An Autonomic Framework for Integrating Security and Quality of Service Support in Databases

Alomari, Firas

http://hdl.handle.net/1920/8282
AN AUTONOMIC FRAMEWORK FOR INTEGRATING SECURITY
AND QUALITY OF SERVICE SUPPORT IN DATABASES

by

Firas Alomari
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

Dr. Daniel A. Menascé, Dissertation Director
Dr. Alexander Brodsky, Committee Member
Dr. Brian L. Mark, Committee Member
Dr. Duminda Wijesekera, Committee Member
Dr. Stephen Nash, Department Chair
Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: Spring Semester 2013
George Mason University
Fairfax, VA
An Autonomic Framework for Integrating Security and Quality of Service Support in Databases

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Firas Alomari
Master of Science
George Mason University, 2006
Bachelor of Science
Yarmouk University, 2001

Director: Dr. Daniel A. Menascé, Professor
Department of Computer Science

Spring Semester 2013
George Mason University
Fairfax, VA
Copyright © 2013 by Firas Alomari
All Rights Reserved
Dedication

To Allah for giving me the strength to carry this work, my family, friends and Professors for their unwavering support and encouragement.
Acknowledgments

Many thanks are due to my advisor, Dr. Daniel Menascé, for his guidance, encouragement, and enthusiasm for my work. His technical and editorial advice was essential for the completion of this dissertation and has taught me valuable lessons on the working of academic research. I would also like to thank my committee members Dr. Alexander Brodsky, Dr. Brian L. Mark, and Dr. Duminda Wijesekera for their advice and comments that help improve this dissertation. I would like to especially thank my wife Asmaa, my daughter and son, Talah and Fiasal, for their love, patience and understanding. Thanks also goes to my loving mother, sisters, and other family members for their prayer and support. Last, but not least, i would like to thank my friends and colleagues for their encouragement and support during the years of my study.
## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>Abstract</td>
<td>xi</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background and Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Research Problem</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Thesis</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Organization</td>
<td>6</td>
</tr>
<tr>
<td>2 Background and Related Work</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Background</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Related Work</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 Database Security</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2 Autonomic Computing</td>
<td>12</td>
</tr>
<tr>
<td>2.2.3 Performance Models</td>
<td>15</td>
</tr>
<tr>
<td>2.2.4 Queuing Networks</td>
<td>15</td>
</tr>
<tr>
<td>2.2.5 Fork and Join</td>
<td>17</td>
</tr>
<tr>
<td>2.2.6 Heuristic Search</td>
<td>22</td>
</tr>
<tr>
<td>3 The ISQDB Framework</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Proposed Framework</td>
<td>23</td>
</tr>
<tr>
<td>3.1.1 Environment</td>
<td>23</td>
</tr>
<tr>
<td>3.1.2 Notation and Controller Policy Definitions</td>
<td>27</td>
</tr>
<tr>
<td>3.1.3 Utility Function</td>
<td>28</td>
</tr>
<tr>
<td>3.2 Controller Architecture</td>
<td>36</td>
</tr>
<tr>
<td>3.3 The Queuing Network Model</td>
<td>41</td>
</tr>
<tr>
<td>3.3.1 Open Model</td>
<td>41</td>
</tr>
<tr>
<td>3.3.2 Closed Model</td>
<td>44</td>
</tr>
<tr>
<td>3.4 Practical Considerations</td>
<td>45</td>
</tr>
<tr>
<td>3.5 Concluding Remarks</td>
<td>47</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Related Fork and Join Approximations</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of a Configurable Multiclass Open FJ</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of a Dynamic Multiclass Open FJ</td>
<td>57</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of a Dynamic Single Class Closed FJ</td>
<td>61</td>
</tr>
<tr>
<td>4.4</td>
<td>Summary of Notations</td>
<td>64</td>
</tr>
<tr>
<td>4.5</td>
<td>Simulation Parameters</td>
<td>64</td>
</tr>
<tr>
<td>4.6</td>
<td>Multiclass Parameters</td>
<td>64</td>
</tr>
<tr>
<td>4.7</td>
<td>Approximation for $K = 10, \ldots, 100$. $R_a = $ Approximate Response Time, and $R_s = $ Simulated Response Time.</td>
<td>71</td>
</tr>
<tr>
<td>4.8</td>
<td>Comparison to Previous Approximations</td>
<td>75</td>
</tr>
<tr>
<td>5.1</td>
<td>Initialization Values</td>
<td>78</td>
</tr>
<tr>
<td>5.2</td>
<td>Model Validation Results</td>
<td>79</td>
</tr>
<tr>
<td>5.3</td>
<td>Initialization Values</td>
<td>92</td>
</tr>
<tr>
<td>5.4</td>
<td>Attacks Detected By Case</td>
<td>96</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Security, performance, and Manageability Trade-offs</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>A Fork and Join (FJ) Construct.</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Traditional Setup</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Envisioned Setup</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Envisioned Setup</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Framework Flowchart</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Union Probability</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>Exponential Averaging Approximation</td>
<td>31</td>
</tr>
<tr>
<td>3.7</td>
<td>Sigmoid Security Utility</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Utility Function for Response Time.</td>
<td>35</td>
</tr>
<tr>
<td>3.9</td>
<td>General Controller Architecture</td>
<td>37</td>
</tr>
<tr>
<td>3.10</td>
<td>Cross Layer ASM Architecture</td>
<td>39</td>
</tr>
<tr>
<td>3.11</td>
<td>Open QN Analytic Model for the Database System with Fork and Join</td>
<td>42</td>
</tr>
<tr>
<td>3.12</td>
<td>Closed QN Analytic Model for an Application with Fork and Join</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>A Fork and Join (FJ) Construct.</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>A Fork and Join Example</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>Average Response Time (in seconds) vs. Utilization for Open QNs</td>
<td>67</td>
</tr>
<tr>
<td>4.4</td>
<td>Average Response Time (in seconds) vs. Number of Jobs for Closed QNs</td>
<td>68</td>
</tr>
<tr>
<td>4.5</td>
<td>Throughput (Jobs per Second) vs. Number of Jobs for a Single Class Closed</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>Model for 4, 8, and 16 Queues</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Relative Error vs. Service Time Heterogeneity</td>
<td>70</td>
</tr>
<tr>
<td>4.7</td>
<td>Approximation vs simulation for homogeneous and heterogeneous case with</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>$\rho = 0.30$.</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>Percent Relative Error vs. Number of Jobs in a Single Class Closed QN for</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Full Fork.</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>Throughput vs Number of Jobs for 4, 8, and 16 Queues.</td>
<td>74</td>
</tr>
</tbody>
</table>
4.10 Percent Relative Error vs. Number of Jobs in a Single Class Closed QN for Full Fork ................................................. 75
5.1 Model Results for Different $\lambda$ ......................................................... 78
5.2 Controller Results .............................................................................. 81
5.3 Experiment Setup .................................................................................. 83
5.4 TPC-W Attacks Examples: SQL Injection (Top) and Cross-site Scripting (XSS) (Bottom) .................................................. 88
5.5 Performance Monitors: Windows Showing Total and Per core CPU Utilization and Maximum Frequency (left) and Windows Performance Toolkit (right) .................................................. 90
5.6 Model Maximum Percent Relative Errors vs. Number of Concurrent Users ................................................................. 90
5.7 Predicted vs. Measured $U_{total}^{T}(\bar{T})$ Per CI: Light Workload (top) and Heavy Workload (bottom) ........................................................................ 91
5.8 Experiment Results: (a) Workload, (b) AC Response Time for Home and Search Page Requests, (c) Controlled System Utility Values, (d) Difference in Global Utility $U_{g}$, (e) Uncontrolled vs. Controlled System Response Time, and (f) Policy Changes Over Time .................................................. 94
5.9 General System Response Time: (a) Evolution in Log Scale and (b) Average Response Time Per Case .................................................. 95
5.10 Requests response Time in Log Scale .................................................. 96
5.11 Experiment Results Under High Workloads: (a) Workload, (b) General Response Time .................................................. 97
5.12 Average Response Time in Log Scale .................................................. 97
5.13 Controlled System Response Time $U(T)$ Under High Loads .................. 98
5.14 Controlled System Utility Under High Loads ........................................ 99
Abstract

AN AUTONOMIC FRAMEWORK FOR INTEGRATING SECURITY AND QUALITY OF SERVICE SUPPORT IN DATABASES

Firas Alomari, PhD
George Mason University, 2013
Dissertation Director: Dr. Daniel A. Menascé

The back-end databases of multi-tiered applications are a major data security concern for enterprises. The abundance of these systems and the emergence of new and different threats require multiple and overlapping security mechanisms. Therefore, providing multiple and diverse database intrusion detection and prevention systems (IDPS) is a critical component of the defense-in-depth strategy for DB information systems. At the same time, an e-business application is expected to process requests with a certain service quality to maintain current customers and attract new ones. It would then be advantageous to use the combination of IDPSs that best meets the security and QoS concerns of the system stakeholders for each workload intensity level. Due to the dynamic variability of the workload intensity, it is not feasible for human beings to continuously reconfigure the system. It is therefore important that current systems be built with adaptive capabilities that can –at run time– dynamically respond to changes in it is surroundings.

This dissertation presents an autonomic computing approach for a self-protecting and self-optimizing database system environment that captures dynamic and fine-grained trade-offs between security and QoS. Specifically, the dissertation presents an Integrated Security and Quality of service support in DB (ISQDB) framework that uses a multi-objective utility
function. The utility functions considers the performance impact of IDPSs on the overall system under a certain workload, the detection and false detection rates of the IDPSs, and high level stakeholder preferences and constraints.

The dissertation starts by describing the general architecture of the controller used in the design of the ISQDB. Then, it describes how a security utility function based on detection rates and a performance utility function based on response time can be developed. The utility is then improved to include false detection rates. The dissertation then presents the design of an autonomic controller that uses combinatorial search techniques and analytic performance models to dynamically search the space of possible system configurations. The dissertation further shows how performance models based on queuing networks are used to estimate the performance of different configurations. Different approximations for fork and join queues were developed to address parallel and concurrent features of the queuing models that support the autonomic controller. The dissertation also shows the viability of the proposed approach in a simulated environment. Then, it describes the implementation and experimental results in an e-commerce system based on the TPC-W benchmark.
Chapter 1: Introduction

1.1 Background and Motivation

The increase in the number and diversity of computing devices in use today, as well as the complexity of enterprise computing environments, pose a maintenance and management challenge to a highly skilled workforce [1]. The high demand for skilled IT personnel is already outstripping supply and labor cost is exceeding equipment costs. As a result of the aforementioned factors, and motivated by the idea that a system should be able to manage itself, there is an overwhelming economic and practical need to automate, as much as possible, today’s system maintenance and administration.

The previous trends have also been a major concern for security professionals. The increasing number of devices in use today implies increase in the attack surface of the environment as well as more exposure to internal and external threats. Moreover, the increase in complexity hinders efforts to effectively manage and configure systems, which in turn has a direct impact on a system’s security posture. Therefore, securing the infrastructure must involve multiple overlapping controls that assist organizations in preventing, detecting, and responding to suspected intrusions. Consequently, this requires deploying comprehensive, redundant, and diverse security services in the organization. Using only one type of intrusion detection and prevention system (IDPS) restricts the range of detected attacks. Therefore, a reasonable approach is not to rely on any single IDPS, but on a combination of several diverse IDP systems that provides complementary features for the decision making process.

Equally, there is a great dependence on databases to provide data storage and retrieval mechanisms for the organization’s information systems. The information stored in databases ranges from personal and financial information, to commercial and government confidential
data. Therefore, the data stored in databases is the organization’s most valuable asset and protecting it from attacks is at the heart of the organization’s information security policy. Moreover, applications accessing these databases are becoming more accessible through the proliferation of on-line and cloud computing services that support an increasing number of business activities. Therefore, traditional intrusion detection and prevention systems (IDPSs) have been extended to databases to provide protection against new types of threats and complement other security mechanisms in protecting database systems.

Similar to traditional IDPSs, database IDPS (DBIDPS) can be broadly classified as misuse and anomaly based. The misuse based DBIDPSs trigger an alarm if there is any significant deviation from a normal profile. Anomaly based DBIDPSs base their decision on a set of rules or signatures that represent malicious activity. DBIDPSs can use the queries issued to the database (syntax-centric) [2, 3] or the actual results of the queries executed (data-centric) [4, 5].

Defensive measures in general and IDPSs in particular tend to impede the operation of IT infrastructure in terms of usability and performance. At one hand, IDPSs need to provide adequate protection against a variety of attacks. On the other hand, the IDPSs must avoid any performance degradation. In fact, recent reports [6–12] indicate that even if the tools that are needed to protect the organization infrastructure are available at their disposal they are not being effectively used, because security gets in the way of using the system for it is intended purposes efficiently. For example, most IDPS products turn on all signatures on the common ports. That makes configuration easy, but it guarantees that the IDPS is going to perform in a sub-optimal way. System throughput and latency will be affected by having extra signatures looking at traffic. This in turn will results in operators turning off security services or using them in an insecure manner. Even if a system is properly configured, the nature of today’s computing environments, dynamic workloads, different user preferences and constraints require dynamic and differentiated level of users support.

Furthermore, recent research has placed more emphasis on network security and network
intrusion detection. However, an effective defense-in-depth requires security and protection services at the application level as well as the network and operating system level. This becomes more critical when considering overlapping (e.g., multiple intrusion detection techniques, or encryption) and competing (e.g., security and performance) system objectives, where conflicts can occur, ultimately reducing user productivity and increasing costs due to security incidents and performance overhead. Therefore, there is an increasing need for systems that can sense their environment and adapt to changes in a way that meets the high level organizational policies.

Autonomic systems [13,14] are being increasingly implemented in various domains as a promising new instrument for designing various degrees of adaptation in computer systems while hiding intrinsic complexities from operators and users. Autonomic systems are ideally capable of balancing performance objectives with security objectives, by dynamically reconfiguring managed devices to relax security requirements in order to meet performance requirements, or vice versa. Generally, an autonomic system mediates between competing objectives of maximized performance and minimized risk. This requires high-level and detailed knowledge of the entire system as well as embedded models of performance and risk analysis that allow an autonomic controller to dynamically analyze the current environment, reconfigure and redeploy system services accordingly.

This work is an attempt to address these issues. Specifically, this dissertation shows how to 1) use available security metrics to define security services offered by intrusion detection systems in a way that would be easily tractable and measurable; 2) quantitatively identify a standard security measure that expresses the utility of using combined IDPSs in terms of their aggregate detection rates; 3) design a system that can dynamically adapt to changes in the workload intensity; and 4) take advantage of heuristic search and queuing models to guide such adaptation.
1.2 Research Problem

Changes in security and performance objectives are no longer occasional occurrences, but expected events, and need to be dealt with dynamically at run time. Therefore, the research problem addressed in this work is that of designing and evaluating a robust and general approach to model-based controllers for self-managing computer systems, that can integrate QoS and security objectives, and demonstrate the effectiveness of these controllers. The controllers should be able to dynamically and automatically determine the values of configuration parameters that optimize a pre-defined QoS function or a utility function that depends on the performance and the security attributes of the autonomic system. Analytic performance models need to be used to guide heuristic search techniques in their exploration of the space of possible configuration values. The next section discusses the contributions of this dissertation.

1.3 Contributions

This research focuses on some essential aspects of autonomic computer systems, namely, self-management and self-protection. More precisely, it demonstrates how analytic performance models can be leveraged for integrating QoS and security in implementing a self-managing system. This is achieved through 1) developing performance models that capture the impact of using different security services on the overall performance, and 2) techniques that select configurations that meet both QoS and security objectives. Therefore, the goal of the work presented in this dissertation is focused on proposing techniques that can be integrated into systems to allow for their dynamic reconfiguration based on 1) variations of workload type and intensity and 2) security and performance objectives set by the system stakeholders. To achieve this goal, a system must be able to 1) sense changes in its environment to decide if reconfiguration is needed and 2) and quantitatively assess different available configurations. This dissertation shows that this goal can be accomplished using a combination of heuristic search algorithms and model driven evaluation of the quality of different configurations.
available to the system. Performance models are used to enable the system to predict the performance of selected configurations. Different security measures allow the system to guide a heuristic search in finding a configuration that maximize both goals based on high level objectives. Therefore, we start by demonstrating how performance models are built and used to predict systems performance. Moreover, to address non-product form features, such as fork-and-join, in our performance models, we propose techniques and algorithms to approximate the performance values of these models. These approximations are then extended to deal with the more general cases of multiple classes of customer requests and different QN types. In the context of performance models that include non-product form features, an assessment of the performance models is conducted as well. Then, we show how these performance models can be used to build a controller that runs regularly to compute performance values for the system and issues reconfiguration commands dynamically. Later, we show how the combinatorial space of different system configurations is searched using heuristic techniques to find solutions that improve and maintain the overall system performance and security.

Another contribution is relevant to quantifying the security services provided by the controller. For this case, a utility function that incorporates the effectiveness of security services and their combinations is developed and used by the controller to compute the security value for each configuration that is used in combination of the performance models to guide the system configurations.

1.4 Thesis

It is possible to use autonomic computing techniques to dynamically determine and use a varying combination of syntax-driven and data-driven intrusion detection systems in databases in order to optimize a desired balance between QoS and security level.
1.5 Organization

This dissertation is organized as follows. Chapter two provides some background information and reviews basic concepts relative to performance models, autonomic systems, and security. It also discusses some of the representative related work found in the literature. Chapter three presents the ISQDB framework and detailed description of suggested mechanisms that should be built into systems to enable self-management. In chapter four, we present our approximations for the fork and join constructs in queuing networks models. This chapter also shows a comprehensive evaluation of the accuracy of the approximation. This is accomplished in the context of simulation environment experiments. Chapter five is concerned with illustrating the effectiveness of the ISQDB framework. It deals with both a simulated environment and a real experimental environment. Chapter six provides concluding remarks and directions for future research.
Chapter 2: Background and Related Work

This chapter provides background and definitions regarding database security, autonomic computing, performance models, and how they fit together in this work. More specifically, concepts related to this work are defined and explained. Then, we discuss relevant related work to each concept.

2.1 Background

Databases are a fundamental component in today’s computer systems. They are the system means for storing, organizing, and accessing data. Most internal business services and e-commerce sites have a database system at their back-end. Applications accessing these databases are becoming more accessible through the proliferation of online services and cloud computing to support an increasing number of business activities. Data found in these databases range from personal information and banking transactions to medical records and commercial secrets. Due to the increasing impact of database applications on organization continuity, there is a growing concern about security and quality of service (QoS). For example, due to the increasing use of online services that are sensitive to QoS requirements and often contain sensitive information, it is very important to provide adequate security and protection services while maintaining the QoS requirements. Furthermore, depending on the nature of the applications, security requirements may be perceived differently by the system owners and/or the end-users.

Security and QoS pose conflicting requirements because security mechanisms require extra resources that inversely affect the QoS [15,16]. A completely secure database application may not be useful if it significantly violates the required QoS goals. On the other hand, a well developed database application that meets QoS requirements without proper
security support can be compromised causing substantial damage to profit and reputation. Even with systems that support both QoS and security, a system administrator should specify and monitor the required QoS and security parameters separately. This in turn puts tremendous pressure on system administrators who, in many cases, are already overwhelmed by the tasks of installing, properly maintaining, and configuring their systems in a way that provides optimal performance. For the most part, security and QoS have been mostly addressed separately. However, an information system must address both challenges effectively. Therefore, there is a need to design and implement autonomic system capabilities that can integrate both security and QoS requirements in database applications.

The security, performance, and manageability triad shown in Fig. 2.1 represents the complexities in balancing between the three different objectives.

![Figure 2.1: Security, performance, and Manageability Trade-offs](image)

Traditionally, database security has been addressed by access control mechanisms [17] and policy enforcement mechanisms [18]. However, these mechanisms are not sufficient and need to be complemented by anomaly detection mechanisms to protect against emerging
threats like impersonation attacks [19], SQL injection [20], and insider threats [18, 21]. Therefore, Intrusion Detection and Prevention Systems (IDPSs) have been proposed to complement traditional security models in databases. Usually, IDPSs are used to detect attacks on networks or operating systems by monitoring system activities and responding to any abnormal activity. Researchers have recently extended classical IDPSs to protect databases from malicious transactions [22].

IDPSs determine the normal behavior of users accessing the database. Any deviation of such behavior is treated as an intrusion. There are two main models of IDPSs: anomaly detection and misuse detection. The former model bases its decisions on the profile of a user’s normal behavior. It analyzes a user’s current session and compares it with the profile representing the user’s normal behavior. An alarm is raised if significant deviation is found during the comparison of the user’s session data and user’s profile. This type of system is well-suited for the detection of previously unknown attacks. The normal profile can be built using the queries issued to the database (syntax-centric) [2, 3] or the actual results of the queries executed (data-centric) [4, 5]. In contrast, a misuse detection model makes decisions based on the comparison of user’s session or commands, with the rules or signatures of known attacks that have previously been used by attackers. It is well-accepted that the main disadvantage of the former approach is false positives, while the latter approach must deal with false negatives. False positives and false negatives are usually referred to as system accuracy and represent one of the challenging factors that have received most attention when deploying IDPSs [23]. However, little research exists on how to combine different and complementary techniques to increase the overall security accuracy. Security solutions also often target specific attack types or threat scenarios. However, databases are subject to a variety of attack types and threat scenarios. Finally, most solutions treat all users uniformly assuming they introduce the same threats to the system, while in reality different user groups pose different threats to the system.

The other challenge that has received relatively less attention is the impact of IDPS mechanisms on the overall system and its QoS requirements. Performance is a major barrier
to the adoption of security mechanisms in computer systems. Specifically, the impact of
the overhead associated with data monitoring (e.g., data collection), analysis, and response
actions for security protection is not comprehensively treated when evaluating a database’s
overall performance. Moreover, it is desirable to use a combination of IDPSs to increase the
effectiveness of the system security capabilities [23, 24]. However, this can have a negative
impact on a system’s performance when the system applies the security techniques on
all activities indiscriminately and with no consideration for the system QoS objectives.
Additionally, for system administrators, configuration and maintenance of QoS and security
becomes very cumbersome and error prone. Consequently, there is a need to design systems
that are able to monitor and manage activities and apply security defense mechanisms
based on 1) the source of activities, 2) the likelihood of activities being malicious, and 3)
system QoS and security requirements. Such systems should be able to manage the security
mechanisms and their deployment automatically without human intervention.

2.2 Related Work

2.2.1 Database Security

Standard database security controls such as access control mechanisms, authentication, en-
cryption technologies and so forth are not of much help when it comes to preventing emerg-
ing attacks such as SQL injection, privilege escalation, and data theft from insiders. The
above mentioned threat scenarios have forced organizations to re-evaluate security strate-
gies for their DBMSs [20], consequently proposing the use of database intrusion detection
systems and database activity monitoring tools to address these emerging threats.

Database intrusion detection can be broadly categorized as syntax-centric, and data-
centric. In syntax-centric methods the database syntax is used to build the normal profile
for users while in data-centric methods the profile is built using the result sets returned
by the query. An example of syntax-centric can be found in [2] that describe the database
IDPS DEMIDS that uses frequent sets of attributes accessed by the user to build a normal
user access profile. Their method can be used for security re-engineering in the organization to define roles and their expected profile. Bertino et al [3] proposed an intrusion detection system for database systems, which is conceptually very similar to DEMIDS. One important fact that they considered is that databases typically have a very large number of users and, therefore, keeping a profile for each single user is not practical. Hence their approach is based on the well known role-based access control (RBAC) model. It builds a profile for each role and checks the behavior of each role with respect to that profile. With roles, the number of profiles to build and maintain is much smaller. Therefore, their approach is usable even for databases with large user population. Others [25–27] have proposed methods leveraging the data dependencies among data items, which are in the form of classification rules, i.e., what data items are most likely to be updated after one data item is updated and what other data items probably need to be read before this data item is updated in the database by the same transaction. A data-centric intrusion detection approach presented in [5] uses the result sets returned by the queries to build a statistical profile of what a user is trying to access rather than how it is expressed. In [4], the retrieved data is analyzed based on a misuse-ability weight. The weight represents the sensitivity level of the data exposed to the user. This measure predicts the ability of a user to exploit the exposed data in a malicious way. In [28], syntax-centric and data-centric methods are used to detect data leakage attacks on the database. The syntax of SQL requests is used to evaluate the diversity or correlation among queries in the session and the results are used to evaluate the data coverage or broadness in the same session. Other techniques that attempts to protect databases are also presented in [29–31].

There have also been many commercial implementations from a variety of companies that deal with database security. For example, database activity monitoring (DAM) is a database security technology for monitoring and analyzing—continuously and in real time—database activity that operates independently of the database management system (DBMS) [20,32]. Database activity monitoring and prevention (DAMP) is an extension to DAM that goes beyond monitoring and alerting to also block unauthorized activities.
One limitation to DAM tools is their inability to monitor privileged users who can log into the database server directly. Moreover, DAM and DAMP usually suffer from their inability to detect well crafted intrusions to public and web databases. There are other implementations from a variety of companies that deal with database intrusion detection, application firewalls, and database activity mentoring [33–38]. We refer the reader to [19, 20, 22] for further details on DAM and database IDPSs.

2.2.2 Autonomic Computing

Autonomic systems (a.k.a self-* systems) are capable of self management by self-configuring, self-optimizing, self-healing, and self-protecting by utilizing MAPE-K or feedback loops [13, 39, 40]. Specifically, autonomic computing calls for embedding complex computational intelligence into the system itself to make its operations autonomous by having the following features:

1. Self-awareness: Enables the system to know about its components and aware of its overall status to provide input for different capabilities to respond to environment changes accordingly. This is usually represented by feedback or controller loops.

2. Self-configuration: Enables the system to respond to environment changes configuring and reconfiguring itself automatically.

3. Self-optimization: Enables the system to pick the configuration that optimizes quality of services metrics.

4. Self-healing: Enables the system to recover from a disastrous or malfunctioning state to normal and safe state automatically.

5. Self-protection: Enables the system to defend itself against malicious activities. This includes deploying and configuring security services automatically without human intervention. Furthermore, the system should be able to assess the impact of these security services on the overall system and be able to balance between different security
and performance requirements. In this work, we will address a particular case, i.e., intrusion detection, to enable the system to provide security and protection services automatically based on requirements and system performance.

There are other desirable capabilities in autonomic systems, such as context-awareness that an autonomic system should be equipped with. For a more detailed review of autonomic systems and communications, please refer to [40–42].

Much research has already been carried out on autonomic systems allowing adaptation of different QoS metrics [43–45]. In databases, autonomic database management systems (ADBMS) have focused on autonomic management of resources for achieving optimal performance [46], autonomic provisioning of databases [47], resource selection for database optimization and tuning [48], policy-based decisions and management [49], isolation of attacks on data and recovery from malicious transactions [50], and role-based access control [51].

Self-protection mechanisms are presented for web applications in [52], grid environment [53], host operating systems [54], and at the network level [55, 56]. Other implementations include IDPSs that adapt to changes in the environment. Most notable is [57] where methods are introduced to enable the IDPS to adapt dynamically to changes in workload intensity. However, their work focuses on a single signature-based network IDPS. Self-cleansing [58] and self-healing [59] systems have been proposed as a way for autonomic systems to provide a degree of self-protection. A self-protecting database system that addresses the insider threat is proposed in [18, 60], where the autonomic policies are directly integrated into the DB application.

The need for solutions that protect sensitive information while maintaining quality of service have been recognized by several research works in different domains [16]. For example, the term Quality of Protection (QoP), has been suggested to extend QoS to effectively accommodate multi-level security. The work in [61] provides different QoP by assigning different priorities to traffic based on traffic abnormality. Adaptation in IDPSs is proposed in [57] to enable an IDPS to adapt dynamically to changes in workload intensity. However,
their work focuses on a single signature-based network IDPS. Other attempts to integrating QoS and security have been proposed. However, most focus on wireless communications [62] and online streaming [63]. In [64], the users of wireless ad-hoc networks are presented with a set of security requirements and end-to-end QoS delay requirements. Depending on a user’s chosen level of security and delay requirements, the middleware adaptor attempts to attain the minimum end-to-end delay while offering the user the highest possible security level, which is proportional to the encryption key-length. The work in [65] introduced a QoP model that takes into consideration authentication times and cryptographic overheads and throughputs. In [66], a security advisory system uses a predetermined attribute decision model to adjust the security level dynamically based on feedback from monitoring components and recommends a security policy that meets the QoS requirements. To enable service providers to advertise Security of Service to their clients, researchers investigated the incorporation of security parameters into the Service Level Specifications (SLS) in [67]. The selected security parameters are integrated to enhance SLA based management of QoS with the generation of network policies that guarantee the reservation of adequate resources for meeting both security and QoS needs.

As indicated above, most prior work addresses security and QoS independently. The trade off between the two are not comprehensively dealt with. In the few cases where security is viewed as a QoS metric, the proposed solutions lack generality and focus on very specific aspects of the trade-off, such as encryption key length versus end to end delay. The approach presented in this research concentrates on integrating IDPS with QoS, and more specifically on using application level IDPSs in a way that guarantees QoS and security requirements are balanced. The previous solutions integrate QoS and security requirements at the network level and use a predetermined security and performance values. Our approach uses dynamic models to assess the performance of different security configuration and chooses one that provides the highest security while meeting the QoS requirement of the system.
2.2.3 Performance Models

As modern systems become more complex, more integrated, and more essential to the conduct of business than ever before, there is an increasing need for performance models that allow practitioners and designers to study and evaluate the performance of systems to answer questions of cost and performance that arise throughout the life cycle of such systems.

There are two main approaches for performance modeling: analytic modeling based on system queuing models and statistical modeling based on learning patterns from previous system performance data [68]. Both approaches have been applied successfully in autonomic computing scenarios. Analytic performance models [43] are used here because of their effectiveness and the availability of efficient algorithms to solve them. In our case, performance models are used to evaluate a very large number of possible performance models dynamically to find an optimal or near-optimal configuration [43]. This involves analyzing a large number of complex configurations before an adequate solution is found.

2.2.4 Queuing Networks

Queuing networks (QN) are defined as a collection of interconnected queues (resource plus waiting line) that provide service to customers (requests, jobs, etc.). Customers move from one queue to another until they complete their execution. Customers with similar characteristics (service times and routing probabilities within the QN) can be grouped into one (i.e., single class) or more (i.e., multiclass) classes. A QN can be open, closed, or mixed. In an open QN, customers enter the network at a certain arrival rate, receive service, and leave the network. In a closed QN, the number of customers in the QN is fixed. In mixed networks, some customer classes are open and the others are closed. The inputs parameters of a QN model consist of workload intensity parameters (e.g., arrival rate, number of customers) and service demand parameters at each resource (i.e., total service time of a request at a resource during all visits to the resource). The residence time of a customer at a resource in a QN is the total time spent by that customer at the resource,
either receiving service or waiting to receive service. A number of techniques are available to solve QNs to obtain performance measures. However, there are some conditions that limit the use of these techniques to model parallel systems and applications.

Queuing Networks (QNs) provide a formal way to study systems in which contention for resources and waiting are a key feature. QNs provide a way to represent systems as a network of \( K \) queues, which can be evaluated analytically, and are used extensively in performance and capacity planning studies due to their simplicity, relative accuracy, and low cost compared to other performance design techniques (e.g., simulation). QNs can be single class or multiclass. A system is modeled as a single class QN if all customers have a similar behavior in the way they consume the various systems resources and multiclass otherwise. We refer the reader to [69, 70] for more details on QNs and their applications.

The BCMP theorem [71] defines a special class of open, closed, and mixed QNs that are amenable to efficient solutions. These QNs are called product-form QNs. The computational complexity for solving multiclass open QNs is \( O(K \times R) \) where \( R \) is the number of classes in the QN (see [69] for example). A number of efficient computational algorithms for solving product-form closed QNs have been developed. The most important ones are the Convolution Algorithm [72] and Mean Value Analysis (MVA) [73].

Single-class MVA is a simple and efficient recursion with computational complexity equal to \( O(N \times K) \) (see [69] for example). However, for multiclass MVA, the complexity of an exact solution is much higher and is equal to \( O(K \times R \prod_{r=1}^{R} (1 + N_r)) \) [70] where \( N_r \) is the number of class \( r \) customers in the QN. As a result of that, approximations for multiclass MVA have been developed and are called Approximate Mean Value Analysis (AMVA) [74].

The MVA approximation is motivated by the observation that the number of class \( r \) customers at each queue increases proportionally to the number of class \( r \) customers in the QN [75]. AMVA starts by assuming that the number of customers is equally distributed among the QN devices. In each iteration, the QN is solved for the full customer population, rather than building from an empty QN. The iteration stops when the maximum relative
difference between queue length values in successive iterations is less than a given tolerance. Unlike exact MVA, the complexity of solving AMVA is $O(K \times R^2)$ for each iteration. It has been shown that few iterations are required to converge to an acceptable tolerance [69]. Therefore, AMVA is the method of choice for solving multiclass closed QNs. The algorithms for solving open and closed, single and multiclass, QNs are well-known in the literature (see [69, 70] for example). This dissertation has extended these algorithms for the case of FJ queues.

2.2.5 Fork and Join

More recently, an increased emphasis has been placed on efficient analytic models and approximations to be used in autonomic computing and self-managing systems [39, 40]. These systems dynamically adjust their configuration to meet system performance objectives at run time. Analytic models are used in such systems to evaluate a very large number of possible performance models dynamically to find an optimal or near-optimal configuration [43]. Therefore, there is an increasing need to develop techniques to efficiently predict system performance dynamically or at design stages. A structure of increasing importance is parallelism, where jobs are replicated and serviced in parallel by different processing units with possibly different processing times. Some examples of what is called *fork and join* (FJ) include:

- Web service composition applications [76] invoke several simultaneous parallel services and only proceed when all parallel requests have completed.

- A combination of Intrusion Detection Systems that work in tandem to protect a computer system; a request is only allowed to proceed after all types of attacks have been inspected for [77].

- A computer system with multiple redundant disk arrays (RAID) [78] where the system queries different disks in parallel and integrate their response into a single result.
• A MapReduce framework [79] where computations that use large datasets are divided into sub problems and the results are aggregated to provide the answer to the original problem.

• A multi-core system with a program that runs two or more routines in parallel; all should be completed in order to start the next routine [80].

• A health care provider requests several tests simultaneously by several labs; the patient’s diagnosis is finalized when all test results are received [81].

There is no exact analytic solution to FJ modeling except for the case of two homogeneous queues and using simulation is computationally expensive. For example, a simulation of 100 queues on a Intel Core i5 machine takes an average of 3 hours. Therefore, there is a need for an analytic approximation to FJ constructs that allows modelers to deal with this type of situations. The approximation must be simple and efficient enough to be used by practitioners and general enough to be used to evaluate several candidate scenarios in different domains with acceptable accuracy.

There have been enhancements in several ways to extend the applicability of analytic models to parallel systems; various improvements have been introduced to model features such as simultaneous resource possession, fork-and-join, and blocking [82]. Furthermore, in cases where no product form solutions exist, hybrid techniques are used by combining approximate solutions and analytic results [73]. However, the complexity of parallel systems, restrictive assumptions on the models, and computation complexity limit the applicability of these techniques.

In the FJ construct shown in Fig. 2.2, there are $K$ branches and each branch has a single queue. Jobs are split on arrival into $J$ ($J \leq K$) tasks to be serviced in parallel by $J$ queues. Only when all $J$ parallel tasks of a job complete, the job can rejoin (synchronize) and leave the system. A job can be split to every queue in the FJ ($J = K$) (i.e., full fork) or to a certain number of queues in a probabilistic way (i.e., dynamic fork). In the dynamic fork case, jobs are first split into a random number of $J$ ($1 \leq J \leq K$) tasks.
Then, $J$ specific queues are selected in a probabilistic way. In multiclass models, each job class can be routed differently in the FJ (i.e., configurable fork). A single queue in the FJ operates like a standard queuing system. When the service times distributions and their moments are the same at all queues visited by the tasks of a job in a FJ construct, we have a homogeneous FJ. Otherwise, we have a heterogeneous one.

The synchronization requirements of a FJ add significant modeling complexity. The only exact method available to solve general networks with non product-form characteristics is to determine and solve the Markov process representation of the network. Generalized Stochastic Petri Nets (GSPN), can be used to specify a system and facilitate the automatic generation of an underlying Markov process, whose solution can be used to compute the performance metrics of interest [83]. Even if the Markov process representation of a GSPN can be found, the solution is usually too costly unless the network has a small state space. However, this is not the case for many practical problems. Therefore, approximation techniques have been proposed to estimate performance measures for FJ constructs in QNs. However, often these approximations are designed with a set of restrictive assumptions that limit their applicability for more general performance planning and design scenarios.
Table 2.1: Related Fork and Join Approximations

<table>
<thead>
<tr>
<th>Service time</th>
<th>This Dissertation</th>
<th>[84–86]</th>
<th>[87, 88]</th>
<th>[89]</th>
<th>[90, 91]</th>
<th>[92]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heterogeneous</td>
<td>Homogeneous</td>
<td>Homogeneous</td>
<td>Homogeneous</td>
<td>Homogeneous</td>
<td>Heterogeneous</td>
<td>Heterogeneous</td>
</tr>
<tr>
<td>Customers</td>
<td>Single and Multiclass</td>
<td>Single Class</td>
<td>Single Class</td>
<td>Single Class</td>
<td>Single Class</td>
<td>Single Class</td>
</tr>
<tr>
<td>QN type</td>
<td>Open and Closed</td>
<td>Open and closed</td>
<td>Open</td>
<td>Open</td>
<td>Open</td>
<td>Open</td>
</tr>
<tr>
<td>Complexity</td>
<td>Open: (O(K)), Closed: (O(NK))</td>
<td>Open: (O(K))</td>
<td>(O(K))</td>
<td>Algorithmic (O(K^2))</td>
<td>Requires solving complex integrals</td>
<td>Algorithmic (O(K^3))</td>
</tr>
<tr>
<td>Fork Policy</td>
<td>Full, dynamic, and configurable fork</td>
<td>Dynamic fork</td>
<td>Full fork</td>
<td>Full fork</td>
<td>Full fork</td>
<td>Full fork</td>
</tr>
</tbody>
</table>

Related work

Exact and efficient solutions for FJ queues are only known for two servers (see [93, 94] for open networks and [86, 88] for closed and open networks). Approximation techniques have been used to compute the average response time for FJ networks for the case of more than two servers. Table 2.1 compares approximations in previous work with the ones presented in this dissertation. Initially, [95] defined bounds for the mean response time for FJ networks, [88] derived an approximation based on the observation that both lower and upper bounds grow at the same rate as the number of servers. The work in [87] extends this by interpolating between light and heavy traffic to approximate homogeneous FJ queues. However, the work in [87, 88] assumes homogeneous queues where customers are grouped into a single class. Furthermore, that work requires obtaining a scaling factor estimated through simulation. The approximation presented in [86] considers a dynamic fork model. However, it only considers single class homogeneous FJ constructs. The approximation presented here considers heterogeneous service demands and multiple classes of jobs. In [90, 91], maximum order statistics is used to approximate the synchronization time in FJ with heterogeneous service demands. However, their approach requires solving integrals for different number of
servers and does not consider multiclass and closed networks or dynamic and configurable forking. In [86], an approximation is presented for open homogeneous FJ with dynamic fork. However, it only considers homogeneous queues and single class models. Ray [89] uses a bubble sort algorithmic approach to find bounds for homogeneous exponential service time FJ networks. Another algorithmic approximation for open FJ networks is given in [92] using a matrix geometric approach. However, the algorithmic solutions presented in [89] and [92] are iterative and their complexity increases with the number of servers in the FJ model. Furthermore, they do not consider multiclass or closed FJ networks. Varki [84,85] presents response time bounds for closed FJ constructs and extends Mean Value Analysis (MVA) [96] to obtain an approximate solution for QNs with single-class closed FJ sub-networks. Furthermore, the work in [84,85] considers the more general case of partial branching of jobs in the FJ. The work in [85], [97] presents non-iterative geometric response time bounds for closed networks with FJ constructs. However, all work in closed networks considers a single class closed QN with homogeneous service times. The work in this dissertation considers single and multiclass heterogeneous closed networks with dynamic branching (i.e., dynamic fork) and presents methods to solve them. In [98], a closed model is analyzed as described by the BCMP theorem [71] but the join point is not modeled. In [84,99], a FJ is considered where the number of jobs at each fork is random. However, their analysis assumes single class homogeneous servers. The approximation approach presented here considers a FJ with heterogeneous servers where different classes may execute jobs on different servers in the FJ.

The comparison discussed in Table 2.1 shows that: 1) The techniques for heterogeneous FJ suffer from high computational cost or the difficulty in obtaining a closed form expression. 2) None of the heterogeneous techniques provide a solution for closed or open multiclass networks. 3) The approximation presented in this dissertation provides a more general solution by considering open or closed, single or multiclass heterogeneous FJ networks with full, dynamic and configurable routing in the FJ constructs.
2.2.6 Heuristic Search

Fundamentally, parts of the problem that arises in this dissertation and in many resource allocation and scheduling problems is a combinatorial search problem that consists of finding those combinations of a discrete set of items that satisfy specified requirements. The number of possible combinations to consider grows very rapidly (e.g., exponentially) with the number of configurations, leading to potentially lengthy solution times and severely limiting the feasible size of such problems [100].

Heuristic search algorithms can be used to search combinatorial search spaces. They provide an approximate solution when finding the exact solution is too slow or too expensive [101]. The most famous ones are Hill-Climbing, Beam Search, and Best-First Search. In this dissertation, we will be using hill-climbing and random restart hill climbing as our heuristics of choice [102]. Hill climbing starts with an arbitrary solution to the problem and then examine solutions incrementally until some improvements are found. In some variants of hill climbing the algorithm defines a set of neighbors and the first neighbor that offers improvements is followed (i.e., greedy hill climbing). In other variants all neighbors are examined and then the ones with best improvements are followed. One limitation in hill climbing is that it might get stuck in a local optimum and never find the optimal solution. The random-restart hill climbing is offered as a way to mitigate this limitation. In this variant, the algorithm iteratively use hill climbing, however after each set of neighbors, the algorithm restarts with a random solution and examines a new set of neighbors.
Chapter 3: The ISQDB Framework

This chapter we presents the general approach of the integrated Security and Quality of Service support for Databases (ISQDB) framework. Particularly, we describe the proposed ISQDB framework environment. The autonomic controller approach used in the framework is described. Then the notations and the controller policy definitions are discussed. Utility functions for security and performance and their interpretations are presented as well. The second section of this chapter presents the controller architecture and describes how heuristic search is performed to find improved configurations. The interactions between different components to find and dynamically reconfigure the system are discussed as well. The last section describes the performance model used in the framework and how to compute performance measures using the model. Some of the work presented in this chapter appears in [103], [104], and in a paper submitted for publication [105].

3.1 Proposed Framework

Next we discus the environment, controller approach and the utility function used in the ISQDB framework.

3.1.1 Environment

We consider an environment where a mix of IDPSs are combined to achieve the highest security level possible (see Fig. 3.1). The IDPSs can use syntax-centric or data-centric techniques. Syntax-centric techniques use the query to evaluate the transactions submitted to the database. Data-centric techniques use the results generated by the query to evaluate the transaction legitimacy. It is desirable to use a combination of different and diverse techniques to provide the best results in terms of detection accuracy [23]. In a traditional
A syntax-centric IDPS sits between the users and the database. A syntax-centric IDPS inspects and evaluates requests before they are submitted to the database. A data-centric IDPS evaluates the result sets returned by transactions before they are passed back to the users. The time required for the evaluation depends on the number and type of techniques used and the data returned by the query. Therefore, the overall response time of the system depends on the workload intensity, the number and type of security mechanisms used, and the overhead associated with each mechanism.

A security policy is a combination of security configurations ranging from applying all security mechanisms for all users to not applying any security mechanism. Based on the threat catalog mentioned earlier, the goal is to find the highest security level that should be selected in a way that the QoS requirements are not compromised. The security level of a system in any environment is usually defined in terms of how effective the security mechanisms are in protecting the system. For example, in secure communication systems, encryption key length can be used to represent the security level. In other systems, the
security level is defined as a discrete value of high, medium, or low, where each value represents a certain assurance level for the system. In the environment of Fig. 3.1, it is desirable to have a security level that correlates to the number of IDPSs used and their detection rate (i.e., true positives).

Security policies are configured manually in the system of Fig. 3.2. Furthermore, in most cases, the configurations available are not granular enough to be effectively used. For example, an administrator can only enable or disable a certain IDPS for all requests. Even in the cases where there is greater flexibility in the configurations, it is then the administrator’s responsibility to know and decide which configuration is appropriate for a specific environment at a certain time. The problem becomes more evident when considering the QoS requirements along with the security requirements. For example, when the system is under a high workload, it might be acceptable to relax the security requirements temporarily to meet increasing demands. Additionally, since in most situations, different system stakeholders view priorities differently, the relaxation in security requirements should ideally be based on predefined stakeholder preferences and risks.

In Fig. 3.3, incoming requests are labeled according to the user’s role. The autonomic
controller sits between the database and the application server to evaluate requests and send them to one or more of IDPSs. The controller is executed at certain intervals to (1) obtain performance values (e.g., arrival rate) from the monitored system; (2) identify possible configurations (i.e., combination of IDPSs) that can be applied to meet the system security objectives; (3) compute performance and security metrics (values) for the identified configurations using functions described in the next sections; (4) and recommend the combination that represents the highest security level that can be selected in a way that meets the QoS requirements of the system for a given workload. As a result of the controller execution, the system is reconfigured to apply the new security policy to incoming requests. Figure 3.4 depicts a flowchart that shows the steps the controller takes to process database requests and enforce the specific role security policies.
3.1.2 Notation and Controller Policy Definitions

In the remainder of this chapter we refer to IDPSs as security mechanisms. The number of syntax-centric and data-centric security mechanisms are denoted by $K$ and $M$, respectively.
Let, \( N = K + M \) be the total number of security mechanisms. We assume that security mechanisms from 1 to \( K \) are syntax-centric and security mechanisms from \( K + 1 \) to \( N \) are data-centric. Let \( R \) be the number of roles in the system and \( A \) be the number of attack categories identified in the environment. Let \( a_{r,j} \) be the likelihood of observing attack \( j \) by role \( r \), \( d_{i,j} \) be the detection rate of attack \( j \) by security mechanism \( i \), and \( o_i \) be the overhead (i.e., service demand) for mechanism \( i \).

The controller policies are an assignment of security mechanisms that are applied to requests originating from a certain role \( r \). The policy \( \vec{\rho}_r \) for role \( r \) is represented by \( \vec{\rho}_r = (\varepsilon_{r,1}, \ldots, \varepsilon_{r,i}, \ldots, \varepsilon_{r,N}) \), \( \varepsilon_{r,i} = 0 \) if security mechanism \( i \) is not used for role \( r \) transactions and equal to 1 otherwise.

The overall system policy is then characterized by the vector \( \vec{\rho} = (\vec{\rho}_1, \ldots, \vec{\rho}_R) \). The controller dynamically adjusts the policy \( \vec{\rho} \) to maintain the system performance and security objectives.

### 3.1.3 Utility Function

The controller optimizes a global utility function. Utility functions have been used to express the value of a system policy to one or more stakeholders as a function of the system objectives and provide a way of trading off between multiple competing system objectives [106]. We consider the global utility function as a function of response time \( T \) and the system security level characterized by the security policy \( \vec{\rho} \). The objective is to design a response time utility function that decreases when the response time increases and a security utility function that increases when the security level increases.

In designing a security utility function, the number of security mechanisms and their detection rate is utilized. Intuitively, combining several diverse mechanisms makes it more difficult for an attack to bypass the entire set of diverse mechanisms. This will result in an increase in the security system effectiveness and reliability. Therefore, it is important to
design a security utility for each role that is a function of the security policy:

\[ U^S_r (\bar{\rho}_r) = f (\bar{\rho}_r), \]  

(3.1)

where the function \( f \) takes into account the security mechanisms applied to role \( r \) as specified by its security policy \( \bar{\rho}_r \). We assume that the detection probabilities of different mechanisms are independent. This can be easily justified because of the diverse mechanisms used. Assuming that the detection rate is a good estimate of the detection probability, the utility of the total detection probability of different mechanisms can be described as follow:

\[ U^S_r (\bar{\rho}_r) = \sum_{j=1}^{A} a_{r,j} \prod_{i=1}^{N} \left( 1 - (1 - \varepsilon_{r,i} \times d_{i,j}) \right)^{-1} \sum_{I \subseteq \{1, \ldots, N\}} \prod_{I \subseteq \{1, \ldots, N\}} P^r_{I,j}, \]  

(3.2)

where \( P^r_{I,j} = \prod_{\forall i \in I} (\varepsilon_{r,i} \times d_{i,j}) \) and \( P^r_{I,j} \) is the probability that role \( r \) attack \( j \) is detected by at least one security mechanism in \( I \). If \( a_{r,j} \) is the probability that attacks of type \( j \) are submitted by role \( r \), then the utility in equation 3.2 is the probability that attacks for users in role \( r \) are detected.

The last two sums represent the probability that at least one of the security mechanisms detects the attack. Due to the combinatorial aspects of (3.2) the full formula can be too cumbersome and it is desirable to simplify it to compute a lower bound on the security utility. Therefore, the goal in designing the security utility function is to design an equation for the security level such that it will be at least as large as the maximum value of the detection rate used in the policy, and the score should increase as additional mechanisms are used, even if the detection probability of the mechanisms is less than the current policy’s maximum mechanism detection probability. For example, consider a policy that uses two mechanisms with detection probability values of 0.31 and 0.46, respectively. The total probability that an attack is detected by any of the two security mechanisms is \( 0.31 + 0.46 - (0.31 \times 0.46) = 0.62 \).
If another mechanism with probability of 0.4 is added to the policy, the new value of the probability will be $0.31 + 0.46 + 0.4 - (0.31 \times 0.46) - (0.31 \times 0.4) - (0.46 \times 0.4) + (0.31 \times 0.46 \times 0.4) = 0.77$.

A lower bound can be obtained through exponential averaging (i.e., $\ln \sum_{i=1}^{N} e^{x_i}$), which produces a value that is at least as large as the maximum value in the data. In the earlier example, exponential averaging for mechanisms with detection probability values of 0.31 and 0.46 normalized in the [0,10] range will produce a value of $\ln(e^{3.1} + e^{4.6}) \div 10 = 0.48$. Adding a mechanism with detection probability of 0.4, the exponential average value becomes $\ln(e^{3.1} + e^{4.6} + e^{4.0}) \div 10 = 0.51$. See Figure 3.5 and Figure 3.6 for example curves for both functions. Therefore, we chose exponential averaging to formulate security utility as follow:

$$U_r^S(p_r) = \sum_{j=1}^{A} a_{r,j} \left( \ln \sum_{i=1}^{N} e^{d_{i,j}} \times \varepsilon_{r,i} \right).$$ (3.3)
The security utility described in the previous equation provides a good measure of the detection quality of the IDPSs combinations. However, IDPSs quality must be characterized by its false detection rate as well as their true detection rate. For example, an IDPS that detects all attacks but reports too many false-alarms is not necessarily better than an IDPS that misses some attacks but reports no false alarms. In fact, most of the IPDS quality measures use detection and false detection rate together to assess its effectiveness of an IDPS. Therefore, the utility function must consider both rates in it is computation. Another important factor in the security utility of a certain configuration is the number of IDPSs used in that policy. The number of IDPSs used in the policy can be thought of as a diversity measure of the policy. The more IDPSs used, the more diverse the security checks performed. One can see that the utility in equation 3.3 can be improved to consider these two important measures.

Using the estimates described earlier, one can extend the previous utility to include the
false detection rate as well as the number of IDPSs. Specifically, let $d_{i,j}$ be the rate at which
IDPS $i$ correctly identifies an attack of type $j$ and is defined as the number of intrusions
detected divided by the total number of intrusions in the observed sample. Similarly, let
$f_{i,j}$ be the frequency with which IDPS $i$ reports an attack of type $j$ in error. Assuming that
the detection rate is a good estimate of the detection probability, the total detection prob-
ability of different combined mechanisms can be estimated using the exponential averaging
described earlier. We use this estimate to formulate a security utility that depends on the
detection rate (true positives), false positive rates for the various attacks for different roles,
and the number of techniques used as follows:

$$D^\text{total}_r (\vec{\rho}_r) = \left[ \sum_{j=1}^{A} a_{r,j} \left( \ln \sum_{i=1}^{N} e^{d_{i,j}} \times \varepsilon_{r,i} \right) \right] \sum_{i=1}^{N} \varepsilon_{r,i}$$

(3.4)

where $A$ is the number of attack types and $a_{r,j}$ is the relative importance of attacks of
type $j$ to users of role $r$ ($\sum_{j=1}^{A} a_{r,j} = 1 \ \forall \ r$). The term $\sum_{i=1}^{N} \varepsilon_{r,i}$ represents the num-
ber of diverse techniques used to evaluate the requests, while the first term in (3.4) (i.e.,
$\left[ \sum_{j=1}^{A} a_{r,j} \left( \ln \sum_{i=1}^{N} e^{d_{i,j}} \right) \right]$) represents the quality of the IDPS combination when compared
to other combinations.

Assuming independence of the combined IDPSs one can estimate the total FP rate $F^\text{total}_r$
by replacing $d_{i,j}$ by $f_{i,j}$ in Eq. (3.4). Thus,

$$F^\text{total}_r (\vec{\rho}_r) = \left[ \sum_{j=1}^{A} a_{r,j} \left( \ln \sum_{i=1}^{N} e^{f_{i,j}} \times \varepsilon_{r,i} \right) \right] \sum_{i=1}^{N} \varepsilon_{r,i}.$$  

(3.5)

We then define a sigmoid-shaped utility function, $U^D_r$, for correct attack detection and a
sigmoid-shaped utility function, $U_{r}^{F}$, for the false positive rate:

$$U_{r}^{D} (\rho_{r}^{d}) = \frac{k}{1 + e^{-\sigma(D_{r}^{total}(\rho_{r}^{d}) - \delta_{r})}}, \quad (3.6)$$

$$U_{r}^{F} (\rho_{r}^{d}) = \frac{k}{1 + e^{-\sigma(F_{r}^{total}(\rho_{r}^{d}) - \delta_{r})}}, \quad (3.7)$$

where $\sigma$ is a sensitivity parameter that defines the sharpness of the curve, $\delta_{r}$ denotes the detection rate goal for role $r$, and $k$ is a normalization constant computed in such a way that the value of the utility function is equal to 1 for the highest level of security (i.e., all IDPSs are used). See an example of a sigmoid security utility functions with different objectives in Fig. 3.7.

The two values can then be combined to obtain a total security utility function for role $r$ as:

$$U_{r}^{S} (\rho_{r}^{d}) = \alpha U_{r}^{D} (\rho_{r}^{d}) + \beta (1 - U_{r}^{F} (\rho_{r}^{d})), \quad (3.8)$$
where $\alpha$ and $\beta$ (s.t. $\alpha + \beta = 1$ and $\alpha, \beta \geq 0$) indicate the preference between detection and false positive rates. The total security utility function is the weighted sum of all roles:

$$U_{\text{total}}^S (\vec{\rho}) = \sum_{\forall r} w_r^s U_r^S (\vec{\rho}_r),$$  

(3.9)

where $w_r^s$ is a weight defined by the system owner to model the security risk and preferences posed by role $r$.

The utility function for the response time $T_r$ for role $r$ can be represented as follows:

$$U_r^T (T_r) = g(\vec{\rho}_r, \vec{d}, \lambda_r),$$  

(3.10)

where $\vec{d} = (o_1, \cdots, o_K, \cdots, o_N)$ is the vector of service demand (CPU overheads), $o_i$, of each IDPS, $\lambda_r$ is the arrival rate of transactions of class (role) $r$. More specifically, we use a sigmoid monotonically decreasing function [44]:

$$U_r^T (T_r) = \kappa_r \frac{e^{\delta(\sigma_r - T_r)}}{1 + e^{\delta(\sigma_r - T_r)}},$$  

(3.11)

where $\delta$ is a sensitivity parameter that defines the sharpness of the curve, $\sigma_r$ denotes the response time goal for role $r$, and $\kappa_r$ is a normalization constant. The value of $\kappa_r$ is computed in such a way that the value of the utility function is equal to 1 when the response time is equal to zero. Thus,

$$\kappa_r = \frac{1 + e^{\delta \sigma_r}}{e^{\delta \sigma_r}}.$$  

(3.12)

See an example in Fig. 3.8 for $\sigma_r = 1, 2, 4$.

The response time total utility function is the weighted sum of all response time utility functions:

$$U_{\text{total}}^T (\vec{T}) = \sum_{\forall r} w_r^t U_r^T (T_r).$$  

(3.13)
Figure 3.8: Utility function for response time.

The total response time function will be a function of the security policy, the overhead of the security mechanisms used for each role, and of the workload intensity (arrival rate $\lambda$).

Thus, the utility function can be written as function $g$ of $\lambda = (\lambda_1, \ldots, \lambda_R)$, $\sigma$, and $\rho$ as:

$$U_{total}^T(\bar{T}) = g(\rho, \lambda, \sigma).$$  \hspace{2cm} (3.14)

The global utility function is a function of all utility functions:

$$U_g(\rho, T) = \alpha U_{total}^T(\bar{T}) + \beta U_{total}^S(\rho),$$  \hspace{2cm} (3.15)

where $\alpha$ and $\beta$ are weights associated with response time and security, respectively, and $\alpha + \beta = 1$ and $\alpha, \beta \geq 0$.

Thus, the global utility can be written as a function $f$ of the policy $\rho$, the IDPSs
overhead $\vec{S}$, the workload $\vec{W}$, and the attack likelihood matrix $A = [a_{r,j}]$:

$$U_g(\vec{\rho}, \vec{T}) = f(\vec{\rho}, A, \vec{W}, \vec{S}).$$ \hspace{1cm} (3.16)

The global utility is then a general measure of the quality of a certain configuration. In contrast, security objectives, response time goals, different weights used in the utility function express specific users’ constraints and preferences that should be maximized but not necessarily always guaranteed.

Different utility functions could be used provided they correspond to rational user expectations such as monotonically decreasing function for response time and monotonically increasing function for security.

### 3.2 Controller Architecture

The controller approach is based on searching the space of values of configurable parameters, i.e., security policies, for a point where the global utility (see Eq. 3.16) is maximized or close to being maximized. The controller uses performance models [69] to evaluate the selected policy $\vec{\rho}$ performance. The use of exhaustive search of all possible configurations is not practical due to the combinatorial size of the search space. Consequently, a heuristic search technique is used to find a near optimal configuration that maximizes the global utility function and meets as close as possible the QoS and security requirements. A state space representing all possible configurations of the system consist of points, where each point in the space represents a security policy and a numerical value associated with the point represents the value of the utility function. Through heuristic search techniques like hill-climbing or beam search, all neighbor configurations are examined and a new configuration with the highest utility value is selected. The search repeats until either the utility does not improve or some maximum number of points traversed has been exceeded.

The architecture of the controller is best described with the help of Figure 3.9. The
picture shows a database system (1) subject to various IDPS systems. These systems are driven by a database of security policies. An autonomic controller dynamically changes the policies in order to optimize the global utility function described in the previous section. The controller has three main components: Workload Analyzer (2), Controller Algorithm (3), and Performance Model Solver (4). The autonomic controller is driven by a Controller Algorithm (3) that runs at regular intervals called controller intervals.

The Workload Analyzer component analyzes the workload characteristics during the controller interval. The current or predicted workload intensity values computed by this component are also used as input parameters to the Performance Model Solver (4) component. This component receives requests from the Controller Algorithm to solve a Queuing
Algorithm 1 Controller Algorithm

```plaintext
Input: \( \vec{\rho}_{\text{curr}} \), Budget

Initialize:
\[ \text{NumEvals} \leftarrow 0; \vec{\rho}_{\text{visited}} \leftarrow \vec{\rho}_{\text{curr}} \]
\[ U_g^{\text{best}} \leftarrow U_g(\vec{\rho}_{\text{visited}}); \vec{\rho}^{\text{best}} \leftarrow \vec{\rho}_{\text{visited}} \]

5: while NumEvals < Budget do
   \[ \vec{\rho} \leftarrow \text{NextNeighbor}(\vec{\rho}_{\text{visited}}) \]
   if \( \nexists \) next neighbor of \( \vec{\rho}_{\text{visited}} \) then break
   end if
   if \( U_g^{}(\vec{\rho}) > U_g^{\text{best}} \) then
      \[ U_g^{\text{best}} \leftarrow U_g(\vec{\rho}); \vec{\rho}^{\text{best}} \leftarrow \vec{\rho} \]
      \( \vec{\rho}_{\text{visited}} \leftarrow \vec{\rho} \)
   end if
   NumEvals \leftarrow NumEvals + 1
end while

15: return \( \vec{\rho}^{\text{best}} \) \( \triangleright \) Return The Best Policy
```

The Controller runs the controller algorithm at the beginning of each controller interval. The search for a new configuration is based on a greedy hill climbing-method [101]. The detailed description of the controller approach is given in Algorithm 1. From the current policy \( \vec{\rho}_{\text{curr}} \), we examine all adjacent policies or neighbors. The neighbors of a policy \( \vec{\rho} \) are the security policies that differ from \( \vec{\rho} \) in only one position. For example, the neighbors of \( (1,0,1,1) \) are \( (0,0,1,1), (1,1,1,1), (1,0,0,1), \) and \( (1,0,1,0) \).

For every neighbor, the algorithm uses the performance model solver to compute the configuration’s expected response time, and in turn uses it to compute the utility value of this neighbor. If the neighbor’s utility is better than that of the current configuration, the search continues with neighbors of the current configuration. The search continues until either no improvements can be made or a maximum number of policies are visited.

Normally, databases are a part of a multi layer applications such as web application.
Typical layers include presentation layer, business logic layer, and data layer. The presentation and business logic layer are usually referred to as the application. As such, Web applications can be seen as an application layer. Likewise, databases can be seen as the data layer. In this kind of layered architecture the controller sits between the data layer and the application layer. Fig. 3.10. The picture shows an autonomic security manager (ASM) that orchestrates between the DB application and the various IDPSs in a web application. These systems are driven by the ASM database of security policies. An autonomic controller dynamically changes the policies in order to optimize the global utility function described in the previous section.

The controller has five main components: Performance Monitor (1), Workload Analyzer (2), Controller Algorithm (3), Utility Function Computation (4) and Performance Model
Solver (5). The Performance Monitor profiles the application at run time to obtain performance statistics such as execution time and number of requests, and then forwards them to the Workload Analyzer. This component analyzes the workload characteristics during the CI to obtain values that will be used in computing the predicted response time. The predicted workload intensity values computed by this component are also used as input parameters to the Performance Model Solver component. This component receives requests from the Utility Function Computation component to solve a Queuing Network (QN) performance model [69] corresponding to a specific security policy configuration of the system. This component takes as input parameters to the QN model the policy parameter values $\vec{\rho}$ and workload intensity values $\vec{W}$. The output of the QN model is the set of resulting performance values for the specific policy used with the predicted workload. The Utility Function Computation component uses the predicted performance values to compute the utility of the proposed configuration ($\vec{\rho}$) and passes this value to the Controller Algorithm.

One limitation to the greedy hill climbing approach described above is that it is more likely to get stuck in a local optimal of the search space. To avoid this, the controller algorithm uses a random restart hill climbing approach. Specifically, if the neighbors of $\vec{\rho}$ offer no improvements a new policy is randomly selected for the next set of neighbors. The new policy is selected in such a way that the same number of active IDPSs (i.e., ones) are selected for the role, however, in a different order. The detailed description of the controller approach is given in Algorithm 2.

In some situations, the system might be subjected to high workloads as a part of normal conditions (e.g., unexpected high demands due to new product) or due to malicious activities (e.g., Denial of Service attacks). These workloads may result in a security policy with no security mechanisms. Therefore, in order to maintain a lower bound on the system security level, one can add constraints on $\vec{\rho}$ to guarantee a minimum security level regardless of the workload intensity and QoS goals.
3.3 The Queuing Network Model

In this section we discuss the queuing network (QN) models used by the controller to estimate the performance of the different configuration under evaluation. The section first discuses the use of open models then closed models are discussed.

3.3.1 Open Model

The controller discussed in the previous section requires the computation of the QoS values for a given configuration. This is obtained through the use of a QN model as in Fig. 3.11. Requests arrive to be serviced by the DB application. The controller, based on the policy, forwards the request directly to the database or to one or more of the syntax-centric mechanisms. Once syntax-centric evaluation is completed for the request, it is then sent to the database. Once the requests are serviced by the database, they are either sent back to the user or sent to one or more data-centric approaches. The IDPS mechanisms can work concurrently. However, the controller must wait for all the mechanisms to finish before it makes a decision. Therefore, the syntax-centric and data-centric mechanisms are modeled

\textbf{Algorithm 2} Controller Algorithm - Random Restart

\begin{verbatim}
Input: \( \rho_{\text{curr}}, \text{Budget}, U_g^{\text{previous}} \)
Initialize:
\( \text{NumEvals} \leftarrow 0; \ \overrightarrow{\rho}_{\text{visited}} \leftarrow \overrightarrow{\rho}_{\text{curr}} \)
\( U_g^{\text{best}} \leftarrow U_g(\overrightarrow{\rho}_{\text{visited}}); \ \overrightarrow{\rho}_{\text{best}} \leftarrow \overrightarrow{\rho}_{\text{visited}} \)
\( \text{found} \leftarrow \text{false}; \)
\textbf{while} \( \text{NumEvals} < \text{Budget} \) \textbf{do}
  \textbf{for all} \( \forall \) neighbors of \( \overrightarrow{\rho}_{\text{visited}} \) \textbf{do}
    \( \overrightarrow{\rho} \leftarrow \text{NextNeighbor}(\overrightarrow{\rho}_{\text{visited}}) \)
    \textbf{if} \( \exists \) next neighbor of \( \overrightarrow{\rho}_{\text{visited}} \) \textbf{then} \textbf{break}
    \textbf{end if}
  \textbf{if} \( U_g(\overrightarrow{\rho}) > U_g^{\text{best}} \) \textbf{then}
    \( U_g^{\text{best}} \leftarrow U_g(\overrightarrow{\rho}); \ \overrightarrow{\rho}_{\text{best}} \leftarrow \overrightarrow{\rho} \)
  \textbf{end if}
\textbf{end for}
\( \text{NumEvals} \leftarrow \text{NumEvals} + 1 \)
\textbf{end while}
\textbf{return} \( \overrightarrow{\rho}_{\text{best}} \)
\end{verbatim}

\( \triangleright \) Randomly restart after examining all neighbours
as concurrent (i.e., fork and join) operations.

Requests are assumed to arrive to the system according to a Poisson process with rate $\lambda_r$ for requests coming from role $r$. The execution of a request consists of 1) controller evaluation, 2) syntax-centric mechanisms if required by the policy, 3) database execution (alternating between the CPU and the disks), and 4) data-centric mechanisms if required by the policy.

The QN model in Fig. 3.11 is a composition of single components (DB, IDPSs). The IDPSs are modeled as fork and join sub-networks. Models for the database, web, and application servers are built using methods described in [69]. The database subsystem in Fig. 3.11 is composed of $L$ devices (e.g., CPU and disks) and receives requests at rates given by $\vec{\lambda} = (\lambda_1, \ldots, \lambda_R)$, where $\lambda_r$ is the arrival rate for class (role) $r$. The service demand law [69] states that $U_{i,r} = \lambda_r \times o_{i,r}$ where $U_{i,r}$ is the utilization of device $i$ by requests of class $r$ and $o_{i,r}$ is the service demand (i.e., total service time) of requests of class $r$ at device $i$. In equilibrium, the total utilization of any device must be less than one (i.e., $U_i = \sum_{r=1}^{R} U_{i,r} < 1$).

Once $U_i$ is known, one can use the arrival theorem and Little’s Law [69] to obtain the
average response time for requests of class $r$:

$$T_r = \sum_{i=1}^{L} \frac{o_{i,r}}{1-U_i}. \tag{3.17}$$

The syntax and data-centric security mechanisms in the model are non-product form queuing sub models in which requests are replicated and serviced in parallel on different queues with different service times for each queue. Exact and efficient solutions for fork and join queues are only known for two servers [88]. Consequently, approximation techniques have been used to compute average response time for fork and join networks [87–89, 92, 107]. None of these approximations considers fork and joins with heterogeneous parallel servers. In chapter 4 we propose the following approximation for the average response time of requests of class $r$ as they pass through a fork and join subnetwork with $D$ heterogeneous servers:

$$T_r = \sum_{i=1}^{D} \frac{1}{i} \times \frac{o_{i,r}}{1-U_i}, \tag{3.18}$$

where $o_{i,r}$ is the service demand (i.e., processing overhead) of requests of class $r$ at IDPS $i$. Equation 3.18 is exact when there is only one server in the fork and join (i.e., $D = 1$). The term $1/i$ in the summation accounts for the synchronization time at the join point and gives more weight to servers with larger service demands. The rationale for this approximation comes from the fact that the expectation of a random variable defined as the maximum of $k$ independent and exponentially distributed random variables with average $S$ is equal to $S \times H_k$ where $H_k = \sum_{i=1}^{k} 1/i$. The utilization of device $i$ in Eq. 3.18 is computed as $U_i = \sum_{r=1}^{R} \lambda_r \times o_{i,r} \times \varepsilon_{r,i}$ to account for the specific role assignment policy. Then, algorithm 3 is used to solve the QN shown in Figure3.11.
### Algorithm 3 Performance Model Algorithm

Input: $K, M, L, \lambda, o, \rho$

for $r = 1 \rightarrow R$ do
  for $i = 1 \rightarrow N$ do
    $U_{i,r} \leftarrow \lambda_r \times o_{i,r} \times \varepsilon_{r,i}$ \hfill \triangleright IDPSs Utilization
  end for
  for $i = 1 \rightarrow L$ do
    $U_{i,r} \leftarrow \lambda_r \times o_{i,r}$ \hfill \triangleright DB Utilization
  end for
end for

for $r = 1 \rightarrow N + L$ do
  $U_i \leftarrow \sum_{r=1}^{R} U_{i,r}$ \hfill \triangleright Fork and Join approximation
end for

$R_K = \sum_{i=1}^{K} \frac{1}{1-U_i} \times \frac{o_{i,r}}{1-K}$ \hfill \triangleright Syntax centric

$R_M = \sum_{i=K+1}^{K+M} \frac{1}{1-K} \times \frac{o_{i,r}}{1-U_i}$ \hfill \triangleright Data centric

$R_{DB} = \sum_{i=1}^{L} \frac{o_{i,r}}{1-U_i}$

$T_r \leftarrow R_K + R_M + R_{DB}$

return $T_r$ \hfill \triangleright Average Response Time

#### 3.3.2 Closed Model

Section 3.3 discusses the use of open models. However, many performance models in general and multi threaded multi layer similar to the one shown in Fig. 3.10 can be modeled by closed QN. This type of model is called a closed QN and is very effective when modelling multi-threaded applications such as Web servers where the performance depends on the concurrency level of the system. In the QN model of Fig. 3.12, $N$ concurrent customers submit requests to the application and then wait for some time (called the think time) after receiving a reply before submitting a new request. Requests move in the model alternating between different components until the execution is complete. The syntax-centric and data-centric mechanisms are modeled as concurrent (i.e., fork and join) operations in the QN model because the ASM must wait for all the mechanisms to finish before it makes a decision.

The QN model in Fig. 3.12 is a composition of single components (e.g., processors, DB disks, IDPSs). The IDPSs are modelled as fork and join sub-networks. Because requests from different roles may have different service times, we use a multiclass QN model in which each role corresponds to a different class. Typically, closed QN models can be solved using Approximate Mean Value Analysis (AMVA) techniques [69]. AMVA is an iterative solution
technique that takes as input the workload intensities, the set of devices in the model, and
the matrix of service times for different model components and generates as output predicted
performance measures per role expressed as $\vec{T} = (T_1, \ldots, T_R)$, and $\vec{X} = (X_1, \ldots, X_R)$, where
$X_r$ is the average throughput of role $r$ requests. However, due to the non-product form
nature of the QN model (i.e., concurrent IDPS execution), we resort to the approximations
presented in [104] to solve the QN model of Fig. 3.12.

The Workload Analyzer component computes the concurrency level and the think times
based on values obtained from the Performance Monitor. Specifically, the concurrency level
is computed as $N_r = \sum_{j} T^j / CI$ where $T^j_r$ is the response time of the $j^{th}$ transaction of role
$r$ executed during a controller interval of duration $CI$. The think time for role $r$ can be
estimated as $Z_r = N_r / X_r - T_r$ using the Interactive Response Time Law [69].

3.4 Practical Considerations

Practically, values can be initially estimated using a site-specific risk analysis and updated
with new traffic accordingly. For example, detection rate values can be estimated using
suitable training data sets and historical information. Initial attack likelihood values can be estimated by analyzing the past behavior of the system leveraging knowledge of user behavior and system administrators concerns. Traffic and attack history are then used to update these values. Typical production environments have a very large number of users. Thus, keeping a normal access profile for each user is impractical. The first step to deal with the large number of users is to classify them into roles. RBAC has been proposed to build a profile for each role and check the behavior of each role with respect to such profile [3]. If RBAC is not being used, the database logs could be clustered to define roles and their expected behavior [2]. Then, attack categories to the database are identified. A second-level classification is then formulated based on (1) the specific scenario to which an attack category applies, and (2) the type of users who are most likely to be the source of such attacks. A catalog of attack probabilities, for each attack category and for each role, as well as the detection probability for each security mechanism can be built by performing rigorous system analysis of past events and historical data related to database threats and attacks. It should be emphasized that although obtaining accurate values is a challenging task our target is rather measuring the relative ordering of different security configuration (i.e., different combinations of IDSs) in terms of their effectiveness and associated risks. More specifically, for the controller to be effective, it needs to qualitatively measure the security level in relative terms to other configurations that are being considered.

The performance related values can be measured experimentally. For example, one can use the total resource utilization available from performance monitors to compute per role utilization. This can be computed as $U_{i,r} = U_i \times f_{i,r}$, where $f_{i,r}$ is an apportionment factor for role $r$ at device $i$ (see [69]). Then, the service demand $o_{i,r}$ can be obtained using the Service Demand Law as $o_{i,r} = U_{i,r}/\lambda_r$ [69].
3.5 Concluding Remarks

This chapter described the controller architecture for the ISQDB. It further described how the IDPSs metrics can be estimated and how the utility functions are obtained. The chapter also discussed the QN models that can be used to estimate the response time for a given configuration. The model solutions and algorithms are described in this section as well. It also briefly discussed some practical information related to the controller.
Chapter 4: Response Time Approximation for Multi-Class Fork and Join in Queuing Networks

This chapter presents approximations for parallelism in computer systems modeled as fork and join (FJ) constructs in analytic models. Particularly, we define the fork and join and the notations that will be used in the chapter. The first section considers the open queuing networks case. The second section considers the closed queuing networks case. In each case, approximations for single class and multi-class queuing networks are presented and evaluated. The work presented in this chapter appears in [104].

4.1 Fork and Join

In the FJ construct shown in Fig.4.1, there are $K$ branches and each branch has a single queue. Jobs are split on arrival into $J$ ($J \leq K$) tasks to be serviced in parallel by $J$ queues. Only when all $J$ parallel tasks of a job complete, the job can rejoin (synchronize) and leave the system. A job can be split to every queue in the FJ ($J = K$) (i.e., full fork) or to a certain number of queues in a probabilistic way (i.e., dynamic fork). In the dynamic fork case, jobs are first split into a random number of $J$ ($1 \leq J \leq K$) tasks. Then, $J$ specific queues are selected in a probabilistic way. In multiclass models, each job class can be routed differently in the FJ (i.e., configurable fork). A single queue in the FJ operates like a standard queuing system. When the service times distributions and their moments are the same at all queues visited by the tasks of a job in a FJ construct, we have a homogeneous FJ. Otherwise, we have a heterogeneous one.

The synchronization requirements of a FJ add significant modeling complexity. The only exact method available to solve general networks with non product-form characteristics is to determine and solve the Markov process representation of the network. Generalized
Stochastic Petri Nets (GSPN), can be used to specify a system and facilitate the automatic
generation of an underlying Markov process, whose solution can be used to compute the
performance metrics of interest [83]. Even if the Markov process representation of a GSPN
can be found, the solution is usually too costly unless the network has a small state space.
However, this is not the case for many practical problems. Therefore, approximation tech-
niques have been proposed to estimate performance measures for FJ constructs in QNs.
However, often these approximations are designed with a set of restrictive assumptions that
limit their applicability for more general performance planning and design scenarios.

4.2 Fork and Join Approximation

We first address open QNs and their extensions. Then, we address closed QNs and show how
MVA can be adapted to solve single class closed models. Finally, we present an algorithm
to solve multiclass closed QNs with FJ sub networks.

4.2.1 Open FJ Model

Figure 4.1 depicts a multiclass FJ queue with \( K \geq 2 \) heterogeneous queues. There are
\( R \) different classes of jobs numbered \( 1, \cdots, r, \cdots, R \). The vector \( \vec{S} = (\vec{s}_1, \cdots, \vec{s}_K) \), where
\( \vec{s}_j = (s_{j,1}, \cdots, s_{j,r}, \cdots, s_{j,R}) \) specifies the service demands (in seconds) of the tasks of jobs of classes 1 through \( R \) at queue \( j \) \((j = 1, \cdots, K)\) of the FJ. For the single class case we drop the class subscript and use the notation \( \vec{S} = (s_1, \cdots, s_K) \). Jobs of class \( r \) \((r = 1, \cdots, R)\) are assumed to arrive at a rate of \( \lambda_r \) jobs/sec. We denote by \( \vec{\lambda} = (\lambda_1, \ldots, \lambda_r, \ldots, \lambda_R) \) the vector of arrival rates of jobs of all classes.

Each class of jobs forks into \( 1 \leq J \leq K \) tasks based on a routing matrix \( A = [K \times R] \). The \( r^{th} \) column of the matrix \( A \) is the vector \( \vec{\varepsilon}_r = (\varepsilon_{1,r}, \cdots, \varepsilon_{K,r}) \) that prescribes how class \( r \) jobs are routed to the \( K \) servers in the fork and join sub-network. For example, if \( K = 3 \) and \( \vec{\varepsilon}_2 = (1, 0, 1) \) jobs of class 2 will branch into queues 1 and 3. After receiving service by the servers at these two queues, tasks proceed to the join point and wait until all its siblings complete service. Only then, the job proceeds to the next queue in the QN or leaves the QN.

Intuitively, a job's service time at a FJ construct is at least equal to the time spent at the server with maximum service time. More specifically, the time it takes to service \( J \) job siblings concurrently that must synchronize after they have all completed is at least \( \hat{S} = \max_{i=1}^{J} \{s_i\} \). However, since it is possible for different jobs to overlap and wait to synchronize before they leave the join, the service time becomes, at most \( O_J \times \hat{S} \), where \( O_J \) is a scaling factor by which \( \hat{S} \) must be increased in order to compute the time it takes for the job to synchronize and leave the FJ. In probability theory, \( O_J \) is known as the mean of the \( J^{th} \) order statistics and the value of \( O_J \) is dependent on the distribution of \( s_i \) [108]. Similar observations can be made about the waiting time. Thus, the waiting time is described by the \( J^{th} \) order statistics of the maximum waiting time of \( J \) siblings \((W \times O_J)\). In view of the above observations about the service and waiting times, one can see that the mean response time, \( T \), of jobs in the FJ is bounded by:

\[
\hat{S} \times O_J \leq T \leq O_J \times (W + \hat{S}) \tag{4.1}
\]
The lower bound in equation (4.1) is obtained by neglecting the waiting time. The upper bound on the other hand, assumes that all job’s siblings will wait at the most utilized server.

The same intuitive observations are made, and a formal proof is provided in [84, 88]. Furthermore, [84, 88] make use of two assumptions to derive and prove theoretical bounds on the mean response time. The first assumes that all servers in the FJ are homogeneous (i.e., \( s_1 = s_2, \cdots = s_K \)). The second assumes that the service times are exponentially distributed with the same mean. Under these assumptions, one can show that if all service times are exponentially distributed with the same mean, the scaling factor \( O_J \) equals the \( J^{th} \) harmonic number \( H_J \), where \( H_J = \sum_{i=1}^{J} 1/i \).

Let us now consider a heterogeneous case with different exponentially distributed service times. Intuitively, this case follows the same rationale as in the homogeneous case. More specifically, the time it takes to service a job will be at least \( \hat{S} = \max_{i=1}^{J} \{s_i\} \). However, in the heterogeneous case, jobs will take less time on average to synchronize because jobs completing from the queues with the highest service times are less likely to wait for jobs of other queues to complete since the other queues have lower service times (i.e., they are faster). In this case, it is harder to efficiently compute a scaling factor for different exponentially distributed service times. Therefore, to obtain a reasonably justified response time approximation, we make use of the following observations: 1) the mean response time is most sensitive to servers with the highest service times; 2) the mean response time grows with the number of parallel servers and with the service time at each server; 3) the mean response time of the heterogeneous case is at most that of the homogeneous case, as long as the maximum service time in the heterogeneous case is equal to that of the homogeneous case, and both have the same number of servers. In view of the previous observations, a reasonable approximation for the mean response time, \( R_r \), for class \( r \) requests is obtained by reordering the queue numbers in the FJ based on the service demands of each queue and on the routing associated to each queue.

Before we present the approximation for \( R_r \), we need to define some notation. For each class \( r \) (\( r = 1, \cdots, R \)) define a sequence \( V_r = (v_r(1), v_r(2), \cdots, v_r(K)) \) such that:
1. \( v_r(k) \in \{1, 2, \ldots, K\} \text{ } \forall k = 1, \ldots, K \),

2. \( v_r(k) \neq v_r(k') \) for \( k \neq k' \), and

3. \( s_{v_r(1),r} \times \varepsilon_{v_r(1),r} \geq s_{v_r(2),r} \times \varepsilon_{v_r(2),r} \geq \cdots \geq s_{v_r(K),r} \times \varepsilon_{v_r(K),r} \).

In other words, \( V_r \) is a reordering of the original queue numbers in the FJ sorted by the third condition above. Then,

\[
R_r = \sum_{k=1}^{K} \frac{1}{k} \times T_{v_r(k)},
\]

where \( T_{v_r(k)} \) is the average response time of class \( r \) requests at queue \( v_r(k) \).

It should be noted that if all the service times are equal, equation (4.2) is similar to the approximations presented in [84, 88]. However, the mean response time approximation in [88] grows as a function of the number of parallel siblings, while in our approximation the mean response time grows as a function of the number of parallel siblings and the service time for each sibling. If the service times are exponentially distributed and the jobs arrive from a Poisson process, we can use the M/M/1 single queue results [109] and rewrite equation (4.2) as:

\[
R_r = \sum_{k=1}^{K} \frac{1}{k} \times \varepsilon_{v_r(k),r} \times s_{v_r(k),r} \frac{1}{1 - \rho_{v_r(k)}},
\]

where \( \rho_{v_r(k)} \) is the total utilization of FJ queue number \( v_r(k) \). The utilization of queue \( k \) \( (k = 1, \ldots, K) \) is computed as

\[
\rho_k = \sum_{r=1}^{R} \lambda_r \times s_{k,r} \times \varepsilon_{k,r},
\]

Equation (4.3) gives more weight to queues with larger service demands because service times are sorted in decreasing order. The routing vector \( \varepsilon \) accounts for the not used servers
so that when a server is not used by a certain class it has no impact on the response time of that class.

For example, consider the open FJ shown in Fig. 4.2 with two job classes and three queues in the FJ. Jobs of class 1 fork into queues 1 and 3 only while jobs of class 2 fork into all three queues. The queue utilizations are computed assuming that the arrival rates for jobs of classes 1 and 2 are $\lambda_1 = 0.2$ jobs/sec and $\lambda_2 = 0.1$ jobs/sec, respectively. According to these products, $V_1 = (1, 3, 2)$ and $V_2 = (3, 1, 2)$. Therefore, according to Eq. (4.3), $R_1$ is computed as

$$R_1 = \frac{1}{1} \cdot \frac{1}{1 - 0.7} + \frac{1}{2} \cdot \frac{1}{1 - 0.8} + \frac{1}{3} \cdot \frac{1}{1 - 0.1}$$

$$= 9.17 \text{ sec}$$  \hspace{1cm} (4.5)

and $R_2$ as

$$R_2 = \frac{1}{1} \cdot \frac{6}{1 - 0.8} + \frac{1}{2} \cdot \frac{3}{1 - 0.7} + \frac{1}{3} \cdot \frac{1}{1 - 0.1}$$

$$= 35.37 \text{ sec.}$$  \hspace{1cm} (4.6)
The values obtained for $R_1$ and $R_2$ by using simulation with a 95% confidence interval are $8.32 \pm 0.09$ sec and $33.04 \pm 0.23$ sec, respectively, which represents 9.03% and 5.9% relative errors. See next section 4.2.2 for more details. It should be noted that more about the validation and comparison with simulation is presented later in the chapter.

According to equation (4.3), when class $r$ jobs are only forked to one server (i.e., $K = 1$), the response time for class $r$ becomes the exact response time for M/M/1 queues (i.e., $R_r = s_{k,r}/(1 - \rho_k)$).

### 4.2.2 Example of a Configurable Multiclass Open FJ

Table 4.1 illustrates an example of an open FJ situation with two job classes and three queues in the FJ. Jobs of class 1 fork into queues 1 and 3 only while jobs of class 2 fork into all three queues. The queue utilizations are computed assuming that the arrival rates for jobs of classes 1 and 2 are $\lambda_1 = 0.2$ jobs/sec and $\lambda_2 = 0.1$ jobs/sec, respectively. The table also shows for each class the products $s_{k,r} \times \varepsilon_{k,r}$. According to these products, $V_1 = (1, 3, 2)$ and $V_2 = (3, 1, 2)$. Therefore, according to Eq. (3), $R_1$ is computed as

$$
R_1 = \frac{1}{1} \cdot \frac{1}{1 - 0.7} + \frac{1}{2} \cdot \frac{1}{1 - 0.8} + \frac{1}{3} \cdot \frac{0.4}{1 - 0.1} = 9.17 \text{ sec} \quad (4.7)
$$

and $R_2$ as

$$
R_2 = \frac{1}{1} \cdot \frac{6}{1 - 0.8} + \frac{1}{2} \cdot \frac{3}{1 - 0.7} + \frac{1}{3} \cdot \frac{1}{1 - 0.1} = 35.37 \text{ sec}. \quad (4.8)
$$

The values obtained for $R_1$ and $R_2$ by using simulation with a 95% confidence interval are $8.32 \pm 0.09$ sec and $33.04 \pm 0.23$ sec, respectively, which represent 9.03% and 5.9% relative errors. These errors are computed as follows: if the analytic model result is within the
simulation confidence interval, the error is zero; otherwise, the error is computed relative to the closest bound of the simulation confidence interval.

<table>
<thead>
<tr>
<th>Class</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing policy ($\vec{e}_r$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Service times (in sec)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Queue utilizations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td>0.1</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>($s_{k,r} \times \varepsilon_{k,r}$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Order of queues ($V_r$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$V_1$</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$V_2$</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### 4.2.3 Dynamic Fork Model

In the FJ considered earlier, jobs of each class were split into exactly $J$ ($J \leq K$) branches. However, in many parallel systems such as RAID disks, jobs may be split into any random number $J$ ($1 \leq J \leq K$) of tasks that are forked (i.e., routed) to queues in the FJ in a probabilistic way. For example, in a RAID disk system, requests are distributed to different number of disks. Another example can be seen in distributed computing where the number of parallel job siblings and their assignment depend on available resources capacity. This kind of dynamic fork arises quite often in heterogeneous parallel systems where the degree of parallelism (i.e., number of parallel siblings) and their assigned resources are determined dynamically to take advantage of faster systems by assigning more jobs to faster queues.
Each queue in the FJ is visited by a certain class of jobs with a certain probability. We assume without loss of generality that the probability of class $r$ jobs visiting queue $k$ is defined by the vector $\vec{\varepsilon}_r$. In other words, $\varepsilon_{k,r}$ represents the probability that queue $k$ is visited by class $r$ jobs. If queue $k$ is not visited by class $r$ jobs, $\varepsilon_{k,r} = 0$. If every class $r$ job visits queue $k$, $\varepsilon_{k,r} = 1$.

It should be noted that $\sum_{k=1}^{K} \varepsilon_{k,r}$ is not necessarily equal to one. For example, consider a special case of dynamic fork in which the number of queues visited by class $r$ jobs is given by the random variable $\tilde{J}_r$ and that the probability of selecting a set of $k$ queues (when $\tilde{J}_r = k$) is the same for all sets of $k$ queues. Then, $\varepsilon_{k,r}$ can be computed as

$$\varepsilon_{k,r} = \sum_{k=1}^{K} Pr [\tilde{J}_r = k] \frac{\binom{K - 1}{k - 1}}{\binom{K}{k}}$$

$$= \frac{1}{K} \sum_{k=1}^{K} k \times Pr [\tilde{J}_r = k]. \quad (4.9)$$

The expression in equation (4.9) is equal to the ratio between the average number of queues selected by class $r$ jobs and the total number of queues in the FJ.

The average response time, $R_r$, for the dynamic case is computed using the process described before for the static and configurable case, with the interpretation that $\varepsilon_{k,r}$ is the probability that class $r$ jobs visit queue $k$.

Consider the FJ in Fig. 4.2 where the service times are the same as the ones in the example presented before. The routing policy is different and therefore, the order of the queues, the utilizations, and the average response times are different: $R_1 = 5.27$ sec and $R_2 = 4.10$ sec. See next section 4.2.4 for more details. The values obtained for $R_1$ and $R_2$ by using simulation with 95% confidence intervals are $5.52 \pm 0.07$ sec and $4.24 \pm 0.04$ sec,
Table 4.2: Example of a Dynamic Multiclass Open FJ

<table>
<thead>
<tr>
<th>Class</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing policy ($\vec{r}$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.7</td>
<td>0.5</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>0.4</td>
<td>0.6</td>
<td>0.3</td>
</tr>
<tr>
<td>Service times (in sec)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Queue utilizations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>($s_{k,r} \times \epsilon_{k,r}$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.4</td>
<td>2.0</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>1.2</td>
<td>0.6</td>
<td>1.8</td>
</tr>
<tr>
<td>Order of queues ($V_r$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$V_1$</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>$V_2$</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

respectively, which represents 3.3% and 2.1% relative errors.

In general QN models, the FJ is one of the devices of the QN. Then, the QN model can be solved by composing all the queues as shown in [69] where the FJ mean response time at the FJ device is computed as described above.

4.2.4 Example of a Dynamic Multiclass Open FJ

Table 4.2 illustrates an example of a dynamic multiclass open FJ. The service times are the same as the ones in the example presented before. The routing policy vectors are different and therefore, the order of the queues, the utilizations, and the average response times are different: $R_1 = 5.27$ sec and $R_2 = 4.10$ sec. The values obtained for $R_1$ and $R_2$ by using simulation with 95% confidence intervals are $5.52 \pm 0.07$ sec and $4.24 \pm 0.04$ sec, respectively, which represent 3.3% and 2.4% relative errors.
4.2.5 Closed FJ QN Model

In many scenarios, one may need to compute performance values for FJ constructs in closed and mixed QN models. In a closed model, there are a finite number of jobs cycling through the system. Closed QNs are quite useful for modeling batch or terminal workloads, such as, a client server system with a known number of clients, heavy load systems, and batch workloads.

The workload intensity parameters in such models are given by the number of customers $N$ in the QN. In multiclass QNs, the workload intensity is given by $\vec{N} = (N_1, \ldots, N_R)$ where $N_r$ is the number of class $r$ jobs in the QN. The next sections show how MVA can be extended to include our approximation for heterogeneous FJ sub-networks for single-class and multiclass closed QNs.

4.2.6 Heterogeneous FJ Single Class Mean Value Analysis (MVA)

The arrival theorem [96] states that in a closed QN, the average number of jobs seen by an arriving job at a device is equal to the average queue length at that device with one less customer in the QN. In MVA, this theorem is used to compute the residence time of a job at a device by adding the job service time to the time it takes to service all the jobs found at the device by the arriving job. Since jobs in FJ constructs are split between the different queues in the fork and join and then rejoined, FJ constructs can be seen as single devices in the QN. For example, if we have a QN with a 3-queue FJ and two regular queues, one can think of this as a closed QN with three devices. An extension of MVA was presented in [85] to compute approximate performance values for QNs with FJ subsystems. However, that extension only considers homogeneous FJ networks. Their approximation assumes that the queues length in the FJ are approximately the same. More specifically, in [85] the FJ sub-network is considered a single device in the QN and the residence time of the FJ sub network is computed as $s_i[H_k + \bar{n}_i(n - 1)]$, where $\bar{n}_i(n - 1)$ is the average queue length at device $i$ (the FJ construct) when there is one less customer in the QN and $H_k = \sum_{j=1}^{k} 1/j$
is the $k$-th harmonic number.

When the service demands in all FJ queues are equal, it is fair to assume that the average queue lengths in the FJ queues are equal. However, in the heterogeneous case, the average queue lengths will be different. Therefore, based on our earlier observations, we propose an approximation that is different from the traditional MVA algorithm and from the extension presented in [85] for homogeneous FJ subnetworks. Our approximation starts when there are zero customers in the QN and proceeds as follows:

1. Let $M$ be the number of non-FJ queues (numbered $1, \cdots, M$) in the QN and let the queues in the FJ queue be numbered $M+1, \cdots, M+K$. Consider each branch of the FJ sub network as a regular queue. Let $s_i$ ($i = 1, \cdots, M+K$) be the service demand of queue $i$. Let $\varepsilon_i$ ($i = M+1, \cdots, M+K$) be the routing probability of queue $i$ in the FJ. To simplify the notation below, let $\varepsilon_i = 1$ for $i = 1, \cdots, M$.

2. Compute the residence time $R_i'(n)$ ($i = 1, \cdots, M+K$) for all queues in the QN, including the $K$ queues in the FJ subnetwork using the standard MVA equation as follows:

$$R_i'(n) = \varepsilon_i \times s_i [1 + \bar{n}_i(n-1)] \quad i = 1, \cdots, M+K,$$

where $\bar{n}_i(n-1)$ is the average queue length at queue $i$ when there is one less customer in the QN. Note that by multiplying the service demand by $\varepsilon_i$ we can handle the configurable and dynamic cases.

3. Reorder the residence times of the FJ devices in descending order.

4. Compute the overall FJ device residence time as follows:

$$R^\ast(n) = \sum_{k=M+1}^{M+K} \frac{1}{k-M} R_k'(n),$$

where $R_{M+1}'(n) \geq R_{M+2}'(n) \geq \cdots \geq R_{M+K}'(n)$. 

59
5. Compute the system throughput by applying Little’s Law and considering that the response time is equal to the residence time at the FJ device plus the sum of the residence times at all non-FJ queues:

\[ X_0(n) = \frac{n}{R^*(n) + \sum_{i=1}^{M} R'_i(n)}. \] (4.12)

6. Compute the average queue length for each queue in the QN, including those in the FJ device using the standard MVA equation based on the residence times of each queue and the overall throughput as follows:

\[ \bar{n}_i(n) = X_0(n) \times R'_i(n) \quad i = 1, \ldots, M + K. \] (4.13)

Due to the fact that the queue length is zero when there are no customers in the network \( \bar{n}_i(0) = 0 \), the previous equations can be used recursively to calculate the residence time in equation (4.10). The residence time can then be used to calculate the average response time in the QN. Contrary to the FJ extension to MVA presented in [85], we do not use the residence time equation for the FJ construct to obtain the queue length at the FJ. Rather, we use the residence times at each individual queue of the FJ. A detailed numerical example is given in section 4.2.7.

The formulation above was presented for a closed QN with a single FJ construct. It is easy to see that the process described above can be easily applied to situations when there are multiple FJ constructs in the QN.

### 4.2.7 Example of a Dynamic Single Class Closed FJ

Table 4.3 shows an example of the computations for a single-class MVA QN with four queues \( M + K = 4 \). The first queue is a delay device (i.e., a non-queueing device that represents the think time of 0.5 sec). Queues 2 through 4 are part of a FJ. The service times at these queues are 1 sec, 4 sec, and 3 sec, respectively, and the routing probabilities
Table 4.3: Example of a Dynamic Single Class Closed FJ

<table>
<thead>
<tr>
<th>n</th>
<th>$R_1$</th>
<th>$R_2'$</th>
<th>$R_3'$</th>
<th>$R_4'$</th>
<th>$R^*$</th>
<th>$X_0$</th>
<th>$\bar{n}_1$</th>
<th>$\bar{n}_2$</th>
<th>$\bar{n}_3$</th>
<th>$\bar{n}_4$</th>
<th>$R_{sim}$</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0%</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
<td>1</td>
<td>2</td>
<td>2.1</td>
<td>3.43</td>
<td>3.93</td>
<td>0.25</td>
<td>0.13</td>
<td>0.25</td>
<td>0.51</td>
<td>0.53</td>
<td>4.17</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>1.25</td>
<td>3.02</td>
<td>3.22</td>
<td>5.15</td>
<td>5.65</td>
<td>0.35</td>
<td>0.18</td>
<td>0.44</td>
<td>1.07</td>
<td>1.14</td>
<td>5.73</td>
</tr>
<tr>
<td>3</td>
<td>0.5</td>
<td>1.44</td>
<td>4.14</td>
<td>4.50</td>
<td>7.05</td>
<td>7.55</td>
<td>0.40</td>
<td>0.20</td>
<td>0.57</td>
<td>1.64</td>
<td>1.79</td>
<td>7.76</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>1.57</td>
<td>5.29</td>
<td>5.85</td>
<td>9.02</td>
<td>9.52</td>
<td>0.42</td>
<td>0.21</td>
<td>0.66</td>
<td>2.22</td>
<td>2.46</td>
<td>9.43</td>
</tr>
<tr>
<td>5</td>
<td>0.5</td>
<td>1.66</td>
<td>6.44</td>
<td>7.26</td>
<td>11.04</td>
<td>11.54</td>
<td>0.43</td>
<td>0.22</td>
<td>0.72</td>
<td>2.79</td>
<td>3.15</td>
<td>11.62</td>
</tr>
<tr>
<td>6</td>
<td>0.5</td>
<td>1.72</td>
<td>7.58</td>
<td>8.71</td>
<td>13.07</td>
<td>13.57</td>
<td>0.44</td>
<td>0.22</td>
<td>0.76</td>
<td>3.35</td>
<td>3.85</td>
<td>13.48</td>
</tr>
<tr>
<td>7</td>
<td>0.5</td>
<td>1.76</td>
<td>8.70</td>
<td>10.18</td>
<td>15.12</td>
<td>15.62</td>
<td>0.45</td>
<td>0.22</td>
<td>0.79</td>
<td>3.90</td>
<td>4.56</td>
<td>15.51</td>
</tr>
<tr>
<td>8</td>
<td>0.5</td>
<td>1.79</td>
<td>9.80</td>
<td>11.68</td>
<td>17.18</td>
<td>17.68</td>
<td>0.45</td>
<td>0.23</td>
<td>0.81</td>
<td>4.43</td>
<td>5.29</td>
<td>17.69</td>
</tr>
<tr>
<td>9</td>
<td>0.5</td>
<td>1.81</td>
<td>10.87</td>
<td>13.20</td>
<td>19.24</td>
<td>19.74</td>
<td>0.46</td>
<td>0.23</td>
<td>0.82</td>
<td>4.96</td>
<td>6.02</td>
<td>19.97</td>
</tr>
<tr>
<td>10</td>
<td>0.5</td>
<td>1.82</td>
<td>11.91</td>
<td>14.74</td>
<td>21.30</td>
<td>21.80</td>
<td>0.46</td>
<td>0.23</td>
<td>0.84</td>
<td>5.46</td>
<td>6.76</td>
<td>21.94</td>
</tr>
</tbody>
</table>

To these queues are 1.0, 0.5, and 0.7, respectively. The first column is the number of jobs $n$ in the QN. The next four columns are the residence times at the four queues. The sixth column shows the residence time at the FJ. For example, for $n = 4$, $R^*$ is computed as $(1/1) \times 5.85 + (1/2) \times 5.29 + (1/3) \times 1.57 = 9.02$ sec. The next column is the total response time, $R$, which in this case is the sum of the residence time at the FJ ($R^*$) plus the residence time at device 1 ($R_1'$). The next column is the throughput ($= n/R$). The next four columns are the average queue lengths at each of the four queues. The column labeled $R_{sim}$ shows the results obtained by simulation and the last column shows the percent relative error between the analytic and simulation results.

### 4.2.8 Heterogeneous FJ Multi class MVA

The algorithm presented before is for networks with a single customer class. This algorithm is very fast and its computation time grows linearly with the number of jobs and the number of queues. For multiple classes, the computational complexity of MVA grows exponentially with the number of customer classes and number of devices. Therefore, approximate MVA (AMVA) techniques have been used to reduce the computational complexity [74].

We consider a closed QN with $M$ non-FJ queues numbered 1, \cdots, $M$ and a single FJ
queue with $K$ queues. The queues of the FJ construct are numbered $M + 1, \ldots, M + K$ in the QN. The notation $R^*_r(\vec{N})$ stands for the average residence time at the FJ device for a customer population of $\vec{N}$ in the QN. Similarly to the single-class case, we define $\varepsilon_{k,r}$ as the probability that class $r$ jobs visit queue $k$ of the FJ for $k = M + 1, \ldots, M + K$ and to simplify the notation we define $\varepsilon_{k,r} = 1$ for $k = 1, \ldots, M$.

The AMVA equations extended for the FJ case are given below. The AMVA throughput equation is

$$X_{0,r}(\vec{N}) = \frac{N_r}{Z_r + R^*_r(\vec{N}) + \sum_{k=1}^{M} R'_{k,r}(\vec{N})}, \quad (4.14)$$

where $Z_r$ is class $r$ customers’ think time and $R^*_r(\vec{N}) + \sum_{k=1}^{M} R'_{k,r}(\vec{N})$ is the average response time, $R_r(\vec{N})$, for class $r$ customers. The average number of class $r$ customers at queue $k$ is given by:

$$\bar{n}_{k,r}(\vec{N}) = X_{0,r}(\vec{N}) \times R'_{k,r}(\vec{N}). \quad (4.15)$$

The residence time equation becomes

$$R'_{k,r}(\vec{N}) = \varepsilon_{k,r}.s_{k,r}[1 + \sum_{t=1}^{R} \bar{n}_{k,t}(\vec{N} - \vec{1}_r)], \quad (4.16)$$

where $\vec{1}_r$ is a vector $(0, 0, \cdots, 1, 0, \cdots, 0)$ of zeros with a 1 in the $r$-th position.

The term $\bar{n}_{k,t}(\vec{N} - \vec{1}_r)$ is approximated as $(N_r - 1)/N_r \times \bar{n}_{k,t}(\vec{N})$ for $t = r$ and $\bar{n}_{k,t}(\vec{N})$ for $t \neq r$ using the Bard-Schweitzer approximation [75].

AMVA starts with the assumption that the number of customers is equally distributed between the devices in the network and stops when successive values of the queue length are sufficiently close. As with our single-class MVA approach, the residence times for the FJ queues are computed individually. In that computation, we multiply the service demands by the corresponding queue routing probability as we did for the single-class case. The
values of the residence times in the FJ are ordered descendingly to compute the overall FJ residence time for each class as

\[ R_r'(\vec{N}) = \frac{1}{k-M} R_{k,r}'(\vec{N}), \quad (4.17) \]

s.t. \( R_{M+1,r}'(\vec{N}) \geq R_{M+2,r}'(\vec{N}) \geq \cdots \geq R_{M+K,r}'(\vec{N}). \) Then, the throughput is computed using the overall FJ residence time according to equation (4.14). Finally, the queue lengths are computed by using the single FJ queues residence time and the system throughput. The approach for computing the performance metrics for the multi-class FJ using an extension to AMVA is described by algorithm given in algorithm 4 in section 4.3.

### 4.3 Approximate Mean Value Analysis for Multiclass QNs with FJ Constructs

The approach for computing the performance metrics for a multiclass QN with FJ queues described in section 4.2.2 of the chapter is given in Algorithm 4. This algorithm extends the AMVA algorithm described in [69]. The notation used in Algorithm 4 is given in Table 4.4.

One should note that the algorithm is presented for a QN with a single FJ device. It is easy to see that the algorithm described can be easily applied to situations when there are multiple FJ constructs in the QN.

### 4.4 Evaluation

To validate our approximation, we compared the analytic with simulation results. Simulations for each configuration were run 30 times, with different and independent seeds in each run, and 95% confidence intervals were computed. The simulation was built using OMNeT++ 4.2 [110]. In each configuration we simulated \( K = 4, 8, 16, 25, 50, 75, 100 \) for different values of the highest utilization of the queues in the FJ: \( \rho = .05, .1, .2, \cdots , .9, .95 \).
in the open case and for $N = 1, \cdots, 35$ in the closed case. We ran each experiment for 300 minutes with a 3-minute warm-up period. It should be noted that for larger values of $K$ and $\rho$, the simulation required significantly longer warm-up periods to attain statistical steady-state.

The service times for each configuration are given in Table 4.5 along with the queue

Table 4.4: Summary of Notations

<table>
<thead>
<tr>
<th>$M$ :</th>
<th>Number of non-FJ queues.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$ :</td>
<td>Number of FJ queues.</td>
</tr>
<tr>
<td>$R$ :</td>
<td>Number of customer classes in the QN.</td>
</tr>
<tr>
<td>$s_{i,r}$ :</td>
<td>Average service time of class $r$ customers at queue $i$.</td>
</tr>
<tr>
<td>$\epsilon_{i,r}$ :</td>
<td>Routing probability of class $r$ at queue $i$.</td>
</tr>
<tr>
<td>$N_r$ :</td>
<td>Number of class $r$ customers.</td>
</tr>
<tr>
<td>$X_{0,r}(\bar{N})$ :</td>
<td>Class $r$ customer’s throughput.</td>
</tr>
<tr>
<td>$R_{i,r}(\bar{N})$ :</td>
<td>Class $r$ customer’s residence time at queue $i$.</td>
</tr>
<tr>
<td>$R_{i}^*(\bar{N})$ :</td>
<td>Class $r$ customer’s residence time at the FJ.</td>
</tr>
<tr>
<td>$\bar{n}_{i,r}(\bar{N} - I_i)$ :</td>
<td>Average number of class $r$ customers at queue $i$ when there is one less class $t$ customer in the QN.</td>
</tr>
</tbody>
</table>

Table 4.5: Simulation Parameters

<table>
<thead>
<tr>
<th>$K$</th>
<th>$s_k$</th>
<th>$\epsilon_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>(1,.75,.5,.25)</td>
<td>(.625 $\forall k$)</td>
</tr>
<tr>
<td>8</td>
<td>(1,.9,.8,.7,.6,.5,.4,.3)</td>
<td>(0.5625 $\forall k$)</td>
</tr>
<tr>
<td>16</td>
<td>(1.95,.9,85,.8,.75,7,.65,.6,.5,5,.45,.35,.3,.25)</td>
<td>(0.53125 $\forall k$)</td>
</tr>
</tbody>
</table>

Table 4.6: Multiclass Parameters

<table>
<thead>
<tr>
<th>$r$</th>
<th>$\epsilon_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0.5625 $\forall k$)</td>
</tr>
<tr>
<td>2</td>
<td>(0.1, 0.2, 0.3, 0.4, 0.5, 0.75, 1, 0.9)</td>
</tr>
<tr>
<td>3</td>
<td>(0, 0, 0, 1, 0, 1, 0, 1)</td>
</tr>
</tbody>
</table>
Algorithm 4 Approximate MVA for Multi-class QNs with FJ Constructs

Input: $\vec{S} = (s_{i,r}, \ldots, s_{M,R})$, $M$, $K$, Tolerance
Input: $\vec{N} = (N_1, N_2, \ldots, N_R)$
Initialize: $m_r \leftarrow$ number of devices visited by class $r$ jobs

for $r = 1 \rightarrow R$ do
  for $i = 1 \rightarrow M + K$ do
    if $s_{i,r} \times \varepsilon_{i,r} > 0$ then
      $\bar{n}_{i,r}^c(\vec{N}) \leftarrow \frac{N_r}{m_r}$
    else
      $\bar{n}_{i,r}^c(\vec{N}) \leftarrow 0$
  end if
end for
repeat
  for $r = 1 \rightarrow R$ do
    for $i = 1 \rightarrow M + K$ do
      $\bar{n}_{i,r}(\vec{N}) \leftarrow \bar{n}_{i,r}^c(\vec{N})$
    end for
  end for
  for $r = 1 \rightarrow R$ do
    for $i = 1 \rightarrow M + K$ do
      $\bar{n}_{i,r}(\vec{N} - \vec{1}_t) \leftarrow \begin{cases} \bar{n}_{i,r}(\vec{N}) & t \neq r \\ \frac{N_r-1}{N_r} \bar{n}_{i,r}(\vec{N}) & t = r \end{cases}$
    end for
  end for
end for
for $r = 1 \rightarrow R$ do
  for $i = 1 \rightarrow M + K$ do
    $R'_{i,r}(\vec{N}) \leftarrow \begin{cases} s_{i,r} \varepsilon_{i,r} & \text{Delay} \\ s_{i,r} \varepsilon_{i,r} [1 + \sum_{t=1}^{R} \bar{n}_{i,t}(\vec{N} - \vec{1}_r)] & \text{LI} \end{cases}$
  end for
end for

$R'_r(\vec{N}) \leftarrow \sum_{k=M+1}^{M+K} \frac{1}{k-M} R'_{k,r}(\vec{N})$
s.t. $R'_{M+1,r}(\vec{N}) \geq R'_{M+2,r}(\vec{N}) \geq \cdots \geq R'_{M+K,r}(\vec{N})$
$R_r \leftarrow R'_r(\vec{N}) + \sum_{i=1}^{M} R'_{i,r}(\vec{N})$

$X_{0,r}(\vec{N}) \leftarrow \frac{N_r}{m_r + R_r}$

for $r = 1 \rightarrow R$ do
  for $i = 1 \rightarrow M + K$ do
    $\bar{n}_{i,r}^c(\vec{N}) \leftarrow X_{0,r}(\vec{N}) \times R'_{i,r}(\vec{N})$
  end for
end for

until $\max_{i,r} |\bar{n}_{i,r}^c(\vec{N}) - \bar{n}_{i,r}(\vec{N})| < \text{Tolerance}$

return $R_r$ for $r = 1, \cdots, R$

▷ Average Response Time
visit probabilities for each configuration used in the dynamic case. The multiclass access
probabilities are given in Table 4.6. The think time in all the closed QN experiments was
fixed at 0.5 seconds and the AMVA algorithm tolerance was fixed at 0.01. The simulation
results were compared to the approximate ones to compute the relative error. We first
check if the approximation result is within the 95% confidence interval of the simulation.
In this case, we assume that there is no error at the 95% confidence level. Otherwise,
we compute the error as the percent difference between the approximation result and the
closest bound of the 95% confidence interval. We ran experiments to verify the accuracy
of the approximation as a function of the degree of heterogeneity of the service demands.
Additionally, we ran experiments to assess the scalability of the approximation for a large
number of queues and jobs. Finally, we ran experiments to compare our approximation to
other approximations presented in the literature.

4.4.1 Results

Figure 4.3 shows the analytic results of the open model described in section 4.2.1 compared
to simulation results. The top plots show the quality of the approximation for the full
fork case ($J = K$). We can see that the approximation tracks the simulated response time
and provides adequate estimates for the response time. The relative error between the
approximate response time values and those obtained by simulation increases slightly with
high utilization. However, the relative error in the worst case scenario at $\rho = 0.95$ is less
than 22%. The results for the dynamic fork model are shown in the middle plots. One
can see that the approximation matches the simulated model and provides better results, in
general, than the full fork case with a maximum relative error of 6%, which for the most part
falls within the simulation’s 95% confidence interval. Moreover, the dynamic fork response
time is lower than that of a similar full fork FJ. This can be explained by the fact that in
the dynamic fork case the number of parallel siblings (i.e., degree of parallelism) is smaller
than that of the full fork case. Consequently, the dynamic FJ case has less synchronization
time at the joint point resulting in a smaller response time and a smaller relative error.
One notable difference between both cases (i.e., full fork and dynamic fork) is that the approximation for the full fork case always provides more pessimistic estimates while in the case of the dynamic fork the estimate is more optimistic, yet closer to the simulated response time. The multiclass case is shown in the bottom plots where jobs of classes 1 and 2 are assigned to queues according to the routing values in Table 4.6 (dynamic fork) and jobs of class 3 are routed based on the policy shown in the same table (configurable fork). The multiclass approximation provides similar accuracies to that of the single class case.

Figure 4.4 compares the response time analytic results for the closed model described in section 4.2.5 with the simulation results. All the graphs for the closed model are a function of the number of jobs in the system (i.e., the customer population). The top plots show the quality of the approximation in the full fork case. We can see that the approximation tracks the simulated results and provides accurate estimates for the response time. Similarly to

Figure 4.3: Average Response Time (in seconds) vs. Utilization for Open QNs
Figure 4.4: Average Response Time (in seconds) vs. Number of Jobs for Closed QNs

the open case, the approximation tends to be more pessimistic. Furthermore, the relative error between the approximate response time and the simulation results increases until the customer population (\(N\)) is equal to five and decreases with \(N\) after that.

The dynamic fork model results shown in the middle plots exhibit similar trends to the open case and provide more accurate results than the full fork case with a relative error of about 5%. Results for a multiclass case are shown in the bottom plots in which the branching policy is given in Table 4.6. The AMVA extension presented in section 4.2.8 provides an accurate approximation and converges at the same rate as the AMVA algorithm.

Figure 4.5 compares the throughput for a single class closed QN obtained through simulation and by the approximation. The top plots are for the full fork case and the bottom ones for the dynamic case. As it can be seen, in both cases, the error decreases as the customer population increases. The graphs also show that the error in the throughput is much smaller in the dynamic case than in the full fork one.
Figure 4.5: Throughput (Jobs per Second) vs. Number of Jobs for a Single Class Closed Model for 4, 8, and 16 Queues

Figure 4.6 shows the relative percentage error for an open FJ with two queues as a function of the service time heterogeneity defined as $(s_1 - s_2)/s_1$ for different values of the maximum utilization (30%, 45%, 60%, and 75%). We can observe that the relative percentage error initially increases with the heterogeneity, and reaches a maximum value for some heterogeneity value and then decreases as the heterogeneity increases. For open QNs, the maximum error occurs at different points for different values of the maximum utilization and tends to occur for lower heterogeneity values for higher utilization and at higher heterogeneity values for lower utilization. Moreover, the maximum error increases with the utilization. For example, for a maximum utilization of 75%, the maximum value of the relative percentage error is close to 12% and it occurs for a heterogeneity value equal to 0.2. However for a utilization of 30%, the maximum value of the relative percentage error is 8% and it occurs for a heterogeneity value of 0.7. We observed a similar behavior for the closed QN case, albeit with a smaller relative error, as shown in the figure.

Additional experiments were run to assess the scalability of the approximations. The experiments were conducted for closed and open QNs with a large number of queues and customers. For open QNs, the results show that the relative error increases for higher
values of the utilization $\rho$. Also, the relative error is relatively invariant with respect to the number $K$ of queues in the FJ. However, for $K > 80$ the FJ actual response time grows at a slower rate than that of the approximation, which in turn results in a higher error rate for $K > 80$. This is due to the fact that adding queues with relatively lower service time to the FJ has a minute impact on the synchronization time, hence, the actual FJ response time will grow slower than the approximation as a function of the number of queues $k$. In closed QNs, the error reaches a peak for a small number of jobs and then decreases as the number of jobs increases. It can also be seen that the error increases with the number of queues. However, that difference is more pronounced at lower loads. This trend is due to the fact that a higher number of jobs results in a higher FJ throughput. More specifically,
the FJ queues become more utilized and it becomes more likely that there is always a job in the queue. This in turn results in less variability in a job’s service time and therefore results in a smaller error for a large number of jobs. The detailed results are presented in section 4.4.2.

Finally, our experiments show that the approximation presented in this chapter compares favorably to approximations presented in the literature while providing solutions for more cases. In the cases where other approaches provide higher accuracy, the cost of evaluating a FJ QN model using these approaches significantly exceeds the cost of evaluating the same model using the approximations presented in this chapter. The results of our comparison are given in section 4.4.3.

4.4.2 Scalability

Table 4.7: Approximation for $K = 10, \ldots, 100$.
$R_a$ = Approximate Response Time, and $R_s$ = Simulated Response Time.

<table>
<thead>
<tr>
<th>$K$</th>
<th>$R_a$</th>
<th>$R_s$</th>
<th>Error</th>
<th>$R_a$</th>
<th>$R_s$</th>
<th>Error</th>
<th>$R_a$</th>
<th>$R_s$</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.88</td>
<td>3.62</td>
<td>6.7%</td>
<td>5.80</td>
<td>5.09</td>
<td>13.3%</td>
<td>14.31</td>
<td>10.75</td>
<td>30.3%</td>
</tr>
<tr>
<td>20</td>
<td>4.74</td>
<td>4.25</td>
<td>10.9%</td>
<td>7.07</td>
<td>5.98</td>
<td>17.2%</td>
<td>17.23</td>
<td>13.18</td>
<td>29.0%</td>
</tr>
<tr>
<td>30</td>
<td>5.24</td>
<td>4.83</td>
<td>8.3%</td>
<td>7.79</td>
<td>6.87</td>
<td>13.1%</td>
<td>18.82</td>
<td>13.99</td>
<td>34.3%</td>
</tr>
<tr>
<td>40</td>
<td>5.58</td>
<td>5.10</td>
<td>9.0%</td>
<td>8.28</td>
<td>6.99</td>
<td>17.7%</td>
<td>19.85</td>
<td>14.92</td>
<td>30.7%</td>
</tr>
<tr>
<td>50</td>
<td>5.84</td>
<td>5.36</td>
<td>8.4%</td>
<td>8.65</td>
<td>7.35</td>
<td>17.5%</td>
<td>19.96</td>
<td>15.19</td>
<td>31.3%</td>
</tr>
<tr>
<td>60</td>
<td>6.05</td>
<td>5.39</td>
<td>11.8%</td>
<td>8.94</td>
<td>7.67</td>
<td>15.9%</td>
<td>21.16</td>
<td>16.10</td>
<td>30.2%</td>
</tr>
<tr>
<td>70</td>
<td>6.22</td>
<td>5.60</td>
<td>10.8%</td>
<td>9.18</td>
<td>7.76</td>
<td>17.7%</td>
<td>21.60</td>
<td>16.46</td>
<td>29.8%</td>
</tr>
<tr>
<td>80</td>
<td>6.36</td>
<td>5.61</td>
<td>13.0%</td>
<td>9.38</td>
<td>7.78</td>
<td>20.3%</td>
<td>21.95</td>
<td>16.51</td>
<td>32.2%</td>
</tr>
<tr>
<td>90</td>
<td>6.48</td>
<td>5.64</td>
<td>14.7%</td>
<td>9.54</td>
<td>7.80</td>
<td>22.0%</td>
<td>22.24</td>
<td>16.63</td>
<td>32.9%</td>
</tr>
<tr>
<td>100</td>
<td>6.59</td>
<td>5.68</td>
<td>15.8%</td>
<td>9.68</td>
<td>7.82</td>
<td>23.2%</td>
<td>22.48</td>
<td>16.71</td>
<td>33.8%</td>
</tr>
</tbody>
</table>

To assess the scalability of the approximation, we compared it to simulation results for a FJ with $K = 1, \ldots, 100$ queues in which each queue has a mean service time of $S_k = 1 - .002k$. Table 4.7 shows that the relative error increases for higher values of utilization $\rho$. Also, the relative error is relatively invariant with respect to the number $K$ of queues in the FJ. However, for $K > 80$ the FJ actual response time grows at a slower rate.
than that of the approximation which in turn results in a higher error rate for $K > 80$. This can be explained by the fact that adding queues with relatively lower service time to the FJ will have a minute impact on the synchronization time, hence, the actual FJ response time will grow slower than the approximation as a function of the number of queues $k$. This suggests that exploring other terms for the approximation for a very large number of queues might be more practical.

To illustrate this issue we run experiments to verify the approximation accuracy for large $K$. Fig. 4.7 shows the approximation versus the actual response time obtained from the simulation. It shows the case of homogeneous FJ and heterogeneous FJ. One can see in the homogeneous case that the simulation and approximation grow at the same rate as a function of $k$. However, the heterogeneous case grows at the same rate as the approximation for smaller $K$. However, after $K > 80$ the actual response time grow substantially at slower rate than that of the approximation. The synchronization time in the heterogeneous case becomes almost negligible because the service times of the new queues that are added to the FJ are significantly faster than the slowest $n$ queues. This behavior can be better observed with Fig. 4.8. Figure 4.8 shows the increase in synchronization time defined as $(T_{k+1} - T_k)$.
with the number of queues in the FJ. One can observe that both cases approximately have close synchronization time with lower number of FJ queues $K$. Then the heterogeneous synchronization time decreases at faster rate than that of the homogeneous case and reaches almost zero with $K \approx 80$. On the other hand, the homogeneous case synchronization time decreases until $K \approx 80$ then stays constant. The results presented earlier is for the open case, however, the same trend applies to the close case. Specifically, one can see in Fig. 4.9 that the throughput for the closed case –after a certain number of jobs– is almost equal for 4, 8 and 16 queues. This indicates that the synchronization time becomes lower as the number of heterogeneous queues increases and that with relatively higher multi programming level in heterogeneous FJ the synchronization time becomes negligible. As indicated earlier this suggests that other terms for $K \geq 80$ might provide better approximation accuracies.

For closed QNs, Fig. 4.10 shows the percent relative error for the response time approximation versus the number of jobs in a single-class closed system for the full fork case with 4, \ldots, 100 queues. The error reaches a peak for a small number of jobs and then decreases as the number of jobs increases. It can also be seen that the error increases with the number
of queues. However, that difference is more pronounced at lower loads. This trend is due to
the fact that a higher number of jobs results in a higher FJ throughput. More specifically,
the FJ queues become more utilized and it becomes more likely that there is always a job in
the queue. This in turn results in less variability in a job’s time to finish from the queue and
therefore, results in a smaller error for a large number of jobs. It should be noted that single
class results are presented here but the same results have been obtained for multi-class QNs
that have been tested.

It should be noted that while the results presented in this section are for single class
models only, a number of multi class simulations were carried out. However, there was
no significant difference in terms of accuracy between single class and multi class QNs
under the same conditions, as the system scales out (i.e., the number of parallel queues
increases), which is in agreement with our analytical results. On the other hand, closed QN
approximations provide a relatively better accuracy over open QN approximations because
a job service demand variability has less impact in a closed QN compared to an open QN.
However, the choice between using one model over the other depends on the system being
modeled, available system measurements, and the workload characteristics of the system. Furthermore, as discussed in [111], which compares open versus closed QNs, one should be careful that not all systems can be modeled accurately as closed QNs.

4.4.3 Comparative Results

Table 4.8 shows that results obtained through approximations presented in this dissertation compares favourably with results obtained using approximations presented in the literature.
under the same conditions in terms of relative error defined as \( \frac{\text{model}-\text{simulation}}{\text{simulation}} \). Moreover, the approach presented here is a general expression. As such, it can be used for single/multiclass open and closed QNs with homogeneous or heterogeneous service times. More specifically, approximations presented in [86–88] provide some improvements in accuracy. However, they only approximate homogeneous FJ queues. Moreover, the approximation in [87,88] require a scaling factor obtained through simulation or traffic interpolation. The approximations presented in [89] for homogeneous FJ queues, and in [92] for heterogeneous FJ provides significant improvements, at the expense of computational complexity. The other heterogeneous FJ approximation presented in [90,91] using maximum order statistics provides similar accuracies. However, their method requires solving complex integrals to find a closed form expression.

4.5 Concluding Remarks

This chapter discussed existing analytical solutions for fork-and-join queuing networks and presented an approximation based on harmonic numbers and assessed it through simulation. The approximation presented in this chapter provides a more general expression than the ones formerly presented in prior work by others. The approximations presented here cover the cases of open, closed, single class, and multiclass queuing networks. Furthermore, the approximation works well with more general fork and join cases such as (1) heterogeneous service demands and (2) dynamic and configurable fork. The relative error of the approximation is approximately between 5-15\% for most of the evaluated cases. The approximation is very effective in dynamic re-configuration and adaptation scenarios that require the solutions for many configurations in a short time at run time. Being able to model fork-and-join queues is very important for building the predictive analytic models used by the autonomic controller of the ISQDB framework presented in this dissertation.
Chapter 5: Evaluation of the ISQDB Framework

This chapter investigates the effectiveness of the proposed framework on a database application. The first section of this chapter deals with a simulated environment. The simulated environment uses an open QN model. The second section considers an experimental environment based on the TPC-W benchmark [112] that is modeled using a multi-class closed QN. In each case, analytic performance models are presented and evaluated in experimental settings and results are reported. Some of the work presented in this chapter appears in [103] and [105].

5.1 Simulation Validation

This section first describes the simulation environment used to evaluate the controller. The results and discussion of the evaluation are presented thereafter.

5.1.1 Simulation Description

We setup an experimental environment that simulates a database system with syntax and data-centric mechanisms to assess the effectiveness of the autonomic controller. The simulation was built using OMNeT++ 4.2 [110]. The service times for the database and the security mechanisms were assumed to be exponentially distributed and the controller interval was fixed at 30 seconds. We executed 10 independent runs, of duration equal to 480 minutes each, with different and independent seeds each run. For each of the ten runs we repeated the simulation with the controller disabled and with a fixed security policy. The initialization values for each role are shown in Table 5.1. The global utility weights used in the utility function of Eq. 3.16 are $\alpha = 0.6$ and $\beta = 0.4$. The detection rate for the IDPSs are assigned based on their ability to detect certain attacks better than others. The attack
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Role 1</th>
<th>Role 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_r^l$</td>
<td>.35</td>
<td>0.65</td>
</tr>
<tr>
<td>$w_r^s$</td>
<td>0.6</td>
<td>0.4</td>
</tr>
<tr>
<td>$\sigma_r$</td>
<td>.25</td>
<td>0.20</td>
</tr>
<tr>
<td>$\rho_r$</td>
<td>(1,0,0,0,1,0)</td>
<td>(1,0,0,0,1,0)</td>
</tr>
</tbody>
</table>

Figure 5.1: Model Results for Different $\lambda$

likelihood matrix $A$ consists of three attacks for each role assigned values that represent each role security concerns regarding specific type of attacks.

5.1.2 QN Model Validation

The models used by the controller are an approximation and need to be validated. The focus of our validation is on the controller’s ability in computing the QoS value for a given configuration with sufficient precision to be useful by the controller. Thus, the model must be able to track the trends in performance reasonably well to be useful. To verify this, the results of the complete model given in Algorithm 3 were compared with those obtained
from the experiment. Figure 5.1 compares the model predictions with actual measurements with .95 confidence interval in terms of average response time for various values of requests arrival rates. The relative error for the results obtained with the analytic model were less than 10% ± 2% for all measured values. The multi-class model of the FJ provided similar accuracy as seen in table 5.2.

5.1.3 Controller Results

Figure 5.2a illustrates the variation of the arrival rate for each role during the experiment. The total arrival rate varies from about 20 req/sec to 65 req/sec. The system is driven to high utilization levels at several times, notably at minutes 150 and 400 when the arrival rate reaches 55 req/sec and 65 req/sec, respectively. The first significant change in global utility (see Fig. 5.2b) occurs at minute 15 when there is a first spike in Role 2 requests. The controller changes the security policy when the response time increases. Then, it recovers to a higher security policy when the arrival rate drops again to about 10. This behavior repeats several times during the experiment. When subject to higher workloads, the controller attempts to first swap and change assignment of the used IDPSs in a way that reduces the response time. However, when the arrival rate of requests reaches a peak,

<table>
<thead>
<tr>
<th>Total Arrival Rate (req/sec)</th>
<th>Utilization (%)</th>
<th>Error (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7.5</td>
<td>7.2</td>
</tr>
<tr>
<td>15</td>
<td>22.5</td>
<td>8.6</td>
</tr>
<tr>
<td>30</td>
<td>45.0</td>
<td>8.4</td>
</tr>
<tr>
<td>40</td>
<td>60.0</td>
<td>10.1</td>
</tr>
<tr>
<td>50</td>
<td>75.0</td>
<td>10.4</td>
</tr>
<tr>
<td>60</td>
<td>90.0</td>
<td>10.1</td>
</tr>
<tr>
<td>65</td>
<td>97.5</td>
<td>10.7</td>
</tr>
</tbody>
</table>
the controller reduces the number of IDPSs used. This is most notably seen at minutes 200 and 400 in which the controller relaxes the security policy to improve the response time until the workload starts decreasing. This behavior is better reflected in the security utility depicted in Fig. 5.2b that shows that the controller makes substantial adjustments in the security policy to address spikes in the system workload.

Figure 5.2c compares the utility of the controlled system to that of the case in which the controller is disabled and the policy in Table 5.4 is used throughout the entire experiment (static case) and with the case in which all IDPSs are disabled (no security case). As it can be seen, the controlled system exhibits a higher utility than the other two cases. Additionally, the controller enables the global utility to recover faster from workload spikes. This is best seen at minute 390 when the controller is able to recover to a higher utility, before the static security policy system, by relaxing the security policy to meet the performance goals and returning to a higher policy as soon as the workload intensity starts to subside.

The analytic model used by the system provided satisfactory estimates as shown in Fig. 5.2d, which shows that the measured utility matches very closely the predicted utility, i.e., the utility computed using response times predicted by the analytic QN model. The model is more sensitive to extreme loads in which case the relative error exceeds 5%.

The evolution of the response time during the experiment depicted in Fig. 5.2e shows that most of the response times for roles 1 and 2 are within their SLOs. A few peaks exceed these SLOs, especially for role 1. However, on average, the SLOs are met as indicated in Fig. 5.2f. This figure shows the average response time for each role with three configurations during the experiment. The average response times for the controlled case are 0.23 sec and 0.18 sec for roles 1 and 2, respectively. These values are below the SLOs of 0.25 sec and 0.20 sec specified in Table 5.4. When the controller is disabled and a static security policy is used, the SLOs for roles 1 and 2 are violated by 56% and 75%, respectively. When no IDPS is used, the average response is the lowest of all cases, as expected. This indicates that the controller performs better than the system with a static policy while still managing to achieve a proper tradeoff between performance and security.
Figure 5.2: Experiment Results: (a) Workload (top left), (b) System Utility (top right), (c) Difference in $U_g$ (middle left), (d) Model Accuracy (middle right), (e) Response Time Per Role (bottom left), and (f) Average Response Time Comparison (bottom right).
5.2 Experimental Validation

The previous validation was carried out in a simulation environment. In this section we present an experimental validation conducted using the TPC-W benchmark and IDPSs setup in a virtualized environment. The section first describes the experimental testbed. Then, we provide information about the TPC-W, the IDPSs, and the implementation details of the controller. Finally, the results are presented and discussed.

The testbed used in our experiment consists of two physical machines (see Fig. 5.3). One machine hosts three virtual machines: Web server, DB server, and IDPS servers. The other machine is used for workload generation. The former machine has 12GB of RAM and an Intel i5 processor. All VMs are equipped with 2GB of RAM and an Intel 2.6 GHz CPU. Except for the IDPS VM, all VMs run Windows 7. The IDPSs are setup on an Oracle Linux machine. The workload generator machine is a Windows machine with 4GB of RAM. The Web server uses Apache Tomcat v7.0 [113] as the servlet engine for the content. The DB server is MySQL 5.5 [114] configured to accept 250 connections. The IDPS VM contains two commercial DB IDPSs and three IDPS prototypes found in the literature (see subsection 5.2.3). All clients were connected by 100Mbps network adapters and setup on the same subnet.

5.2.1 Experimental Testbed

5.2.2 TPC-W

We implemented our controller in the TPC-W benchmark. TPC-W is a transactional Web e-Commerce benchmark that emulates the operation of an on-line bookstore. TPC-W specifies 14 unique Web interactions, each of which must be invoked according to a certain navigational pattern described by a Customer Behavior Model Graph (CMBG) [115]. Typically, a continuous period of time during which a client is using the Web site is referred to as a User Session which consists of a sequence of consecutive individual interactions. These interactions differ in the number of outbound database requests, dynamic content,
and number of embedded images. In general, these interactions can be roughly classified as browsing or as ordering type. The TPC-W defines three standard interaction mixes based on the two different types. The mixes use different weights of the ordering and browsing types as follows:

- browsing mix: 95% browsing and 5% ordering;
- shopping mix: 80% browsing and 20% ordering;
- ordering mix: 50% browsing and 50% ordering.

TPC-W defines the probabilities and transitions from one page to the other using the CMBG. During each session, the client starts with the home page and client cycles through a process of sending a transaction request, receiving the response web page and selecting the next interaction (i.e., transactions) request. Each type of Web interaction in TPC-W is modeled as a different role in the DB and in the QN model (i.e., $R = 14$).

The TPC-W benchmark is designed for web and database performance benchmarks. However, one can think of each Web page as different security risks. For example, form-based pages are susceptible to SQL injection threats while ordering and admin pages are
more susceptible to CrossSite scripting threats. As a result each role in our test represent
a certain risk to the database application.

We used a modified Java implementation of TPC-W from the University of Wisconsin [116] to make it compatible with Tomcat and MySQL. Except for the payment gateway emulator, it implements all functionalities in the TPC-W specification. The DB is configured to store 10,000 items and 144,000 customers. A servlet invokes a DB handler, which uses a driver manager and implements its own connection pooling for DB interactions. This implementation contains a DB handler that the servlet calls for DB interactions. The handler uses a driver manager and implements its own connection pooling. We used the newer JNDI data source instead of managing DB connections and requests [113].

5.2.3 IDPSs

We used two commercial DB IDPSs (GreenSQL [38] and Oracle DB Firewall [33]) and we implemented three additional IDPS prototypes based on ideas available in the literature. GreenSQL is an open source DB firewall that monitors DB activity and blocks attacks. This IDPS identifies malicious attacks by comparing every query’s structure with its signature of known attacks. Also, GreenSQL provides a learning mode that profiles the current system usage profile and detects intrusions that violate learned profiles. GreenSQL can be used in a prevention detection mode and in a detection only mode. In prevention mode, it can either block the request and close the connection, return an empty result set, or return a DB error. Oracle DB Firewall provides protection against DB attacks through SQL grammar and policy-based checks. Both IDPSs have different configurations that can be fine tuned to provide finer controls and customization. We use both systems in an intrusion prevention mode. They were set up as proxy servers that sit between the DB and the application. A proxy server receives a request, conducts the evaluation, and, if the request does not trigger an alarm it is sent to the DB. Otherwise, an empty result set is sent back to the application. This setup was implemented by listeners (i.e., proxy) that sit between the IDPSs and the DB, avoiding extensive modifications to the IDPSs.
We implemented three additional IDPSs. First, we implemented a syntax based IDPS similar to the one presented in [3,29]. In this technique, the DB query syntax is translated into a binary vector used to build profiles that correlate to the expected user behavior. When a new query is submitted to the DB, a distance between a vector that represents the specific query is computed and then compared to the profile of that request’s role. In our implementation, a simple analysis of the source code is performed to characterize each request’s transactions and build the profiles for each role requests.

The second IDPS implemented is based on [5], which proposes a data-centric approach to DB intrusion detection. That approach builds profiles for different user groups. A profile is a statistical vector of different attributes (e.g., min, max, average, and count) of the result set returned by the queries. This vector is then compared to result sets for every access to the DB. If a certain deviation exists from the learned profile, the query is triggered as malicious.

For the third IDPS, we implemented a simplified version of the technique presented in [4], which is a data-centric IDPS that takes into account more elaborate measures such as sensitivity of the data as well as the amount of data retrieved by the query to develop a misusability score. The score requires elicitation of domain experts and data owners to reflect the relative importance of the data. These scores can then be used for anomaly detection by triggering an alarm if a score exceeds the normal threshold. We use a sensitivity level in the Orders table that is based on the total amount of the order. The Customers table sensitivity is based on the YTD amount as well as the number of customers in the query. The CreditCard information sensitivity is based on the number of records returned in the result set.

5.2.4 ASC Implementation

The security controller is implemented as a single thread in the Web application server. In the TPC-W implementation, servlets invoke the DB using a dedicated database handler. This handler is modified to check for specific request policies before sending the request to
the DB. If IDPS evaluation is required, the query is sent to the specific IDPS using a server socket. As soon as the evaluation of the different techniques is completed, the request is forwarded to the DB. If the policy specifies using a data-centric approach, the results returned by the DB are sent to the specified IDPSs to be examined. After the evaluation is complete, the results are returned to the servlets and displayed in the servlet HTML code. If at any time an intrusion is detected, an empty result set is returned to the servlet.

The controller and IDPSs described earlier are implemented in Java. To manage concurrency aspects of each component we use the Executor Service available in the Java concurrency package. The Executor provides a method that can be called to run the controller after a certain delay or at a certain rate. In our case, the controller is invoked every CI. Database requests are directed towards the database handler, which then relays them to the database after the IDPSs decisions. The IDPSs proxy handles requests using multi threaded sockets. When a new request arrives, a new thread is forked to handle the request and send the response back to the controller.

The Performance Monitor in the controller is implemented with the help of the Java Execution Time Measurement Library (JETM) [117]. JETM provides a lightweight performance profiling for J2EE applications. It can be used by the servlet filters or in the actual servlet code. JETM provides basic performance measurements such as min, max, average, and number of executions. The Performance Monitor is invoked by the controller to profile monitors and provide the information to the Workload Analyzer. Namely, it provides the total number of executions per type of request as well their average execution time. In larger applications with larger number of monitors other performance monitoring APIs such as JAMon [118] that provides a greater flexibility in managing monitors data can be used.

5.2.5 Workload

The TPC-W specifications includes a workload generator. The generator is a closed loop session emulator, where each virtual user represents a browser called the emulated browser EB. The number of EBs represents the system’s workload. The interactions of a specific
EB are described by the CMBG that defines transitions and frequencies in which certain pages are visited. Each client opens a session to the front-end Web server using a persistent HTTP connection, issues a series of requests for the duration of the session, and then closes the connection. Within each session, the client makes repeated requests, parses the server’s response, waits a variable amount of time, and then follows a link embedded in the response. According to the TPC-W specifications, the EBs represent three different type of workloads: browsing, shopping, and ordering mixes. In each mix, web interactions have different access frequencies. We subject the system to a combination of the three mixes.

Due to the lack of real DB attacks, we used synthetic attack queries. We generated 27 different attacks of three categories: 13 SQL injection queries, 9 stored Cross-site scripting, and 6 queries that represent role policy violations. The TPC-W code was modified where appropriate to allow these attacks to succeed. Specifically, most of the input validations in the servlets were disabled. For example, most of the input validation is disabled in admin requests and ordering pages to enable Cross-site scripting. Prepared statements and parameterization were disabled for the search and order inquiry pages to enable SQL injections (see Fig. 5.4). The attack requests are injected randomly using a JMeter instance running on the workload generator machine. The JMeter workload is identical to the EBs workload. Specifically, we use a random think time between consecutive requests. The session navigation pattern is implemented using the MarkovSession Controller plug-in presented in [119].

5.2.6 Practical Considerations

Typical production environments have a very large number of users, which makes it impractical to keeping a normal access profile for each user. One alternative is to classify users into roles. RBAC has been proposed to build a profile for each role and check the behavior of each role with respect to such profile [3]. Alternatively, DB logs could be clustered to define roles and their expected behavior [2]. Then, attack categories to the DB are identified. A second-level classification is then formulated based on (1) the specific scenario to which an
Chapter 5: Security and Performance

TPC Web Commerce Benchmark (TPC-W)

Order Inquiry Page

Username:
Password:

Search Request Page

Click on one of our latest books to find out more!

Search by:

A catalog of attack probabilities for each attack category and for each role, as well as the detection probability for each security mechanism can be built by performing rigorous system analysis of past events and historical data related to DB application threats and attacks. Based on this catalog, initial detection rates and attack likelihood values can be estimated by leveraging historical and training data, and domain expert’s knowledge of stakeholders behaviors and concerns. It should be emphasized that although obtaining
accurate values is a challenging task, our target is rather measuring the relative ordering of different security configurations (i.e., different combinations of IDPSs) in terms of their effectiveness and associated risks. More specifically, for the controller to be effective, it needs to qualitatively measure the security level in relative terms to other configurations that are being considered.

The performance values in our approach are measured experimentally. Specifically, the service demands of different components of the QN model are obtained by running a relatively large number $RN$ of role $r$ requests for a period of $T = 15$ minutes, during which the utilization of each device $i$ is measured. Then, using the Service Demand Law, one can obtain the service demand at each device as $s_{i,r} = U_i / (RN/T)$. In this type of setting, the entire utilization of the device is attributed to role $r$ requests. In the virtual systems in our environment, the total CPU or disk utilization for each component is apportioned (i.e., $U_i = U_{total} \times U_{VM}$) accordingly to reflect its share of the actual utilization. After the utilization is known, the device service demand for role $r$ requests can be computed.

One should note that features of today’s systems present some challenges in obtaining accurate system measurements and should be carefully considered. For example, most current systems have multiple cores and use voltage scaling and hyper-threading to save energy and improve performance. These characteristics should be considered when obtaining system utilization from typical performance monitors. Specifically, the utilization should be scaled (i.e., apportioned) appropriately in such cases. For example, a Windows performance monitor may show a 100% CPU utilization (see Fig. 5.5a). However, close examination may reveal that the CPU was only using 70% of its maximum frequency. Similar observations can be made about different performance monitor measurements. In these situations, tools that provide accurate measures must be considered. For example, in our Windows environment, we use the Windows Performance Toolkit [120] that monitors the kernel traces, thus providing higher accuracy and finer details on the system resources including specific CPU cycles per process, disk I/Os, and CPU voltages (see an example in Fig. 5.5b). For accurate CPU measurements, we use the Intel Performance Counter Monitor [121]. In Linux
machines we use `iostat` to obtain I/O and CPU measurements. After obtaining accurate utilization measurements, the model can be built and parameterized by methods described in [69].

![Performance Monitors: Windows Showing Total and Per core CPU Utilization and Maximum Frequency (left) and Windows Performance Toolkit (right)](image)

**Figure 5.5:** Performance Monitors: Windows Showing Total and Per core CPU Utilization and Maximum Frequency (left) and Windows Performance Toolkit (right)

### 5.2.7 Model Validation

![Model Maximum Percent Relative Errors vs. Number of Concurrent Users](image)

**Figure 5.6:** Model Maximum Percent Relative Errors vs. Number of Concurrent Users

To verify how well the performance model described in the previous section matches the
actual system, we compared performance model predictions with measurements obtained in the experimental setting described in the next section. The maximum absolute relative error $E$ is defined as

$$E = \max_r \{\text{abs}[(T^\text{model}_r - T^\text{measured}_r)/T^\text{measured}_r]\}. \quad (5.1)$$

Figure 5.6 shows the variation of the error $E$ for both response time and throughput as a function of the number of concurrent customers. The maximum relative error is less than 5% for most measured values. The error increases at very light and very high loads. Most significantly, at high loads, $E$ reaches 22% for response time and 14% for the throughput. A high service time variability between different types of request and the hidden contention between them across multiple devices results in higher error rates. A similar observation was made in [122] where they refer to this as bottleneck switching.

To further illustrate this point, we compare the predicted response time utility with the one computed from actual system measurements obtained by the Performance Monitor under a high workload ($\sum_{r} N_r = 100$) and a lighter workload ($\sum_{r} N_r = 70$). As it can be

![Graph showing predicted vs. measured response time utility for light and heavy workloads.](image-url)
seen in Fig. 5.7, the model tracks the actual response time utility fairly well with relatively low errors. Under a higher workload intensity, the model is more sensitive to service time variabilities; yet it is still able to track the trends in performance reasonably well to be useful to the autonomic controller.

One should note that one of the salient features of QN modeling is its ability to capture intrinsic system complexity fairly well. Therefore, if higher accuracy is deemed necessary one can model more system components such as memory, network, and hardware and software contention. Also, one might use techniques such as the one presented in [123] for systems with high variability.

5.2.8 Experiment Description

<table>
<thead>
<tr>
<th>Role</th>
<th>$w_r^s$</th>
<th>$w_r^f$</th>
<th>$\delta_r$</th>
<th>$\sigma_s$</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.075</td>
<td>0.025</td>
<td>3</td>
<td>1.85</td>
<td>(0.5, 0.4, 0.1)</td>
</tr>
<tr>
<td>2</td>
<td>0.075</td>
<td>0.025</td>
<td>3</td>
<td>2.1</td>
<td>(0.5, 0.25, 0.25)</td>
</tr>
<tr>
<td>3</td>
<td>0.05</td>
<td>0.1</td>
<td>3</td>
<td>0.825</td>
<td>(0.5, 0.25, 0.25)</td>
</tr>
<tr>
<td>4</td>
<td>0.05</td>
<td>0.075</td>
<td>3</td>
<td>2.6</td>
<td>(0.5, 0.25, 0.25)</td>
</tr>
<tr>
<td>5</td>
<td>0.1</td>
<td>0.025</td>
<td>3</td>
<td>0.65</td>
<td>(0.5, 0.25, 0.25)</td>
</tr>
<tr>
<td>6</td>
<td>0.1</td>
<td>0.05</td>
<td>3</td>
<td>0.5</td>
<td>(0.5, 0.25, 0.25)</td>
</tr>
<tr>
<td>7</td>
<td>0.05</td>
<td>0.15</td>
<td>3</td>
<td>0.5</td>
<td>(0.5, 0.25, 0.25)</td>
</tr>
<tr>
<td>8</td>
<td>0.05</td>
<td>0.1</td>
<td>3</td>
<td>0.77</td>
<td>(0.5, 0.25, 0.25)</td>
</tr>
<tr>
<td>9</td>
<td>0.1</td>
<td>0.025</td>
<td>3</td>
<td>0.65</td>
<td>(0.5, 0.25, 0.25)</td>
</tr>
<tr>
<td>10</td>
<td>0.075</td>
<td>0.05</td>
<td>3</td>
<td>0.45</td>
<td>(0.6, 0.2, 0.2)</td>
</tr>
<tr>
<td>11</td>
<td>0.075</td>
<td>0.15</td>
<td>3</td>
<td>1.58</td>
<td>(0.6, 0.2, 0.2)</td>
</tr>
<tr>
<td>12</td>
<td>0.05</td>
<td>0.05</td>
<td>3</td>
<td>0.77</td>
<td>(0.6, 0.2, 0.2)</td>
</tr>
<tr>
<td>13</td>
<td>0.1</td>
<td>0.1</td>
<td>3</td>
<td>1.9</td>
<td>(0.3, 0.35, 0.35)</td>
</tr>
<tr>
<td>14</td>
<td>0.05</td>
<td>0.075</td>
<td>3</td>
<td>0.9</td>
<td>(0.3, 0.35, 0.35)</td>
</tr>
</tbody>
</table>

We executed 10 independent runs, of duration equal to 75 minutes each, with different
and independent seeds for the workload generator in each run with the initialization values shown in table 5.3. For each of the ten runs, we repeated the experiment with the controller disabled and with a fixed security policy that has all IDPSs enabled (Full Policy), and with the controller disabled and a static security policy that is pre-configured with some IDPSs enabled and others disabled (Pre-configured Policy). The pre-configured policy was selected after analyzing the likelihood matrix in such a way to maximize the overall detection rate of the system.

5.2.9 Results

Figure 5.8a illustrates the variation of the workload in requests per second during the experiment. The number of requests per second varies between about 50 req/sec at minute 20 to about 140 req/sec at minute 130. This is reflected in the response time for requests to the Home page and Search shown in Fig. 5.8b, where the response time exceeds 1 and 3 seconds respectively (the response time SLO) at high workload intensity values and then decreases when the workload subsides. The controller, in response to the increase in response time responds by changing the security policy to meet the response time objectives. When the response time recovers, the security policy reverts to a stronger security policy. Figure 5.8c depicts the variation of the controlled response time utility, the security utility, and the global utility over time during the experiment. The figure shows that the controller makes slight modifications to the security policy and keeps the security policy around 0.8 most of the time. However, in response to the violation of response time objectives, the security utility decreases as a result of relaxing the security policy to improve the response time. When the response time decrease back to its SLO, the higher security policy is applied again. This pattern occurs several times during the experiment.

Figure 5.8d compares the global utility of the controlled system to that in which the controller is disabled (Full Policy and Pre-Configured policies are shown in this case) with 95% confidence intervals at some points. It can be seen that the controller exhibits a higher utility than the uncontrolled case. This can be explained by the fact that the response time
Figure 5.8: Experiment Results: (a) Workload, (b) AC Response Time for Home and Search Page Requests, (c) Controlled System Utility Values, (d) Difference in Global Utility $U_g$, (e) Uncontrolled vs. Controlled System Response Time, and (f) Policy Changes Over Time.

of the uncontrolled case always violates its response time objectives while the controlled system meets its objective on average as it can be seen in Fig. 5.8e for Home page requests. This figure shows the average response time of the uncontrolled case (Full Security policy
and Pre-configured policy) normalized by the response time of the controlled case. We can see that the Full Security policy exhibits a response time significantly higher than the controlled case. The pre-configured case is closer to the controlled case but it is still higher.

The policy modifications made during the controlled case can be seen in Fig. 5.8f. This figure shows a distance value between consecutive policies defined as \(1 - [n(\rho_{1,1})/(n(\rho_{1,1}) + n(\rho_{0,1}) + n(\rho_{1,0}))] = \Delta\), where \(n(\rho_{1,1})\) is the count of bits where the old policy and the new policy are one, \(n(\rho_{0,1})\) is the number of bits where the old policy is zero and the new policy is one, and \(n(\rho_{1,0})\) is the number of bits where the old policy is one and the new policy is zero. One can see that the controller changes the policy frequently while at the same time keeping higher security utility value. In fact, this is the reason behind the improvements in the response time in the controlled case. This change results in distributing different role requests among different IDPSs in such a way that each IDPSs is not a bottleneck of the system.

![Figure 5.9: General System Response Time: (a) Evolution in Log Scale and (b) Average Response Time Per Case](image)

In fig.5.9a the overall response time evolution is shown for the three cases. One can see that the full policy case is clearly higher than that of the controlled case. The response time of the pre configured case is closer to the controlled case response time for most of the time. However, detailed examination reveals that at the 95/% confidence the average the
response time of the controlled case is lower than that of the pre configured case as can be seen in fig 5.9b.

![Figure 5.10: Requests response Time in Log Scale](image)

Table 5.4 shows the detection rate (i.e., recall), false positive rate, precision, and the F-score (defined as $2 \times \text{precision} \times \text{recall}/(\text{precision}+\text{recall})$) for the controller, the preconfigured case, and the Full Policy. The F-score shows the tradeoffs between the detection and false positive rates. The table shows that the uncontrolled system with Full Policy detects more attacks but has a higher false positive rate than the controlled system. However, the controlled case provides a comparable F-score to the Full Policy case illustrating that the controller provides a high overall detection quality when the detection and false detection rates are considered.

<table>
<thead>
<tr>
<th>Attack Type</th>
<th>Controller</th>
<th>Pre-configured</th>
<th>Full Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_r$ (Recall)</td>
<td>0.75</td>
<td>0.53</td>
<td>0.83</td>
</tr>
<tr>
<td>$f_r$</td>
<td>0.03</td>
<td>0.01</td>
<td>0.05</td>
</tr>
<tr>
<td>Precision</td>
<td>0.87</td>
<td>0.89</td>
<td>0.78</td>
</tr>
<tr>
<td>F-score</td>
<td>0.81</td>
<td>0.67</td>
<td>0.81</td>
</tr>
</tbody>
</table>
In some situations, the system might be subjected to high workloads as a part of normal conditions (e.g., unexpected high demands due to new product) or due to malicious activities (e.g., Denial of Service attacks). One concern in this scenario would be that this kind of workloads may result in a security policy with no security mechanisms. In order to examine this further we run an experiment that attempts to subject the system to a very relatively higher workloads. The workload is shown in Fig. 5.11a, the number of requests start at about 12 requests per seconds at CI=2 and reaches 280 requests per seconds at CI 150. The response time evolution is shown in Fig. 5.11b one can see that the response time grows
with the workload as expected. The uncontrolled case shows the highest response time compared to the two other cases. However, the other two cases response time grows with the workload and both are relatively close and have more variability in the response time. However, the controlled case response time is on average less than that of the pre-configured case as shown in Fig. 5.12.

![Graph showing controlled system response time](image)

Figure 5.13: Controlled System Response Time $U(T)$ Under High Loads

In Fig. 5.13 the response time utility for the full policy case, the pre-configured case and the controlled is shown. One can see that at CI=1 the full policy utility decreases quickly and collapses at CI=16 with response time utility about 0.05 and it doesn’t recover during the experiment. On the other hand, the pre-configured case and the controlled case utilities manage to stay about 0.60 with controlled case being slightly higher than that of the pre-configured case. At CI=45 the controlled case drops to 0.20 however it recovers to a higher utility by CI=50. One notable observation in this case is that until this point the pre-configured case seems to provide similar effectiveness to the controlled case. However, after CI=75 the pre-configured case drops at CI=75 and then at CI=90 and the controlled case provides higher utility on average during the experiment. This can be explained by the
fact that at lighter loads the controlled case tries to apply higher security, thus resulting in a slight decrease in response time. However, soon after the workload reaches higher levels the controller changes the policy to meet the response times SLO.

![Figure 5.14: Controlled System Utility Under High Loads](image)

One noteworthy observation in this experiment is the fact that after the system exceeds its SLO the security utility does not drop beyond 0.45 as seen in Fig. 5.14. This can be explained by the fact that the controller response time at this high loads will not improve significantly even if more IDPSs are removed from the policy, thus no significant improvement will be seen in the global utility. In fact, the global utility might even decrease due to the lower security policy. Therefore, it is more advantageous for the global utility to use more security and improve the security utility rather than attempting insignificant improvements to the response time utility. This observation suggests that the controller in high workloads will react in a predictable manner. However, in order to maintain a lower bound on the system security level, one can add constraints on $\rho^T$ to guarantee a minimum security level regardless of the workload intensity and QoS goals.
5.3 Concluding Remarks

This chapter presents a validation of the ISQBD. In the course of our validation we presented results for a simulated and experimental environment. The results showed the viability of our framework and illustrated how the controller outperforms static security policies by routing requests and allocating resources in a manner that improves the security while minimizing the response time. Specifically, we designed and implemented an autonomic controller that searches for a near-optimal security policy that improves the QoS and security of the system according to a global utility function. The controller uses analytic performance models to estimate the performance impact of security policies and makes assignment decisions dynamically. The experiment showed the viability of our framework and illustrated how the controller outperforms static security policies by routing requests and allocating resources in a manner that improves the security while minimizing the response time. Furthermore, the experiments showed that this can be accomplished with small or no impact on the overall security of the system.
Chapter 6: Conclusion and Future Work

6.1 Conclusion

This dissertation demonstrates the need to jointly address QoS and security requirements in databases and presents a framework, based on autonomic computing, to orchestrate between these often conflicting requirements. We design an autonomic controller that searches for a near-optimal security policy that improves the QoS and security of the system according to a global utility function. The controller uses analytic performance models to estimate the performance impact of security policies and makes assignment decisions dynamically. We present an efficient approximation for fork and join constructs in analytic models. The approximation can be used to estimate performance values in parallel constructs in QNs. The approximation is validated using a simulated environment for different open and closed queuing networks configurations. The experiments show the viability of our ISQDB framework and illustrates how the controller outperforms static security policies by routing requests and allocating resources in a manner that improves the security while minimizing the response time.

Specifically, we started by giving a brief background and related work in chapters one and two. In chapter three we presented the ISQDB framework. The framework was illustrated, utility functions were presented and the controller architecture explained. The heuristic search algorithm and QN models were presented in this chapter also. Chapter four presented a fork and join approximation with detailed description and multiple extensions. The chapter further validates the approximation with the results of the extensive evaluation that was carried out as part of this work. Chapter five presents the validation results of the ISQDB framework in two environments. It starts with a simulation environment validation and provide the results of the simulation evaluation thereafter. The experimental validation
is then presented with detailed description of the environment as well as discussion about implementation details as well as information about the ISDPSs and workload generation. Finally, the results are presented and discussed.

6.2 Future Work

Future work on this topic can be carried out in several important directions. For example, our current controller can be improved in a number of ways. It could be extended to use workload forecasting techniques and dynamic controller intervals to respond to workload variations more effectively. One can also explore the effectiveness of other search techniques besides hill-climbing. The policy neighbors definition can be improved and other search techniques besides hill climbing might provide a better performance. In environments where resources can be provisioned dynamically, the performance model can use self adaptation to update its values. In these environments the controller can be extended to make use of detailed performance profiling APIs to automatically sense resource changes and recompute the performance model values if necessary. Moreover, the current approach does not consider specific IDPSs parameters. The controller can be extended to include IDPSs parameters that impact security and performance. Furthermore, it is worth exploring applying the approach presented here for network IDPSs.
Bibliography
Bibliography


107


[94] F. Baccelli, “Two parallel queues created by arrivals with two demands: The m/g/2 symmetrical case,” 1985.


Firas B. Alomari is a security professional with over ten years of experience in information security and risk management. He received his Bachelor of Science in computer science from Yarmouk University, Jordan in 2001. He earned his Master of Science in information system from George Mason University in 2006. He is currently an application security project manager with the Internal Revenue Service. Prior to that, he held different information security and auditing roles at the Department of Defense, Department of Commerce, and Department of Treasury.