ARCHITECTURE-BASED RELIABILITY ANALYSIS OF WEB SERVICES

Cobra Mariam Rahmani

University of Nebraska at Omaha

Follow this and additional works at: https://digitalcommons.unomaha.edu/studentwork

Part of the Computer Sciences Commons

Please take our feedback survey at: https://unomaha.az1.qualtrics.com/jfe/form/SV_8cchtFmpDyGfBLE

Recommended Citation

https://digitalcommons.unomaha.edu/studentwork/2878

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.
ARCHITECTURE-BASED RELIABILITY
ANALYSIS OF WEB SERVICES

By

COBRA (MARIAM) RAHMANI

A DISSERTATION

Presented to the Faculty of
The Graduate College at the University of Nebraska
In Partial Fulfillment of Requirements
For the Degree of Doctor of Philosophy

Major: Information Technology

Under the Supervision of
Dr. Azad Azadmanesh and Dr. Harvey Siy

Omaha, Nebraska
April, 2012

Supervisory Committee:
Dr. Azad Azadmanesh
Dr. Harvey Siy
Dr. Robin Gandhi
Dr. Andrew W. Swift
Architecture-based Reliability Analysis of Web Services

Cobra (Mariam) Rahmani, Ph.D in IT
University of Nebraska, 2012
Advisors: Azad Azadmanesh & Harvey Siy

In a Service Oriented Architecture (SOA), the hierarchical complexity of Web Services (WS) and their interactions with the underlying Application Server (AS) create new challenges in providing a realistic estimate of WS performance and reliability. The current approaches often treat the entire WS environment as a black-box. Thus, the sensitivity of the overall reliability and performance to the behavior of the underlying WS architectures and AS components are not well-understood. In other words, the current research on the architecture-based analysis of WSs is limited.

This dissertation presents a novel methodology for modeling the reliability and performance of web services. WSs are treated as atomic entities but the AS is broken down into layers. More specifically, interactions of WSs with the underlying layers of an AS are investigated. One important feature of the research is investigating the impact of dynamic parameters that exist at the layers, such as configuration parameters. These parameters may have negative impact on WSs performance if they are not configured properly. WSs are developed in house and the AS considered is JBoss AS. An experimental environment is setup so that controlled service requests can be generated and important performance metrics can be recorded under various configurations of the AS. On the other hand, a simulation model is developed from the source code and run-time behavior of the existing WS and AS implementations. The model mimics the
logical behavior of the WSs based on their communication with the AS layers. The simulation results are compared to the experimental results to ensure the correctness of the model. The architecture of the simulation model, which is based on Stochastic Petri Nets (SPN), is modularized in accordance to the layers and their interactions. As the web services are often executed in a complex and distributed environment, the modularized approach enables a user or a designer to observe and investigate the performance of the entire system under various conditions. In contrast, most approaches to WSs analyses are monolithic in that the entire system is treated as a closed box.

The results show that 1) the simulation model can be a viable tool for measuring the performance and reliability of WSs under different loads and conditions that may be of great interest to WS designers and the professionals involved; 2) Configuration parameters have big impacts on the overall performance; 3) The simulation model can be tuned to account for various speeds in terms of communication, hardware, and software; 4) As the simulation model is modularized, it may be used as a foundation for aggregating the modules (layers), nullifying modules, or the model can be enhanced to include other aspects of the WS architecture such as network characteristics and the hardware/operating system on which the AS and WSs execute; and 5) The simulation model is beneficial to predict the performance of web services for those cases that are difficult to replicate in a field study.

**Keywords**

To Hashem and Niki
ACKNOWLEDGEMENTS

I would like to gratefully and sincerely thank my advisors, Dr. Azad Azadmanesh and Dr. Harvey Siy. Under their guidance, patience, and help, I have grown and evolved professionally. I am thankful for their mentorship, support, and direction in the last five years. The free and lively discussions in easy moments and encouraging advice in difficult times has made this journey so much smoother and enlightened.

My special thanks to other dissertation committee members - Dr. Robin Gandhi and Dr. Andrew Swift - for their advice and critique of the dissertation. In addition, I would like to thank Dr. William Mahoney and the Nebraska University Center for Information Assurance (NUCIA) for sponsoring my research.

I also want to thank Samir Chekkal, the author of the Javashot product, for the extension he made to Javashot to make it usable in our research.

I need to extend my sincere thanks to Dr. Roshanak Roshandel from Seattle University, Dr. Lotfollah Najjar, Dr. Mansour Zand, Dr. Massoum Mousavi, and all staff members of the College of Information Science and Technology for providing me with continuous support during the last five years.

My friends – Zahra Abbasi, Zahra Karimi Dehkordi, Fahimeh Rezaei, and Narges Tahmasbi, I thank you for your friendship, and help during all these years.

To my spouse and best friend Hashem Abedini, I am forever grateful for your help and support. And to my beloved daughter, Niki, you are the joy of my life every day. Thank you for cuddles and giggles, and smiles and kisses. Thanks for letting maman do her work! Finally, to my family for your encouragement and support – my parents, sisters and brother – Thank you so much!
# Table of Contents

1 Introduction ............................................................................................................. 1
   1.1 Introduction .................................................................................................... 2
1.2 Web Services ..................................................................................................... 3
   1.2.1 Standard Protocols and Formats for Web Services ................................. 4
   1.2.2 Service Oriented Architecture ................................................................. 6
1.3 Reliability Analysis .......................................................................................... 11
   1.3.1 Architecture-based Reliability Analysis ................................................... 12
   1.3.2 Web Service Reliability Analysis ............................................................... 15
1.4 Research Objectives ......................................................................................... 18
1.5 Dissertation Organization ................................................................................ 22

2 Literature Review and Background ..................................................................... 23
   2.1 Introduction .................................................................................................... 24
   2.2 Challenges of Web Service Reliability .......................................................... 24
   2.3 Simulation-based Reliability Research ........................................................... 27
   2.4 Experimental-based Reliability Research ....................................................... 30
   2.5 Petri Nets ....................................................................................................... 35
   2.6 Summary ......................................................................................................... 43

3 The Research Methodology .................................................................................. 45
   3.1 Introduction .................................................................................................... 46
   3.2 Static Analysis ................................................................................................ 46
   3.3 Dynamic Analysis ......................................................................................... 48
   3.4 Form Architecture Model ............................................................................... 49
   3.5 Parameter Estimation .................................................................................... 51
   3.6 Form the Petri net Model .............................................................................. 56
   3.7 Gather and Analyze Results .......................................................................... 57
   3.8 Summary ......................................................................................................... 57

4 Experimental and Simulation Results ................................................................... 59
   4.1 Introduction .................................................................................................... 60
   4.2 The Black-Box Petri net model ...................................................................... 63
      4.2.1 The Experimental Setup and Theoretical Model .................................... 63
      4.2.2 The Simulation Model ........................................................................... 70
   4.3 Architectural Analysis of the Web Service Environment ............................... 76
      4.3.1 Petri Net Modeling ................................................................................. 82
   4.4 Summary ......................................................................................................... 106

5 Summary and Conclusion .................................................................................... 108
   5.1 Summary ......................................................................................................... 109
   5.2 Limitations and Future Work ......................................................................... 110
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>an example of a bookstore web service</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>an example message of SOAP protocol</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>an early incarnation of SOA</td>
<td>7</td>
</tr>
<tr>
<td>1.4</td>
<td>an example of a web service invocation</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>main components of Petri nets</td>
<td>36</td>
</tr>
<tr>
<td>2.2</td>
<td>a) before the transition fires, b) after the transition is fired</td>
<td>37</td>
</tr>
<tr>
<td>2.3</td>
<td>a producer/consumer Petri net</td>
<td>38</td>
</tr>
<tr>
<td>2.4</td>
<td>input and output gates in SAN</td>
<td>40</td>
</tr>
<tr>
<td>2.5</td>
<td>a SAN model for TMR with repair</td>
<td>42</td>
</tr>
<tr>
<td>3.1</td>
<td>the proposed approach of architecture-based software reliability</td>
<td>47</td>
</tr>
<tr>
<td>3.2</td>
<td>architecture of JBoss AS, showing the interaction between layers and config.</td>
<td>50</td>
</tr>
<tr>
<td>3.3</td>
<td>an example Java program and its partial Javashot output file in Dot format</td>
<td>53</td>
</tr>
<tr>
<td>3.4</td>
<td>graph representation of Javashot output</td>
<td>54</td>
</tr>
<tr>
<td>3.5</td>
<td>the call graph after computing percentage of transitions between nodes</td>
<td>54</td>
</tr>
<tr>
<td>3.6</td>
<td>an overview of JRAT interface</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>Duke’s Bank Application Architecture</td>
<td>61</td>
</tr>
<tr>
<td>4.2</td>
<td>Duke’s Bank web service Interfaces and request example</td>
<td>62</td>
</tr>
<tr>
<td>4.3</td>
<td>the experimental setup</td>
<td>63</td>
</tr>
<tr>
<td>4.4</td>
<td>graphical interface of SoapUI client</td>
<td>64</td>
</tr>
<tr>
<td>4.5</td>
<td>average response time of the web service based on number of virtual users</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>with default setting in JBOSS AS (MaxThread: 250 and AcceptCount: 100)</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>request rejection rate of the application server based on virtual users</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>with default setting in JBOSS AS (MaxThread: 250 and AcceptCount: 100)</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>the SAN model of JBoss AS serving the requests</td>
<td>72</td>
</tr>
<tr>
<td>4.8</td>
<td>rejected HTTP request rate from experimental (SoapUI), theoretical and SAN</td>
<td>76</td>
</tr>
<tr>
<td>4.9</td>
<td>output files from two different instrumentation tools used in the study</td>
<td>77</td>
</tr>
<tr>
<td>4.10</td>
<td>dynamic call graphs extracted from Javashot profiler</td>
<td>79</td>
</tr>
<tr>
<td>4.11</td>
<td>the SAN model for requesting HTTP threads in the application server</td>
<td>86</td>
</tr>
<tr>
<td>4.12</td>
<td>the SAN mode showing the interaction between Apache, JBoss and Web service</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>layers</td>
<td></td>
</tr>
<tr>
<td>4.13</td>
<td>the SAN model for interaction between JBoss and Database layers</td>
<td>89</td>
</tr>
</tbody>
</table>
Figure 4.14 the SAN sub-model for returning the response to Apache .........................................................91
Figure 4.15 the hierarchical model of the overall web service system .................................................................92
Figure 4.16 actual and SAN based rejection rate of Duke’s bank web service (based on JBoss As default setting in Table 4.5) ..........................................................................................................................................................93
Figure 4.17 Http rejection rate in SAN and Actual experiments (based on JBoss As default setting in Table 4.5) …94
Figure 4.18 Http rejection rate in the improved SAN model and actual experiments (SAN parameters set using the values in Table 4.5) ..........................................................................................................................................................98
Figure 4.19 DB rejection rate based on the maximum DB connection instances (using the SAN model and parameters set in Table 4.5) ..........................................................................................................................................................99
Figure 4.20 web service error rate based on the number of clients (using the SAN model) when the web service failure probability is set at 0.01 ................................................................................................................................................100
Figure 4.21 JBoss AS error rate and HTTP rejection rate based on the number of users (using the SAN model) …102
Figure 4.22 database rejection rate based on different database service rate .........................................................103
Figure 4.23 reliability graph of the system under different loads (SoapUI) ............................................................104
Figure 4.24 estimated reliability graph of the system using SAN model simulation .............................................105
Figure 4.25 comparisons of reliabilities between the SAN model and the experimental approach for three different loads ..........................................................................................................................................................106
List of Tables

Table 2.1 a summary of relevant research studies (S: Simulation, E: Experimentation, WB: White-box, BB: Black-box, SAN: Stochastic Activity Network, MC: Markov Chain, R: Reliability, P: Performance) ............................................. 33
Table 4.1 primary SAN Model parameters .................................................................................................................. 73
Table 4.2 sample data extracted from SoapUI client ..................................................................................................... 74
Table 4.3 request rejection rate of the sixteen tests for the three models: SoapUI, theoretical, nd the SAN model ...... 75
Table 4.4 the timing information extracted from JRAT (sequence extracted from Javashot) ........................................ 81
Table 4.5 parameters set for Petri net model .................................................................................................................. 84
Table 4.6 test results from SoapUI and SAN models ..................................................................................................... 95
Table 4.7 test results from SoapUI and SAN models after adjustment in the average response time ...................... 97
Table 4.8 JBoss case probabilities for the three activities TJBoss0, TJBoss1, and TJBoss2 in the submodels of Figures 4.12 and 4.13 ........................................................................................................... 101
CHAPTER 1

INTRODUCTION
1.1 **INTRODUCTION**

Internet globalization has provided the unprecedented opportunity for enterprises to develop and deliver electronic services, which are becoming a promising technology for building distributed and complex systems. Web services as one of these technologies have permeated our lives due to their ease of use, flexibility and reduction of cost [Bak98 and Huh05]. These services, e.g. electronic shopping, online auction, and banking services, are the distributed computing technologies that follow the client/server paradigm [Glo]. Figure 1.1 shows a simple example.

![Diagram of a bookstore web service](image)

**Figure 1.1** an example of a bookstore web service

In this hypothetical web service in Figure 1.1, the client (user) asks for the ISBN of a particular book by providing the author’s name and the book’s title. If the web service client, which can be a web-based application, wants to access the book’s ISBN, it contacts the web service deployed on the server side, and sends a service request asking for its ISBN. The server would return the ISBN through a service response.
Due to its distributed nature and because an increasing number of important services are being offered in this fashion, understanding the reliability of web services has gained momentum recently. In particular, as reliability of web services highly depends on the underlying architecture on which services are run and how the services are developed, a better understanding of the relationship between the web service reliability and its architecture induces higher quality software products. In this dissertation, a methodology for constructing an architecture-based reliability model will be presented.

The rest of this chapter is organized into the following. Subsection 1.2 provides an introduction to the general paradigm of web services. Specifically, the subsection gives information about the general paradigm of web services that include a discussion of their architecture, known as Service Oriented Architecture, and common web standards. Subsection 1.3 is devoted to the reliability aspect of web related and in general software systems. It discusses the approaches to reliability and performance analyses. The discussion provides the foundation to better understand the research contributions and the methodology in achieving the objectives. Subsection 1.4 lays the objectives of this research study. Finally, subsection 1.5 gives the organization of chapters for the rest of the dissertation.

1.2 WEB SERVICES

As indicated, web services often follow the client/server paradigm. The software on the server side on which a web service is deployed is often referred to as middleware and is a key component in Service Oriented Architectures (SOA) [Wso]. SOA is the overarching
architecture for building and delivering web services, where each service follows a well-defined standard. The architecture is a set of principles and methodologies that allows for interoperability, discovery, and reusability of services. On the other hand, web services are realization of SOA. Simply they are applications over the Internet, which provide services to customers.

Web services can be combined into more complex services and be used across wide spectrums of enterprises. For this to happen, web services must be platform-neutral and follow a standardized way of implementation. According to World Wide Web Consortium (W3C), a web service is a software system designed to support interoperable machine-to-machine interaction over a network [Web]. This means that web services must follow a standard communication mechanism that use Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), Universal Description, Discovery and Integration (UDDI), and Web Services Description Language (WSDL) [Wsd]. XML provides for encoding the messages in various forms of data structures. SOAP is the protocol for transferring data. UDDI is a directory for discovery of and storing information about web services. WSDL is the interface which defines the service. This is the application interface not the user interface. The following subsection describes the main protocols/languages involved in providing a web service.

1.2.1 STANDARD PROTOCOLS AND FORMATS FOR WEB SERVICES

HTTP- the Hypertext Transfer Protocol (HTTP) is a protocol for collaborative and distributed software systems. This is the base protocol of data communication in the web.
HTTP is a protocol for sending requests and receiving responses in the client/server model. In HTTP protocol, a client submits a HTTP request to a server. Then the server, which provides a service, returns a response message to the client. A response may contain completion status information about the request and also may contain any information regarding the errors in request message [Htt].

**XML**- Extensible Markup Language (XML) is a language that provides a set of rules for encoding human-readable as well as machine-readable documents. It is a text-based data format and although its design focuses on documents, it is widely used for the representation and serialization of various data structures, for example in web services [Xml]. XML protocols are messaging protocols where messages are encoded in XML, with particular envelope formats and serialization strategies.

**SOAP**- Simple Object Access Protocol (SOAP) is an XML protocol for transmitting structured information in the implementation of web services. It relies on XML for its message format, and usually relies on other protocols such as HTTP for message negotiation and transmission. SOAP provides a basic messaging framework upon which web services can be built [Soa]. An example of SOAP message is shown in Figure 1.2.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap=http://www.w3.org/2003/05/soap-envelope>
  <soap:Header/>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Figure 1.2 an example message of SOAP protocol
**UDDI**- Universal Description, Discovery and Integration (UDDI) is a platform-independent, XML-based registry for businesses worldwide to register and locate their web services. UDDI is an open industry initiative, sponsored by the Organization for the Advancement of Structured Information Standards (OASIS), enabling businesses to publish service listings and discover each other and define how the services or software applications interact over the Internet [Udd].

**WSDL**- the Web services Description Language (WSDL) is an XML-based language to define the functionality offered by a web service. A WSDL description of a web service (also referred to as a WSDL file) defines how a web service can be called, what parameters it expects and what output it returns [Wsd].

1.2.2 **SERVICE ORIENTED ARCHITECTURE**

Thomas Erl [Erl05] has defined Service Oriented Architecture as a form of technology architecture that adheres to the principle of service-orientation. SOA is built with a set of web services designed to collectively automate one or more business processes [Erl05]. Figure 1.3 shows an early model that is mostly inspired by the initial set of web service standards. The model defined SOA as an architectural pattern built around three basic components: the service requestor, the service provider, and the service registry.
As presented, WSDL provides a model in XML format for describing what a web service offers. A service description in WSDL separates abstract-service functionality from details such as how and where the service is offered. SOAP provides the messaging format used by the services and its requester. UDDI provides the standardized service registry format. It also provides an infrastructure that supports the description, publication, and discovery of service providers. UDDI further assists in the technical details for accessing the web services. A core aspect of UDDI is how it organizes information about services and the providers of services. Information entities (UDDI data) are organized in a data model and stored in a UDDI service registry. Although there are several UDDI’s application programming interfaces (APIs), it is worth to bring two major API of UDDI registry. The first is publishing information about a service to a registry, which publishes, deletes, and updates registry–related information. The second one is searching a UDDI registry for information about a service.
Web services have an interface described in a machine-processable format (specifically WSDL). Other systems interact with the web services in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other web related standards. Because of increasing attention to internet-based services, many companies have begun to rely on web services for their enterprise applications, and as a result their reliability more and more is dependent on these web services that are often provided by various enterprises.

As an example of SOA, the interaction between the server and the web service client in Figure 1.1 is broken into the following steps presented in Figure 1.4.
The steps are explained as follows:

a. This step is an optional step for a case where a web service client has no information about the address of the target web service. Therefore, in the first step the client has to discover a web service that provides the information needed by the client. Optionally a web service provider can register a service with a web services registry such as Universal Description Discovery and Integration (UDDI). In the example of Figure 1.4, to know about a bookstore web service, a web service client attempts to discover the address of a desired web service.

b. The web service registry responds by sending a list of servers that are able to provide the services needed by the client. Depending on the type of services registered in the web service registry, it may send a single or more servers’ information to the client. In this example, the client is provided with Server 2 information.

c. After locating the server containing the web service, the client needs to know how to invoke the web service and what information is provided by the web service. For example, the method that client has to invoke is called "String getBookISBN(String bookTitle, String bookAuthor)". So the client needs to know the method name and the input and output parameters. In general, the client needs the web service to describe itself.
d. The server side of the web service responds in an XML type language called WSDL, which defines all the information about the web service that a client must know before invoking the service. For example, the method "String getBookISBN(String bookTitle, String bookAuthor)" is defined in the WSDL file, which shows how a web service client can send a request to the web service.

e. The invocation can be done using SOAP. Therefore, this hypothetical client makes a SOAP request to ask about the ISBN of a book. As described earlier, SOAP request is an XML format message itself which is used for transmitting request and response messages in web services. In the example of Figure 1.4, the name of the method (getBookISBN) and the parameters’ values are included in a SOAP message and are sent to the server.

f. In response to receiving the author’s name and the book title, the web service returns the corresponding ISBN of the requested book. An error message will be sent to the client if the SOAP request from the client was incorrect or any other failures happened in all the intermediate hardware, software, or network layers.

In general web services have the following fundamental characteristics [Glo]:

- As web services use standard XML languages, they are platform-independent as well as language-independent. This means that a web service client can be programmed in a language like C and running under UNIX, while the web service itself is written in Java and running under Windows.
Most web services use HTTP protocol for transmitting web service requests and responses. This is a major advantage to build an Internet-scale application, since most of the Internet's proxies and firewalls easily work with HTTP packets.

1.3 RELIABILITY ANALYSIS

With the increasing prevalence of and reliance on web services, the reliability of SOA implementations has come under scrutiny. Reliability is considered as one of the main characteristics of system dependability. It is defined as “probability of failure free operation of a computer program in a specified environment for a specified period of time” [Mus84]. Since the early 1970s, numerous software reliability models have been proposed. The most popular ones are Software Reliability Growth Models (SRGM), which are used in different research works such as [Goe85, Rah10, Ram82, and Woo96]. Reliability growth models are the ones that apply statistical distribution model in order to predict the reliability of software based on their previous failure information. The previous failure information can be gathered from testing phases or during the maintenance of the system. These models treat the software system as black-boxes, i.e. the internal structure of the software system is ignored. They can be considered as oblivious reliability models because failure sources are not taken into account in calculating the reliability estimate. They also are very useful to analyze the reliability of custom standalone systems [Gok06b]. However, with the widespread use of distributed, heterogeneous, and component-based software, it can be inappropriate to model the overall failure process of such systems using one of the several software reliability
growth models such as models presented in [Pha00 and Yos91]. Recently (after 1990), architecture-based software reliability analysis has started receiving a great deal of attention [Gok97, Kri97, Yac04, and Gok07] as software applications have grown in size and complexity. The architecture-based, also referred to as white-box, analysis takes into account the components and interaction among the components. An overview of architecture-based reliability analysis is presented in the next section.

1.3.1 ARCHITECTURE-BASED RELIABILITY ANALYSIS

A thorough discussion of architecture-based reliability approaches and their limitations is presented in [Gok07 and Gos01a]. Architecture-based software reliability allows for 1) better understanding of the system reliability based on its component reliabilities, 2) identifying the critical components with profound effect on the overall reliability, and 3) selecting an architecture that is more reliable for a given application.

Common structural properties in architecture-based reliability analysis include system decomposition, software architecture, failure behavior, and finally combination of software architecture and failure behavior:

**System decomposition**- Although there is no universally definition for a component, it is conceived as a logically independent unit of the system which performs a well-defined function. Component definition is a user level task that depends on the factors such as system being analyzed and possibility of getting the required data. There is a trade off in defining the components. Partitioning the software system into too many small
components may pose difficulties in measurements, parameterization, and solution of the model. On the other hand, too few components may cause the distinction of how different components contribute to the system failures to be lost. The level of decomposition clearly depends on the tradeoff between the number of components, their complexity and the available information about each component [Gos01a].

**Software architecture**- System components and their interactions shape the software architecture. The architecture of an application may not always be readily available. In such cases, it has to be extracted from the source code or the object code of the application. Software architecture may also include the information about the execution time of each module [Gos01b].

**Failure behavior**- Failure can happen during an execution period of any component or during the control transfer between two components. Failure behavior is concerned with the frequency of failures of a specific type during a period of time. The failure behavior can be specified by component reliability, a constant failure rate, or time-dependent failure intensity. Assessing the failure behavior of software components depends on many factors such as whether or not component code is available, how well a component has been tested, and whether it is a reused or a new component [Gok06b and Lyu96].

**Combination of software architecture and failure behavior**- Based on the approaches used to combine the software architecture with the failure behavior, two major models can be recognized [Gos01b]:
Path-based models - These models consider all the execution paths of the program [Sho76, Yac99, and Gok06b]. A sequence of components along various paths can be extracted either by testing or experimentally. The reliability in each path is obtained by multiplying the reliabilities of the components along that path. Finally, the overall reliability of the software is computed by averaging the reliabilities over all paths.

State-based models - These models estimate software reliability using control flow graph that form the software architecture [Gok98, Lit79, and Che80]. The main literature approaches for these models are Markov Chain (MC) and Stochastic Petri Net (SPN). MC assumes that the transfer of control between components has a Markov property that can be Discrete Time Markov Chain (DTMC), Continuous Time Markov Chain (CTMC), or Semi Markov Process (SMP) [Kub89]. The generation of a MC by hand for complex systems is rather tedious. The detailed required knowledge of MC has led to a new modeling approach that is more intuitive to understand, easier to develop, and can provide for a more concise specification of a model. This new formalism modeling called SPN is capable of translating a model into CTMC and automatically solving it without user involvement. Some examples of packages based on SPN are SPNP [SPN], SHARPE [SHA], UltraSAN [Cou91], and Mobius [Mob].

In general, the increase in popularity of architecture-based reliability approaches can be contributed to the following:
Many software systems no longer execute on a single host. Therefore, by treating them as monolithic entities, reliability evaluation of these systems may no longer reflect a true reliability estimate.

Architecture-based reliability analysis provides better understanding of the relationships among components, guiding the improvement of architecture designs.

Architecture-based reliability analysis can identify the critical components that show the most sensitivity in the overall system reliability.

1.3.2 Web Service Reliability Analysis

As web services have many advantages, they are used in many applications throughout the internet. Therefore, high level reliability and performance are expected from them and a number of approaches to reliability prediction and performance evaluations have been attempted so far. These approaches are very broad and they cover different aspects of web service reliability and performance [Cao03, Van01, and Zhe10]. However, to analyze the reliability of web services, one may need to investigate the failure types in web-based applications. The main causes of failures in web applications have been analyzed in a report from Carnegie Mellon University [Per05]. This report has used information from some major companies such as Google, IBM, Intel, Microsoft, Hewlett-Packard, Oracle, and Sun. Based on this report, the significant cause of failure is not logical or computational error, but rather system overload, configuration errors, and resource exhaustion. Many of failures appear because of lack of sharing resources in run-
time environment, and human/operator errors. These failures can become a major problem in reliability degradation of web services. To the best of our knowledge, many of architecture-based reliability approaches applied in service-based environment do not consider these failures [Gra05 and Zho06]. Although some authors [Gra06] claim that architecture-based reliability approaches can be applied to service-oriented computing applications, no solid work exists to prove this. For instance in [Gra05], the author presents an approach to the reliability prediction of an assembly of services, that uses the architecture-based reliability analysis. This approach does not provide a definition of failure in a service-oriented environment and consequently it does not test its presented solution on a real-world case study.

The definition of software reliability assumes that a concrete definition of a failure exists. This meaning of failure may change depending on the application or the environment in which the application runs. Because of the inherent complexity of web service environment [Krk08] and the limitations described, this study follows the reliability definition presented by Hwang, et al. [Hwa08], which defines the web service reliability as “the probability the web service successfully responds within a reasonable period of time”. This definition assures that a web service can fail through a malfunctioning or unavailable service at runtime. Therefore any time-out or resource exhaustion situation caused from misconfiguration of underlying shared resources is considered a failure as well.
Besides the issues discussed above, several other limitations have been observed in previous research works as presented as follows:

- Many of the reliability and performance research methods consider monolithic and non-distributed software system in their analysis. Due to increasing use of distributed systems, presenting a new approach to analyze complex and non-monolithic systems is required. As these systems cannot be presented as one black-box system, the reliability approaches need some modifications and enhancements when they are used for distributed and network-based software systems, such as web services. Also many of these software systems are dependent on several underlying layers such as web server, application server, and databases. These layers form what is called the middleware. A web server, such as Apache [Apa], is the front end software that delivers requests to the application server on which the web service runs. A database stores data relevant to web services. Therefore, one cannot underestimate the importance of these layers in the reliability analysis of the applications.

- A large number of studies on reliability and performance analysis of software systems are considering just theoretical analysis without investigating the real-world applications and tackling the challenges of complex software systems. Since many of these approaches validate their models using simulation-based analysis or simple applications, the applicability of these approaches for real-world complex software systems is unknown.
There are several research papers which use simulation tools such as Markov chains to model the system and analyze its reliability/performance. However, the software systems under study are simple and rudimentary. Since presentation of a complex system using Markov chain models may probably lead to state explosion, it would be difficult to expand the model for large systems.

1.4 **Research Objectives**

To tackle the limitations provided in the previous section, we conducted a study on “reliability” evaluation of a banking web service deployed on middleware software called JBoss AS [Jbo]. This study investigated both black-box and white-box (also referred to architecture based) reliability analysis. The overall web service infrastructure will be considered as a multilayered system, with each called-layer serving the calling-layer. Since a service is managed by an underlying framework such as a middleware, the effect of underlying layer configuration parameters on reliability of a deployed web service would be modeled and a simulation based approach will be presented to estimate the reliability of the software system. This research aims to use simulation-based approach as well as an empirical approach to a) simulate the overall web service system and its internal structure b) predict the expected failure rate and reliability of a web service system using its presented architecture, and c) validate the simulation-based reliability model using empirical results gathered from actual test-cases in the lab environment.

In order to conduct the research, an experimental environment is set up to collect runtime information such as the web service response time, the overall time spent in each
software layer, and the failure probability of requests made to the web service. During the study, the software system, i.e. web service, application server (JBoss AS), storage, and etc., is instrumented to form the call-graph. This call-graph that represents the behavior of the system at run time is then transformed to Petri net models. The call-graph can be transformed into either a black-box or a white-box model. Therefore, Petri net models can be developed as simple, representing the black-box, or as a complex model representing the white-box approach. Furthermore, the complex model can be built as a flat or a hierarchical structure where the Petri net sub-models are associated with the software system components/layers. Obviously, the simulation results of the Petri nets for both the black-box and the white-box must be the same. Since the black-box approach is simpler and more manageable, its analyses results will be used to validate those of the white-box approach. The main contributions of the research can therefore be summarized as the following:

- The research will be aimed at both simulation and empirical, in addition to a simple analytical model that estimates the system performance. As indicated, the majority of research works are theoretical or simulation-based approach. The main contribution of the model is using empirical research results to validate the simulation-based outcomes.

- The entire software system is involved. To the best of our knowledge, no solid research exists on reliability analysis that involves both web services and the underneath architecture on which the services are run. As presented in the previous section, most of the research studies model a partial system and not considering a complete web service
system. In this work, we made an effort to present a complete picture of the system using static and dynamic analyses of the web service environment in general. The web services are simple and are treated as atomic entities, whereas the middleware software is treated as a white-box.

- One of the major hurdles to optimum performance is misconfiguration of system parameters. So, special focus will be paid to some of the important configuration parameters. Configuration parameters such as HTTP thread pool in web server and DB connection instance pool have a major impact on failure rate and consequently estimated reliability of the system. Therefore, these two parameters are considered in the final reliability model.

- A good portion of research studies on reliability growth of software systems are based on the history of failures to predict the current or future reliability. When it comes to web services, the failure history may not be a good metric to measure reliability because the entire environment is so dynamic and changes continuously. This research study, builds a simulation model that allows the user to dynamically change the parameters of interest and immediately observe the results. This will provide the user or the system administrator great insight in how to manage the sensitive elements that have greater impact on the overall health of the system.
This research attempts to include the major aspects of web service environment that affect the reliability of web services. The advantages of the proposed multilayer approach include the following:

- The web service environment contains a large number of internal applications, utilities, and enormous number of classes, e.g. Java classes. The proposed multilayer approach provides for a more manageable environment in achieving the reliability of a web service. Specifically, the multilayer approach leads to a more straightforward reliability modeling, as layers and configuration parameters are easier to define. For example, web service, web server, application server and database can be considered as four different layers.

- As failure rate of each layer and configuration parameters are estimated separately, analyzing the reliability of each layer based on their failure rate would be more straightforward. If one wishes to focus on specific layer reliability at a finer granularity, the other layers can stay intact as this approach can have the plug-and-play capability.

- The analytical and simulation-based model can validate each other to ensure the correctness of the results. Therefore, the simulation model can be applied to complex real-world case studies.
1.5 **Dissertation Organization**

The rest of the dissertation is organized as follows. Chapter 2 presents the background information and some related works. Different concepts such as previous research works, Petri net and Stochastic Activity Network (SAN) are explained in this chapter. Chapter 3 discusses the research methodology, which proposes a new approach in architecture-based software reliability. This new architecture-based reliability model is tailored towards service based applications. This approach presents a layered model in a service oriented system and specifically concentrates on atomic web services in the context of an application server. Chapter 4 includes the results of the simulation-based and the actual model from the presented architecture-based reliability model using a case study named Duke’s Bank web service [Duk]. Finally chapter 5 concludes the research and discusses future research.
CHAPTER 2

LITERATURE REVIEW AND BACKGROUND
2.1 INTRODUCTION

This chapter presents an overview of related works on reliability analysis of web services and software systems in general. Although there are different ways of categorizing the related research endeavors, to better appreciate the research contributions, our approach is to classify them based on two main tracks: simulation and experimental.

The chapter is organized into the following sections. Section 2.2 provides an overview of major challenges in web service reliability. It discusses the major levels involved in delivering services that may impact the reliability of web services. Sections 2.3 and 2.4 are devoted to the reliability analysis of several research works based on the simulation and the experimental tracks, respectively. Section 2.5 provides a summary for Petri net models. Section 2.6 presents an overview of this chapter.

2.2 CHALLENGES OF WEB SERVICE RELIABILITY

Reliability analysis of web services is a challenging task. Recall that web service reliability is defined as the probability the web service successfully responds within a reasonable period of time. To measure the reliability of web services accurately, one needs to understand their structure. Their reliability is impacted by several factors including the performance and fault-tolerance of the web services end-points, the reliability features of the protocols, and the transport mechanisms used to access the web services [Err05]. According to Erradi et al. [Err05], reliability of web services needs to address four levels of challenges:
• **Service provider level** - Reliability at this level focuses on the service hosting container. To guarantee user satisfaction, a service provider must ensure the database server; the application server and the web server operate properly. The service provider must also be concerned with redundancy of computation and data (using standby servers and redundant network connections) to achieve fault tolerance, and to provide load sharing to achieve a good balance between cost and performance.

• **Transport level** - Reliable messaging is a major challenge at this level. Several technologies implement reliable massage delivery for web services at the transport layer. HTTPR [Tod01], developed by IBM, is one such technology that reliably delivers the HTTP packets. This protocol, however, seized to exist due to lack of consensus among major industry participants. WS-ReliableMessaging (WSRM) [WSR] is a more recent protocol that is aimed at improving the reliability of web services. More specifically, it ensures SOAP messages are delivered reliably between distributed applications in the presence failures, such as failures in the network or software failures. WSRM is concerned with the endpoints, i.e. the SOAP client and the SOAP server. Thus, WSRM does not guarantee reliability for multi-hop messaging over different protocols as it assumes that reliable transport protocols will be available for the entire path between the sender and the receiver.

• **SOAP message level** - Reliability at SOAP message level requires reliable messaging protocols that are transport-independent. SOAP-based protocols like web service reliability are promising specifications aiming to standardize the rules for
message delivery such as duplicate elimination, ordered delivery, persistence, and acknowledgment. These protocols might also cause extra inefficiency when the underlying transport layer already provides reliable messaging.

- **Business process level** - The reliability analysis approaches in this level mainly concentrate on composed web services rather than atomic ones. Addressing reliability at the business process level makes an effort to provide dependable composition of web services. It also involves thorough specification of the dependable behavior of individual web services and their composition to ensure the dependability of the resulting systems.

Among these presented levels, the proposed research work is concerned with the first level, i.e. the service provider level, and a state-based analysis (defined in a later section) tool is used to model the service provider. As web services at this level, similar to most of the internet-based applications, depend on several software layers and their interactions (e.g. web server, application server, and database), their reliability analysis is different from standalone applications with a simple architecture. A complex call-graph between different layers of such an application and the difficulty of modeling this call-graph makes it challenging to present a stable reliability model. For this main reason, most approaches to reliability analysis of web services and in general software systems are based on hypothetical and simple case studies. To understand the state of the art of these research works, we have divided these research works to two main categories. The first category focuses on simulation-based analyses of software systems reliability and
performance in general, and the second category concentrates on experimental aspect of performance and reliability analyses.

2.3 **Simulation-based Reliability Research**

Some of the most common approaches in architecture-based reliability analysis are based on Markov, Petri net, and Stochastic Reward Net models [Che80, Gok06a, Rah11, and Wel01]. The majority of the research in this area is theoretical with less emphasis on experimental analysis to support the theoretical results [Sin01, Sou06, and Zho06]. For example in [Che80], which is one of the pioneer works in architecture-based software reliability, the author presents a user-oriented reliability model to measure the reliability of the software. In this research, the reliability of the software system depends both on the reliability and the utilization probabilities of the software components. This paper transforms a simple software system architecture to a Markov model and then evaluated to determine the overall system reliability. Although this is one of the seminal works in reliability analysis of a software system, the case study is hypothetical, very simple, and does not take into account the complexity and the distributed nature of many today’s software systems.

In [Zho06], the authors focus on performance and reliability of web service composition. They propose a Petri net (discussed later) based approach to predict the reliability of web service composition. This work presents a transformation algorithm from Business Process Execution Language (BPEL), which is the de facto industry standard of Web services composition specification, to stochastic Petri nets (SPN) models. However, this
paper emphasizes on theoretical aspect of research and does not include any experimental analysis.

In another work [Sou06], the authors presented a simulation-based model for performance prediction and evaluation of application servers using Petri nets. In order to do this, a Petri net model that represents the pooling mechanism of Enterprise Java Bean (EJB) in JBoss Application Server [Jbo] was designed. EJB is a managed, server-side component architecture for modular construction of enterprise applications [Ejb]. This is an interesting work since it investigates the utilization of simulation models for performance prediction of an application server by looking at the pooling mechanism used in the EJB container. However, in this paper, the numerical illustration of the presented simulation model is based on hypothetical input and not real experiments.

In [Sin01], the authors present a new approach to reliability analysis of the component-based systems. This analysis framework is integrated with Unified Modeling language (UML) models [Fow03]. In this model, the authors propose a theoretical model to predict the reliability of the system in the early phases of system analysis and design. In the early phases of software development, e.g. requirement elicitation and system analysis, use-case diagrams and sequence diagrams [Fow03] are often used. Instead of using traditional use-case and sequence diagrams, the authors suggest using some annotated information about the expected system usage patterns and failure probabilities of the components. This model makes an effort to analyze the system reliability before the actual implementation of the system. Although, this study tries to demonstrate a new reliability
prediction approach, the methodology lacks the steps to validate the model using a real-world application that is most likely more complex.

Gokhale et al. [Gok06a] have proposed an analysis methodology based on the Stochastic Reward Net (SRN) [Sak98] modeling paradigm to quantify the performance and reliability tradeoffs in the process-based and thread-based web server software architectures. The authors illustrated the value of the methodology with several examples in hypothetical situations.

In [Wel01], a framework for construction of Colored Petri Net models [Jen87] for performance analysis of a hypothetical web server is presented. It was demonstrated how the framework can be used to build a Colored Petri Net\(^1\) (CPN) model of a web server environment consisting of a HTTP web server and web clients connected to a LAN. Although the model may be used to investigate the performance of a web server; the model itself is simplistic in the sense that it only simulates a web server with clients connected to the web server without any experimental validation.

In [Rah11], the authors presented a simulation-based model using Petri nets that models the reliability of a web service deployed in an application server. This methodology used concentrates on interdependencies between web services and their underlying layers. Like many other presented studies in this category, this paper also lacks the experimental analysis of a real case study.

\(^1\) CPN is a specific type of Petri Nets that is well suited for studies where communication, synchronization, and resource sharing are important.
In general, although the research efforts described are very useful in principle for reliability estimation of software systems, they lack the experimental analysis and the inclusion of real-world applications that are often large, complex, and distributed.

2.4 EXPERIMENTAL-BASED RELIABILITY RESEARCH

There are some studies that use a combination of theoretical and experimental analyses in reliability estimation of software systems [Xia10, Van01, Cao03, Zhe10, and Gos05]. Xiao and Dohi [Xia10] focused on the relationship between the Apache server error rate and the system’s performance. They developed a probability model to describe the relationship between the error rate which is one of the representative reliability measures in Apache web servers and the system parameters which reflect on the web server's system performance. They implemented a simple client server system and carried out an experiment to measure both the error rate and the system parameters such as the available virtual memory size. This is an interesting work since it is focused on the theoretical analysis of a real-world system as well as comparing it with empirical error rate data. Although it is concentrated on performance analysis of a web-based system deployed in a popular web server, it did not analyze the internal architecture of the system.

In [Van01], the authors focused on a performance evaluation model of a web server system, which is based on a queuing model and evaluated the effectiveness of the model through the experiments in a lab environment. This research considers the impacts of the client workload and the server hardware/software configuration on performance analysis.
of web servers. The result of performance analysis can be used for tuning and sizing a web server. The study also concentrates on experimental analysis of a web server, but it does not cover more complex internet-based software systems such as application servers and web services. Also it lacks reliability analysis of the system as it focuses on performance estimation.

In [Cao03], the authors proposed another queuing model to evaluate the performance parameters of a web server such as the response time and the blocking probability of services. The response time is considered the time difference between when a request is sent and when a successful reply is fully received. An HTTP request sent by a client is considered to be blocked either when the maximum number of connections in the server has been reached or the TCP connection is timed out at the client computer. A TCP connection will be considered as timed out by a client when it takes too long for the server to return an ACK in the SYN-ACK of the 3-way TCP handshake [TCP]. The blocking probability was then estimated as the ratio between the number of blocking events and the number of connection attempts in a measurement period. The authors then have presented a queuing model of a web server and obtained several expressions for web server performance metrics such as average response time, throughput and blocking probability. The authors also validated the model through four sets of experiments.

The authors in [Zhe10] evaluated the response time and throughput of web services by collecting and analyzing large sets of data from geographically distributed locations. The authors propose a collaborative reliability prediction model, which applies the past failure
data of other similar users to predict the web service reliability for the current user. This collaborative mechanism is proposed as a mechanism that is applied to the past failure data of the web service. The key idea is that the web service users are encouraged to pass their individually observed web service past failure data to the users of other similar web services. This failure data of similar web services are used to estimate the reliability of web service under study. This work mainly focuses on service-oriented systems, which are publicly accessible and are invoked by a lot of service users. Although it is possible to estimate the reliability of web services based on failure data observed from user, the web service itself and the underneath architecture of the servers are treated as a black-box system and thus the internal architecture of the system is not considered.

Finally, Goseva et al. [Gos05] presented an empirical as well as theoretical study of architecture-based software reliability on a large open source application with 350,000 lines of C code. They emphasized on theoretical and experimental results on a large scale field study to test and analyze the architecture-based software reliability. Although the work provides valuable insights, the application considered is a standalone system and not a distributed or internet-based application.

A summary of these two tracks of research is presented in Table 2.1. As presented in this table, there are not much white-box reliability models that are focused on actual real world systems. Although [Gos05] is the most relevant research study, since it entails a white-box reliability model as well as a realistic software system, it does not support a service oriented environment. To the best of our knowledge, this dissertation is a unique
in that it employs a combination of simulation, experimentation, white-box reliability analysis, and uses a real-world service oriented environment.

Table 2.1 a summary of relevant research studies (S: Simulation, E: Experimentation, WB: White-box, BB: Black-box, SAN: Stochastic Activity Network, MC: Markov Chain, R: Reliability, P: Performance)

<table>
<thead>
<tr>
<th>Reference</th>
<th>Category</th>
<th>Granularity Approach</th>
<th>Model Type</th>
<th>System Type Observation</th>
<th>Obtained Metric</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Che80</td>
<td>S</td>
<td>WB</td>
<td>MC</td>
<td>Hypothetical component-based system</td>
<td>R</td>
<td>One of the earliest pioneer works; measures system reliability; determines modules most critical to system reliability (sensitivity analysis)</td>
</tr>
<tr>
<td>Gok06a</td>
<td>S</td>
<td>WB</td>
<td>SRN</td>
<td>Web server</td>
<td>P, R</td>
<td>Quantifies performance &amp; reliability tradeoffs between process-based &amp; thread-based web server</td>
</tr>
<tr>
<td>Rah11</td>
<td>S</td>
<td>WB</td>
<td>SAN</td>
<td>Web service</td>
<td>R</td>
<td>Accounts for the configuration parameters and the major layers of the underlying software</td>
</tr>
<tr>
<td>Sin01</td>
<td>S</td>
<td>WB</td>
<td>UML, Bayesian</td>
<td>Hypothetical system</td>
<td>R</td>
<td>Estimates reliability in early phase of software development</td>
</tr>
<tr>
<td>Source</td>
<td>Type</td>
<td>Model</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>-------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sou06</td>
<td>S</td>
<td>WB</td>
<td>DSPN&lt;sup&gt;2&lt;/sup&gt; Application Server (JBoss AS)</td>
<td>Performance modeling of EJB container in JBoss Application Server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wel01</td>
<td>S</td>
<td>WB</td>
<td>CPN HTTP web server</td>
<td>Analyzes performance of a web server in various conditions such as arrival rate and alternative configurations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zho06</td>
<td>S</td>
<td>WB</td>
<td>SRN Hypothetical composed web service</td>
<td>Web service composition is described in BPEL&lt;sup&gt;3&lt;/sup&gt; specification, which is then transformed to SRN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cao03</td>
<td>E &amp; S</td>
<td>BB</td>
<td>Queuing Web server</td>
<td>Evaluates the response time and blocking probability of the server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gos05</td>
<td>E</td>
<td>WB</td>
<td>Statistical GCC package</td>
<td>A reliability analysis of a large scale software package implemented in C language.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Van01</td>
<td>E &amp; S</td>
<td>BB</td>
<td>Queuing Web server</td>
<td>Implemented a simulation tool for tuning and sizing a web server</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

<sup>2</sup> Deterministic and Stochastic Petri Nets (DSPN) is an extension of GSPN that allows for transitions to be associated with a constant delay.

<sup>3</sup> Business Process Execution Language (BPEL) is short for Web Service BPEL (WS-BPEL) which is a standard language for the specification of executable and abstract business processes with web services.
2.5 PETRI NETS

One of the major problems in the area of modeling has to do with the great detail that must be invested to describe the system. This difficulty often leads to complex models and huge state space. Petri nets and their advances have contributed to the mitigation of this difficulty. A Petri net is a modeling language that describes the stochastic system behavior graphically. A basic Petri net graph consists of the following components: places, tokens, transitions, input arcs, and output arcs. These graphical components that are shown in Figure 2.1 are explained below:

- **Places**: These are similar to variables in high level languages that hold values.

- **Tokens**: Tokens show the value of a place. A snapshot of these values in all places represents the state of the system. The snapshot of the system is referred to as marking of the system.
- **Transitions**: Transitions represent the system activities. The state of the system changes through these activities. A thin bar, also called *immediate transition*, is an instantaneous activity that takes no time. Whereas, thick bars are used to show *timed-transitions* that require some time for an activity to complete. For instance a timed-transition could represent the time for generation or transmission of a packet. The Petri nets with timed-transitions are called *timed Petri nets*.

- **Input arcs**: These arcs connect a place to a transition. The places connected to the input arcs are called *input places*.

- **Output arcs**: These arcs connect transitions to places. These places are called *output places*.
A transition is enabled if for each input place of the transition, the number of tokens in
the place is at least equal to the arcs connecting them to the transition. When a transition
is fired, a token is consumed for each input arc, and a token is produced in each output
place for each corresponding output arc. An example is shown in Figure 2.2.

![Diagram of Petri net]

Figure 2.2 a) before the transition fires, b) after the transition is fired.

The transition time is the time period between the transition is enabled and the time the
transition fires. The transition will fire if it stays enabled for the entire time period;
otherwise it becomes disabled. Since immediate transitions fire instantaneously, they
have higher priority over timed-transitions and can fire indeterministically.

Figure 2.3 shows an example of Petri nets [BAL]. This Petri net simulates the producer
and consumer problem by using a buffer to communicate the items produced to the
consumer. In this model, *produce* indicates that the system is ready to produce an item.
When *T1* fires, an item produced is placed in *send*. If there is a free buffer available,
transition *T2* will fire, which takes a buffer space and fills it by placing a token in
*busy_buffers*. At this moment, the item can be consumed if the consumer is ready to
consume it. The consumer readiness is indicated by having a token in *receive*. Once the
consumer is ready and an item is available in *busy_buffers*, *T3* will fire indicating that the
consumer is ready to consume the item. Upon firing of *T3*, the consumer removes the
item from *busy_buffer* by returning its space to *free_buffers* and places a token in
consume indicating that the item is going to be consumed. The item will be consumed when $T4$ fires. This deposits a token in receive indicating that the consumer is ready to consume another item. This Petri net is a non-stop model and automatically produces tokens and sends them to an intermediate buffer where they will be consumed by the consumer part of this model.

Set of places: $P = \{P1, P2, P3, P4, P5, P6\}$

Set of transitions: $T = \{T1, T2, T3, T4\}$

Initial marking: $M_0 = \{1, 0, 0, 2, 1, 0\}$

Figure 2.3 a producer/consumer Petri net

There are many extensions to Petri nets, e.g. Colored Petri nets [Jen87]. One category of Petri nets is called Stochastic Petri Nets (SPN). SPN [Bau02] is a subsidiary of timed Petri net that adds non-deterministic time through randomness of transitions. Generalized Stochastic Petri Nets (GSPN) is a SPN performance analysis tool that uses the exponential random distribution, and thus conversion to Markov Chains is automated. Stochastic Activity Network (SAN) is a structural extended version of a GSPN with many added features such as enabling functions, probabilities for transitions, and reward functions [San01].
The simulation package, i.e. Mobius [Mob], used in this research is based on SAN. Mobius provides a user friendly graphical interface and a high-level modeling formalism with which detailed performance, dependability, and performability models can be specified relatively easily [Man10]. This is a software package for studying the reliability, availability, and performance of various systems, such as networks and distributed systems. One advantage of Mobius is the ability to build hierarchical models. The hierarchical models are very useful for complex systems. This type of modeling has similar advantages to modular programming and one may present a complex model with several smaller sub-models. To support hierarchical modeling, Mobius has the option of composed model. Different submodels can be connected together through identifying sets of shared state variables. For instance, it is possible to compose two Petri net models by holding a particular place in common. That allows for interaction between the submodels, since both can read from and write to the identified state variable [Man10].

Although SANs are an extension of GSPNs, they do not enhance the modeling power of SANs [Tri93]. However, SANs allow for a much more concise description of the system than GSPNs. SANs consist of four primitive objects: places, activities, input gates, and output gates. Places have the same meaning as in Petri net world. Figure 2.4 shows the symbols for the other three objects. In the terminology of SANs, transitions are called activities, which can be of either timed or instantaneous type. So, activities represent actions of the modeled system. Case probabilities, represented as small rectangles to the right of an activity, model the probabilistic outcomes once the activity completes. If no circles are used, the outcome occurs with probability of one.
Input gates (IG) use enabling predicates that control execution of the network activities. An enabling predicate is a Boolean predicate that defines whether the activity that the gate is connected to is enabled. The other arcs emanating from the gate are connected to the places that the input gate depends on. An activity is enabled if all predicates connected to the activity are true and there are enough tokens in the input places to enable the activity. In Figure 2.4, the activity (timed-transitions) is enabled if the IG predicate is true. These predicates that are written in C language can be simple or as complex as what the user wishes them to be. This enhances the modeling flexibility tremendously in comparison to most SPN formalisms that use inhibiting arcs with simple predicates to disable/enable transitions.

Finally, an output gate (OG) is used to change the marking of the system when the activity it is connected to completes. The other arcs connected to the Output Gate are from the places whose markings are affected when the activity completes. Figure 2.4 shows that the marking of two places are affected by the OG. The completion function defined for an output gate may use if-then-else statements and in general be complex. If an activity is directly connected to an output place, this is equivalent to an output gate that increments the output place [Man10].
To have a better understanding of IGs, OGs, and transition cases, Figure 2.5 depicts an example of a Triple Modular Redundancy (TMR) system [TMR] with repairable spares. In perfect working conditions, all three modules that exist in TMR produce the same output. TMR majority-votes the output of the modules to mask the effect of at most one faulty module. Therefore, the system survives if there are at least two healthy modules. This passive mode of operation can be turned into an active TMR, if the system is capable of self-reconfiguration by removing the failed module and replacing it with a spare. In Figure 2.5, as the result of a failure, i.e. when \textit{Failure} fires, \textit{OG} checks to see if there are any spares to replace the failed module. If so, the failed module is deposited in \textit{ToRepair} to be repaired by incrementing \textit{ToRepair}. Once the \textit{Repair} activity is finished, two outcomes are possible. Either the module is repaired with some probability \( p \) or not repaired with probability of \( 1 - p \). If the unit is repaired, it is put back in \textit{Spares} to be used in the future. However, if the unit is damaged beyond repair, it is moved to \textit{Unrepairable} place. The value of \( p \) is set using the global variable \textit{recovery\_probability}.

The \textit{else} clause on the \textit{OG} code indicates that there are no spares left and there is less than two healthy modules. In that case, a token is placed in \textit{Down} signaling that the system has failed. The \textit{IG}, which depends on \textit{Down}, then disables the activities \textit{Failure} and \textit{Repair} in order to seize execution of the model.
Figure 2.5 a SAN model for TMR with repair

OG function:
if (Spares \rightarrow Mark() > 0) {
    TMR \rightarrow Mark() = TMR \rightarrow Mark() + 1;
    Spares \rightarrow Mark() = Spares \rightarrow Mark() - 1;
    ToRepair \rightarrow Mark() = ToRepair \rightarrow Mark() + 1;
}
else {
    if (TMR \rightarrow Mark() < 2) Down \rightarrow Mark() = 1;
}

IG predicate:
Down \rightarrow Mark() == 0

Global variable:
recovery_probability: 0.95
As indicated, Mobius is the simulation package chosen for this research. The reasons for adopting Mobius are:

- Mobius is based on SANs, which allow for describing system behavior in a convenient and graphical way that is easy to understand.

- Mobius provides a user friendly interface and allows for hierarchical submodeling of Petri nets.

- It is much easier to work with Mobius rather than dealing with Markov chains directly. In Mobius, SANs are converted to Markov chains automatically.

- Mobius allows for natural modeling of system parameters, e.g. configuration parameters, and complex behaviors can be expressed compactly.

- SANs support system dependability evaluations in the form of reward functions, e.g. at specific points of time or over time durations.

2.6 SUMMARY

This chapter presented a review of related works on software reliability and performance. The previous research works in software reliability and performance analysis are discussed in this chapter. Furthermore, the chapter provided an introduction to Petri nets
and specifically SANs that are going to be used in the following chapters\textsuperscript{4}. The next chapter will discuss the methodology in achieving the dissertation objectives.

\textsuperscript{4} In this dissertation, we may use SAN and Petri net interchangeably.
CHAPTER 3

THE RESEARCH METHODOLOGY
3.1 INTRODUCTION

Recall from the previous chapter that the objective of the proposed research is to evaluate the reliability of a web service environment by transforming it into a set of layers first. To obtain the layers, one needs to have a good understanding of the static code structure of the elements involved in the web service environment. Furthermore, due to the complexity of the web service environment, extraction of the layers and understanding the flow control among the layers is aided by making service requests and observing the run-time call-graph. Such instrumentation of the application server further assists in obtaining the rates of transitions among the layers. The call-graph and the transition among the layers determined by the rates are needed to simulate the stochastic behavior of the web service environment using Mobius. The flowchart in Figure 3.1 describes the general approach in evaluating the reliability of the web service environment. The following sections describe the major components shown in Figure 3.1.

3.2 STATIC ANALYSIS

*Extract main layers* – To discover the architecture, it is necessary to investigate the logical and/or physical layers of the system. Although the extraction can be done at a detailed level such as class levels or at a more abstract level such as application level, a suitable and manageable level of extraction needs to be determined. The major layers at the service provider level are obtained using Structure 101[STR]. Structure 101 is a software product that is able to produce multiple views helpful for understanding the architecture. However, due to the complexity of the application server and the limitation
of Structure 101 in producing a satisfactory view of the architecture, manual search in research papers and digging into forums and books [Jam09] [Mar06] have provided valuable knowledge to convince ourselves of the major layers.

Extract configuration parameters – Misconfiguration of parameters can have profound effect on the reliability of web service systems. These parameters are often configured by deployers using XML files before each application server starts up. In this study, JBoss As is considered as the main application server in the experimental analysis. JBoss Application Server (JBoss AS) [Jbo] is an open-source Java based application server.
Since it is Java-based, it operates cross-platform: usable on any operating system that supports Java. At least two major configuration parameters have been identified in the JBoss application server. They are HTTP thread pool in the web container of the web server implemented in JBoss AS and database connection pool of the data access layer [Rag03, Imr07]. Web container is a web server component that provides the network services over which request and response are sent. Data access layer is a layer of software which provides simplified access to data stored in persistent storage. Both of these resources control the number of threads that exist in the application server while the web service is running.

3.3 DYNAMIC ANALYSIS

The purpose of the dynamic analysis phase is to augment the layers with some extra code for gathering information about the runtime behavior of the system.

**Instrument application server with a profiler** - Since the architecture-based software reliability needs insights into the dynamic behavior of the system [Gos05], the software system is instrumented with two different profilers called JRAT [JRA] and Javashot [JSH]. The instrumentation enables one to better understand the difference in behavior of the application server for various web services, and collect relevant information about the deployed web services, e.g. service time and service status. By a different behavior it is meant that the call-graph might change.
JRAT computes information about the time spent in each function, class, or even in a package. JRAT presents its own user interface to show the information gathered from a program in run-time. On the other hand, Javashot generates a good picture of the call-graph in the standard Dot graph description language [Dot] as a text file. More details about JRAT, Javashot and the Dot language will be provided in the next section.

**Execute web service test cases** – Various service requests are created in order to collect three different sets of information. The first set of information represents the frequency or the probability of the interaction among the layers, the second set contains information about the execution times of each layer, and the third is used to collect failure data about each layer. In order to compute the probability of transferring program control from a layer to another layer, a Perl program is written to extract this information from the Javashot output file. Another Perl program is written to gather the average time spent in each method/class/package from JRAT output. The way that each program computes the results is presented in section 3.5. In addition SoapUI [SUI] is another tool that is used to run the test-cases to gather failure data (discussed in next chapter).

### 3.4 FORM ARCHITECTURE MODEL

Considering the major layers and resources extracted from the previous steps, an architecture model will be built. Although the main layers and their relationships are developed during the static and dynamic analyses, running several test cases during the dynamic analysis can provide confidence in the correctness of the previous steps and also assist in the discovery of layer relationships.
For instance, the test cases generated for a case study, called the Duke’s Bank web service [Duk], have produced the layers and the shared resources in Figure 3.2. These layers are consistent with the static analysis. The case study is run on JBoss AS 4.2.2. The figure shows that a request is initiated to the web server at the service provider site using the SOAP/HTTP protocol. This diagram also presents an overall architecture view of the main JBOSS AS components responding to a Duke’s Bank web service client. HTTP thread pool in web container, EJB instance pool in EJB container, and Database connection pool in the data access layer are shown as the three resources in JBOSS AS. Since EJB component in this case study uses the main thread of the program and does not create any new thread in the application server, this resource has not been considered in the reliability modeling. However, two other resources, i.e. HTTP thread pool and database connection pool are considered for reliability analysis. The web server is responsible for interpreting the SOAP messages and communicating the requests to the application server that manages the services provided by the web service layer.

Figure 3.2 architecture of JBoss AS, showing the interaction between layers and configuration parameters
3.5 Parameter Estimation

The purpose of this phase is to attain the information needed to build the stochastic behavior among and within the layers.

*Estimate transition probabilities between layers* - Transition probabilities among layers can be estimated from the information collected from Javashot instrumentation. A sample output from Javashot instrumentation is presented in Figure 3.3. This Figure shows a partial view of a graph presented in Dot format. Dot is a plain text graph description language. It is a simple way of describing graphs that both humans and computer programs can use. Dot graphs are typically files that end with the .gv or .Dot extension [Dot]. Figure 3.3 shows a sample Java program and the resulting Javashot output in Dot format. This Dot format graph starts with “digraph” keyword which means this graph is a directed graph. The first node of the graph is named “Start” which is the starting node of the graph that is manually added to dynamic call graph. Each line in the Dot file represents a method called from a class, and is labeled with its associated method name. Each unique label, regardless of how many times its associated method is called, represents one node of the call-graph. For instance, “main” is the first method called from the class MainClass. When a calling class returns the control of the program to the caller, the line style will be a dashed line. The corresponding visual representation is shown in Figure 3.4.

We then wrote a Perl program to compute the transition probability by enumerating the number of times a particular transition from a node A has transferred to a node B, divided
by the total number of times that node A transferred control to other nodes. This is illustrated in Figure 3.5. The intensity of communication between nodes is shown by the thickness of the edges in the graph. The larger nodes mean that they are used more often than other nodes. As presented in Figure 3.5, the transition probabilities between the Start node and its neighbors have not been calculated in the study, since this node is not part of the actual call-graph.

As a dynamic call graph may contain a huge number of calls and also large number of classes involved, a complete graph extraction is too complex to be converted to a Petri net model. To alleviate this, instead of considering individual classes, we consider the second-level packages (e.g., org.apache, org.jboss, etc.) as our layers. Using a Perl program, we collect the Javashot output and aggregate the number of edges between second-level packages and then compute the transition probabilities as explained in the previous paragraph.
Figure 3.3 an example Java program and its partial Javashot output file in Dot format
Figure 3.4 graph representation of Javashot output

Figure 3.5 the call graph after computing percentage of transitions between nodes
**Estimate time spent in each layer** - By instrumentation of the application server using JRAT during the dynamic analysis phase, information on time spent in each layer can be evaluated. JRAT assists in preparing a profile for each test case, which consists of the time spent in the methods, classes, packages, and consequently the layers called. Figure 3.6 shows a partial view of JRAT user interface. Since JRAT displays the time spent in millisecond in each method, another Perl program is written to compute the average time spent in each layer by adding the time spent in each method that belongs to the same layer.

![JRAT Interface](image)

Figure 3.6 an overview of JRAT interface
Gather failure probability for each layer – There are different types of failures in the web service that are considered in this study. Two types of failures are tailored toward the configurations parameters of JBoss AS such as the maximum HTTP thread pool and the maximum DB connection thread pool. Furthermore, the web service failure rate and the JBoss failure rate, due to some functional errors, are hypothetically injected in the final Petri net model. These injected failures help to understand the sensitivity of system reliability based on each component or layer reliability.

3.6 Form the Petri Net Model

Once transition rates, timed activities, and failure probability of each layer, and the configuration parameters are determined from the previous sections, the overall reliability of the composed model can be evaluated.

Basically the dynamic call graph extracted from dynamic analysis will be transformed to a SAN model. While the main layers involved in the call graph are extracted from dynamic analysis, each layer will be transformed to a timed transition. States of software modeled in SAN are shown by places before and after each layer/timed transition. The conditions that check the configuration parameters’ values in software are shown using input and output gates. The primary configuration parameters set in the software are modeled as initial tokens.

For simplicity purpose in the model, the software is divided to different pieces where each piece presents atomic models in SAN. Each piece of software is a partial call graph
of the software. For example, the interaction between application server and database is a piece of call graph that is modeled as an atomic model. Finally a complete hierarchical model will be built by connecting all of atomic models. The hierarchical SAN model will be presented in the chapter four.

3.7 Gather and Analyze Results

Using the model formed in the previous phase (subsection 3.6), the final phase of this approach is to run the Petri net model, collect, and analyze the results. Some parameters of interest are the estimation of web server and database rejection rates of requests. Also of interest is determining the rates or the available resources that provide the optimum performance under certain conditions. One important aspect of this step is to ensure that the Petri net model results match that of the experimental outcomes. These analyses will be described in detail in the next chapter.

3.8 Summary

This chapter has focused on the strategy on how a complex web service environment will be partitioned into layers so that these layers and their interactions can be simulated by a hierarchical Petri net model. In general the following general steps have been discussed in the next chapter:

- Extraction of the architecture – This requires understanding of the system architecture, the major components, and the configuration parameters in order to acquire the major layers and their relationships.
Gathering behavioral information by running test cases – Test cases are needed to estimate the experimental failures, flow-control probabilities among the layers, and the time spent in each layer. The information collected will be converted into rates, as the Petri net transitions are based on rates. The estimated parameter rates fall into three major categories: the execution rate in each layer, the transition rates among the layers, and the rate of failure for some of the layers/components that can fail due to various conditions.

Hierarchical Petri net modeling – Once the layers and their flow control relationships are obtained, a hierarchical model representing the stochastic behavior of the web service system will be built. Although the entire architectural model can be a flat complex model, the entire architectural behavior can be treated as a hierarchy of submodels. This is similar to software modularization or structured programming. In this way, each submodel can behave in concert with the rest of the model, or can be modified without affecting the rest of the submodels. This allows for great flexibility in manipulating and evaluating the overall model.

Gathering results – The final stage is to run the model and gather results. One aspect of this step is the validation of the results, such as overall reliability, by comparing the simulation results with the actual reliability information extracted from actual test cases in the LAN environment.
CHAPTER 4

EXPERIMENTAL AND SIMULATION RESULTS
4.1 INTRODUCTION

To analyze the reliability of web services, a case study of a web service is implemented and tested in the LAN environment. The web service, called the Duke’s Bank web service [Duk], is a banking service which is transformed from an open source JEE Application called Duke’s Bank application (Figure 4.1). This figure shows the overall architecture of Duke’s Bank application. A bank client sends a HTTP request to the web container, which is responsible to receive the request in HTTP protocol and send it to another component named Entity Java Bean (EJB) container. The EJB container consists of the Duke’s Bank EJBs, which are a special type of classes in Java. These EJBs are the classes that are responsible for the banking business. The EJBs are named TxControllerSessionBean, AccountControllerSessionBean, and CustomerControllerSessionBeans. The first class, TxControllerSessionBean, is responsible for all the banking transactions such as transferring funds from an account to another account. AccountControllerSessionBean is another class which performs all the tasks related to a specific account. For example, it presents the account balance if an Account ID is given. The third class, CustomerControllerSessionBeans, performs all the functions with regard to a bank customer. It determines whether the customers’ accounts are valid or not. The web service built from this application is designed by adding a service-endpoint element to the application. A service-endpoint is an XML file that specifies a specific class that provides the interface of a web service. The class that is selected to be used as a web service is AccountControllerSessionBean, which is a Java class that provides the functionality of accessing the banking accounts information. A WSDL file, which is a descriptive XML file to define the web service, is also generated to describe this new web
service. This new web service is able to provide two services: 1) receives customer ID and returns all account numbers of the customer, and 2) receives an account ID and return the balance of accounts. The first service is performed by a method named `getAccountsOfCustomer()` and the second service is presented by a method named `getAccountBalance()`.

The next step to run this web service is writing a web service client. This client is built using software named SoapUI [SUI]. SoapUI is an open source testing tool solution for service-oriented architectures. With a graphical user interface, SoapUI is capable of generating and mocking service requests in SOAP-HTTP format. Figure 4.2 shows the method calls and the requests that are submitted from the web service client in Soap protocol format.

![Figure 4.1 Duke’s Bank Application Architecture](image-url)
<table>
<thead>
<tr>
<th>Web service Interface</th>
<th>Soap Request sent from Client</th>
<th>Interface Description</th>
</tr>
</thead>
</table>
  `<soapenv:Header/>`  
  `<soapenv:Body>`  
  `<tel:getAccountsOfCustomer>`  
  `<String_1>200</String_1>`  
  `<tel:getAccountsOfCustomer>`  
  `</soapenv:Body>`  
  `</soapenv:Envelope>` | This Interface allows web service clients to send a Bank Customer ID and receive the account ID that belong to this customer. |
  `<soapenv:Header/>`  
  `<soapenv:Body>`  
  `<tel:getAccountBalance>`  
  `<String_1>5005</String_1>`  
  `<tel:getAccountBalance>`  
  `</soapenv:Body>`  
  `</soapenv:Envelope>` | This Interface allows web service clients to send a Bank Account number and receive the balance of that account. |

Figure 4.2 Duke’s Bank web service Interfaces and request example

The rest of this chapter organized into the following sections. Sections 4.2 and 4.3 treat the web service environment as a black box and a white box, respectively. The reason for
using the black box approach is that it can be used to validate the white box approach. In other words the results obtained from the architectural analysis must be the same or closely match those of the black box approach. In the architectural analysis the middleware architecture is broken into layers and each layer is developed into an atomic Petri net model. Section 4.4 provides a summary of this chapter.

4.2 THE BLACK-BOX PETRI NET MODEL

4.2.1 THE EXPERIMENTAL SETUP AND THEORETICAL MODEL

The experimental environment consists of two hosts (client and server) remotely located from each other in a LAN. The host server on which JBoss AS is running is excluded from running other tasks to ensure the consistency of data sets collected. The client host generates service requests to the JBoss AS. The bandwidth of the LAN is shared with other users not relevant to this experiment. Tools like Wireshark [Wir] and Ping [Pin] are used to measure the round trip delay (RTD), excluding the time spent in the hosts. In comparison to the time spent in the server, the RTD is observed to be so minute that it is ignored in the experimental analyses. The system structure is illustrated in Figure 4.3.

![Figure 4.3 the experimental setup](image)

<table>
<thead>
<tr>
<th></th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OS</strong></td>
<td>Windows 7</td>
<td>Windows Vista</td>
</tr>
<tr>
<td><strong>Processor</strong></td>
<td>1.67 GHz</td>
<td>2.66 GHz</td>
</tr>
<tr>
<td><strong>RAM</strong></td>
<td>1.00 GB</td>
<td>2.00 GB</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>SoapUI (version 3.6.1)</td>
<td>JBoss AS (4.2.2)</td>
</tr>
</tbody>
</table>
SoapUI is installed on the client that generates service requests to the server. There are two main parameters in SoapUI load testing tool that can be set to control the workload of the application server: number of threads representing the virtual users (clients), and the number of requests (runs) generated per thread/users. For example, if the number of threads is set to 20 and number of runs per thread is set to 10, then there are 20 clients, each sending 10 SOAP-HTTP requests for a total of 200 requests. The notion of threads in Soapui are different from the threads in JBoss AS; the first one shows the number of clients connecting to the web service and the second one presents the actual threads that are initiated in the server to process the clients’ requests. As shown in Figure 4.4, SoapUI is capable of generating several performance parameters such as the average response time, $avg$, transactions per second, $tps$, and the number of transaction requests failed in the process, which is denoted as $err$. $Avg$ is the average time difference between the times a request is sent out until the response is received. $Tps$, also called arrival rate, is the average number of requests generated by the clients per second.

![Figure 4.4 graphical interface of SoapUI client](image)

Figure 4.4 graphical interface of SoapUI client
The experiments have been conducted with sixteen different load configurations. For each load test, SoapUI returns the values for \( \text{avg}, \ tps, \) and \( \text{err} \). Each test is repeated 5 different times and the average of the returned values are calculated. Hereafter, rather than creating new terms, the same terms, i.e. \( \text{avg}, \ tps, \) and \( \text{err} \) are used to represent the averages of the five runs. The total time \( T \) for each load test configuration is computed as the following:

\[
T = \frac{cnt}{tps} \tag{4.1}
\]

where \( cnt \) is the total number of requests for each test. Consequently, the error and success rates for each load are computed by:

\[
\text{error}_{rate} = \frac{\text{err}}{cnt} \tag{4.2}
\]

\[
\text{success}_{rate} = \frac{cnt - \text{err}}{cnt} = 1 - \text{error}_{rate} \tag{4.3}
\]

The error reports generated by SoapUI consist of all types of potential errors that are generated by the network, application server, and web service itself. However, this study considers only the errors that are generated by JBoss AS when the HTTP requests are rejected. These requests that are not satisfied are referred to as \( \text{HTTP rejections} \) or simply \( \text{rejections} \). Therefore, \( \text{error}_{rate} \) will be replaced with \( \text{rejection}_{rate} \).

The actual rate of requests rejected can be obtained from (4.2). This rate can be estimated in a different way. Recall that \( \text{avg} \) is the average response time for one request. Therefore, the service rate of JBoss AS is \( 1/\text{avg} \). In order for the JBoss AS not to reject any incoming request, it should be able to allocate a thread to each incoming request to keep up with the arrival rate. In other words, JBoss AS will reject no requests if the following holds:
\[
\frac{1}{\text{avg}} \times \text{threads} \geq t\text{ps} \Rightarrow \frac{1}{\text{avg}} \times \text{threads} - t\text{ps} \geq 0
\]

where \( \text{threads} \) is the total number of threads that can be allocated to requests in JBoss AS. Otherwise some requests will be rejected. That is, if sufficient number of threads cannot be allocated, rejections will occur. Therefore, number of requests rejected per unit of time will be:

\[
rejection_{\text{rate}} = (t\text{ps} - \frac{1}{\text{avg}} \times \text{threads})
\]

Consequently, the estimated total number of requests rejected is:

\[
rejection_{\text{est}} = rejection_{\text{rate}} \times T = (t\text{ps} - \frac{1}{\text{avg}} \times \text{threads}) \times T \tag{4.4}
\]

If \( rejection_{\text{rate}} \leq 0 \), no requests are rejected. The accuracy of the \( rejection_{\text{rate}} \) depends on the value of \( \text{threads} \), which will be referred to as \( \text{threshold} \). Thus, the \( \text{threshold} \) is the lower bound on the ultimate number of threads that can be allocated to ensure no rejection occurs. This is because any value higher than the threshold value underestimates and any value lower than the threshold overestimate the number of errors. Therefore, the following will be used instead of (4.4).

\[
rejection_{\text{est}} = (t\text{ps} - \frac{1}{\text{avg}} \times \text{threshold}) \times T \tag{4.5}
\]

\( \text{Threshold} \) is a number that is computed from the experimental analyses. This number is mostly dependent on the hardware power rather than software characteristics. More CPU
power and higher amount of memory are mostly enhancing the power of server and increasing the number of threads that a server can initiate to process the web service requests and consequently increasing the threshold number. There are two parameters in JBoss AS to control the number of initiated threads that process the web service requests. *MaxThread* is the number of actual threads that process each web service requests and *acceptCount* is the number of received requests in a queue. If the number of requests that are received by the application server is higher than *MaxThread* (default value is 250), then the next requests are stored in a queue (default value is 100). If the queue is full and more requests are coming, then the application server will reject the received requests.

Theoretically, in JBoss AS, when the *maxThread* and *acceptCount* configuration parameters in *server.xml* file are set to 250 and 100 respectively, it is expected that JBoss AS handle 350 requests. However, in real world experiments, there are many factors such as memory, processor and type of operating system that may affect the actual number of threads in JBoss AS and that is why a threshold value is set based on experimental test cases. More importantly, the response time evaluated by SoapUI includes the time spent in the queuing system and the time actually spent on servicing the request. With this experiment setup in the lab and by running different tests, *threshold* is estimated to be around 315. This means that 315 is the threshold number of threads in JBoss AS that does not provide any rejection in server side and provides the simulation results to closely match that of the performance results from the load tests.

It is expected that the response time, *avg*, to increase as the number of requests increases. Figure 4.5 shows the *avg* numbers for different number of users in the real experiments,
where each user sends 10 requests. The figure exhibits that the response time increases up to a point, levels-off for a moderate range of requests, and then increases again.

![Graph showing average response time vs number of virtual users]

Figure 4.5 average response time of the web service based on number of virtual users with default setting in JBOSS AS (MaxThread: 250 and AcceptCount: 100)

The reduction in response time beyond 370 up to around 500 users is counterintuitive because as there are more requests the response time ought to increase instead. During this period, Figure 4.6 shows that the rejection rate is increased exponentially. The reason for decrease of average response time is that SoapUI uses the total requests in calculating the average response time, regardless of whether a request is successful or rejected. The rejected requests have less average response time rather than successful requests and that is why the overall average response time is decreased in this area. It is for this reason that the average response time of 15.11 in Figure 4.5 can be interpreted as the true response time because at about 370 users the number of blocked requests is low or almost
nonexistent. After this point the average response time is not accurate because of increasing rejected requests by application server.

Since SoapUI includes the rejection count in the evaluation of average response time, one way to find a good estimate of the actual response time is to use Wireshark and evaluate the response time for each successful request from the time the request is received by the server until the response is sent back to SoapUI. This requires filtering the blocked requests and evaluating the response time for each individual successful request. This seems to be infeasible because of the large number of requests. The other approach is to use 15.11 as the estimate of the actual response time for loads more than 370, where the rejections really start. This value shows the peak of response time when the system utilizes all the resources for high level loads without being biased by the rejected requests. Therefore, in this study \( \text{avg} = 15.11 \) is used in simulation model as the closest approximation of average response time for high level loads.

![Figure 4.6 request rejection rate of the application server based on virtual users with default setting in JBOSS AS (MaxThread: 250 and AcceptCount: 100)](image)
4.2.2 The Simulation Model

As indicated before, Mobius will be used for the simulation purpose [Mob]. Mobius can solve SAN models, either mathematically or by simulation. Because of the types of reward rates used we have found it easier and less time consuming to work with the simulation solver. Figure 4.7 provides a Petri net model based on Mobius for the service requests that arrive at the server (JBoss AS) side. Recall that Duke’s Bank web service running on JBoss AS is the web service used in this experiment. This web service receives Customer ID and returns all Account ID of the customer.

Recall that the timed and immediate transitions are shown by thick and thin bars, respectively. Further recall that each flat dot at the output end of a transition indicates the probability of taking a different path to the rest of the model once the transition fires. The place called Request is initialized to cnt, the number of total HTTP requests for each load test experiment. The rate of the \( T_{arrival} \) transition gives the rate of arrivals per unit of time, which is equal to \( tps \). The HTTP thread instance pool in JBoss AS is represented by the ThreadPool place, which is initialized to \( maxThread \) extracted from the configuration file named server.xml. An output gate, \( OG \), shown as a solid triangle that leads to either Start or RejectedRequest represents the conditions for blocking of HTTP requests. For instance, if Start has reached its maximum capacity, \( OG \) will redirect the token to RejectedRequest; otherwise the token is added to Start. The rejected requests are accumulated in Down via the \( T_{03} \) transition activities. \( TJW \) is the service rate for the web services. Activity \( TJW \) is enabled only if ThreadPool and Start are not empty. When \( TJW \) fires, a token in ThreadPool representing an available threads in JBoss AS is allocated to
a request in \textit{Start}. Once the request is serviced, the thread, through $T01$ (case 1), is released to \textit{ThreadPool} to be used by a next request. $T01$ (case 2) is set to zero, but it can be set to a non-zero value due to any errors that may happen during response time.

Table 4.1 shows the parameters and their values used in the SAN model. In the table, the impulse reward functions count the number of activities fired. The number of activities fired at $T_{arrival}$ shows the total number of requests that has entered the Petri net model. Similarly, the number of activities fired at $T03$ represents the number of requests rejected. Let’s call the number of requests rejected as $rejection_{SAN}$, i.e. the number of $T03$ activities fired. $T02$ shows the number of requests failed due to reasons other than rejection by JBoss. These failures may happen due to some functional errors in the web service. In this example, the failure probability of these types of failures, i.e. $TJW$ probability case 2, is considered zero, but it can be set to a non-zero value. Note that the sum of the case probabilities on each activity must be equal to 1.

Since the threads in JBoss AS are executed in parallel, the $TJW$ rate in the table ensures that the requests are serviced in parallel based on the maximum number of threads allowed in JBoss AS, i.e. threshold. On the other hand, $OG$ presents the rejection condition. As it is shown in the Table 4.1, there can be at most $queue + threadpool$ tokens in \textit{Start}. However, in real-world case, if there are $x < threadpool$ tokens in \textit{ThreadPool}, the maximum number of tokens in \textit{Start} is $(queue + x)$. Since the value of $x$ changes depending on the number of available threads in each time, the maximum number of tokens in \textit{Start}, i.e. $(queue + x)$, continuously changes as well. This causes the value of
threshold, representing the speed at which the Petri net model services the requests to be dynamic. In other words, the value of threshold needs to be throttled each time \( x \) changes. This makes it difficult to predict an appropriate value for threshold that meets the rejection rate observed by the experiments performed using SoapUI. Consequently, the maximum value of Start is set at the fixed value (threadpool + queue). This in turn makes the value of threshold to be a fixed value. As it will be shown shortly, this approach has shown the performance of the Petri net model to be very close to that of the rejection rate reported by SoapUI.

![SAN model of JBoss AS serving the requests](image)

Figure 4.7 the SAN model of JBoss AS serving the requests

Around sixteen different performance tests have been conducted on the banking web service. Each test is repeated five times and the average of extracted data is calculated and considered as the experimental data. Table 4.2 shows a sample data extracted from SoapUI for some of the tests.
Table 4.1 primary SAN Model parameters

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Parameter value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests</td>
<td>Initialized to ( Cnt ) (from SoapUI)</td>
</tr>
<tr>
<td>TArrival rate</td>
<td>Initialized to ( Tps ) (from SoapUI)</td>
</tr>
<tr>
<td>ThreadPool</td>
<td>Initialized to ( maxThread ) (from server.xml in JBoss AS)</td>
</tr>
<tr>
<td>OG</td>
<td>( threadpool = maxThread ) &amp; ( queue = acceptCount ) from server.xml in JBoss AS ( maxThread ) default: 250 &amp; ( acceptCount ) default: 100</td>
</tr>
<tr>
<td></td>
<td>If ( \text{Start} \rightarrow \text{Mark}() &lt; (\text{threadpool} + \text{queue}) )</td>
</tr>
<tr>
<td></td>
<td>( \text{Start} \rightarrow \text{Mark}() = \text{Start} \rightarrow \text{Mark}() + 1;</td>
</tr>
<tr>
<td></td>
<td>( \text{Else} )</td>
</tr>
<tr>
<td></td>
<td>( \text{RejectedRequest} \rightarrow \text{Mark}() = \text{RejectedRequest} \rightarrow \text{Mark}() + 1;</td>
</tr>
<tr>
<td>TJW rate</td>
<td>( \text{avg} ) from SoapUI</td>
</tr>
<tr>
<td></td>
<td>( \text{threshold} = 315 )</td>
</tr>
<tr>
<td></td>
<td>If ( \text{Start} \rightarrow \text{Mark}(()) &lt; \text{threshold} ) Return ((1/\text{avg}) \times (\text{Start} \rightarrow \text{Mark}()));</td>
</tr>
<tr>
<td></td>
<td>( \text{Else} ) Return ((1/\text{avg}) \times \text{threshold});</td>
</tr>
<tr>
<td>T01 probability</td>
<td>1</td>
</tr>
<tr>
<td>case1</td>
<td></td>
</tr>
<tr>
<td>T01 probability</td>
<td>0</td>
</tr>
<tr>
<td>case2</td>
<td></td>
</tr>
<tr>
<td>TJW probability</td>
<td>1</td>
</tr>
<tr>
<td>case1</td>
<td></td>
</tr>
<tr>
<td>TJW probability</td>
<td>0</td>
</tr>
<tr>
<td>case2</td>
<td></td>
</tr>
<tr>
<td>Impulse Reward</td>
<td>Total number of requests: If TArrival fires then return 1;</td>
</tr>
<tr>
<td>Functions</td>
<td>Number of requests blocked: If T03 fires then return 1;</td>
</tr>
</tbody>
</table>
Table 4.2 sample data extracted from SoapUI client

<table>
<thead>
<tr>
<th>Users or threads</th>
<th>Runs per thread (virtual user)</th>
<th>Cnt</th>
<th>Avg (sec)</th>
<th>Tps</th>
<th>Total time (sec)</th>
<th>Number of requests rejected</th>
<th>Request rejection rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>10</td>
<td>2000</td>
<td>7.46</td>
<td>24.68</td>
<td>82.55</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>350</td>
<td>10</td>
<td>500</td>
<td>13.76</td>
<td>22.92</td>
<td>153.25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>352</td>
<td>10</td>
<td>3520</td>
<td>13.83</td>
<td>22.25</td>
<td>158.21</td>
<td>0.5</td>
<td>0</td>
</tr>
<tr>
<td>354</td>
<td>10</td>
<td>3540</td>
<td>13.69</td>
<td>22.61</td>
<td>156.58</td>
<td>5</td>
<td>0.001</td>
</tr>
<tr>
<td>360</td>
<td>10</td>
<td>3600</td>
<td>14.54</td>
<td>22.15</td>
<td>162.67</td>
<td>24.2</td>
<td>0.006</td>
</tr>
<tr>
<td>370</td>
<td>10</td>
<td>3700</td>
<td>15.11</td>
<td>21.78</td>
<td>170.46</td>
<td>185.4</td>
<td>0.05</td>
</tr>
<tr>
<td>380</td>
<td>10</td>
<td>3800</td>
<td>13.71</td>
<td>24.13</td>
<td>157.64</td>
<td>332</td>
<td>0.08</td>
</tr>
</tbody>
</table>

For several load tests, the values of avg and tps returned by SoapUI, shown in Table 4.2, are used in the SAN model of Figure 4.7. The only exception is that the avg values for the tests with higher than 380 users are considered 15.11 sec, as explained in section 4.2.1. As indicated, each test is repeated five times and the average of extracted data is calculated and considered as experimental data. Table 4.3 shows the results for all the tests obtained from SoapUI, the theoretical equation (4.5), and from running the SAN model. As it is shown in the table, the worst prediction of request rejection rate happens when there are 380 users, which is $0.13 - 0.08 = 0.05$, and $0.14 - 0.08 = 0.06$ for the theoretical and the SAN model, respectively. This probably happened because of imprecise avg value which is estimated based on SoupUI results.
Table 4.3 request rejection rate of the sixteen tests for the three models: SoapUI, theoretical, and the SAN model.

<table>
<thead>
<tr>
<th>Number of simultaneous users</th>
<th>Request rejection rate (SoapUI)</th>
<th>Request rejection rate (rejection\textsubscript{est} / cnt)</th>
<th>Request rejection rate (rejection\textsubscript{SAN} / cnt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>0</td>
<td>0</td>
<td>0.01</td>
</tr>
<tr>
<td>350</td>
<td>0</td>
<td>0</td>
<td>0.04</td>
</tr>
<tr>
<td>352</td>
<td>0</td>
<td>0</td>
<td>0.03</td>
</tr>
<tr>
<td>354</td>
<td>0.001</td>
<td>0</td>
<td>0.03</td>
</tr>
<tr>
<td>360</td>
<td>0.006</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>370</td>
<td>0.05</td>
<td>0.04</td>
<td>0.05</td>
</tr>
<tr>
<td>380</td>
<td>0.08</td>
<td>0.13</td>
<td>0.14</td>
</tr>
<tr>
<td>400</td>
<td>0.15</td>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td>500</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>600</td>
<td>0.52</td>
<td>0.5</td>
<td>0.49</td>
</tr>
<tr>
<td>650</td>
<td>0.54</td>
<td>0.52</td>
<td>0.52</td>
</tr>
<tr>
<td>700</td>
<td>0.51</td>
<td>0.52</td>
<td>0.52</td>
</tr>
<tr>
<td>800</td>
<td>0.5</td>
<td>0.54</td>
<td>0.53</td>
</tr>
<tr>
<td>1000</td>
<td>0.54</td>
<td>0.51</td>
<td>0.51</td>
</tr>
<tr>
<td>1200</td>
<td>0.55</td>
<td>0.52</td>
<td>0.52</td>
</tr>
<tr>
<td>1500</td>
<td>0.51</td>
<td>0.5</td>
<td>0.49</td>
</tr>
</tbody>
</table>

Figure 4.8 displays graphically the request rejection rate for the three different models shown in Table 4.3. The figure shows that the rejection rate of the theoretical and the SAN models closely match the ones provided by SoapUI.
The SAN model here have considered the entire environment as a black box and utilized the parameters extracted from the empirical testing, such as the average response time and arrival rate, in order to predict the rejection rate of HTTP requests by the application server. The next section utilizes the architectural information to analyze the web service environment. In other words, architectural-based analysis will be used.

4.3 ARCHITECTURAL ANALYSIS OF THE WEB SERVICE ENVIRONMENT

As presented in the previous chapter, the architectural analysis of a web service starts with the static analysis of the system on which the web service runs. The static analysis helps to extract the main components of a system. One of the main factors affecting the performance and reliability is the configuration parameters. The main concentration of this thesis is on two major thread pools, i.e. HTTP thread pool and database connection pool. The sensitivity of overall reliability of the web service to these configuration
parameters are studied in this section. To complete the architectural view of the system, dynamic analysis based on system execution is also needed. Code instrumentation is used to gather information related to time and dynamic dependencies among the software components of the system. Time and dynamic call graph extracted from code-profilers are used to construct the Petri Net model of the system. The two major tools used for code instrumentation are JRAT [JRA] and Javashot [JSH]. JRAT is used to gather the time spent in each function and consequently a software component/layer. Javashot is used to extract the dynamic call graph from run-time execution. The results extracted from JRAT and Javashot profilers are in text format. An output example is presented in Figure 4.9a and Figure 4.9b.

```
digraph
  org.apache.tomcat.util.net.JIOEndpoint$Worker{
    START->
  org.apache.tomcat.util.net.JIoEndpoint$Worker [label="1:run"]
  org.apache.tomcat.util.net.JIoEndpoint$Worker->
  org.apache.tomcat.util.net.JIoEndpoint$Worker [label="2:await"]
  org.apache.tomcat.util.net.JIoEndpoint$Worker->
  org.apache.tomcat.util.net.JIoEndpoint$Worker [label="3", style=dashed]
  org.apache.tomcat.util.net.JIoEndpoint$Worker->
  org.apache.tomcat.util.net.JIoEndpoint$Worker [label="4:setSocketOptions"]
  org.apache.tomcat.util.net.JIoEndpoint$Worker->
  org.apache.tomcat.util.net.JIoEndpoint$Worker [label="5", style=dashed]
  ...
  org.apache.tomcat.util.net.JIoEndpoint$Worker->[START [label="52386", style=dashed]]
```

Figure 4.9 output files from two different instrumentation tools used in the study
Although Figure 4.9 shows the extracted results from JRAT and Javashot, there are still some difficulties to dynamic analysis. The main issue is that the information extracted from system run-time is very complex to work with. The second difficulty is that the full coverage of the source code cannot be guaranteed in a complex system, and thus the call graph is mostly based on user interactions. To deal with these limitations and the challenges of working with complex and large amount of information extracted from instrumentation, an abstraction model is developed and a visualization tool is used to present a better understanding of abstracted system. For this reason, the Javashot output is filtered by an intermediate Perl\textsuperscript{5} program which transforms and condenses the output to a more readable format that shows the major layers and component calls, so that a more abstract and manageable call graph can be produced as shown in Figure 4.10.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure410.png}
\caption{a) third layer of abstraction}
\end{figure}

\begin{footnote}
\textsuperscript{5} A copy of the Perl program is provided in the Appendix A.
\end{footnote}
b) second layer of abstraction

Figure 4.10 dynamic call graphs extracted from Javashot profiler

Figure 4.10 shows two different abstraction models extracted from the Javashot profiler. These two graphs are built using ZGRViewer [ZGR] package that takes as input a text-based graph language, i.e. DOT language. Figure 4.10a shows the components of the software in the third layer of abstraction. To better understand the dynamic call graph and decrease the level of complexity, a second level abstraction is built using Javashot output. Figure 4.10b presents this simplified dynamic call graph that contains probability transition between components (layers). These probabilities have been calculated by enumerating the number of times a particular transition from a layer A has transferred to a layer B, divided by all the number of times that layer A transferred control to other layers. On the other hand, since this web service does provide a limited amount of functionality, a complete coverage of the source code can be guaranteed by calling all those services provided in the web service using test cases.
As presented in Figure 4.10.b, four different layers are extracted from the primary complex call graph. *Org.apache* is an open source implementation that contains the Java Servlet technology [Apa]. This technology is a Java component used to extend the capabilities of servers that host applications access via a request-response programming model [Ser]. A servlet container such as *Org.apache.tomcat* (also known as web container) which is a part of this layer is a component of the web server that interacts with the servlets [Wco]. *Org.jboss* is the main component of the application server. This layer is responsible for all the interactions between the deployed applications and other layers such as the web server and database. Logging, sending messages between components, and supporting secure transactions are other responsibilities of this layer. *Org.hsqldb* is an open source implementation of HSQL database. This database is implemented in Java and added as an internal package to JBoss AS. *Com.sun.ebank* is the Duke’s Bank web service layer. This layer contains all the packages and classes implemented to support a small banking business.

As shown in Figure 4.9a, JRAT is able to show the time spent in each method excluding the time spent in the (dependent) called methods. Therefore, another Perl program is written that adds up the time spent in each major layer/component that was obtained by the previous Perl program.
Table 4.4 the timing information extracted from JRAT (sequence extracted from Javashot)

<table>
<thead>
<tr>
<th>Layer name</th>
<th>Average time (sec)</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.org.apache</td>
<td>0.049</td>
<td></td>
</tr>
<tr>
<td>2.org.jboss</td>
<td>0.06</td>
<td>Webserver-JBoss-Webservice</td>
</tr>
<tr>
<td>3.org.jboss</td>
<td>0.007</td>
<td></td>
</tr>
<tr>
<td>4. com.sun.ebank</td>
<td>0.0075</td>
<td></td>
</tr>
<tr>
<td>5.org.jboss</td>
<td>0.0086</td>
<td></td>
</tr>
<tr>
<td>6.com.sun.ebank</td>
<td>0.0075</td>
<td></td>
</tr>
<tr>
<td>7.org.jboss</td>
<td>0.0283</td>
<td></td>
</tr>
<tr>
<td>8.org.hsqldb</td>
<td>0.0003</td>
<td></td>
</tr>
<tr>
<td>9.org.jboss</td>
<td>0.0005</td>
<td>JBoss-Database</td>
</tr>
<tr>
<td>10.org.hsqldb</td>
<td>0.0013</td>
<td></td>
</tr>
<tr>
<td>11.org.jboss</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>12.org.hsqldb</td>
<td>0.0022</td>
<td></td>
</tr>
<tr>
<td>13.org.jboss</td>
<td>0.0039</td>
<td></td>
</tr>
<tr>
<td>14.org.hsqldb</td>
<td>0.0013</td>
<td></td>
</tr>
<tr>
<td>15.org.jboss</td>
<td>0.01003</td>
<td></td>
</tr>
<tr>
<td>16.org.hsqldb</td>
<td>0.0019</td>
<td></td>
</tr>
<tr>
<td>17.org.jboss</td>
<td>0.0382</td>
<td></td>
</tr>
<tr>
<td>18.org.hsqldb</td>
<td>0.0001</td>
<td></td>
</tr>
<tr>
<td>19.org.jboss</td>
<td>0.062</td>
<td></td>
</tr>
<tr>
<td>20.org.apache</td>
<td>0.00</td>
<td>Webserver-Response</td>
</tr>
</tbody>
</table>
Table 4.4 presents the time information computed from JRAT output. This table displays the actual sequence of calls when a request is sent from a web service client to the server. As presented in Table 4.4, the call sequence is partitioned into three parts, i.e. Webserver-JBoss-Webservice, JBoss-Database, and Webserver-Response. Each part represents a partial view of call graph. Once the major layer sequence calls and their timing are obtained, they are transformed to atomic Petri net models and connected according to the interactions among layers, as explained in the following section.

4.3.1 Petri Net Modeling

In order to make the simulation model manageable the entire Petri net model is partitioned into the following sub-models:

- **Request**: This sub-model handles the timing for request arrivals and checks whether there are sufficient resources for the requests to be processed further by the web server. This is a part that is responsible for thread initiation in server side.

- **Webserver-JBoss-Webservice**: This sub-model shows the interaction between Apache, JBoss and the Database layers.

- **JBoss-Database**: This sub-model handles the interaction between the database and the application server.
- **Webserver-Response**: This submodel returns the outcome of the service back to the web server, i.e. Apache, to be communicated to the client, and releases the resources to be used by future requests.

Looking at the logical organization of the sub-models, one can interconnect the sub-models to reach the following logical sequence of execution:

\[
Request\text{-}Webserver\text{-}JBoss\text{-}Webservice\text{-}JBoss\text{-}Database\text{-}JBoss\text{-}Webserver\text{-}Response
\]

This sequence of execution can be observed from Table 4.4. During the presentation of sub-models, at times references are made to the conditional statements and the global variable. Therefore, Table 4.5 lists the initial values and the statements used in the overall simulation model. These values, which also include the transition rates, will be explained later, but in general most of these values are extracted from Figures 4.10 and Table 4.4.
### Table 4.5 parameters set for Petri net model

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Parameter value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarrival rate</td>
<td>21~46</td>
</tr>
<tr>
<td>TJBoss0 probability case1</td>
<td>0.23</td>
</tr>
<tr>
<td>TJBoss0 probability case2</td>
<td>0.77</td>
</tr>
<tr>
<td>TJBoss0 probability case3</td>
<td>0.0</td>
</tr>
<tr>
<td>TJBoss1 probability case1</td>
<td>0.64</td>
</tr>
<tr>
<td>TJBoss1 probability case2</td>
<td>0.36</td>
</tr>
<tr>
<td>TJBoss1 probability case3</td>
<td>0.0</td>
</tr>
<tr>
<td>TJBoss2 probability case1</td>
<td>0.13</td>
</tr>
<tr>
<td>TJBoss2 probability case2</td>
<td>0.87</td>
</tr>
<tr>
<td>TJBoss2 probability case3</td>
<td>0.00</td>
</tr>
<tr>
<td>TWebservice probability case1</td>
<td>0.99</td>
</tr>
<tr>
<td>TWebservice probability case2</td>
<td>0.01</td>
</tr>
<tr>
<td>TApache rate</td>
<td>( \text{StartThread} \rightarrow \text{Mark()} \times (1/( \text{Avg of Apache1})) = \text{StartThread} \rightarrow \text{Mark()} \times 20.4 )</td>
</tr>
<tr>
<td>TJBoss0 rate</td>
<td>( P01 \rightarrow \text{Mark()} \times (1/(\text{Avg of JBoss0})) = \text{P01} \rightarrow \text{Mark()} \times 16.66 )</td>
</tr>
<tr>
<td>TJBoss1 rate</td>
<td>( P02 \rightarrow \text{Mark()} \times (1/(\text{Avg of JBoss1})) = \text{P02} \rightarrow \text{Mark()} \times 22.7 )</td>
</tr>
<tr>
<td>TWebservice rate</td>
<td>( P03 \rightarrow \text{Mark()} \times (1/(\text{Avg of TWebservice})) = \text{P03} \rightarrow \text{Mark()} \times 66.66 )</td>
</tr>
<tr>
<td>TDB</td>
<td>( P04 \rightarrow \text{Mark()} \times (1/(\text{Avg of TDB})) = \text{P04} \rightarrow \text{Mark()} \times 140.84 )</td>
</tr>
<tr>
<td>TJBoss2</td>
<td>( \text{P05} \rightarrow \text{Mark()} \times (1/(\text{Avg of TJBoss2})) = \text{P05} \rightarrow \text{Mark()} \times 8.64 )</td>
</tr>
<tr>
<td>threshold</td>
<td>315</td>
</tr>
</tbody>
</table>
The following describes the submodels:

**The Request Sub-model** - Figure 4.11 provides the first sub-model when service requests made by the Duke’s Bank web service client arrive at the server side. The sub-model shows whether an arriving request should be accepted by the application server or be rejected due to the lack of resources (threads). In JBOSS AS, Apache Tomcat is the default web server. The major configuration parameters are the number of threads in the threadpool and database connection instance pool. These resources can be configured by

<table>
<thead>
<tr>
<th>Submodel</th>
<th>Description</th>
</tr>
</thead>
</table>
| Twebserver | // Jrattime is the overall average time computed by JRAT that is 0.29 sec  
// Soapuiavg is the avg time computed from Soupui (Table 4.2)  
If (Start→Mark() < threshold)  
    Return ((1/(Soapuiavg-Jrattime)) * (Start→Mark()))  
Else  
    Return ((1/(Soapuiavg-Jrattime)) * (threshold)) |
| Initial Mark(DBConnection) | 20 |
| Initial Mark(Threadpool) | 250 |
| OG1 | // default values: queue = 100, threadpool = 250  
If(Start→Mark()) <= (queue+threadpool)  
    Start→Mark() = Start→Mark()+1  
Else  
    RejectedRequest→Mark() = RejectedRequest→Mark()+1 |
| OG2 | // default value: dbthreads = 20  
If (DBStart→Mark() <= dbthreads)  
    DBStart→Mark() = DBStart→Mark()+1  
Else  
    DBRejected→Mark() = DBRejected→Mark()+1 |

Initial Mark(DBConnection) = 20  
Initial Mark(Threadpool) = 250  
OG1 // default values: queue = 100, threadpool = 250  
If(Start→Mark()) <= (queue+threadpool)  
    Start→Mark() = Start→Mark()+1  
Else  
    RejectedRequest→Mark() = RejectedRequest→Mark()+1  
OG2 // default value: dbthreads = 20  
If (DBStart→Mark() <= dbthreads)  
    DBStart→Mark() = DBStart→Mark()+1  
Else  
    DBRejected→Mark() = DBRejected→Mark()+1
the administrator using server.xml and hsqldb-ds.xml files, respectively. Tarrival gives the rate of arrivals per unit of time. Transition Twebserver represents the acceptance rate of the web server. The output gate OGI prevents the queue overflow, so the requests that cannot be queued are simply rejected and guided to the rejected place, i.e. RejectedRequest. Therefore, if a request can make it to the Start place, it will be serviced by the web server, although it might encounter some delay due to the sharing of the threads in ThreadPool. Once accepted, the token enters StartThread to be serviced by the web server, i.e. Apache, which is the starting point of the Webserver-JBoss-Webservice subsequence of operations. It should be observed that this submodel is mostly similar to Figure 4.7, with the difference that TJW in Figure 4.7 is the rate at which the requests are serviced. This rate is based on the total time from the points the requests are made till the time the responses are available, whereas Twebserver in Figure 4.11 shows a portion of this rate associated with the web server rate only. In other words, TJW is partitioned in the hierarchical model.

Figure 4.11 the SAN model for requesting HTTP threads in the application server
The Webserver-JBoss-Webservice Sub-model - After the thread allocation phase, the Apache layer is the first layer that serves the HTTP requests, i.e. the web server requests. Based on Figure 4.10b, Figure 4.12 is a Petri net sub-model that shows the execution sequence Webserver-JBoss-Webservice. This figure displays a StartThread node which is the starting point to process a request. The StartThread node is connected to the web server. As shown in Figure 4.10b, the org.apache communicates with org.jboss and org.jboss in turn communicates with com.sun.bank (web service layer) node. Note that communication over each link may happen multiple times.

In the Figure 4.12, TApache is the service rate of Apache. In this model, JBoss layer is divided to two timed transitions in order to build the Petri net easier and understandable. Each timed transition, i.e. JBoss0 and JBoss1, represents different packages of JBoss AS that communicates with Apache web server and Duke’s Bank web service respectively. As shown in the model, a request waiting in StartThread passes through Apache and JBoss with some probability of being re-serviced (loops). If no failure occurs (Case 1 and 2 of TJBoss0), the request will finally be handed over to the next layer of execution by having a token deposited into P02. During the execution, two possible forms of failures are considered. The web service itself could be the cause of a failure; in which case a token is deposited in WebServiceFailed, or the application server might encounter a logic fault leading to a failure of the server. In case of a JBoss failure (Case 3 of TJBoss0 and TJBoss1), a token will be placed in JBossFailed and this enables T100 to be fired that deposits a token in PartialFailure. Note that when a transition with probability cases is fired, only one of the cases will be followed. For example, once Twebservice fires, one of
the two cases is taken, i.e. either the execution is successful (case 1) or the execution fails (case 2).

As the probability of success is expected to be much higher than failure probability, a request will finally be deposited to ToDBConnect through case 2 of TJBoss1. ToDBConnect is the connection point to the JBoss-Database subsequence modeled in Figure 4.13. As shown in the figures, this place is shared between the two sub-models in Figure 4.12 and Figure 4.13.

![Diagram](image)

Figure 4.12 the SAN mode showing the interaction between Apache, JBoss and Web service layers

**The JBoss-Database Sub-model** - The sub-model in Figure 4.13 is the next step of simulation that captures the interaction between the JBoss layer and the Database layer, i.e. Hsqldb.
Once a request is ready to be serviced by the database layer, i.e. when \textit{ToDBConnect} is not empty, the output gate \textit{OG2} checks to ensure a connection can be made to the database. This decision is based on the global variable \textit{dbthreads}, which is the maximum number of instances for database connections (see Table 4.5). If a database connection cannot be made, it will be rejected by depositing a token in \textit{DBRejected}. Once rejected, the request’s thread that was allocated from \textit{ThreadPool} (see Figure 4.11) will be returned to \textit{ThreadPool}. This token, through activity \textit{TDBdown}, will be deposited into \textit{DBReturn}, which is one of the connection points to the next sub-model that finally deposits the thread into \textit{ThreadPool}. One can easily see that places \textit{DBRejected} and \textit{DBReturn} can be merged into one place. However, using two different places, one can use the number of times activity \textit{TDBdown} is fired to count the number of requests failed due to lack of database resources.

On the other hand, if the request is transferred to \textit{DBStart}, i.e. there is an available database instance in \textit{DBConnection}, the activity \textit{TDBStart} will fire. When the request
reaches $P04$, it is ready to use the database. Thus, $TDB$ is the service rate of the database. Once the service is performed, the request will be transferred to $P05$, indicating that it is ready to go back to JBoss, through activity $TJBoss2$.

Once activity $TJBoss2$ fires, the token produced can take three different directions based on three different probabilities. The first probability is the probability of coming back to the database (case 1). The second probability is the probability of entering the next step of finishing interaction with the database (case 2). The third probability is the probability of JBoss failure. The JBoss failures considered in this model are the ones that do not cause a crash in the system. Therefore the model is designed to return the database instance to $DBConnection$. If the request reaches $P06$, indicating a success, the database resource allocated previously is also returned to $DBConnection$. Furthermore, the thread allocated from $ThreadPool$ (see Figure 4.11) will be returned by placing a token in $MainReturn$. $MainReturn$ is another place shared with the next sub-model in Figure 4.14.

**The Webserver-Response Sub-model** - This phase simply returns the resources to the HTTP thread pool, i.e. returning resources to $ThreadPool$, to be used in the future requests. Note that the place $ThreadPool$ in Figure 4.14 is shared with $ThreadPool$ in Figure 4.11. In the figure, any titles for places can be used. However for simplicity, the titles used are the same as those in the previous models. Recall that $MainReturn$ represents the successful return of a request. $DBReturn$, $WebServiceFailed$, and $PartialFailure$ are three other paths that show the directions of the program in case of a request failure happening in the database, web service, and JBoss AS, respectively. A
keen reader observes that these places can simply be shared with *ThreadPool*, so that this submodel would not be needed. However, having this submodel provides for more readability, and the fact that the transitions in this submodel can be used to count the number of failures for each of web server, web service, database, and JBoss.

![Diagram of SAN sub-model](image)

*Figure 4.14 the SAN sub-model for returning the response to Apache*

The last step of forming a hierarchical model is to create a composed model that includes all these atomic submodels. This is done through the “composed” command of Mobius, by informing it which submodels to be included in the composed model. During the composition process the relationship among the atomic models are defined by creating the shared states. For example, *MainReturn* in Figure 4.13 is shared with *ThreadPool* in Figure 4.11.
To see the overall architecture, Figure 4.15 shows all the submodels and their relationships. The dashed lines show where the places in the submodels are connected (shared). Note that the dashed lines are not a part of the SAN models, but are drawn in this figure for easier understanding of the hierarchical modeling of the web service system.

Figure 4.15 the hierarchical model of the overall web service system
4.4 RESULTS

In this section, results computed from the experimental (actual) model and the SAN simulation model are presented and compared. Figure 4.16 displays the effect of arrival rate on rejection rate of requests. Figure 4.16a shows the results of actual tests in the lab and 4.16b displays the trend of predicted failure rate by SAN under various arrival rates up to 50. Both graphs show that with increasing the arrival rate to the server, the HTTP rejection rate will be increased.

a) experimental HTTP request failure rate (Soapui)  
b) SAN-based HTTP request failure rate (Mobius)

Figure 4.16 actual and SAN based rejection rate of Duke’s bank web service (based on JBoss As default setting in Table 4.5)
Figure 4.17 displays the increasing trend of rejection rate based on the number of simultaneous users. Recall that SoapUI initiates a thread for each user in the client side and each user sends ten requests. As it is shown in the figure, there are some deviations in the SAN prediction from the actual results. The first deviation is around 380 users that the SAN model underestimates the real failure data, and this trend of underestimation continues as the number of requests increases. The discrepancy can be clearly seen from the last two columns of Tables 4.6 when the number of total users in the test is higher than 380, but the average response time (column 2) for each request tends to decrease.
Two reasons can be explained for this anomaly. The first reason is that SoapUI uses the total requests in calculating the average response time, regardless of whether the request is successful or rejected. It is for this reason that the average response time is decreased when the actual rejection rate is increased around 380 users. Therefore, average time of

<table>
<thead>
<tr>
<th>Number of Total Users</th>
<th>Average Time (s)</th>
<th>Tps (Arrival rate)</th>
<th>Actual failure rate</th>
<th>SAN predicted failure rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>7.46</td>
<td>24.68</td>
<td>0.0</td>
<td>0.013</td>
</tr>
<tr>
<td>350</td>
<td>13.76</td>
<td>22.92</td>
<td>0.0</td>
<td>0.037</td>
</tr>
<tr>
<td>352</td>
<td>13.83</td>
<td>22.25</td>
<td>0.0</td>
<td>0.030</td>
</tr>
<tr>
<td>354</td>
<td>13.69</td>
<td>22.61</td>
<td>0.001</td>
<td>0.030</td>
</tr>
<tr>
<td>360</td>
<td>14.54</td>
<td>22.15</td>
<td>0.006</td>
<td>0.044</td>
</tr>
<tr>
<td>370</td>
<td>15.11</td>
<td>21.78</td>
<td>0.05</td>
<td>0.055</td>
</tr>
<tr>
<td>380</td>
<td>13.71</td>
<td>24.13</td>
<td>0.087</td>
<td>0.056</td>
</tr>
<tr>
<td>400</td>
<td>13.89</td>
<td>25.13</td>
<td>0.156</td>
<td>0.092</td>
</tr>
<tr>
<td>500</td>
<td>10.70</td>
<td>35.186</td>
<td>0.41</td>
<td>0.148</td>
</tr>
<tr>
<td>600</td>
<td>10.997</td>
<td>41.902</td>
<td>0.522</td>
<td>0.300</td>
</tr>
<tr>
<td>650</td>
<td>11.38</td>
<td>43.97</td>
<td>0.544</td>
<td>0.355</td>
</tr>
<tr>
<td>700</td>
<td>12.1</td>
<td>44.014</td>
<td>0.512</td>
<td>0.395</td>
</tr>
<tr>
<td>800</td>
<td>14.46</td>
<td>45.284</td>
<td>0.506</td>
<td>0.508</td>
</tr>
<tr>
<td>1000</td>
<td>19.77</td>
<td>42.94</td>
<td>0.540</td>
<td>0.624</td>
</tr>
<tr>
<td>1200</td>
<td>23.34</td>
<td>44.22</td>
<td>0.552</td>
<td>0.690</td>
</tr>
<tr>
<td>1500</td>
<td>31.75</td>
<td>41.75</td>
<td>0.515</td>
<td>0.759</td>
</tr>
</tbody>
</table>
15.11 seconds in load test of 370 users can be interpreted as the true response time. This is because up to this point of requests the number of rejected requests is low or almost nonexistent.

The second reason can be contributed to JBoss AS. From 370 to around 500 users, the server is using the resources that are already setup and activated, so the system can reallocate them to other requests, such as the thread resources. Beyond 500 users, the server overhead, such as the time taken to reject requests, accumulates as the rejection rate increases. To verify that 15.11 seconds is the closest estimation to actual average response time, several simulation tests are made. In these new tests, 380, 400, 500, 600, 650, 700, 800, 1000, 1200, 1500 users are selected and their average times in the simulation model are replaced by the number 15.11. The new outcomes from the simulation modeling are presented in Table 4.7 and a new graph comparing the simulation and the SoapUI model is presented in Figure 4.18. The simulation model results presented in Table 4.7 and Figure 4.18 are obtained based on SAN parameter values set in Table 4.5. The only exception is that the Soapuiavg is altered to 15.11 seconds for load tests higher than 380 users.

As it is shown in Table 4.7 and Figure 4.18, the deviation values between the simulation model and the actual numbers with this adjustment in average time are very low. Recall that the same adjustment is made in section 4.2.2 for the black-box modeling of the system. In both cases, it is shown that SoapUI estimated average time needs to be
carefully validated in high loads. This validation can be done using other tools such as Wireshark.

Table 4.7 test results from SoapUI and SAN models after adjustment in the average response time

<table>
<thead>
<tr>
<th>Number of Total Users</th>
<th>Average Time (s)</th>
<th>Tps (Arrival rate)</th>
<th>Actual failure rate</th>
<th>SAN predicted failure rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>7.46</td>
<td>24.68</td>
<td>0.0</td>
<td>0.013</td>
</tr>
<tr>
<td>350</td>
<td>13.76</td>
<td>22.92</td>
<td>0.0</td>
<td>0.037</td>
</tr>
<tr>
<td>352</td>
<td>13.83</td>
<td>22.25</td>
<td>0.0</td>
<td>0.030</td>
</tr>
<tr>
<td>354</td>
<td>13.69</td>
<td>22.61</td>
<td>0.001</td>
<td>0.030</td>
</tr>
<tr>
<td>360</td>
<td>14.54</td>
<td>22.15</td>
<td>0.006</td>
<td>0.044</td>
</tr>
<tr>
<td>370</td>
<td>15.11</td>
<td>21.78</td>
<td>0.05</td>
<td>0.055</td>
</tr>
<tr>
<td>380</td>
<td>15.11</td>
<td>24.13</td>
<td>0.087</td>
<td>0.13</td>
</tr>
<tr>
<td>400</td>
<td>15.11</td>
<td>25.13</td>
<td>0.156</td>
<td>0.162</td>
</tr>
<tr>
<td>500</td>
<td>15.11</td>
<td>35.186</td>
<td>0.41</td>
<td>0.397</td>
</tr>
<tr>
<td>600</td>
<td>15.11</td>
<td>41.902</td>
<td>0.522</td>
<td>0.494</td>
</tr>
<tr>
<td>650</td>
<td>15.11</td>
<td>43.97</td>
<td>0.544</td>
<td>0.514</td>
</tr>
<tr>
<td>700</td>
<td>15.11</td>
<td>44.014</td>
<td>0.512</td>
<td>0.518</td>
</tr>
<tr>
<td>800</td>
<td>15.11</td>
<td>45.284</td>
<td>0.506</td>
<td>0.530</td>
</tr>
<tr>
<td>1000</td>
<td>15.11</td>
<td>42.94</td>
<td>0.540</td>
<td>0.505</td>
</tr>
<tr>
<td>1200</td>
<td>15.11</td>
<td>44.22</td>
<td>0.552</td>
<td>0.519</td>
</tr>
<tr>
<td>1500</td>
<td>15.11</td>
<td>41.75</td>
<td>0.515</td>
<td>0.490</td>
</tr>
</tbody>
</table>
In the third experiment, Figure 4.19 shows the rejection rate of the database in the SAN simulation model due to varying the maximum size of the database instance pool. This rejection rate is computed by dividing the number of requests rejected by the database divided by the total number of requests sent to the database layer. Figure 4.19 presents the rejection rate under various loads of 200, 400, 700, and 1200 users. The rest of the parameters are stayed fixed as listed in Table 4.5. The figure demonstrates that if the maximum number of database connection instances is less than a certain number, reliability is reduced drastically. As the figure shows, a lower bound in the range of 6 – 8 is a good choice. Consequently, reliability of the system improves as the size of the connection instance pool in the database increases.
In another experiment, web service error rate is computed based on the number of simultaneous users in SAN. As presented in Table 4.5, the web service error probability is set to 0.01. Figure 4.20 displays that no variation of the web service error rate is detected when the error probability of web service is set to a fixed number, i.e. 0.01. The figure further shows that the HTTP rejection rate is also untouched when compared to Figure 4.18. This error rate is computed by dividing the number of web service requests failed by the total number of requests entering the web service layer. The number of failures can be determined by counting the number of fires in the activity TWSFailed of Figure 4.14.
In the next experiment, Figure 4.21 shows the error rate of JBoss AS which is computed in the SAN model. This SAN model uses the same parameter values from Table 4.5. The only exceptions are related to the three JBoss layer transition probabilities, which are presented in Table 4.8. These transitions are for the submodels in Figures 4.12 and 4.13. The JBoss AS error probability is presented by $T_{JBoss0}$ probability case3, $T_{JBoss1}$ probability case3, and $T_{JBoss2}$ probability case3 in Table 4.8. In the experimental analysis of this study, there were no actual JBoss errors seen in the test cases. But this number is considered 0.01 in order to investigate about how JBoss error probability can affect the overall failure rate.

Figure 4.20 web service error rate based on the number of clients (using the SAN model) when the web service failure probability is set at 0.01
Table 4.8 JBoss case probabilities for the three activities TJBoss0, TJBoss1, and TJBoss2 in the submodels of Figures 4.12 and 4.13

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Parameter value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TJBoss0 probability case1</td>
<td>0.225</td>
</tr>
<tr>
<td>TJBoss0 probability case2</td>
<td>0.765</td>
</tr>
<tr>
<td>TJBoss0 probability case3</td>
<td>0.01</td>
</tr>
<tr>
<td>TJBoss1 probability case1</td>
<td>0.635</td>
</tr>
<tr>
<td>TJBoss1 probability case2</td>
<td>0.355</td>
</tr>
<tr>
<td>TJBoss1 probability case3</td>
<td>0.01</td>
</tr>
<tr>
<td>TJBoss2 probability case1</td>
<td>0.125</td>
</tr>
<tr>
<td>TJBoss2 probability case2</td>
<td>0.865</td>
</tr>
<tr>
<td>TJBoss2 probability case3</td>
<td>0.001</td>
</tr>
</tbody>
</table>

As displayed in Figure 4.21, the error rate of JBoss AS is stable and equal to 0.05 where the HTTP rejection rate is increasing due to the increase of web service clients in the SAN model. These numbers are computed from dividing of number of JBoss error due to transition probabilities presented in Table 4.8 (the number of fires in the activity TJBossFailed of Figure 4.14) by number of total requests received in JBoss layer.
In the next experiment, Figure 4.22, the trend of database rejection rate is shown when the service rate of the database layer (DB rate) is manually changed in the SAN model. The other parameters of SAN are based in Table 4.5. Three different database service rates are considered in this experiment. The first rate is a normal rate based on the actual tests in the LAN environment, i.e. when the service rate is around 140 requests per second. This shows that the database rejection rate is zero. When the response time of the database is higher, i.e. when the service rate is reduced to 2, less than 5% of the requests are rejected. But at a rate of 1, i.e. one request per second, on average 30% of the requests are rejected. This shows that, as expected, with decreasing the service rate of the database, the rejection rate tends to increase.

Figure 4.21 JBoss AS error rate and HTTP rejection rate based on the number of users (using the SAN model)
The overall reliability of a system is defined as [Sho02]:

\[ R = e^{-\int z(t) dt} \]

where \( z(t) \) is the instantaneous rate of failures. It is this function that really defines the reliability model. The function can be defined in various ways. For example, it could be a rate proportional to the number of faults in the system, a rate that varies depending on time, or could be a constant. In a number of practical studies this function represents a constant failure \( \lambda \), i.e. \( z(t) = \lambda \). This reduces the reliability function \( R \) to:

\[ R = e^{-\lambda t} \]  \hspace{1cm} (4.1)

The reliability function in (4.1) presents the probability of an error free operation during \([0, t]\). With regard to this study, \( \lambda \) is the rejection rate of HTTP requests. Furthermore, \( R \) is the probability that no failures of requests occur during the time interval \([0, t]\). Recall
that web service reliability (section 1.3 of chapter 1) has been defined as the probability the web service “successfully” responds within a reasonable period of time; otherwise the web service is assumed to have failed. So the HTTP rejection of requests is treated as the failures since the requests are not successful.

Figure 4.23 presents the reliability graph based on the actual HTTP rejection rate extracted from load tests using the SoapUI client. Each line in the graph shows reliability of the system under a load test with a specific number of virtual users who send the requests simultaneously. As shown in Figure 4.23, the probability of success is very high for 200 users or less. Also, there is a huge impact on reliability when the number of users is doubled from 200 to 400. The reliabilities for high loads tend to be similar.

![Figure 4.23 reliability graph of the system under different loads (SoapUI)](image-url)
Figure 4.24 presents the reliability graph under the same condition of Figure 4.23 computed in the SAN simulation model. This SAN model also uses the same parameter values presented in Table 4.5.

Figure 4.24 shows the same results of Figures 4.23 and Figure 4.24 for three different loads. The graph shows that the simulation model prediction is very close to the actual reliability estimation. However, there is a deviation on the first load with 200 users. This is because of the difference in the estimated rejection rate of 0.013 rather than the actual rejection rate of zero from SoapUI. These results should be expected because the rejection rates in the two models are very close to each other.
4.4 SUMMARY

Presenting a precise simulation modeling whose results match of those gathered the field study is a big challenge. Since the realistic parameters such as the average response time and the arrival rate are dependent on various factors such as network loads, speed of the hardware used, and software specifications, it is important to analyze the output results, validate them, and tune the results before using them as input in simulation models. One of the most important lessons learned in the simulation process of this study is the challenges of understanding the testing environment and the results that sometimes seem reasonable but yet are not a representation of real cases.

This chapter compared the field tests against the simulation model. We have been able to design a hierarchical Petri net model capable of producing results very close to those in
the experimental model. Two Petri net models have been designed in this study, a black-box analysis and a hierarchical model for the white-box analysis. Since the black-box approach was easier to develop and it assisted in the development of the hierarchical model. The rejection rates of the requests for these two models are shown to be very close to each other. For one interested in the architectural analysis of a middleware, the white-box SAN is more suitable since it is more flexible to various input parameters of interest. For example, the black-box approach (Figure 4.7) has no provision for determining the effect of the database on the overall rejection rate or reliability of the system, whereas such effects can be determined by setting the proper parameters in the hierarchical model of Figure 4.15. Furthermore, dealing with the hierarchical approach, one can easily understand the relationship among various components of the middleware.
CHAPTER 5

SUMMARY AND CONCLUSION
5.1 SUMMARY

This dissertation presents a new approach in architecture-based reliability analysis of service-based system. A service-based system is modeled using a layered architecture style and then it is mapped to a Petri net model (SAN) in order to estimate the overall reliability of the system under various conditions and scenarios. As the accuracy of reliability analysis is a reflection of the parameters involved in the evaluation study, the main contribution of this research study is the reliability modeling of the entire web service system that embodies the common components of web service, web server, application server, and database as well as major configuration parameters in the middleware. One of the main factors in reducing the web service reliability is the misconfiguration of parameters of the underlying middleware such as web server and application server.

This research complements the research efforts presented in [Cao03, Sou06, Wel01, and Xia10]. The same as [Cao03], the presented approach in this dissertation evaluates the response time and rejection rates of the server but unlike [Cao03] the presented approach is a white-box approach which concentrates on internal components of the system and their interactions. However this study is an architecture-based approach the same as what is presented in [Sou06], it covers all the layers involved in JBoss AS and it is not just focused on a particular part of the application server such as EJB container. Therefore it presents a more precise view of system reliability in overall. [Wel01] is also simulates the performance analysis of a web server in various conditions such as arrival rate & alternative configurations but it only focuses on simulation-based analysis and it lacks the
experimental validation of the presented results. [Xia10] also provides the statistical analysis of web server performances, but it is a black-box analysis that cannot analyze the architecture of web server itself.

Although failures originated from misconfiguration are one of the common types of failures in web services, there is not much attention paid to these failures in most of architecture-based software reliability approaches. It has been shown that by developing a model based on the multilayer approach and inclusion of the appropriate configuration parameters more accurate reliability analysis and prediction of web services is possible. In addition, this research has shown that indeed it is possible to create a simulation model whose performance and reliability analysis closely matches that of the field study. Once the simulation model was able to correctly map the field study results, it has been shown (Chapter 4) that reliability and performance measures can be predicted under different case studies that would otherwise be very difficult to measure in a filed study. The next section discusses some of the limitations and possible future works.

5.2 LIMITATIONS AND FUTURE WORK

There are several avenues to extend this research further. The following will discuss the limitations of the presented work as well as the possible future work.

a. The actual test cases on Duke’s Bank web service, which are used to extract the failure rate and transition probabilities between layers, have been executed in a LAN environment. Since the network delay in a LAN environment is lower than in a Wide area network, e.g. Internet, The effect of the network layer in the TCP/IP suit
protocols is not considered as a separate component in the presented model. The actual delay computed between the sample web service and its client is calculated with Ping [Pin]. The delay is around 1 to 3 milliseconds in our lab environment, which is negligible compared to the overall average response time. Therefore, it is of interest to investigate the effect of the network layer on the response time in non LAN environments. It is conjectured that this layer can be added as a separate layer in the hierarchical model.

b. As presented in Chapter 3, the actual failure probability of each layer or component is required in order to estimate the overall reliability of the system using the SAN model. The precise estimation of failure probability in each layer is one of the main challenges of architecture-based reliability modeling. In this study two types of failures in the layers are considered. The first type is related to failures that happen because of misconfiguration of the software. In the presented SAN model in Chapter 4, the HTTP rejection and database rejection rates are affected by this category of failures. The second types of failures originate from other parts of the software such as faults in the implementation of the methods or dependencies between classes. Estimation of these failure probabilities for each layer requires more investigation in the code and possibly it requires data mining approaches to extract the knowledge from the failure repository of each software product or its developers’ forums. Because data mining approaches are outside of the scope of the project, hypothetical failure probabilities of the layers are injected to the SAN model. Hence, one avenue of further investigation about web service reliability is to compute the average failure
probability of each underlying layer using code inspection or data mining approaches of the failure history.

c. Although failures occurring in the system prevent web services to finish successfully, none of the failures considered in this research were assumed to cause a total system failure. Hence our experimental test case did not require a reboot of the system that might even take different periods of time depending on the load of the host at the time of restart. In our experimental test cases, there was no case that causes the complete shutdown of the system. However, if a shutdown happens because of a malfunction in the software system, the SAN model would need to flush the requests in the pipeline and account for the time need to reboot the middleware.

d. The performance measures obtained from the hierarchical model have been based on transient times, i.e. the model has been run for various but fixed periods of time. An interesting avenue of research is to enhance the model to account for steady state measures, i.e. to measure the performance of the model under various conditions in the long run.

e. This research assumed that the web services used are atomic. However there are composite services that consist of atomic web services that are executed in a collaborative way and they might reside on the same or different servers. The servers may even belong to different service providers. Thus, a single atomic service may not be sufficient to address a complex service requirement. As a future research study, it
would be beneficial to investigate how the SAN model might be enhanced to account for these types of services in order to determine their effects on the overall performance of the web service system. Obviously, the addition of composite web services will provide the possibility of treating the services in either a black box or a white-box form. Adding the network layer, as indicated in one of the previous future items, will even make the SAN model more complex in that each atomic service of a composite one might be involved with different networking conditions such as round trip delays.

f. As indicated in items b and c, there can be different forms of failures. Understanding and estimating these failures are challenging tasks. For instance, failures can be caused by logical faults (accidental faults), intrusions, or even misconfiguration caused by human errors. This research did not distinguish between accidental and intentional failures caused by security breaches. However, if one wishes, the failure probabilities on JBoss and web service transitions can be further partitioned and possibly other submodels may be developed to investigate such failures in more detail.

Inclusion of all such failures may render the entire simulation model impractical. One feasible approach for future enhancement is to categorize the failures based on their impacts rather than their sources, in order to reduce the number of failure cases. For instance, Denial of Service (DoS) attack and a logic fault leading to a crash have the same impact in that both faults have the impact of total failure. Consequently, the same Petri net logic may be used to treat both cases.
BIBLIOGRAPHY


[Pha00] Hoang Pham, Software Reliability, Published by Springer-Verlag Singapore, 2000.


This Perl program filters the DOT files produced by Javashot (see Chapter 4). The program receives the number of a package level as input and removes all sequence of method calls made below that level.

use warnings;
use strict;

my $prereg = "";
my $preregtemp = "";
my $preline = "";
my $lineno = 1;
my @nodes = "";
my @package1 = "";
my @package2 = "";
my $callno = 1;

my $level = 2;

open FH, " > outputgraph.dot" or die " can't write on file: $!\n";

# Read all lines of the file received in command line

while (<>) {

  # check if the line has *.* pattern
  my $temp = $_;
  if ($temp =~ m/(.*?\.\*.*/) \./) {
    $preregtemp = $1;

    # check if there is a call across different layers
    if ($prereg ne $1) {
      
      # split the whole line to show the nodes (caller and callee)
      @nodes = split /\./, $preline;
      @nodes = split /\./, $preline;

      # split the nodes to have the packages
      if ($nodes[0]) {
        @package1 = split /\./, $nodes[0];
        @package1 = split /\./, $nodes[0];
      }
      if ($nodes[1]) {
        @package2 = split /\./, $nodes[1];
        @package2 = split /\./, $nodes[1];
      }
    }
  }
}
# print the packages of caller in the graph
my $count = $level;

while ($count > 0)
{
    if ($package1[$level-$count]) {
        print "eachlayer1: $package1[$level-$count] ";
        print FH "$package1[$level-$count]"
        if ($count != 1) {print FH "_";
    }
}
$count--;
# end of while

if ($nodes[0] & $nodes[1])
{
    print FH "->"
};
if (($nodes[0]) && ($nodes[1] eq "") && ($callno eq 1))
{
    print FH "\n"
};

# print the packages of callee in the graph
my $count = $level;

while ($count > 0)
{
    if ($package2[$level-$count]) {
        print "eachlayer2: $package2[$level-$count] ";
        print FH "$package2[$level-$count]"
        if ($count != 1) {print FH "_";
    }
}
$count--;
# end of while
if ($nodes[1]) {
    print FH "\[\text{label=\"$callno\"}\]";
    print FH "\n";
    $callno++;
}

# print whole line in stdout
print $preline;
print "\nlineno = $callno";
$preline=$temp;
$prereg= $preregtemp;

} # end of if ($prereg ne $1)

else {
    $preline = $temp;
}

} # end of if ($_ =~ m/(.*?\..*?)/)

$lineno++;
} # end of while
print FH "}";

# The purpose of this Perl program is to print out the time spent in each package level in
# milliseconds. It receives two input files from the command line. One file from JRAT profiler
# output that contains the time spent in each method and another file as the result of the previous
# program that removed the sequence of calls below a particular level.

%Jratmethodtime = ();
%Jshottime = ();

open(INPJrat, "<$ARGV[0]") or die("Cannot open file '$ARGV[0]' for reading\n");
open(INPJshot, "<$ARGV[1]") or die("Cannot open file '$ARGV[1]' for reading\n");

while ($line=<INPJrat>){
    chomp ($line);
    if ($line =~ /.*\./) {
        ($methodname, $time)= split (";", $line);
        if (exists $Jratmethodtime{$methodname}) {
            $Jratmethodtime{$methodname} = ($time+$Jratmethodtime{$methodname})/2;
        }
        %Jratmethodtime = ();
        %Jshottime = ();
        open(INPJrat, "<$ARGV[0]") or die("Cannot open file '$ARGV[0]' for reading\n");
        open(INPJshot, "<$ARGV[1]") or die("Cannot open file '$ARGV[1]' for reading\n");
        while ($line=<INPJrat>){
            chomp ($line);
            if ($line =~ /.*\./) {
                ($methodname, $time)= split (";", $line);
                if (exists $Jratmethodtime{$methodname}) {
                    $Jratmethodtime{$methodname} = ($time+$Jratmethodtime{$methodname})/2;
                }
else {
$Jratmethodtime{$methodname} = $time;
}
}# end of if
}# end of while

$var1 = "org.jboss";
$var2 = "org.apache";
$flag = 0;
$sum = 0;
$totaltime=0;
while ($line=<INPJshot>){
  chomp ($line);
  if ($line =~ m/(.*?..*?..?)/) {
    $layer = $1 . $2;
    if (($layer eq $var1) || ($layer eq $var2)) {
      if (exists $Jratmethodtime{$line}){
        $sum = $sum + $Jratmethodtime{$line};
        $totaltime=$totaltime+$Jratmethodtime{$line};
      }
    } else {
      if ($var2 eq "null"){
        $var2 = $layer;
      if (exists $Jratmethodtime{$line}){
        $sum = $sum + $Jratmethodtime{$line};
        $totaltime=$totaltime+$Jratmethodtime{$line};
      }
    } else {
      print " layer: $var1 - $var2   $sum \n";
      $var1 = $layer;
      $var2 = "null";
      $sum = 0;
      if (exists $Jratmethodtime{$line}){
        $sum = $sum + $Jratmethodtime{$line};
        $totaltime=$totaltime+$Jratmethodtime{$line};
      }
    }
  }
}
}
```perl
    }# end of if
    }# end of while

    print " layer: $var1 - $var2   $sum \n";
    print " totaltime: $totaltime \n";
```