Architectural Reliability Analysis of Framework-Intensive Applications:  
A Web Service Case Study

M. Rahmani, A. Azadmanesh, H. Siy

College of Information Science and Technology, University of Nebraska-Omaha, Omaha, Ne 68182, USA

Abstract

A novel methodology for modeling the reliability and performance of web services (WSs) is presented. To present the methodology, an experimental environment is developed in house, where WSs are treated as atomic entities but the underlying middleware is partitioned into layers. WSs are deployed in JBoss AS. Web service requests are generated to a remote middleware on which JBoss runs, and important performance parameters under various configurations are collected. In addition, a modularized simulation model in Petri net is developed from the architecture of the middleware and run-time behavior of the WSs. The results show that 1) the simulation model provides 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 substantial impact on the overall performance; 3) the simulation model provides a basis for aggregating the modules (layers), nullifying modules, or to include additional aspects of the WS architecture; and 4) the model is beneficial to predict the performance of WSs for those cases that are difficult to replicate in a field study.

Keywords: Architecture-based Software Reliability; Petri Net; Service Oriented Architecture; Web Service; Application Server

1. Introduction

Internet globalization has provided the unprecedented opportunity for enterprises to develop and deliver electronic services such as online shopping and banking services. These services, called web services (WSs) are becoming a promising technology for building distributed and complex systems. World Wide Web Consortium (W3C) defines a web service a software system designed to support interoperable machine-to-machine interaction over a network (Web, 2012).

WSs are commonly delivered through an infrastructure based on the Service-Oriented Architecture (SOA). WSs are deployed using an application server (AS), part of the software ecosystem comprising the middleware connecting service requesters to providers. The middleware is a complex and intertwined combination of various libraries, components and their interfaces, forming an application framework upon which web services can be deployed. Because the majority of code execution of a web service normally occurs within the components of this framework, web applications are a type of framework-intensive applications (Dufour et al., 2007). Due to the growing complexity of such framework-intensive applications as well as society’s increasing dependence on their services, the reliability of these systems has become critical. In this study, WSs are treated as atomic entities but the middleware is partitioned into layers. WSs are developed in house and the application server (AS) considered is JBoss AS (JBoss, 2012). An experimental model is developed that anchors the analytical results to empirical observations. In parallel, a simulation model is developed that provides an operational view of the behavior and inner working of the overall distributed system. The architecture of the simulation model follows a hierarchical structure, 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 to observe and investigate the performance of the entire system under various conditions. The architecture-based analysis, which takes into account the components, e.g. middleware layers, and their interactions, is a form of white-box analysis. In contrast, most approaches to WSs analyses are monolithic in that the entire system is treated as a black-box.

Although this study complements some of the research work in literature (Cao et al., 2003; Souza et al., 2006; Xia and Dohi, 2010; Wells et al., 2001), the contribution of this study can be summarized as follows:

- One cannot underestimate the importance of middleware in the overall reliability of the applications. This study conducts performance analysis of the middleware based on its major architectural layers. To the best of our knowledge there is no solid architecture-based reliability analysis of the middleware.
- A large number of studies consider theoretical analysis without investigating the real-world applications and tackling the challenges of complex software systems (Cheung, 1980; Singh et al., 2001; Zhong and Qi, 2006). 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. This study uses a combination of analytical, experimental, and simulation models.
- Configuration errors occur when operators define incorrect settings for system parameters, e.g., specifying insufficient number of threads to service user requests entering the web server. Configuration parameters play an important role in
the performance of web services, but not much attention has been made in the failure analysis of web services caused by these error types. Research in Pertet and Narasimhan (2005) has shown misconfiguration of web-based and framework-intensive software can contribute to nearly forty percent of failures in web applications. This study pays a close attention to these types of failures.

- The software system under study by a number of research papers is often simple and rudimentary (Cao et al., 2003; Sing et al., 2001; Souza et al. 2006; Wells et al., 2001; Zhong and Qi, 2006). In addition, the common model presentation used is Markov chains, which may lead to state explosion for large systems. Instead, this study presents an architectural model using a specific type of Petri Nets called Stochastic Activity Network (SAN) (Sanders and Meyer, 2001). The model is simple enough to build, which is based on the static and dynamic code analysis of the middleware. Static analysis is used to determine the main layers and configuration parameters of the middleware. Dynamic analysis assists in the extraction of the timing information of the layers and the call-graph of a particular web service execution.

The multilayer and the reliability approaches proposed in this study have the following advantages:

- The application server contains a large number of internal applications, utilities, and enormous number of Java classes. The proposed multilayer approach offers a more manageable model in order to analyze the reliability of a web service.
- As failure rate of each layer and configuration parameters are estimated separately, analyzing the reliability of the system would be more insightful to better understand the effect of each layer on the overall reliability of the system.
- The analytical, experimental and the simulation-based models can validate each other to ensure the correctness of the results. Hence, the simulation model can be used to test the performance of the web environment under various conditions.

The rest of the paper is organized as follows. Section 2 provides some background on web service reliability and prior approaches to reliability modeling. Section 3 describes the three proposed models. Section 4 provides the performance results of the three models and their comparison. Section 5 shows how system reliability might be characterized based on inputs from the simulation model. Section 6 provides some concluding remarks and some avenues for extending the current research.

2. Background and Literature Review

WSs follow a client/server paradigm, where the middleware running on a server connects clients to the desired WSs. The middleware is formed by multiple layers. The application server layer, e.g. JBoss AS, is the main engine of the middleware that provides the environment for running applications regardless of what the application might be. One of the major tasks of the server is to hide the intrinsic details of common programming tasks such as security, persistent storage, and queuing requests, so that application developers can concentrate on the business logic rather than on periphery aspects. It is through this layer that web services can communicate with other layers. The web server layer, e.g. Apache (Apache, 2012), is the front end software that delivers requests to the application server on which web services are deployed. The database layer stores data relevant to web services.

2.1 Web service reliability

The reliability of a software system is a probabilistic measure of correctly delivering services during a period of time. The measured reliability depends on fault types considered, such as crash faults, software faults, exception faults, time-outs, resource exhaustion, misconfiguration of the underlying shared resources, etc. For example, one may want to obtain the reliability of the system based only on resource exhaustion. On the other, as fault sources are often numerous, to make reliability analysis manageable, attempts can be made to categorize the faults based on their impacts rather than their origin. For instance, an expected message not received in the client side might be due to a fault in the network, or a fault in the server side software component or because the configuration parameters in the middleware are not set correctly. So the spectrum of faults with the impact of not receiving messages can be categorized as omission faults. In general, a possible categorization of faults based on their impacts that collectively captures all kinds of faults is timing faults, omission faults, and arbitrary faults (Azadmanesh et al., 2008; Srinivasan and Azadmanesh, 2013). A timing fault occurs when an expected correct message is not received within a predetermined period of time. An omission fault happens when no response is received. Finally, an arbitrary fault is the one that behaves in any manner other than timing or omission faults. For instance an arbitrary fault can cause an erroneous value to be received or when two conflicting values are transmitted as response to the same input values. This study is mostly concerned with the timing (time-out) faults of web services. More specifically, this study follows the definition presented by Hwang, et al. (2008), which defines the web service reliability as “the probability the web service successfully responds within a reasonable period of time”. This form of timing faults can be caused by many factors such as delays in the middleware, inadequate setting of configuration parameters, or network problems. A study (Pertet and Narasimhan, 2005), which used information from Google, IBM, Intel, Microsoft, Hewlett-Packard, Oracle, and Sun,
indicates that the main cause of failures in web applications is not logical or computational errors. Rather, the most frequent causes of failure are due to system overload, configuration errors, resource exhaustion, human/operator errors, and lack of resource sharing. Furthermore, to the best of our knowledge, many architecture-based reliability studies do not consider these failures (Grassi, 2005; Zhong and Qi, 2006). Although some studies (Grassi and Patella, 2006; Grassi, 2005) claim that architecture-based reliability approaches can be applied to service-oriented computing applications, no solid work exists to confirm this. For instance, Grassi (2005) presents an approach to the reliability prediction of an assembly of services that uses the architecture-based reliability analysis. The 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.

2.2 Related work

Due to space limitation, a comprehensive enumeration of research efforts on architecture-based reliability analysis is infeasible. Therefore, the following provides a limited discussion of the relevant work loosely partitioned into simulation-based and experimental-based.

Simulation-based reliability models - Some of the most common approaches in architecture-based reliability analysis rely on Markov and Petri net model formulation (Cheung, 1980; Gokhale et al., 2006; Rahmani et al., 2011; Wells et at., 2001). The majority of the research in this area is theoretical with less emphasis on experimental analysis to support the theoretical results (Singh et al., 2001; Souza et al., 2006; Zhong and Qi, 2006). For example the work in Cheung (1980), which is one of the pioneer works in architecture-based software reliability, transforms a simple software system architecture to a Markov model to determine the overall system reliability. 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 Zhong and Qi (2006), the authors focus on performance and reliability of web service composition. The 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 net (SPN) models. The work emphasizes on theoretical aspect of research and does not include any experimental analysis. In another work (Souza et al., 2006), the pooling mechanism of Enterprise Java Bean (EJB) of JBoss AS is investigated using Petri net simulation. However, the numerical illustration of the presented simulation model for performance prediction is based on hypothetical input and not real experiments. In Sing et al. (2001), the authors propose a theoretical model integrated with Unified Modeling language (UML) models (Fowler, 2003) to predict the reliability of the system in the early phases of system analysis and design. Instead of using traditional use-case and sequence diagrams, the authors suggest using some annotated diagram about the expected system usage patterns and failure probabilities of the components. 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. (2006) have proposed an analysis methodology based on the Stochastic Reward Net (SRN) (Sakaguchi and Somani, 1998) 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 Wells et al. (2001), a framework is presented to construct Colored Petri Net models (Jensen, 1987) for performance analysis of a hypothetical web server and web clients connected to a LAN. The model is simplistic without any experimental validation. In Rahmani et al. (2011), the authors developed a Petri net model to evaluate the reliability of a web service deployed in an application server. The methodology concentrates on interdependencies among 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.

Experimental-based reliability models - There are some studies that use a combination of theoretical and experimental analyses in reliability estimation of software systems (Xiao and Dohi, 2010; Mei et al., 2001; Cao et al., 2003; Zheng and Lyu, 2010; Goseva-Popstojanova et al., 2005). Xiao and Dohi (2010) developed a probability model to describe the relationship between the error rate in Apache web servers and the system parameters such as available virtual memory size. Although the work is concentrated on performance analysis of a web-based system deployed in a popular web server, it does not analyze the internal architecture of the system, i.e. the system is treated as a black-box. In Mei (2001), the authors consider the impacts of the client workload and the server hardware/software configuration on performance analysis of web servers. The authors use queuing model to evaluate the effectiveness of the model through the experiments in a lab environment. But the work does not cover more complex internet-based software systems such as application servers and web services. In Cao et al. (2003), 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. An HTTP request sent by a client is considered blocked either when the maximum number of connections in the server is reached or the TCP connection times out at the client computer. The authors 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. Zheng and Lyu (2010) proposed a collaborative reliability prediction model to evaluate the response time and throughput of web services by collecting and analyzing the past failure data from geographically distributed locations. In the study, the web service itself and the underneath architecture of the distributed
servers are treated as a black-box and thus the internal architecture of the system is not considered. Goseva et al. (2005) presented an empirical as well as a theoretical study of architecture-based software reliability on a large open source application with 350,000 lines of C code. Although the work provides valuable insights, the application considered is a standalone system and not a distributed or internet-based application. Finally, Harkema et al. (2004) experimented with the performance of CORBA middleware using the simulation tool Extend (Krahl, 2002) that allows users without advanced simulation programming experience to connect pre-built blocks and embed various performance parameters to simulate system behavior. Although the study does not include an analytic component or white-box analysis, it conducts dynamic analysis by tagging requests and tracing their executions within the middleware. Furthermore, the study includes some performance analysis based on system configuration parameters such as thread pool size for servicing requests. The authors validated their performance model by comparing the simulation results against those from lab experiments.

3. Description of Performance Models

To investigate the performance and reliability of the web service system as described, three models are developed: experimental, analytical, and simulation. The following sections describe these models in more detail.

3.1 Experimental model (EM)

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 4.2.2 is running is excluded from running other tasks to ensure the consistency of data sets collected. The Duke’s Bank application (Duke, 2012) is transformed into a web service and deployed on JBoss AS, which uses Apache Tomcat as the default web server. The client generates service requests to JBoss AS. The bandwidth of the LAN is shared with other users not relevant to this experiment. Tools like Wireshark (2012) and Ping 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 so negligible that it is ignored in the EM analyses.

On the client side, SoapUI (2012) is used to generate controlled service request loads to the server. SoapUI is an open source testing tool solution for service-oriented architectures. With a graphical user interface, SoapUI is capable of generating mock service requests in SOAP format. There are two main parameters in SoapUI that can be set to control the workload of the application server: number of threads representing the virtual clients (users), and the number of requests (runs) generated per virtual client. For example, if the number of virtual clients is set to 20 and number of runs per client is set to 10, then there are 20 clients (users), each sending 10 SOAP requests for a total of 200 requests. SoapUI measures 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. The experimental model (EM) setup is illustrated in Fig. 1.

3.2 Analytical model (AM)

The total time \( T \) for each load test configuration is computed as follows:

\[
T = \frac{cnt}{tps} \tag{1}
\]

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

\[
error_{rate} = \frac{err}{cnt} \tag{2}
\]
The SoapUI error reports consist of all types of potential errors that are generated by the network, application server, and web service itself. However, this study mostly considers the errors that are generated by JBoss AS due to the HTTP requests that are rejected. Therefore, error_rate is treated as rejection_rate.

Although (2) provides the actual rate of requests rejected, this rate can be estimated in a different way. Recall that avg is the average response time for one request. Therefore, the service rate is 1/avg. In order for JBoss AS not to reject any incoming request, it should be able to use enough threads to keep up with the arrival rate. Thus, JBoss will reject no requests if the following holds:

\[
\frac{1}{\text{avg}} \times \text{threads} \geq \text{tps} \Rightarrow \frac{1}{\text{avg}} \times \text{threads} - \text{tps} \geq 0
\]

Otherwise some requests will be rejected. Thus, the number of requests rejected per unit of time, i.e. rejection_rate, is:

\[
(tps - \frac{1}{\text{avg}} \times \text{threads})
\]

Consequently, the total number of requests rejected is:

\[
\text{rejection}_{\text{est}} = (tps - \frac{1}{\text{avg}} \times \text{threads}) \times T
\]  

The parameters tps and T can easily be calculated from the results returned by SoapUI. Hence, the accuracy of (4) relies on threads, which will be referred to as threshold. This is because any value higher than the threshold value underestimates and any value lower than the threshold overestimates the number of rejections. Thus, the measured number of requests rejected in (4) is modified as shown in (5). Section 4 will show how closely this analytical equation can estimate the number of requests rejected.

\[
\text{rejection}_{\text{est}} = (tps - \frac{1}{\text{avg}} \times \text{threshold}) \times T
\]

### 3.3 Simulation models

The simulation models are developed using Petri nets, which are graphical models for the formal specification and analysis of concurrent and discrete-event systems. Since the introduction of classic Petri nets, a number of variants have been introduced. Stochastic Petri Nets (SPN) is a subsidiary of timed Petri nets 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) (Sanders and Meyer, 2001) is a structurally extended version of GSPN with many features such as the ability of creating complex enabling functions, probabilistic choices upon completion of activities, and reward functions. In the world of SANs, transitions (actions) are referred to as activities, which can be of two kinds: timed and instantaneous. The firing (activation) time of timed activities is exponentially distributed. Once enabled, instantaneous activities complete in zero-time, and thus have priority over timed activities. Reward functions are used to measure performance or dependability related issues. The SAN model describing the behavior of the web service system is constructed using Mobius (2012). 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.

Two different SAN models are developed: black-box simulation model (BSM) and white-box simulation model or architecture-based simulation model (ASM). In BSM, the whole web service system is considered as one timing transition and thus architectural information is not applied into the model. On the other hand, in ASM, the middleware is broken into layers and each layer is developed into an atomic SAN model. Since BSM is easier to develop and the fact that it follows the AM in (5), its performance results will be used to validate those obtained by ASM.

#### 3.3.1 Black-box simulation model (BSM)

Recall that this study uses the Duke’s Bank web service. This web service receives a customer id and returns the associated customer’s bank account numbers to the client. Fig. 2 provides the SAN model for BSM for the service requests that arrive at the server (JBoss AS) side. In the figure, the timed and instantaneous activities are shown by thick and thin bars, respectively. A flat dot on the right side of an activity represents a probabilistic choice that leads to taking a different path in the model once the activity completes. The place called Requests is initialized to cnt, the number of total HTTP requests for each test case. The activity rate of Tarrival is 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. Once Tarrival completes, the token generated, through the output gate OG, shown as a solid triangle, will be deposited in either Start or Blockedrequests. OG represents the conditions for rejecting HTTP requests. For instance, if Start has reached its maximum capacity, OG will redirect the token to Blockedrequests; otherwise the token is added to Start. The rejected requests are accumulated in Down via T03 activity. Activity TJW, which represents the service rate of the application.
server, is enabled only if *Threadpool* and *Start* are not empty. When *TJW* is activated, a token in *Threadpool* representing an available thread in JBoss AS is allocated to a request in *Start*. Once the request is serviced, the thread, through *T01*, is released to *Threadpool* to be used by a next request. In case the server fails to service the request (lower case of *TJW*), the allocated thread is returned to *Threadpool* via *T02*. The failure probability (lower case) of *T01* is the probability that the service request is failed by means other than failures caused by the application server, such as a failure in the network or unregistered service. In case of such a failure, the allocated thread is returned to the thread pool via *T04*. These failure probabilities are set to zero but can be set to non-zero values. Note that the sum of the case probabilities on each activity must be equal to 1.

![BSM for JBoss serving the requests.](image)

**3.3.2 Architecture-based simulation model (ASM)**

In BSM, the entire middleware is treated as a black-box, represented by the activity *TJW* in Fig. 2. In ASM, the *TJW* activity will be partitioned into finer SAN submodels based on the major architectural layers of the middleware. This allows for finer granularity of analysis such as the impact of sensitive components on the overall reliability of the system that is otherwise not possible in the black-box approach.

As this study concentrates on web service environment and underlying middleware, the failure probability of the operating system and hardware on which the application server runs are not considered. The phases to convert the layered approach to a modularized Petri net model are shown in Fig. 3 and explained below.

![The proposed approach for ASM performance analysis of framework-intensive applications.](image)
**Static analysis** - The main purpose of this phase is to obtain information about the static structure of the system and defining the main components/layers of the architecture. A suitable and manageable level of abstraction needs to be determined. Major layers of the web service system are determined using the architectural tool Structure101(2012). This tool is capable of producing several views helpful for understanding the architecture. The overall overview of layers and interested configuration parameters extracted from static analysis are presented in Fig. 4. The main layers determined are the web server (Tomcat), application server (JBoss AS), database (hsqldb), and the web service application itself, which in our case study is the Duke’s Bank web service (Duke, 2012).

![Diagram of Web Service System](image)

*Fig. 4. The overall overview of Web Service system showing the interaction between layers and configuration parameters.*

As presented in performance modeling literature review (Dufour et al., 2007; Imre et al., 2007), configuration parameters can influence the performance and reliability of web services. Misconfiguration of these parameters can cause delay in the system tasks and degrade overall dependability. As shown in the figure, there are at least two main components in JBoss in the form of shared resources that have direct impact on performance and reliability: *HTTP thread pool* of the web server and *database connection instance* of the data access layer. These components can be configured before each startup of the application server. The performance impact of these components will be investigated in a later section.

**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. Although the main layers are extracted during the static analysis, instrumentation of the middleware at run time leads to the discovery of the call-graph and the timing information of the software layers. When combined with the key configuration parameters and failure probabilities of the layers, the dynamic analysis provides deeper understanding of how various components affect the overall reliability and system performance.

Since the running code in the middleware is very complex with multiple threads running simultaneously, we can prune the code by pointing the starting class that initiates a specific thread for dynamic analysis and avoid monitoring all other threads in the middleware that may not be related to the service itself. By specifying a thread and tracing its execution path, the section of code for reliability perspective can be isolated, which results in the reduction of code that needs to be considered for reliability modeling. We start with *JIOEndpoint$Worker* thread which handles the client’s requests.

**Instrument middleware with profilers** - Since the call-graph and timing information of the middleware are needed for SAN-based reliability analysis, the first step to gather these data is to instrument the software with a profiler (Goseva-Popstojanova et al., 2001). A profiler like JRAT (2012) is able to provide information about the time spent for each method-call but unable to accurately create the call-graph. Another profiler named Javashot (2012) is used to extract the flow of method-calls during the execution of a thread. Javashot is a Java-based profiler that uses Java instrumentation capabilities, e.g. Javassist (2012), to capture the dynamic execution flow of a Java program. Instrumentation with these tools combined not only gathers all information about the middleware run-time behavior, but also collects information from the deployed service.

**Execute test cases** - Various service requests are created in order to collect two different sets of information. The requests are sent from a SoapUI client to extract bank accounts of a customer. A request consists of one transaction of hsqldb database. To build the call-graph, the first set of information is gathered by Javashot. The second set acquired by JRAT contains information about the execution times spent for each method-call. A sample of the two sets of information is presented in Fig. 5. As shown in Fig. 5a, Javashot generates its output in .dot format that represents a directed graph. Javashot is aimed to gather the execution trace of a specific thread from a multi-threaded environment. The output can be used as input to visualization tools such as Graphviz (2012) or ZGRViewer (2012).
A final version of this article has appeared in Journal of Systems & Software. This version has not been fully edited

Architecture modeling and parameter estimation - With one request received from a banking user, there were a total of 29,650 calls from the middleware logged by Javashot, in which 289 calls were from direct calls to the web service itself. Because of the amount of information gathered by Javashot, visualizing the whole dynamic call graph and transforming it to a SAN model is cumbersome. Therefore, a program is developed that takes the Javashot output and produces an abstract view of the dynamic flow graph with fewer number of components/layers, and computes the transition probabilities between these high level layers. This program can produce the graphical view in any details. For example, Fig. 6a shows a third-level abstraction. Blue nodes represent different components in JBoss AS such as: org.jboss.mx and org.jboss.system. Yellow nodes represent the Apache web server components. Red and green nodes are the web service and database components, respectively. The larger nodes mean that they are used more often than other nodes. The intensity of communication between nodes is shown by the color of the edges, with red edges indicating more intensity.

When considering a second-level abstraction, then this graph can be compacted to a graph with five nodes, where the same color nodes are merged into a single node, as shown in Fig. 6b. The labels on the links indicate the probability of calls to nodes, which will be explained further shortly. To compute the transition probability from a node A to node B, the number of times node A has transferred control to node B is divided by the total number of times that node A transfers control to other nodes including B.
As indicated, a call graph may contain a huge number of calls, so a complete graph extraction is too complex to be converted to a Petri net model. To alleviate this, the second-level packages are used, as shown in Fig. 6b. So each node in the figure represents a layer, as shown in Fig. 4. In the figure, Org.apache is an open source implementation that contains the Java Servlet technology (Apache, 2012). This technology is a Java component used to extend the capabilities of servers that host applications access via a request-response programming model (Java Servlet, 2012). A servlet container such as Org.apache.tomcat, also known as Web Container (2012), which is a part of the Org.apache layer is a component of the web server that interacts with the servlets. Org.jboss is the main component of the application server that is responsible for the interactions between the deployed applications and other layers. Logging, sending messages between components, and supporting secure transactions are other responsibilities of this layer. Org.hsqldb is an open source implementation of hsqldb database (Hsqldb, 2012). 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. Also, as indicted, JRAT is able to record the time spent in the methods, classes, and packages, and consequently the layers called. To collect the time spent in each of the nodes in Fig. 6b, another program is developed to compute the average time spent in each layer by adding the time spent in each method that belongs to the same layer.

Build Petri net model – In this phase, the call sequence, as shown in Fig. 6b, is transformed into a modularized Petri net model consisting of the following atomic submodels: Request, Webserver-JBoss-Webservice, JBoss-Database, and Webservice-Response. Each atomic submodel represents a partial view of the call-graph, which are interconnected according to the interactions among the layers. The purpose of each submodel is described below:

Request: This submodel 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 on the server side.

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

JBoss-Database: This submodel handles the interaction between the database and JBoss AS.

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.

As indicated, these submodels fit the communication among the layers in Fig. 6b. Note that communication over each link may happen multiple times, e.g. communication between JBoss and database in JBoss-Database. But for a submodel such as JBoss-Database to connect to another submodel, database must first return control to JBoss. The SAN model for each of the submodels is explained in the following:

Request Submodel — Fig. 7 provides the first submodel when service requests made by the Duke’s Bank web service client arrive at the server side. The submodel shows whether an arriving request should be accepted by the application server or be rejected due to the lack of resources (threads). As indicated, the major configuration parameters are the number of threads in the threadpool and database connection instance pool. These resources can be configured by the administrator using server.xml and hsqldb-ds.xml files in JBoss AS, respectively. Tarrival gives the rate of arrivals per unit of time. Activity Tmware represents the acceptance and setup rates, which includes services such as queuing, operating system tasks, resource allocation and activation. The output gate OGI prevents the queue overflow, so the requests that cannot be queued are simply rejected and guided to 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, which is the starting point of the Webserver-JBoss-Webservice subsequence of operations. It should be observed that this submodel is mostly similar to Fig. 2, with the difference that TJW in Fig. 2 is the rate at which the requests are serviced. This rate is based on the total time from the point the requests are made till the time the responses are available, whereas Tmware in Fig. 7 shows a portion of this rate associated with setup and resource allocation. In other words, TJW is partitioned in the modularized approach throughout the time activities as shown in Fig. 7–10.
A final version of this article has appeared in Journal of Systems & Software. This version has not been fully edited

Webserver-JBoss-Webservice Submodel- Fig. 8 is a SAN model that shows the interaction between the Apache, JBoss, and the web service layers. After the thread allocation phase, the Apache layer is the first layer that serves the HTTP requests. StartThread is the starting point in the web server to process a request made. As presented in Fig. 6b, org.apache communicates with org.jboss and org.jboss in turn communicates with com.sun.bank node (web service layer). TApache is the service rate of Apache. In this submodel, the JBoss layer is divided into two timed activities JBoss0 and JBoss1, which represent different packages of JBoss AS that communicate with the Apache web server and Duke’s Bank web service respectively.

![Fig. 8. The Webserver-JBoss-Webservice submodel.](image)

A request waiting in StartThread passes through Apache and JBoss with some probability of being resserviced (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 JBossFail and this enables T100 to be fired that deposits a token in PartialFailure. Note that when an activity contains probability cases, only one of the cases will be followed. For example, once TWebservice activity completes, 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, some requests will end up in ToDBConnect through case 2 of TJBoss1. ToDBConnect is the connection point to the JBoss-Database submodel shown in Fig. 9. In other words, this place is shared between the two submodels.

JBoss-Database Submodel - The submodel in Fig. 9 captures the interaction between the JBoss layer and the Database layer, i.e. hsqldb.

![Fig. 9. The JBoss-Database submodel.](image)

Once a request is ready to be serviced by the database layer, i.e. when ToDBConnect is not empty, the output gate OG2 checks to ensure a connection can be made to the database. This decision is based on the global variable dbthreads, which is the maximum number of instances for database connections. This variable can be set in server.xml configuration file in JBoss AS. If a database connection cannot be made, a token will be placed in DBRejected. Once rejected, the request’s thread that was allocated from ThreadPool (see Fig. 7) will be returned to ThreadPool. DBRejected is one of the connection points to the next submodel that finally deposits the thread into ThreadPool.

On the other hand, if the request is transferred to DBStart and there is an available database instance in DBCconnection, the activity 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. This activity is based on the time spent in the packages that interact with the database server. These packages are different from those associated with TJBoss0 and TJBoss1 in Fig. 8.
Once activity TJBoss2 completes, the token produced can take three different directions based on three different probability cases. The first probability case is the probability of coming back to the database. The second probability case is the probability of entering the next step of finishing interaction with the database. The third probability case is the probability of JBoss failure. The JBoss failures considered in this submodel are the ones that do not cause a crash in the system. Therefore the submodel 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 Fig. 7) will be returned by placing a token in MainReturn. MainReturn is a place that is shared with the next submodel in Fig. 10.

Webserver-Response Submodel – This submodel, shown in Fig. 10, 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 Fig. 10 is shared with ThreadPool in Fig. 7. In the figure, any labels for places can be used. However, to easily see which places are shared, the shared places among submodels use the same labels. For example, ThreadPool in Fig. 7 and Fig. 10 are shared. Recall that MainReturn represents the successful return of a request. DBRejected, WebServiceFailed, and PartialFailure are three other paths 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, including this submodel provides more readability, and the fact that the activities can be used to report the number of unsuccessful requests due to failures in the webserver, database, web service, or the JBoss layer.

![Fig. 10. The Webserver-Response submodel.](image)

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.

Gather and analyze results – This is the last phase of ASM development shown in Fig. 3. In this phase, results are collected, analyzed, and compared against the experimental results provided by SoapUI.

4. Performance Analyses

The previous section described the three models used in this study. This section is concerned with running these models and analyzing their results. As the EM is aimed to embody a realistic situation, the AM, BSM, and ASM performance results ought to be close to those of EM. In other words, the EM results are used to validate the AM, BSM, and ASM results.

4.1 Experimental Model (EM) Performance

The experiments have been conducted with sixteen different load configurations. For each load test, SoapUI returns the values for avg, tps, and err. Table 1 shows a sample data extracted from SoapUI. Each test is repeated five times and the average of the returned values are calculated. The data collected from SoapUI will be used to test the performance of the AM, BSM, and ASM. For instance, tps will be used as the arrival rate for the SAN models.
Table 1. Sample data extracted from SoapUI

<table>
<thead>
<tr>
<th>Users (threads)</th>
<th>Runs per thread</th>
<th>Cnt</th>
<th>Avg (sec)</th>
<th>Tps</th>
<th>Total time (sec)</th>
<th>Average 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>82.55</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>350</td>
<td>10</td>
<td>3500</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>

Theoretically, when the configuration parameters `maxThread` and `acceptCount` in `server.xml` file are set to 250 and 100 respectively, it is expected that JBoss AS handle 350 requests (250 in the thread pool and 100 in the queuing system). `MaxThread` is the maximum number of threads allowed in the thread pool of the web server and `acceptCount` is the maximum number of requests allowed to wait in the web server queuing system. 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 that can be devoted to requests. From the test cases performed on SoapUI while observing the JBoss performance, as long as the number of threads allocated is less than 315, the server will be able to service the requests with approximately no rejections to occur. This threshold value depends on the characteristics of the server, for instance how fast the server is able to service requests. Once this value is determined, it stays fixed and can be used to predict the server performance under various scenarios as explained in the rest of this study.

On the other hand, it is expected that the response time, `avg`, to increase as the number of requests increases. The response time evaluated by SoapUI includes the time spent in the queuing system and the time actually spent on servicing the request. Fig. 11 shows the `avg` numbers for different virtual users in the lab, where each user sends 10 requests in total. The figure exhibits that the response time increases up to a point, levels-off for a moderate range of requests, and then increases again.

![Fig. 11. Average response time based on number of virtual users.](image)

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. There are two reasons for this paradoxical behavior. The first reason is that the status code of response messages for rejected requests indicates no error, so SoapUI is unable to distinguish these responses from other responses that are serviced normally. Obviously, with increasing number of users, the rejection rate should increase. But, since the rejected requests take less time to respond to and SoapUI uses the total requests in calculating the average response time, regardless of whether a request is successful or rejected, this phenomenon will contribute to lower overall average response time for a limited period of time. It is for this reason that the average response time of 15.11 in Fig. 11 can be interpreted as the true response time because at about 370 users, the number of rejections is low or almost nonexistent. The second reason is contributed to JBoss AS, which can be explained as follows. 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 new requests without the need of setup.
time for new arrivals. Beyond 500 users, the server overhead, such as the time taken to reject the requests, accumulates as the rejection rate increases, which leads to sharp increase in the response time.

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 a packet capture tool such as 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, which is infeasible due to large 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 timing of rejected requests. Therefore, in this study \( \text{avg} = 15.11 \) and \( \text{threads} = 315 \) are used as the closest approximation of the two parameters before rejections occur.

### 4.2 Analytical Model (AM) Performance

Inserting \( \text{avg} = 15.11 \) and \( \text{threshold} = 315 \) obtained from the EM analysis into (5) produces the following:

\[
\text{rejection}_{\text{est}} = \left( \frac{1}{15.11} \times 315 \right) \times T
\]

The comparison of rejected requests between EM and AM will be shown in the next section when the simulation model is covered. The reason is to provide a comprehensive graphical view of the three models in one graph rather than using a graph for each model performance.

### 4.3 Simulation Models Performance

#### 4.3.1 BSM performance

Table 2 shows the parameters and their values for the SAN model of BSM shown in Fig. 2. The impulse reward functions count the number of activations. For instance, the number of activations at \( T_{\text{arrival}} \) shows the total number of requests entering the Petri net model. Similarly, the number of activations at \( T_{03} \) represents the number of requests rejected. Let \( \text{rejection}_{\text{BSM}} \) be the number of these rejections, i.e. the number of \( T_{03} \) activations.

<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>( T_{\text{arrival}} ) rate</td>
<td>Tps (from SoapUI)</td>
</tr>
<tr>
<td>threshold</td>
<td>315</td>
</tr>
<tr>
<td>queue</td>
<td>acceptCount (100, from server.xml)</td>
</tr>
<tr>
<td>threadpool</td>
<td>maxThread (250, from server.xml )</td>
</tr>
<tr>
<td>Threadpool</td>
<td>Initialized to threadpool</td>
</tr>
</tbody>
</table>

**OG**

If (Start \( \rightarrow \text{Mark()} \) < (threadpool + queue))

\[
\text{Start} \rightarrow \text{Mark()} = \text{Start} \rightarrow \text{Mark()} + 1;
\]

Else

\[
\text{Blockedrequests} \rightarrow \text{Mark()} = \text{Blockedrequests} \rightarrow \text{Mark()} + 1;
\]

**TJW rate**

If (Start \( \rightarrow \text{Mark()} \) < Threadpool \( \rightarrow \text{Mark()} \))

\[
\text{Return} \left( \left( \frac{1}{\text{avg}} \right) \times (\text{Start} \rightarrow \text{Mark()}) \right) ;
\]

Else

\[
\text{Return} \left( \left( \frac{1}{\text{avg}} \right) \times \text{threshold} \right) ;
\]

**T01 prob. case1**

1

**T01 prob. case2**

0

**TJW prob. case1**

1

**TJW prob. case2**

0

**Impulse reward functions in SAN**

Total number of requests made:

If \( T_{\text{arrival}} \) fires then return 1;

Number of requests rejected:

If \( T_{03} \) fires then return 1;
Table 3. Request rejection rate for all sixteen tests

<table>
<thead>
<tr>
<th>Number of simultaneous users</th>
<th>EM Request rejection rate (SoapUI)</th>
<th>AM Request rejection rate (rejection_{req} /cnt)</th>
<th>BSM Request rejection rate (rejection_{BSM} /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>

The Mark() function indicates the number of tokens in a place. For example, Start $\rightarrow$ Mark() represents number of tokens in Start. Since the threads in JBoss are executed in parallel, the if-clause of the TJW rate in the table ensures that there are enough threads to be allocated to the requests. The else-clause, however, is not intuitive enough. The else-clause should be looked at with the OG condition in mind. JBoss AS allows a maximum of queue requests to be queued. In other words, if Threadpool is empty, there can be at most queue tokens in Start. Consequently, if there are $x < \text{threadpool}$ tokens in Threadpool, the maximum number of tokens in Start is $(\text{queue} + x)$. Since the value of $x$ changes depending on the available threads, the maximum tokens in Start, i.e. $(\text{queue} + x)$, continuously changes as well. This causes threshold that represents the speed at which the SAN model services the requests to be dynamic. In other words, threshold needs to be throttled each time $x$ changes. This makes it difficult to predict an appropriate threshold value that meets the rejection rate observed by the experiments performed using SoapUI. Thus, the maximum value of Start is set at the fixed value $(\text{threadpool} + \text{queue})$. This in turn makes threshold to be a fixed value. As it will be shown shortly, this approach has shown that performance of the SAN model is very close to that of the rejection rate reported by SoapUI.

Table 3 shows the results for the 16 tests obtained from SoapUI, the analytical equation (6), and from running the SAN model. Fig. 12 displays graphically the request rejection rate for the three different models shown in Table 3. The figure shows that the rejection rate of AM and BSM closely matches the one provided by SoapUI. The worst discrepancy in rejection rate happens at 380 users, which is $0.13 - 0.08 = 0.05$ and $0.14 - 0.08 = 0.06$ for AM and BSM, respectively.

Fig. 12. HTTP rejection rate.
4.3.2 ASM performance

Table 4 presents the actual sequence of calls obtained from Javashot for the call-graph of ASM shown in Fig. 6b. The sequence of calls is from the moment a request is sent from a web-service client to the server until a response is generated. The call sequence is partitioned into three atomic models according to Fig. 8 - 11, i.e. Webserver-JBoss-Webservice, JBoss-Database, and Webserver-Response. The table also shows the timing information computed by JRAT profiler. Some calls took a small fraction of time that is logged by JRAT as zero milliseconds. To make the table simple enough, such calls are not shown.

Table 4. Aggregated timing information extracted from JRAT

<table>
<thead>
<tr>
<th>Layer name</th>
<th>Average time spent in each sublayer (sec)</th>
<th>SAN submodel name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.org.apache</td>
<td>0.0490</td>
<td>Webserver-JBoss-Webservice</td>
</tr>
<tr>
<td>2.org.jboss</td>
<td>0.0600</td>
<td></td>
</tr>
<tr>
<td>3.org.jboss</td>
<td>0.0070</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>JBoss-Database</td>
</tr>
<tr>
<td>8.org.hsqldb</td>
<td>0.0093</td>
<td></td>
</tr>
<tr>
<td>9.org.jboss</td>
<td>0.0005</td>
<td></td>
</tr>
<tr>
<td>10.org.hsqldb</td>
<td>0.0013</td>
<td></td>
</tr>
<tr>
<td>11.org.jboss</td>
<td>0.0010</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>0.0620</td>
<td></td>
</tr>
<tr>
<td>20.org.apache</td>
<td>0.0000</td>
<td>Webserver-Response</td>
</tr>
<tr>
<td><strong>Total time:</strong></td>
<td><strong>0.29</strong></td>
<td></td>
</tr>
</tbody>
</table>

Looking at the sequence of the atomic models in Table 4, one can interconnect the models to reach the following logical sequence of execution:


The timing information is used to compute the activity rates for the atomic models. For example, the rates for TApache, TJBoss0, and TJBoss1 activities in Fig. 8 are 1/0.049, 1/0.06, and 1/(0.007+0.0086+0.0283), respectively. These and other activity rates for all submodels are shown in Table 5. Additionally, the composed model of ASM needs conditional statements and global variables. Therefore, Table 5 also lists the statements, the initial values, and the case probabilities extracted from the output generated by Javashot.
Table 5. Global parameters setup for ASM

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Parameter value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarrival rate range</td>
<td>21…46</td>
</tr>
<tr>
<td>TJBoss0 prob. case1</td>
<td>0.23</td>
</tr>
<tr>
<td>TJBoss0 prob. case2</td>
<td>0.77</td>
</tr>
<tr>
<td>TJBoss0 prob. case3</td>
<td>0.0</td>
</tr>
<tr>
<td>TJBoss1 prob. case1</td>
<td>0.64</td>
</tr>
<tr>
<td>TJBoss1 prob. case2</td>
<td>0.36</td>
</tr>
<tr>
<td>TJBoss1 prob. case3</td>
<td>0.0</td>
</tr>
<tr>
<td>TJBoss2 prob. case1</td>
<td>0.13</td>
</tr>
<tr>
<td>TJBoss2 prob. case2</td>
<td>0.87</td>
</tr>
<tr>
<td>TJBoss2 prob. case3</td>
<td>0.00</td>
</tr>
<tr>
<td>TWebService prob. case1</td>
<td>0.99</td>
</tr>
<tr>
<td>TWebService prob. case2</td>
<td>0.01</td>
</tr>
<tr>
<td>threshold</td>
<td>315</td>
</tr>
</tbody>
</table>

queue acceptCount (100, from server.xml)
dbthreads DBConnection (20, from server.xml)
DBConnection Initialized to dbthreads
threadpool maxThread (250, from server.xml)
ThreadPool Initialized to threadpool

TApache rate
// From Table 4: 1/0.049 = 20.4
StartThread  Mark() * 20.4

TJBoss0 rate
// From Table 4: 1/0.06 = 16.67
P01  Mark() * 16.67

TJBoss1 rate
// From Table 4: 1/(0.007+0.0086+0.0283) = 22.8
P02  Mark() * 22.8

TJBoss2 rate
// From Table 4:
// 1/(0.0005+0.001+0.0039+0.01003+0.0382) =8.65
P05  Mark() * 8.65

TWebService rate
// From Table 4: 1/(0.0075+0.0075) =66.67
P03  Mark() * 66.67

TDB
// From Table 4:
// 1/(0.0003+0.0013+0.0022+0.0013+0.0019+0.0001) =140.85
P04  Mark() * 140.85

Tmware
// From Fig. 11 and Table 4: 1/(15.11 – 0.29) = 0.067
If (Start  Mark() < threshold)
  Return ((Start  Mark()) * 0.067)
Else
  Return (threshold * 0.067)

OG1
If (Start  Mark()) <= (queue+threadpool)
  Start  Mark() = Start  Mark()+1
Else
  RejectedRequest  Mark() =
  RejectedRequest  Mark()+1

OG2
If (DBStart  Mark()) <= dbthreads)
  DBStart  Mark() = DBStart  Mark()+1
Else
  DBRejected  Mark() = DBRejected  Mark()+1

Fig. 13 displays the effect of arrival rate on rejection rate of requests. Fig. 13a shows the results of actual tests in the lab and Fig. 13b displays the trend of predicted failure rate by ASM under various arrival rates up to 50 requests per second. Both graphs show that with increasing the arrival rate to the server, the HTTP rejection rate will be increased. These rates are extracted from Tables 1 and 3 and should provide a better view on the rejection rate trend. The fluctuation at the early part of
the graph in Fig. 13a can be caused by rounding errors, minute changes in communication delay, and averaging the rejection rates for the 5 repetitions of each load test. In spite of these changes, Fig. 13b shows ASM is capable of providing a good prediction of performance similar to that of the realistic case as the arrival rate changes.

![Fig. 13](image)

(a) HTTP request rejection rate for EM (SoapUI)  
(b) HTTP request rejection rate for ASM.

Fig. 13. The EM and ASM rejection rate of Duke’s bank web service (based on JBoss As default setting in Table 5).

Fig. 14 displays the increasing trend of rejection rate based on the number of simultaneous users reported by both SoapUI and ASM. Recall that SoapUI initiates a thread for each user in the client side and each user sends ten requests to the web service. The figure shows that ASM is able to closely predict the HTTP rejection rate reported by SoapUI. Furthermore, comparing this figure and Fig. 12 shows that ASM closely tracks the performance of AM and BSM.

![Fig. 14](image)

Fig. 14. HTTP rejection rate of EM (using SoapUI) and ASM (using the parameter values in Table 5).

In the third experiment, Fig. 15 shows the rejection rate of the database for the ASM 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. The rejection rate is presented under various loads of 200, 400, 700, and 1200 simultaneous users. The rest of the parameters are stayed fixed as listed in Table 5. The figure demonstrates that if the maximum number of database connection instances is less than a specific threshold, performance 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 as the number of simultaneous users in ASM changes. As presented in Table 5, the web service error probability is set to 0.01. Fig. 16 displays that the web service error rate is very low and stable when the error probability of web service is set to 0.01. The figure further shows that the HTTP rejection rate is also untouched when compared to Fig. 14. This is because the HTTP rejections are calculated before the requests enter the web service layer. According to Fig. 7, HTTP rejections are obtained by collecting them in place Down. 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 activations in TWSFailed of Fig. 10.

In the next experiment, Fig. 17 shows the error rate of JBoss AS computed by ASM. The model uses the same parameter values from Table 5, with the exception of the three JBoss layer transition probability values, which are shown in Table 6. These transitions are for the submodels in Fig. 8 and Fig. 9. The JBoss AS error probability is represented by TJBoss0 probability (case 3), TJBoss1 probability (case 3), and TJBoss2 probability (case 3). In the EM analysis of this study, no actual JBoss errors were encountered. But the probability of these errors in ASM is set at 0.01 in order to investigate how JBoss errors affect the overall failures of requests.
As displayed in Fig. 17, the error rate of JBoss AS is stable and equal to 0.05, whereas the HTTP rejection rate is increasing due to the increase of web service clients in ASM. The JBoss error rate is computed from dividing the number of JBoss errors due to case probabilities presented in Table 6 (number of activations of TJBossFailed in Fig. 10) by the number of total requests received in the JBoss layer. Also note that, similar to Fig. 16, the HTTP rejections are calculated before the requests enter the rest of the system to be serviced. Thus, as Fig. 17 shows, the rate of HTTP rejections does not change.

In the next experiment, the trend of database rejection rate is shown in Fig. 18 when the service rate of the database layer (DB rate) is manually changed in ASM. The other parameters are based on Table 5. Three different database service rates are considered in this experiment. The first rate is in accordance to the actual tests in the LAN environment, i.e. when the service rate is around 140 requests per second. This number is calculated from the reciprocal of the total time spent in the database (org.hsqldb) (see Table 4). With this service rate, the database rejection rate is zero. Based on ASM, 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. As expected, this shows that with decreasing the service rate of the database, the rejection rate tends to increase. It needs to be noted that the reason for the decrease in the rejection rate for the first few sets of users is that the arrival rates used for these users are extracted from the experiment results for which a sample table is shown in Table 1. From the table, it is observed that the arrival rate is decreasing as the number of users increases. Because of this, as the arrival rate decreases, the rejection rate should decrease in Fig. 18. Furthermore, the total time for servicing the requests in Table 1 is increasing for the first few rows, while the time chosen to run the ASM model is fixed for all arrival rates.
Fig. 18. Database rejection rate based on different database service rates.

Finally, Fig. 19 shows the rate of web service failures estimated by ASM as the web service error probability is increased. The web service error rate is obtained by comparing the number of web service failures against the total number of requests that are not rejected at the time the requests are made (see Fig. 7). The error probability is set by changing the value of case 2 of TWebservice activity in Fig. 8 (also see Table 5). The value of other parameters are the same as those shown in Table 5. The results show that web service error rate increases as the error probability increases but slows down as it becomes high. Furthermore, one might wonder that only 60% of web service requests has failed even though the error probability is 1. The reason lies within Fig. 8, because some of the requests entering the system might end up in the database through activity TJBoss1 without going through TWebService activity (see also Fig. 6b), or some of the requests might be partially serviced through TWebService activity and then loop back to JBoss. The figure also exhibits that varying arrival rates will not have any effect in the error rate trend of web services.

Fig. 19. Web service rate of failures as the probability of web service errors is increased.

5. ASM-based System Reliability

The overall reliability of a system is defined as (Shooman, 2002):

\[ R = e^{-\lambda(t)} \]
where \( z(t) \) is the hazard function representing 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. \( z(t) = \lambda \). Therefore, using \( \lambda \) as the hazard function reduces \( R \) to the following:

\[
R = e^{-\lambda t}
\] (7)

The reliability function in (7) is the probability of an error free operation during \([0, t]\). Also recall that web service reliability has been defined as the probability a web service “successfully” responds within a reasonable period of time; otherwise the web service is assumed to have failed. As the rejected requests will never be able to respond successfully, \( \lambda \) is treated as the rejection rate of HTTP requests. Although one might decide to use a different hazard function, the choice of using \( \lambda \) as the hazard function is motivated by the following:

1) the simplicity of the mathematical calculation since the reliability function in (7) uses a single parameter \( \lambda \);
2) the web service arrivals are independent of each other and are averaged over a number of runs; and
3) the rejection rates from the EM are obtained over a number of experiments and are averaged, which closely match those of the ASM;
4) the SAN models developed in this study use the exponential distribution model in (7) with parameter \( \lambda \) for different activities, for instance to represent the average rate of arrivals obtained from EM.

It needs to be noted that, although the Mobius package is capable of using different distribution models or even mixing distribution models, using exponential distribution has produced a good approximation of rejections with those of the experimental results.

Fig. 20 exhibits the reliability graphs based on the actual HTTP rejection rate obtained from EM and ASM. Each curve in the figure shows reliability of the system under a load test with a specific number of virtual users who send the requests simultaneously. The results indicate that 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. Finally, the reliabilities for high loads drop sharply but tend to be similar. Although the graphs show that the EM and ASM reliabilities closely match each other because their rejection rates are similar, the purpose of the graphs is to simply show the pattern of their reliabilities rather than validating one model by the other.

![Reliability graphs](image)

Fig. 20. Reliability graphs of EM and ASM under different loads.

Fig. 21 shows the same results as in Fig. 20 for three different loads. The graph shows that the ASM 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 in ASM compared to the actual rejection rate of zero from SoapUI. On the same token, the reason for having a linear graph rather than a curve graph of reliability for 200 users is because the rejection rate is so close to zero that can be considered negligible. The same reason holds for 360 users in Fig. 20. In general, the higher the number of users is, the larger the rejection rate will become, leading to steeper graphs. Furthermore, as indicated, the reliability results for EM and ASM are similar because the rejection rates in the two models are very close to each other.
6. Conclusion

This research has presented 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 and performance of the system under various conditions and scenarios. As the accuracy of the 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. Although parameters misconfiguration is 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 modularized SAN model for ASM along with the inclusion of the appropriate configuration parameters results in better understanding of the reliability prediction of the web services. In addition, this research has shown that indeed it is possible to create a simulation model whose performance and reliability analysis closely match that of realistic laboratory experiments that can easily be expanded into larger environments. Once the simulation model is able to correctly map the experimental results, it has been shown that reliability and performance measures can be predicted under different case studies that would otherwise be very difficult to measure in a filed study, such as the example in Fig. 17. Furthermore, it is possible to obtain the reliability effect of the individual layers on the overall reliability by adjusting the activity rate of the service time or the failure rate in the individual submodels.

There are many avenues to extend this research further, some of which are explained below. For instance, 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 that of 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. For example, the actual delay computed between the sample web service and its client using Ping (2012) 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.

The precise estimation of failure rate 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, which affect the HTTP rejection and database rejection rates. The second type of failures originate from other parts of the software such as faults in the implementation of methods or dependencies between classes. As this study has used hypothetical failure probabilities for the layers, estimation of these failure probabilities for each layer requires more investigation in the code and possibly requires data mining approaches to extract the knowledge from the failure repository of each software product or its developers’ forums.

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.

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 ASM 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, composite web services can be treated as a black box or a white-box architecture. Adding the network layer, as indicated previously will make ASM more complex in that each atomic service of a composite one might be involved with different networking conditions such as round trip delays.

As indicated in the Introduction section, there are different types of faults. Understanding and estimating these fault types are challenging tasks. This research did not distinguish between these fault types or consider various failure sources that belong to the same fault type. For instance, Denial of Service (DoS) attack and a logic fault leading to a crash have two different failure sources that can be placed in the same fault category because both have the same impact of failing to provide services. Hence, the same SAN logic can be developed to treat these faults. Consequently, it is an interesting avenue of research to investigate fault types, their impact, or determine which ones are more dominant.

Acknowledgements

This research is funded in part by Department of Defense (DoD)/Air Force Office of Scientific Research (AFOSR), NSF Award Number FA9550-07-1-0499, under the title “High Assurance Software”.

References


A final version of this article has appeared in Journal of Systems & Software. This version has not been fully edited