Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 275 (2011) 123–142

www.elsevier.com/locate/entcs

Web Application Performance Modeling Using Layered Queueing Networks

Yasir Shoaib[1] Olivia Das[2]

Electrical and Computer Engineering
Ryerson University
Toronto, Canada

Abstract

In this paper, a Layered Queueing Network (LQN) performance model is used for studying an Apache-PHP web application with PostgreSQL backend-database. Performance evaluation is done by obtaining load test measurements and by solving the LQN model. Model validation is performed by comparing the model results with the load test results. With average error of 3.77% for throughput and 12.15% for response times the model is shown to capture the web application's performance. Furthermore, performance analysis is done to determine the system configuration which would ease the identified bottleneck resource.

Keywords: Performance modeling and validation, Layered Queueing Networks, Software Performance Engineering, Performance measurement, Load Testing, PHP

1 Introduction

The complexity and nature of web applications require their development to follow a Software Development Life Cycle (SDLC). As users expect web applications to be quick and responsive, attention to system's performance is paid from an early software development stage and followed throughout the SDLC by means of a well-defined systematic process: Software Performance Engineering (SPE) [23]. As part of the SPE process, measurements and performance modeling can be used to study a system. To directly assess if an application will meet the required performance objectives based on available resources, for the purposes of capacity planning [24], a measurement-based approach is adopted. The behaviour of the system under given customer workload can provide results which can help identify performance bottlenecks [25]. Performance modeling, which uses performance models, also finds use

[1] Email: yasir.shoaib@ryerson.ca
[2] Email: odas@ee.ryerson.ca

doi:10.1016/j.entcs.2011.09.009

in capacity planning by means of predicting system's performance and by pinpointing system bottlenecks. Other uses of modeling include capacity provisioning i.e. allocating and preparing resources to handle the demands and finding application configuration parameters that meet the desired objectives [8].

For performance modeling, use of well known Queueing Network (QN) models is very common. However, basic QN models fall short in being able to account for software contention as seen in software servers [28]. The QNs depict software as customers only, whereas software servers behave at times as servers when they are serving their client requests and behave other times as clients themselves when requesting service from other servers (such as database) [28]. If effects of queueing due to software contention are ignored then response times are understated leading to inaccurate results. Furthermore, aspects such as parallel software execution that is seen through creation of a child process from an existing process (fork) cannot be directly represented [29].

Layered Queueing Networks (LQN) [3] analytical models are based on extended QNs and are designed to eliminate the aforementioned shortcomings of QN. In case of Remote Procedure Calls (RPC), the queueing that occurs at lower layers due to software contention is included in the upper layer response time of an LQN model [3]. Furthermore, they can also model nested RPCs [9]. The parallel execution of software and servers which send "early reply" [9] (i.e. some processing at the server is to happen in second phase after reply is sent to the client) can also be modeled [28]. They are well-suited to depict both complex software applications and the hardware resources that these software entities run on [3]. With the ability to incorporate varying degrees of details in the model, LQN performance modeling can easily be integrated with the SDLC. LQNs are ideal for representing the interactions and intricacies of multi-tier application and this work therefore uses LQN for performance modeling.

In this paper, an LQN performance model of a Linux Apache-PHP web application with PostgreSQL backend-database is presented. The solution of the model provides performance metrics such as steady-state throughput and response times. The model results are compared with measurement results derived from load testing the system for model validation. The analysis of the model results is provided and includes identification of the bottleneck resource with mention of an approach that would scale the system by avoiding the bottleneck resource saturation early.

The paper is organized as follows: Section 2 lists related works in the area of web performance modeling. Section 3 gives a brief overview of LQN. Section 4 provides the design details of Web Application. Section 5 describes the Load Testing setup. Section 6 presents the LQN model of the application. Section 7 presents the measurements and model evaluation results with further analysis to ease bottleneck resource. Section 8 presents the conclusions and future work.

2 Related Work

One of the earlier works pertaining to web system performance analysis is done by Slothouber [26] where a four-station open QN model consisting of Client, Web server and Network of a simple file Web server has been presented and analyzed.

Kounev & Buchmann [10] describe a closed queueing model of SPECjAppServer2002 (J2EE) benchmark comprising of Client, Application Server Cluster, Database Server and Production Line Stations. The paper explains how service demands which represent the demands that requests place on the computing resources and serve as inputs to the model were obtained through use of Operational Laws of QNs. The model validation shows high accuracy of performance prediction with average error of 2% for throughput, 6% for CPU utilization and 18% for response time.

Urgaonkar *et al.* [8] present a closed QN model of multi-tier internet applications which considers caching, concurrency limits and multiple session-based class requests. The model is validated through two J2EE applications: RUBiS and RUBBoS.

Liu *et al.* [27] describe a closed QN model of a 3-tiered web application comprising of Apache Web server, Tomcat Application server and MySQL Database server. To model the concurrency limits such as maximum number of concurrently running threads/processes of Apache and MySQL servers, multi-station queues are used.

The above works are very useful representations of web system modeling and their measurements; however, with respect to modeling they encounter the same drawbacks as those of QNs, which LQN overcomes through easily representing large software and hardware systems while also incorporating aspects such as software contention that affects the performance.

Previously, LQN performance models have been used for studying of software and web systems [3,4,5,6,7,18,19,20,21]. Tiwari and Mynampati [7] modeled the SPECjAppServer2001 EJB benchmark as a simple LQN model, validated by measurements obtained from an earlier work by Kounev and Buchmann [17]. Ufimtsev and Murphy [6] model a JavaEE ECPerf benchmark based on LQN EJB templates [5]. Xu *et al.* [21] model a J2EE bank application using LQN templates and performs model validation. In the aforementioned work, the test system for measurements comprises of client load generator, EJB Application server and Database server machines. Tools like JProbe and sar were used for profiling and obtaining usage information of resources such as CPU, network and disk. For tests, the beans were either accessed sequentially or in random order. The validation results for higher client numbers reported errors ranging from 6.2% to 23.9% (sequential) and from 2.1% to 24.5% (random). However, as Urgaonkar *et al.* [8] indicate, most of these works have focused on Java Enterprise applications. In this work we utilize the versatility of LQN to study a PHP web application.

It is interesting to realize that many works have been done with regards to modeling but very few compare the model results with actual measurements. Alongside

LQN modeling, only Tiwari and Mynampati [7] and Xu *et al.* [21] from above use measurements to perform model validation, and these are considered as similar pieces of works to ours.

Also, similar to our work, Dilley *et al.* [22] and Pastsyak *et al.* [2] have used LQN to model a CGI Web server and an Apache-PHP-MySQL system, respectively. The CGI Web server model [22] incorporates serving of both dynamic and static contents by the server, however no database tier is considered in the study. The response time from the model evaluation is found to match closely with response time measurements over a period of two months. Measurement data collected through custom instrumentation serve as input parameters to the LQN model and are also used for model-validation. The web system LQN model by Pastsyak *et al.* [2] is evaluated for 40 users and compared with the load test results, which show the successful prediction of performance by the model. The system consists of two Web servers, a Load balancer and a Database server. Other performance modeling formalisms such as Stochastic Process Algebra (SPA) and Stochastic Petri Nets (SPN) have also been evaluated in this paper.

However, in contrast to these two aforementioned papers ([22,2]) our work uses LQN activities to define in detail the web software entities and the precedence of interactions between them.

3 Layered Queueing Networks

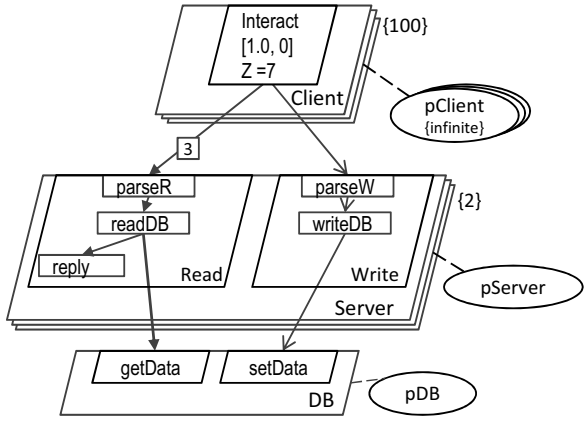

Fig. 1. Example LQN Model

An example LQN model is shown in Figure 1. The outermost parallelograms represent the *Tasks* and the parallelogram within the *Tasks* corresponds to the *Entries*. *Entries* are akin to customer classes of QNs [9]. Furthermore, each *Entry* can be sub-divided into smaller units of work known as *Activities* [9], represented by rectangles. *Processors* are shown as oval shapes. Multiplicity of *Tasks* signifies multiple threads of a software process shown in the figure within braces.

Communication between the *Entries* of *Tasks* can be of three types: *Synchronous*, *Asynchronous* and *Forwarding*. In *synchronous* communication the re-

questing task (client) blocks until a response is received from a server task. In *asynchronous* interaction the client task does not block after sending a request. In *forwarding* communication, the server (task A) that received the request forwards the request to another task (task B). Task A at this point starts execution in phase two and when task B is done processing its request, it sends a response back to the client after which task B will begin its execution in phase two. In Figure 1, *synchronous* calls are shown with 'normal' arrows and *asynchronous* calls are shown as 'vee' arrows.

Figure 1 represents a system where there are 100 *Clients* interacting with a single-processor, two-threaded *Server* process. The *Client* initiates the requests performing the *Interact* operation. The data is stored in the *DB* database where data access and manipulation from the *Client* happens through three synchronous Read(s) and an asynchronous Write operation on the Server. Once the *Client* receives a response back from the *Server*, it thinks for 7 seconds (Z=7) and then initiates another set of requests for *Read* and *Write*, repeating this process infinitely. Each entry has an associated service time per phase. Due to lack of space, only the *Client* task's service times are presented shown within square brackets. The arrows show the direction in which the requests are made. Unless specified otherwise, the frequency of interactions is one.

Inputs to the LQN model are scheduling discipline of the hardware resources, customer workload intensity and the service demands of the customers for the model components at each phase. The main performance metrics available from LQN model evaluation are steady-state throughput, response times, and utilizations of the modeled components.

4 Web Application

In this section, the web application whose performance has been studied through measurements and LQN performance modeling, is introduced. The Process-Flow Diagram of the application is also presented, which will help in understanding of the system for modeling purposes.

4.1 Overview

The web application under study, MyBikeRoutes-OSM, is a repository of bicycle routes that allows the users to create and share bicycle routes from anywhere in the world. Users can search for the best bike route between given source and destination locations, i.e. best path search. The best path search functionality is currently a prototype where the search is essentially performed using the roadways instead of the bicycle routes. This however does not affect the study as our main focus is towards the system's performance. The application uses OpenLayers JavaScript API to display OpenStreetMap [3] (OSM) maps [11]. In the particular case of this application, the maps are generated on a Browser through an order-

[3] http://www.openstreetmap.org/

ing of pre-rendered map tiles/images residing on the server. PostgreSQL database functions as the storage facility of the bike routes data and provides best path routing features, where the routing functionality is made available by the pgRouting [4] project [12]. For displaying bike routes data or for route-search, OpenLayers at client-side basically communicates with the PostgreSQL backend-database through an Apache-PHP server.

MyBikeRoutes-OSM derives from the MyBikeRoutes [5] web application [13], which is functionality-wise similar but an online web replica of its derivative project. The apparent difference is the use of mapping services from Google Maps API [6] [14] to display Google Maps and route information by MyBikeRoutes website, where the bicycle routes data is housed in a MySQL backend-database. Furthermore, the best path search on the MyBikeRoutes site actually uses bike routes for the search. Here the client-side JavaScript interacts with Google Maps API, whereas the MyBikeRoutes-OSM project uses pgRouting at the Database server layer to provide the search functionality.

MyBikeRoutes and MyBikeRoutes-OSM projects are both recent development endeavours and there are subsequent enhancements of these applications that have been envisioned for future. However, before proceeding with involved development efforts, it is best to know about system bottlenecks and find the performance indicators following the SPE process. Thus, we have analyzed the performance of MyBikeRoutes-OSM application in this work.

4.2 Process Flow Diagram

Figure 2 displays the process flow of the web application. As a user visits the website from their browser, the bicycle routes are displayed on the OSM map. Next, the user may draw bicycle routes or perform a best path search between their chosen source and destination locations. If a search is performed then the best path algorithm is run and the best path is displayed on the map with the option to then save the results. On the other hand, the user may draw and store bicycle routes to share them with other users.

5 Load Testing

Load Testing is a measurement-based approach where behaviour of the System Under Test (SUT) is studied under different workloads. The outputs include actual throughputs and response times. In our case, load testing is used to study the system's performance, obtain model service demands and validate performance model results. For this work, a free open-source load tester, JMeter [7] [15], was used for load testing.

[4] http://www.pgrouting.org/
[5] http://www.mybikeroutes.com
[6] http://code.google.com/apis/maps/index.html
[7] http://jakarta.apache.org/jmeter/

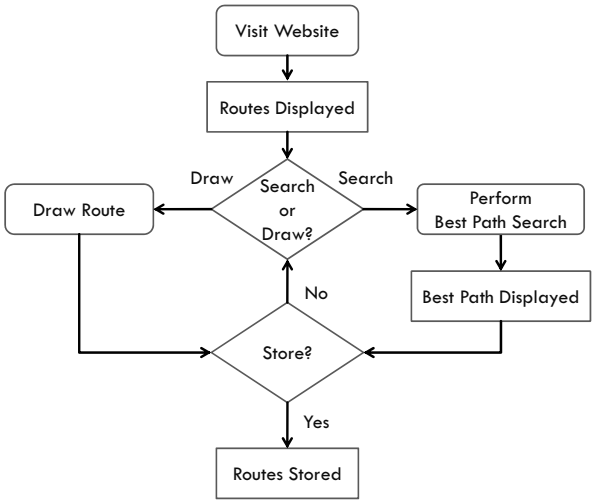

Fig. 2. Process Flow Diagram: MyBikeRoutes-OSM

5.1 Configuration and Topology

Figure 3 outlines the Remote Testing setup (Master-Slave configuration) for the measurements, where the Jmeter-Master machine initiates the test while the Jmeter-Slave is the actual client machine that simulates the virtual users for sending requests to the Server under test. Each of these machines have identical physical configuration. The Server runs the Apache-PHP Application Server and PostgreSQL Database.

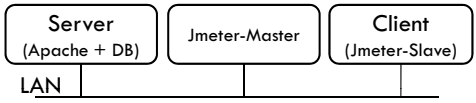

Fig. 3. Load Test Topology

The following are the machine configurations and also the details of software used: Pentium 4 3.4 GHz (32-bit), 993 MB RAM, Ubuntu 10.04, 1000 Mb/s Network, Apache Prefork 2.2.14, PHP 5.32, PostgreSQL 8.4, JMeter 2.3.4.

5.2 Test Plan

As a part of the SPE process, it is essential to determine the performance-intensive scenarios of the application early [23]. To achieve this, first, all the HTTP requests generated by Mozilla Firefox browser while navigating the MyBikeRoutes-OSM application were recorded in JMeter. A load test with a single user was run to determine the performance intensive requests, i.e. requests which had high response time. From the previous step the base scenario (or base test plan in JMeter) was created by removing non-critical requests. Figure 4 shows the scenario's sequence diagram for one user session. The figure represents the actions of a *User* who initi-

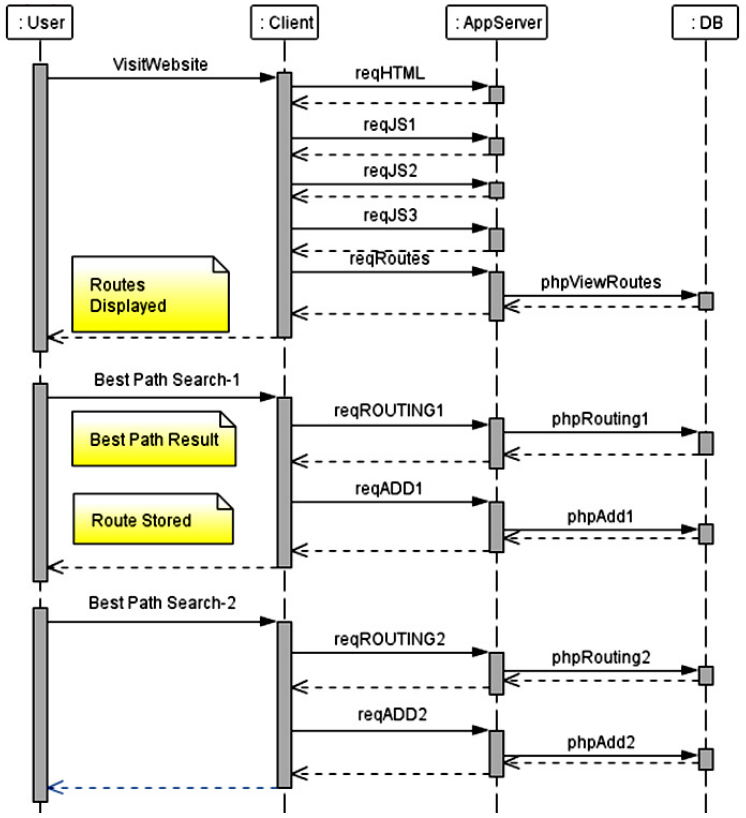

Fig. 4. Base Scenario Sequence Diagram

ates the web communication. The user actions are shown for clarity, however, the actual load test plan consists of the HTTP requests of the Client/Browser only. The *AppServer* or Application Server represents the Apache-PHP server and the *DB* corresponds to the backend-database.

Based on Figure 4, in all, there are nine request classes sent to the Application Server in one complete user session and the initial five requests consists of an HTML file, three JavaScript files, and bike routes data, representing the requests that are sent on visiting the site. In this first group, except for the last request of bike routes data which interacts also with the Database, all the other requests just interact with the Application Server. After this, one request for best path search and another to save the best path results is run. Very similar requests are again made for the second bike routes search, however with different start and end destinations. This final set of requests, communicate with both the Application Server and the Database.

For the load tests, a think time of 7 seconds was added before calling *reqHTML*. This scenario which includes think time of 7 seconds will be referred as *Base-Scenario* from this point onwards. The load tests were run for a duration of 1800 sec for each N Virtual Users, where $N = 1, 2, 4, 6, 10, 20, 30, 40, 50, 80, 120$. To make sure that the Client machine could simulate the users and was not the bottleneck, JMeter tests were run in command-line mode with summary reporting.

Furthermore, by applying *Little's Law* to the JMeter results it was easily verified that the number of users active in the SUT were close to the number of virtual users initiated by JMeter [30]. The results obtained from the load tests were the session throughput and average session response time. These results are presented in section 7.

6 LQN Performance Modeling

The sequence diagram described earlier in Figure 4 is used for LQN model creation. The following section explains the LQN model.

6.1 Base-Scenario Model

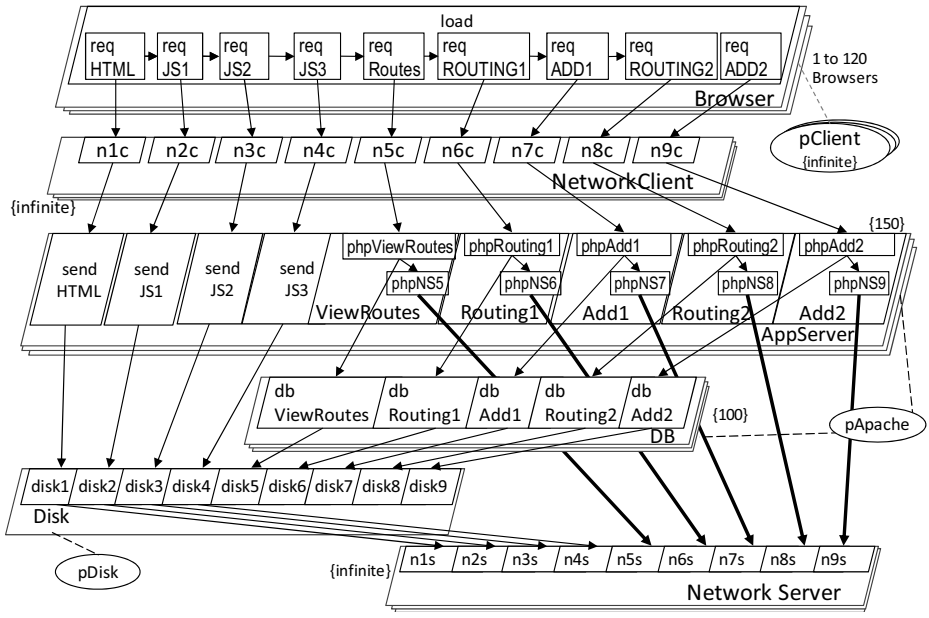

Fig. 5. Base-Scenario LQN Model

Figure 5 shows the LQN model of *Base-Scenario* in a load test. *Reply* activities are not shown to ease in understanding of the figure. There are *N Browsers* running on Infinite Servers (*pClient*). The model is evaluated separately for each value of *N*. This is a closed model where once a request session is completely processed, the customer is sent back to begin a new set of requests after waiting for a given think time. Since the processing at the client machine did not include browser rendering or page generation in the load test, the service demands for the *Browser* entries in the model are set to 0. Each *Browser* sends requests to the *AppServer* task through the entries of the *NetworkClient*, which represents the delay incurred when sending messages through the LAN from the *Browser* to the *AppServer*. Similarly, *NetworkServer* represents delay due to sending reply from the *AppServer*. The two

network tasks are modeled as infinite threads running on Infinite Servers. Based on the *MaxClients* [31] directive of Apache server, which sets a limit to the maximum number of processes that are available to handle client requests concurrently, the *AppServer* multiplicity is set to 150. Similarly, based on *max_connections* [16] parameter of PostgreSQL database, which specifies the maximum concurrent connections to PostgreSQL database, the *DB* task multiplicity is set to 100. Both *AppServer* and *DB* tasks execute on the same uni-processor (*pApache*), which has Processor Sharing (PS) scheduling discipline. The disk is modeled by the *Disk* task, which has nine entries. Each entry relates to the nine requests issued by the *Browser*, i.e the first request (*reqHTML*) has its disk service demand provided by *disk1* entry, then *reqJS1* has the disk service demand provided by *disk2*, and following the same pattern for other requests.

An assumption regarding *Disk* entries has been made in the model. For a request, only one interaction with the *Disk* entries is assumed, i.e. if both the *AppServer* and *DB* tasks perform certain number of disk I/Os for a particular request, then only one call to the *Disk* at the end of the nested interaction is depicted in the model. In this case, the service time of the respective *Disk* entry is for one visit only found by multiplying the number of disk I/Os and the average service time at the disk. For finding the number of disk I/Os, the *Forced Flow Law* (refer section 6.2) has been used, where both the system throughput and disk I/O throughputs have been found through measurements. Similar ideas have been used previously by Liu *et al.* [27] and by Kounev & Buchmann [10] for the database-tier in their works. Like-wise this work also makes the same assumption as the *Disk* entries for the *DB* entries.

There are five request classes that require database access, i.e. *reqViewRoutes*, *reqROUTING1*, *reqROUTING2*, *reqADD1* and *reqADD2*. There is one SQL query executed for *reqViewRoutes*, which just involves retrieving the routes data. Both the routing requests (*reqROUTING1* and *reqROUTING2*) require running three queries relating to start and end points and finally the routing search. There is one query executed to insert a route for *reqADD1* and *reqADD2* requests. However, as mentioned in the earlier paragraph, each entry of *DB* task has only one visit made to it in the model, i.e. *dbViewRoutes*, *dbRouting1*, *dbAdd1*, *dbRouting2* and *dbAdd2* have only one visit from the upper *AppServer* task layer. Based on this, considering *dbViewRoutes*, the service times of each SQL query executed for *dbViewRoutes* was summed and presented finally as the service time of the *dbViewRoutes* entry. Similarly, service times for other entries of the *DB* task have been found. The reader may refer to section 6.2 for further details regarding the calculations of service demands for the entries. In the following paragraph, the sequence of execution that the model represents is explained in detail.

In the model, the *reqHTML* activity initiates the *Browser* requests to which the *AppServer* responds back after execution of *sendHTML*. Any disk operations by *sendHTML* happens using the *disk1* entry at the *pDisk*. The network delay of *NetworkClient* is also incorporated as the request goes from *reqHTML* through the *n1c* entry to the *sendHTML* entry. Following *reqHTML*, there are three JavaScript

(*reqJS1*, *reqJS2*, *reqJS3*) and *reqRoutes* requests that are sent sequentially, completing the actions of a web site visit. In a similar pattern, the other four remaining requests are also sent in order and processed. Some *AppServer* activities need to retrieve or amend data on the *DB*, forming a nested interaction, where the *AppServer* only sends reply back to the Browser when the *AppServer's* request has been responded back by the *DB*. Each *AppServer* reply also guarantees incorporation of the Network delay in the response time by interacting either directly or indirectly through a *Disk* with the *NetworkServer* before sending any reply. This models a complete session of the *Base-Scenario* of the application. The service demands for the *Base-Scenario* model are discovered by applying the *Utilization Law*, details of which are presented in the following sections.

6.2 Discovering Service Demands

If the utilizations and throughputs of system resources are available – from load testing or monitoring – the *Utilization Law* can be applied to derive service demands. Previously, Kounev & Buchmann in [10] have used the *Utilization Law* to find service demands and this work uses the same approach. For Linux, utilities such as *iostat* and *sar* are useful monitoring tools to accomplish this job. The following paragraph gives a brief overview of the calculations related to queueing theory that involve finding service demands using the aforementioned ideas.

Consider a station i in a queueing system and a request class c, with average utilization of the station due to requests of c as $U_{c,i}$, average system throughput as X_c, and throughput at station i as $X_{c,i}$, then the service demand ($D_{c,i}$) at the station due to the request class can be calculated by applying *Utilization Law* as follows, $D_{c,i} = U_{c,i}/X_c$ [29]. Service demand is also the product of number of visits ($V_{c,i}$) and the average service time per visit ($S_{c,i}$), i.e. $D_{c,i} = V_{c,i} * S_{c,i}$. Here, $V_{c,i}$ represents the number of visits made to a station for each request of a class-request [29,32]. If $X_{c,i}$ and X_c are known then $V_{c,i} = X_{c,i}/X_c$ (*Forced Flow Law*) [29]. Thus, given X_c and $U_{c,i}$ then $D_{c,i}$ can be derived. And then if $V_{c,i}$ is found from *Forced Flow Law*, then $S_{c,i}$ can be calculated where both $V_{c,i}$ and $S_{c,i}$ serve as inputs to the LQN model. For this work, X_c was found using JMeter, and $U_{c,i}$ was found from sar utility. $X_{c,i}$ is required for disk I/O service demands and also found from running sar.

To determine service demands, the first step was to create separate JMeter tests with one user load having no think time for each request class of *Base-Scenario*, thereby creating nine tests. Since there is only one user, the contention due to queueing is at the lowest. Each test was run for duration of 900 seconds while the SUT was monitored using sar. Utilization output of sar includes iowait% and idle%, which specify the % of time the CPU is waiting for IO processing and the % of time CPU is not processing, respectively. The CPU utilization can then be obtained by subtracting the idle% from 100. The throughput obtained from JMeter for each test and the CPU utilization obtained from sar was used to find the total service time of each request at the SUT processor including service time for disk I/O. Furthermore, sar was used to find the average disk service time, and the disk tps (throughput

per second). The number of disk I/Os was found by dividing the disk tps by the JMeter throughput, thereby using the *Forced Flow Law*. As mentioned earlier in section 6.1, only one visit from the upper layers to the disk entries is assumed in the model, therefore, the average disk service time was multiplied by the number of disk I/Os to obtain the normalized service time for one disk I/O, which is the input for the *Disk* entries at the *pDisk*. The service time found for a request's disk I/O was subtracted from the total service time at the SUT processor, thereby giving the service time for the request class at the *AppServer* task at the *pServer* processor. Note that if the CPU utilization was obtained earlier by subtracting both idle% and iowait% from 100 then there would be no need to subtract the request's disk I/O from the total service time at the SUT. The total Network delay was found by subtracting the total service time at the SUT by the JMeter response time for the request. Note for obtaining the *NetworkClient* and *NetworkServer* service times, the Network delay is divided by 2. An example of above follows.

Example1: For the first request class (*reqHTML*) load test, the average CPU Utilization was 59.18%, i.e. the % of time the CPU was not idle. The JMeter throughput was 61.44 requests/s and the average response time was 11 ms. Applying *Utilization Law*, the total service demand at the SUT processor and disk would be $0.5918/61.44 = 9.63$ ms. The disk I/O service time was found to be 0.36 ms and the disk tps was 1.69. Using *Forced Flow Law*, the average number of disk visits was $1.69/61.44 = 0.02751$. Therefore, the normalized service time for the *disk1* entry is $0.02751 * 0.36 = 0.01$ ms. The service time at the *sendHTML* entry is 9.63 - 0.01 = 9.62 ms. Based on this the total Network delay is 11 - 9.63 = 1.37 ms, and therefore the service time for *n1c* and *n1s* entries is $1.37/2 = 0.685$ ms.

For requests that involve database calls, the following approach was adopted. For request classes that have very small service times of about 1 ms at the *DB* task, such as start and stop point queries of the routing search (4 of such queries in total), the EXPLAIN ANALYZE [16] query command of PostgreSQL was used. The latter command provides the runtime details of a query that is to be evaluated. The EXPLAIN ANALYZE command was executed five times for each start and stop queries and the average runtime was used as the service time.

For the other longer queries, i.e. the two routing searches, the viewing routes query, and the two insertion of new routes queries, JMeter tests with one user load and no think time were run for a duration of 900 sec. Applying *Utilization Law*, and following similar calculations as for *Example1* the service demands were found.

Note that only one visit is made from the *AppServer* layer entries/activities to a corresponding database query based on the web application, therefore for a "database query" the visit count is one and the service demand is equal to the service time. However, it is key to realize that based on the web application, the Application Server may call multiple database queries for a particular request class, i.e. for *reqROUTING1* request, one visit is made for start point, one for end point and finally one for routing, thereby a total of three queries are called. In this case, the assumptions regarding the model have been made as clearly outlined in section 6.1. For the *reqROUTING1* request example just considered, the service

Request class	AppServer (ms)	DB (ms)	Disk (ms)	Network (ms)
reqHTML	9.62	-	0.01	1.37
reqJS1	0.85	-	0.01	0.14
reqJS2	4.8	-	0.01	1.19
reqJS3	0.64	-	0.02	0.34
reqViewRoutes	95.55	29.38	0.06	2.01
reqROUTING1	211.67	252.72	0.28	0.0002*
reqADD1	53.42	0.43	0.32	0.0002*
reqROUTING2	186.16	52.09	0.23	0.0002*
reqADD2	53.43	0.41	0.33	0.0002*

Table 1
Service Demand Parameters for Base-Scenario Model

times of the three queries is summed to form one entry, *dbRouting1*.

Based on above methodology and calculations presented, the service times for each entry of the *Base-Scenario* model are shown in Table 1. Here, the *reqHTML* service time on the *AppServer* is 9.62 ms, which corresponds to the *sendHTML* entry service time. Similarly, *reqHTML* service time on *Disk* is 0.01ms corresponding to the *disk1* entry. The service time on Network entries (*n1c* and *n2c*) due to *reqHTML* is 1.37/2 =0.685ms each. Similar pattern follows for other request classes. Note that the service times for Network entries with (*) were found to have negative service times. The *reqROUTING1*, *reqADD1*, *reqROUTING2* and *reqADD2* classes had network service times of -0.00067, -0.00117, -0.00048 and -0.00117 (ms) respectively. This is considered as an anomaly and therefore, the network service times for each of these have been changed to 0.0002 ms, i.e. *n1c* and *n1s* each have service times of 0.0001 ms. Future work would include identifying the reason for such an anomaly.

7 Results

In this section the performance metrics obtained from the model evaluation are presented along with the measurements in both tabular and graphical formats for model validation. Based on the results the performance objective for the *Base-Scenario* is defined. Furthermore, performance analysis is done to achieve performance objectives.

7.1 Base-Scenario Performance Results

The model evaluation has been carried out for up to 150 users. This number was chosen because the *AppServer* task multiplicity is set to 150, which signifies the maximum number of running Apache processes. The model was evaluated on a dual Intel Xeon 3.0 Ghz CPU with hyper-threading enabled, running on 2 GB of RAM. Taking an average of 10 runs for 120 users case, the *Base-Scenario* model was solved in 376ms while utilizing 47% of CPU.

Table 2 shows the throughputs and response times from *Base-Scenario* model evaluation and compares them with the measurements. The relative error% for the throughput from the model match very closely with the actual results with an average error of 3.77%. The response time results show that for 6 users the error%

	Throughput (sessions/s)			Response Time (s)		
Users	**Load Test**	**LQN**	**Error%**	**Load Test**	**LQN**	**Error%**
1	0.12558	0.12567	0.07%	0.951	0.957	0.63%
2	0.24844	0.24864	0.08%	0.951	1.044	9.78%
4	0.49897	0.48027	3.75%	0.987	1.329	34.65%
6	0.73930	0.68434	7.43%	1.077	1.768	64.16%
10	0.92838	0.96871	4.34%	3.719	3.323	10.65%
20	1.03278	1.09454	5.98%	12.156	11.273	7.26%
30	1.05149	1.08406	3.10%	21.182	20.672	2.41%
40	1.02687	1.07721	4.90%	31.205	30.135	3.43%
50	1.04190	1.07176	2.87%	39.629	39.649	0.05%
80	1.01935	1.06507	4.49%	68.317	68.118	0.29%
120	1.01470	1.06017	4.48%	105.873	106.182	0.29%
150		1.05922			134.623	
AVG ERROR			3.77%			12.15%

Table 2
LQN Model Results - Base-Scenario

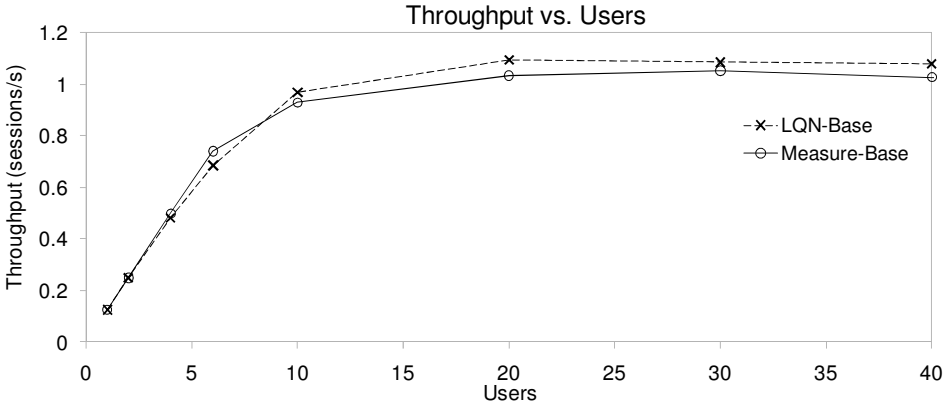

Fig. 6. LQN Base-Scenario (Throughput vs. Users) - 40 users

is higher than other user counts, however as also seen the error% is very close to accurate for low numbers of 1 and 2 users and also for higher numbers of 10–120 users. For response times the average error is 12.15%.

Figure 6 and Figure 7 show the graphs for Throughput vs. Users, and Response Time vs. Users, respectively for up to 40 users from results presented above in Table 2. As seen from both these graphs, the model closely follows the behaviour of the system's performance and continues to do so for higher user counts as well. Figure 8 shows the *pApache* CPU utilization from LQN evaluation that approaches about 100% utilization for more than 20 users. This is the hardware bottleneck and saturates before other resources.

Further model analysis has been done between 5 and 20 users to pinpoint the throughput saturation point and visualize the system behaviour closely in this range. Figure 9 and Figure 10 shows the throughput and response times graphs for up to 20 users. From the model analysis, the throughput initially rises although with a settling rate of increase with the maximum throughput of about 1.1 sessions/s occurring at 19 users after which the throughput continues to remain stable, de-

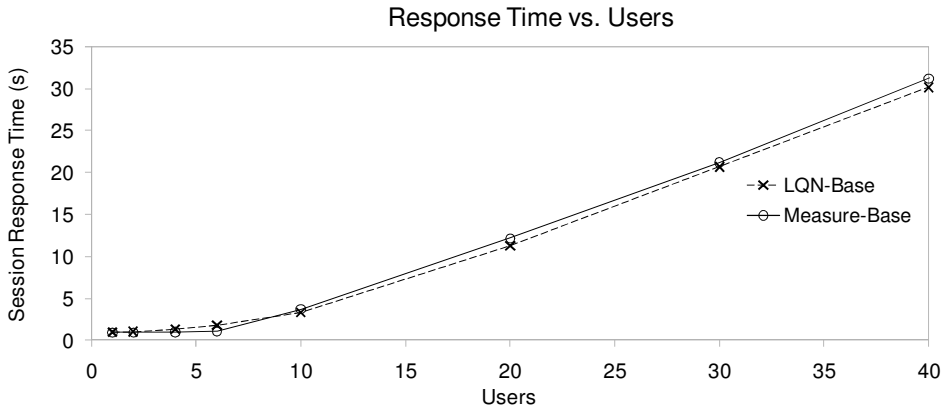

Fig. 7. LQN Base-Scenario (Response Time vs. Users) - 40 users

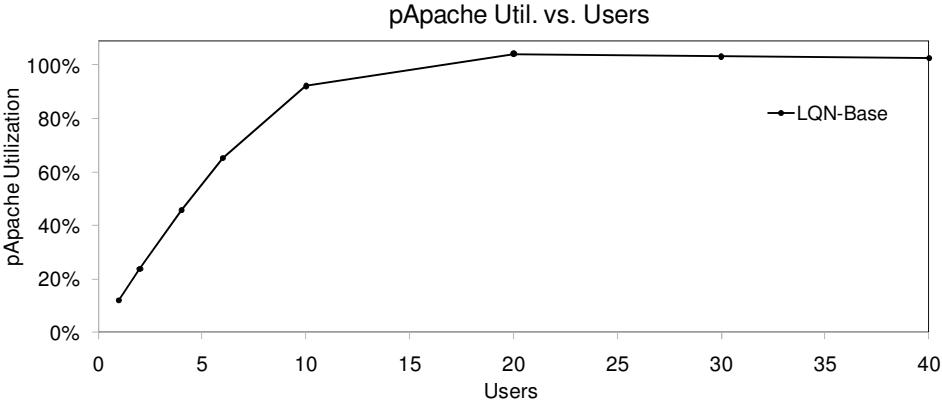

Fig. 8. LQN Base-Scenario (pApache Utilization) - 40 users

creasing very slightly for increasing users as seen from Figure 6. The response time remains stable initially until it increases and continues to rise steadily for increasing users. The model shows a higher response times in comparison to measured results until 7 users after which the model continues to follow in parallel the measurements just slightly understating the response times.

Lazowska *et al.* [29] provide rough error percentages for model validation. For multiple classes throughput error between 5% and 10% and for response time error between 10% and 30% are considered acceptable. For our model results, all the throughputs are very accurate with average error lower than 5%. Considering response times, the exceptions are 4 and 6 users which have high error% however in real situations the system will probably witness higher number of users. One possible reason for this shortcoming is that CPU cache hit ratio is higher with lower user counts and CPU caching has not been considered in the model.Based on average response time error of about 12%, the model represents the measurement results. After the model validation, the next steps will be to determine the improvements

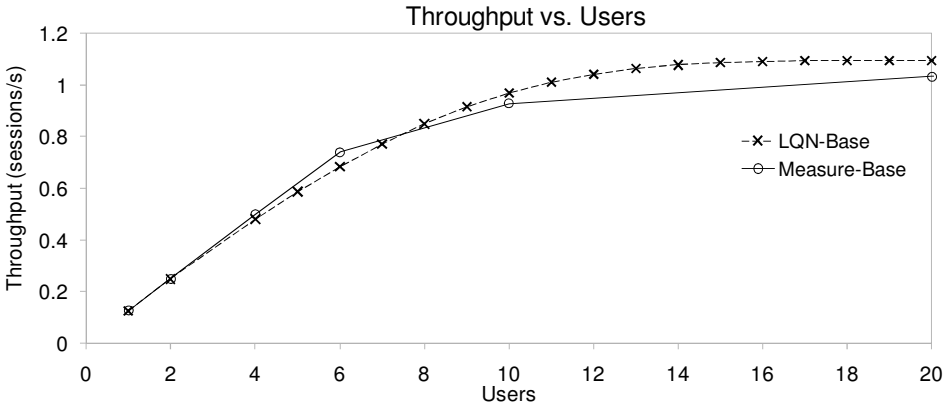

Fig. 9. LQN Base-Scenario (Throughput vs. Users) - 20 users

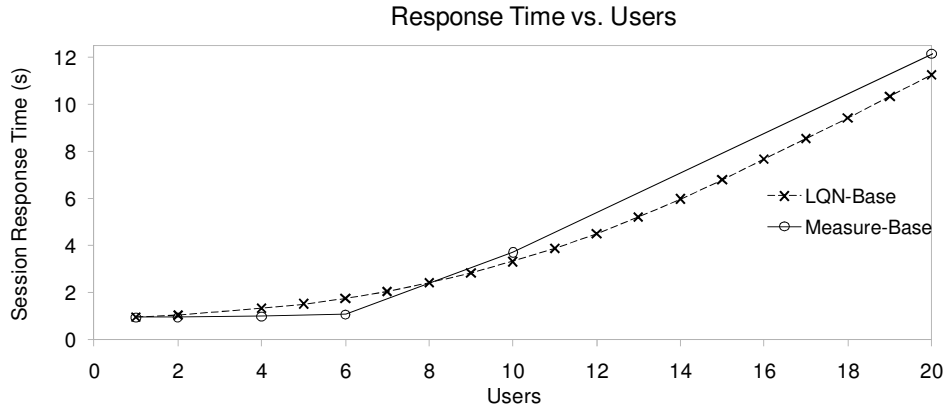

Fig. 10. LQN Base-Scenario (Response Time vs. Users) - 20 users

to the system.

The above results are helpful but they show that web application will not be able to support large number of users at satisfactory response times, showing poor performance. Based on the functionality provided by the web application, a reasonable performance objective is to sustain 40 to 50 users with a session response time of 12 seconds without think time. Considering such as objective, the web application performance has to be improved. Through previous analysis the bottleneck resource has already been identified to be *pApache* CPU. Next, we use intuitive modifications to the model such that performance objectives are met.

7.2 Attaining Performance Objectives

In this subsection different configurations that may achieve the performance objectives defined earlier have been evaluated and the results from the evaluation are presented to determine how successfully the objectives can be met.

7.2.1 Overview

To achieve performance objectives, designs that ease the bottleneck have to be determined. From previous analysis it is found that the *pApache* server is the bottleneck for the MyBikeRoutes-OSM web application. To scale the system, options include adding threads, using a multi-processor system or using copies (replicas) of the server [20]. Since the bottleneck is hardware, addition of threads of *AppServer* task or the *DB* task will not be helpful. We evaluate the performance model for two possible solutions: (i) Multi-processor machine and, (ii) Separating *AppServer* and *DB* task into separate identical machines. Following are the details of the modifications:

(i) Multiprocessor *pApache*: The models have been evaluated for processors with multiplicity of two and four, which are referred as *Base-Scenario-m2* and *Base-Scenario-m4* models respectively. (Note that LQNS did not support PS scheduling for *pApache* multiprocessor. For two and four multiprocessors, First Come First Serve discipline was used).

(ii) Separate machine for Database: Instead of having both Application and Database software servers run on the *pApache* machine, the database-tier can be deployed on a separate and identical machine. The changes to the model include creating a new *pDB* processor which will be hosting the *DB* task and its entries while the *Disk2* task on *pDisk2* will now handle the disk I/O for the *DB* task. The previous *pDisk* is renamed as *pDisk1*, and the Disk task is renamed as *Disk1* which will handle disk I/O for *AppServer*. The model will be referred as *SeparateDB-Scenario*. There are two assumptions made for this model. One is that service times for the disk I/O is divided by two to derive the service times of each entries of *Disk1* and *Disk2* tasks. Second, no network delay between the *pApache* and the *pDB* machines are considered. In a LAN environment the delay would not be very large, however, for very detailed study significant delays should be considered.

7.2.2 Performance Analysis

Figure 11 and Figure 12 show the throughput and response time graphs comparing the *Base-Scenario*, *Base-Scenario-m2*, *Base-Scenario-m4* and *SeparateDB-Scenario* models. As depicted, *Base-Scenario* can sustain 22 users within a response time of 12 seconds, whereas the separate database can sustain 30 users, the dual processor 40 users, and quad-processor 80 users within this response time with respective throughputs of about 1 sessions/s, 1.5 sessions/s, 2 sessions/s and 4 sessions/s. From the model evaluations both dual and quad processors meet the performance objective set earlier. Although having a dedicated separate machine for the database was an attractive choice, the results suggest towards choosing from multiprocessor system to scale the system. Choice between dual or quad processor would be based on cost-benefit analysis, where it is to be determined if it sustaining 80 user with a higher throughput justifies spending extra on a quad-processor. For this work, we choose quad-processor considering that sustaining the increasing future demands on the web application will require more processing power.

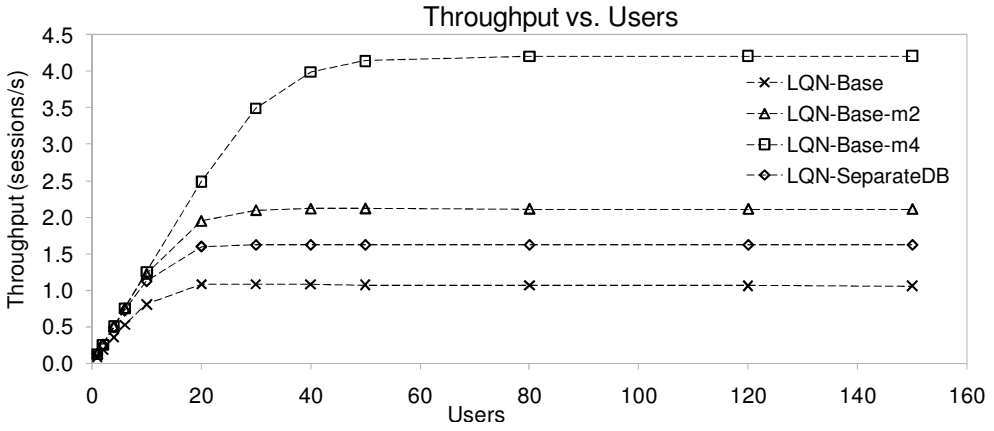

Fig. 11. Performance Analysis (Throughput vs. Users)

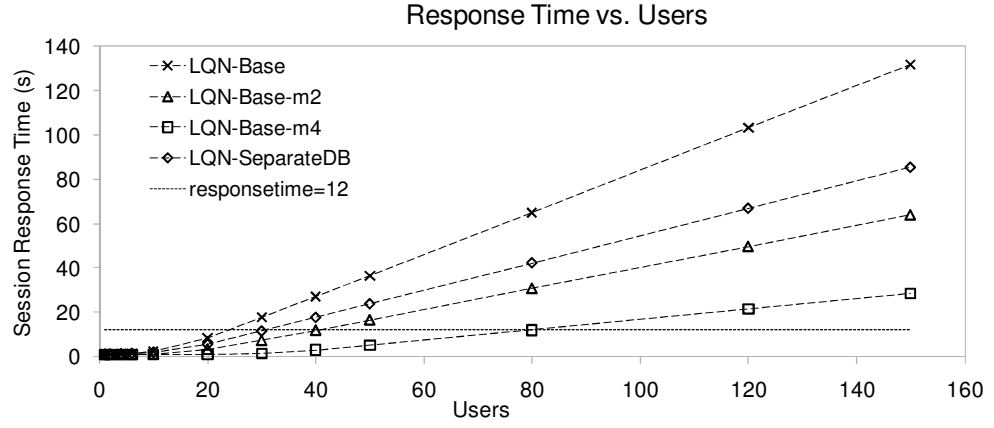

Fig. 12. Performance Analysis (Response Time vs. Users)

8 Conclusions

A LQN performance model of MyBikeRoutes-OSM web application has been introduced in this work. For the base scenario described, performance results have been obtained from both measurement-based and model-based evaluations, the methodologies of which have clearly been explained and the results analyzed. JMeter was used for load testing and *Utilization Law* was applied to obtain service demand parameters. The model is validated by comparison with the load test results. With average error of 3.77% for throughput and 12.15% for response times the model is shown to capture the web application's performance. The analysis of the base model shows that the processor running the Application Server is the hardware bottleneck. To ease the bottleneck such that desired performance objectives are satisfied, modeling is used to represent the configuration options. The best configuration is found to use a multiprocessor machine for Application Server instead of having separate machines for Application and Database servers.

Future work includes incorporating caching in the performance model. Also, estimation of network delay using network utilization data from OS is a strong candidate for future research.

One of the strengths of adopting LQN model based approach as seen from this work is the short time required for model creation, manipulation and evaluation with good accuracy. Before modification of any system component, the effect of the change can be predicted and a decision can be reached just through quick model evaluation.

References

[1] T. A. Israr, D. H. Lau, G. Franks, and M. Woodside, "Automatic generation of layered queuing software performance models from commonly available traces," In *Proceedings of the 5th international Workshop on Software and Performance*, 2005, pp. 147-158.

[2] A. Pastsyak, Y. Rebrova, and V. Okulevich, "Performance Prediction of Client-Server Systems By High-Level Abstraction Models," In *4th Software Engineering Conference (Russia) 2008 (SEC(R) 2008) (Oct 23-24 2008)*, 2008. [Online]. Available: http://2008.cee-secr.org/en/etc/secr2008_alexander_pastsyak_performance_prediction_of_client-server_systems.pdf.[Accessed Jan 16, 2011].

[3] G. Franks, P. Maly, M. Woodside, D. C. Petriu and A. Hubbard, *Layered Queueing Network Solver and Simulator User Manual*, Real-time and Distributed Systems Lab, Carleton University, Ottawa, 2005. [Online]. Available: http://www.sce.carleton.ca/rads/lqns/LQNSUserMan.pdf. [Accessed Dec 28, 2010].

[4] T. Liu, S. Kumaran, and J. Chung, "Performance Engineering of a Java-Based eCommerce System," In *Proceedings of the 2004 IEEE international Conference on E-Technology, E-Commerce and E-Service (Eee'04)*, 2004, pp. 33-37.

[5] J. Xu and M. Woodside, "Template-Driven Performance Modeling of Enterprise Java Beans," In *Proc. Workshop on Middleware for Web Services*, Enschede, Netherlands, 2005, pp. 57-64.

[6] A. Ufimtsev, and L. Murphy, "Performance modeling of a JavaEE component application using layered queuing networks: revised approach and a case study," In *Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems*, 2006, pp. 11-18.

[7] N. Tiwari, and P. Mynampati, "Experiences of using LQN and QPN tools for performance modeling of a J2EE Application," In *Proc. of Computer Measurement Group (CMG) Conference*, 2006, pp. 537548.

[8] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "Analytic modeling of multitier Internet applications," *ACM Trans. Web*, vol. 1, no. 1, pp. 1-35, May 2007.

[9] G. Franks, "Performance Analysis of Distributed Server Systems," PhD Thesis, Report OCIEE-00-01, Carleton University, Ottawa, Ontario, Canada, December 1999.

[10] S. Kounev, and A. Buchmann, "Performance modeling and evaluation of large-scale J2EE applications," In *Proceedings of the Computer Measurement Group's International Conference (CMG'03)*, 2003, pp. 273-283.

[11] M. Haklay and P. Weber, "OpenStreetMap: User-Generated Street Maps," *IEEE Pervasive Computing*, vol. 7, no. 4, pp. 12-18, Oct 2008.

[12] M. F. M. Firdhous, D. L. Basnayake, K. H. L. Kodithuwakku, N. K. Hatthalla, N. W. Charlin and P. M. R. I. K. Bandara, "Route Advising in a Dynamic Environment A High-Tech Approach," in *Innovations in Computing Sciences and Software Engineering*, T. Sobh, and K. Elleithy, Eds. 2010, pp. 249-254.

[13] Y. Shoaib, N. Vasandani, A. Sinha, and A. Goel, "MyBikeRoutes.com," 2008. [Online]. Available: http://www.mybikeroutes.com. [Accessed Jan 16, 2011].

[14] G. Svennerberg, *Beginning Google Maps API 3*, 2nd edition, Apress, 2010.

[15] E. Halili, *Apache Jmeter*, Packt Publishing, 2008.

[16] N. Matthew, and R. Stones, *Beginning Databases with PostgreSQL*, Apress, 2005, p. 314.

[17] S. Kounev and A. Buchmann, "Performance modelling of distributed e-business applications using Queuing Petri Nets," In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '03)*. IEEE Computer Society, Washington, DC, USA, 2003, pp. 143-155.

[18] M. Tribastone, P. Mayer and M. Wirsing, "Performance prediction of service-oriented systems with layered queueing networks," in *Leveraging Applications of Formal Methods, Verification, and Validation*, T. Margaria and B. Steffen, Eds. Springer Berlin / Heidelberg, 2010, pp. 51-65.

[19] X. Wu and M. Woodside, "Performance Modeling from Software Components", in *Proceedings of the 4th international workshop on Software and performance (WOSP '04)*, 2004, pp. 290-301.

[20] T. Omari, G. Franks, M. Woodside, and A. Pan, "Solving layered queueing networks of large client-server systems with symmetric replication," In *Proceedings of the 5th international workshop on Software and performance (WOSP '05)*, 2005, pp. 159-166.

[21] J. Xu, A. Oufimtsev, M. Woodside, and L. Murphy, "Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates," In *Proceedings of the 2005 conference on Specification and verification of component-based systems (SAVCBS '05)*. Article 5, 2005.

[22] J. Dilley, R. Friedrich, T. Jin, and J. Rolia, "Web server performance measurement and modeling techniques," *Performance Evaluation - Special issue on tools for performance evaluation*, vol. 33, no. 1, pp. 5-26, June 1998.

[23] C.U. Smith and L.G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley, 2002.

[24] D. Peng, Y. Yuan, K. Yue, X. Wang and A. Zhou, "Capacity planning for composite web services using queueing network-based models," in *Advances in Web-Age Information Management*, Q. Li, G. Wang and L. Feng, Eds. Springer Berlin / Heidelberg, 2004, pp. 439-448.

[25] Daniel A. Menasce, "Load Testing of Web Sites," *IEEE Internet Computing*, vol. 6, no. 4, pp. 70-74, July 2002.

[26] L. P. Slothouber, "A model of web server performance," In *Proceedings of the 5th International World Wide Web Conference*, 1996.

[27] X. Liu , J. Heo , L. Sha, "Modeling 3-Tiered Web Applications," in *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '05)*, September 27-29, 2005, pp. 307-310.

[28] X. P. Wu, "An Approach to Predicting Performance for Component Based Systems," MASc Thesis, Carleton University, Ottawa, Ontario, Canada, July 2003.

[29] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Prentice-Hall, 1984.

[30] N. J. Gunther, "What is guerrilla capacity planning?" in *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*, Springer: Berlin Heidelberg, 2007, pp. 1-16.

[31] S. Vugt, "Setting Up Web Services," in *Beginning Ubuntu Server Administration*, Apress, 2008, pp. 313-328.

[32] P. J. Denning and J. P. Buzen, "The Operational Analysis of Queueing Network Models," *ACM Computing Surveys (CSUR)*, vol. 10, no. 3, pp. 225-261, September 1978.