.

[131] N. Sadati, "A novel approach to coordination of large-scale systems; part I interaction prediction principle," *IEEE Int'l Conf. on Industrial Technology*, dec. 2005, pp. 641 –647.

[132] N. Sadati, "A novel approach to coordination of large-scale systems; part II interaction balance principle," *IEEE Int'l Conf. on Industrial Technology*, dec. 2005, pp. 648 – 654.

[133] N. Sadati and G. A. Dumont, "A Reinforcement Learning Approach to Intelligent Goal Coordination of Two-Level Large-Scale Control Systems," *Advances in Reinforcement Learning*, P. A. Mellouk, ed., http://www.intechopen.com/books/advances-in-reinforcement-learning/a-reinforcement-learning-approach-to-intelligent-goal-coordination-of-two-level-large-scale-control.

[134] K. Shafi and H. A. Abbass, "Biologically-inspired Complex Adaptive Systems approaches to Network Intrusion Detection," *Inf. Secur. Tech. Rep.*, vol. 12, September 2007, pp. 209–217.

[135] N. Shankaran, J. S. Kinnebrew, X. Koutsoukos, C. Lu, D. C. Schmidt, and G. Biswas, "An Integrated Planning and Adaptive Resource Management Architecture for Distributed Real-Time Embedded Systems," *IEEE Transactions on Computers*, vol. 58, no. 11, 11/2009 2009, pp. 1485 –1499.

[136] J. Shi, R. Amgai, S. Abdelwahed, A. Dubey, J. Humphreys, M. Alattar, and R. Jia, "Generic Modeling and Analysis Environment Design for Shipboard Power System," *Marinelive*, 11/2012 2012.

[137] C. Sinclair, L. Pierce, and S. Matzner, "An application of machine learning to network intrusion detection," *Computer Security Applications Conference, 1999. (ACSAC '99) Proceedings. 15th Annual*, 1999, pp. 371 –377.

[138] M. S. Squillante, D. D. Yao, and L. Zhang, "Web traffic modeling and Web server performance analysis," *SIGMETRICS Perform. Eval. Rev.*, vol. 27, no. 3, 1999, pp. 24–27.

[139] W. Stallings, *Operating Systems: Internals and Design Principles*, 7 edition, Prentice Hall, March 2011.

[140] R. Sterritt and M. Hinchey, "SPAACE IV: Self-Properties for an Autonomous and Autonomic Computing Environment," *Engineering of Autonomic and Autonomous Systems (EASe), 2010 Seventh IEEE International Conference and Workshops on*, march 2010, pp. 119 –125.

[141] G. Tesauro, "Online Resource Allocation Using Decompositional Reinforcement Learning," *AAAI*, 2005, pp. 886–891.

[142] G. Tesauro, W. E. Walsh, and J. O. Kephart, "Utility-Function-Driven Resource Allocation in Autonomic Systems," *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, Washington, DC, USA, 2005, pp. 342–343, IEEE Computer Society.

[143] M. G. S. A. Titli;, *Systems: Decomposition, Optimisation, and Control*, Pergamon Press, 1978.

[144] W. Truszkowski, M. Hinchey, J. Rash, and C. Rouff, "NASA's swarm missions: the challenge of building autonomous software," *IT Professional*, vol. 6, no. 5, 2004, pp. 47 – 52.

[145] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, 2005, pp. 291–302.

[146] A. Verma, P. Ahuja, and A. Neogi, "Power-aware dynamic placement of HPC applications," *Proceedings of the 22nd annual international conference on Supercomputing*, New York, NY, USA, 2008, ICS '08, pp. 175–184, ACM.

[147] P. Volgyesi and A. Ledeczi, "Component-based development of networked embedded applications," *Euromicro Conference, 2002. Proceedings. 28th*, 2002, pp. 68 – 73.

[148] C. Wang, C.-J. Ong, and M. Sim, "Distributed model predictive control of dynamically decoupled linear systems with coupled cost," *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, dec. 2009, pp. 5420 –5425.

[149] M. Wang, N. Kandasamy, A. Guez, and M. Kam, "Adaptive Performance Control of Computing Systems via Distributed Cooperative Control: Application to Power Management in Computing Clusters," *ICAC '06. IEEE International Conference on*, june 2006, pp. 165 – 174.

[150] O. C. Wang N, Schmidt DC, "Overview of the CORBA Component Model," 2001, p. 557571.

[151] C. Wells, *The OceanStore Archive: Goals, Structures, and Self-Repair*, master's thesis, University of California, Berkeley, May 2001.

[152] J. Wildstrom, P. Stone, E. Witchel, R. J. Mooney, and M. Dahlin, "Towards Self-Configuring Hardware for Distributed Computer Systems," *The Second International Conference on Autonomic Computing*, June 2005, pp. 241–249.

[153] M. Woodside, T. Zheng, and M. Litoiu, "The Use of Optimal Filters to Track Parameters of Performance Models," *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, Torino, Italy, September 2005, p. 74.

[154] M. Woodside, T. Zheng, and M. Litoiu, "Service System Resource Management Based on a Tracked Layered Performance Model," *ICAC '06: Proceedings of the third International Conference on Autonomic Computing*. June 2006, pp. 175–184, IEEE Press.

[155] J. Xu, M. Zhao, and J. A. Fortes, "Cooperative Autonomic Management in Dynamic Distributed Systems," *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, Berlin, Heidelberg, 2009, SSS '09, pp. 756–770.

[156] J. Ye, J. Loyall, R. Shapiro, R. Schantz, S. Neema, S. Abdelwahed, N. Mahadevan, M. Koets, and D. Varner, "A model-based approach to designing QoS adaptive applications," *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, 2004, pp. 221–230.

[157] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang, "Modeling Hard-Disk Power Consumption," *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2003, pp. 217–230, USENIX Association.

[158] F. Zhang and S. T. Chanson, "Power-Aware Processor Scheduling under Average Delay Constraints," *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, Washington, DC, USA, 2005, pp. 202–212, IEEE Computer Society.

[159] J. Zhang, J. G. Dai, and B. Zwart, "Law of Large Number Limits of Limited Processor-Sharing Queues," *Math. Oper. Res.*, vol. 34, November 2009, pp. 937–970.

[160] J. Zhang and B. Zwart, "Steady state approximations of limited processor sharing queues in heavy traffic," *Queueing Systems*, vol. 60, 2008, pp. 227–246.

[161] T. Zheng, M. Woodside, and M. Litoiu, "Performance Model Estimation and Tracking Using Optimal Filters," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, May–June 2008, pp. 391–406.

[162] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai, "Tracking time-varying parameters in software systems with extended Kalman filters," *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*. October 2005, pp. 334–345, IBM Press.

[163] V. V. Zyuban and P. M. Kogge, "Inherently Lower-Power High-Performance Superscalar Architectures," *IEEE Trans. Comput.*, vol. 50, March 2001, pp. 268–285.

APPENDIX A

LIST OF PUBLICATIONS

A.1 Journal Publications

1. Mehrotra Rajat, and Abdelwahed Sherif, "Towards Autonomic Performance Management of Large Scale Data Centers using Interaction Balance Principles," *Journal of Cluster Computing, Springer Publishers* (accepted).

2. Dubey Abhishek, Mehrotra Rajat, Abdelwahed Sherif, and Tantawi Asser, "Performance Modeling of Distributed MultiTier Enterprise Systems," *ACM Performance Evaluation Review, vol. 37, no. 2, pp.9-11, September, 2009.*

A.2 Book Chapters

1. Mehrotra Rajat, Banicescu Ioana, Srivastava Srishti, and Abdelwahed Sherif, "A Power-Aware Autonomic Approach for Managing Scientific Applications in a Data Center Environment," in *Handbook on Data Centers. Springer Publishers* (submitted).

2. Mehrotra Rajat, Dubey Abhishek, Abdelwahed Sherif, and Tantawi Asser, "A Power-Aware Modeling and Autonomic Management Framework for Distributed Computing Systems," in *Handbook of Energy-Aware and Green Computing. Chapman and Hall/CRC Press Taylor and Francis Group LLC. Dec 2011.*

A.3 Conference Publications

1. Mehrotra Rajat, Abdelwahed Sherif, and Erradi Abdelkarim, "A Distributed Control Approach for Autonomic Performance Management in Cloud Computing Environment," *6th IEEE/ACM International Conference on Utility and Cloud Computing, December 9-12, 2013, Dresden, Germany* (accepted).

2. Mehrotra Rajat, Dubey Abhishek, Abdelwahed Sherif, and Rowland Krisa, "RFD-Mon: A Real-Time and Fault-Tolerant Distributed System Monitoring Approach," *The Eighth International Conference on Autonomic and Autonomous Systems (ICAS-2012).*

3. Chen Qian, Mehrotra Rajat, Dubey Abhishek, Abdelwahed Sherif, and Rowland Krisa, "On State of The Art in Virtual Machine Security." *IEEE South East Conference (SouthEastCon-2012).*

4. Mehrotra Rajat, Banicescu Ioana, and Srivastava Srishti, "A Utility Based Power-Aware Autonomic Approach for Running Scientific Applications." *13th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-12).*

5. Monceaux Weston, Deland E. Evans, Keith N. Rappold, Cary D. Butler, Abdelwahed Sherif, Mehrotra Rajat, and Dubey Abhishek, "Implementing Autonomic Computing Methods to Improve Attack Resilience in Web Services." *2011 DoD High Performance Computing Modernization Program Users Group Conference, 441445, 2011.*

6. Mehrotra Rajat, Dubey Abhishek, Abdelwahed Sherif, and Monceaux Weston, "Large Scale Monitoring and Online Analysis in a Distributed Virtualized Environment," *IEEE International Conference and Workshop on Engineering of Autonomic and Autonomous Systems, 0:19, 2011.*

7. Mehrotra Rajat, Dubey Abhishek, Abdelwahed Sherif, and Tantawi Asser, "Integrated Monitoring and Control for Performance Management of Distributed Enterprise Systems," *18th Annual IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, pp.424-426, 2010.*

A.4 Technical Reports

1. Mehrotra Rajat and Abdelwahed Sherif, "Application of Interaction Balance Principle for Optimal Control in the Distributed Web Service Deployment," *Technical Report, MSU-ECE-13-002, Mississippi State University, May 2013.*

2. Jia Rui, Mehrotra Rajat, Abdelwahed Sherif, and Erradi Abdelkarim, "An Implementation of a Real-Time Monitoring Framework to Support Fault Management in Distributed Systems," *Technical Report, MSU-ECE-12-002, Mississippi State University, May 2012.*

3. Mehrotra Rajat, Dubey Abhishek, and Abdelwahed Sherif, "RFDMon: A Real-Time and Fault-Tolerant Distributed System Monitoring Approach," *Technical report for Institute for Software Integrated Systems (ISIS), Vanderbilt University, 2011.*

4. Mehrotra Rajat, Dubey Abhishek., Abdelwahed Sherif, and Tantawi Asser, "Model Identification for Performance Management of Distributed Enterprise Systems," *Technical report for Institute for Software Integrated Systems (ISIS), Vanderbilt University, 2010.*

A.5 Posters

1. Srivastava Srishti, Mehrotra Rajat, Banicescu Ioana, and Abdelwahed Sherif, "A Model-based Framework for Autonomic Performance Management of Cloud Computing Systems," *15th SIAM Conference on Parallel Processing for Scientific Computing, Savannah, GA, Feb-2012.*

2. Mehrotra Rajat, Dubey Abhishek, Abdelwahed Sherif, and Tantawi Asser, "Performance Modelling of Distributed Multi-Tier Enterprise Systems," *NSF-CAC (Center for Autonomic Computing) Bi-Annual Meeting / Workshop @ Biosphere-2, Arizona, Fall-2011.*

3. Mehrotra Rajat, Qian Chen, Dubey Abhishek, Abdelwahed Sherif, and Rowland Krisa, "Automated Model-based Security Management of Web Services," *20th USENIX Security Symposium, San Francisco, CA, Aug-2011.*

4. Jia Rui, Mehrotra Rajat, Dubey Abhishek, and Abdelwahed Sherif, "Model-based Framework for Autonomic Resource Management of Web Services Platforms,"*NSF-CAC (Center for Autonomic Computing) Bi-Annual Meeting / Workshop @ Xerox, Dallas, Spring-2011.*

5. Mehrotra Rajat, Dubey Abhishek, and Abdelwahed Sherif,"Model-based Adaptive Performance Control of Cloud Computing Systems,"*NSF-CAC (Center for Autonomic Computing) Bi-Annual Meeting / Workshop @ Xerox, Dallas, Spring-2011.*

6. Mehrotra Rajat, Dubey Abhishek, Abdelwahed Sherif, and Tantawi Asser, "Integrated Monitoring and Control for Performance Management of Distributed Enterprise Systems," *18th Annual IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, pp.424-426, 2010.*

7. Mehrotra Rajat, and Abdelwahed Sherif, "Hierarchical Limited Look-ahead Controller for a Distributed Multi-Tier Enterprise Systems," *NSF-CAC (Center for Autonomic Computing) Bi-Annual Meeting / Workshop @ Mississippi State University, Fall-2009*