Performance Modelling of Distributed E-Business Applications using Queueing Petri Nets

Samuel Kounev Alejandro Buchmann

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

Abstract

In this paper [1] we show how Queueing Petri Net (QPN) models can be exploited for performance analysis of distributed e-business systems. We study a real-world application and demonstrate the benefits, in terms of modelling power and expressiveness, that QPN models provide over conventional modelling paradigms such as Queueing Networks and Petri Nets. As shown, QPNs facilitate the integration of both hardware and software aspects of system behavior in the same model. In addition to hardware contention and scheduling strategies, using QPNs one can easily model simultaneous resource possession, synchronization, blocking and contention for software resources. By validating the models presented through measurements, we show that they are not just powerful as a specification mechanism, but are also very powerful as a performance analysis and prediction tool. However, currently available tools and techniques for QPN analysis are limited. Improved solution methods, which enable larger models to be analyzed, need to be developed. By demonstrating the power of QPNs as a modelling paradigm in realistic scenarios, we hope to motivate further research in this area.

1 Introduction

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

1. What are the maximum load levels that the system will be able to handle in the production environment?
2. What would the average response time, throughput and resource utilization be under the expected workload?
3. How would performance change if load is increased? Does the system scale?
4. Which components have the largest effect on the overall system performance and are they potential bottlenecks?
5. What hardware and software resources are needed to guarantee that SLAs are met?

The above are typical capacity planning questions [10], whose answers are all, but straightforward to find. Instead of seeking concrete answers, 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 clearly defined methodology. Clearly, the issues of sizing and capacity planning need to be addressed in a more formal and systematic way if performance is to be guaranteed.

Different approaches have been proposed in the literature for performance analysis and prediction. Most of them exploit analytical models whose analysis is based on Markov Theory [6]. Queueing Networks (QNs) [14] and Generalized Stochastic Petri Nets (GSPNs) [5] are among the most popular modelling formalisms that have been used in the past decade. However, as argued in [1], both have some serious short-comings. While Queueing Networks provide a very powerful mechanism for modelling resource contention and scheduling strategies, they are not as

[1]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.

suitable for representing blocking and synchronization of processes. Generalized Stochastic Petri Nets, on the other hand, lend themselves very well to modelling blocking and synchronization aspects, but have difficulty in representing scheduling strategies. Bause [1] has proposed a new modelling formalism called *Queueing Petri Nets (QPNs)* that combines Queueing Networks and Petri Nets into a single formalism and eliminates the above disadvantages. QPNs allow queues to be integrated into places of Petri Nets. This enables the modeler to easily represent scheduling strategies and brings the benefits of Queueing Networks into the world of Petri Nets. In [7] it is shown that QPNs have greater expressive power than QNs, Extended QNs and SPNs. [4] further shows how this could be exploited in order to integrate software and hardware performance models using Hierarchical QPNs. QPNs take model expressiveness to a completely new level and allow much more complex aspects of system behavior to be modelled. However, this modelling power has hardly been exploited in the last years and very few, if any, practical applications have been reported. The latter is largely due to the fact that only a few methods have been developed for the analysis of QPNs and they all have some inherent limitations. As of the moment, most methods available are based on Markov Chains which suffer the well known state space explosion problem. This imposes a limit on the size of the models that can be analyzed and is a major hurdle to the practical application of QPNs.

Even though the state space explosion problem is a great barrier, we believe that there is yet a significant potential in QPNs that hasn't been exploited. In our opinion, QPNs lend themselves very well to modelling modern e-business systems based on distributed component architectures. In this paper we demonstrate this by looking at a real-world e-business application and showing how QPN models could be exploited for performance analysis and prediction in capacity planning studies. We focus on the benefits, in terms of modelling power and expressiveness, that QPNs provide over conventional modelling paradigms such as Queueing Networks and Petri Nets. The application we have chosen is part of the newly released SPECjAppServer2001 benchmark [12], which represents a heavy-duty business-to-business e-commerce workload. Our goal is to demonstrate the viability of QPNs as a modelling paradigm and motivate further research on techniques for their analysis.

2 Conventional Modelling Paradigms

Before we look at the QPN modelling approach we include a brief introduction to the conventional Queueing Network (QN) and Petri Net (PN) modelling paradigms from which the QPN formalism originates.

2.1 Queueing Networks

A *queueing network* consists of a set of interconnected queues. Each *queue* represents a service station, which serves *requests* (also called *jobs*) sent by customers. A *service station* consists of one or more servers and a waiting area which holds requests waiting to be served. When a request arrives at a service station, its service begins immediately if a free server is available. Otherwise, the request is forced to wait in the waiting area or the service of another request is preempted in case the arriving request has a higher priority. The time between successive request arrivals is called *interarrival time*. Each request demands a certain amount of service, which is specified by the length of time a server is occupied serving it, i.e the *service time*. The *queueing delay* is the amount of time the request waits in the waiting area before its service begins. The *response time* is the total amount of time the request spends at the service station, i.e. the sum of the queueing delay and the service time.

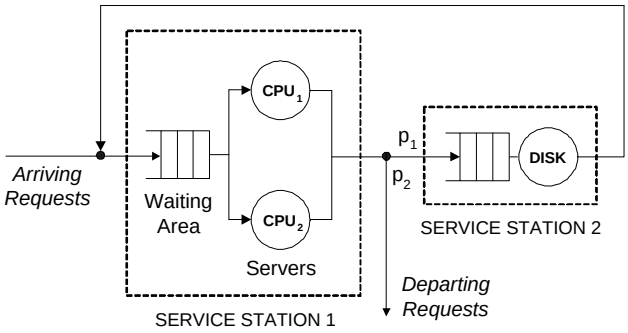

Figure 1: A Basic Queueing Network

Figure 1 shows a basic queueing network with two queues, i.e. service stations. Arriving requests first visit service station 1, which has two servers (representing CPUs). After requests are served by one of the servers, they move to service station 2 (representing a disk device) with probability p_1 or leave the network with probability p_2. Requests completing service at station 2 return back to station 1. The interconnection of queues in a queueing network is described by the paths requests may take which are specified by routing probabilities. A request might visit a service station multiple times while it circulates through the network. The total amount of service time required, over all visits to the station, is called *service demand* of the request at the station. Requests are usually grouped into *classes* with all requests in the same class having the same service demands.

The algorithm which determines the order in which requests are served at a service station is called *scheduling strategy* (or scheduling/queueing discipline) [6]. Some typical scheduling strategies are:

- **FCFS (First-Come-First-Served)**: Requests

are served in the order in which they arrive. This strategy is typically used for queues representing I/O devices.

- **LCFS (Last-Come-First-Served)**: The request that arrived last is served next.
- **PS (Processor-Sharing)** All requests are assumed to be served simultaneously with the server speed being equally divided among them. This strategy is typically used for modelling CPUs.
- **IS (Infinite-Server)**: There is an ample number of servers so that no queue ever forms. Service stations with IS scheduling strategy are often called *delay resources* or *delay servers*.

A queueing network in which requests arrive from an external source, get served in the network and then depart is said to be *open*, for e.g. the queueing network in Figure 1. A queueing network in which there is no external source of requests and no departures is said to be *closed*. In a closed queueing network it is assumed that the number of requests of each class circulating in the network is constant. It is also possible that a queueing network is open for some request classes and closed for others in which case the queueing network is called *mixed*.

Typical measures of interest for a queueing network include queue lengths, response times, *throughput* (the number of requests served per unit of time) and *utilization* (the fraction of time that a service station is busy). A relationship, known as *Little's Law* [10], relates the throughput $\mathcal{X}$ of a service station with the average number of requests $\mathcal{N}$ in it and their average response time $\mathcal{R}$. The relation is shown in Equation 1 and holds under the condition that the service station is in *steady state*, i.e. the number of requests arriving per unit of time is equal to the number of those completing service.

$$\mathcal{N} = \mathcal{X}.\mathcal{R} \tag{1}$$

Another relationship that we will use later, relates the service time $\mathcal{S}$ of requests at a station with its utilization $\mathcal{U}$ and throughput $\mathcal{X}$. This relationship is known as *Utilization Law* [10] and is shown in Equation 2.

$$\mathcal{U} = \mathcal{X}.\mathcal{S} \tag{2}$$

Queueing Networks provide a very powerful mechanism for modelling *hardware contention* (contention for CPU time, disk access and other hardware resources). A number of efficient analysis methods have been developed for certain classes of queueing networks, which enable models of realistic size and complexity to be analyzed. However, queueing networks are not as suitable for modelling *software contention* (contention for processes, threads, database connections, locks, latches and other software resources), as well as blocking, synchronization and simultaneous resource possession. Even though extensions of queueing networks, such as *Extended Queueing Networks* [6], provide some limited support for modelling software contention and synchronization aspects they are rather restrictive and inaccurate. For further details on queueing networks we refer the reader to [6].

2.2 Petri Nets

Petri Nets were introduced in 1962 by Carl Adam Petri. An ordinary *Petri Net* (also called *Place-Transition Net*) is a bipartite directed graph composed of places, drawn as circles, and transitions, drawn as bars. A formal definition [5] is given below:

Definition 1 (PN) *An ordinary Petri Net (PN) is a 5-tuple* $PN = (P, T, I^-, I^+, M_0)$*, where:*

1. P *is a finite and non-empty set of* ***places****,*
2. T *is a finite and non-empty set of* ***transitions****,*
3. $P \cap T = \emptyset$*,*
4. $I^-, I^+ : P \times T \rightarrow \mathbb{N}_0$ *are called backward and forward* ***incidence functions****, respectively,*
5. $M_0 : P \rightarrow \mathbb{N}_0$ *is called* ***initial marking****.*

The incidence functions I^- and I^+ specify the interconnection between places and transitions. If $I^-(p,t) > 0$, an arc leads from place p to transition t and place p is called an *input place* of the transition. If $I^+(p,t) > 0$, an arc leads from transition t to place p and place p is called an *output place* of the transition. The incidence functions assign natural numbers to arcs, which we call *weights* of the arcs. When each input place of transition t contains at least as many tokens as the weight of the arc connecting it to t, the transition is said to be *enabled*. An enabled transition may *fire*, in which case it destroys tokens from its input places and creates tokens in its output places. The amounts of tokens destroyed and created are specified by the arc weights. The initial arrangement of tokens in the net (called *marking*) is given by the function M_0 which specifies how many tokens are contained in each place. When a transition fires, the marking may change. Figure 2 illustrates this using a basic petri net with 4 places and 2 transitions. The petri net is shown before and after firing of transition t_1, which destroys one token from place p_1 and creates one token in place p_2.

Different extensions to ordinary Petri Nets have been developed in order to increase the modelling convenience and/or the modelling power. *Colored Petri Nets (CPNs)* introduced by K. Jensen [8] are one such extension. The latter allow a *type (color)* to be attached to a token. A color function C assigns a set of colors to each place. This set specifies the types of tokens that can reside in the place.

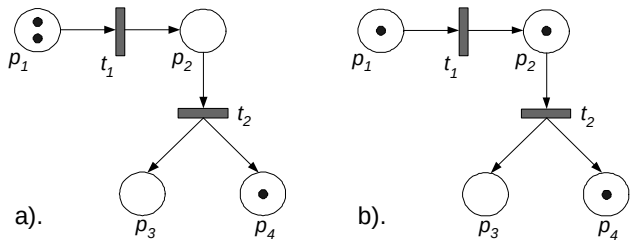

Figure 2: An Ordinary Petri Net before and after firing transition t_1

In addition to introducing token colors, CPNs also allow transitions to fire in different *modes (transition colors)*. The color function C assigns a set of colors to each transition and incidence functions are defined on a per color basis. A formal definition of CPNs [5] follows:

Definition 2 (CPN) *A Colored Petri Net (CPN) is a 6-tuple* $CPN = (P, T, C, I^-, I^+, M_0)$*, where:*

1. P *is a finite and non-empty set of* ***places****,*
2. T *is a finite and non-empty set of* ***transitions****,*
3. $P \cap T = \emptyset$*,*
4. C *is a* ***color function*** *defined from* $P \cup T$ *into non-empty sets,*
5. I^- *and* I^+ *are the backward and forward* ***incidence functions*** *defined on* $P \times T$*, such that* $I^-(p,t), I^+(p,t) : C(t) \rightarrow C(p)_{MS}$*,* $\forall (p,t) \in P \times T$ [2]
6. M_0 *is a function defined on* P *describing the* ***initial marking*** *such that* $M_0(p) \in C(p)_{MS}, \forall p \in P$

Other extensions to ordinary Petri Nets allow temporal (timing) aspects to be integrated into the net description [5]. In particular, *Stochastic Petri Nets (SPNs)* attach an exponentially distributed *firing delay* to each transition, which specifies the time the transition waits after being enabled before it fires. *Generalized Stochastic Petri Nets (GSPNs)* allow two types of transitions to be used: immediate and timed. Once enabled, immediate transitions fire in zero time. If several immediate transition are enabled at the same time, the next transition to fire is chosen based on *firing weights* (probabilities) assigned to the transitions. Timed transitions fire after a random exponentially distributed firing delay as in the case of SPNs. The firing of immediate transitions always has priority over that of timed transitions. A formal definition of GSPNs [5] follows:

Definition 3 (GSPN) *A GSPN is a 4-tuple* $GSPN = (PN, T_1, T_2, W)$*, where:*

1. $PN = (P, T, I^-, I^+, M_0)$ *is the underlying ordinary Petri Net,*
2. $T_1 \subseteq T$ *is the set of timed transitions,* $T_1 \neq \emptyset$*,*
3. $T_2 \subset T$ *is the set of immediate transitions,* $T_1 \cap T_2 = \emptyset$*,* $T = T_1 \cup T_2$*,*
4. $W = (w_1, ..., w_{|T|})$ *is an array whose entry* $w_i \in \mathbb{R}^+$
   - *is a (possibly marking dependent) rate of a negative exponential distribution specifying the firing delay, when transition* t_i *is a timed transition, i.e.* $t_i \in T_1$ *or*
   - *is a (possibly marking dependent) firing weight, when transition* t_i *is an immediate transition, i.e.* $t_i \in T_2$*.*

By combining definitions 2 and 3, *Colored Generalized Stochastic Petri Nets (CGSPNs)* can be defined [5].

Petri Nets are a very powerful tool for both qualitative and quantitative system analysis. Unlike queueing networks, they easily lend themselves to modelling blocking and synchronization aspects. However, Petri Nets have the disadvantage that they do not provide any means for direct representation of scheduling strategies [7]. The attempts to eliminate this disadvantage have led to the emergence of Queueing Petri Nets (QPNs).

3 The QPN Modelling Formalism

In the following we give a brief introduction to the QPN formalism based on [1].

3.1 Basic Queueing Petri Nets

The main idea in the creation of the QPN formalism was to add queueing and timing aspects to the places of Colored Generalized Stochastic Petri Nets [5]. This is done by allowing queues (service stations) to be integrated into places of CGSPNs. A place of a CGSPN which has an integrated queue is called a *queueing place* and consists of two components, the *queue* and a *depository* for tokens which have completed their service at the queue. This is depicted in Figure 3.

The behavior of the net is as follows: tokens, when fired into a queueing place by any of its input transitions, are inserted into the queue according to the queue's scheduling strategy. Tokens in the queue are not available for output transitions of the place. After completion of its service, a token is immediately moved to the depository, where it becomes available for output transitions of the place. This type of queueing place is called a *timed queueing place*. In addition to timed queueing places, QPNs also introduce *immediate queueing places*, which allow pure scheduling aspects to be described. Tokens in immediate queueing places can be viewed as being served immediately. Scheduling in such places has priority over scheduling/service in timed queueing places and firing of timed transitions.

[2] The subscript MS denotes multisets. $C(p)_{MS}$ denotes the set of all finite multisets of $C(p)$.

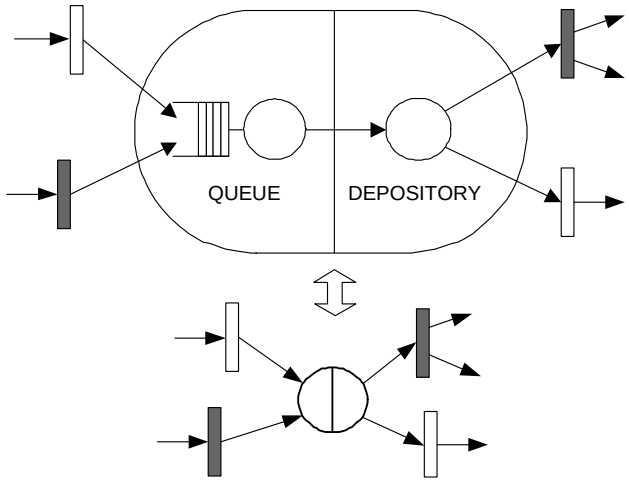

Figure 3: A Queueing Place and its Shorthand Notation

The rest of the net behaves like a normal CGSPN. An enabled timed transition fires after an exponentially distributed delay according to a race policy. Enabled immediate transitions fire according to relative firing frequencies and their firing has priority over that of timed transitions. We now give a formal definition of a QPN and then present an example of a QPN model.

Definition 4 (QPN) *A Queueing Petri Net (QPN) is a 8-tuple* $QPN = (P, T, C, I^-, I^+, M_0, Q, W)$, *where:*

1. $CPN = (P, T, C, I^-, I^+, M_0)$ *is the underlying Colored Petri Net*
2. $Q = (q_1, ..., q_{|P|})$ *is an array whose entry* q_i
   - *denotes the description of a queue taking all colors of* $C(P)$ *into consideration, if* p_i *is a timed place or*
   - *equals the keyword untimed, if* p_i *is an untimed place.*
3. $W = (w_1, ..., w_{|T|})$ *is an array of functions whose entry* w_i *is defined on* $C(t_i)$ *and* $\forall c \in C(t_i) : w_i(c)$ *is*
   - *the description of a probability distribution function specifying the firing delay due to color* $c \in C(t_i)$, *if transition* t_i *is a timed transition or*
   - *is a weight specifying the relative firing frequency due to color* $c \in C(t_i)$, *if transition* t_i *is an immediate one.*

Example 1 (QPN) *Figure 4 shows an example of a QPN model of a central server system with memory constraints based on [5]. Place* p_2 *represents several terminals, where users start jobs (modelled with tokens of color "o") after a certain thinking time. These jobs request service at the CPU (represented by the -/C/1-PS queue, where C stands for Coxian distribution) and two disk subsystems (represented by the -/C/1-FCFS queues). To enter the system each job has to allocate a certain amount of memory. For simplicity, the amount of memory needed by each job is assumed to be the same, which is represented by a token of color "m" on place* p_1. *According to definition 4 we have the following:*

$QPN = (P, T, C, I^-, I^+, M_0, Q, W)$ *where*

- $CPN = (P, T, C, I^-, I^+, M_0)$ *is the underlying Colored Petri Net as depicted in Figure 4*
- $Q = (untimed, -/M/\infty - IS, -/M/1 - PS, untimed, -/M/1 - FCFS, -/M/1 - FCFS)$
- $W = (w_1, ..., w_{|T|})$, *where all transitions are immediate and* $\forall c \in C(t_i) : w_i(c) := 1$, *since we want their firings to be equally likely.*

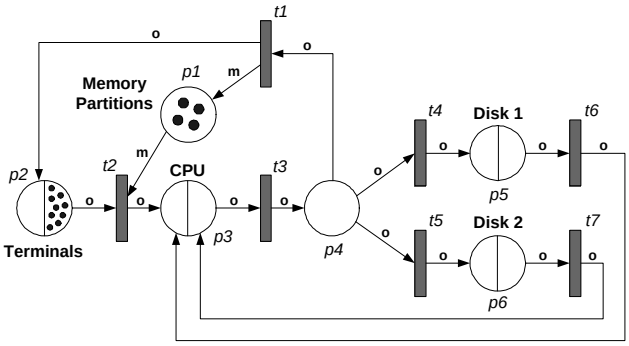

Figure 4: A QPN Model of a Central Server System with Memory Constraints (based on [5])

3.2 Hierarchical Queueing Petri Nets

As already mentioned, the main hurdle to the quantitative analysis of QPNs is the fact that most analysis techniques available are based on Markov Chains and are therefore susceptible to the state space explosion problem. More specifically, as one increases the number of queues and tokens in a QPN, the size of the state space of the underlying Markov Chain grows exponentially and quickly exceeds the capacity of today's computers. This imposes a limit on the size and complexity of the models that are analyzable. An attempt to alleviate this problem was the introduction of *Hierarchically-Combined Queueing Petri Nets (HQPNs)* [2]. The main idea is to allow hierarchical model specification and then exploit the hierarchical structure for efficient numerical analysis. This type of analysis is termed *structured analysis* and it allows models to be solved, which are about an order of magnitude larger than those analyzable with conventional techniques.

HQPNs are a natural generalization of the original QPN formalism. In HQPNs a queueing place may contain a whole QPN instead of a single queue. Such a place is called a *subnet place* and is depicted in Figure 5. A subnet place might contain an ordinary QPN or again a HQPN allowing multiple levels of nesting. For simplicity, in this paper we restrict ourselves to two-level hierarchies. We use the term *High-Level QPN (HLQPN)* to refer to the upper level of the HQPN and the term *Low-Level QPN (LLQPN)* to refer to a subnet of the HLQPN.

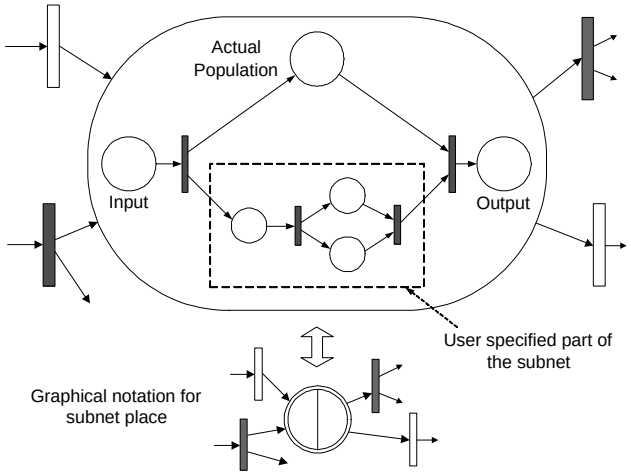

Figure 5: A subnet place and its shorthand notation

Every subnet of a HQPN has a dedicated input and output place, which are ordinary places of a colored Petri Net. Tokens being inserted into a subnet place after a transition firing are added to the input place of the corresponding HQPN subnet. The semantics of the output place of a subnet place is similar to the semantics of the depository of a queueing place: tokens in the output place are available for the output transitions of the subnet place. Tokens contained in all other places of the HQPN subnet are not available for the output transitions of the subnet place. Every HQPN subnet also contains $actual - population$ place, which is used to keep track of the total number of tokens fired into the subnet place.

3.3 QPN Analysis Tools

To the best of our knowledge [11], there is currently only one tool available that supports modelling and analysis using QPNs. This is the HQPN-Tool presented in [3]. The latter supports a number of structured analysis methods for HQPNs, such as Structured Power, Structured SOR, Structured JOR among others. These methods provide exact solution of the model's underlying Markov Chain, and therefore the results obtained are independent of the solution method used.

After this brief introduction to the QPN formalism, we will now proceed to give a short overview of the SPECjAppServer2001 benchmark, whose order entry application will be the subject of our study in the rest of the paper.

4 The SPECjAppServer2001 Benchmark

SPECjAppServer2001 is a newly released [3] benchmark for measuring the performance and scalability of J2EE application servers. SPECjAppServer2001 is the successor of the popular ECperf benchmark prototyped and built by Sun in conjunction with application server vendors under the Java Community Process (JCP).

4.1 SPECjAppServer2001 Business Model

The SPECjAppServer2001 workload is based on a large distributed application claimed to be big and complex enough to represent a real-world e-business system [12]. The SPECjAppServer2001 designers have chosen manufacturing, supply chain management, and order/inventory as the "storyline" of the business problem to be modelled. This is an industrial-strength distributed problem, that is heavyweight, mission-critical and requires the use of a powerful and scalable infrastructure.

SPECjAppServer2001 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 6 illustrates these domains and gives some examples of typical transactions run in each of them. In this paper we will concentrate on the customer domain, which hosts an order entry application running four transaction types: NewOrder, ChangeOrder, OrderStatus and CustomerStatus.

4.2 SPECjAppServer2001 Application Design

All the activities and processes in the four domains described above are implemented using Enterprise Java Bean (EJB) components [13] assembled into a single J2EE application, which is deployed on the *System Under Test (SUT)*. The only exception is for the interactions with suppliers, which are implemented using a separate J2EE application that runs in a Web Container on a dedicated machine. A relational DBMS is used for data persistence [9].

The benchmark can be run in two modes. In the first mode only the order entry application in the customer domain is run, while in the second mode both the order entry and the manufacturing applications are

[3] Although released in 2002, the benchmark was named SPECjAppServer2001 because it uses the workload of the ECperf 1.1 benchmark, which was released in 2001.

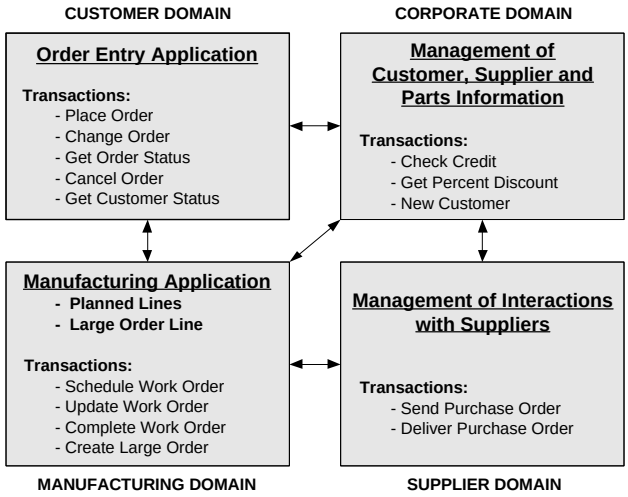

Figure 6: The SPECjAppServer2001 Business Model

run. For further details on SPECjAppServer2001, we refer the reader to [12] and [9] .

5 Case Study: Order Entry Application

In this section we proceed to build a QPN-based performance model of the SPECjAppServer2001's order entry application, validate it against measured data and then show how this model can be exploited for the purposes of performance prediction and sizing in the capacity planning process.

5.1 Motivation

Recall that the order entry application was 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. *CustomerStatus*: lists all orders placed by a given customer

Now imagine the following hypothetical scenario: A company is about to introduce an online ordering service for its customers and chooses to implement the service using a J2EE application. Assume that this application is the order entry application of SPECjAppServer2001. Before putting the application into production the company decides to conduct a capacity planning study in order to come up with an adequate sizing and configuration of the deployment environment. We assume that the company initially plans to deploy the application in the deployment environment depicted in Figure 7. This environment uses a cluster of WebLogic servers (WLS) as a container for the J2EE application and Oracle 9i as a database server (DBS) for persistence. We assume that all machines in the WLS cluster are identical.

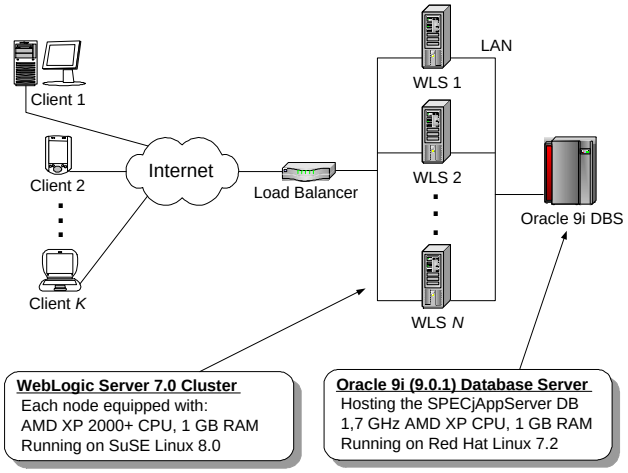

Figure 7: Deployment Environment

The company is interested in finding answers to the following questions:

1. What level of performance does the system provide under load?
2. Are there potential system bottlenecks? Does the system scale?
3. How many application servers would be needed to guarantee adequate performance?

In addition, the company needs to find optimal values for the following configuration parameters:

1. Number of threads in WLS thread pools
2. Number of JDBC connections in WLS database connection pools
3. Number of shared server processes of the Oracle server instance

We now proceed to show how these issues can be approached with the help of QPN-based 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* [10] and in its simplest form includes three major steps:

1. Describe the types of requests that arrive at 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 (*service demand*) for each request class at each resource.

In our scenario we have four request classes corresponding to the four order entry transaction types. The following resources are used during their processing:

1. The CPU of a WebLogic server
2. The network between a WebLogic server and the database server
3. The CPU of the database server
4. The disk subsystem of the database server (I/O)

In addition to this, each transaction uses a WLS thread, a database connection and a database server process during its processing.

In order to determine the service demands we conducted some experiments with the order entry application and measured the time spent by each transaction at each resource. For the database server, we measured the service demands using the Oracle 9i Intelligent Agent. For the application server, we instrumented WebLogic to measure the CPU time spent in the classes that implement the four order entry transactions and based on this we were able to determine the service demands of the four request types. As to network service demands, we decided to ignore them since all communications were taking place over a 100MBit LAN and communication times were negligible. Table 1 reports our service demand measurements for the four request classes in our system.

Table 1: Workload Service Demands

TX-Type	WLS-CPU	DBS-CPU	DBS-I/O
NewOrder	70ms	53ms	12ms
ChangeOrder	26ms	16ms	6ms
OrderStatus	7ms	4ms	0ms
CustomerStatus	10ms	5ms	0ms

5.3 First Cut System Model

We start by proposing a simple QPN system model which does not utilize any hierarchical structures and is depicted in Figure 8. For now we assume that there is a single application server in the WLS cluster.

The following types of tokens (token colors) are used in the model:

Token "x" represents a request sent by a client for execution of a particular transaction. In the case of multiple request classes, a separate token color (e.g. "x", "y", "z",...) should be used for each request class.

Token "t" represents a WLS thread.

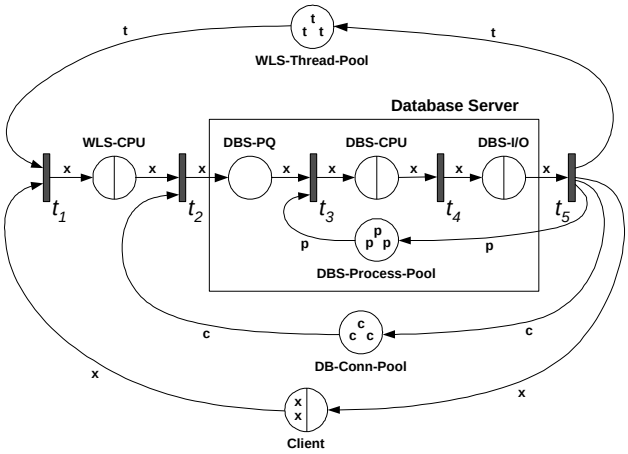

Figure 8: Flat QPN System Model

Token "c" represents a JDBC connection to the DBS.

Token "p" represents a DBS process.

In the following we describe the places of the model:

Client Queueing place with IS scheduling strategy used to represent clients sending requests to the system. The service time of this place corresponds to the average client think time.

WLS-CPU Queueing place with PS scheduling strategy used to represent the CPU of WLS.

DBS-CPU Queueing place with PS scheduling strategy used to represent the CPU of the DBS.

DBS-I/O Queueing place with FCFS scheduling strategy used to represent the disk subsystem of the DBS.

WLS-Thread-Pool Ordinary place used to represent the thread pool of WLS. Each token in this place represents a WLS thread.

DB-Conn-Pool Ordinary place used to represent the JDBC connection pool of WLS. Tokens in this place represent JDBC connections to the DBS.

DBS-Process-Pool Ordinary place used to represent the process pool of the DBS. Tokens in this place represent Oracle processes.

DBS-PQ Ordinary place used to hold incoming requests at the DBS while they wait for a server process to be allocated to them.

We now take a look at the life-cycle of a client request in our system model. Every request (modelled by token of color "x") is initially at the queue of place Client, where it waits for a user-specified think time. After the think time elapses, the request moves

to the Client depository, where it waits for a WLS thread to be allocated to it, before its processing can start. Once a thread is allocated (modelled by taking a token of color "t" from place WLS-Thread-Pool), the request moves to the queue of place WLS-CPU, where it receives service from the CPU of WLS. It then moves to the depository of the place and waits for a JDBC connection to be allocated to it. The JDBC connection (modelled by token "c") is used to contact the database and make any updates required by the respective transaction. A request sent to the database server arrives at the place DBS-PQ (DBS Process Queue) where it waits for a server process (modelled by token "p") to be allocated to it. Once this is done, the request receives service first at the CPU and then at the disk subsystem of the database server. This completes the processing of the request, which is then sent back to place Client releasing the held DBS process, JDBC connection and WLS thread.

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

- Number of requests of each request class in the initial marking.
- Service demands of request classes at the queues of places WLS-CPU, DBS-CPU and DBS-I/O (see Table 1).
- Average client think time (service time at the queue of place Client).
- Number of WLS threads (tokens "t"), JDBC connections (tokens "c") and Oracle server processes (tokens "p") in the initial marking.

5.4 Hierarchical System Model

The model described above is a flat QPN model and its corresponding state space grows exponentially with the number of requests in the system. This means that only relatively small instances of the model (with relatively small number of requests) are analyzable. We will now show how hierarchical structuring can be exploited in order to alleviate the state space explosion problem and enable larger models to be analyzed.

The idea is to isolate the database server and model it using a separate nested QPN. In order to do this we replace the database server part of the original flat QPN with a single subnet place which we call "DBS". This place represents the entire database server and is expanded into a low-level QPN containing the original DBS queues of our flat model. Figures 9 and 10 show the high-level and low-level QPN of our new system model.

By specifying our model in a hierarchical fashion we can now exploit structured analysis techniques [2], which enables us to solve models with much higher number of requests.

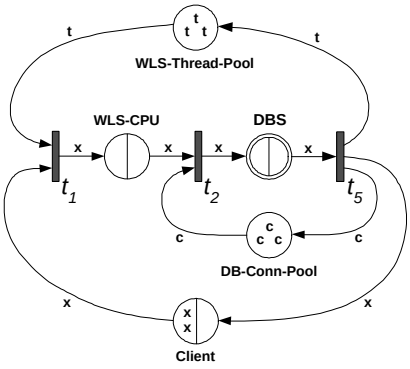

Figure 9: Model's High-Level QPN

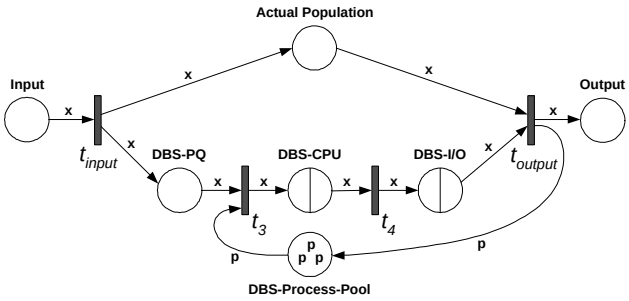

Figure 10: Model's Low-Level QPN

5.5 Model Analysis and Validation

We will now proceed to analyze several different instances of our models introduced in the previous section and then validate them by comparing results from the analysis with measured data. We start by taking a look at the mathematical laws and formulas that we will be using in our analysis. Our presentation is based on the following notation:

$\mathcal{D}$: *Service demand* of a queue, i.e. the amount of time required for a token (request) to be served at the queue.

μ : Average *service rate* of a queue, i.e. the number of tokens (requests) served at the queue per unit of time.

$\mathcal{N}$: Average *token population* of a queue, place or depository, i.e. the average number of tokens (requests) in it.

$\mathcal{U}$: Average *utilization* of a queue, place or depository, i.e. the probability that there is a token (request) in it.

$\mathcal{X}$: Average *throughput* of a queue, place or depository, i.e. the number of tokens (requests) that leave it per unit of time. Note that, at steady state, the rate at which tokens leave a queue/place/depository is equal to the rate at which they enter it.

$\mathcal{R}$: Average *residence time* of a token (request) at a queue, place or depository.

All models that we will be considering in this section will be based on the hierarchical system model presented in the previous section. We will be using the *HQPN-Tool* [3] to solve the models. Based on the solution of the model's underlying Markov Chain the HQPN-Tool reports the distribution of the number of tokens at each place in steady state. For queueing places the latter is reported separately for the queue and for the depository of the place. Using the distributions, one can easily derive the average token population $\mathcal{N}$ (which is done automatically and reported by the tool) and the utilization $\mathcal{U}$ of each queue in steady state. The following trivial relations hold:

$$\mathcal{N} = \sum_{i=0}^{\infty} i.p_i \tag{3}$$

$$\mathcal{U} = 1 - p_0 \tag{4}$$

where p_i is the probability that there are i tokens in the queue.

Note that, since places Client, WLS-CPU, DBS-PQ, DBS-CPU and DBS-I/O form a closed chain with respect to the flow of requests in the system, using the "flow-in = flow-out" principle from Queueing Theory [14], we can conclude that the throughputs of requests through these places in steady state must be equal. The same applies also to the queues and depositories of these places, i.e. they have the same throughput as the places themselves. Furthermore, this holds both for total request throughputs, as well as for throughputs of particular request classes.

The following trivial relationship holds [10]:

$$\mu = \frac{1}{\mathcal{D}} \tag{5}$$

Using this formula we can derive the service rates of the queues in our model for the different request classes based on their service demands provided in Table 1. Given that the service rates of all queues are load-independent, we can then use the following relation (which follows from the Utilization Law) in order to derive the throughput $\mathcal{X}$ of a queue in the case of a single request class:

$$\mathcal{X} = \mathcal{U}.\mu \tag{6}$$

In the case of multiple request classes we can derive the throughput of each request class using Little's Law. Recall that the latter relates the throughput $\mathcal{X}$ of a queue with the average number of requests $\mathcal{N}$ in it and their average residence time $\mathcal{R}$. The relation is shown in Equation 7 below and can be applied both with respect to all requests of the system, as well as, with respect to separate request classes.

$$\mathcal{X} = \frac{\mathcal{N}}{\mathcal{R}} \tag{7}$$

We apply this formula to the Client queue for each request class in the system. The average number $\mathcal{N}$ of requests of each class is reported by the tool. The residence time of requests $\mathcal{R}$ is also known since the queue has "Infinite Server" scheduling strategy and therefore the residence time of all requests is equal to the service time of the queue. However, the service time of the queue is per definition equal to the client think time, which is an input parameter to the model. Substituting $\mathcal{N}$ and $\mathcal{R}$ in Equation 7 we can calculate the throughput of each request class.

Now that we know how to find the throughput of requests in the system, we can derive the residence time of requests at every place, queue or depository using the following equation, which again follows directly from Little's Law:

$$\mathcal{R} = \frac{\mathcal{N}}{\mathcal{X}} \tag{8}$$

We are now ready to start analyzing some concrete instances of our models.

5.5.1 Scenario 1: Single Request Class

We start with a simplified scenario in which we have a single application server and a single request class, the NewOrder transaction. Our goal is to analyze the behavior of the system with respect to this transaction. Assume that we have 80 clients in the system with average think time of 200ms and there are 60 WLS threads, 40 JDBC connections and 30 DBS processes available. We use this data to parameterize our hierarchical system model from section 5.4.

Table 2 summarizes our analysis results for this scenario. It reports the calculated throughputs and residence times of requests at the most important queues and depositories. It also reports the average token population of places WLS-Thread-Pool, DB-Conn-Pool and DBS-Process-Pool. We have used subscripts "Q" and "D" to distinguish between queues and depositories of places.

The total end-to-end request response time ($\mathcal{R}_{Total}$) is equal to the time needed for a request to make a complete cycle through the queueing places of the system. The latter can be calculated by summing up the residence times of requests at the queues and depositories of all queueing places plus the residence time at the ordinary place DBS-PQ.

$$\begin{aligned}\mathcal{R}_{Total} = \; & \mathcal{R}_{Client_D} + \mathcal{R}_{WLS-CPU_Q} + \\ & + \mathcal{R}_{WLS-CPU_D} + \mathcal{R}_{DBS-PQ} + \\ & + \mathcal{R}_{DBS-CPU_Q} + \mathcal{R}_{DBS-I/O_Q}\end{aligned} \tag{9}$$

Table 2: Analysis Results for Scenario 1

PLACE	$\mathcal{N}$	$\mathcal{U}$	$\mathcal{X}$	$\mathcal{R}$ $[ms]$
Client_Q	2.85	0.94	14.28	200
Client_D	17.14	1.00	-//-	1200
WLS-CPU_Q	56.67	1.00	-//-	3967
WLS-CPU_D	0.00	0.00	-//-	0
DBS-PQ	0.00	0.00	-//-	0
DBS-CPU_Q	3.11	0.75	-//-	218
DBS-I/O_Q	0.20	0.17	-//-	14
WLS-Thread-Pool	0.00	0.00		
DB-Conn-Pool	36.67	1.00		
DBS-Process-Pool	26.67	1.00		

Note that the above sum excludes the Client queue, since the time spent at it corresponds to the client think time. It also excludes the depositories of places DBS-CPU and DBS-I/O, because requests never wait at them.

Table 3 compares results obtained from the model analysis with results obtained from measurements and shows the modelling error for the most important performance metrics. The measurements were collected by running an experiment in which the specified workload was injected into the system over a period of 40 minutes. Measurements were taken after the first 10 minutes, which the system needed to reach steady state.

Table 3: Modelling Error for Scenario 1

METRIC	Model	Measured	Error
WLS-CPU Utilization	100%	100%	0%
DBS-CPU Utilization	75%	65%	15%
NewOrder Throughput	14.28	13.43	6.3%
NewOrder Resp.Time	5399ms	5738ms	5.9%
Thread Queue Length	17.14	18	4.7%

As we can see from Table 2, requests spend 1200ms on average at the Client depository waiting for a WLS thread to be freed. On the other hand, there are plenty of DB connections and DBS processes available and there is no contention for these resources as indicated by the residence times of requests at WLS-CPU_D and DBS-PQ, both of which are zero. Since, on average, there are only 3.31 requests served concurrently at the database server we can decrease the number of available JDBC connections and DBS processes significantly without impacting performance in any negative way. In fact, doing this could even improve the overall performance since JDBC connections and DBS processes cost memory and reducing them will increase the amount of memory available for the application server and database server, respectively.

It would be interesting to see what would change if we decrease the number of available WLS threads to 40. This would limit the number of requests processed concurrently by the application server. Table 4 repeats the analysis for 40 WLS threads.

Table 4: Analysis Results for Sc. 1 with 40 Threads

PLACE	$\mathcal{N}$	$\mathcal{U}$	$\mathcal{X}$	$\mathcal{R}$ $[ms]$
Client_Q	2.85	0.94	14.28	200
Client_D	37.14	1.00	-//-	2601
WLS-CPU_Q	36.67	1.00	-//-	2568
WLS-CPU_D	0.00	0.00	-//-	0
DBS-PQ	0.00	0.00	-//-	0
DBS-CPU_Q	3.11	0.75	-//-	218
DBS-I/O_Q	0.20	0.17	-//-	14
WLS-Thread-Pool	0.00	0.00		
DB-Conn-Pool	36.67	1.00		
DBS-Process-Pool	26.67	1.00		

As we can see, the change does not have any impact on the overall system throughput, since the application server is in both cases saturated to its full capacity. However, one might at first be misled to believe that because of increased contention for threads, the end-to-end request response time would also increase, which according to our model as well as our measurements is not the case. This is because reducing the level of concurrency in the application server, results in requests being served faster and this compensates the longer time spent queueing for threads. In both cases the application server is completely utilized and the rest of the system remains unaffected by the change. Table 5 shows the modelling error with 40 threads. As we can see in both cases the QPN models are quite accurate in representing the system.

As demonstrated, QPNs enable us to integrate in the same model both hardware and software aspects of system behavior. Using queueing places we can easily model hardware contention and scheduling strategies, with the same flexibility as in traditional Queueing Networks. However, in addition to this, QPNs also empower us to represent some further aspects of system behavior such as simultaneous resource possession, blocking and contention for software resources (threads, connections and processes). The latter is not possible, at least at this level of accuracy, using conventional modelling paradigms such as Queueing Networks and Petri Nets. Even though extensions of Queueing Networks, such as the so-called *Extended Queueing Networks*, provide some limited support for modelling software contention, they are way too restrictive and inaccurate to compare with the modelling power demonstrated in the above examples.

Table 5: Mod. Error for Scenario 1 with 40 Threads

METRIC	Model	Measured	Error
WLS-CPU Utilization	100%	100%	0%
DBS-CPU Utilization	75%	65%	15%
NewOrder Throughput	14.28	13.41	6.4%
NewOrder Resp.Time	5401ms	5742ms	5.9%
Thread Queue Length	37.14	40	7.1%

5.5.2 Scenario 2: Multiple Request Classes

We will now look at a scenario in which we have two classes of requests in the system - NewOrder and ChangeOrder. We again use our hierarchical model from section 5.4 as a basis. However, this time we define two types of request tokens (NewOrder and ChangeOrder) so that we can distinguish between the two request classes. Trying to solve the resulting HQPN model for high values of the number of NewOrder and ChangeOrder clients, we ran into state space explosion of the underlying Markov Chain. Therefore we make some simplifications in order to come up with a model that is analyzable. First of all, we assume that there are plenty of JDBC connections and DBS Server processes and drop places DB-Conn-Pool and DBS-Process-Pool. In addition, we assume that there are only 20 clients in the system (10 NewOrder and 10 ChangeOrder), the average think time is 1 sec and there are 10 WLS threads available. Tables 6 and 7 report the analysis results and modelling error, respectively, for this scenario. As we can see, the QPN models are also quite accurate in representing the system behavior with multiple request classes. The modelling error for most performance metrics remains under 10%.

5.5.3 Scenario 3: Multiple Appl. Servers

In this final scenario we will generalize our initial setting to allow multiple application servers to be used. We again use the hierarchical system model from section 5.4 as a basis, but modify the HLQPN to include multiple WLS queueing places - one per application server. The new HLQPN is depicted in Figure 11.

To simplify things we do not include WLS-Thread places for the application servers in the new model. In fact, if we were to have WLS-Thread places for the application servers, we would also need some way to distinguish between requests in the DBS subnet originating from different application servers. This is because we need to know at which application server to release a thread after completing service at the DBS subnet. One way to implement this is using the notion of *tags* [4] which are automatically added to tokens upon entry into the DBS subnet to keep track of their origin. However, this functionality is currently not supported by the HQPN-Tool and therefore we don't include any

Table 6: Analysis Results for Scenario 2

PLACE	$\mathcal{N}$	$\mathcal{U}$	$\mathcal{X}$	$\mathcal{R}$ $[ms]$
WLS-Thread-Pool	6.68	0.99		
Over All Request Classes				
Client_Q	16.67	1.00	16.67	1000
Client_D	0.00	0.00	-//-	0
WLS-CPU_Q	2.14	0.76	-//-	128
DBS-CPU_Q	1.00	0.54	-//-	59
DBS-I/O_Q	0.16	0.14	-//-	9
Over NewOrder Requests				
Client_Q	7.45	1.00	7.45	1000
Client_D	0.00	0.00	-//-	0
WLS-CPU_Q	1.64	0.70	-//-	220
DBS-CPU_Q	0.79	0.46	-//-	107
DBS-I/O_Q	0.10	0.06	-//-	14
Over ChangeOrder Requests				
Client_Q	9.22	1.00	9.22	1000
Client_D	0.00	0.00	-//-	0
WLS-CPU_Q	0.50	0.35	-//-	54
DBS-CPU_Q	0.21	0.19	-//-	23
DBS-I/O_Q	0.06	0.09	-//-	7

Table 7: Modelling Error for Scenario 2

METRIC	Model	Measured	Error
WLS-CPU Utilization	76%	77%	1.2%
DBS-CPU Utilization	54%	64%	15.6%
Avg.free WLS-Threads	6.68	7	4.5%
NewOrder Throughput	7.45	7.47	0.2%
NewOrder Resp. Time	341ms	318ms	7.2%
ChgOrder Throughput	9.22	9.15	0.7%
ChgOrder Resp. Time	84ms	104ms	19.2%

WLS-Thread places in our model. We assume that there are 30 NewOrder clients in the system and that there is no contention for WLS threads, JDBC connections or Oracle processes. The client think time is again 1 sec. Table 8 summarizes the analysis results for 2 and 3 application servers, respectively, and Table 9 reports the modelling error. As seen from the results, the modelling error remains under 10% and our QPN models perform quite well as a performance prediction tool.

6 Summary and Conclusions

In this paper we showed how QPN models can be exploited for performance analysis of distributed e-business systems. We studied a real-world application and demonstrated the modelling power and expressiveness of the QPN formalism, by showing how it enables us to integrate both hardware and software aspects of system behavior in the same model. In addition to hardware contention and scheduling strategies, QPNs empower the modeler to easily represent simul-

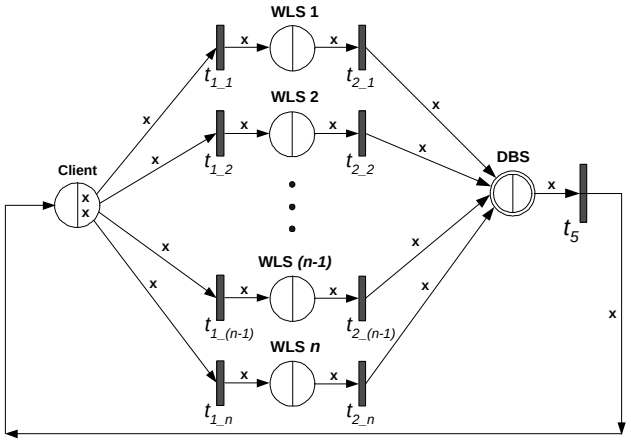

Figure 11: High-Level QPN Model with N Appl. Servers

Table 8: Analysis Results for Scenario 3

PLACE	$\mathcal{N}$	$\mathcal{U}$	$\mathcal{X}$	$\mathcal{R}$ $[ms]$
For 2 Application Servers				
$\texttt{Client}_Q$	18.28	1.00	18.28	1000
$\texttt{WLS1-CPU}_Q$	1.68	0.64	9.14	184
$\texttt{WLS2-CPU}_Q$	1.68	0.64	9.14	184
$\texttt{DBS-CPU}_Q$	8.07	0.96	18.28	441
$\texttt{DBS-I/O}_Q$	0.27	0.21	-//-	15
For 3 Application Servers				
$\texttt{Client}_Q$	18.42	1.00	18.42	1000
$\texttt{WLS1-CPU}_Q$	0.72	0.43	6.14	117
$\texttt{WLS2-CPU}_Q$	0.72	0.43	6.14	117
$\texttt{WLS3-CPU}_Q$	0.72	0.43	6.14	117
$\texttt{DBS-CPU}_Q$	9.05	0.98	18.42	491
$\texttt{DBS-I/O}_Q$	0.28	0.22	-//-	15

taneous resource possession, synchronization, blocking and contention for software resources. The latter is not doable to this extent using conventional modelling paradigms such as Queueing Networks and Petri Nets. By validating the models presented, through measurements, we showed that QPNs are not just powerful as a specification mechanism, but are also very powerful as a performance analysis and prediction tool. However, if this power is to be exploited to its full potential, improved solution methods and software tools for QPNs need to be developed which enable larger models to be analyzed.

Acknowledgments

We gratefully acknowledge the many fruitful discussions with Dr. Falko Bause from the University of Dortmund and his cooperation in providing us with the HQPN-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.

Table 9: Modelling Error for Scenario 3

METRIC	Model	Measured	Error
For 2 Application Servers			
WLS-CPU Utilization	64%	68%	6%
DBS-CPU Utilization	96%	91%	5%
NewOrder Throughput	18.28	17.56	4%
NewOrder Resp. Time	640ms	693ms	8%
For 3 Application Servers			
WLS-CPU Utilization	43%	44%	2%
DBS-CPU Utilization	98%	97%	1%
NewOrder Throughput	18.42	17.61	5%
NewOrder Resp. Time	623ms	673ms	7%

References

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

[2] F. Bause, P. Buchholz, and P. Kemper. Hierarchically Combined Queueing Petri Nets. In *Proc. of 11th Intl. Conference on Analysis and Optimization of Systems, Discrete Event Systems, Sophie-Antipolis (France)*, 1994.

[3] F. Bause, P. Buchholz, and P. Kemper. QPN-Tool for the Specification and Analysis of Hierarchically Combined Queueing Petri Nets. In *Quantitative Evaluation of Computing and Communication Systems, Lecture Notes in Computer Science, No. 977, Springer-Verlag*, 1995.

[4] 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), Freiberg (Germany)*, 1997.

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

[6] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains - Modelling and Performance Evaluation with Computer Science Applications.* Wiley, New York, 1998.

[7] F. Bause. "QN + PN = QPN" - Combining Queueing Networks and Petri Nets. Technical report no.461, Dept. of CS, University of Dortmund, Germany, 1993.

[8] K. Jensen. *Coloured Petri Nets and the Invariant Method.* Mathematical Foundations on Computer Science, Lecture Notes in Computer Science 118:327-338, 1981.

[9] 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.*

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

[11] University of Aarhus. Petri net tool database. Department of Computer Science - DIAMI. `http://www.daimi.au.dk/PetriNets/tools/`.

[12] Standard Performance Evaluation Corporation (SPEC). SPECjAppServer2001 Documentation. Technical report, September 2002. `http://www.spec.org/osg/jAppServer/`.

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

[14] K. S. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Science Applications.* Wiley, New York, 2nd edition, 2002.