PERFORMANCE MODELING AND EVALUATION OF LARGE-SCALE J2EE APPLICATIONS*

Samuel Kounev Alejandro Buchmann

Department of Computer Science
Darmstadt University of Technology, Germany
{skounev,buchmann}@informatik.tu-darmstadt.de

Modern J2EE applications are typically based on highly distributed architectures comprising multiple components deployed in a clustered environment. This makes it difficult for deployers to estimate the capacity of the deployment environment needed to guarantee that Service Level Agreements are met. This paper looks at the different approaches to this problem and discusses the difficulties that arise when one tries to apply them to large, real-world systems. The authors study a realistic J2EE application (the SPECjAppServer2002 benchmark) and show how analytical models can be exploited for capacity planning.

1 Introduction

Over the past couple of years, the Java 2 Enterprise Edition Platform (J2EE) has established itself as major technology for developing modern e-business solutions. This success is largely due to the fact that J2EE is not a proprietary product, but rather an industry standard, developed as the result of a large industry initiative led by Sun Microsystems, Inc. The goal of this initiative was to establish a standard middleware framework for developing enterprise-class distributed applications in Java. Over 30 software vendors have participated in this effort and have come up with their own implementations of J2EE, the latter being commonly referred to as *J2EE Application Servers*.

The aim of J2EE is to enable developers to quickly and easily build scalable, reliable and secure applications without having to develop their own complex *middleware services*. Developers can concentrate on the business and application logic and rely on the J2EE Application Server to provide the infrastructure needed for scalability and performance. One of the key services within this infrastructure, provided by most J2EE application servers, is clustering. A *cluster* is a group of servers that act in a coordinated fashion to provide access to a set of applications in a scalable manner. When a J2EE application is deployed in a clustered environment, its components are transparently replicated on the servers participating in the cluster. Client requests are then load-balanced across cluster nodes and in this way scalability can be achieved.

Modern J2EE applications are typically based on highly distributed, multi-tiered architectures comprising multiple components deployed in a clustered environment. The inherent complexity of the latter makes it difficult for system deployers to estimate the size and capacity of the deployment environment needed to guarantee that Service Level Agreements (SLAs) are met. Deployers are often faced with questions such as the following:

- What hardware and software resources are needed to guarantee that SLAs are met? More specifically, how many application servers need to be included in the clusters used and how fast should they be?
- What are the maximum load levels that the system will be able to handle in the production environment?
- What would the average response time, throughput and resource utilization be under the expected workload?
- Which components have the largest effect on the overall system performance and are they potential bottlenecks?

*This work was partially funded by BEA Systems, Inc. as part of the project "Capacity Planning and Performance Analysis of J2EE Applications and Web Services" and the Deutsche Forschungsgemeinschaft (DFG) as part of the PhD program "Enabling Technologies for E-Commerce" at Darmstadt University of Technology.

The above are typical capacity planning questions [MA00]. In seeking answers to them deployers often rely on their intuition, ad hoc procedures, expert opinions or general rules of thumb. As a result, the overall system capacity is unknown and capacity planning and procurement are done without a strictly defined methodology. Clearly, the issues of sizing and capacity planning need to be approached in a more formal and systematic way if performance is to be guaranteed.

This paper studies a real-world J2EE application of a realistic complexity and shows how to exploit analytical performance models in order to address the problems discussed above. The application studied is the SPECjAppServer2002 J2EE benchmark [Sta02], which represents a heavy-duty business-to-business e-commerce workload. The paper starts by giving an overview of the different approaches to performance analysis of distributed systems. It then concentrates on analytical modeling and discusses the different types of analytical performance models and their advantages and disadvantages. Following that, the paper introduces the SPECjAppServer2002 benchmark and shows how to develop a queueing network model of a SPECjAppServer2002 deployment. This model is then used to predict system performance under several different configurations. Finally, results obtained from the model are validated through measurements.

2 Approaches to Performance Analysis

Two broad approaches help carry out performance analysis of distributed systems:

- Load Testing
- Performance Modeling

In the first approach, load-testing tools are used to generate artificial workloads on the system and measure its performance. Sophisticated load testing tools can emulate hundreds of thousands of 'virtual users' that mimic real users interacting with the system. While tests are run, system components are monitored and performance metrics (e.g. response time, latency, utilization and throughput) are measured. Results obtained in this way can be used to identify and isolate system bottlenecks, fine-tune application components and predict the end-to-end system scalability. Unfortunately this approach is extremely expensive, since it requires setting up a production-like testing environment to conduct the tests. Moreover, it is not applicable in the early stages of system development when the system is not available for testing.

In the second approach, performance models are built and then used to analyze the performance and scalability characteristics of the system under study. Models represent the way system resources are used by the workload and capture the main factors determining the behavior of the system under load. Performance models can be grouped into two common categories: *simulation models* and *analytical models* [MA00].

Simulation models are software programs that mimic the behavior of a system as requests arrive and get processed by various resources. The structure of a simulation program is based on the states of the simulated system and events that change the system state. Simulation programs measure performance by counting events and recording the duration of time spent in different states. The main advantage of simulation models is their great generality and the fact that they can provide very accurate results. However, this accuracy comes at the cost of the time taken to develop and run the models. Usually lots of long runs are required in order to obtain estimates of needed performance measures with reasonable confidence levels [MA00].

Analytical models are a cost-effective alternative to simulation. They are based on mathematical laws and computational algorithms used to generate performance metrics from model parameters.

3 Analytical Performance Models

Analytical models can be broadly classified into *state-space models* and *non-state-space models* [BGMT98]. The most commonly used state-space models are *Markov Chains*. The latter consist of a set of states and a set of labeled transitions between the states. States represent conditions of interest in the system under study - for example the number of requests waiting to use a resource. A number of concise specification techniques for Markov Chains are available in the literature. *Queueing Networks (QNs)* [Tri02] and *Stochastic Petri Nets (SPNs)* [BK02] are among the most popular examples. Moreover, a number of extensions both to Queueing Networks and to Stochastic Petri Nets have been proposed, which further enhance their modeling power. Examples are Extended Queueing Networks [BGMT98] and Queueing Petri Nets [Bau93]. The main problem with Markov Chains, however, is that as one increases the size of the model, the underlying state space grows exponentially and quickly exceeds the capacity of today's computers [KB03]. This is the so-called *state-space explosion* problem and is a major hurdle to the practical application of Markov Chains and state-space models in general.

The continuing need to be able to solve larger models has led to the emergence of non-state-space models. The most popular example are the well-known *Product-Form Queueing Networks (PFQNs)* [BGMT98], which have been widely used in the past decades. PFQNs are a specific class of queueing networks, for which it is possible to derive steady-state performance measures without resorting to the underlying state space. Relatively large PFQNs can be solved by means of

simpler equations. However, many practical queueing networks (so-called *Non-Product-Form Queueing Networks - NPFQNs*) do not satisfy the requirements for a product-form solution. In such cases, it is often possible to obtain accurate approximations using techniques based on the solution methods for PFQNs.

In the past decades, non-state-space models have been the subject of rigorous research and considerable advances have been made in the development of techniques for their solution. Many books and research papers have been written on the topics of performance modeling and prediction using product and non-product-form queueing networks. However, when it comes to practical applications in the field of e-business, examples quoted are usually targeting highly-specialized and rather unrealistic scenarios. The goal of this paper is to demonstrate how non-state-space queueing network models can be exploited for performance analysis of real-world, J2EE-based e-business systems. It discusses the problems that arise from the limitations of the models in terms of expressiveness and shows how they can be circumvented. The reason for choosing to use queueing networks is that they have the advantage that a large number of efficient methods are available for their solution which can handle relatively large models [KB03].

4 The SPECjAppServer2002 Benchmark

SPECjAppServer2002 is a new industry standard benchmark for measuring the performance and scalability of J2EE-based application servers. The benchmark was released in November 2002 and is essentially a J2EE1.3/EJB2.0 port of the existing SPECjAppServer2001 benchmark. Both SPECjAppServer2001 and 2002 are based on the popular ECperf benchmark which was prototyped and built by Sun Microsystems in conjunction with application server vendors under the Java Community Process (JCP). Server vendors can use the SPECjAppServer benchmarks to measure, optimize and showcase their product's performance and scalability. Their customers, on the other hand, can use them to gain a better understanding and insight into the tuning and optimization issues surrounding the development of modern J2EE applications.

4.1 SPECjAppServer2002 Business Model

The SPECjAppServer2002 workload is based on a distributed application claimed to be large enough and complex enough to represent a real-world e-business system [Sta02]. The benchmark designers have chosen manufacturing, supply chain management, and order/inventory as the "storyline" of the business problem to be modeled. This is an industrial-strength distributed problem, that is heavyweight, mission-critical and requires the use of a powerful and scalable infrastructure.

SPECjAppServer2002 models businesses using four domains: *customer domain* dealing with customer orders and interactions, *manufacturing domain* performing "just in time" manufacturing operations, *supplier domain* handling interactions with external suppliers and *corporate domain* managing all customer, product, and supplier information. Figure 1 illustrates these domains and gives some examples of typical transactions run in each of them.

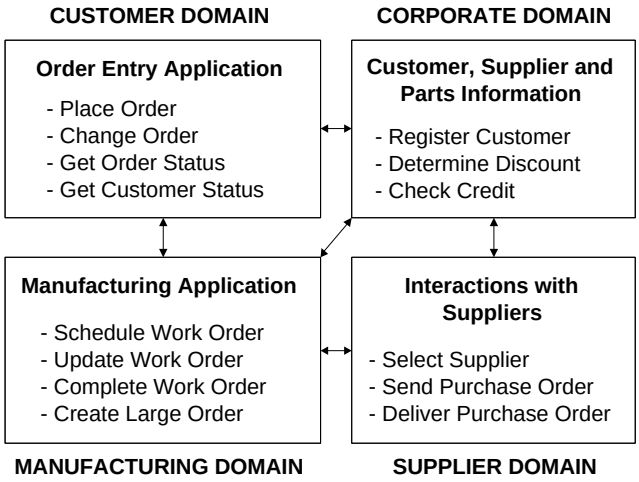

Figure 1: SPECjAppServer2002 Business Domains

The customer domain models customer interactions using an order entry application, which provides some typical online ordering functionality. Orders can be placed by individual customers as well as by distributors. Orders placed by distributors are called *large orders*.

The manufacturing domain models the activity of production lines in a manufacturing plant. Products manufactured by the plant are called *widgets*. There are two types of production lines, namely *planned lines* and *large order lines*. Planned lines run on schedule and produce a predefined number of widgets. Large order lines run only when a large order is received in the customer domain. The unit of work in the manufacturing domain is a *work order*. Each work order is for a specific quantity of a particular type of widget. When a work order is created, the bill of materials for the corresponding type of widget is retrieved and the required parts are taken out of inventory. As the widgets move through the assembly line, the work order status is updated to reflect progress. Once the work order is complete, it is marked as completed and inventory is updated. When inventory of parts gets depleted, suppliers need to be located and *purchase orders (POs)* need to be sent out. This is done by contacting the supplier domain, which is responsible for interactions with external suppliers.

4.2 SPECjAppServer2002 Application Design

All the activities and processes in the four domains described above are implemented using Enterprise Java Bean (EJB) components (adhering to the EJB 2.0 specification [Sun]) assembled into a single J2EE application which is deployed in an application server running on the *System Under Test (SUT)*. The only exception is for the interactions with suppliers which are implemented using a separate Java servlet application called *Supplier Emulator*. The latter is deployed in a Java-enabled web server on a dedicated machine. The supplier emulator provides the supplier domain with a way to emulate the process of sending and receiving purchase orders to/from suppliers.

The workload generator is implemented using a multithreaded Java application called *SPECjAppServer Driver*. The latter is designed to run on multiple client machines, using an arbitrary number of Java Virtual Machines to ensure that it has no inherent scalability limitations. A relational DBMS is used for data persistence [KB02] and all data access operations use entity beans which are mapped to tables in the SPECjAppServer database.

The throughput of the benchmark is driven by the activity of the order entry and manufacturing applications. The throughput of both applications is directly related to the chosen *Transaction Injection Rate*, which determines the number of order entry requests generated and the number of work orders scheduled per second. The summarized performance metric provided after running the benchmark is called **TOPS** and it denotes the average number of successful **T**otal **O**perations **P**er **S**econd completed during the measurement interval. Readers interested in more detail on the SPECjAppServer2002 EJBs, the database model and transactions implemented can refer to [Sta02].

5 Modeling SPECjAppServer2002

This section discusses how to build and validate a queueing network model of SPECjAppServer2002 and, then, how to exploit this model for the purposes of performance prediction in the capacity planning process.

5.1 Motivation

Imagine the following hypothetical scenario: A company is about to automate its internal and external business operations with the help of e-business technology. The company chooses to employ the J2EE platform and develops a J2EE application for supporting its order-inventory, supply-chain and manufacturing operations. Assume that this application is the one provided in the SPECjAppServer2002 benchmark. Assume also that the company plans to deploy the application in the deployment environment depicted in Figure 2. This environment uses a cluster of WebLogic servers (WLS) as a J2EE container and an Oracle database server (DBS) for persistence. We assume that all machines in the WLS cluster are identical.

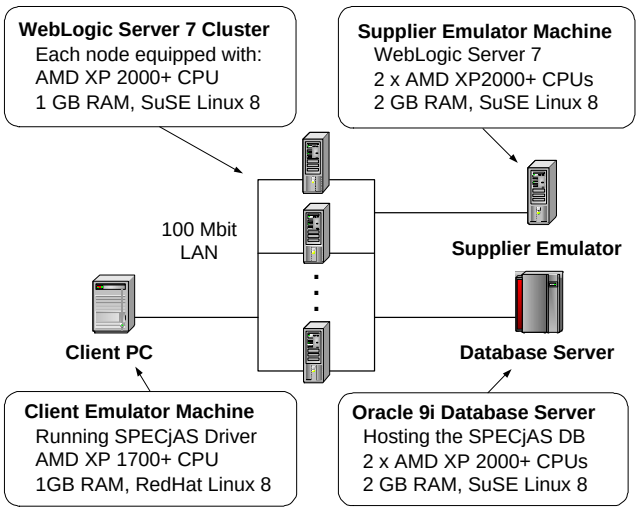

Figure 2: Deployment Environment

Before putting the application into production the company conducts a capacity planning study in order to come up with an adequate sizing and configuration of the deployment environment. More specifically, the company needs answers to the following questions:

- How many WebLogic servers would be needed to guarantee adequate performance under the expected workload?
- For a given number of WebLogic servers, what level of performance would the system provide? What would the average transaction throughput and response time be? How utilized (CPU/Disk utilization) would the WebLogic servers and the database server be?
- Will the capacity of the database server suffice to handle the incoming load?
- Does the system scale or are there any other potential system bottlenecks?

These issues can be approached with the help of queueing network-based performance models.

5.2 Workload Characterization

The first step in the capacity planning process is to describe the workload of the system under study in a qualitative and quantitative manner. This is called *workload characterization* [MA98] and in its simplest form includes four major steps:

1. Describe the types of requests that are processed by the system (called *request classes*).
2. Identify the hardware and software resources used by each request class.

3. Measure the total amount of service time (called *service demand*) for each request class at each resource.

4. Give an indication of the number of requests of each class that the system will be exposed to. The latter is often termed *workload intensity*.

As already discussed, the SPECjAppServer2002 workload is made up of two major components - the *order entry application* in the customer domain and the *manufacturing application* in the manufacturing domain. Recall that the order entry application is running the following four transaction types:

1. *NewOrder*: places a new order in the system

2. *ChangeOrder*: modifies an existing order

3. *OrderStatus*: retrieves the status of a given order

4. *CustStatus*: lists all orders of a given customer

We map each of them to a separate *request class* in our workload model. The manufacturing application, on the other hand, is running production lines. The main unit of work there is a *work order*. Each work order produces a specific quantity of a particular type of widget. As already mentioned, there are two types of production lines: planned lines and large order lines. While planned lines run on a predefined schedule, large order lines run only when a large order arrives in the customer domain. Each large order results in a separate work order. During the processing of work orders multiple transactions are executed in the manufacturing domain, i.e. scheduleWorkOrder, updateWorkOrder and completeWorkOrder. Each work order moves along 3 virtual *stations*, which represent distinct operations in the manufacturing flow. In order to simulate activity at the stations, the manufacturing application waits for a designated time at each station.

One way to model the manufacturing workload would be to define a separate request class for each transaction run during the processing of work orders. However, this would lead to an overly complex model and would limit the range of analysis techniques that would be applicable for its solution. Second, it wouldn't be of much benefit, since after all, what most interests us is the rate at which work orders are processed and not the performance metrics of the individual work-order-related transactions. Therefore, we model the manufacturing workload only at the level of work orders. We define a single request class *WorkOrder*, which represents a request for processing a work order. This keeps our model simple and as will be seen later is enough to provide us with sufficient information about the behavior of the manufacturing application.

Altogether, we end up with 5 request classes: NewOrder, ChangeOrder, OrderStatus, CustStatus and WorkOrder. The following resources are used during their processing:

- The CPU of a WebLogic server (WLS-CPU)
- The Local Area Network
- The CPUs of the database server (DBS-CPU)
- The disk drives of the database server (DBS-I/O)

In order to determine the service demands at these resources, we conducted a separate experiment for each of the 5 request classes. In each case, we deployed the benchmark in a configuration with a single WebLogic server and then injected requests of the respective class into the system. During the experiment, we monitored the system resources and measured the time requests spend at each resource during their processing. For the database server, we used the Oracle 9i Intelligent Agent, which provides exhaustive information about CPU consumption, as well as I/O wait times. For the application server, we monitored the CPU utilization using operating system tools and then used the *Service Demand Law* [MA98] to derive the CPU service demand: the service demand $\mathcal{D}$ of requests at a given resource is equal to the average resource utilization $\mathcal{U}$ divided by the average request throughput $\mathcal{X}$, during the measurement interval (assuming, of course, that requests of just one type are processed during the experiment), i.e.

$$\mathcal{D} = \frac{\mathcal{U}}{\mathcal{X}} \tag{1}$$

We decided we could safely ignore network service demands, since all communications were taking place over a 100 MBit LAN and communication times were negligible. Table 1 reports the service demand measurements for the 5 request classes in our workload model. Figure 3 summarizes these measurements in a graphical form.

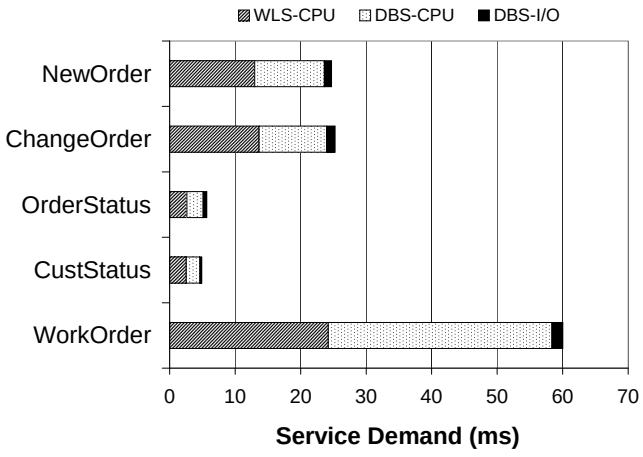

Figure 3: Workload Service Demands

Table 1: Workload Service Demands

TX-Type	WLS-CPU	DBS-CPU	DBS-I/O
NewOrder	12.98ms	10.64ms	1.12ms
ChangeOrder	13.64ms	10.36ms	1.27ms
OrderStatus	2.64ms	2.48ms	0.58ms
CustStatus	2.54ms	2.08ms	0.3ms
WorkOrder	24.22ms	34.14ms	1.68ms

As we can see from Table 1, database I/O service demands are much lower than CPU service demands. This stems from the fact that data is cached in the database buffer and disks are usually accessed only when updating or inserting new data. However, even in this case I/O overhead is minimal since the only thing that is done is to flush the database log buffer, which is performed with sequential I/O accesses. Here we would like to point out that the current version of the benchmark uses relatively small data volumes for the workload intensities generated. This results in data contention [KB02] and as we will see later causes some difficulties in predicting transaction response times since data contention does not easily lend itself to analytical modeling using conventional techniques.

Once we know the service demands of the different request classes, we proceed with the last step in workload characterization, which aims to quantify workload intensity. For each request class, we must specify the rates at which requests arrive. We should also be able to vary these rates so that we can consider different scenarios. To this end, we modified the SPECjAppServer2002 driver to allow more flexibility in configuring the intensity of the workload generated. Specifically, the new driver allows us to set the number of concurrent order entry clients simulated, as well as their average *think time*, i.e. the time they "think" after receiving a response from the system, before they send the next request. In addition to this, we can specify the number of planned production lines run in the manufacturing domain and the time they wait after processing a work order before starting a new one. In this way, we can precisely define the workload intensity and transaction mix. We will later study in detail several different scenarios under different transaction mixes and workload intensities.

5.3 Building a Performance Model

In this section, we build a queueing network model of our SPECjAppServer2002 deployment environment. We first define the model in a general fashion and then customize it to our concrete workload scenarios. We use a closed model, which means that for each instance of the model the number of concurrent clients sending requests to the system is fixed. Figure 4 shows a high-level view of our queueing network model. For a formal definition of our model's queues in Kendall's notation see Appendix A.

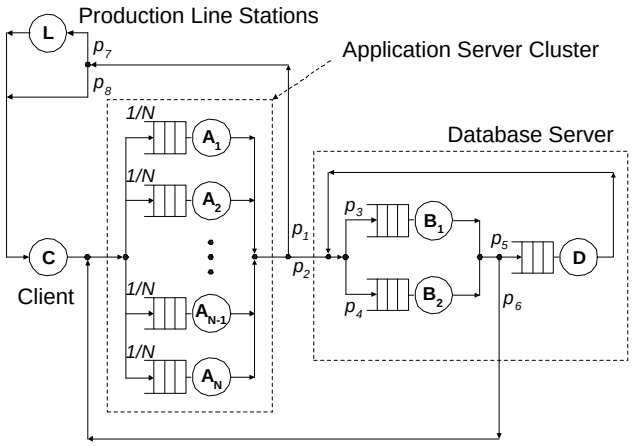

Figure 4: Queueing Network Model of the System

Following is a brief description of the queues used:

C : "Infinite Server" (IS) queue (also called *delay resource*) used to model the client machine which runs the SPECjAppServer driver and emulates virtual clients sending requests to the system. The service time of order entry requests at this queue is equal to the average client think time, while the service time of WorkOrder requests is equal to the average time a production line waits after processing a work order before starting a new one. Note that times spent on this queue are not part of system response times.

$A_1..A_N$: "Processor Sharing" (PS) queues used to model the CPUs of the *N* WebLogic servers.

B_1, B_2 : "Processor Sharing" (PS) queues used to model the two CPUs of the database server.

D : "First-Come-First-Served" (FCFS) queue used to model the disk subsystem (made up of a single 100GB disk drive) of the database server.

L : "Infinite Server" (IS) queue (delay resource) used to model the virtual production line stations in the manufacturing domain. Only WorkOrder requests ever visit this queue. Their service time at the queue corresponds to the average delay at the production line stations simulated by the manufacturing application during work order processing.

The model is a closed queueing network model with the 5 classes of requests (jobs) defined in the previous section. The behavior of requests in the model is defined by specifying their respective routing probabilities p_i and service demands at each queue which they visit. We discussed the service demands in the previous section. To set the routing probabilities we examine the lifecycle of client requests in the queueing network. Every

request is initially at the client queue C, where it waits for a user-specified think time. After the think time elapses, the request is routed to a randomly chosen queue A_i, where it queues to receive service at a WebLogic server CPU. We assume that requests are evenly distributed over the N WebLogic servers, i.e. each server is chosen with probability $1/N$. Processing at the CPU may be interrupted multiple times if the request requires some database accesses. Each time this happens, the request is routed to the database server where it queues for service at one of the two CPU queues B_1 or B_2 (each chosen equally likely so that $p_3 = p_4 = 0.5$). Processing at the database CPUs may be interrupted in case I/O accesses are needed. For each I/O access the request is sent to the disk subsystem queue D and after receiving service there, is routed back to the database CPUs. This may be repeated multiple times depending on routing probabilities p_5 and p_6. Having completed their service at the database server, requests are sent back to the application server. Requests may visit the database server multiple times during their processing, depending on routing probabilities p_1 and p_2. After completing service at the application server requests are sent back to the client queue C. Order entry requests are sent directly to the client queue (for them $p_8 = 1$, $p_7 = 0$), while WorkOrder requests are routed through queue L (for them $p_8 = 0$, $p_7 = 1$), where they are additionally delayed for 1 second. This delay corresponds to the 1 second delay at the 3 production line stations imposed by the manufacturing application during work order processing.

In order to set routing probabilities p_1, p_2, p_5 and p_6 we need to know how many times a request visits the database server during its processing and for each visit how many times I/O access is needed. Since we only know the total service demands over all visits to the database, we assume that requests visit the database just once and need a single I/O access during this visit. This allows us to drop routing probabilities p_1, p_2, p_5 and p_6 and leads us to the following simplified model depicted in Figure 5.

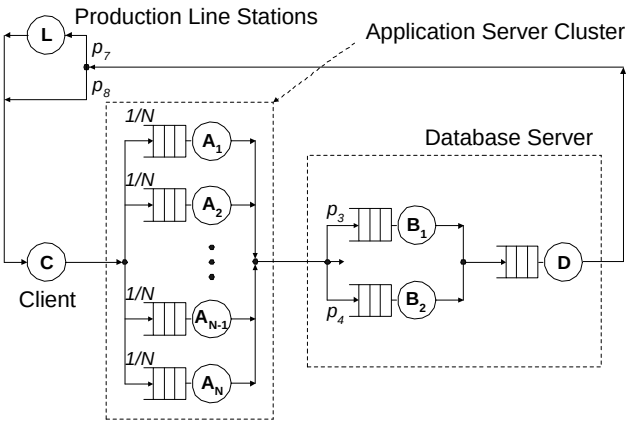

Figure 5: Simplified QN Model of the System

The following input parameters need to be supplied before the model can be analyzed:

- Number of order entry clients (NewOrder, ChangeOrder, OrderStatus and CustStatus).
- Average think time of order entry clients - *Customer Think Time*.
- Number of planned production lines generating WorkOrder requests.
- Average time production lines wait after processing a work order, before starting a new one - *Manufacturing (Mfg) Think Time*.
- Service demands of the 5 request classes at queues A_i, B_j, and D (see Table 1).

In our study we consider two types of deployment scenarios. In the first one, large order lines in the manufacturing domain are turned off. In the second one, they are running as defined in the benchmark workload. The reason for this separation is that large order lines introduce some asynchronous processing, which in general is hard to model using queueing networks. We start with the simpler case where we don't have such processing and then show (in section 5.4.4) how large order lines can be integrated into our model.

5.4 Model Analysis and Validation

We now proceed to analyze several different instances of the model introduced in the previous section and then validate them by comparing results from the analysis with measured data. We first consider the case without large order lines and study the system in three scenarios representing low, moderate and heavy load, respectively. In each case, we examine deployments with different number of application servers - from 1 to 9.

Table 2 summarizes the input parameters for the three scenarios that we consider.

Table 2: Model Input Parameters for the 3 Scenarios

Parameter	Low	Moderate	Heavy
NewOrder Clients	30	50	100
ChangeOrder Clients	10	40	50
OrderStatus Clients	50	100	150
CustStatus Clients	40	70	50
Planned Lines	50	100	200
Customer Think Time	2 sec	2 sec	3 sec
Mfg Think Time	3 sec	3 sec	5 sec

5.4.1 Scenario 1: Low Load

A number of analysis tools for queueing networks have been developed and are available free of charge for noncommercial use [BGMT98]. We employed the

PEPSY-QNS tool [BK94] (Performance Evaluation and Prediction SYstem for Queueing NetworkS) from the University of Erlangen-Nuernberg. We chose PEPSY because it supports a wide range of solution methods (over 30) for product- and non-product-form queueing networks. Both exact and approximate methods are provided, which are applicable to models of considerable size and complexity. For the most part, we have applied the *multisum method* [Bol89] for solution of the queueing network models in this paper. However, to ensure plausibility of the results, we cross-verified them with results obtained from other methods such as *bol_aky* and *num_app* [BK94]. In all cases the difference was negligible.

Table 3 summarizes the results we obtained for our first scenario. We studied two different configurations - the first one with 1 application server, the second with 2. The table reports throughput (X) and response time (R) for the 5 request classes, as well as CPU utilization (U) of the application server and the database server. Results obtained from the model analysis are compared against results obtained through measurements and the modeling error is reported.

As we can see from the table, while throughput and utilization results are extremely accurate, the same does not hold to this extent for response time results. This is because when we run a transaction mix, as opposed to a single transaction, some additional delays are incurred which are not captured by the model. For example, delays result from contention for data access (database locks, latches), processes, threads, database connections, etc. The latter is often referred to as *software contention*, in contrast to *hardware contention* (contention for CPU time, disk access and other hardware resources). Our model captures the hardware contention aspects of system behavior and does not represent software contention aspects. While software contention may not always have a big impact on transaction throughput and CPU utilization, it usually does have a direct impact on transaction response time and therefore real (measured) response times are higher than the ones obtained from the model. In [Men02] and [RS95] some techniques are presented for estimating delays incurred from software contention. However, they are rather approximative and attempting to apply them to our scenario leads to technical difficulties stemming from the size and complexity of our system. [KB03], [BBK97] discuss some more accurate approaches for integrating both hardware and software contention aspects into the same model. The latter however suffer the state-space explosion problem and are currently not applicable for systems of the size of the one studied in this paper.

From Table 3 we see that the response time error for requests with very low service demands (e.g. OrderStatus and CustStatus) is much higher than average. This is because the processing times for such requests are very low (around 10ms) and the additional delays from software contention, while not that high as absolute values, are high relative to the overall response times. The results show that the higher the service demand of a request type, the lower the response time error is. Indeed, the requests with the highest service demand (WorkOrder) always have the lowest response time error.

5.4.2 Scenario 2: Moderate Load

This time we have 260 concurrent clients interacting with the system and 100 planned production lines running in the manufacturing domain. This is twice as much as in the previous scenario. We study 2 deployments - the first with 3 application servers, the second with 6. Table 4 summarizes the results from the model analysis. Again we obtain very accurate throughputs and utilizations, and accurate response times. The response time error does not exceed 35%, which is considered acceptable in most capacity planning studies [MA00].

5.4.3 Scenario 3: Heavy Load

We have 350 concurrent clients and 200 planned production lines in total. We consider three configurations - with 4, 6 and 9 application servers, respectively. However, we slightly increase the think times in order to make sure that our single machine database server is able to handle the load. Table 5 summarizes the results for this scenario. For models of this size, the available algorithms do not produce reliable results for response time and therefore we only consider throughput and utilization in this scenario.

5.4.4 Scenarios with Large Order Lines

We now consider the case when large order lines in the manufacturing domain are enabled. The latter are activated upon arrival of large orders in the customer domain. Each large order generates a separate work order, which is processed asynchronously at one of the large order lines. As already mentioned, this poses a difficulty since queueing networks provide very limited possibilities for modeling this type of asynchronous processing. As shown in [KB03], other state-space-based models such as Queueing Petri Nets are much more powerful in such situations. However, as discussed in section 3, they suffer a state-space explosion problem and are currently not applicable for systems of the size and complexity of SPECjAppServer2002. Therefore, we don't have a choice but to somehow try to integrate the large order lines into our queueing network model.

Since large order lines are always triggered by NewOrder transactions (for large orders) we can add the load they produce to the service demands of NewOrder requests. To do that we rerun the NewOrder experiments with the large order lines turned on. The additional load leads to higher utilization of system resources and in this way impacts the measured

Table 3: Analysis Results for Scenario 1 - Low Load

	1 Application Server			2 Application Servers		
METRIC	Model	Measured	Error	Model	Measured	Error
NewOrder Throughput	14.59	14.37	1.5%	14.72	14.49	1.6%
ChangeOrder Throughput	4.85	4.76	1.9%	4.90	4.82	1.7%
OrderStatus Throughput	24.84	24.76	0.3%	24.89	24.88	0.0%
CustStatus Throughput	19.89	19.85	0.2%	19.92	19.99	0.4%
WorkOrder Throughput	12.11	12.19	0.7%	12.20	12.02	1.5%
NewOrder Response Time	56ms	68ms	17.6%	37ms	47ms	21.3%
ChangeOrder Response Time	58ms	67ms	13.4%	38ms	46ms	17.4%
OrderStatus Response Time	12ms	16ms	25.0%	8ms	10ms	20.0%
CustStatus Response Time	11ms	17ms	35.2%	7ms	10ms	30.0%
WorkOrder Response Time	1127ms	1141ms	1.2%	1092ms	1103ms	1.0%
WebLogic Server CPU Utilization	66%	70%	5.7%	33%	37%	10.8%
Database Server CPU Utilization	36%	40%	10%	36%	38%	5.2%

Table 4: Analysis Results for Scenario 2 - Moderate Load

	3 Application Servers			6 Application Servers		
METRIC	Model	Measured	Error	Model	Measured	Error
NewOrder Throughput	24.21	24.08	0.5%	24.29	24.01	1.2%
ChangeOrder Throughput	19.36	18.77	3.1%	19.43	19.32	0.6%
OrderStatus Throughput	49.63	49.48	0.3%	49.66	49.01	1.3%
CustStatus Throughput	34.77	34.24	1.5%	34.80	34.58	0.6%
WorkOrder Throughput	23.95	23.99	0.2%	24.02	24.03	0.0%
NewOrder Response Time	65ms	75ms	13.3%	58ms	68ms	14.7%
ChangeOrder Response Time	66ms	73ms	9.6%	58ms	70ms	17.1%
OrderStatus Response Time	15ms	20ms	25.0%	13ms	18ms	27.8%
CustStatus Response Time	13ms	20ms	35.0%	11ms	17ms	35.3%
WorkOrder Response Time	1175ms	1164ms	0.9%	1163ms	1162ms	0.0%
WebLogic Server CPU Utilization	46%	49%	6.1%	23%	25%	8.0%
Database Server CPU Utilization	74%	76%	2.6%	73%	78%	6.4%

Table 5: Analysis Results for Scenario 3 - Heavy Load

	4 App. Servers			6 App. Servers			9 App. Servers		
METRIC	Model	Msrd.	Error	Model	Msrd.	Error	Model	Msrd.	Error
NewOrder Throughput	32.19	32.29	0.3%	32.22	32.66	1.3%	32.24	32.48	0.7%
ChangeOrder Throughput	16.10	15.96	0.9%	16.11	16.19	0.5%	16.12	16.18	0.4%
OrderStatus Throughput	49.59	48.92	1.4%	49.60	49.21	0.8%	49.61	49.28	0.7%
CustStatus Throughput	16.55	16.25	1.8%	16.55	16.24	1.9%	16.55	16.46	0.5%
WorkOrder Throughput	31.69	31.64	0.2%	31.72	32.08	1.1%	31.73	32.30	1.8%
WebLogic Server CPU Utilization	40%	42%	4.8%	26%	29%	10.3%	18%	20%	10.0%
Database Server CPU Utilization	87%	89%	2.2%	88%	91%	3.3%	88%	91%	3.3%

NewOrder service demands (see Table 6). While this incorporates the large order line activity into our model, it changes the semantics of NewOrder jobs. In addition to the NewOrder transaction load, they now also include the load caused by large order lines. Thus, performance metrics (throughput, response time) for NewOrder requests no longer correspond to the respective metrics of the NewOrder transaction. Therefore, we can no longer quantify the performance of the NewOrder transaction on itself. Nevertheless, we can still analyze the performance of other transactions and gain a picture of the overall system behavior. Table 7 summarizes the results for the three scenarios with large order lines enabled. For lack of space, this time we only look at one configuration per scenario: the first one with 1 application server, the second with 3 and the third with 9.

Table 7: Analysis Results for Scenarios with Large Order Lines

	Low/1AS		Moderate/3AS		Heavy/9AS	
METRIC	Model	Error	Model	Error	Model	Error
ChangeOrder Throughput	4.79	6.4%	19.09	3.5%	15.31	4.5%
OrderStatus Throughput	24.77	2.9%	49.46	2.3%	48.96	3.1%
CustStatus Throughput	19.83	2.4%	34.67	2.1%	16.37	1.9%
WorkOrder Throughput	11.96	5.7%	23.43	2.6%	29.19	1.2%
WebLogic Server CPU Utilization	80%	0.0%	53%	1.9%	20%	0.0%
Database Server CPU Utilization	43%	2.4%	84%	2.4%	96%	1.0%

Table 6: NewOrder Service Demands with the Large Order Lines running

TX-Type	WLS-CPU	DBS-CPU	DBS-I/O
NewOrder	23.49ms	21.61ms	1.87ms

5.5 Conclusions from the Analysis

We used our queueing network model to predict system performance in several different configurations varying the workload intensity and the number of application servers available. The results enable us to give answers to the questions we started with in section 5.1. For each configuration we obtained approximations for the average request throughput, response time and server utilization. Depending on the service level agreements (SLAs) and the expected workload intensity, we can now determine how many application servers we need in order to guarantee adequate performance. We can also see for each configuration which component is mostly utilized and thus could become a potential bottleneck (see Figure 6). In scenario 1, we saw that by using a single application server, the latter could easily turn into bottleneck since its utilization would be twice as high as that of the database server. The problem is solved by adding an extra application server. In scenarios 2 and 3, we saw that with more than 3 application servers as we increase the load, the database CPU utilization approaches 90%, while the application servers remain in all cases less than 50% utilized. This clearly indicates that, in this case, our database server is the bottleneck.

6 Summary and Conclusions

This paper discussed the different approaches to performance analysis of large J2EE applications, focusing on analytical modeling, since it is usually much more cost-effective than simulation or load-testing. We studied a real-world J2EE application of a realistic complexity and modeled it using a non-product-form queueing network. With the help of the latter, we analyzed several different deployment environments under different workload intensities. In each case, we used the model to predict system performance (transaction throughput, response time and CPU utilization) and, then, validated results through measurements. The model was extremely accurate in predicting transaction throughput and CPU utilization, and a bit less accurate in predicting transaction response time. The average modeling error for throughput was 2%, for CPU utilization 6% and for response time 18%. In the course of the study, we discussed the problems that arise from limited model expressiveness, on the one hand, and from application size and complexity, on the other hand. We proposed different methods to address these problems and illustrated them through practical examples.

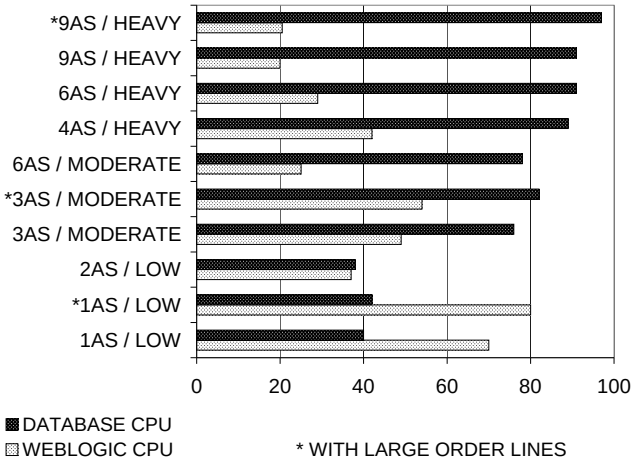

Figure 6: Server Utilization in Different Scenarios

In spite of the limitations of queueing network models to deal with software contention and the high degree of mistrust with which the J2EE industry looks at analytical models in general, this paper demonstrated that the latter are very powerful as a performance prediction tool and lend themselves well to modeling J2EE applications of a realistic size and complexity. This can be exploited in the capacity planning process for large e-business systems.

Acknowledgments

We gratefully acknowledge the cooperation of Dr. Gunter Bolch from the University of Erlangen in providing us with the PEPSY-QNS tool. We also acknowledge the use of BEA's WebLogic application server and

the support of our colleague Matthias Meixner from the University of Darmstadt. Last, but not least, we would like to thank Prof. Ethan Bolker from the University of Massachusetts at Boston and Margaret Greenberg from Grunberg Haus for helping us to improve the paper and shape it for a CMG audience.

References

[Bau93] F. Bause. Queueing Petri Nets - A formalism for the combined qualitative and quantitative analysis of systems. In *Proc. of the 5th International Workshop on Petri Nets and Performance Models, Toulouse (France)*, 1993.

[BBK97] F. Bause, P. Buchholz, and P. Kemper. Integrating Software and Hardware Performance Models Using Hierarchical Queueing Petri Nets. In *Proc. of the 9. ITG / GI - Fachtagung Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, (MMB'97), Germany*, 1997.

[BGMT98] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains - Modelling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons, Inc., 1998.

[BK94] G. Bolch and M. Kirschnick. The Performance Evaluation and Prediction SYstem for Queueing NetworkS - PEPSY-QNS. Technical Report TR-I4-94-18, University of Erlangen-Nuremberg, Germany, June 1994. `http://www4.informatik.uni-erlangen.de/Projects/PEPSY/en/pepsy.html`.

[BK02] F. Bause and F. Kritzinger. *Stochastic Petri Nets - An Introduction to the Theory*. Vieweg Verlag, second edition, 2002.

[Bol89] G. Bolch. *Performance Evaluation of Computer Systems with the help of Analytical Queueing Network Models*. Teubner Verlag, Stuttgart, 1989.

[KB02] S. Kounev and A. Buchmann. Improving Data Access of J2EE Applications by Exploiting Asynchronous Processing and Caching Services. In *Proc. of the 28th International Conference on Very Large Data Bases - VLDB2002*, 2002.

[KB03] S. Kounev and A. Buchmann. Performance Modelling of Distributed E-Business Applications using Queueing Petri Nets. In *Proc. of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software - ISPASS2003*, 2003.

[MA98] D. Menasce and V. Almeida. *Capacity Planning for Web Performance: Metrics, Models and Methods*. Prentice Hall, Upper Saddle River, NJ, 1998.

[MA00] D. Menasce and V. Almeida. *Scaling for E-Business - Technologies, Models, Performance and Capacity Planning*. Prentice Hall, Upper Saddle River, NJ, 2000.

[Men02] D. Menasce. Two-Level Iterative Queueing Modeling of Software Contention. In *Proceedings of the 10th IEEE Intl. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, 2002.

[RS95] J. Rolia and K. Sevcik. The method of layers. *IEEE Tr. Software Eng.*, 21(8):689–700, 1995.

[Sta02] Standard Performance Evaluation Corporation (SPEC). SPECjAppServer2002 Documentation. Specifications, November 2002. `http://www.spec.org/osg/jAppServer/`.

[Sun] Sun Microsystems, Inc. Enterprise JavaBeans 1.1 and 2.0. Specifications. `http://java.sun.com/products/ejb/`.

[Tri02] K. S. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Science Applications*. John Wiley & Sons, Inc., second edition, 2002.

Appendix A

Table 8: Formal Queue Definitions

Queue	Type	Description
$A_1..A_N$	$-/M/1-PS$	WLS CPUs
B_1, B_2	$-/M/1-PS$	DBS CPUs
D	$-/M/1-FCFS$	DBS Disk Subsystem
C	$-/M/\infty-IS$	Client Machine
L	$-/M/\infty-IS$	Prod. Line Stations

Table 8 presents a formal specification of our model's queues (i.e. queueing stations) in Kendall's notation. With queueing stations defined in this way, we end up with a non-product-form queueing network. Note that we could have chosen to model the CPUs of the database server using a single $-/M/2-PS$ queue instead of two separate $-/M/1-PS$ queues. However, many efficient analysis techniques for non-product-form queueing networks do not support $-/M/m-PS$ queues and therefore we chose the first option so that we have more flexibility when analyzing our models.